HEC MONTRÉAL

Optimal Order Execution with Deep Reinforcement Learning

par

Brittany Rockwell

Mémoire présenté en vue de l'obtention

du grade de maîtrise ès sciences en analytique d'affaires

Département de sciences de la décision

Décembre 2019

Résumé

L'ordre optimal d'exécution est un problème important auquel font face les gros joueurs des marchés financiers, étant donné qu'il requiert de balancer le risque de détenir les actifs contre le risque de volatilité engendré par la vente d'une large position. Si un participant place un ordre d'exécution trop large, le risque de marché s'accroît, affectant ainsi le prix des actifs à travers les mécanismes traditionnels d'offres et de demande. Si le participant garde sa position trop longtemps, il accroît son risque d'être exposé à un mouvement défavorable des prix. Ce travail est le premier exemple connu de l'application de l'algorithme '*Twin Delayed Deep Deterministic Policy Gradient'* (TD3) ou '*Deep Deterministic Policy Gradient'* (DDPG) au problème d'ordre optimal d'exécution. Il s'agit aussi de la première application d'apprentissage par renforcement pour exploiter le biais du processus de prix Brownien dans le contexte boursier. Nous démontrons que le TD3 peut converger, relativement rapidement, vers la stratégie optimale, le *prix moyen pondéré en fonction du temps* (TWAP), lors d'un processus de prix "*martingale*", tout en utilisant un espace d'action continu. De plus, nous démontrons que le TD3 surpasse significativement le modèle de référence TWAP et se révèle être plus stable que l'algorithme DDPG, pour un processus de prix Brownien lorsque le paramètre de biais est posé à 10%.

Mots clés : apprentissage par renforcement, ordre optimal execution, apprentissage en profondeur

Abstract

Optimal order execution is an important problem faced by large market participants, as it requires the participant to balance the risk of holding the asset through price volatility with the market impact risk of exiting a large position. If a participant places too large an order, the risk of market impact increases, thereby affecting the price of the asset through standard supply-demand dynamics in the order book. If the participant holds the position too long, he/she increases the risk of exposure to unfavorable price movements. This work is the first known example of applying *Twin Delayed Deep Deterministic Policy Gradient* (TD3) or *Deep Determistic Policy Gradient* (DDPG) algorithm to the optimal order execution problem. It is also the first known work where reinforcement learning is used to exploit the bias in a skew-normal Brownian motion price process in trading. We demonstrate that TD3 can achieve relatively quick convergence to the optimal policy of a martingale price process when using a continuous action-space. Further, we demonstrate how TD3 significantly outperforms the baseline time-weighted average price strategy and proves to be more stable than the DDPG algorithm in a skew-normal Brownian motion price process with the skew parameter set to only 10%.

Keywords : optimal order execution, reinforcement learning, stock trading, deep learning, actor-critic, dynamic optimization

Table of Contents

Résumé....3
Abstract....5
List of Abbreviations....14
Acknowledgements....16
Introduction....18
- Optimal Order Execution....19
  - Order Book....20
  - Problem....21
  - Traditional Methods....24
- Reinforcement Learning....25
  - Markov Decision Processes....26
    - State....27
    - State Transition Model....27
    - Action....28
    - Reward....28
    - Agent....29
    - Optimizing the Value Functions using Bellman's Optimality Equation....30
- Tabular Methods....33
  - Dynamic Programming....34
- Solving MDPs using Reinforcement Learning....34
  - On-policy vs. Off-policy....35
  - Exploration vs Exploitation....35
  - Model-free vs Model-based....36
  - Monte-Carlo Learning....36
  - Temporal Difference Learning....37
    - SARSA vs. Q-Learning....38
- Approximate Methods....39
  - Linear Approximation....40
  - Non-Linear Approximation....41
    - Linear vs Non-Linear Approximation....41
    - Neural Networks....43
- Policy Gradients....44
  - Actor-Critic Methods....45

Deep Reinforcement Learning for Optimal Order Execution....46
Deep-Q Learning (DQN)....47
Experience Replay....49
Delayed Target Network Updates....49
Pseudo Code....50
Double Q-Learning (DDQN)....53
Algorithmic Differences between DQN and DDQN....54
Deep Deterministic Policy Gradients (DDPG)....55
Pseudo Code....56
Twin Delayed Deep Deterministic Policy Gradient (TD3)....57
Architectural Comparison with DDPG....57
Pseudo Code....59
Optimal Order Execution using Reinforcement Learning....59
Setup....60
Evaluation Metric and Reward....61
Independence Assumption and the Quadratic Penalty....64
Transaction Costs....65
Hypotheses....65
Methodology....66
Data....66
Brownian Motion as a Price Process for Order Execution....67
Technical Specifications....71
Feature Engineering....71
Raw Features....71
Derived from Raw Features....72
Algorithm Selection....73
Training....74
Experiments....74
Hyper-parameters....74
Pre-training....75
Tuning....76
Train and Test Data....77
Comparison with Ning et al. 2018....78

Setup and Data....78
Algorithm....78
Reward....79
Analysis....79
Results and Analysis....80
Experiment 1: Agent Performance at Learning Optimal Policy....80
Brownian Motion Comparative Returns to Steady Sell-off....81
First Half of Training Action Distribution....83
Second Half of Training Action Distribution....84
Experiment 2: Agent Performance in Different Degrees of Bias....84
Brownian Motion Price Process with Bias +.001....85
Brownian Motion Price Process with Bias +1%....87
Brownian Motion Price Process +10%....89
Discussion....92
Summary of Problem....92
Summary of Model....93
Summary of Results....93
Limitations....96
Contribution....96
Future Research....97
Conclusion....98
Bibliography....100

Index of Figures

Figure 1: Linear Model Solution for Separating Coloured Points.....42
Figure 2: Non-Linear Model Decision Boundaries for Separating Coloured Points.....42
Figure 3: Linear Regression vs 1-Neuron Neural Network.....43
Figure 4: Deep Q-Learning with Experience Replay.....51
Figure 5: Double Q-Learning with Experience Replay.....54
Figure 6: Deep Deterministic Policy Gradients.....56
Figure 7: TD3 (left) vs DDPG (right) Architectures.....57
Figure 8: Twin Delayed Deep Deterministic Policy Gradients (TD3).....58
Figure 9: Probability Density Function of Skew-Normal with Different Skew Parameters.....70
Figure 10: Example of Standardized Brownian Motion Price Process: 10 Episodes.....81
Figure 11: Target Network Relative P&L for TD3 vs TWAP Experiment 1.....82
Figure 12: Target Network Relative P&L for TD3 vs DDPG Price Process with .1% Bias.....86
Figure 13: Standardized Price Process of 1% Skewed Brownian Motion.....87
Figure 14: Target Network Relative P&L for TD3 vs DDPG Price Process with 1% Bias.....88
Figure 15: Example of 10% Skew Brownian Motion Price Process: 10 Episodes.....89
Figure 16: Target Network Relative P&L for TD3 vs DDPG Price Process with 10% Bias.....90

Index of Tables

Table 1: List of TD3 Hyper-parameters.....75
Table 2: Relative P&L Experiment 1.....83
Table 3: First Half Action Distribution Experiment 1.....83
Table 4: Second Half Action Distribution Experiment 1.....84
Table 5: Relative P&L Experiment 2.....87
Table 6: Relative P&L Experiment 3.....89
Table 7: Relative P&L Experiment 4.....91

List of Abbreviations

Deep Q-Learning: DQN

Double Deep Q-Learning: DDQN

Deep Deterministic Policy Gradient: DDPG

Deep Reinforcement Learning: DRL

Limit Order Book: LOB

Monte-Carlo Learning: MC

Machine Learning: ML

Optimal Order Execution: OOE

Reinforcement Learning: RL

Root Mean Square Propagation: RMSProp

Temporal Difference Learning: TD

Time-weighted Average Price: TWAP

Twin Delayed Policy Gradient: TD3

Volume-weighted Average Price: VWAP

Acknowledgements

I would like to take the opportunity to thank my advisor, Professor Laurent Charlin from the Department of Decision Sciences at HEC Montreal, for his continuous support throughout this process. He encouraged my intellectual pursuits, all while providing invaluable grounding guidance in how I could achieve them. By showing a genuine commitment to my success, he taught me important lessons about mentorship and for that, I am better for, both academically and professionally.

I would also like to thank Professor Manuel Morales from the Department of Mathematics and Statistics at the University of Montreal and National Bank of Canada. Because of his commitment to my development, I was able to share my work on several occassions with the academic and professional communities. Public speaking has become slightly less fear-inducing because of these experiences, if only slightly.

Last, I would like to thank my partner, friends and family for their ongoing encouragement throughout this process. For work or thesis, I have missed several special occassions over the years. I appreciate their unwaivering support and understanding more than they know, and I am looking forward to this new chapter with them.

Introduction

Optimal order execution is an important problem faced by large market participants, as it requires the participant to balance the risk of holding the asset through price volatility with the market impact risk of exiting a large position. If a participant places too large an order, the risk of market impact increases, thereby affecting the price of the asset through standard supply-demand dynamics in the order book. If the participant holds the position too long, he/she increases the risk of exposure to unfavorable price movements.

The intent of this work is to demonstrate the feasibility of using deep reinforcement learning (DRL) in practical applications related to financial stock trading. Many of the hurdles facing the adoption of DRL in trading relates to the length of time it takes more traditional DRL methods to converge. In true market conditions, stock price movements are non-stationary, where patterns are quickly eliminated because of market efficiency. It is imperative for algorithms to converge quickly and adjust to new patterns accordingly if they are to ever be used in live trading.

This thesis attempts to answer questions of signal detection and time to convergence by explicitly creating synthetic price processes that closely resemble that of real-time data so that tangible and definitive conclusions of performance can be made. Answers to questions of this nature are important for real-world implementation in financial stock trading where training on historical data alone is simply not enough (López

de Prado, 2018). The successful implementation of machine learning requires that patterns uncovered in training data can be generalized to hold-out data. Off-line machine learning fails frequently in non-stationary financial time-series where patterns change quickly and abruptly, so on-policy methods can be applied to mitigate the risk of over-fitting. For this reason, ground truth comparisons using synthetic price processes are used to highlight the strength and flexibility of DRL in financial stock trading. A great deal of work was done to create experiments that closely resemble true market conditions to demonstrate the algorithm's ability to detect and exploit varying levels of signal in a stock's price process. The reason for opting for synthetic data that closely resembles true short-term price movements is so the experiments can confirm the algorithm's ability to converge to an optimal policy. Further, injecting varying degrees of synthetic bias into the price process can help us understand the true speeds at which the algorithm converges and its ability to adjust to new patterns when learning in real-time.

Optimal Order Execution

Optimal Order Execution (OOE) is a dynamic optimization problem that can be modeled as a Markov Decision Process. First, this section will discuss the concept of an order book, order types and the impact an order can have on market prices and thus the trader's return. Second, we will outline game dynamics of the OOE problem. Finally, there will be a review of the more traditional methods in solving the OOE problem, as well as their limitations.

Order Book

The order book represents a record of all active orders for a financial instrument on the market at any given time. Each instrument possesses its own order book and changes as new orders for the instrument are received by the market. The book contains two sides: the ask side, which is the side of orders signalling a desire to sell at a given price and the bid side, which signals a desire to buy at a given price. The most favourable ask price is naturally higher than the most favourable bid price, demonstrating a seller's desire to maximize the price at which they sell and a buyer's desire to minimize the price at which they buy for; the difference between these two most favourable orders is called the spread.

