Imperial College London

Imperial College London

Department of Mathematics

Distributional Reinforcement Learning for Optimal Execution

Author: Toby Weston (CID: 01796807)

A thesis submitted for the degree of

MSc in Mathematics and Finance, 2019-2020

Abstract

When trading a financial asset, large orders will often incur higher execution costs as the trader uses up the available liquidity. To reduce this effect, orders are split and executed over a short period of time. Theoretical solutions for how to optimally split orders rely on models of market environments. These fail to take into account market idiosyncrasies and tend to oversimplify a complex optimisation problem.

Deep Q learning provides a set of methodologies for learning an optimal solution from real experience. Successful application would allow models of the trading environment to be sidestepped in favour of direct interaction with the financial markets. Deep Q learning has previously been applied to the problem of optimal execution and has shown promise, both in simulated environments and on historical data.

In the last few years many improvements have been suggested for the vanilla deep Q learning algorithm. Distributional reinforcement learning in particular has shown to outperform value based deep Q learning on a selection of Atari games. Given the highly stochastic nature of the trading environment it is reasonable to assume that it would perform well for the problem of optimal execution.

In the following work we will outline the principles behind distributional reinforcement learning and show that it can outperform value based deep Q learning for optimal execution. To the best of our knowledge this is the first time distributional reinforcement learning has been used for optimal execution.

Acknowledgements

I would like to thank Paul Bilokon for supervising me and providing me with invaluable advice throughout the thesis. I would additionally like to thank Jörg Osterrieder and the team for giving me the opportunity to present my work at the 5th European Conference on Artificial Intelligence in Industry and Finance. Thank you also to David Finkelstein, Andrew Muzikin and Alexander Panin of DQR for their feedback and suggestions for improvements to the historical trading agents. Finally, thank you to my girlfriend Kate for putting up with me during the writing of this thesis.

Contents

List of Figures

List of Tables

Introduction

Large orders of a financial asset are commonly split it into smaller components and executed over a period of time. Selling quickly may result in higher execution costs. Conversely, executing over a longer period may place the investor at risk of adverse price movements and may be incompatible with the exogenous factors that led the agent to place the order in the first place. The length of time used to execute an order will depend on the investor's tolerance for risk and their reasons for making the order. It is therefore common to pose the problem of optimal execution as the execution of a large order over a predefined time window such that execution costs are minimised. Traditionally this problem has been solved by using a stochastic model of the trading environment and finding the solution that minimises the execution cost using dynamic programming [1]. This assumption of a fixed model arguably oversimplifies a highly complex system.

Unlike model based solutions, deep reinforcement learning is *model free*. Instead it is able to *'learn'* from real experience. This experience typically comes from the reinforcement learning agent's interactions with its environment. The agent observes the current state of the environment, selects an action based on this observation and then receives a reward and observes the new state of the environment. Based on this experience, the agent models the future value of choosing certain actions given the current state. This model of future rewards is then used to adjust its strategy.

The application of reinforcement learning to optimal execution has been proposed in several papers [2, 3, 4]. These attempts have generally been limited to tabular reinforcement learning and dynamic programming. Recent advances in deep reinforcement learning have shown impressive results in various fields, from the game Go [5] to robotics [6]. Since Deep Q Learning was introduced in 2013 [7], many papers have been published showing marked improvements to the original algorithm [8]. Some of these advancements have been incorporated into solutions for optimal execution, in particular *Double Deep Q Learning* [9, 10].

One modification that has shown particular promise is Distributional Reinforcement Learning [11, 12, 13]. Rather than approximating the expected value of the future rewards from taking a particular action in a state, distributional reinforcement learning aims to model the full distribution of returns. Distributional reinforcement learning has shown to both outperform value-based Deep Q Learning and also allow for decisions to be made based on the whole distribution rather than just the expected value. Given the highly stochastic nature of the trading environment, modelling the distribution of the returns is something that could conceivably improve performance. As far as we are aware this is the first time distributional reinforcement learning has been applied to the problem of optimal execution.

In Chapter 1 we will discuss some key topics from Market Microstructure, introducing limit order markets for the trading of financial assets and the effect of trading on the limit order book. We will then briefly introduce the traditional optimal execution solutions of Almgren and Chriss [14] in Section 1.2.

The aim of Chapter 2 is to introduce the reinforcement learning agents we will use to trade. Section 2.1 will introduce the basic concepts behind reinforcement learning needed to understand the deep Q learning. We will use Section 2.2 to discuss deep Q learning. We will start out with a basic version of the *DQN* algorithm and demonstrate some of the improvements that can be made to stabilise the algorithm and improve performance. In particular, we will introduce *Double Deep Q Networks* [15] and *n-step methods* [16] as improvements to the original DQN algorithm. The final part of Chapter 2 will be dedicated to distributional reinforcement learning. This will discuss the theoretical concepts behind the algorithms and some guarantees of convergence. We then define the Quantile Regression DQN algorithm which we shall use to showcase the results of distributional reinforcement learning for optimally executing an order. We will finish by introducing a method for efficiently exploring the state space using distributional reinforcement learning [17].

The focus of Chapter 3 will be the application of these techniques to the field of optimal execution. There are various difficulties in translating agents from simulated to real world environments and these are discussed in Section 3.1. We will then explain how the agent is designed and the restrictions imposed. Finally we will explain the design and rationale behind both a simulated environment and an environment using historical data.

Chapter 4 will review the performance of these agents, trained on the previously introduced environments. We will start by using a simulated environment to compare the performance of QR-DQN to the traditional DQN algorithm and the effectiveness of using the distribution learned using QR-DQN for efficient exploration of the state space. We will then evaluate the performance of QR-DQN in a more realistic historical environment.

Chapter 1

Market Microstructure and Optimal Execution

1.1 Limit Order Markets

A large number of financial assets are traded using electronic limit order markets, including the majority of equities and derivatives [18]. These limit order markets will be the environment that the trading agents considered in the proceeding chapters will operate in. While a thorough treatment of limit order markets can be found in [19], we will begin by giving a brief overview of the most important features.

Limit order markets are venues for the trading of an asset or multiple assets. This asset could be, for example, a stock, US Dollars or Bitcoin. They allow market participants seeking to buy or sell the asset to have their orders matched by the exchange. While it can generally be assumed that all market participants are seeking to make money, we can divide them into three main groups based their motivation for trading [1]. The first group are fundamental traders. These are participants who trade based on exogenous factors. For example an Exchange Traded Fund, or *ETF*, attempting to replicate an index will need to rebalance their portfolio regularly to accurately track the index. The second group are informed traders. These are traders who profit based on information that is not reflected in the current stock price. These traders may depend on non publicly available information, fundamental analysis, statistical arbitrage or simply trading on recent news before the market has had time to react. The third group are market makers. These are generally professional traders who make orders on both sides of the order book and attempt to profit from the spread.

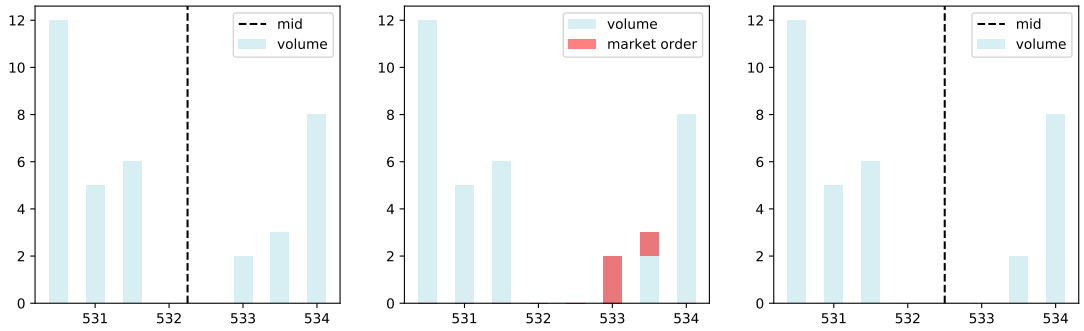
Limit order markets allow for buy and sell orders of the asset being traded to be placed in two distinct ways. Limit orders are obligations to buy or sell the asset at a certain price. They are valid until they are filled, expire, or are cancelled by the trader. If a counterparty executes against the limit order then the trader who placed it must fulfill the order unconditionally. When placing a limit order the trader specifies whether the order is buy or sell, the maximum volume of the asset she is prepared to buy, and the worst price that the she is willing to execute the order at. For a buy limit order this would be a maximum price and for a sell limit order, a minimum. All limit orders are collected together to make up the limit order book. The limit order book consists of a discrete grid of prices and the volume of the asset available at each price. Limit orders may be placed at any of these discrete price points but not between them.

When we discuss the limit order book it is helpful to introduce a few important terms. The minimum difference in price permissible between limit orders is referred to as the *tick size* and the best price available in the limit order book is referred to as the *top of the book*. There are various technical definitions of *liquidity* however it is sufficient to consider it as the volume of the asset available to buy or sell in the limit order book. This is alternatively referred to as the *depth* of the order book.

The second order type we will consider are market orders. In contrast to limit orders, a trader placing a market order specifies only a quantity of the asset and the direction of the order, buy or sell. There is no guaranteed price but the order will be fulfilled immediately, conditional on there being sufficient liquidity in the limit order book. Market orders are executed against the limit order book at the best available price. When a market order arrives if the size of the market order

is less than the volume available at the top of the book then the entire order is executed at this price. If the order size is larger than combined size of the limit orders available at the top of the book then the process is repeated at the next best price until either the market order is fulfilled or there are no remaining limit orders. This concept is known as *walking the order book*. The result of this is the price that the asset is bought or sold at becomes progressively worse as the order size increases. Figure 1.1 illustrates this process and shows the effect of a large market order on the limit order book.

There are many other order types beyond limit and market orders and their availability varies across exchanges. More information on these and their impact on the market can be found in both [1] and [19]. For the sake of simplicity we will restrict our attention to limit and market orders.



(a) 12:43:01.00 (b) 12:43:01.05 (c) 12:43:01.05

Figure 1.1: A model of the effect of an incoming market order on the limit order book. The left hand pane shows the limit order book at 12:43:01. A market order of size $3,000$ arrives at 12:43:01.05. Since the volume available at the top of the limit order book is $2,000$, the order is split between the best ask price, 533, and the second best ask price 533.5. The effect of the market order on the limit order book can be seen in the third panel. The mid price has increased from 532.25 to 532.5 and the spread has widened.

The *bid price*, is the price of the best available buy limit order, and the *ask price*, is the price of the best available sell limit order. From these two prices we can derive the *mid price*,

$$mid = \frac{bid + ask}{2}.$$

This represents one approximation of the current value of the asset, priced by the market. The difference between the bid and ask prices is called the *bid ask spread* or simply the *spread* when there is sufficient context.

The limit order book consists of a queue of many unexecuted limit orders. While the rules for priority in the queue vary between exchanges, the majority operate using price and time priority [18] and we will restrict our attention to these. Price priority means that limit orders with a better price are executed first. If there are multiple limit orders at the same price then the limit order placed first will be the first to be executed. This is sometimes referred to as *"first in first out"*.

1.2 Models for Optimal Execution

For the remainder of this work we will be considering the predicament of a fundamental trader who wishes to enter into, or exit a large position. The trader wishes to minimise the cost of execution. To avoid unnecessary repetition we will only consider a trader wishing to exit a position. The problem of optimal acquisition is symmetric and any concepts for optimal liquidation can be applied to acquisitions with only minor adjustments.

When trading on limit order markets the trader must decide between placing limit orders and market orders. The first will often result in a better price, the second guarantees immediate execution and reduces the risk of adverse price moves prior to execution. This tradeoff between speed and price is a recurring theme in optimal execution. What is considered *optimal* often depends on the individual requirements of the final investor. If, for instance, the agent is liquidating

a position in reaction to a piece of news, they may prefer to execute quickly before the market reacts.

1.2.1 Optimal Execution Using Market Orders

When considering a theoretical environment for optimal execution we will restrict ourselves to execution using only market orders. When placing market orders traders face an execution cost which increases with the size of the order. As noted previously this is due to the concept of walking the order book and can be seen in Figure 1.1. In this example the best ask price is 533 however the average price for the market order is 533.17.

When splitting a large order, the original large market order is referred to as the *parent*, split into a number of *child* orders. A solution for the optimal way in which to split large orders was first discussed by [14]. Since then, the same concepts have been expanded upon and discussed extensively. In the following section we shall loosely follow [1], citing explicit results where appropriate.

In order to evaluate an agent we must be able to compare the performance to a benchmark. The benchmark price is commonly taken to be the mid price when the order was first submitted [1]. The cost of execution or *slippage* can therefore be measured as the difference between the actual cash received from liquidating the position and the cash value at the benchmark price.

Child orders must be executed fast enough so as to minimise the risk of adverse price moves but slow enough so as to minimise slippage. We therefore frame the problem as taking place within a defined time window. We require that the entire position be liquidated by a given time T and seek the optimal trading rate v_t for all $t \in [0, T]$ so as to minimise the execution cost.

To determine the optimal rate of trading we can model the trading environment and derive the rate of trading that minimises the execution cost for this model. To do so we introduce the following stochastic processes,

- $v = (v_t)_{0 \leq t \leq T}$, the rate at which the agent trades.
- $Q^v = (Q^v_t)_{0 \leq t \leq T}$, the agents remaining inventory.
- $S^v = (S^v_t)_{0 \leq t \leq T}$ the mid price process of the asset.
- $\hat{S}^v = (\hat{S}^v_t)_{0 \leq t \leq T}$ the execution price of the asset, the price the agent receives for selling at time t.
- $X^v = (X^v_t)_{0 \leq t \leq T}$, the agent's cash process.

We aim to exit a position of size R by time T. Given we are considering the liquidation of a position, the inventory process can be written as

$$dQ^v_t = -v_t dt \qquad\qquad Q^v_0 = q.$$

Market impact is commonly modelled as being made up of a temporary and permanent impact. We model the temporary trading impact using a function $f : \mathbb{R}_+ \to \mathbb{R}_+$. We will assume this to be a linear function of the rate of trading such that $f(v) = kv$, where k is a constant. Various studies [20, 21] have shown that the temporary market impact is perhaps more accurately modelled as a concave function, $f(v) = k\sqrt{v}$ being one suggestion. As discussed in [1, page 89] however, a linear temporary market impact function appears to be a good approximation and one which simplifies the derivation of the optimal trading speed.

We model the permanent market impact as a function $g : \mathbb{R}_+ \to \mathbb{R}_+$. This is also taken to be a linear function of the rate of trading so that $g(v) = bv$.

Using the permanent market impact function we assume that the dynamics of the midprice are given by,

$$dS^v_t = -g(v_t)dt + \sigma dW_t, \qquad\qquad S^v_0 = S, \tag{1.2.1}$$

where $(W_t)_{0 \leq t \leq T}$ is a standard Brownian motion and σ represents the volatility of the stock. Using the temporary market impact f, we can model the execution price of the stock as

$$\hat{S}^v_t = S^v_t - (1/2\Delta + f(v_t)), \tag{1.2.2}$$

where Δ is the bid ask spread. Using the simplifying assumption that $\Delta = 0$ and substituting in our linear temporary impact function we obtain,

$$\hat{S}_t^v = S_t^v - kv_t.$$

As the speed of trading increases the temporary impact means that the execution price becomes worse, simulating the effect of the market order walking the book.

Under these assumptions we can write out the execution cost of liquidating the position as,

$$EC^v = RS_0 - \mathbb{E}\left[\int_0^T \hat{S}_t^v v_t dt\right] = RS_0 - \mathbb{E}\left[\int_0^T (S_t - kv_t)v_t dt\right].$$

We wish to find the rate of trading v so as to minimise the execution cost. By using the Hamiton Jacobi Bellman, or HJB, equation, we can minimise the execution cost subject to the constraint that the entire position has been sold by time T. This derivation is carried out in full in [1, page 140]. From this we find that the optimal trading speed is in fact constant, and given by,

$$v_t^* = \frac{R}{T}.$$

This strategy is known as Time Weighted Average Price or TWAP. In practice, given we do not trade continuously this strategy consists of splitting the parent order into equally sized child orders and submitting these at equally spaced intervals across the trading window $[0, T]$. The frequency with which these child orders are submitted will depend on the asset being traded. We will refer to this as the trading frequency.

While this strategy is optimal for the simulated market we cannot guarantee that this model of the market is an accurate representation. The unrealistic nature of Black-Scholes dynamics for an asset's price, given in Equation (1.2.1), is well documented [22] and the linear market impact model is a simplistic assumption. Some of these flaws could be solved by the use of a more sophisticated model. However, this may not be tractable and would likely still be a crude approximation. Additionally, this model considers only execution using market orders. A superior solution could be found by using a mix of market and limit orders. Simulating the limit order book however, adds an additional layer of complexity.

For these reasons we will consider alternative approaches to liquidating a position in an optimal way. While suboptimal in real trading conditions, TWAP provides a useful benchmark with which to compare other strategies.

Chapter 2

Deep Reinforcement Learning

In Section 1.2 we used a theoretical model of the market to find the optimal trading rate for the agent given certain observable variables. In contrast to this, reinforcement learning provides a set of methods with the aim of learning this optimal solution through real interactions with the environment. This avoids the previously discussed issues surrounding unrealistic models of financial markets. Recent publications have shown it to be a powerful tool for finding locally optimal strategies, AlphaZero [5] being a prime example.

2.1 Introduction to Reinforcement Learning

In the following section we shall provide a brief introduction to tabular reinforcement learning. For this we shall follow [16] closely, citing specific results where applicable.

2.1.1 Markov Decision Processes

Markov decision processes represent an idealised structure for the environment in which we pose the reinforcement learning problem. They allow us to define concepts in a mathematically rigorous way and derive the convergence results found in later sections. Consider an agent which is able to interact with its environment via a finite set of admissible actions $\mathcal{A}$. This interaction happens at a series of discrete time steps which we index using $t \in \mathbb{N}$. In general there is no requirement for t to be finite. However, for optimal execution we will continue assuming that the agent must liquidate the full position in a defined time window $[0, T]$. This is known as an *episodic* problem. The process terminates either at time T or when the agent has no remaining position in the asset.

At each time step t, the agent observes the state of the environment $S_t \in \mathcal{S}$ and, based on this observation, chooses an action $A_t \in \mathcal{A}(S_t) \subset \mathcal{A}$. The set $\mathcal{A}(S)$ represents the actions available at state S, however we will often drop the state argument for simplicity and assume that the set of admissible actions is not state dependent. Based on the action chosen, the agent then receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and transitions to state $S_{t+1} \in \mathcal{S}$. Here, $\mathcal{R}$ and $\mathcal{S}$ denote the set of possible rewards and states respectively.

When an action A_t is chosen by the agent, the state transitioned to, and accompanying reward, are defined by a probability distribution which completely characterises the dynamics of the process. We can therefore define the dynamics of the process as a four argument function,

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a]. \tag{2.1.1}$$

We will assume that the state process $\{S_t\}_t$ is Markov: the current state represents all available information about the environment, independent of past events. Formally for any $s_{t+1} \in \mathcal{S}$ and $r \in \mathcal{R}$,

$$\mathbb{P}[S_{t+1} = s_{t+1}, R_{t+1} = r|S_t = s_t, A_t = a_t] =$$
$$\mathbb{P}[S_{t+1} = s_{t+1}, R_{t+1} = r|S_t = s_t, A_t = a_t, ..., S_0 = s_0, A_0 = a_0],$$

for all $t \in \mathbb{N}$. A Markov Decision Process or *MDP* is the name given to the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$. In some reinforcement learning literature a discount factor γ is included in this tuple. In the following sections we will be modelling all reinforcement learning problems as MDPs. In this section, while

discussing tabular reinforcement learning we will additionally assume that the state and action spaces are of finite cardinality. In this case the MDP is known as a finite MDP.

2.1.2 Optimal Policies

Reinforcement learning aims to provide a mapping from the states $\mathcal{S}$ to the set of actions $\mathcal{A}(\mathcal{S})$ that is in some way optimal. This mapping is referred to as the strategy, or *policy*, π that the agent follows. An agent following policy π in state s chooses action a with probability $\pi(a|s)$. Alternatively, for deterministic policies it is common to write $\pi(s)$ to denote the action chosen in state s under a policy π. We now define two important functions. Given an agent currently in state s, the value function $v_\pi : \mathcal{S} \to \mathbb{R}$ is the expected total discounted reward over the remainder of the episode if policy π is followed. Formally,

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=1}^{T} \gamma^{k-1} R_{t+k} | S_t = s \right], \tag{2.1.2}$$

where γ is the discount factor. In a similar way, we define $q_\pi(s, a)$ to be the expected future discounted reward if an agent in state s chooses action a, and policy π is followed for the remainder of the episode. We can write this as

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=1}^{T} \gamma^{k-1} R_{t+k} | S_t = s, A_t = a \right]. \tag{2.1.3}$$

This is known as the action value function for policy π. Since the problems we are considering are episodic, the discount factor is unnecessary and we set $\gamma = 1$. We will continue to include it for completeness.

By manipulating Equation (2.1.3) and defining $G_t = \sum_{k=t+1}^{T} \gamma^{k-1} R_k$, the future discounted returns, we can derive the following recursive relationship,

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a \right] \\ &= \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \\ &= \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \Big[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \Big], \end{aligned} \tag{2.1.4}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ where $p(s', r|s, a)$ defines the dynamics of the MDP, given in Equation (2.1.1). The equality given by Equation (2.1.4) is known as the Bellman Equation for $q_\pi(s, a)$. By a similar derivation we can obtain the Bellman Equation for $v_\pi(s)$,

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big]. \tag{2.1.5}$$

To find an optimal policy we must be able to compare policies. We assert that $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. Moreover, $\pi > \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$ and $v_\pi(s) > v_{\pi'}(s)$ for at least one $s \in \mathcal{S}$. Using this means of comparing policies we can define the optimal policy π^* such that $\pi^* \geq \pi$, for any admissible policy π. The optimal policy is guaranteed to exist but is not necessarily unique. [16, page 62]. We can similarly define the optimal value and action value functions as

$$v_*(s) = \max_\pi v_\pi(s), \qquad q_*(s, a) = \max_\pi q_\pi(s, a),$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. By using these definitions with the Bellman Equation from Equation (2.1.4) we obtain,

$$\begin{aligned} q_*(s, a) &= \mathbb{E} \left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} q_*(S_{t+1}, a') \big| S_t = s, A_t = a \right] \\ &= \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \Big[r + \gamma \max_{a' \in \mathcal{A}} q_*(s', a') \Big]. \end{aligned} \tag{2.1.6}$$

This relationship is known as the Bellman Optimality Equation for the action value function.

Finding π^* requires two distinct processes. In order to compare to policies we are required to know or estimate the value functions or action value functions of both policies. Policy evaluation consists of estimating the value function v_π, or alternatively q_π, for a given policy π. By using the Bellman Equation defined in Equation (2.1.5) we can compute v_π iteratively using the update rule,

$$v_{k+1}(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S},r\in\mathcal{R}} p(s',r|s,a)\Big[r+\gamma v_k(s')\Big], \tag{2.1.7}$$

with v_0 chosen arbitrarily. For episodic problems, v_π exists and is unique, and the sequence $\{v_i\}_{i=0,..,k}$ is guaranteed to converge to v_π as $k\to\infty$ [16, page 74]. In a similar way we can estimate the action value function q_π for a given state s and action a,

$$q_{k+1}(s,a) = \sum_{s'\in\mathcal{S},r\in\mathcal{R}} p(s',r|s,a)\Big[r+\gamma \sum_{a'\in\mathcal{A}} \pi(a'|s')q_k(s',a')\Big]. \tag{2.1.8}$$

The second process is *policy improvement*. At each step the agent has the choice of whether to continue using the current policy or switch to a new policy. We introduce the following theorem from [23, page 47],

Theorem 2.1.1 (Policy Improvement Theorem)**.** *Let π and π' be policies such that, for all $s\in\mathcal{S}$,*

$$q_\pi(s,\pi'(s)) \geq v_\pi(s),$$

then policy $\pi'\geq\pi$.

A sketch of this proof for deterministic policies is given in [16, page 78]. We now introduce a special deterministic policy known as the *greedy* policy π'. Consider an agent following an arbitrary policy π. We can construct the greedy policy by choosing the action $a\in\mathcal{A}$ which maximises the action value function $q_\pi(s,a)$ for the current state $s\in\mathcal{S}$. Explicitly,

$$\begin{aligned}\pi'(s) &= \arg\max_{a\in\mathcal{A}} q_\pi(s,a)\\ &= \arg\max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S},r\in\mathcal{R}} p(s',r|s,a)\Big[r+\gamma v_\pi(s')\Big]\end{aligned} \tag{2.1.9}$$

where the second line follows from Equation (2.1.4). By definition the greedy policy satisfies the conditions of Theorem 2.1.1. The greedy policy π' is therefore at least as good as the original policy π. This process of changing the original policy to be greedy with respect to the action value function is known as policy improvement. It can be easily shown that at each state $s\in\mathcal{S}$, policy improvement results in strictly better policy unless the policy is already optimal for that state [16, page 79]. While the greedy policy presented here is deterministic, the results equally apply to any policy which assigns zero probability to any non-maximal action.

To find the optimal policy we require both policy evaluation and policy improvement. The definition of the greedy policy given in Equation (2.1.9) assumes that the action value function q_π is known. In general we only have an approximation of the action value function. *Policy iteration* describes the alternating process of policy evaluation, and policy improvement. Policy evaluation improves the approximation of the action value function and policy improvement improves the policy π,

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} ...$$

E represents the process of policy evaluation and I that of policy improvement [16, page 80]. When carrying out policy evaluation at each step one could iterate as per Equation (2.1.7) until the action value function had suitably converged. The potentially computationally intensive nature of this step however may make this an impractical proposal. An alternative to this is to cut the policy evaluation step short after a small number of iterations. An extreme form of this is known as *value iteration*, where we use just one iteration per policy evaluation step. This allows us to simplify the policy iteration process to,

$$\begin{aligned}v_{k+1}(s) &= \max_{a\in\mathcal{A}} \mathbb{E}[R_{t+1}+\gamma v_k(S_{t+1})|S_t=s,A_t=a]\\ &= \max_{a\in\mathcal{A}} \sum_{s'\in\mathcal{S},r\in\mathcal{R}} p(s',r|s,a)\Big[r+\gamma v_k(s')\Big],\end{aligned} \tag{2.1.10}$$

for all $s \in \mathcal{S}$, where v_0 is chosen arbitrarily. This sequence can be shown to converge in the limit to $v_*(s)$ for all $s \in \mathcal{S}$ for episodic problems [16, page 83].

2.1.3 Temporal Difference Learning

The update rule given by Equation (2.1.10) requires that each update is applied to every state $s \in \mathcal{S}$. A sweep of the state space may very well be computationally demanding and impractical to carry out at every step. In addition, in a trading environment we do not know the underlying dynamics of the MDP and an update of the form given in Equation (2.1.10) is therefore impossible. Instead we seek to learn them from interactions with the environment.

Temporal Difference, or *TD*, methods update an estimate Q of q_π by using experience sampled when following the policy π. It should be noted that Q is still dependent on the policy followed however the subscript π is not included to maintain consistency with the literature[1]. A one step or $TD(0)$ update can be written as,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.1.11}$$

where α is a parameter controlling the learning rate [16, pages 120-129]. For now we ignore the concept of n step or $TD(\lambda)$ updates which we shall return to in Section 2.2.6. By comparing this update rule to Equation (2.1.8), we note that they are conceptually similar but with a few important differences. The TD(0) update rule, does not require explicit knowledge of the dynamics of the MDP, $p(s', r|s, a)$. Rather, the update is based on unbiased samples S_{t+1} and R_{t+1} of the possible rewards and next states. Further, unlike Equation (2.1.8), the TD(0) update only moves the estimate for Q towards the target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ rather than setting them to be equal. The rate at which Q is improved is controlled by the parameter α. While the policy π does not directly appear in the TD(0) update rule, it is present implicitly given that S_{t+1} and R_{t+1} were sampled under policy π given state S_t.

As was the case in the previous section, the estimate $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ with which we update the value function, is a bootstrapped estimate. It relies on an approximation of the action value function at the next state $Q(S_{t+1}, A_{t+1})$. under mild stochastic convergence conditions on the step size parameter α and for a fixed policy π, the action value function $Q(S_t, A_t)$ updated using the TD(0) update rule has been shown to converge to the true function in the limit with probability 1 [16, page 124-126].

2.1.4 Exploration versus Exploitation

When attempting to apply the process of Temporal Difference Learning to policy improvement we immediately face a problem that is more succinctly explained by means of an example.

Consider an agent operating in a simple stock market environment with inventory s. With a slight abuse of notation we use the state to denote the number of shares remaining. At the beginning of the episode the agent has two shares, $S_0 = 2$ and the episode terminates at the smallest t such that $S_t = 0$. At each step t the agent can sell a quantity $A_t \in \mathcal{A}(S_t)$ shares where $\mathcal{A}(S_t) = \{a \in \{0, 1, 2\} : a \leq S_t\}$. The reward R_t the agent receives for a sale at time t is a deterministic function of A_t given by $R_t = \sqrt{A_t}$. A diagrammatic representation of the problem is given in Figure 2.1. This model is a simplistic representation of optimal execution in a market with temporary impact; the cash received decreases with the rate of trading.

We initialise the action value function as 0 for all states and actions and consider an agent following the greedy strategy defined in Equation (2.1.9), breaking any ties at random. One possible trajectory that the agent could take for the first episode is,

$$S_0 = 2, A_0 = 2, S_1 = 0, R_1 = \sqrt{2}$$

We now update the value function for using the TD(0) update given in Equation (2.1.11), using $\alpha = 0.5$, for both states visited. The updated estimates are,

$$Q(s, a) = \begin{cases} \sqrt{2}/2 & \text{for } s = 2, a = 2 \\ 0 & \text{otherwise,} \end{cases}$$

[1] Implicit in the lack of subscript is the assumption that we are attempting to approximate an optimal policy, rather than an arbitrary policy π. While there may be many optimal policies, all of them will have identical action value functions.

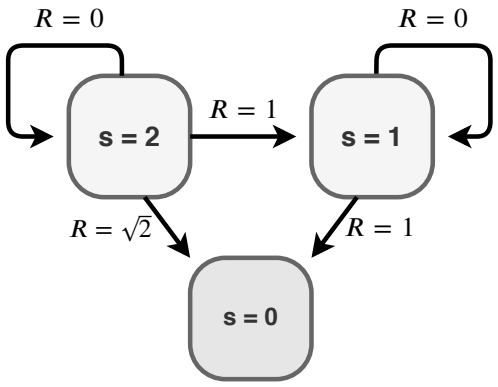


Figure 2.1: A simple model of a deterministic trading environment with two shares and temporary impact. State $s = 0$ is a terminal state.

If, as above, the agent chooses to sell both shares at $t = 0$ in the first episode then the next episode and all future episodes will follow the same trajectory as the first. In other words the value policy has converged to a suboptimal policy. The optimal action $A_1 = 1$ will never be chosen and its value therefore is not learned.

This simple example introduces the problem of exploration versus exploitation. The update rule defined in Equation (2.1.10) is guaranteed to converge because the iterative updates are applied to all $s \in \mathcal{S}$. Temporal difference learning only samples actions taken by the current policy and some states may never be visited.

The solution to this problem is to use a more exploratory strategy. Some of the most popular are *epsilon greedy* strategies. For a given episode i and time step t, given a small $\epsilon_i \in (0, 1)$, choose a uniformly random action from $\mathcal{A}$ with probability ϵ_i, otherwise choose a greedy action. If multiple actions have the same action value function then there may be several optimal actions. In this case the greedy action is chosen at random from the set of optimal actions. Formally, we can write epsilon greedy policies as,

$$\pi_\epsilon(a|S_t = s) = \begin{cases} (1-\epsilon)/|\mathcal{A}^*(S_t)| + \epsilon/|\mathcal{A}(S_t)| & \text{if } a \in \mathcal{A}^*, \\ \epsilon/|\mathcal{A}(S_t)| & \text{otherwise,} \end{cases}$$

where we define $\mathcal{A}^*(S_t) = \{a \in \mathcal{A}(S_t) : q(S_t, a) = \arg\max_{a' \in \mathcal{A}} q(S_t, a')\}$, the set of optimal actions.

One choice of ϵ_i would be a constant over all episodes, $\epsilon_i = \epsilon$, for some constant $\epsilon \in (0, 1)$. It is more common however to choose ϵ_i to be a decreasing function of i so that the exploratory affect decreases as the number of training episodes increases. This reflects the decreasing uncertainty in the action values as the agent is trained. Hence forth all ϵ-greedy strategies considered will use a decreasing epsilon. At each step, epsilon is multiplied by a constant $\epsilon_d \in (0, 1)$ until it reaches a minimum value $\epsilon_m \in (0, 1)$.

Alongside ϵ-greedy strategies, there are also a class of strategies where exploration is prioritised for states with a more uncertain value. These include *Upper Confidence Bound*, or UCB policies for single state problems [16, page 35] which unfortunately do not generalise to multistate problems. We will discuss other options for this in Section 2.3.4.

2.1.5 Q Learning

When introducing TD learning we made the assumption that the policy being learned and the policy that the samples were generated from were one and the same. This is referred to as *on policy* learning. *Off policy* learning instead involves using two policies, a *target policy* and a *behaviour policy*. The agent interacts with the environment following the behaviour policy. The experience generated from these interactions is used to update the target policy and, optionally, the behaviour policy. On policy learning is in fact a special case of off policy learning where the behaviour and target policies coincide.