There are primarily two types of orders that can be submitted to the market: limit orders and market orders. A limit order signals to the market a promise to pay at a given price for a specified volume of stocks. In a way, it is a conditional promise to pay which is triggered by changes in price competitiveness in the order book. If the desired price is never realized, the limit order will never be placed. Conversely, a market order is an unconditional promise to trade a given number of shares, where the price is determined by the most competitive limit orders in the order book. For example, if a trader places a market order looking to buy a specific number of stocks, the order will be fulfilled automatically at the lowest available ask price across all limit orders. Say for instance, there are two limit orders in the book on the ask side. Limit order A is to sell 50 shares at $10 and limit order B is to sell 100 shares at $11. Limit order A is clearly more competitive than B based on price. If we were to place a market order to buy 100 shares, we would fill limit order A at $10/share, but we would still need 50 more shares to fill our market order of 100. Because limit order A no longer exists, we would have to travel

deeper into the book to hit limit order B, where we would buy 50 shares at the less optimal $11/share. Limit order B would still exist, but rather than 100 shares being listed at $11/share, there would only be 50 remaining. The most competitive limit order on the ask side is now limit order B for 50 shares at $11/share. The average price per share paid using the market order was $10.5 per share: ($10·50A+$11·50B)/100 shares = $10.5/share.

As can perhaps be inferred from the above example, large market orders tend to impact the price of an asset because it realizes several orders in the order book simultaneously. With every order that is filled, the most competitive price must go further and further up or down the order book, thus trading at increasingly sub-optimal price points. In this work, large orders are penalized using a quadratic penalty, which eats into the agent's return by simulating the need to go deeper into the order book. These dynamics directly contribute to why order execution optimization problems have been a point of discussion in the financial trading literature.

Problem

OOE is a one-sided trading problem and therefore is much simpler than the more popular trading game where the participant must 'buy', 'hold' or 'sell' stocks to maximize returns. OOE consists of a trader being given X total inventory to sell at the beginning of the episode, each episode being of a fixed length in time T number of seconds. When T amount of time has elapsed, we say the current step is terminal and the game is over. There are N decision points over the course of an episode, we refer to these as *k steps*

where $k \in [0, 1, .., N]$. Each k is separated by a discrete time interval t where t $\in [0, 1, ..,$ $N]$. Each t spans $\frac{T}{N}$ seconds, or referred to as M seconds moving forward. At each step k, we must determine the amount of inventory to sell x_k such that $\sum x_k$ equates to X to ensure full liquidation when the episode terminates. Rather than submitting one large order at the beginning of each step and negatively impacting available prices in the order book, we would like to equally disperse x_k throughout the time interval t by following a uniform schedule at 1 second time-increments. To obtain the size of the per second order increment, we must compute $\frac{x_k}{M}$. Once the order is complete at the end of time interval t we move to step $k+1$ where a new amount x_{k+1} must be determined to then be sold over the course of $t+1$. For example, say our episode length T is 3600 seconds (1 hour) with 5 steps N every 720 seconds M, and we must sell 100,000 shares X. We must decide how much inventory x_k to sell at the beginning of every k step. To mitigate the risk of impacting the order book, we wish to sell equal sized orders every 1-second that sum to x_k throughout t. We compute the size of the incremental per second orders using $\frac{x_k}{720}$. We are setting a new uniform schedule at the beginning of every step. Once x_k has been decided, we are locked in until 720 seconds has elapsed, and we have liquidated all x_k. Naturally, selling every second exposes us to any price fluctuations over the course of the 720 second time interval. Define $i \in [0, 1, .., 720]$ where i represents each second within

the time interval *t*. We calculate our step-wise return as $p_i \frac{x_k}{720}$ where p_i is the price at time-increment *i*.

Next, we discuss how the baseline is determined in the OOE problem. Volume-weighted Average Price (VWAP) strategy is often used as a benchmark in algorithmic trading because it represents the average performance of a trader over the course of a given time-horizon if the volume traded by the trader was always proportional to the volume trading in the market at the time of each trade (Berkowitz et al., 1988). VWAP is determined by finding the average price at which the instrument was traded at over a given time horizon weighted by the ratio of volume to total volume for which the trade price was realized. VWAP is used as a benchmark for strategists looking to gauge their performance relative to how the market performed historically in a limit-order-only setting. Recall the impact executed orders will have on the order book, where larger orders must trade with limit orders deeper into the book and thus trading at increasingly unfavourable prices. Conversely, Time-weighted Average Price (TWAP) is the average price of a security over a specified time, regardless of volume and order book state. The OOE problem in this work will only allow the agent to execute market orders at equidistant time-intervals while applying a quadratic penalty (Ning et al., 2018). The quadratic penalty will simulate the impact the market order will have on the return of the agent and any transaction costs incurred from order size. TWAP, where the agent sells the same amount at each of the 3600 orders and is used as the performance benchmark in this work. Baseline TWAP simulates a steady-sell off at each time step increment *i* for each *t*. For example, if the

agent must sell 100,000 shares, the agent will sell $\frac{100{,}000}{3600} = 27.77$ shares a second or 20,000 shares every step *k*.

Traditional Methods

The OOE problem is a well studied dynamic optimization problem. This section will outline several past methods that have been proposed, mostly involving traditional optimization and significant modelling assumptions. Ning, Ling, & Jaimungal (2018) assert that researchers have attempted to solve the OOE problem by following a fairly established methodology: first, select a stochastic model or stochastic volatility process, estimate model configurations using past trading data, then specify an objective function and solve the problem by way of stochastic optimal control. They argue that these traditional methods are limited in circumstances where the price process is highly complex or unknown. This section details a few of these more traditional methods.

One of the earliest works in the field of optimal order execution is that of Almgren & Chriss (2001). The authors formulated the problem as to maximize returns in lieu of trading frictions, such as trading expenditures and price volatility. The objective was to find the optimal balance in minimizing the impact of liquidation on total return while positively benefiting from price volatility where the authors opted to model the price process as a Brownian motion and used dynamic programming methods under three core assumptions: risk-averse, risk-neutral and risk-neutral gain due to serial correlation. It is important to note that this work served as a stepping stone for several other works in the optimal order execution space (Cartea & Jaimungal, 2015; Hendricks, n.d.; Kato, 2017;

Romero & Bautista, 2016; Zhou et al., 2017). As automated trade execution became more prevalent with the adoption of telecommunications infrastructure in trading, understanding the interplay between volatility risk and reward in inventory sell-off became more important as well. For our purposes, the limitations in the Almgren & Chriss paper have more to do with the assumptions underlying the methodology, whereas much of the later works building off of the Almgren-Chriss model still follow this same traditional methodology outlined by Ning, Ling & Jaimungal (2018). Because of this, much of the literary review will focus on model-free reinforcement learning and its ability to solve the OOE problem.

Reinforcement Learning

It is difficult to outline the relevant developments of reinforcement learning without referencing the important work of Sutton and Barto in *Introduction to Reinforcement Learning*, originally published in 1998, with a second edition recently published in 2018. The authors draw on work from colleagues and personal research to paint a cohesive picture of how reinforcement learning (RL) evolved from optimal control problems in the 1950s to more advanced topics and research frontiers.

Like Sutton and Barton's book, this section begins by summarizing much of the general concepts in RL that are fundamental to this thesis, such as historical context, tabular methods and approximate methods. This section deviates away from Sutton and Barton by detailing recent advances in the field of RL, particularly developments in model-free sensor-like input learning, a branch of DRL that has heavily influenced the

advent of the TD3 algorithm, which is the primary model used in this work. The first part of this section outlines the core elements of traditional RL to better understand its limitations in the broader context of sequential decision-making. Second, recent advancements in the field will be discussed to show theoretical improvements to these limitations. Third, in order to demonstrate the feasibility of DRL to learn from sensor-like data alone, several examples of DRL exceeding human-level performance on real-world applications will be provided. The primary purpose of this section is to provide insight into algorithm choice, design and game elements (reward, observation and action space, etc.) for the reader.

Markov Decision Processes

A Markov Decision Process (MDP) is a way of formulating a sequential decision-making problem where we have full knowledge of the environment, that is we know the transition probability function and reward function. It is important to make this distinction early on; MDPs relate to the formulation of a problem, whereas RL relates to the methods used to solve for them when there exists no formal definition of the transition probability function or the reward function. An MDP is a way of formulating problems requiring sequential decision-making, where an agent interacts with its environment through taking one action a from a restricted set of actions A, thereby influencing both its immediate and future rewards. An environment is a set of states s, where states can change from one to another as a direct result of the agent's actions, where the next state is determined using the state-transition model. The agent is goal oriented, as it attempts to learn an optimal policy π^* that will maximize its reward function. MDPs will generally use 4 elements to produce the trajectories it uses to learn the optimal policy: $(s_t, p(s_{t+1}|s_t, a_t), a_t, r_t)$. We define

these components in the following section. We have found the structural flow of certain resources on the topic, namely the MDP documentation of Ritchie Ng (2018), to be excellent and have tried to shape our explanation of the MDP problem in a similar fashion for clarity.

State

A state s_t is typically a vectorized representation of the agent's environment at time t. A central assumption of MDPs is that they possess the Markov property, that the future state only depends on the current state, s_t and action a_t, as opposed to, for example, the history of previous states.

Another important facet of a state is that of a terminal state. Terminal states are states in which a condition was met that ends the game. Terminal conditions can be constraints on a wide range of factors, for instance, maximum number of steps an agent is permitted to take, maximum reward and lives lost.

State Transition Model

MDPs can define a state transition model that describes the way in which the environment s_t will transition to s_{t+1} when selecting a_t. We refer to this as state-transition probabilities which are defined as $P(s_{t+1}|s_t, a_t)$.

Action

Agents interact with the environment using available actions A in each state s_t, thereby entering a new state s_{t+1}. Actions are defined by the action space A and can be both continuous or discrete in nature. The way the action space is designed has several implications, such as algorithm restrictions and varying complexity. These implications will be discussed in detail in sections discussing the optimal order execution problem in a reinforcement learning setting.

Reward

Reward R_t is measure of utility derived from the reward function and is attributed when an agent chooses an action a_t within a given state s_t that subsequently transitions to s_{t+1} using the transition probability function. Rewards are aggregated as the agent interacts with its environment:

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^k R_{t+k+1} \tag{1}$$

Where k represents the number of steps before encountering a terminal state and γ signifies the discount rate and is typically within the range [0, 1]. The objective of an MDP is to *maximize* G_t. Higher values in γ emphasize the importance of future rewards to the agent, a concept central to both traditional MDPs and modern RL. Once future returns are aggregated, we can then derive the state-value function V_π and action-value function Q_π such that:

$$V_\pi(s) = E_\pi[G_t \mid S_t = s]$$

and

$$Q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a]$$

Where the value function describes the expectation of how valuable it is to be in state *s* and the action-value function describes the expectation of how valuable it would be to take action *a* in state *s.* To complete our explanation of an MDP, we will next describe the agent and common methods for solving these types of problems.