Off policy learning allows for the use of a more exploratory policy as the behaviour policy, for example an ϵ-greedy policy, and a deterministic, often greedy, strategy as the target. *Q-learning*

is an off policy learning algorithm originally introduced by [23]. Q-learning uses a greedy target policy and an exploratory behaviour policy. The $TD(0)$ update rule from Equation (2.1.11) is replaced by,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)], \tag{2.1.12}$$

where there is no assumption that the optimal policy is followed when choosing A_t. Under some mild conditions it can be shown that $Q(s, a) \to q_*(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ as the number of samples tends to infinity [23, page 227], assuming a finite MDP.

2.2 Deep Q Learning

Country to our assumptions so far, the cardinality of the state space $|\mathcal{S}|$ for a trading environment may well not be finite. So far we have described tabular methods for reinforcement learning. At each step, an estimate of either the action value function $q_\pi(s, a)$ or value function $v_\pi(s)$ is updated for state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ independent of all other states and actions. While powerful for small state spaces, this approach becomes impractical as the sizes of the state and action spaces grow. The action value function must be updated for every individual combination of state and action without any generalisation.

An alternative is to use function approximation to estimate the action value function q_π. Traditionally, linear function approximators have been popular for this purpose [24] however the range of true value functions that they can approximate is limited. Alternatively, neural networks provide a highly flexible class of functions that we may use to approximate q_π.

2.2.1 Deep Q Learning Update

One of the earliest successes using a form of deep learning as a non linear function approximator for q was TD-Gammon, a reinforcement learning solution to Backgammon which achieved superhuman performance when trained entirely through self play [25].

Recent developments have led to the Deep Q Network, or *DQN*, algorithm introduced in [7]. Let Q represent a neural network, referred to as a *Q-network*, with weights θ. We recall the concept of policy evaluation, introduced in Section 2.1.2. For deep Q learning this process involves training the Q-network over a number of iterations, indexed using i. Since the weights θ will change as the network is trained we refer to the weights at iteration i as θ_i. At each iteration we train the network by minimising the loss function,

$$L_i(\theta_i) := \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s, a; \theta_i))^2 \right], \tag{2.2.1}$$

where the target y_i for the update at iteration i is given by

$$y_i = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a]. \tag{2.2.2}$$

The notation L_i makes it explicit that this loss function changes with each iteration. The notation ρ is used to denote that experience is generated under the behaviour policy, and ε denotes the environment.

When minimising the loss function given in Equation (2.2.1), rather than compute the expectations directly, we may use samples generated under the behaviour policy, and minimise using stochastic gradient decent. We can define an experience to be a tuple of the form $(s_t, a_t, s_{t+1}, r_{t+1})$, where $s_t \in \mathcal{S}$ and $a_t \in \mathcal{A}(s_t)$ denote the state at time t and the action taken at time t respectively. The variables $r_{t+1} \in \mathbb{R}$ and $s_{t+1} \in \mathcal{S}$ denote the reward received and state transitioned to respectively, as a result of taking action a_t from s_t.

When training the Q-network, at the i^{th} iteration, given an experience (s, a, s', r) we minimise,

$$(r + \gamma \max_{a' \in \mathcal{A}(s)} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))^2 \tag{2.2.3}$$

with respect to θ_i. This update rule directly follows from the Q learning update defined in Equation (2.1.12) with the important difference that we have not limited ourselves to one iteration per timestep. As with tabular Q-learning, Deep Q Learning is *off policy* since the observations can be generated from a behaviour policy ρ, different from the greedy target policy.

2.2.2 Artificial Neural Networks

DQN aims to approximate the action value function $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ with a neural network. Artificial neural networks allow for flexible function approximation in high dimensions. Since the state may be a high dimensional object and the action value a non trivial function, neural networks are well suited to the problem. A comprehensive introduction can be found in [26]. The aim of this section is rather to introduce concepts directly related to reinforcement learning.

We will be using *Feedforward Neural Networks* for the Q-network. Feedforward neural networks are essentially alternate compositions of affine functions and non linear *activation functions*. Neural networks consist of an input, a number of hidden layers and an output. Each layer is made up of a number of *units* or *nodes*. We can define the neural network with r layers with the i^{th} layer having d_i units as being a function of the form

$$f = \sigma_r \circ L_r \circ ... \circ \sigma_1 \circ L_1,$$

where $L_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ for all $i = 1, ..., r$ is a function of the form,

$$L_i(x) := W^i x + b^i, x \in \mathbb{R}^{d_{i-1}}, \tag{2.2.4}$$

parameterised by a weights matrix $W^i = [W^i_{j,k}]_{j=1,...,d_i, k=1,...,d_{i-1}} \in \mathbb{R}^{d_i \times d_{i-1}}$ and bias vector $b^i = (b^i_1, ..., b^i_{d_i}) \in \mathbb{R}^{d_i}$ [27, Definition 2.1]. If $I, O \in \mathbb{N}$ denote the dimension of the input and output respectively then we define $d_0 = I$ and $d_r = O$. In this definition σ_i represents the i^{th} activation function. The models we shall consider will all have the same number of units in each layer and will use *ReLU* activation functions. ReLU stands for *Rectified Linear Unit* and is defined as,

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} , \end{cases}$$

for any $x \in \mathbb{R}$.

The layer defined in Equation (2.2.4) is known as a *fully connected layer*. For time series inputs, or any input where the adjacency of the input is important, we may wish to make this structure more visible to the network. One way of achieving this is by using *convolutional layers*. A one dimensional convolutional layer with d inputs is a function $C_k : \mathbb{R}^d \to \mathbb{R}^{d-l-l'}$ given by

$$C_k(x)_i := \sum_{j=-l'}^{l} k_j x_{i+j}, \quad i = l' + 1, ..., d - l,$$

where $l, l' \in \mathbb{N}$ are parameters such that $l + l' + 1 < d$, and $\boldsymbol{k} = (k_{-l'}, ..., k_l)$ is the kernal [27]. The kernal is typically a learnable set of weights $k_1, ..., k_l$.

In supervised learning, neural networks are generally trained using stochastic gradient decent, outlined in [26, pages 82, 150]. For a given observation, this adjusts the weights and biases of the network by a small amount in the direction that maximises the reduction in the error for that observation. This relies on the availability of labelled training data, examples of network inputs along with the expected output.

In the case of DQN we are using the network to approximate the action value function, the true value of which will generally be unknown. Instead, the loss function given in Equation (2.2.1) uses a bootstrapped estimate of the action value function $\mathbb{E}_{s' \sim \varepsilon}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a]$ as the target. This is sometimes termed *semi-gradient decent* [16, page 202]. This use of a bootstrapped estimate of the value function speeds up training and allows for the agent to learn throughout the episode. It does however come at the cost of increased instability and the potential for the action value estimate to diverge. In the following sections we will discuss ways in which to stabilise training and reduce the possibility of divergence.

We can often reduce the training time and improve the performance by standardising the inputs and targets for the neural network. While there is technically no necessity for the inputs to be standardised, doing so makes it less likely that the network converges to a local minimum [28]. Additionally it means that inputs with differing scales are treated equally. We can see this by considering a trading agent with two inputs, the time elapsed, in hours, and the remaining number of shares of a $10,000$ share position. If the agent were to liquidate the position over the period of an hour, the elapsed time would range over the interval $[0, 1]$ but the remaining position would be in the interval $[0, 10000]$. The network would therefore be more sensitive to changes in the remaining position than the elapsed time.

2.2.3 Planning and Replay Buffers

Reinforcement Learning algorithms can be classified as either *model free* or *model based* methods [16, page 159]. In this work we have considered only model free methods however it is instructive to provide a brief overview of the alternatives as justification of various design choices we will make for our learning agents.

Model based methods rely on a parametric model of their environment. This model is used for *planning*. Planning, in this context, refers to the process of using simulated experience generated by the agent's model of the environment, rather than real experience from interactions with the environment. A planning agent can either use real experience to improve its model of the environment, known as *indirect reinforcement learning*, or directly improve the value function in the manner discussed in previous sections.

One argument in favour of indirect methods is that they make more efficient use of the available experience. DYNA-Q is a planning agent which supplements real experience with simulated experience, generated using a model of the environment. In some circumstances this agent has been shown to significantly outperform a model free agent [16, Example 8.1].

An alternative to the use of parametric models for planning is by using *experience replay* [29]. When the agent interacts with the environment, the experience, is recorded. At regular intervals during training, the agent "remembers" a selection of past experiences and uses these to train the Q-network.

To be precise, at each time step t the agent adds the tuple defining the interaction: the state, action, next state and reward $(s_t, a_t, s_{t+1}, r_{t+1})$, to a replay buffer $\mathcal{D}$. It then selects, uniformly at random, a number of past experiences and trains the network on these experiences. We term the size of the sample the *batch size* B, a hyperparameter which must be specified when designing the agent. This process is in many senses similar to the use of planning and aims to solve the same fundamental issues. Rather than simulating experiences using a model based on previous interactions, we instead simply reuse these previous interactions. Experience replay has been shown to outperform DYNA style planning on the Atari Learning Environment and is in general a more stable solution [30].

Experience replay has the dual benefit of making more efficient use of real experiences and decorrelating the experiences used to train the agent. Each batch of training experiences is a random sample from all previous interactions and not necessarily sequential. If the agent were to be trained only using experience generated at the current time step this would influence the action chosen at the following time step. This would result in the samples used to train the network being correlated, possibly leading to damaging feedback loops. In some cases this has been shown to lead to divergence [31]. This decorrelating effect is a particularly important feature of experience replay.

To reduce memory requirements and prioritise more relevant recent experiences for training we specify a maximum number of experiences for the replay buffer. When the replay buffer is full, experiences can be removed either at random or based on time priority, with less recent experiences being removed first. The use of experience replay has been shown to improve both absolute performance and efficiency, assuming that the environment does not change rapidly [24, 29].

2.2.4 Target Network

One cause of instability when training Q-networks is that the target used to train the network depends on the current network parameters. An update that increases $Q(s_t, a_t)$ also often increases $Q(s_{t+1}, a)$ for all $a \in \mathcal{A}$ and since $Q(s_{t+1}, a)$ is present in the target when updating $Q(s_t, a)$, there is potential for feedback loops or divergence [24].

A solution to this problem is to introduce a separate network which is used to calculate the target. Specifically, we use a separate *target network*, to estimate the value of the next state, and update the weights of the target network to be equal to the Q-network periodically, rather than at every iteration. Given an experience (s, a, s', r), we can modify the target used in Equation (2.2.3) to be,

$$r + \gamma \max_{a' \in \mathcal{A}(s)} Q(s', a'; \theta^-), \tag{2.2.5}$$

where θ^- denotes the target network. In practise the target network is a duplicate of the Q-network, with weights updated to match the current Q-network every C steps. Explicitly, at step

$t = iC$ for all $i \in \mathbb{Z}$, we set

$$\theta^- := \theta_i.$$

For the agents in simulated environment that this technique was conceived for, this lag C was large at $10,000$ steps [24]. With limited training data the target lag must be much smaller. A value of 14 was chosen in previous implementations for optimal execution [9] and when evaluating agent performance in Chapter 4, a lag of 50 will be used. For this reason we introduce a small modification to the original implementation. Rather than updating the target network weights to be the same as the current Q-network weights, we instead update them to the network weights C steps in the past. At step $t = iC$ for all $i \in \mathbb{Z}$, we set

$$\theta^- := \theta_{i-C}.$$

This ensures that there is always a delay of at least C steps between the current Q-network and the target network. We could of course update the target network at every step to be a copy of the Q-network C steps prior. This would maintain a constant lag at an additional computational cost. Additionally it would require a copy of the Q-network for the previous C steps to be stored resulting in greater use of memory. By only updating the network every C steps we are only required to store one copy of the Q-network weights and the computational cost remains almost the same as the original implementation.

2.2.5 Maximisation Bias and Double Deep Q

In stochastic environments the use of Q learning can often result in overestimation of the action value function. This is due to maximisation bias when updating the estimate of the greedy policy. From Equation (2.1.12) we have that the target for an update to $Q(S_t, A_t)$ is $R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a)$. Taking the expectation we have

$$\begin{aligned}\mathbb{E}[R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a)] &= \mathbb{E}[R_{t+1}] + \gamma \mathbb{E}[\max_{a \in \mathcal{A}} Q(S_{t+1}, a)] \\ &> \mathbb{E}[R_{t+1}] + \gamma \max_{a \in \mathcal{A}} \mathbb{E}[Q(S_{t+1}, a)],\end{aligned}$$

the last step following by Jensen's inequality given that the maximum is a strictly convex function. It is therefore apparent that the expectation of the target used in the standard Q learning update is larger than the expectation of the true action value.

That this has the potential to lead to sub-optimal strategies can be illustrated by the following example. Consider a problem with three possible actions. The first action leads to a reward $R_1 = 1/2$, the second and third each result in a normally distributed reward R_2 and R_3, both with mean 0 and variance 1. The expected reward $\mathbb{E}[R_i] = 0$ for $i \in \{1, 2\}$, however,

$$\mathbb{E}\left[\max_{i \in \{1,2\}} R_i\right] = \frac{1}{\sqrt{2}} > \frac{1}{2} = \mathbb{E}[R_1],$$

using [32] for the moments of the sum of two normally distributed variables. After a single sample it could therefore be expected that an agent would choose a sub-optimal action.

Double Q Learning, introduced in [33], provides a solution to this problem in the tabular setting. We write θ and θ' to denote two sets of parameters for the neural network used to approximate the action values Q. We alternate between updating the two sets of parameters θ and θ'. We then rewrite the Q learning update from Equation (2.1.12) as,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, \arg\max_{a \in \mathcal{A}} Q(S_{t+1}, a; \theta_t); \theta'_t) - Q(S_t, A_t)\right],$$

By maximising over a set of independent estimates for the action values we avoid maximisation bias. The affect of the maximisation bias for DQN can in fact be quantified and we can give a lower bound in the case where all action values are equal [15, Theorem 1],

Theorem 2.2.1. *Consider a state s in which all the true optimal action values are equal at $Q_*(s, a) = V_*(s)$ for some $V_*(s)$. Let Q_t be arbitrary value estimates that are on the whole unbiased*

in the sense that $\sum_a (Q_t(s,a) - V_*(s)) = 0$ *but that not all are correct, such that* $1/m \sum_a (Q_t(s,a) - V_*(s))^2 = C$ *for some* $C > 0$*, where* $m \geq 2$ *is the number of actions in* s*. Under these conditions,*

$$\max_a Q_t(s,a) \geq V_*(s) + \sqrt{\frac{C}{m-1}}.$$

This lower bound is tight. Under the same conditions the lower bound on the absolute error of the Double Q learning estimate is 0.

From the theorem the linear dependence of the magnitude of the bias on the standard deviation of the sample is clear. In trading environments where the signal to noise ratio is low the importance of Double Q Learning becomes apparent.

Double Deep Q Learning, or DDQN, was introduced in [15] and provides analogous results for Deep Q Learning. Rather than use a separate network we instead use the target network introduced in Section 2.2.4. The target of the update given by Equation 2.2.5 for iteration i therefore becomes

$$r + \gamma Q(s', \arg\max_{a \in \mathcal{A}} Q(s', a; \theta_i); \theta^-), \tag{2.2.6}$$

for a given experience (s, a, s', r) where θ^- are the weights for the target network. Crucially, the target network is still used to estimate the value of the next action however this action is chosen by maximising over the current Q-network. By using the target network rather than the separate network from Double Q Learning, the additional computation and memory requirements are minimised.

2.2.6 N Step Methods

The TD(0) update rule shown in Equation (2.1.11) uses only the state and reward at $t+1$ when updating $Q(S_t, A_t)$. Future rewards received beyond this point are estimated using the action value function at S_{t+1}. Since training experiences are sampled out of sequence from the experience replay buffer we can extend this update to incorporate the states visited and rewards received at steps beyond $t+1$. This is of course provided these interactions have already happened. In the extreme this involves using all rewards received up until the termination of the episode. Methods which update the target using the whole episode are known as Monte Carlo methods. In practise Monte Carlo methods do not perform as well as Temporal Difference learning [16, page 128].

Rather than using the whole episode as with Monte Carlo methods, we can instead look ahead for a small number of steps. These methods are known as n-step methods or alternatively *Eligibility Traces*. For this work we will focus on n-step methods which were found to improve performance of DQN agents in nearly all the Atari games available in the Atari Learning Environment [8, 34].

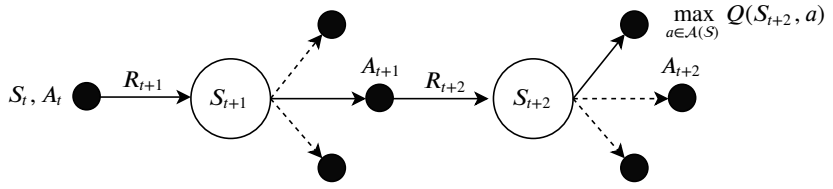


Figure 2.2: A representation of the n-step Tree Backup Algorithm for the greedy policy. Solid lines represent the action which maximises the action value over the set of possible actions for that state.

In practise there are several ways to implement n-step updates. Here we discuss the n-Step Tree Backup Algorithm [16, page 152]. Figure 2.2 shows a diagrammatic representation of the algorithm for the special case of Q learning. When updating state $Q(S_t, A_t)$, if the action taken at $t+1$, A_{t+1} is the same as the one which would be chosen given the current network weights, $\max_{a \in \mathcal{A}(\mathcal{S})} Q(S_{t+1}, a)$, then the prediction of future value given by $\max_{a \in \mathcal{A}(\mathcal{S})} Q(S_{t+1}, a)$ in Equation (2.1.12) can be replaced with the actual reward observed and the process repeated with the

next state until $t+n$. We can write the update rule for a single update to $Q(S_t, A_t)$ as an algorithm,

Algorithm 1: n-step Tree Backup Update for Q Learning

Result: Returns the target for a deep Q update G
For a Sequence $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, ...$
Set $k = 1$
Set $G = R_{t+1}$
while $A_{t+k} = \arg\max_{a \in \mathcal{A}(S_{t+k})} Q(S_{t+k}, a)$ *and* $k \leq n$ **do**
  if S_{t+k} *is terminal* **then**
    return G
  end
  Set $G = G + R_{t+k+1}$
end
Set $G = G + \max_{a \in \mathcal{A}(S_{t+k})} Q(S_{t+k}, a)$
return G

This process depends of course on the requisite future experiences being present in the replay buffer. For this reason we choose to use time priority when removing experiences from the replay buffer and only train using experiences which are at least n steps removed from the agent's current time step.

2.2.7 Combining the Improvements to DQN

The combination of function approximation of the value function, bootstrapping and off policy learning can in some circumstances lead to instability, and has for this reason been termed *the deadly triad* by Sutton and Barto [16, page 264]. Many improvements have been proposed to stabilise and enhance the basic DQN algorithm. Each of the concepts introduced in the preceding sections: target networks, experience replay, DDQN and n-step methods, have been shown to go some way towards increasing the stability of the DQN algorithm [31]. While other adjustments to DQN have been shown to improve performance, for example those implemented by the Rainbow Algorithm [8], these improvements in particular have been hypothesised to provide the greatest improvement to stability [31]. Additionally, they are the features of *Rainbow* which would seem to be most beneficial for the trading environment.

By combining these improvements to vanilla DQN we obtain the following algorithm, stated in full for clarity. The performance of this algorithm in the trading environment will be evaluated in Section 3.3.1.

Algorithm 2: Full algorithm for Double Deep Q Learning with n-step tree back up, target network and experience replay.

Initialise θ
Set $\hat{\theta}, \theta^- = \theta$
for *episode = 1,...,M* **do**
  Initialise s_0
  Set $\hat{\theta}, \theta^- = \theta$
  for *t=0,...,T* **do**
    Select action a_t from behaviour policy distribution $\pi_\rho(a|s_t)$
    Execute action a_t and observe reward r_{t+1} and new state s_{t+1}
    Add experience $(s_t, a_t, r_{t+1}, s_{t+1})$ to the replay buffer $\mathcal{D}$
    if *Replay buffer is full* **then**
      Remove least recent experience buffer
    end
    Sample a minibatch of experiences $\{(s_{t_j}, a_{t_j}, r_{t_j+1}, s_{t_j+1})\}_{j=1,...,B}$ uniformly at random from $\mathcal{D}$
    for $j = 1, ..., B$ **do**
      # *n-step tree backup*
      Set $k = 1$
      Set $G_j = r_{t_j+1}$
      while $a_{t_j+k} = \arg\max_{a \in \mathcal{A}(s_{t_j+k})} Q(s_{t_j+k}, a; \theta)$ *and* $k \leq n$ *and* s_{t_j+k} *is not terminal* **do**
        Set $G_j = G_j + r_{t_j+k+1}$
      end
      if s_{t_j+k} *is not terminal* **then**
        Set $a^* = \arg\max_{a \in \mathcal{A}(s_{t_j+k})} Q(s_{t_j+k}, a; \theta^-)$
        Set $G_j = G_j + Q(s_{t_j+k}, a^*; \theta^-)$
      end
    end
    Perform minibatch gradient update with respect to weights θ using loss $(G_j - Q(s_{t_j}, a_{t_j}; \theta))^2$.
    Set $s_t = s_{t+1}$
    Set $t = t + 1$
    if *episode is divisible by C* **then**
      Set $\theta^- = \hat{\theta}$ and
      Set $\hat{\theta} = \theta$
    end
  end
end

2.3 Distributional Reinforcement Learning

So far the goal in this chapter has been to predict the value function of a state or state action pair. This is equivalent to estimating the expected value to the agent of a particular state, or an action taken in that state. From this perspective the Bellman Equation given in Equation (2.1.4) could be equivalently written as

$$\begin{aligned} Q(s,a) &= \mathbb{E}[R(s,a)] + \gamma \mathbb{E}[Q(S',A')] \\ &= \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R(s_t,a_t)\right] \\ &=: \mathbb{E}[G(s,a)], \end{aligned} \tag{2.3.1}$$

where s and a are respectively the current state and action, and S' and A' denote the random next state and action. The reward R is also random and conditioned on the current state and action. For the remainder of the section we will in general use the convention that capital letters denote random variables.

In the following section we will outline an alternative, more general approach and consider the distribution, rather than the expected value, of the return for a particular state action pair. Recent works [11, 8, 12, 13] have shown impressive performance gains using a collection of methods falling under the umbrella of Distributional Reinforcement Learning. We will cover a few of the theoretical results underpinning distributional reinforcement learning and discuss the advantages of estimating the full distribution. Finally we will discuss an implementation using quantile regression and it's advantages over standard DQN.

2.3.1 The Distributional Perspective

In order to understand some of the theoretical results underpinning the use of distributional reinforcement learning we begin by revisiting the Bellman Equation for expected values in the context of Fixed Point Theory. We note the following definition, [35, Definition 2.1],

Definition 2.3.1. Let (X,d) be a metric space and let $f : X \to X$ be a mapping.

1. A point $x \in X$ is called a fixed point of f if $x = f(x)$.
2. f is called a *contraction* if there exists a fixed constant $h < 1$ such that,

$$d(f(x), f(y)) \leq h d(x,y), \text{ for all } x, y \in X.$$

Using these definitions we state the following theorem by Banach,

Theorem 2.3.2 (Banach Contraction Principle)**.** *Let (X,d) be a metric space, then each contraction map $f : X \to X$ has a unique fixed point.*

Proof of this theorem is omitted as it is not directly related to distributional reinforcement learning but is given in [35, page 35]. Banach's contraction principle gives us a powerful tool for proving whether repeated application of an operator leads to convergence. This allows us, in principle, to obtain a some theoretical guarantees as to the convergence of an update rule. It is important to note that Banach's contraction principle is not if and only if. Convergence is still possible even if the operator is not a contraction.

When considering the Bellman equation for expected values, given in Equation (2.3.1), we can define the *Bellman operator* $\mathcal{T}^\pi$ to be

$$\mathcal{T}^\pi Q(s,a) := \mathbb{E}[R(s,a)] + \gamma \mathbb{E}_{P,\pi} Q(s',a'). \tag{2.3.2}$$

The mapping $\mathcal{T}^\pi$ is a contraction mapping [11, page 2] and by Theorem 2.3.2 has a unique fixed point Q^π, the action value function under policy π. By repeated application of $\mathcal{T}$, Q converges to Q^π, as observed in Section 2.1.2.

We recall from Section 2.1.2 the random future returns after time t,

$$G(s_t,a_t) = \sum_{k=t}^{T} \gamma^{k-t} R(s_k,a_k),$$

where s_t and a_t denoting the current state and action have been added to emphasise the dependence of the future returns on both[2]. While not explicitly stated before, it should be clear that these future returns are random, and following [11], we now consider the full distribution of G[3]. We can generalise the Bellman Equation given in Equation (2.3.1) to the following recursive relationship for the distribution,

$$G(s,a) \overset{D}{=} R(s,a) + \gamma G(S', A'),$$

where D denotes equality in distribution. For an agent in state s taking action a, let $G_\pi(s,a)$ be the random future returns under policy π. The reward function $R \in \mathcal{R}$ is now viewed as a random variable. We define the transition operator $P^\pi : \mathcal{R} \to \mathcal{R}$ as

$$P^\pi G(s,a) \overset{D}{:=} G(S', A'),$$

where $S' \sim p(\cdot|s,a)$ and $A' \sim \pi(\cdot|S')$,. The distribution $p(\cdot|s,a)$ is simply the dynamics of the MDP which we defined in Equation (2.1.1) when introducing MDPs. Analogous to the policy evaluation operator for the expected action value, defined in Equation (2.3.2), we can define the distributional Bellman operator $\mathcal{T}^\pi$ to be,

$$\mathcal{T}^\pi G(s,a) \overset{D}{:=} R(s,a) + \gamma P^\pi G(s,a) \tag{2.3.3}$$

The distribution $\mathcal{T}^\pi G(s,a)$ comprises of three sources of randomness. The random reward, the randomness of the transition operator P^π and the randomness in the value of the next state.

We now require the following definition [11, page 3],

Definition 2.3.3 (Wasserstein Metric)**.** Let $F, G : \mathbb{R} \to [0,1]$ be two cumulative distributions. The p-Wasserstein metric can be defined as,

$$d_p(F,G) := \inf_{(U,V)} ||U - V||_p,$$

where the infimum is taken over all pairs of random variables (U,V) with marginal cumulative distributions F and G respectively. If $p < \infty$ this can be written as,

$$d_p(F,G) = \left(\int_0^1 |F^{-1}(u) - G^{-1}(u)|^p du \right)^{1/p}.$$

The Wasserstein Metric is a metric over value distributions [11, Lemma 2]. We can use this to define the maximal Wasserstein metric.

Definition 2.3.4 (Maximal Wasserstein Metric)**.** For two value distributions $G_1, G_2 \in \mathcal{R}$, if d_p denotes the p-Wasserstein metric then we can define the maximal Wasserstein metric to be

$$\bar{d}_p(G_1, G_2) := \sup_{s,a} d_p(G_1(s,a), G_2(s,a)).$$

We are now in a position to state the main result of this Section [11, Lemma 3], which is proved in Appendix A.

Lemma 2.3.5. *Let $\bar{d}_p$ denote the Maximal p-Wasserstein metric and $\mathcal{T}^\pi$ be the distributional Bellman operator defined previously. Then, $\mathcal{T}^\pi : \mathcal{Z} \to \mathcal{Z}$ is a contraction in $\bar{d}_p$*

By Theorem 2.3.2 we therefore have that $\mathcal{T}^\pi$ has a unique fixed point. From the definition of $\mathcal{T}^\pi$ it can be seen that the fixed point is G_π. If, by repeated application of $\mathcal{T}^\pi$, we obtain a sequence $\{G_k\}$, then assuming all moments are bounded we have that $G_k \to G_\pi$ in $\bar{d}_p$ as $k \to \infty$ for all $1 \leq p \leq \infty$.

[2]The fact that we have specified the time step t in this definition may at first seem problematic given we wish to define G for an arbitrary state and action. It is important to note however that the time step will always be part of the state therefore for a given state the time step is uniquely defined.

[3]We should note that some notation differs from [11] and [12] in order to maintain consistency with the notation used in Chapter 2, taken from [16]. In particular we denote the random future returns using G rather than Z and the state by S or s rather than X or x.

2.3.2 Quantile Regression

Distributional RL was originally implemented using an algorithm referred to as $C51$. While outperforming DQN this algorithm has various flaws. One particular problem was the fact that a fixed support had to be defined for the reward distribution to be projected onto. This causes particular problems in a trading environment where the rewards depend significantly on the state.

Instead of defining a fixed support and projecting the distribution onto it, an alternative implementation fixes a selection probabilities and models their location. This technique, referred to as *Quantile Regression DQN*, or QR-DQN was introduced later [12] and showed a substantial performance improvement. For the curious reader, an implementation of C51 has been provided in the included code but for the sake of brevity it will not be formally evaluated.

Quantile Regression is a method for estimating conditional quantile functions [36]. The τ^{th} quantile of a random variable X with cumulative distribution function F can be defined as

$$\inf\{x : F(x) \geq \tau\}. \tag{2.3.4}$$

Quantile regression aims to model the τ^{th} quantile of a quantity conditional on a set of covariates.

Consider a set of N quantiles $\{q_i\}_{i=1,\dots,N}$ of the value distribution, G. Each quantile q_i is defined by cumulative probability τ_i for $i = 1, ..., N$, as in the Equation (2.3.4). These cumulative probabilities are uniformly distributed across $(0, 1)$, so that $\tau_i = i/(N + 1)$ for $i = 1, .., N$. We model each quantile q_i using a model[4] $\psi_i : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$. We use ψ to denote the full set of quantile approximation functions, $\psi = \{\psi_i\}_{1 \leq i \leq N}$. This collection of quantiles can be used to approximately model the distribution of G. This approximation is referred to as a *quantile distribution* and is defined as

$$G_\psi(s, a) := \frac{1}{N} \sum_{i=1}^{N} \delta_{\psi_i(s,a)}, \tag{2.3.5}$$

where δ_z denotes a Dirac delta function at $z \in \mathbb{R}$. This translates to approximating the cumulative distribution for G by a step function as shown in Figure 2.3, adapted from [12].

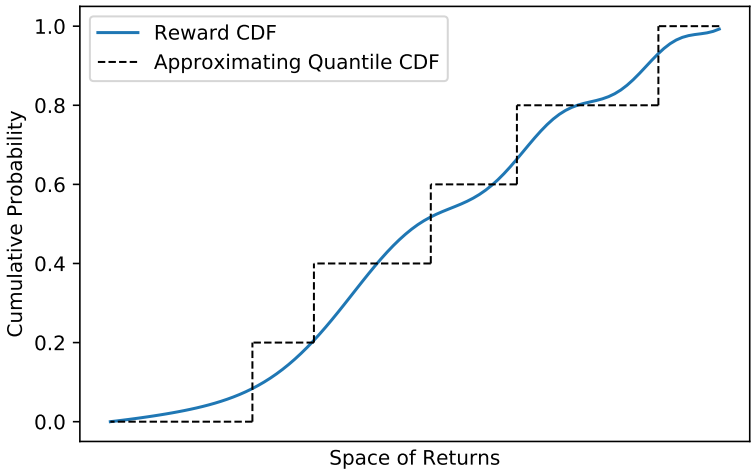


Figure 2.3: An example quantile distribution with $N = 4$. The blue line shows the CDF of the random future returns $G(s, a)$ for an arbitrary state action pair (s, a). The dotted line shows the CDF for the approximating quantile distribution. This approximation minimises the area between the blue curve and the dotted step function, the 1-Wasserstein error.

We now seek to find a projection $\Pi_{W_1} G : \mathcal{R} \to \mathcal{R}_Q$, mapping a value distribution onto the space of quantile distributions so that,

$$\Pi_{W_1} G := \underset{G_\theta \in \mathcal{R}_Q}{\arg\min}\, W_1(G, G_\theta), \tag{2.3.6}$$

[4]These are denoted using ψ_i rather than θ_i as in [12, 13] to avoid confusion with the Q-network weights.

where W_1 denotes the 1-Wasserstein metric from Definition 2.3.4. This is the mapping shown in Figure 2.3.

We can achieve this by minimising the *quantile regression loss* individually for each quantile approximation function ψ_i [12, Lemma 2]. This loss can be defined as follows,

Definition 2.3.6 (Quantile Regression Loss)**.** Let G be a distribution variable Given a quantile $\tau \in [0, 1]$, we can define the quantile regression loss for τ to be

$$\mathcal{L}_{QR}^{\tau}(\theta) := \mathbb{E}_{\hat{G}\sim G}[\rho_\tau(\hat{G} - \theta)],$$

where the quantile loss function ρ_τ is given by,

$$\rho_\tau(u) = u(\tau - \mathbb{1}\{u < 0\}),$$

for all $u \in \mathbb{R}$.

The quantile regression loss function is not smooth at 0. We therefore use the Quantile Huber Loss, shown in Figure 2.4. This is simply a modification of the quantile regression loss to be an asymmetric squared loss in a small region around 0, and the standard quantile regression loss outside of this region.