Agent

The goal of the agent is to maximize future aggregated rewards $\sum \gamma^{t-1} R_t$ where $t \in T$ before reaching a terminal state and γ represents the discount factor. The agent determines the expected total reward G_t at each time step, with the inclusion of a discount rate to consider that future rewards may or may not be as important as more immediate ones. An agent's policy π can be either deterministic or stochastic:

- *Deterministic policy*: a deterministic policy intakes a state and produces an action such that:

$$a = \pi(s)$$

- *Stochastic policy*: a stochastic policy intakes an action given a state and produces a probability of selecting that action given that state such that:

$$P_{\pi}[A=a \mid S=s] = \pi(a|s)$$

The agent selects the action according to policy π and recovers the next state s_{t+1} by using the state transition probability function P such that:

$$P(s_{t+1} \mid s, a) = P[S_{t+1}=s_{t+1} \mid S_t=s,\ A_t=a]$$

Using the information from its new transition, the agent computes the reward from committing action a_t in state s_t and ending up in the next state s_{t+1}. The optimal policy for a state is the policy in which the value-function and the action-value function are maximized such that:

$$\pi*(s_t) = argmax_{\pi} V_{\pi}(s_t) = argmax_{\pi} Q_{\pi}(s_t, \pi(s_t)) \quad (2)$$

Optimizing the Value Functions using Bellman's Optimality Equation

The problem arises when we need to estimate future discounted rewards to obtain G_t; although it is simple to compute our immediate rewards, how can we account for future rewards that have yet to occur? The Bellman equation (BE), an equation conceived by Richard E. Bellman, describes the relationship between the value function (or action-value function) in the current state s_t and the value function (or action-value function) in the

next state s_{t+1} (Bellman, 1952). According to the BE, the action-value function can be reformulated to express this relationship such that:

$$\begin{aligned} Q_\pi(s,a) &= E_\pi[G_t \mid S_t=s, A_t=a] \\ &= E_\pi[R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+ \ldots \mid S_t=s, A_t=a] \\ &= E_\pi[R_{t+1}+\gamma(R_{t+2}+\gamma R_{t+3}+ \ldots) \mid S_t=s, A_t=a] \\ &= E_\pi[R_{t+1}+\gamma G_{t+1} \mid S_t=s, A_t=a] \\ &= E_\pi[R_{t+1}+\gamma Q_\pi(s_{t+1}, a_{t+1}) \mid S_t=s, A_t=a] \end{aligned}$$

Likewise, for the state-value function:

$$V_\pi(s) = E_\pi[R_{t+1}+\gamma V_\pi(s_{t+1}) \mid S_t=s]$$

The interplay between the state-value function and the action-value function is critical for understanding concepts described in the pseudo-code of the various DRL algorithms we will discuss later on in this work. The value of a given state determined by the state-action function is the probability-weighted sum of all action-state values such that:

$$V_\pi(s) = \sum_{a \in A} \pi(a \mid s) Q_\pi(s,a)$$

Using similar mechanics, the action-value function can be determined by taking the sum of all state-values of future states which follow the transition probabilities of arriving in that state given the available actions at s_t such that:

$$Q_{\pi}(s,a) = R_{(a,s)} + \gamma \sum_{s_{t+1} \in S} P(s_{t+1} \mid s,a) V_{\pi}(s_{t+1})$$

Recall equation (2), the argmax procedure for finding the optimal policy π* where π* is equivalent to maximizing the state-value function and the action-state function. We can now reformulate our value functions as Bellman optimality equations such that:

$$V_{*}(s) = max_{a \in A} (R_{s,a} + \gamma \sum_{s_{t+1} \in S} P(s_{t+1} \mid s_t, a_t) V_{*}(s_{t+1}))$$

Likewise, for the action-value function:

$$Q_{*}(s,a) = R_{(a,s)} + \gamma \sum_{s_{t+1} \in S} P(s_{t+1} \mid s,a) max_{a_{t+1} \in A} Q_{*}(s_{t+1}, a_{t+1})$$

The MDP is solved by updating the state-value and action-value functions for every state until convergence. Both the *V*(s)* and *Q*(s, a)* are interchangeable when:

$$V_{*}(s) = \max_{a} Q_{*}(s,a)$$

Which simply means that the maximum expected total discounted reward when starting from state *s* is the maximum *Q(s, a)* over all actions, which touches again on the relationship expressed by equation (2).

Tabular Methods

An MDP with full-knowledge of the environment requires the explicit definition of the transition probability function and reward function. These types of MDPs can be solved using dynamic programming methods. In the event that the MDP does not explicitly define a transition probability function or a reward function, RL methods can be used to solve the MDP, such as Monte-Carlo learning, temporal-difference learning and policy gradient methods. Tabular methods either know the dynamics of an environment or record experiences accumulated by interacting with that environment and reference them *directly* to determine the optimal action to take in each state. Because of this *direct* reference to dynamics or state experiences, tabular methods can only be applied to environments of small dimensionality and where the action and state-space are discrete, otherwise they are no longer tractable. For example, tabular methods can be used by an adventurer as a log system of past experiences in order to determine the optimal route to cross a dangerous terrain. If the adventurer were to analyze a log of every continuous movement they ever took while navigating the terrain, they would become easily overwhelmed if the dimensionality of the terrain was large. Likewise, if they tried to analyze a log of each individual step they ever took when exploring the terrain, they would also be overwhelmed. The adventurer would benefit greatly from simplifying the log. For instance, they could have limited their actionable movements to “up”, “down”, “left” and “right”, while only creating a new log entry every 100 meters or when coming into contact with a point of interest, at which point, they could note whether they were closer to their objective or not. Similar to our adventurer, tabular methods require a certain level of abstraction in order to determine the optimal policy.

Dynamic Programming

Dynamic programming (DP) methods break down a problem into a series of sub-problems to then store each solution to a relational structure. It determines the optimal policy by recursively solving for all possible immediate sub-problems of the final optimal state. In order for this approach to be successful, the problem's optimal solution must be made up of the optimal solutions of all sub-problems and that the number of possible sub-problems is small, with overlapping structure. From our previous example, it assumes the adventurer already possesses a map of the terrain and recursively solves for the optimal path to the start of the journey. As discussed previously, DP requires the MDP to be formulated explicitly with a transition probability function and a reward function.

Solving MDPs using Reinforcement Learning

RL is simply an approach for solving MDPs where the transition probability function or the reward function are not available. We will use this section to transition from traditional MDPs to RL-based solutions by discussing a few particularities of RL and when they are most appropriately used. First, we will briefly discuss concepts such as on-policy vs off-policy learning, the exploration-exploitation dichotomy and model-free vs model-based approaches. Second, we will conclude our discussion on tabular methods but in environments where the transition or reward function are unknown. Third, we will introduce approximate methods. Fourth, we will discuss a more modern approach for formulating an RL-based solution called policy gradient methods. Last, we will tie together both concepts of value function approximation and policy gradient methods to introduce the actor-critic algorithm class.

On-policy vs. Off-policy

On-policy methods attempt to improve upon the policy used to produce decisions, whereas off-policy methods generate observations with a predetermined policy and optimizes for another. In other words, the value function is updated using the action that is expected to optimize the value function in off-policy learning, whereas the value function is updated using the action chosen from the policy that generated the observations in on-policy learning.

Exploration vs Exploitation

The exploration exploitation dichotomy is an important concept in RL. Without prior knowledge of the game dynamics, the agent has to collect experiences that are unlike the experiences it already knows in order for it to learn anything new. We call this process "exploration". In competing fashion, the agent must also be able to execute what it has already learned in order to maximize the reward. We call this process "exploitation". Most methods will typically have a predetermined exploration policy, such as ε-greedy. ε-greedy is an exploration policy where the agent either follows its learned policy with probability $P(\pi) = (1-\varepsilon)$ or chooses an action from the action-state randomly. ε will decrease as the agent interacts with the environment. As ε reduces, exploitation, that is to select the action the highest value estimate according to the learned value function, begins to increasingly guide the agent's decisions (follow more closely the actions producing the highest value function estimate).

Model-free vs Model-based

Model-free and model-based RL relates to the approach used to solve an MDP when both the transition probability and reward functions are unavailable. Model-based reinforcement learning attempts to solve for the MDP by learning the transition probability and reward functions directly and then solving the MDP by applying planning methods, like dynamic programming, to the learned model. The next section will discuss dynamic programming in greater detail. Model-free reinforcement learning is a vein of RL that attempts to learn the optimal policy directly, without necessarily learning a model of the environment. There have been a number of algorithmic innovations in Deep Learning that have made model-free methods much more popular in recent years, where problem sets that were previously deemed to be overly-complex are now tractable with model-free DRL.

Monte-Carlo Learning

Monte-Carlo learning (MC) is a tabular method that can be used when the model of the environment is unknown. MC generates episodic trajectories by exploring the environment until termination. An episodic trajectory refers to a collection of state-action pairs (s, a) accumulated from the first step t to a terminal step. Once the episode terminates, the value of each state visited in the trajectory can be updated such that:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)] \tag{3}$$

Recall, G_t is the discounted cumulative reward of the episode, therefore the value function in equation (3) can only be updated once the episode is complete. α represents the learning rate which dictates the magnitude of the update to the value estimate. MC is a method that can be used when the transition probabilities or reward function of an environment are unknown. Although MC allows us to operate in ambiguity, it tends to be infeasible in problems with large state-spaces and action-spaces. It also fails when episodes are too long or termination is not required. Long episodes are problematic for MC learning because it becomes increasingly difficult to pinpoint which action or series of actions led to a favourable or unfavourable terminal reward.

Temporal Difference Learning

Temporal difference (TD) learning solves the MC drawback of not getting intermittent feedback by breaking up episodes into trajectories of action-state experiences by estimating the future discounted reward and updating the value function for every new trajectory in the table of experiences using a bootstrapping procedure. We define this algorithm as TD(0) such that:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{4}$$

Where α represents the learning rate and $R_{t+1}+\gamma V(s_{t+1})$ is defined as the TD-target in equation (4). The 0 in TD(0) is used to describe the single step moving from t to $t+1$. Experiences, or trajectories, consist of the action taken in a given state along with the resulting reward and future state. This inclusion of the future state and reward creates a

recursive relationship across trajectories and becomes reflected in the value update. This means that estimation of future rewards are obtained from the onset, rather than upon termination or realization of a reward. TD(0) learning leverages bootstrapping, which refers to the update of estimated values for each observation independently, regardless of the ordering of when the observations occurred. It removes the need to rely on sequencing all time-steps in an episode chronologically beyond a time-step's neighbouring steps to learn optimal policies or value functions like in MC.

SARSA vs. Q-Learning

SARSA is an on-policy algorithm that stands for "state-action-reward-state-action" which refers to its trajectory formulation. SARSA was originally published under the name Modified Connectionist Q-Learning (Rummery & Niranjan, 1994). SARSA is an on-policy algorithm and its updates are formulated as such:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{5}$$

In equation (5), notice how the action is selected preemptively without reference to an argmax procedure for selecting the next state's action. This is because the action selected using the (typically) ε-greedy policy is also the action used in the action-value update.

Conversely, Q-Learning is a type of temporal difference learning that uses an off-line policy to generate actions such as ε-greedy, but uses the Q-value produced from selecting

the optimal action for the next state to perform the update. The Q-learning trajectories are (state-action-reward-state). The update for the Q-values can be formulated as such:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (6)$$

In equation (6), the action producing the highest action-value estimate for the next state is selected as part of the TD-target in the Q-function update.

Now, let us revisit our adventurer's dilemma of crossing the terrain by discussing a few problems with solely using logs to determine the optimal path. For instance, what if the adventurer had to cross a very large and complex geographical area? Carrying an extremely detailed set of logs of every experience the adventurer ever had over the terrain, including information about what not to do, would be impractical. Approximate methods can be seen as a parametric mapping of the best route that was derived by learning from past logged experiences, without having to necessarily reference the logs explicitly to know the optimal route.

Approximate Methods

Tabular methods prove to be intractable in large state-spaces. Approximate methods parameterize the learned policy or value function and update incrementally as new observations are encountered by the agent. This greatly alleviates the computational

burden of holding observations in memory, as the learned optimal policy can simply be mapped by either linear or non-linear approximation methods.