Definition 2.3.7 (Quantile Huber Loss)**.** Let $\kappa \in \mathbb{R}_+$, given a quantile $\tau \in [0, 1]$. We can define the quantile Huber Loss to be,

$$\rho_\tau^\kappa(u) := |\tau - \mathbb{1}\{u < 0\}| \frac{\mathcal{L}_\kappa(u)}{\kappa},$$

where $\mathcal{L}_\kappa(u)$ is the Huber loss function,

$$\mathcal{L}_\kappa(u) := \begin{cases} \frac{1}{2}u^2, & \text{if } |u| \leq \kappa, \\ \kappa(|u| - \frac{1}{2}\kappa), & \text{otherwise,} \end{cases}$$

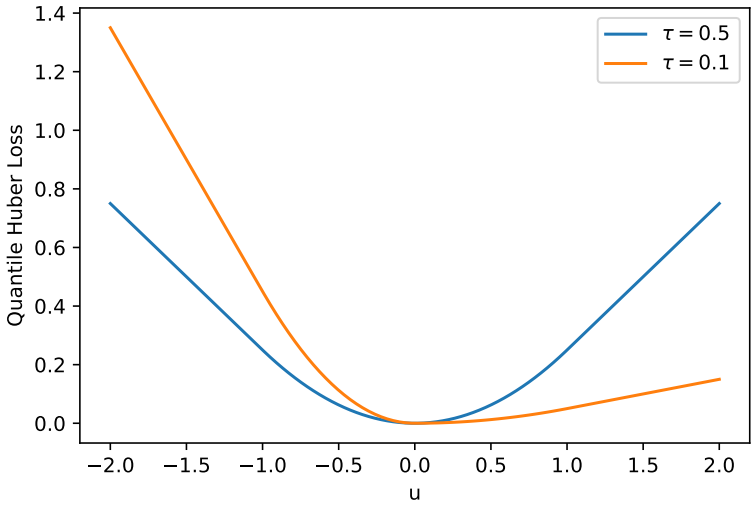


Figure 2.4: Quantile Huber Loss function with $\kappa = 1$ for two different values of τ. This is a modified version of the asymmetric quantile regression loss which is smooth at 0.

Using these concepts we are then able to show that the distributional Bellman operator is a contraction in the ∞-Wasserstein metric [12, Proposition 2].

Proposition 2.3.8. *Let Π_{W_1} be the quantile projection defined in Equation (2.3.6) and $G_1, G_2 \in \mathcal{R}$ two value distributions for an MDP with countable state and action spaces. If $\bar{d}_p$ denotes the maximal p-Wasserstein metric then,*

$$\bar{d}_\infty(\Pi_{W_1}\mathcal{T}^\pi G_1, \Pi_{W_1}\mathcal{T}^\pi G_2) \leq \gamma \bar{d}_\infty(G_1, G_2),$$

where $\mathcal{T}^\pi$ is the Bellman operator from Equation (2.3.3).

The proof of this proposition is quite involved and can be found in [12, pages 10-11]. By Proposition 2.3.8 we have that $\Pi_{W_1}\mathcal{T}^\pi$ is a contraction. Therefore, by Theorem 2.3.2 we have that the operator $\Pi_{W_1}\mathcal{T}^\pi$ has a unique fixed point G_π. If $\{G_k\}$ is a sequence of distributions generated by repeated application of this operator, we have that $G_k \to G_\pi$ as $k \to \infty$ in $\bar{d}_\infty$. Since $\bar{d}_\infty \geq \bar{d}_p$ for all $p \in [1, \infty]$ we have that $G_k \to G_\pi$ as $k \to \infty$ in $\bar{d}_p$ for all $p \in [0, \infty]$.

For each of the quantile functions ψ_i for $i = 1, ..., N$, we can train a Q-network to approximate it by performing gradient decent on a slight modification of Equation (2.2.3) from value based Deep Q Learning. Explicitly, we replace the value based loss function given in Equation (2.2.1) with the Quantile Huber Loss, from Definition 2.3.7, where the quantile τ varies between the quantile functions $\{\psi_i\}_{1 \leq i \leq N}$. Given an experience (s, a, s', r), for each quantile function ψ_i at policy evaluation iteration j we perform gradient decent on

$$\rho^\kappa_{\tau_i}(r + \gamma \max_{a'} Q(s', a'; \theta_{j-1}) - Q(s, a; \theta_j)),$$

where $\rho^\kappa_{\tau_i}$ is the Quantile Huber Loss from Definition 2.3.7 and τ_i is the quantile for the i^{th} quantile function ψ_i. We can of course continue to use the adaptions to DQN introduced in Section 2.2. Including both target networks and Double Deep Q, we would instead perform gradient decent on,

$$\rho^\kappa_{\tau_i}(r + \gamma Q(s', \arg\max_{a \in \mathcal{A}} Q(s', a; \theta_j); \theta^-) - Q(s, a; \theta_j)),$$

for each quantile function ψ_i, analogous to the single update for the expected value in Equation (2.2.5) for DQN. When implementing this in practise κ was set to 1 as suggested [12].

In practise, rather than use N Q-networks, we use a single network to approximate all of the quantiles. We can then either treat the quantile as an input to the network, or have a network with N outputs, one for each quantile [12]. We will discuss the exact network architecture further in Section 3. N is treated as an additional parameter and it's value will be listed when evaluating. An alternative implementation to this is to allow the Q-network to approximate the quantile function in a continuous way, referred to as an *Implicit Quantile Network* [13], however this will not be discussed further here.

We can define the set of optimal value distributions to be the value distributions of all optimal policies $\{G^{\pi^*} : \pi \in \Pi^*\}$, where Π^* denotes the set of optimal policies. While QR-DQN is used to model the full distribution of the future returns, the optimal policy still relates to the maximum expected return. The expected value for the quantile distribution defined in Equation (2.3.5) is simply,

$$\mathbb{E}[G_\psi(s, a)] = \frac{1}{N} \sum_{i=1}^{N} \psi_i(s, a), \tag{2.3.7}$$

assuming, as in the original definition, that the quantiles are uniformly distributed. The basic algorithm for QR-DQN, omitting the improvements discussed for DQN for simplicity, is therefore,

Algorithm 3: Base algorithm for QR-DQN.

At each training period, given mini batch $\{(s_j, a_j, r_j, s'_j)\}_{j=1,..,B}$:
for $j = 1, ..., B$ **do**
  Set $Q(s_j, a_j) := 1/N \sum_{i=1}^{N} \psi_i(s_j, a_j)$
  Set $a* := \arg\max_{a' \in \mathcal{A}(s'_j)} Q(s'_j, a')$
  for *i=1,..,N* **do**
    Set $\mathcal{T}\psi_i = \begin{cases} r_j & \text{if } s'_j \text{ is terminal,} \\ r_j + \gamma \psi_i(s'_j, a^*) & \text{otherwise} \end{cases}$
    Perform gradient decent on $\rho^\kappa_{\tau_i}(\mathcal{T}\psi_i - \psi_i(s_j, a_j))$
  end
end

2.3.3 Why Use Distributional Reinforcement Learning?

There are several factors motivating distributional approximation rather than value approximation. The use of distributional reinforcement learning has been shown to result in impressive performance gains over equivalent value based reinforcement learning algorithms [12, 11, 13]. In fact, the Rainbow algorithm, designed to showcase improvements to the vanilla DQN algorithm showed distributional reinforcement learning to be one of the modifications that resulted in the greatest performance increases[5] [8].

The reason for distributional reinforcement learning's superior performance is still not well understood. In the tabular setting, or when using a linear function approximator, distributional reinforcement learning has been shown to perform identically to value based reinforcement learning [37]. It has therefore been hypothesised that modelling the distribution leads to improved performance when coupled with deep neural networks. In particular, it has been proposed that semi gradient updates to the distribution of returns results in a different expected value than if an equivalent update were applied to the expected value directly [37].

The performance improvements shown by distributional reinforcement learning have generally been demonstrated using the ALE [34]. Given most of the Atari games have few, if any, random elements, in a trading environment it is conceivable that many of these benefits could be amplified. Returns from trading are always stochastic in contrast to the predominantly deterministic outcomes of the Atari games.

Alongside the potential for an improved policy there are inherent advantages of estimating the distribution. Implicit in many reinforcement learning solutions to optimal execution is the assumption that the agent wishes to maximise their expected return. A risk averse trader may have a preference for a policy with lower variance in the returns, even at the cost of a lower expected return. Distributional reinforcement learning raises the prospect of an agent that could make decisions based on the approximate distribution of the returns, for example deviating from a baseline policy only if the return will be higher with some large probability.

Even if this information is not used to modify the trading strategy directly, it could still allow the traders to make informed decisions when using reinforcement learning solutions for optimal execution. When liquidating a position, the distribution of the expected returns could be inferred from historical results however this is backwards looking. A distributional reinforcement learning agent could in theory give forward looking insight into the approximate distribution of the returns based on the current state of the market. This would allow for decisions at a macro level surrounding whether to liquidate now or wait, or even whether to use an alternative strategy.

Finally, although distributional reinforcement learning aims to model the intrinsic uncertainty in the environment, a by product of this is that we capture some measure of parametric uncertainty. Given the distribution of the returns for a certain action, we can assign a degree of certainty to its value. This information can be used to explore the state space more efficiently and is the topic of the next section.

2.3.4 Efficient Exploration

When using Q-learning we require a behaviour policy that is sufficiently exploratory. This was discussed in Section 2.1.4 and we introduced the ϵ-greedy strategy. One criticism of ϵ-greedy strategies is the fact that they explore at random. With probability ϵ, an action is selected uniformly at random, regardless of how frequently it has been chosen prior to this, or its estimated value.

Distributional reinforcement learning allows for the approximate distribution to be used to make a more informed choice when choosing an exploratory action. In short, rather than choosing a random action with probability ϵ, actions are chosen based on their estimated value and variance. This technique is similar to Upper Confidence Bound strategies for stationary tabular problems, however it was recently introduced in the distributional reinforcement learning setting [38].

When discussing distributional reinforcement learning we make the distinction between *parametric* and *intrinsic* uncertainty. Intrinsic uncertainty refers to the randomness in the underlying environment. In a trading environment, however many parameters we include in our model, there will always be stochasticity in the evolution of the price of the underlying asset. Conversely, parametric uncertainty refers to the uncertainty due to the uncertainty in the parameters of the model.

[5] In this case the C51 algorithm [11] was used however the performance of C51 was later surpassed by QR-DQN[12]

Distributional reinforcement learning aims to model the intrinsic uncertainty of the MDP [11]. This occurs in the limit, as the number of training episodes on independent samples tends to infinity. For a finite number of training episodes, the model also captures parametric uncertainty. While it is difficult to differentiate between the intrinsic and parametric uncertainty for a model, the parametric uncertainty decreases over time as the model converges to the true reward distribution.

From quantile regression theory, the parametric uncertainty decays at a rate,

$$c_t = c\sqrt{\frac{\log t}{t}},$$

where c is a constant factor [38]. We wish to prioritise actions with greater estimated value and those with greater parametric uncertainty. For a given state $s \in \mathcal{S}$, let $\hat{\mu}_{s,a}$ and $\hat{\sigma^2}_{s,a}$ denote the mean and variance of the expected returns if action $a \in \mathcal{A}(s)$ is selected. We can obtain both these estimates using the quantile distribution introduced in Equation (2.3.5). The mean $\hat{\mu}_{s,a}$, is given in Equation (2.3.7), and analogously the variance can be estimated as,

$$\hat{\sigma}^2_{s,a} = \frac{1}{N-1}\sum_{i=1}^{N}(\psi_i(s,a) - \hat{\mu}_{s,a})^2,$$

where $\psi_i(s,a)$ denotes the i^{th} quantile function for action a and state s. We introduce the following deterministic exploratory policy [38],

$$\pi_E(s) := \underset{a' \in \mathcal{A}(s)}{\arg\max}(\hat{\mu}_{s,a'} + c_t\hat{\sigma}_{s,a'})$$

for any state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$. As the parametric uncertainty decreases, so does the exploration bonus given by $c_t\hat{\sigma}_{s,a}$. Using this exploratory policy as the behaviour policy for QR-DQN has been shown to dramatically increase the speed at which the agent learns [38].

Chapter 3

Deep Execution

The discussion surrounding the potential for reinforcement learning in optimal execution is not a new one. Early studies showed promise with the use of Dynamic Programming and Tabular Q Learning [2, 3] to acquire or liquidate large positions in an asset using either limit orders or market orders. Alongside pure reinforcement learning solutions, the use of reinforcement learning to enhance the theoretical models of Almgren-Chriss, discussed in Section 1.2 have also been proposed [4]. Recently, deep reinforcement learning has been applied to the optimal execution problem [9, 10]. These studies have demonstrated some success for agents executing using only market orders on both simulated and historical data. In the following chapter we will attempt to apply the deep reinforcement learning agents discussed in Chapter 2 to the problem of optimal execution for an agent using a combination of market and limit orders.

The trading environment is modelled as an MDP. While it is unlikely that the environment we will construct is truly Markovian we treat it as approximately one. To quote Nevmyvaka et al. *'we will essentially be treating a partially observable environment as if it were fully observable to us'* [3].

The problem of optimal execution is episodic. The agent must exit a position in an asset in a defined time window. The number of time steps is therefore finite and $t \in [0, T]$, where T is the terminal time as in Section 1.2. Given the problem is episodic we set the discount factor $\gamma = 1$. This differs from the approach taken in some other implementations [9, 10]. The states that the agent encounters represent all observable information available to the agent. There is a natural divide here between private and public variables. Private to the agent are the remaining position, elapsed time and, if using limit orders, the currently unexecuted limit orders that the agent has placed. Public variables are relevant observable information from the market, or wider environment. This could include the stock price, historical stock prices and order book data but need not be limited to market data. There is potential to include external factors such as the price of correlated assets or breaking news that the agent may wish to condition on when choosing an action.

The action space has the potential to cover a range of possible order sizes, types and directions. If using limit orders then it could also include the cancelling of any unexecuted limit orders. Most of these actions are essentially continuous; the number of shares, or an amount of currency are technically discrete but the units are sufficiently small that we can treat them as continuous quantities. Reinforcement learning algorithms for continuous action spaces are available [39], in particular *Proximal Policy Optimisation* algorithms [40] have been shown to perform well for the problem of optimal execution [10]. DQN and QR-DQN require discrete action spaces and we choose a finite set of actions representing different numbers of units of the asset to sell at each time step.

We wish to structure the agent in such a way that it is distinct from its environment to as great an extent as possible. This enables the same agent to be used across multiple environments and means that the agent could be transitioned to a real world trading environment with minimal modification. We will therefore discuss the design of the environments and agents separately.

3.1 Difficulties in Applying Reinforcement Learning to Execution

WHile reinforcement learning algorithms have achieved a superhuman level of performance on a number games [24, 25, 5], there are several difficulties in generalising these techniques to real world environments. A fundamental list of problems is discussed in [41]. Of these we will discuss a few key challenges that relate to the field of optimal execution in particular. Many of these issues have not been satisfactorily solved, however where relevant to the trading environment we will suggest specific solutions.

The results discussed in the Atari papers have been achieved using simulated environments. The availability of data is limited only by the training time and computational power available. The strong performance achieved in the Arcade Learning Environment is the result of training on tens to hundreds of millions of frames. Since agents could act every three or four frames this results in the number of agent interactions with the environment being in the order of tens of millions. This is many orders of magnitude larger than the volume of available training data for an optimal execution algorithm. The setup used in [9] looked at three, hour long, trading periods per day for a year. Even if each hour were treated identically, this would be equivalent to just 750 episodes available for training and a maximum of 7500 agent interactions with the environment.

The lack of a simulated environment raises a second issue. Optimal execution assumes relatively large orders. Any reinforcement learning agent trained without a model would rely on interactions with the real market. This would be a costly experiment given the volume of data required to train an agent to a satisfactory standard.

Financial markets are not directly comparable with games where reinforcement learning has achieved success. The Atari games are primarily deterministic, stationary problems, although they do involve a small stochastic element [11]. Financial markets on the other hand are stochastic and generally regarded to be highly non stationary [42].

As with most machine learning solutions, reinforcement learning algorithms do not make it easy to infer the rationale behind their decisions. This raises problems from an accountability perspective and makes them difficult to audit. This can be a particular problem when they perform in an unexpected manner. Reinforcement learning models are only as good as the data they have been trained on and a sudden regime change or unpredictable event may result in adverse performance.

Finally, changes to the trading rate must be made in close to real time. In the simulated trading environments discussed in this chapter there is no restriction on the time required to make decisions. This is not true in a real trading environment where any lag between the arrival of information and output of a decision can be costly.

We will address some of these problems in turn. Firstly the reinforcement learning agents for the Atari environments obtain scores which are, on average, multiple times better than those achieved by human experts. For the following, our aim is more modest: to beat a benchmark strategy TWAP. This has been shown in some scenarios to be achievable even with limited market data [9]. Further, it has been shown that if training data is constrained, greater data efficiency is possible when when training DQN agents by tweaking the agent parameters [30]. Specifically, this can be achieved by increasing both the batch size B, and the number of steps n to look ahead when replaying experiences from the replay buffer. We can also make more efficient use of the data by simplifying the inputs. While the Atari papers used raw images as the model input, we will instead use preprocessed features. This is carried out with the aim of reducing the time taken to train models to an acceptable standard.

We can partially resolve the second issue by using historical data, modified to simulate the effect of the agents actions on the market. While this is an imperfect solution, many of the theoretical execution models used in trading, TWAP and VWAP for example, have been derived based on theoretical model for market impact.