Linear Approximation

Linear approximation uses a linearly parameterized model, such as linear regression, to approximate the policy or value function of the agent. Each element within the vectorized state s is represented by x_i , sometimes referred to as the state's features, and has a weight w_i which belongs to the linear model. As the model learns, the weights will shift to better estimate the value or policy, thus reducing the estimation error. The value function is formulated as such:

$$\hat{v}(s, \boldsymbol{w}) = \boldsymbol{w}^T x(s) = \sum_{i=1}^{d} w_i x_i(s). \tag{7}$$

For simplicity, we only show the state-value function as opposed to also showing the action-value function. Linear approximators, such as linear regression, can use a gradient-based method to shift its weights to improve the mapping between its inputs and outputs. Practically speaking, the model will adjust its weights as to more accurately predict the outcome of taking a particular action within a given state. With this learned approximation, the agent will either choose to "exploit" this knowledge by selecting the action which is expected to produce the most favourable future outcome or it will choose to "explore" by following another policy. An interesting property of linear approximation methods is that finding a local optimum close to the global optimum is nearly guaranteed

(Sutton & Barto, 2018). This is a very strong property when comparing linear to non-linear methods, largely because there is no guarantee the local optimum is near a global optimum in non-linear approximation in high-dimensional state spaces. This distinction is important when considering stock trading environments where linear methods are insufficient for approximating the policy or value function (Ning et al., 2018).

Non-Linear Approximation

In problems where the value function is unknown and linear models fail to achieve satisfactory performance, non-linear methods are often used, although their convergence to an optimal policy is not guaranteed. The algorithm used in this work falls under the umbrella of non-linear approximation and it is important to discuss both their advantages and disadvantages. Often, non-linear methods have several hyper-parameters that require significant tuning to achieve satisfactory performance. Further, when policies must be learned in non-stationary game environments, like OOE, non-linear methods often cannot converge fast enough and thus have a tendency to fail on-line (Mnih et al., 2013; Roibu, 2019; Sutton & Barto, 2018). Specifics on the network architecture used in our experiments will be discussed in later sections.

Linear vs Non-Linear Approximation

Whether a model can guarantee convergence to its global optima is not the de facto reason for selecting it as the model best suited for the problem at hand. For instance, take the figures 1 and 2 below:

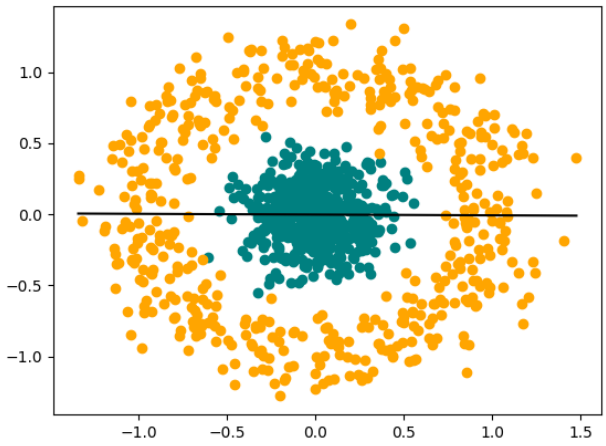

Figure 1: Linear Model Solution for Separating Coloured Points

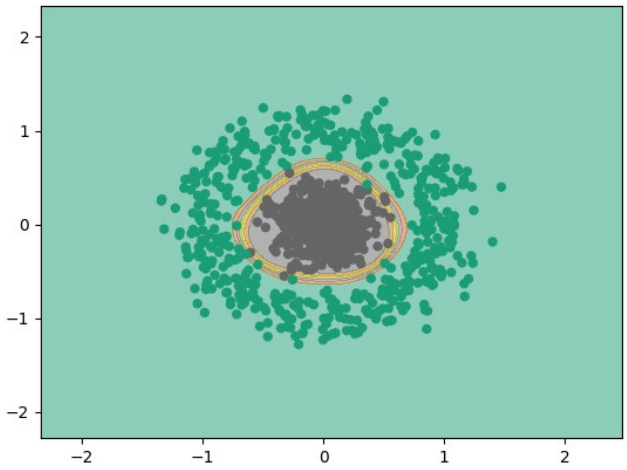

Figure 2: Non-Linear Model Decision Boundaries for Separating Coloured Points

Both figures have the same data points. Figure 1 depicts a linear model's solution when attempting to separate two colours of points. By virtue of the linear model's properties, we have most likely minimized the loss function to the best of the model's abilities. We can know definitively that changing the derivative of the tangent in either direction will either improve or worsen our loss, and if we continue to iterate, eventually the loss will no longer continue to improve. Clearly, the model does a poor job of fitting our data. Figure 2 fits the data using a highly flexible non-linear model, say, a neural network. The neural network does not guarantee a global optimum as is the case for the linear method. It does however, do a better job of fitting the data. This was a simple toy example to demonstrate in what scenarios a linear model is very much sub-optimal, despite guaranteeing converging to a global optimum.

Neural Networks

Neural networks are non-linear function approximators. Their architectures are a series of weights and biases, much like the weights and biases in the linear approximator described above in equation (7), but with the inclusion of a non-linear activation function applied to its output. Below is a simple comparison of the two architectures:

One feature comparison:

linear regression: $b + x_i * w = y_i$

1 neuron neural network: $activation(b + x_i * w) = y_i$

Two feature comparison:

muliple linear regression: $b + x1 * w1 + x2_i * w2 = y_i$

1 neuron neural network: $activation(b + x_i * w1 + x_i * w2) = y_i$

Figure 3: Linear Regression vs 1-Neuron Neural Network

As the simple example in figure 3 demonstrates, the activation function is the central differentiator between the linear and non-linear approximator in their simplest forms. Although the activation function allows us to find non-linear mappings between inputs and outputs while approximating the value-function, it also is the main contributor to neural networks being very prone to over-fitting. Similar to the linear approximator, neural networks must adjust their weights to produce a better mapping between their inputs and outputs. They too use a gradient-based algorithm to do so, with the inclusion of the chain-rule to back-propagate the error through the network to determine the derivative with respect to each weight. This principle is called *Back-propagation* and was first applied to neural networks by Rumelhart, Hinton, and Williams (1986) in their ground-breaking

work *Learning Representations by Back-propagating Errors.* Although we could dive into neural networks by discussing the algorithmic expression of back-propagation, it might stray us too far from our primary purpose. There are two important takeaways from this section: first, neural networks are very similar to linear approximators with the inclusion of activation functions to induce non-linearity and back-propagation to compute the derivatives of the weights, second the only differentiator between a neural network and a deep neural network is the former only has one hidden-layer. A single hidden-layer refers to the function or set of functions that connect the input and outputs directly. Thus, two hidden-layers would mean there are two sets of functions that connect the input and output, where the activated output of the first set of functions becomes the input of the second set of functions. When we refer to a "deep" neural network, we are simply stating that the architecture has at least two hidden-layers. DRL is simply RL that uses some variant of a neural network with more than one hidden-layer as its nonlinear function approximator.

Policy Gradients

Unlike typical action-value methods, which refer to a value function to choose actions, policy gradient methods use a parametrized policy. Rather than minimizing a value loss, policy gradient methods attempt to maximize performance by computing learned probabilities of each action in the discrete case, and probability distributions in the continuous case (Sutton & Barto, 2018). There are several reasons why policy gradient methods tend to outperform value function methods:

1. Value-function methods are simplest in discrete action spaces, which is impractical for OOE. Having probability distributions to determine preferred actions produces less erratic changes to the gradient-based updates of the non-linear approximator.
2. Policy gradient methods can learn appropriate levels of explorations, rather than using deterministic ones, like ε-greedy, that may not follow an appropriate search schedule for the problem at hand.
3. Policy gradient methods do not require implementing a complicated value function, such as the Bellman equation.

Actor-Critic Methods

The actor-critic method, the RL modelling type in which TD3 falls under, leverages both policy gradients and value function approximation. The architecture consists of an *actor*, a separate network that uses state information to produce an optimal action. Actions taken by the actor are then provided to the critic along with the state. The critic then produces a value estimate which is then used to compute the gradient for the *critic* network. The actor is updated using the resulting reward determined by the critic. Actor-critic algorithms are discussed in great detail, both theoretically and practically, when the DDPG and TD3 are introduced in later sections.

Deep Reinforcement Learning for Optimal Order Execution

Much of the recent revival in reinforcement learning can be attributed to several ground-breaking contributions to the field of deep reinforcement learning, where numerous adaptations in algorithmic design have enabled agents to learn policies directly from sensor-like data (Doria, Dawson, and Vindiola 2015; Mnih et al. 2013; Moreno-Vera 2019; Roibu 2019; Zhai et al. 2016). Sensor-like data refers to computer-generated information that is not altered or pre-processed by humans prior to training the model. For example, pixel-data used to render every frame of a video game can be used as the input to an approximator that will learn to map the sensor-like input and action to a reward or value estimate. Drawing on the knowledge we gained from our MDP and RL introduction, each frame, or several consecutive stacked frames, define the state of our trajectory. The agent's actions are the available moves in the video game. The reward can be events from the video game itself, such as the loss of a game "life" or beating the computer opponent at a task. When we refer to the next state, we would simply collect the resulting next frame or set of frames after performing an action. In the context of this work, these works are leveraged to understand how model-free reinforcement learning can be extended to single-side stock trading.

First, this section will cover the first known example of DRL successfully using sensor-like data as input to learn policies able to achieve human-level performance in the Atari arcade games using experience replay (Mnih et al. 2013). The second work discussed introduces a method to reduce over-estimation in the value function (van Hasselt, Guez, and Silver 2015). Extensions to continuous action-spaces were made via the

advent of the Deep Deterministic Policy Gradient method which was a departure from depending solely on value function learning but suffered from similar over-estimation in the network's value estimation (Lillicrap et al. 2015). Finally, Twin Delayed Deep Deterministic Policy Gradient (TD3) by Fujimoto (2018) will be discussed as this is the primary algorithm used in this work due to it circumventing many of the limitations discussed from the previous works.

Deep-Q Learning (DQN)

DQN with experience replay was the first example of model-free reinforcement learning leveraging solely sensor-like data to achieve human-level game play performance on the famous Atari arcade games (Mnih et al., 2013). This paper was so successful that the arcade games are now used extensively in the field of DRL as experimental environments (Doria et al., 2015; Mnih et al., 2013; Moreno-Vera, 2019; Roibu, 2019; Zhai et al., 2016). It is important to note that Mnih et al. (2013) were not the first researchers to introduce reinforcement learning in conjunction with non-linear approximation using neural networks (Tesauro, 1995), but several pivotal improvements were made to the algorithm that led to its impressive performance gains and thus is largely responsible for putting DRL in the forefront of modern artificial intelligence research.

The mechanics of DQN (Deep Q-Network), are straight forward and build off the basics of Q-learning that were discussed previously, with the exception that it uses a deep convolutional architecture, which is a type of deep neural network that fairs particularly well in image recognition tasks. The DQN variant used in the original Atari paper

introduces three modifications which will be discussed in great detail later in this section (Mnih et al., 2013). The first modification by the authors is how they use a target network, θ^-, while training. The target network is a duplicated set of parameters to the action-value network, θ, but updated less frequently as to solve the common occurrence of chasing a moving target in Q-learning. Q-learning is said to chase a moving target because it estimates the action-value function and the target using the same set of parameters. By having a fixed set of parameters that is separate from the training network, DQN can be run on-line, despite being an off-policy algorithm. The second modification is the addition of an experience replay buffer, which helps decorrelate trajectories and stabilizes learning. The third modification was in how the authors stacked four consecutive frames as a single state for two reasons: it decorrelated the states by reducing the likelihood of no movement taking place from one state to another and it provides information to the model about *how* things are moving in the game across frames. Similar to Q-learning, the optimal action-value function for the DQN is defined as the best possible future expected return by following a policy π mapping sequence *s* to actions *a.* The following equation defines the gradient update for the DQN:

$$\Delta\theta = \alpha[(r + \gamma max_a \hat{Q}(s_{t+1}, a_t, \theta^-) - \hat{Q}(s_t, a_t, \theta))]\nabla_w \hat{Q}(s_t, a_t, \theta)$$
$$\text{At every } T \text{ steps}:$$
$$\theta^- \leftarrow \theta \tag{8}$$

Where $r + \gamma max\, \hat{Q}(s_{t+1}, a_{t+1}, \theta_{i-1})$ is the TD-target and $Q(s, a, \theta_i)$ represents the Q-value prediction associated with being in the current state *s* and committing action *a* in that same state. Recall from equation (6) the TD-target and action-value functions, they

are similar to those of equation (8), with the inclusion of the network weights referenced as θ as well as the target weights referenced as θ^-. The gradient is computed by finding the change in the weights of the network due to the value-estimation error between what the action-value network anticipated the future discounted reward to be and the target network's value estimate of the future next state.

Experience Replay

One of the chief reasons the DQN was so successful was due to the introduction of a mechanism called experience replay using a replay buffer. A replay buffer serves as a storage of experiences accumulated by the agent as it iteratively interacts with the environment. The parameters of the original DQN are updated using gradient descent. The replay buffer benefits the network in two ways: first, it enables learning from the same experience several times, second, it decorrelates learning from consecutive states and thus stabilizes training. These two improvements were previously considered as major hurdles in the application of DRL. Using some variant of a replay buffer is now included in nearly all subsequent implementations of DRL because of its consistent impact on performance.

Delayed Target Network Updates

Delaying the target network update is beneficial for several reasons. It allows for experimentation in the value-function without affecting the real-time performance of the target network, the model *learns* off-line but can be implemented in real-time where updates happen only after a certain number of additional experiences have been accrued in

the replay buffer. This stabilizes performance by preventing the target network from being overly responsive to sudden changes in the environment.

Pseudo Code

In order to better outline the DQN algorithm's internal workings, pseudo code is provided (Mnih et al., 2013). The pseudo-code is from the original DQN paper, although we recognize some customs have changed since its conception. We have attempted to keep the concepts from the original pseudo-code in subsequent algorithm definitions in hopes of maintaining linearity and consistency across explanations. We recognize that more current examples of pseudo-code may have slight differences, however, we build on the original works and make subsequent changes to pseudo-code from more recent works to demonstrate algorithmic improvements more explicitly that follow the original conventions.

Deep Q-Learning with Experience Replay in Atari Experiments

(1) Initialize replay memory D to capacity N
(2) Initialize action-value function Q with random weights θ
(3) Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
(4) For episode = 1, M ***do***:
  (5) Initialize sequence $s_1 = \{x_1\}$ and preprocess sequence $\phi_1 = \phi(s_1)$
  (6) For t = 1, T ***do***
    (7) With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
    (8) Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    (9) Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    (10) Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    (11) Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    (12) Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    (13) Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to network parameter θ
    (14) Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 4: Deep Q-Learning with Experience Replay

Take note of the first three lines from the DQN's pseudo-code in figure 4; the replay buffer is defined along with an observation count limit N, then the value-function is initialized randomly and then copied over to the target network. On line 5, the first state is derived from pre-processing the current features, where ϕ represents the pre-processing function, such as the transformation performed to stack four consecutive frames to create a single state. Line 6 begins the episode. On line 7, the first transition is created by selecting an action that either maximizes the estimated value of the state or, if a randomly generated number between 0 and 1 falls below 1-ε, then randomly selects an action. Keep in mind that ε will decay as the algorithm continues to interact with the environment. This decay implies that random exploration of the action-space occurs less and less frequently as training progresses. Recall, this is the concept of trade-off between exploration and exploitation in action, which was discussed in previous sections. Line 8, describes the

retrieval of the reward from executing that action in that state and on line 9, pre-process the resulting state in order to create a trajectory to then store in the replay buffer on line 10. Once there exists a minimum number of transitions in the replay buffer, line 11 randomly samples a mini-batch and line 12 produces a value estimation of each transition's initial state using the value-function if it is non-terminal. Line 13 computes the gradient by finding the squared loss between the estimation of the value of the past state with the target's value estimation of the next state. Finally, on line 14, the newly learned weights from the value-function network are copied over to the target network after a predetermined number of steps.

It is important to note that the same network which produces the estimate for valuing the actions, which subsequently chooses the action which produces the highest value estimate of the future next state, is also used to derive the gradient from the value-estimation error which is then used to update the target network. In other words, the same target network that selects *a'* also values its selection and is used for computing the gradient update. This tends to lead to the target being overly optimistic in assessing the value of a particular action-state pair, and is amplified in highly stochastic environments where action-state outcomes are highly variable. It is as though the network rewards itself for selecting what it believes to be the best action, which welcomes counterproductive learning behaviour, such as being overly optimistic about the value of an action-state pair and slowing convergence by muddying the gradient update with this over-optimism.

Double Q-Learning (DDQN)

As previously mentioned, there were a few significant drawbacks with the initial implementation of DQN that could be improved upon, such as the tendency to over-estimate the value of a particular action. The difference between DQN and DDQN is in the estimation of the Q-values of the resulting state using the target network. Recall, in the DQN pseudo-code, the target network that evaluated the current state's best action is the same network of weights that were copied over from the action-value function. This leads to an overestimation of the target y and thus is slower to converge and produces poorer results (van Hasselt, Guez, and Silver 2015). In DDQN, the target network is initialized using a different set of weights and is subsequently updated using soft-updates, not copied directly from the action-value function. This decouples the networks completely and essentially trains two completely different networks tasked with evaluating the value of the current state to select an action and evaluating the value of the resulting state and computing the loss from the two.

Algorithmic Differences between DQN and DDQN

Double Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta'' = \theta'$ *
For episode = 1, M ***do***:
  Initialize sequence $s_1 = \{x_1\}$ and preprocess sequence $\phi_1 = \phi(s_1)$
  For t = 1, T ***do***
    With probability ϵ select a random action α_t
    otherwise select $\alpha_t = argmax_a Q(\phi(s_t), a; \theta)$
    Execute action α_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \hat{Q}(\phi_{j+1}, a_j; \theta') \text{ **} & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to network parameter θ
    Update target network parameters $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$ ***
  End For
End For

** Entirely new weights are initialized for the value evaluation. Experiences are randomly assigned to either value function for updating, thereby creating two separate learned networks.*
** Although the action selection remains the same as in the DQN, the value function is decoupled and therefor the max operation can be removed.
*** Soft-updates to the target network occur at every step rather than direct copies of the action-value function.

Figure 5: Double Q-Learning with Experience Replay

Pseudo-code from the DQN paper is shown in figure 5, with updates from the DDQN in red (van Hasselt, Guez, and Silver 2015). The first change signifies a completely new set of weights to initialize the target network as opposed to a direct copy of the value-action network. Next, the target network is used to evaluate the next state's value using the action chosen by the action-value function. The gradient is computed by determining the loss between what the target network's value of the next state was given the action chosen by the action-value function and the value placed on the previous state by the action-value

function. Updates to the target networks are made using soft-updates at every step, rather than complete copies as was the case for the DQN.

Deep Deterministic Policy Gradients (DDPG)

DDPG was the policy gradient equivalent of the DQN algorithm as it used sensor-like data and a model-free approach to find an optimal policy in the Atari arcade games necessitating a continuous action-space (Lillicrap et al., 2015). The DDPG is an *actor-critic* method because it uses both a value-function (critic) and a policy gradient approach (actor) in its architecture. The implications for the inclusion of the actor are numerous :

1. Decoupling the action from the value estimate gives the flexibility of using policy gradient methods for the policy network (actor) and value-function methods for the critic.

2. Using policy gradient methods for action selection gives us the ability to operate in a continuous action-space, unlike value-function methods which strictly need a distinct action to estimate the value of the action-state pair, a task that is otherwise computationally infeasible in continuous action spaces.

3. Decoupling the networks also means the two can follow different update schedules or learning rates which can add more fine-control during training.

4. Continuous action spaces allow for smoother network updates because value estimates can be gradual, as opposed to possibly large updates due to sharp changes in value estimates from discrete actions.

Pseudo Code

Deep Deterministic Policy Gradients

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, **M do**
  Initialize a random process N for action exploration
  Receive initial observation state s_1
  for t=1, T ***do***
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta_{\mu'})|\theta^{Q'})$
    Update critic by miniminizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 6: Deep Deterministic Policy Gradients

We show DDPG pseudo-code from the original paper in Figure 6 (Lillicrap et al., 2015). Notice the inclusion of the actor network and its respective target network on the first two lines. Like DQN and DDQN, DDPG's authors included experience replay which helps decorrelate experiences and stabilizes learning. Note how the target networks are updated at every time step using 'soft-updates' dictated by τ, where larger values will ensure smaller updates. Further, notice the soft-updates are performed on both the actor and the critic networks.

Twin Delayed Deep Deterministic Policy Gradient (TD3)

The TD3 algorithm is nearly identical to that of the DDPG. Three modifications were introduced, however, to better address approximation errors in the DDPG algorithm (Fujimoto, 2018).

Architectural Comparison with DDPG

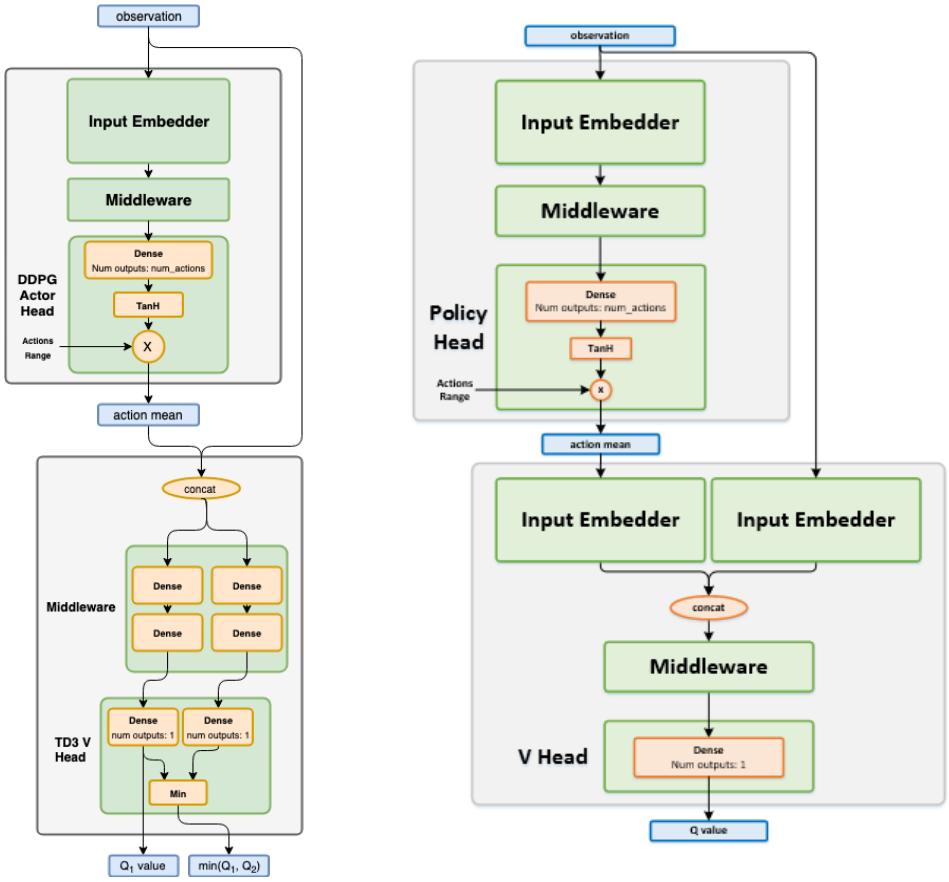

Figure 7: TD3 (left) vs DDPG (right) Architectures

Figure 7 is a visual representation of the architectural differences between TD3 and DDPG (Caspi et al., 2019). Notice they are virtually identical except for the special middle-ware and head.