This is not the only solution. One important advantage of off policy algorithms is that they can be trained using trajectories generated under a different policy. In theory they could be trained using trajectories generated by different agents. This is known as offline learning. If this could be achieved then the experience of human traders or production algorithms could be leveraged to speed up the learning process. This approach would minimise or completely avoid the use of a simulated environment and allow the agent to learn the true market impact of it's actions. In practise there are several ways of achieving this aim, one promising solution is *Deep-Q learning*

from Demonstrations [43].

In contrast to many real world applications for reinforcement learning algorithms, it is relatively easy to impose safety restrictions for trading. The action spaces we will consider will be heavily restricted, and we will consider only a small subset of the possible actions an agent could feasibly choose. Both the limit order and market order sizes will be restricted to a maximum and minimum.

Finally we note that any reinforcement learning solution must be low latency. This is not a problem limited to reinforcement learning models. The latency in algorithmic trading models is a problem regardless of the model and may mean that the results achieved in simulated environments are not entirely realistic [44]. This latency is not something we will attempt to model but should be borne in mind when evaluating the results in Chapter 4.

3.2 Agent Design

As in the theoretical setting we denote the length of the trading period as T. Practically, in a real trading environment, we cannot trade continuously. Instead we choose to trade every Δt seconds, where Δt depends on the asset being traded.

We will consider two possible state spaces, $\mathcal{S}$. The first will have a state consisting of the current inventory, Q, of the agent and the elapsed time, t. The second will additionally include the values of various market variables at the current time step and n previous time steps. As noted in Section 2.2.2, it is important that the neural net inputs are appropriately standardised. Both the inventory and time elapsed are mapped to be between -1 and 1. The current and historical market data inputs will be standardised by the subtracting the mean and dividing by the standard deviation of a small sample taken at random from the full set of training data.

In Section 2 we assumed that the value of each action was taken to be an independent output of the Q-network. This approach is well suited to the discrete action space of the Atari Learning Environment [34]. An alternative approach [9] is instead to use both the state and action as an input to the network. Both approaches were trialled. While both were found to produce similar results, the latter approach is more attractive. By treating the actions as an input we are able to reduce the dimensionality of the output when considering an agent using both limit orders and market orders. Since the actions are now inputs to the neural network they were scaled to be in the range $[-1, 1]$ with the actions representing the smallest and largest market orders taking the values of -1 and 1 respectively.

While we will evaluate the performance of several types of agent for comparison purposes, we will predominantly be using the QR-DQN Agent discussed in Section 2.3.2. The full architecture for the Q-network used for this agent is shown in Figure 3.1. The market data inputs for the previous n time steps are first preprocessed using L_1 convolutional layers of U_1 nodes. This network is included with the aim of extracting some sort of signal allowing the agent to modify its speed of trading to reflect the current state of the market.

This preprocessed input is then fed into the main network consisting of L_2 fully connected layers of U_2 nodes along with the 'private' components of the state, the remaining position and elapsed time, and the action representing the size of the next market order. The outputs are the predictions for the N quantiles of the returns distribution given the current state and action.

The treatment of the outputs of network used to preprocess the market data inputs is an important consideration. In particular, how should the preprocessed market data be combined with the actions and private state inputs? One option would be simply to concatenate these inputs with the outputs of the convolutional net. This was found to lead to poor performance. In order to force interaction, the output of the convolutional net was instead multiplied with the remaining inputs using the element-wise Hadamard product. This method was inspired by a similar solution used in the design of Implicit Quantile Networks[1] [13].

When training the network we use the Adam [45] optimiser. This has been shown to perform well when training both DQN and QR-DQN Agents [8, 12]. All parameters for Adam are set to their defaults apart from the learning rate ν which we will specify when evaluating the agents in Chapter 4.

[1] Implicit Quantile Networks are an alternative to QR-DQN for distributional reinforcement learning [13]. They are not used in this work

3.2.1 Action Space Using Only Market Orders

We will initially discuss an agent which may only trade using market orders. Following [9, 10] we choose a discrete, relatively small action space $\mathcal{A}$. We restrict the agent to exclusively selling the asset and allow it to choose between a small number of *lots* of the asset. Each lot consists of multiple units of the asset, where the lot size depends on the size of the position, R, to liquidate. At each time step $t = i\Delta t$, for some integer $i \in [0, \lfloor T/\Delta t \rfloor]$, the agent chooses the number of lots to sell via a market order. We note that this differs from the implementation discussed in [9]. The agent decides on the size of the order at each step rather than setting a constant trading rate for the next n time steps. In other words the trading frequency and decision frequency are one and the same.

3.2.2 Action Space Using Limit and Market Orders

For the agent using both limit and market orders we expand the action space. As we will continue to only consider an agent exiting a position we will only allow the agent to make sell market orders or ask side limit orders. To simplify the agent we will considerably restrict the action space as follows:

1. Limit orders may only be placed at the best ask price.
2. Limit orders are capped at 4× TWAP trade size
3. Limit orders may not be cancelled voluntarily by the agent
4. If best ask price moves to be more competitive than the agent's limit orders then all of the agent's limit orders are involuntarily cancelled.

With the exception of Restriction 2, each of these limitations has been chosen to simplify the agent design but it is feasible that the agent would perform better if some of these were relaxed. We limit the action space to a small number of combinations of limit order and market order sizes. The exact choice will be discussed in Section 4.

3.2.3 Rewards

The reward structure for a reinforcement learning agent is an important feature; if poorly defined it can lead to an agent taking unintended shortcuts. When defining the reward structure we must do so in a way which makes it easy for the agent to learn the optimal solution. Care must be taken to do so in an unbiased way, specifically by avoiding using a structure which incorporates unrealistic prior knowledge of the environment. In Section 4.1 we will test whether a reinforcement learning agent converges to the optimal solution, TWAP, in a simulated market. A facetious choice of reward might be,

$$r_t = \mathbb{1}\{a_t = a^*\},$$

where a^* denotes the action corresponding to TWAP. There is no doubt that the agent would quickly converge to the correct strategy but we would gain no confidence about the convergence to an optimal strategy when it differs from TWAP. More subtly, a reward standardised using the returns under TWAP would be equally flawed as it assumes knowledge of the market impact functions that we would not have in a real trading environment.

For the problem of optimal liquidation, we wish to maximise the cash received from liquidating a position over a defined time window. A natural reward might therefore be the cash received from a market order or alternatively the execution cost of the order, as a negative number.

Since the stock price is a stochastic process these rewards will vary at random, regardless of the agent's actions. One implementation of an execution cost reward structure would be to define the reward at time t as

$$r_{t+1} = c_{t+1} - a_t S_t,$$

where ${c_{t+1}}^2$ is the cash received as a result of a market order of volume ${a_t}^3$ being placed when

[2]Recall that it is standard practise to use the subscript $t+1$ for the reward received due to an action at time t. The same logic has been applied to the cash received for a market order at time t.

[3]We use a to denote the volume of the market order as a describes the action taken.

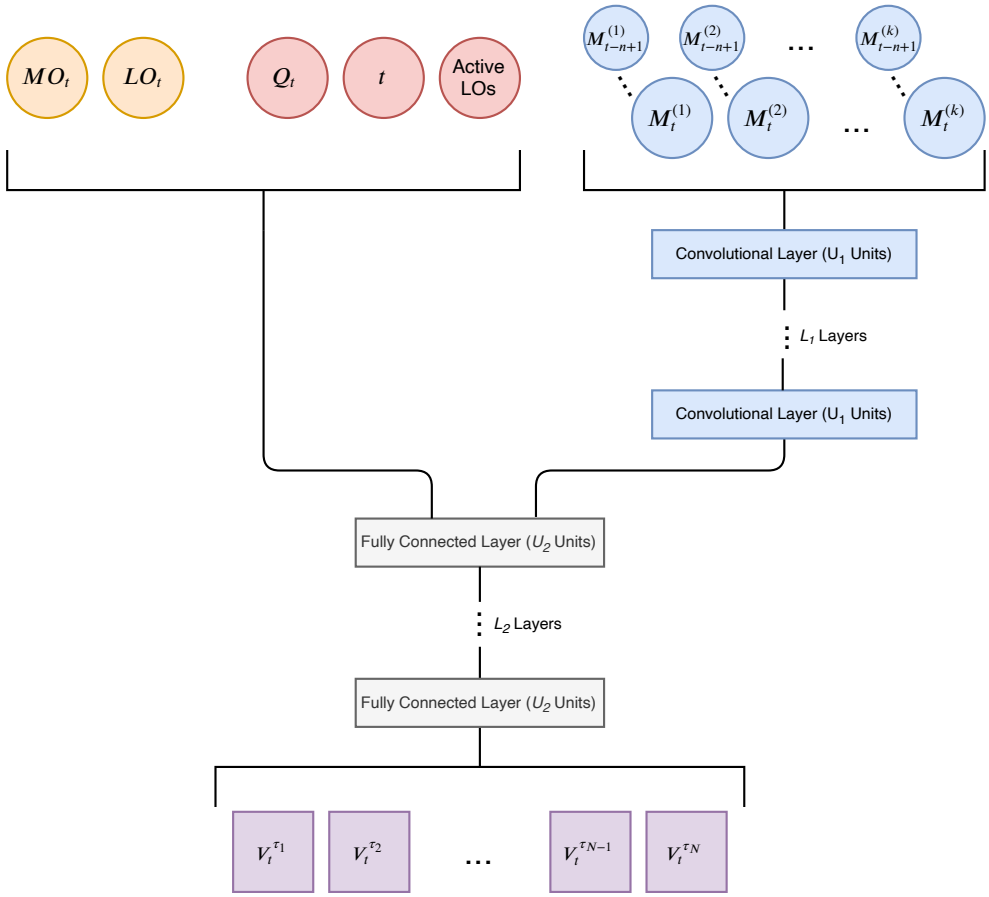


Figure 3.1: Full architecture for the Q-network of the QR-DQN agent. Circles represent inputs and squares outputs. Shown in orange are the action inputs to the network. In red are the private agent inputs, the inventory, elapsed time and volume of unexecuted limit orders posted by the agent. In blue, are a selection of k market variables. For each market variable the current value is provided along with the n previous values. This set of market variables is preprocessed using a L_1 layer convolutional neural network of U_1 units. The output is then flattened, combined with the other inputs and fed into a L_2 layer neural network of U_2 units. This produces the final output, the estimated value of the state and actions for N quantiles.

the mid price of the stock was S_t. This approach would go some way to mitigating the affect of random fluctuations in the price of the underlying asset.

This approach is flawed. Consider an agent chooses a sequence of actions that make other market participants aware that it is attempting to sell a large position. The stock price would drop and later market orders would be filled at a worse price as a result. The reward, however, would be similar to that received if the agent had traded more discretely due to it being standardised by the current mid price.

We instead choose to define the reward as,

$$r_{t+1} = c_{t+1} - a_t S_0,$$

calculating the execution cost using the mid price at the beginning of the episode.

A second consideration is the treatment of any remaining position at the end of the trading period. We wish the agent to fully exit the position by time T. There are several approaches to achieving this goal. One was would be to modify the reward structure so as to heavily penalise an agent for any remaining position in the asset at $t = T$. This approach could be advantageous if the end of the period T is not a hard deadline. The agent would then learn to continue trading beyond T, but only if penalty for trading past T was fully compensated by the additional return.

An alternative choice, taken here, is to place a market order to sell any remaining position at the final time step. This is irrespective of the action chosen by the agent. This approach is similar to that taken in other papers [9, 10].

When introducing Artificial Neural Networks in Section 2.2.2, we stated that their performance is improved when the target for the network is standardised. This presents a few problems as we need a realistic estimate for the average reward the agent can achieve and the variance in the rewards over the episode. As previously discussed we need to infer appropriate quantities systematically by analysing the data rather than using our knowledge of the parameters used in the simulated environment. We can do this by initially training an agent with little reward scaling. The mean and standard deviation of the rewards for the trained agent can then used to approximate the mean rewards for a general agent.

3.3 Environment Design

3.3.1 Simulated Market Environments

The first environment we shall consider is a simulated environment, similar to the Almgren Chriss environment introduced in Section 1.2. The advantage of a simulated environment is twofold. When training an agent in real time or using historical data, the number of episodes we can use for training is limited by the availability of data. When using simulated data the number of training episodes is limited only by the computational power and training time available. Secondly, we can evaluate how the agent performs in a controllable environment. In the cases where we can derive an optimal solution we can see how quickly, if ever, the agent converges to this solution. Low signal to noise ratios are a persistent problem in trading. By varying the volatility of the underlying asset, the agent's performance can be evaluated in relation to varying noise. This allows for the development of a robust solution before training the agent on real market data.

For the simulated environment we generate a midprice path by assuming the asset price follows a simple geometric Brownian motion, as shown in Equation (1.2.1). The solution to this SDE is well known and the derivation can be found in textbooks [46]. At each time step we can sample the stock midprice as

$$S_t := S_{t-1} \exp\left\{\left(g\left(\frac{V_t}{\Delta t}\right) - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t} Z_t\right\},$$

where the permanent market impact $g(v) = bv$, for some parameter b and $Z_t \sim N(0,1)$ is sampled from a standard normal distribution. Let us assume that at time t a market sell order of size V_t is received. We assume that V_t is always small enough that there is sufficient liquidity and the cash C_t received from the sell order is calculated as,

$$C_t := \left(S_t - f\left(\frac{V_t}{\Delta t}\right)\right) V_t, \tag{3.3.1}$$

where $f(\cdot)$ is a function modelling the temporary market impact. This is identical to Section 1.2 but adjusted for discrete rather than continuous time. We will take $f(\cdot)$ to be $f(x) = kx$, for some parameter k.

3.3.2 Historical Market Environments

We are able to create a more realistic environment using historical data. While we can obtain historical price paths and order book data, we still need to model the affect of the agents actions on the market. This is achieved by using the temporary market impact function $f(\cdot)$ from the previous section. For simplicity we choose not to incorporate permanent market impact. For market orders the implementation is relatively straightforward. The cash received for a market order is given in Equation (3.3.1), where S_t is the stock price obtained from historical data.

Modelling the limit order book introduces an additional layer of complexity. We seek to build a realistic model of the limit order book using historical data, modelling the affect of the agent's actions on the market. As with market orders, certain assumptions must be made about the responses of market participants to the actions of our agent and the affect of market variables which are not observable in historical data. When building an environment to train and evaluate an agent we wish to minimise the affect of any assumptions on the performance of the agent, erring towards conservative assumptions where possible.

To simplify the environment we consider only the top of book. As with market orders we allow limit orders to be placed every Δt seconds. To simulate the limit order book we require the following market variables for each time $t = i\Delta t$, where $i \in [0, T/\Delta t]$[4]:

- Best bid price B_t
- Best bid size y_t^B
- Best ask price A_t
- Best ask size y_t^A
- Total volume buy market orders filled during the previous time interval Δt at price x, $M_t(x)$

While it may seem unintuitive to use M_t to denote the market orders for the previous time interval, all the variables above are measurable at time t. At time t, the market orders for the following time interval are not measurable. The first four limit order book variables are updated at the last tick of the previous time interval. The final variable, the volume of market orders, can be calculated as the sum of the volume of all market orders filled between $t - \Delta t$ and t.

In order to model the limit order book environment using historical data we make the following assumptions:

1. Limit orders are only cancelled at the back of the queue, ie. a *last in first out* or LIFO policy.
2. Limit orders are processed instantaneously.
3. Limit orders placed by the agent are small enough to have no effect on the actions of other market participants.
4. If the best bid and ask prices cross and only the agent offers the best ask price then the full volume of the agent's limit orders at this price are filled.

We discuss each of these assumptions in turn. One property of limit order books is that we are not able to directly observe the individual participant orders. Only the total volume available at each price is observable. We are therefore unable to accurately calculate the new position of the agent's limit order in the queue after it has been placed. Instead we can calculate the net volume of ask side limit orders which have been placed or cancelled over the previous second as

$$y_t^A - M_t(A_t) - y_{t-1}^A,$$

assuming that $A_t = A_{t-1}$. Assumption 1 states that any cancelled limit orders are those at the back of the queue thereby minimising the affect on the position of the position of the agent's limit

Figure 3.2: Example to demonstrate the treatment of cancelled limit orders. The plot shows the development of the limit order book on a Bitcoin market between $02:40:19$ and $02:40:20$. Each bar shows the volume available at the best ask price, 7795.5. The agent has two limit orders active, at different positions in the queue. At $02:40:19$ there are $608,083$ units available and at $02:40:19$ there are $508,855$ units. Given the volume of market orders over the one second interval was $62,684$, the remainder of the difference must be cancelled limit orders. Assumption 1 states that all cancelled limit orders were at the back of the queue, excluding the limit orders of the agent.

orders. If the faster execution of the agent's limit orders is beneficial then this is a conservative assumption.

Assumptions 2 and 3 are more straightforward. When trading the agent will inevitably face a small delay between the observation of the market state and the marketplace receiving the agent's limit order or market order instruction. To simplify the environment we ignore this delay. Additionally we do not attempt to model the effect that the agent's trading has on the decisions of other market participants. By restricting the maximum size of limit orders which can be posted we aim to minimise this effect.