Twin Delayed Deep Deterministic Policy Gradients (TD3)

Randomly initialize critic network s $Q(s,a|\theta_1^Q)$ and $Q(s,a|\theta_2^Q)$
Randomly initialize actor network $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target networks Q'_1, Q'_2 and μ' with weights $\theta^{Q'_1} \leftarrow \theta^{Q_1}$, $\theta^{Q'_2} \leftarrow \theta^{Q_2}$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, **M do**
  Initialize a random process N for action exploration
  Receive initial observation state s_1
  for t=1, T ***do***
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    $\tilde{a} \leftarrow \pi_{\phi'}(s) + \epsilon, \ \epsilon \sim clip(N(0, \tilde{\sigma}), -c, c)$*
    Set $y_i = r_i + \gamma min_{k=1,2} Q'_k(s_{i+1}, \tilde{a})$**
    if t mod d then ***
      Update critic by miniminizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
      Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

      Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

* *Target policy smoothing*
** *Clipped double Q-learning*
*** Delayed update of target and policy networks

Figure 8: Twin Delayed Deep Deterministic Policy Gradients (TD3)

Pseudo Code

In order to highlight TD3's algorithmic improvements over DDPG, pseudo-code is provided in Figure 8 (Fujimoto, 2018). The improvements are written in red. In all, these three modifications can be summarized as follows:

1. Smooth the action selection of the target network.
    - This modification injects noise into the agent's actions for continuous exploration and thus produces a more varied replay buffer.
2. Select the minimum value estimate produced by the critics.
    - This modification reduces over-estimation in the value estimate of the action. Recall the impact that selecting the maximum value estimate can have on the model's performance as seen in DQN/DDQN. Likewise, taking the more conservative estimate of the two critics reduces the likelihood of over responsiveness from the actor and target networks.
3. Delay the update for actor and target networks.
    - This modification stabilizes learning by not allowing the actor and targets to respond too drastically to noise in the critic's estimation.

Optimal Order Execution using Reinforcement Learning

There exists a select few works in the OOE body of research that use RL, but few have applied many of the recent advances discussed earlier in this thesis. Important to note, we have chosen to simulate data to understand precisely if and when the algorithms converge to the known optimal policy for bench-marking purposes. In the case of a Martingale price process, this is to follow a TWAP schedule, that is, a Time-Weighted

Average Price schedule, where identically sized market orders are submitted at equidistant time intervals apart. In the real-world, the true price process is unknown, therefore we would like to understand the effectiveness of model-free reinforcement learning in approximating these types of problems. By construction, these approaches are not given the model-dynamics, making them potentially good candidates for real-world applications where model-dynamics are unknown. The problem setup for this thesis was very much inspired by (Ning et al., 2018) but diverges away from their work in regard to algorithm choice and minor changes in game setup and data. We have dedicated a section, called "Comparison with Ning et al. 2018" describing key differences between both works just before the results and analysis section.

Setup

Naturally, model-based, such as DP, could potentially outperform model-free methods when the price process is artificially generated and therefore known, but this would not shed light on the central question of this thesis of whether model-free reinforcement learning could be applied in practice to OOE when model-dynamics are unknown and model-based methods are inapplicable. To emulate a real-world scenario, the model-free problem is constructed as such: The agent is given 100,000 shares at the beginning of an episode lasting 1 hour. The agent must find the optimal sell-off schedule that maximizes its return by choosing how much remaining inventory to sell at the start of every step, of which there are 5, each lasting for a period of 12-minutes. The amount of inventory decided upon at each step will be executed as equal-sized market orders every second over the course of the 12 minutes, in other words, the agent will submit 720 equidistant orders every step. This forces the agent to submit market orders that are

subject to price fluctuations during this time. The returns are compared to that of a TWAP schedule, where no decisions can be taken at the beginning of the episode. This is equivalent to selling ~28 shares every second for 1 hour, subject to all price fluctuations during that time. In the event of the agent not fully liquidating before the end of the episode, we enforce full terminal liquidation.

Evaluation Metric and Reward

The agent's reward is the relative profit and loss (P&L) and is used to assess agent performance. It is returned to the agent after each episode as the accumulated reward incurred over the course of the episode. P&L is considered relative because it is the performance in terms of profit when compared to the profit generated from the TWAP baseline strategy:

$$P\&L = \sum_{k=0}^{N-1} \sum_{i=0}^{M_k-1} \left(x_{t_{k,i}} p_{t_{k,i}} - z\left(\frac{x_{T_k}}{M_k}\right)^2 \right). \tag{9}$$

P&L for each observation is contingent on the aggregated step-wise return from liquidating stock inventory at the going market price. An observation is defined as a state-action pair (*s, a*). *N* represents the total number of steps in an episode, k is the step identifier, i represents the second-level time-increment identifier and M represents the total number of time-increments (number of seconds) per step *k*. Like Ning et al. (2018), we define $\frac{x_{T_k}}{M_k}$ as the total number of shares sold in step *k* over the total number of

seconds M in step k, whereas $x_{t_{k,i}}$ is the amount of inventory sold at every second over the course of step k, multiplied by $p_{t_{k,i}}$, which is the corresponding price of the asset at every second over the course of step k. Please refer to the section "Problem" earlier in this work to revisit these concepts if needed. The expression $-z\left(\frac{x_{T_k}}{M_k}\right)^2$ in Equation 9 represents the quadratic penalty, where z is the penalty coefficient. Large orders are clearly penalized as this term is subtracted from the brute P&L before calculating the relative P&L. The use of the quadratic penalty is also borrowed from Ning et al. (2018) and simulates market impact and transaction costs incurred from large orders.

Recall, the OOE problem is a single-side trading problem so the return will always be positive relative to profit at the beginning of the episode. We found that using brute profit as a reward was slower to converge likely because the brute profit is largely dependent on the value of the state which can add unnecessary complexity to the task of the agent. For example, it is more difficult to discern which action was truly optimal when the price process was trending upwards and any variation of actions over the course of the episode will lead to higher profits by virtue that we are only selling inventory and not buying inventory. This problem of action-indifference being overlooked is a very real problem in traditional value function methods. The action-value function is the sum of two components: the *advantage* and the value estimate. *Advantage* refers to the value of taking that action over all other possible actions. It can be used to decouple the value of the state from the action-value function to produce a measure of value for an action over all other

possible action. By doing this, we could have potentially mitigated the issue of action-indifference (Schulman et al., 2018).

Rather than modifying the TD3 algorithm, we opted to compare its return to the baseline TWAP by recursively bootstrapping the last terminal state's relative P&L and limiting the number of steps in the episode. Because relative P&L is only meaningful at the end of an episode, we chose to produce sparse rewards by delaying the reward until the terminal state where intermittent steps produce no reward signal. If we did not delay the reward, the algorithm would not be able to operate on-line because of the need to wait until the end of the episode to know relative P&L despite needing to take actions throughout the episode, which would defeat the purpose of wanting to assess whether TD3 could be appropriate for live trading. Intermittent relative P&L does not encourage the right behaviour from the agent. For example, take an episode where the price increases in the last few steps, selling most inventory on the first step would produce a very high relative P&L on step 1 and small relative P&L subsequently. Conversely, if the price were to decrease slightly, the same strategy would produce very high relative P&L on step 1 and small relative P&L subsequently. Since TWAP is also measured at the same prices throughout the episode, the relative P&L is the same in either case, despite there being completely opposite changes in the price; This is definitely not the type of reward structure we are looking for. We also found that awarding brute P&L at the end of every step was adding unnecessary noise to the reward because of major price differences across episodes. We could have standardized the P&L to the price at the beginning of the episode to compute the reward, but instead, we chose to aggregate the brute P&L throughout the episode and compute the relative P&L using Equation 9 at termination. Using 5 steps was

found to allow for enough flexibility to regularly beat the TWAP baseline by adjusting inventory amounts regularly but was short enough to allow for the reward signal's efficient recursion across the bootstrapped trajectories upon updating the policy and value estimates.

Independence Assumption and the Quadratic Penalty

In this version of OOE, we assume action-state independence across time steps. In other words, the agent has no impact on the market beyond its immediate step's action. This assumption is supported by one of the first works using RL to address the OOE problem (Dempster & Leemans, 2006). The authors demonstrate that the price impact on the order book of a large executed trade is realized immediately, but reverts to the expected price shortly after, had the trade never occurred. This implies that market participants do not stray from their strategy based on the strategy of one participant. Although the authors test using upwards of 1M shares over 2 to 8 minutes on highly liquid stocks, one would be tempted to question if the assumption still holds today, nearly 14 years after the study was published and before high-frequency trading became the industry norm for executing orders. To mitigate this risk, this implementation's version of OOE limits the number of shares to 100,000 over the course of 1 hour, an amount that is not likely to impact the strategies of competing market participants in the case of highly liquid stocks. The quadratic penalty serves as the immediate impact on the price and is revised after 720 seconds of trading.

Transaction Costs

The quadratic penalty we use from equation (10) also serves as a way to account for transaction costs. Submitting large orders will incur large per share fees. This means that the quadratic penalty serves 2 purposes: to reflect the impact the size of an order will have on the order book immediately after an action and the per share transaction costs incurred from submitting large orders.

Hypotheses

1. The TD3 converges to the optimal policy, which is a steady-sell off TWAP schedule when the price process is that of a Martingale which will be explained in great detail in the next section. We consider this to be the first experiment.

2. When we bias the Martingale price process by skewing the distribution from which its price movements are drawn from, we hypothesize the TD3 algorithm will outperform the DDPG algorithm and consistently surpass the baseline TWAP. Three different degrees of skewness are tested, we consider these as 3 separate experiments for the 2nd hypothesis. We believe TD3 will surpass TWAP because, with the inclusion of a bias in the price process, we open up the possibility for the algorithms to exploit a pattern. Exploitable patterns are used to beat the baseline steady-sell of strategy, without them, the price process is a Martingale and the algorithm hypothetically should converge to TWAP or below TWAP, because of the loss incurred from not being able to exploit the given bias.

Methodology

Data

We simulate price data by initiating an independent process over the course of 250 trading days. This implies that volumes sold do not directly affect the price process. We explain this assumption in the previous sections describing the independence assumption and the quadratic penalty. Four trading hours are accounted for each day. Each hour consists of 5 steps. Each step is made up of 720 seconds of prices. The total amount for each price process is roughly 3.6M price points, 5k price points over the course of each 12 minute trajectory.

These price points can be seen as rolled-up (forward-filled) tick data where the last price is used at the end of every second-level time increment. Tick data is a term used in finance to represent the transaction-like data produced by each newly submitted quote or trade on the market. Tick data is event driven, not time-driven. To 'roll-up' tick data, which is a term also used in finance, means to take the last available price and forward-filling it to equidistant time intervals. For instance, a 1 second roll-up for the first three seconds of a minute where $s \in [0, 1, 2]$ would mean to take the last available price between: the previous minute's 59^{th} second and the start of the current minute where $s = 0$, the start of the current minute $s = 0$ and the beginning of the first second $s = 1$, and finally from the start of the first completed second $s = 1$ to the start of the second $s = 2$. It is a technique used to standardize stock data to time, which is helpful when wanting to reduce the size of the available data or to standardize episodes and steps in reinforcement learning environments that may be time-sensitive (i.e. the agent must liquidate all inventory after

60 minutes of trading). This also helps simplify the problem-space by restricting the agent to a fixed number of decisions and therefore given equal amounts of information across all time-intervals.

Brownian Motion as a Price Process for Order Execution

A Brownian motion (BM) or Wiener process, is a stochastic process derived from approximating the movements of a random walk using the central limit theorem. Different variants of BM are often used in finance to generate stock price processes. For instance, Almgren & Chriss (2001) use arithmetic Brownian random walk with zero drift to generate their price processes for their work in OOE given the high-frequency of the data in the OOE problem. Conversely, Geometric Brownian motion (GBM) is a well studied proxy for modelling the price movements of stocks in medium to long-term stock trading applications (Gatheral & Schied, 2011; Reddy & Clinton, 2016). GBM takes into account a drift parameter and also enforces a non-negativity constraint on the process, which is desirable when modelling stock prices over the medium and long-term. Almgren & Chriss (2001) argue that the geometric variant should be used in highly volatile or long-term trading settings but that in short-term trading, the differences between the two are negligible. We decide to generate our initial price process in the same manner as Almgren & Chriss (2001) since we use a 1-hour time-horizon and the model is exposed only to relative price changes rather than brute prices, as will be discussed further in the feature engineering section of this thesis. Further, our reward of P&L is relative to TWAP and so also is unchanged by adopting this type of price process.

When a stock's price fluctuates independently of its historical behaviour and possesses no drift, where each step's fluctuation from the starting price falls within a normal distribution, it supports the *Efficient Market Hypothesis,* that in a fair and competitive market with symmetrical access to information, future price is independent of historical price (Malkiel & Fama, 1970). When there is no drift added to this process, where the expected price of a stock is equivalent to its current price, it is referred to as a *Martingale*.

This thesis will use four variants of arithmetic Brownian motion with zero drift: one without bias (a martingale) and three with varying degrees of skew in how the error term is distributed. To compute the price process, we use Scipy, a popular Python library for scientific applications. The formula for Brownian motion from the Scipy documentation (2017) (*Brownian Motion—SciPy Cookbook documentation*, 2017) is as follows:

$$X(0) = X_0$$

$$X_{t+dt} = X(t) + N(0, (\delta)^2 dt; t, t+dt)$$

Where $N(a, b; t_1, t_2)$ is a normally distributed random variable with mean a and variance b. The parameters t_1, t_2 are two sequential moments in time and are predetermined. This means that from every step to its corresponding next step, the change in x is solely determined by the variance, because the expected value of x at the next time-step is simply x. This implies that, if we assume movements in x follow a normal distribution, no historical information of x is relevant for predicting x, except for the current price of x. Despite the wide-scale use of BM to represent stock prices, several researchers have postulated that short-term price movements have a tendency to not

follow a normal distribution (Dhesi et al., 2012; Fama, 1965; Imperial, 2018). For this reason, we introduce varying degrees of slight bias to the normally-distributed movements of the BM process to understand the model's ability to quickly detect and exploit such discrepancies. We bias the normal distribution by using the skew-normal distribution, which introduces a parameter α that serves as a regulator for skewness where α=0 is a normal distribution without bias (Azzalini & Capitanio, 1999). The standard normal distribution is defined as:

$$PDF\ of\ N(0,1) = \phi(x) = \frac{1}{\sqrt{2\pi}} e^{\frac{-1}{2}x^2}$$

$$CDF\ of\ N(0,1) = \Phi(x) = \int_{-\infty}^{x} \phi(t)\, dt$$

We can then generate the skew-normal distribution by introducing the parameter α:

$$PDF\ SN(x) = 2\,\phi(x)\,\Phi(ax) \tag{10}$$

We ignore the location and scale parameters in our experiments, which can be included in alternative versions of the skew-normal distribution, and so we do not define them in equation (10) for simplicity. The first publication of using Azzalini's skew-normal distribution to bias the price movements in a Brownian motion price process for a financial instrument was authored by (Eling et al., 2010). The authors demonstrate that Azzalini's skew-normal distribution outperformed classical skewness coefficients when performing goodness-of-fit tests on hedge fund returns. Several works have leveraged the use of skew-normal distribution when generating a price process with Brownian motion since (Doostparast, 2017; Maeda & Jacka, 2018; Zhu & He, 2018). The 2nd, 3rd and 4th experiments will use a skewness parameter of .1%, 1% and 10% respectively, all of which

are very small skews when compared to most of the previously mentioned works. We provide examples of the probability distributions of different skewness parameters for reference in Figure 9.

As mentioned previously, the intent of this work is to determine the overall efficiency and proficiency of the TD3 algorithm to identify and exploit very slight bias in a noisy price process. In the context of OOE, if the price process is a Martingale without skew-normal bias such that α=0 as depicted in Figure 9, the optimal policy would be to do a steady sell-off of inventory at every time step, that is, following a steady TWAP strategy. For a refresher, we ask that you refer to previous sections of this thesis, namely where the TWAP strategy was described in great detail with examples.

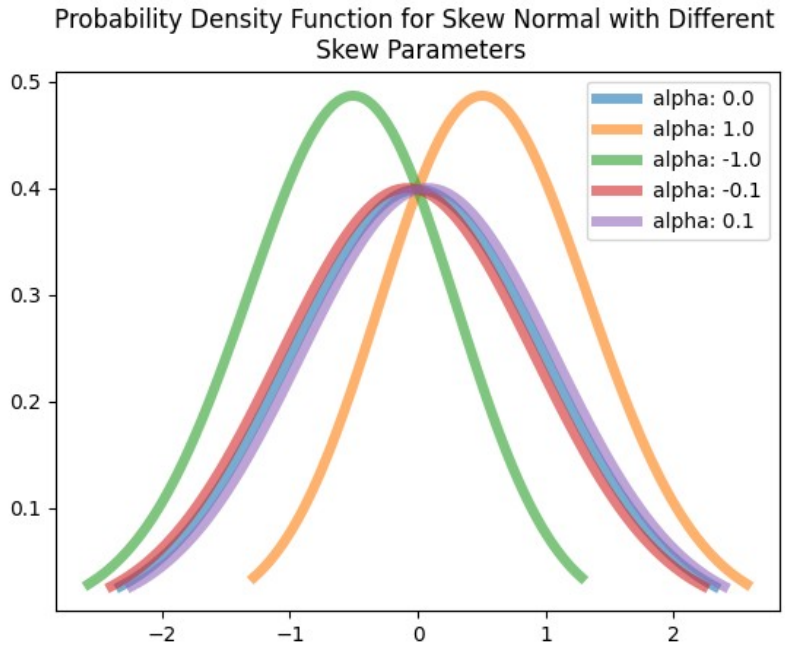

Figure 9: Probability Density Function of Skew-Normal with Different Skew Parameters

Technical Specifications

The agents were trained using an Ubuntu virtual machine with 32 GB of RAM with 500 GB of disk space. Python 3.6 was used for all experiments. Python 2.7 was used for visualization purposes due to incompatibility issues with certain libraries. All software, libraries and agents used are open-source and belong to the public domain to at least the extent to which this thesis uses them.

Feature Engineering

Several features were derived from the data of each respective simulated stock. The data was "rolled" up to second increments to simulate a standardized time scale, meaning features were derived temporally from the last simulated price of each second. Tick data is event driven and must be transformed to know when decision breaks should occur at the start of each step. This 'fill-forward' procedure was explained in detail with examples in the section of the thesis describing the data.

Raw Features

- Last price: The last traded price of the instrument. Roll-up tick prices at second-level time increments.
- Step: identifier for step number
- Observation: identifier for each collection of 5 steps of price for each day.

Derived from Raw Features

- Scaled Price: The last price is scaled between [-1, 1] by subtracting the mean and dividing by 2 standard deviations computed from the historical 'last price' from the previous step.

- Remaining Inventory: The amount of inventory remaining at each step. This is determined during game-play. It is obtained by dividing the remaining inventory by the starting maximum inventory to ensure the value represents a proportion.

- Time Remaining: The amount of time remaining upon entering each step. This is determined during game-play. It is obtained by dividing the remaining number of steps by the total maximum number of steps in each hour to ensure the value represents a proportion.

- Price Transformed p: the original price is transformed to fall between -1 and 1. The price at the beginning of the hour is subtracted from every price increment. This means that only extreme outliers fall outside the given range. This approach was directly taken from (Ning, Ling, and Jaimungal 2018).

- Quadratic Variation: Similar to Ning et al. (2018), we use quadratic variation to model the degree of variability in price movements from one step to the next.

  Quadratic Variation can be modeled as such:

$$QV_{T_k} = \sum_{i=0}^{M_{k-1}-1} \left(p_{t_{k-1,i}} - p_{t_{k-1,i-1}}\right)^2, \quad \forall k \in 1,\ldots,N-1$$

Where T represents the hour, k represents the step and i represents the second-level time increment.

Algorithm Selection

The most promising preliminary results were obtained with the TD3 and DDPG algorithms (when the problem was given a continuous action space) over DQN and DDQN (when the action space was discrete). Discretizing the action-space refers to separating a continuous action space into non-ordinal classes. This is consistent with other works related to applying DRL in trading-type tasks, where policy gradient methods tend to outperform purely value-based methods in an online setting (Dempster and Leemans 2006; Deng et al. 2015, 2017; Moody and Saffell 2001). For this reason, we chose to stay with algorithm types that can operate in continuous action-spaces.

Incremental performance of the target network is recorded to demonstrate the agent's ability to learn to generalize and operate in an on-line setting. Minor changes were made to the author's original implementation in Pytorch and are detailed in later sections (Fujimoto, 2018). For tractability, the agent's actions are limited between [-1, 1] using a *tanh* function while training. The output is then scaled between [0, 1] for human interpretation to determine the percentage of inventory the agent wishes to liquidate at that step. In order to obtain the relative P&L of that action-state pair, we convert the percentage to unit-volume x in Equation 9.

Training

Experiments

Four experiments were conducted, each with a simulated stock using a different price process. For the first experiment, TD3 is compared to baseline TWAP. The 2nd , 3rd and 4th experiments compare TD3 and DDPG to baseline TWAP. An account of how each agent was trained is given in the subsections detailing each experiment. Observations from each experiment during training are noted for each figure. We provide an example of 10 episodes of standardized price processes under the different biases in Figures 10, 13 and 15. We show the inventory sell-off schedule changes from experiment 1 in Tables 3 and 4. We report relative P&L of the test results over the course of training in Figures 11, 12 and 14, where the x-axis at y=0 is baseline TWAP. We also provide summary statistics for returns in Tables 2, 5, 6 and 7.

Hyper-parameters

Training settings were taken from the original TD3 paper. All experiments were conducted with the following parameters for both TD3 and DDPG where applicable:

#	Name	Value	Definition
1	Seed	0	The seed at which all stochastic processes will rely on.
2	Start Time-steps	100	The number of initial steps taken before training begins. Used to collected varied observations.
3	Eval Frequency	100	The number of training steps that occur before the target network is evaluated on the test set.
4	Max Steps	10,000	Maximum number of training steps taken.
5	Exploration Noise	0.1	Level of noise added to the agent's action.
6	Batch Size	100	Number of instances included in each mini-batch during training.
7	Discount	0.99	The rate at which future rewards are discounted. Higher puts more emphasis on future returns.
8	Tau	0.0005	Target update rate. Soft updates of target weights are used to encourage steady and gradual learning.
9	Policy Noise	0.95	Level of noise added to the target network's actions.
10	Noise Clip	0.5	Range (+/-) to restrict policy noise.
11	Policy Frequency	2	Number of times the critic networks are updated compared to the actor.