The final assumption states that if the bid and ask prices cross then all the agents limit orders at that price are filled. We do not judge this to be an unrealistic assumption and it serves to keep the environment from deviating too greatly from the historical data.

[4]For each of these variables, the subscript is commonly used in the literature to refer to the depth of the limit order book. Since we are considering only the top of book we have used it here to denote the time.

Chapter 4

Training and Evaluation

In the following chapter we will evaluate the agents and environments introduced in Chapter 3. In each case, performance will be evaluated regularly during training, and plotted.

4.1 Simulated Data

In Section 3.3.1 we outlined a simulated environment in which we could train and test reinforcement learning agents for optimal execution. In this environment the optimal solution for minimising execution costs is TWAP, and we can measure the success of an agent by whether it converges to this strategy. For this environment we consider an agent trading 10 times over the course of an hour. While this is perhaps an unrealistically low trading rate, it is effectively equivalent to the agent choosing a trading rate 10 times over the course of an hour and has been chosen to mirror the setup of similar studies for comparison purposes [10, 9].

4.1.1 Comparison of Agents

The values used for the parameters discussed in Section 3.3.1 are given in Table 4.1. For this first study the parameters have been chosen to mirror the agent comparison of [10, Environment 1]. The size of position has been set to 1 for simplicity and the market impact parameters adjusted to reflect this. The units do not matter in this case since we consider only an unspecified large position.

Parameter	Description	Env 1
k	Temporary Market Impact	0.0186
b	Permanent Market Impact	0
T	Period Length	10
R	Initial Position	1
σ	Stock Volatility (daily)	0.0083
S_0	Initial Price	1

Table 4.1: Parameters for the simulated execution environment used to compare the agents.

The action space for each agent consists of 11 actions. Given at this stage we are not specifying a precise quantity of the asset to liquidate, we instead refer to the actions as a choice of what proportion of the original position to liquidate. Unlike other implementations [9], the action space is independent of the state. The action space is a uniform partition of the interval $[0.05, 0.15]$. Since $T = 10$, liquidating 0.1 at each time step corresponds to TWAP. The action space therefore ranges from 0.5 to 1.5 times the TWAP trading speed.

Two agents were tested using Environment 1. A DDQN Agent, introduced in Section 2.2 and a QR-DQN Agent from Section 2.3. The DDQN Agent uses an ϵ-greedy behaviour policy and the QR-DQN Agent the efficient exploration policy detailed in Section 2.3.4.

The parameters used for each agent are shown in Table 4.2. The number of quantities N, the learning rate ν and the batch size, B, were chosen to be identical to those used for the original QR-DQN Agent [12]. Agent performance for the QR-DQN Agent was in fact particularly

sensitive to the choice of learning rate for the Adam optimiser ν however this value resulted in good performance. Due to the smaller training period, the target lag was changed from the original implementation of DQN [7]. A sweep was conducted over a small range of values and $C = 50$ resulted in the best performance. While much smaller than the value used when evaluating agents in the ALE [24, 7], this is significantly larger than the value of 14 chosen by Ning et al. when training RL agents using historical data [9]. Similarly a sweep was conducted for the tree horizon, the epsilon decay and the exploratory coefficient c for the behaviour policy described in Section 2.3.4.

Parameter	Description	DDQN	QR-DQN
ϵ_d	Epsilon decay	0.9992	N/A
ϵ_m	Epsilon minimum	0.02	N/A
c	Exploratory Coefficient for π_E	N/A	150
C	Target Lag	50	50
B	Batch Size	32	32
	Buffer Size	3000	3000
n	Tree Horizon	4	4
ν	Adam Learning Rate	0.00005	0.00005
N	Number of Quantiles	N/A	200
	Reward Scaling Mean	0.98	0.98
	Reward Scaling Standard Deviation	0.01	0.01

Table 4.2: Parameters used for the DDQN and QR-DQN Agents evaluated on the simulated environment

The Q-network architecture for each agent is summarised in Table 4.3. A sweep was conducted to find the optimum number of layers and units per layer. No architecture resulted in dramatic performance improvements.

Parameter	Description	Value
$\mathcal{S}$	State Space	$[t,Q]$
L_1	Number of Convolutional Layers	0
L_2	Number of Fully Connected Layers	3
U_2	Number of Fully Connected Units	27

Table 4.3: Parameters for the neural network architecture, shown in Figure 3.1, for both agents evaluated in the simulated market environment.

Fifteen instances of each agent were trained over $200,000$ episodes, resulting in close to 2 million interactions with the environment for each agent. Every 500 episodes, training was paused and a greedy version of the agent was evaluated on 25 episodes. While 25 episodes per evaluation period may seem low, it is important to note that for this environment the state space consists of only the time elapsed and the agent's remaining position. The mapping between states and actions is therefore deterministic, and the actions taken will be identical between episodes assuming that no training has taken place. The average cash received and the count of optimal actions per episode from liquidating the position over the evaluation period for each agent was logged and the results shown in Figure 4.1.

The performance of the DDQN agent is similar to the results obtained by previous studies of simulated environments [10]. The average reward increases consistently but does not converge to TWAP in the time frame considered. While the average reward appears to still be increasing slowly at the end of the training period, it is uncertain whether it would ever converge fully and in a real world environment with limited training data, the volume required to do so may be impractically large. In addition the algorithm exhibits *plummeting* behaviour, noted in previous studies [47]. This refers to the large and abrupt degradation in performance at certain points during training.

The returns received for actions close to TWAP become difficult to distinguish from the returns under TWAP due to the stochasticity in the rewards. It is therefore easier to see convergence by looking at the average number of times the optimal action was chosen over the evaluation period. The DDQN agent shows only small performance improvements after around $20,000$ training episodes. It then chooses the optimal action around 6 times throughout the episode. We can see

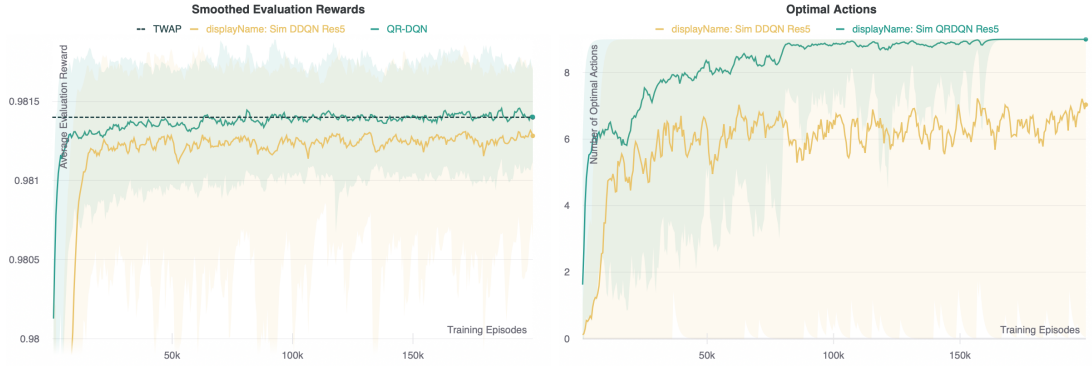


Figure 4.1: The smoothed cash received from liquidating the position (left) and optimal action count (right) over a 25 episode evaluation period plotted against the number of episodes of training for each agent. The average cash received using a TWAP policy is shown by the dashed line. For each agent, the bold line shows the mean and the shaded area shows the range of the minimum and maximum across all instances of the agent. The number time steps where an action can be selected is 9 as the final action is always to liquidate any remaining position.

from the yellow shaded area however, that the range in the number of times the optimal action was chosen per episode varies from 0 to 9 throughout the entire training period.

The QR-DQN Agent performs significantly better, learning faster and earning a higher average reward than DDQN in every evaluation period. All instances of the QR-DQN agent had converged to TWAP by episode $150,000$.

4.1.2 The Quantile Regression Agent

From this point forward we will consider only the QR-DQN Agent. In the previous section we chose to use the exploratory policy from Section 2.3.4 as the behaviour policy for the QR-DQN agent. To justify this choice, Figure 4.2 shows the first $100,000$ training episodes for 30 QR-DQN Agents, 15 using an ϵ-greedy behaviour policy, the remainder the efficient exploration behaviour policy. The agents are in all other respects identical and use the same parameters given in Table 4.2. It can be seen that while the performance of the two agents is similar for the initial episodes, the QR-DQN agent using the efficient exploration behaviour policy begins to outperform by $20,000$ training episodes. As with the previous comparison it is easier to see this by looking at the count of optimal actions rather than the volatile average rewards.

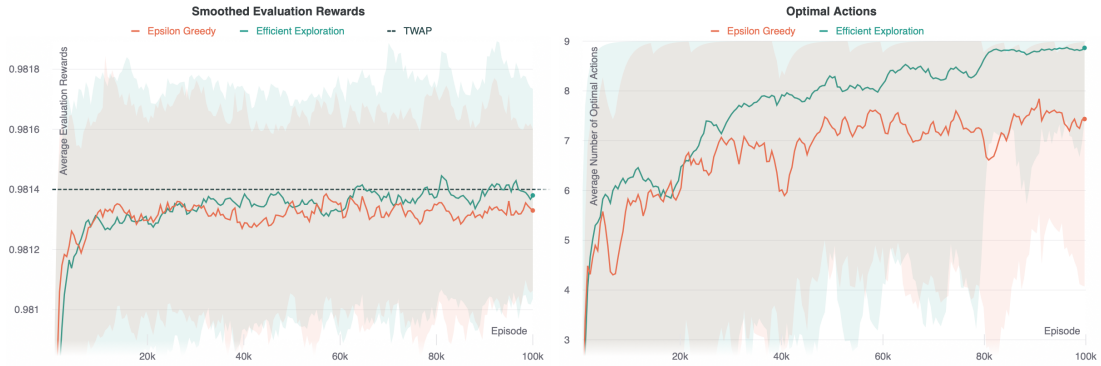
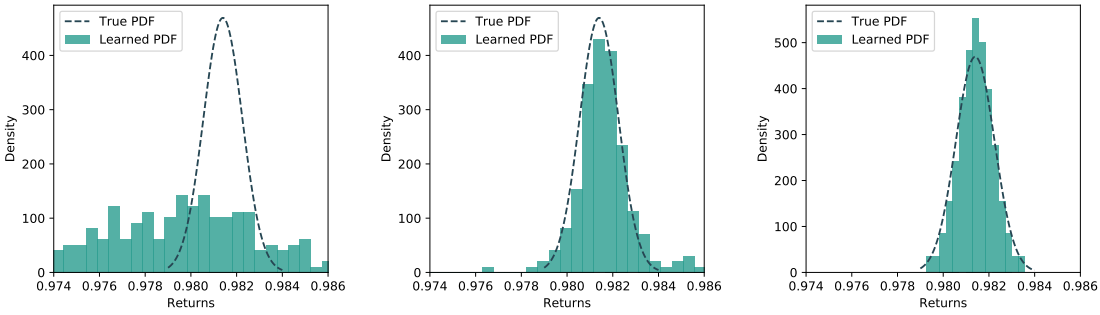


Figure 4.2: The smoothed average rewards and number of optimal actions over each evaluation period plotted against the number of training episodes for the QR-DQN Agent using an ϵ-greedy and efficient exploration behaviour policy. The bold line indicated the mean and the shaded region the minimum and maximum across all agents.

This result is not unexpected. With no prior training, both behaviour policies will be choosing actions largely at random. As the agent begins to learn the action value function however, the efficient exploration policy will choose actions based on how likely they are to be optimal rather

than the purely random exploration of the ϵ-greedy strategy.

While the QR-DQN agent learns the optimal policy within a reasonable number of training episodes, it is instructive to examine how well the learned quantile distribution approximates the value distribution. Figure 4.3 shows the approximate density function of the returns during the episode, learned by the agent. The initial distribution is very much dependent on the scaling applied to the returns before being processed by the network. The distribution is centred around 0.98, the value used to standardise the returns. The use of a simulated environment enables us to plot the true probability density function for the returns under TWAP. It can be seen that the approximation improves rapidly with training and by $2,000$ episodes it is already close to the true density function. Interestingly, even by $200,000$ training episodes the density function does not perfectly match the true density function for the returns and appears to underestimate the variance. The cause of this is not immediately clear.



(a) 0 training episodes (b) $2,000$ training episodes (c) $200,000$ training episodes

Figure 4.3: The learned distribution for the returns associated with the optimal action, TWAP, as the number of training episodes increase. The theoretical distribution of the returns is shown by the dashed line.

4.2 The Historical Environment

We now evaluate the QR-DQN agent using the efficient exploration policy from Section 2.3.4 in the historical market environment introduced in Section 3.3.2. The historical data we will be using a month of level 2 Bitcoin data starting from the 19^{th} of March 2019. As a volatile asset, Bitcoin represents a challenging problem for the reinforcement learning agent.

Before training or testing the agent, some minor preprocessing was conducted on the data set. Weekends and holidays were stripped out. Three columns were added: *market order imbalance*, defined as

$$\frac{\text{buy orders} - \text{sell orders}}{\max(\text{buy orders}, \text{sell orders})},$$

order imbalance, defined as

$$\frac{\text{bid size} - \text{ask size}}{\max(\text{bid size}, \text{ask size})},$$

where *bid size* and *ask size* represent the size of the bid and ask limit orders at the top of the book, and the *spread*

$$\text{ask} - \text{bid}.$$

The market data inputs to the Q-network, shown as the blue circles in Figure 3.1, were taken to be the *order imbalance*, *market order imbalance*, *mid price* and *spread*. This state space was chosen so as to reduce complexity and show that the agent can perform well even without large numbers of inputs. It could reasonably be expected that an agent would perform better with a larger state space. It would be an straightforward extension, for example, to include the bid and ask sizes, and more levels of order book data.

Table 4.2 shows the parameters used for the historical environment. Due to the limited data, we considered only a very small trading window of 200 seconds. Since the trading frequency was set to every 2 seconds this meant that $T = 100$. With a larger training data set it is reasonable

to assume that the length of the episode could be increased. The temporary impact was chosen to be 100 basis points, smaller than the simulated environment but still relatively large compared to observed values [20]. For this environment we consider the liquidation of a large order and set the initial position to $10,000$ satoshi units of Bitcoin. The actions will again correspond to the liquidation of different proportions of the original position. Since we now consider a specific number of units we divide the rewards by the initial position so that they are comparable with the simulated environment. We scale the temporary market impact coefficient for the same reason.

Parameter	Description	Historical Env
k	Temporary Market Impact	0.01
b	Permanent Market Impact	0
T	Period Length	100
R	Initial Position	10,000

Table 4.4: Parameters for the simulated execution environment used to compare the agents.

In order to fairly evaluate the agents the data set was partitioned into a training and testing set. The final two days of the month were used to test the agent. Each episode was started from a time uniformly selected from within the training and testing data sets, depending on whether the agent was being trained or evaluated. This approach meant that the same data could be used to train the agent multiple times making more efficient use of the data. The disadvantage of this approach, of course, is that the Q-network may start to overfit the training data if training is not cut short.

Parameter	Description	QR-DQN
c	Exploratory Coefficient for π_E	200
C	Target Lag	50
B	Batch Size	64
	Buffer Size	6000
n	Tree Horizon	100
ν	Adam Learning Rate	0.00005
N	Number of Quantiles	200
	Reward Scaling Mean	0.99
	Reward Scaling Standard Deviation	0.005

Table 4.5: Parameters used for the QR-DQN agent evaluated in the historical environment.

Table 4.5 shows the parameters used for the QR-DQN agent. The tree horizon was increased from the agent in the previous section to account for the longer episode length and to make the most of the limited dataset, as suggested by [30]. The buffer size and batch size were also increased for the same reason. Since the temporary impact is smaller than for the simulated environment, the reward scaling was adjusted to accommodate this.

Parameter	Description	Value
$\mathcal{S}$	State Space	$[t,Q]$
L_1	Number of Convolutional Layers	3
U_1	Number of Convolutional Units	10
L_2	Number of Fully Connected Layers	3
U_2	Number of Fully Connected Units	27

Table 4.6: Parameters for the neural network architecture, shown in Figure 3.1, for the QR-DQN agent evaluated in the historical market environment.

4.2.1 Execution using only Market Orders

We start by evaluating an agent only permitted to place market orders. While this is not a limitation that would realistically be applied for an optimal execution solution, it does enable us to compare the agents' performance to the TWAP benchmark. If exceeded, this shows that

the agent can learn to exploit market idiosyncrasies to achieve a better price when liquidating a position in an asset.

Throughout the episode, market orders were placed every 2 seconds. At each trading opportunity the agent could choose the size of the market order. The action space was restricted to just three actions close to TWAP corresponding to 99%, 100% and 101% of TWAP. With a larger action space the agent could potentially yield superior returns however restricting the action space meant that learning could be sped up, an important consideration due to the limited data.

5 QR-DQN Agents were trained over $10,000$ episodes resulting in approximately 1 million interactions with the environment for each agent. As with the simulated environment, every 500 training episodes, the agent was evaluated for 400 episodes. Figure 4.4 shows the smoothed, average evaluation rewards over each evaluation period, against the number of training episodes. It can be seen that the average reward exceeds the TWAP benchmark after only $1,000$ episodes however the agent does not appear to improve further. By the end of the training period the agent has reduced the execution cost of exiting the position by 1%. While only a modest improvement, it is likely that the agent would perform better with a larger set of training data. This can be seen by looking at the action values shown in Figure B.1. From this it can be seen that the agent quickly begins to overestimate the value of state action pairs due to overfitting of the training data.

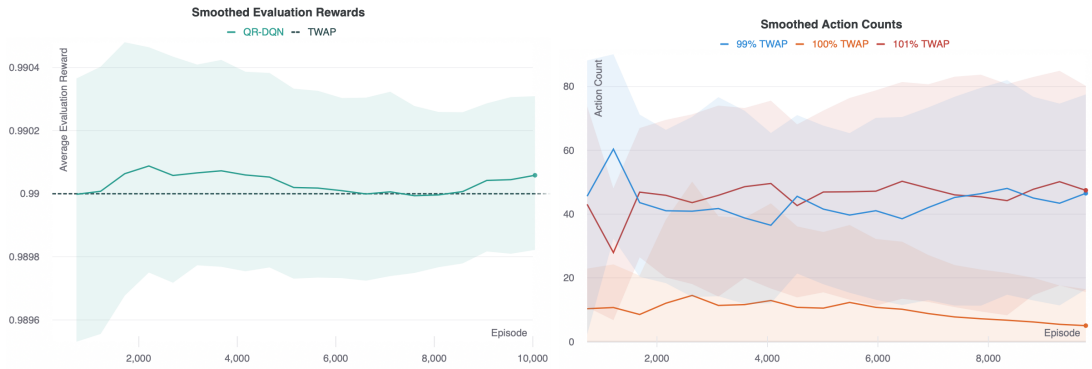


Figure 4.4: The smoothed average rewards and number of times each action is chosen over each evaluation period plotted against the number of training episodes for 5 QR-DQN Agents compared to TWAP.

The action counts provide an interesting overview of the policy learned by the agent. Throughout the training period it can be seen that the counts for the large and small market orders dominate.

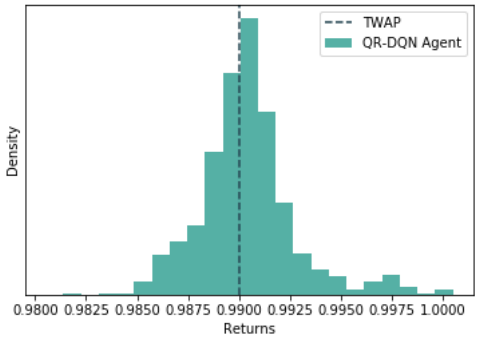


Figure 4.5: A fully trained QR-DQN agent was selected at random and evaluated over $1,000$ episodes on a days worth of evaluation data. The returns are shown in the histogram. The average return under TWAP (0.99) is shown by the dashed line. The average return for the agent was 0.9904.

The collection of fully trained agent were evaluated on previously unseen day of data. Each agent was run for $10,000$ episodes, each starting at a random point throughout the day. The evaluation rewards were recorded and compared to TWAP. An example of the returns for one of

these agents is shown in Figure 4.5. All 5 agents showed a statistically significant reduction in the cost of execution when compared to TWAP with 95% confidence.

4.2.2 Execution using Limit and Market Orders

We now evaluate a QR-DQN agent permitted to make both limit and market orders in the same historical environment. The action space was chosen to be a limited number of combinations of market orders and limit orders of different sizes. Specifically, the order size for market and limit orders could be selected independently from the set $\{0\%, 100\%, 200\%, 300\%, 400\%\}$, where each action represents a percentage of the TWAP order size.

We use the same environment as the market order only agent, with the slight modification of a 5 second trading frequency. This results in fewer episodes in the training dataset. Training was therefore curtailed after only $5,000$ episodes to minimise overfitting. Five QR-DQN agents using the parameters given in Table 4.5 were trained. The smoothed, average evaluation rewards are shown in Figure 4.6.

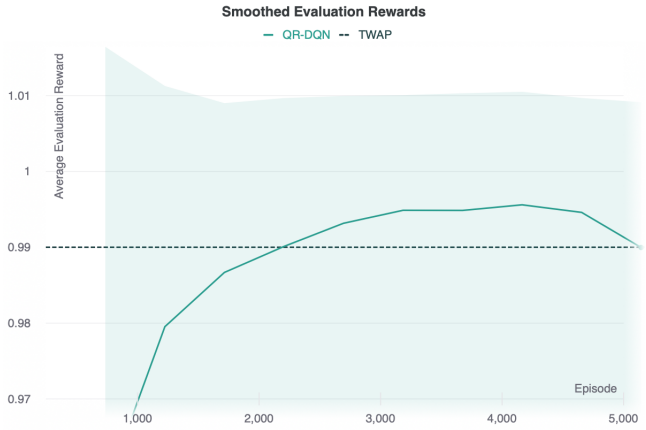


Figure 4.6: The smoothed average rewards against the number of training episodes for 5 QR-DQN Agents compared to TWAP. The mean evaluation reward is shown by the solid green line. The shaded area represents the range in the evaluation rewards.

Due to the wider range of actions, the cash received over each evaluation period starts from a much lower base compared to the market order agent. Given the actions available to the market order only agent were restricted to being close to TWAP this is not unexpected. After $2,000$ training episodes it can be seen that the agent begins to outperform TWAP, reducing execution costs by around 50% at its peak. It should be noted that benchmarking this algorithm against TWAP is not an entirely fair comparison given TWAP relies only on market orders. After $4,000$ training episodes the performance quickly starts to drop off, perhaps due to overfitting of the limited training data. As with the market order agent, it could be expected that performance would improve further with more training data.

In order to get an idea of the agents strategy, an agent was selected at random from the set of fully trained agents. This agent was evaluated over $1,000$ episodes on a previously unseen day of evaluation data. The number of times each action was chosen for each time step in the episode was plotted in Figure 4.7. It can be seen from this that the agent appears to place large limit orders at the beginning of the episode, utilising market orders to a greater extent as the episode progresses. The agent appears not to use the full length of the execution window. This is likely due to the limited set of actions available to it. With more granular control over the order size performance could likely be improved.

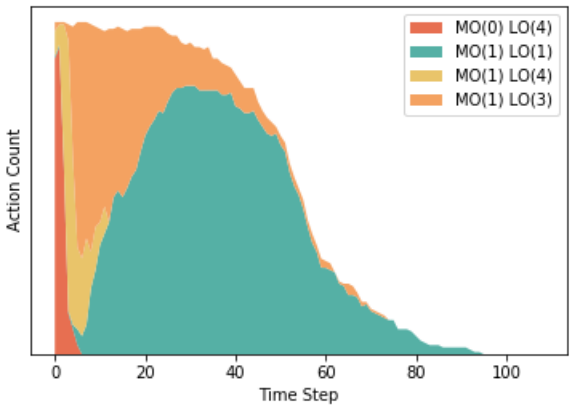


Figure 4.7: A plot of the frequency with which a selection of the most used actions were chosen at each time step for $1,000$ evaluation episodes in the historical environment. The numbers in brackets refer to the sizes of the market order (MO) and limit order (LO) chosen with relative to the TWAP trading rate.

Conclusion

Recent years have seen impressive progress in reinforcement learning. Some of these developments have been discussed in this thesis. The results shown in Chapter 4 demonstrate that reinforcement learning has potential as a tool for optimal execution. In particular QR-DQN can be seen to be a promising alternative to DQN. By considering the distribution, the agent learns the optimal strategy faster and performs better in a simulated environment. Being off policy, it has the potential to learn from previously generated price trajectories. If it could be shown that an agent can reach an acceptable standard using offline learning then this would be a significant step in reducing the barriers to real world use.

The results in a historical environment show that reinforcement learning agents are able to learn market idiosyncrasies and adjust the rate of trading to take advantage of these throughout the execution period. While the improvements are modest, the training data set consisted of just a months worth of data and it could be expected that with a larger training dataset, performance improvements could be more substantial.

The potential to use both limit and market orders was also demonstrated. This scenario is more realistic in the real world. Reinforcement learning is a highly flexible tool and this shows that the agent can be adapted to use complex action spaces with more than one order type.

While we have showed that there is scope for the use of reinforcement learning in optimal execution, there are many avenues left open for future work. The state and action spaces chosen for the experiments discussed were small. The inclusion of more features and experimentation on different asset classes could show to what extent the agents can be refined for specific markets. The emerging discussion surrounding offline learning also shows great promise for the training of optimal execution agents using real data. This would be a particularly interesting direction to take in future work.

Many of the difficulties in applying reinforcement learning to optimal execution are not limited to the trading environment. Work on the application of reinforcement learning to other fields is likely to be transferable to finance. Given the ever growing volume of work on the subject, the production use of reinforcement learning for financial applications seems a very real prospect.

Appendix A

Technical Proofs

Lemma 2.3.5. Let $\bar{d}_p$ denote the Maximal p-Wasserstein metric and $\mathcal{T}^\pi$ be the distributional Bellman operator defined previously. Then, $\mathcal{T}^\pi : \mathcal{Z} \to \mathcal{Z}$ is a contraction in $\bar{d}_p$

Proof. Let $G_1, G_2 \in \mathcal{R}$ and consider the Wasserstien metric d_p from Definition 2.3.4.

$$\begin{aligned} d_p(\mathcal{T}^\pi G_1(s,a), \mathcal{T}^\pi G_2(s,a)) =& d_p(R(s,a) + \gamma P^\pi G_1(s,a), R(s,a) + \gamma P^\pi G_2(s,a)) \\ \leq& \gamma d_p(P^\pi G_1(s,a), P^\pi G_2(s,a)) \end{aligned}$$

by properties of d_p. Using the definition of P^π we have,

$$\gamma d_p(P^\pi G_1(s,a), P^\pi G_2(s,a)) \leq \gamma \sup_{s',a'} d_p(G_1(s',a'), G_2(s',a'))$$

This is the maximal Wasserstien metric and so we therefore have,

$$\begin{aligned} \bar{d}_p(\mathcal{T}^\pi G_1, \mathcal{T}^\pi G_2) &:= \sup_{s,a} d_p(\mathcal{T}^\pi G_1(s,a), \mathcal{T}^\pi G_2(s,a)) \\ &\leq \gamma \sup_{s',a'} d_p(G_1(s',a'), G_2(s',a')) \\ &\leq \sup_{s',a'} d_p(G_1(s',a'), G_2(s',a')) \\ &= \bar{d}_p(G_1, G_2) \end{aligned}$$

□

Appendix B

Supplementary Figures

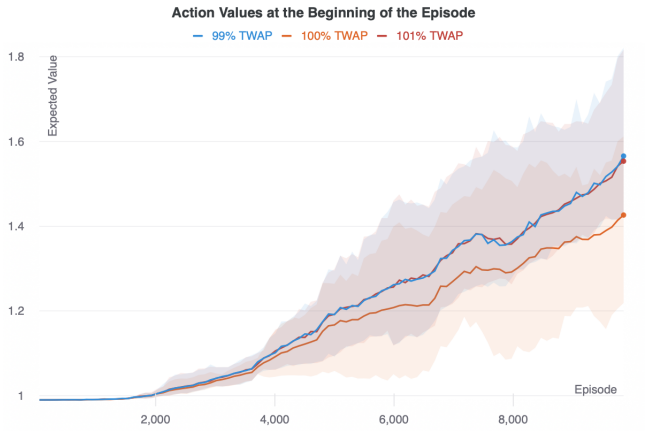


Figure B.1: The estimated value of each of the three actions at the beginning of the episode as the number of training episodes increases. It can be seen from this that the agent begins to overfit the training data after around $2,000$ training episodes.

Appendix C

Code and Implementation

All reinforcement learning algorithms were implemented from scratch using Tensorflow [48]. Performance was monitored using Wandb [49]. The full code base can be found at `github.com/Geofffff/ThesisCode`.

Bibliography

[1] Jose Penalva Alvo Cartea, Sebastian Jaimungal. *Algorithmic and High Frequency Trading.* Cambridge University Press, 2015.

[2] Gerald Tesauro and Jonathan L. Bredin. Strategic sequential bidding in auctions using dynamic programming. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, AAMAS '02, page 591–598, New York, NY, USA, 2002. Association for Computing Machinery.

[3] Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. volume 2006, pages 673–680, 01 2006.

[4] Dieter Hendricks and Diane Wilcox. A reinforcement learning extension to the Almgren-Chriss model for optimal trade execution. *IEEE/IAFE Conference on Computational Intelligence for Financial Engineering, Proceedings (CIFEr)*, 03 2014.

[5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[6] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *Int. J. Rob. Res.*, 32(11):1238–1274, September 2013.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[8] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *ArXiv*, abs/1710.02298, 2018.

[9] Brian Ning, Franco Ling, and Sebastian Jaimungal. Double deep Q-learning for optimal execution. 12 2018.

[10] Kevin Daberius, Elvin Granat, and Patrik Karlsson. Deep execution — value and policy based reinforcement learning for trading and beating market benchmarks. *SSRN Electronic Journal*, 01 2019.

[11] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 449–458. JMLR.org, 2017.

[12] Will Dabney, Mark Rowland, Marc G Bellemare, and Rémi Munos. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[13] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. *arXiv preprint arXiv:1806.06923*, 2018.

[14] Robert Almgren and Neil Chriss. Optimal execution of portfolio transactions. *Journal of Risk*, pages 5–39, 2000.

[15] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. 09 2015.

[16] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[17] Kamyar Azizzadenesheli, Emma Brunskill, and Animashree Anandkumar. Efficient exploration through bayesian deep Q-networks. 02 2018.

[18] Christine Parlour and Duane Seppi. Limit order markets: A survey. *Handbook of Financial Intermediation and Banking*, 12 2008.

[19] Thierry Foucault, Marco Pagano, and Ailsa Roell. *Market Liquidity: Theory, Evidence, and Policy*. Oxford University Press, 2013.

[20] Robert Almgren, Chee Thum, Emmanuel Hauptmann, and Hong Li. Direct estimation of equity market impact. *RISK*, 18, 04 2005.

[21] Emilio Said, Ahmed Bel Hadj Ayed, Alexandre Husson, and Frédéric Abergel. Market impact: A systematic study of limit orders. *Market Microstructure and Liquidity*, 3(03n04):1850008, 2017.

[22] Dean Teneng. Limitations of the Black–Scholes model. *International Research Journal of Finance and Economics*, pages 99–102, 01 2011.

[23] Christopher Watkins. Phd thesis, learning from delayed rewards. 01 1989.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[25] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[26] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[27] Mikko Pakkanen. Lecture notes in deep learning, December 2019.

[28] Warren S Sarley. comp.ai.neural-nets FAQ, part 2 of 7: Learning, Oct 2002.

[29] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8(3–4):293–321, May 1992.

[30] Hado Van Hasselt, Matteo Hessel, and John Aslanides. When to use parametric models in reinforcement learning? 06 2019.

[31] Hado Van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*, 2018.

[32] S. Nadarajah and S. Kotz. Exact distribution of the max/min of two gaussian random variables. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):210–212, 2008.

[33] Hado V. Hasselt. Double Q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[34] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Int. Res.*, 47(1):253–279, May 2013.

[35] Saleh Almezel, Qamrul Hasan Ansari, and Mohamed Amine Khamsi. *Topics in Fixed Point Theory*. Springer, Cham, 2014.

[36] Roger Koenker. *Quantile Regression*. Econometric Society Monographs. Cambridge University Press, 2005.

[37] Clare Lyle, Marc G Bellemare, and Pablo Samuel Castro. A comparative analysis of expected and distributional reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4504–4511, 2019.

[38] Borislav Mavrin, Shangtong Zhang, Hengshuai Yao, Linglong Kong, Kaiwen Wu, and Yaoliang Yu. Distributional reinforcement learning for efficient exploration. *arXiv preprint arXiv:1905.06125*, 2019.

[39] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2016.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[41] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019.

[42] Thilo A Schmitt, Desislava Chetalova, Rudi Schäfer, and Thomas Guhr. Non-stationarity in financial time series: Generic features and tail behavior. *EPL (Europhysics Letters)*, 103(5):58003, 2013.

[43] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, et al. Deep Q-learning from demonstrations. *arXiv preprint arXiv:1704.03732*, 2017.

[44] Irene Aldridge. *High-Frequency Trading*. John Wiley & Sons, Ltd, 2010.

[45] PD Kingma and J Ba. Adam: A method for stochastic optimization, arxiv (2014). *arXiv preprint arXiv:1412.6980*, 106, 2015.

[46] John C. Hull. *Options, futures, and other derivatives*. Pearson Prentice Hall, Upper Saddle River, NJ [u.a.], 6 edition, 2006.

[47] Marlos C Machado, Marc G Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research*, 61:523–562, 2018.

[48] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[49] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.