Table 1: List of TD3 Hyper-parameters

Pre-training

Pre-training is done to introduce more variation in the training data and to expose the algorithms to fringe cases early on. Observations are stored, 100 in the case of our implementation, by interacting with the environment using randomly sampled actions. We can see this as an off-line warm-up of 100 steps before allowing the target network to actually make any on-line decisions. Fringe cases could be: the enforcement of terminal liquidation at the last step, not getting a reward if there is no inventory remaining despite

the action being to sell, etc. This practice would be used in a real-world setting as well, so that the model is provided the opportunity to generate fringe trajectories without actually making those mistakes on-line.

Tuning

The hyper-parameters were found by manually tuning the algorithms but were generally similar to the original papers. Parameters that were most notably tweaked: number of steps, eval frequency, number of starting steps, batch size, tau and policy noise. Number of steps, eval frequency, number of starting steps, batch size were greatly reduced to fit the size of the problem. Further, Atari games are closed environments that can afford many starting steps with no repercussions. Conversely, active trading does not have this luxury, so only 100 starting steps were allowed compared to the 10,000 starting steps permitted by the original paper. Further, the learning rate was reduced to 0.0005 from 0.005 to prevent converging too quickly to a sub-optimal policy. Policy noise with 0.95 standard deviation as opposed to 0.2 was changed. Recall, policy noise adds an epsilon drawn from a normal distribution with 0 mean and 0.95 standard deviation and is clipped to 0.5 on either side of the predicted action. If we had to stipulate as to why these two parameters had to be adjusted, it would most likely be because of the algorithm's ability to easily over-fit. We would also assume that there are an infinite number of optimal policies due to the continuous-action space and therefore adding quite a bit of noise to the action allows for several trajectories to bootstrap off of one another.

Train and Test Data

RL does not tend to segment data in the same manner one would split data into train, validation and test sets like in supervised machine learning. For the TD3 algorithm, for instance, the training is governed by the action and value functions, whereas the evaluation of the algorithm is done using the target networks. It is very important to highlight, there are fewer iterations of testing than training because we train the action and value networks for several steps between every evaluation step of the target network. This is done to allow the algorithm to learn something before evaluating it every time.

Recall, there are small updates to the target networks from the action and value networks throughout training. The resulting target network is what we use to then evaluate what the model has learned. This means, the action-value networks are used for training and generate trajectories that will be stored in the replay buffer following an off-line policy. In the case of the TD3 algorithm, the trajectories have a lot of noise added to their actions to make the algorithm more generalizable. Simultaneously, the target networks follow a learned optimal policy and draw from either a true Brownian motion price process or a true skew-normal Brownian motion price process. In the event that we would continue to generate trajectories from the last point of the price process, the data would follow the same principles and so the target network would continue to perform as expected. The interest in RL as opposed to dynamic programming is that the agent will continue to adapt as it interacts with an unknown environment, even if the price process changes.

Comparison with Ning et al. 2018

Now that we have gone over the OOE problem as well as a number of core DRL algorithms in detail, we would like to compare and contrast with a paper we used as inspiration for much of this work, particularly in how we set up the OOE problem.

Setup and Data

This thesis largely uses the same setup as Ning at al. (2018). Both works use hour-long episodes and have actions determine how much of the agent's inventory should be sold at each of the 5 steps. The action-space in the Ning et al. paper uses a discretized formulation of how much inventory should be sold at each step whereas this work uses a continuous action-space. The action-space has a direct impact on which algorithms can be used. As for data, Ning et al. use real market data to train their agent, whereas we simulate data. They too use the same train and test mechanics as this work.

Algorithm

Since Ning et al. discretize their action space, they use value-function methods as opposed to actor-critic methods. They use the DDQN with experience replay as their agent. Because we formulate our action-space as a proportion of current inventory between [0, 1], we can leverage TD3.

Reward

Unlike the reward from Equation 9, the reward used in Ning et al. is awarded at the end of every step and is a brute number whereas we directly use the relative P&L as a sparse reward by only making it available at the end of the episode when relative P&L can be computed. This was done because the brute reward signal could be heavily influenced by the state (i.e. the price at that episode) regardless of the action whereas using relative P&L to TWAP seemed to help alleviate much of this burden.

Analysis

Ning et al. only use DDQN in relation to TWAP, whereas we compare TD3 to TWAP in a synthetic Martingale price process and outline the different actions the algorithm takes while converging to the optimal policy TWAP. We also compare TD3 to DDPG in relation to baseline TWAP in the presence of bias.

Results and Analysis

The following section details the results from the experiments conducted on the 4 different price processes. The first experiment depicts TD3's performance when compared to TWAP in the event that the price process is a Martingale. Secondly, we compare the performance of TD3, TWAP, and DDPG in 3 experiments where we introduce different degrees of slight skew to the arithmetic Brownian random walk with zero drift price process.

Experiment 1: Agent Performance at Learning Optimal Policy

At the beginning of each experiment's section, we give an example of 10 episodes of standardized price processes without skew for reference in Figure 10. The first experiment's analysis and results are presented in three parts: First, we show in Figure 11 the changes in test performance of relative P&L over the course of training. We discuss the properties of how the algorithm converged by providing a visual representation of the incremental improvements with every additional evaluation step. Second, we present summary statistics of the action distribution for the first half of training and compare them to the final segment of training in Tables 3 and 4 respectively. The summary statistics of the action distribution at every time step during an episode tells us that the algorithm isn't just simply matching the returns of the optimal policy, but is actually converging towards selling off 20% of starting inventory at every time step. Third, we report a few summary statistics on the entire test performance in Table 2.

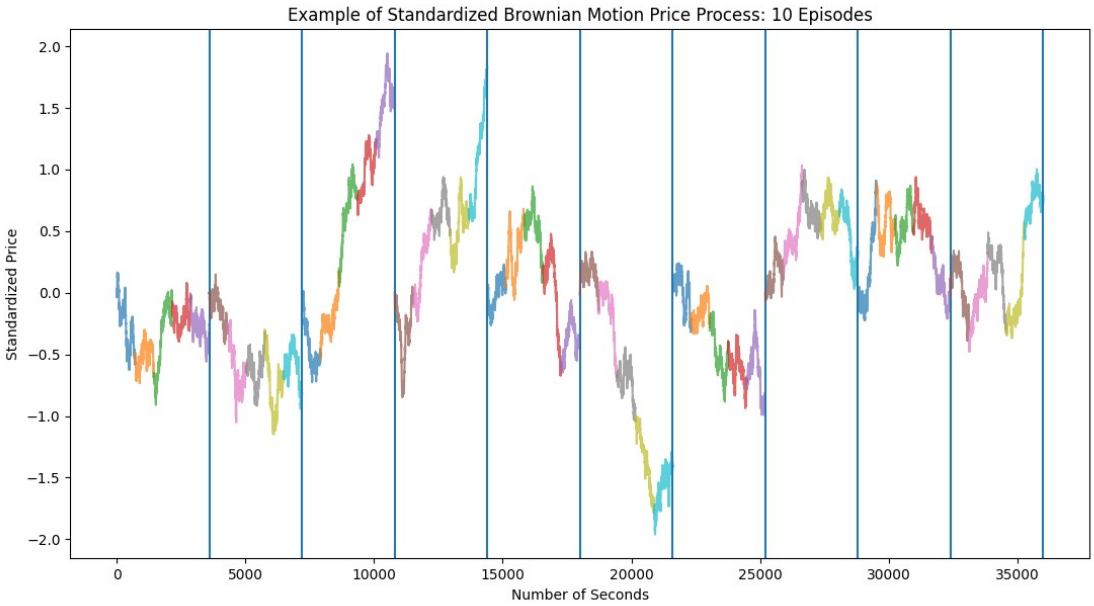

Figure 10: Example of Standardized Brownian Motion Price Process: 10 Episodes

Brownian Motion Comparative Returns to Steady Sell-off

Figure 11 depicts TD3's relative P&L when tested against the TWAP strategy, which is the optimal policy in the case of a Martingale. TWAP represents the value along the x-axis where y=0. This means Figure 11, like Figures 12, 14 and 16, depicts the average episodic return obtained on every evaluation iteration relative to TWAP in their respective experiments. Every evaluation iteration consists of sampling 10 trajectories from the price process and computing average relative P&L over those same 10 trajectories. Since we scaled back the number of training iterations for experiment 1, we will evaluate the target network every 10 training steps to maintain the total number of evaluations across all experiments (100 total evaluation iterations per experiment). TD3 must learn that the optimal policy is to sell 20% of starting inventory at every step. This strategy would optimize for the quadratic penalty on size of orders and the price volatility over the course of the hour.

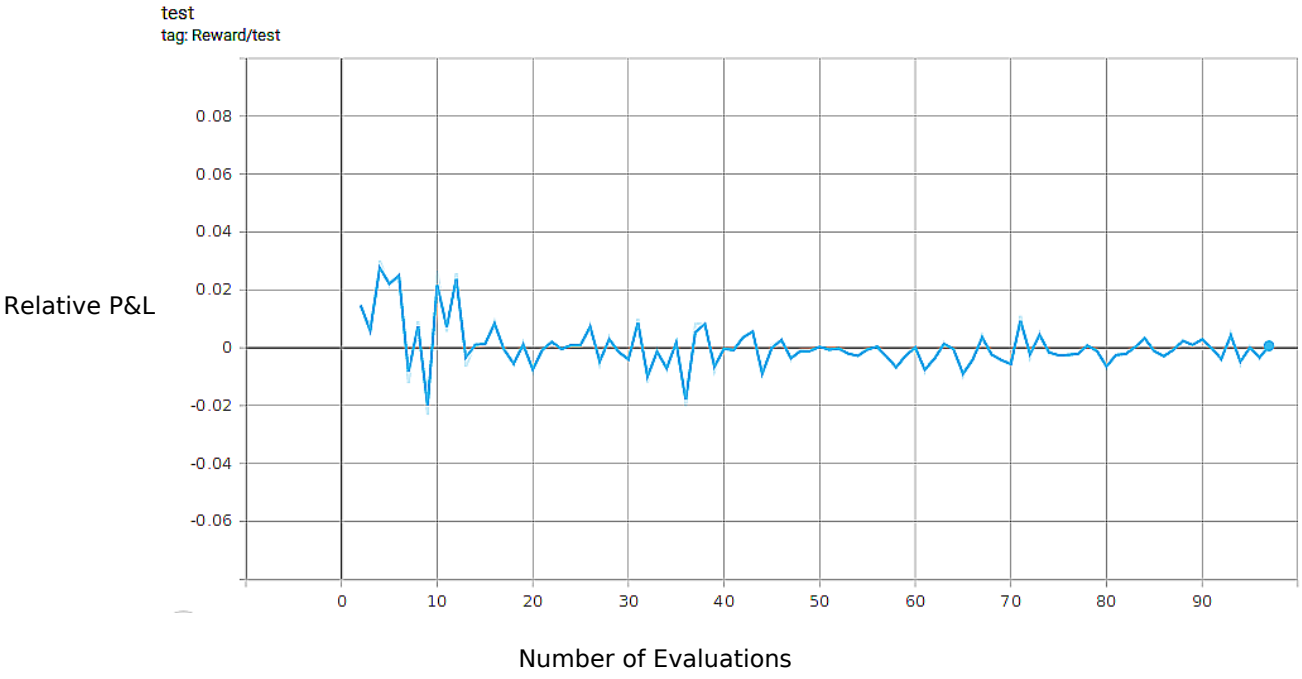

Figure 11: Target Network Relative P&L for TD3 vs TWAP Experiment 1

Notice how TD3 goes through a period of high volatility but converges very quickly to the optimal policy. The hyper-parameters used for the first experiment are identical to the second (see table 1 for more details), but they are scaled back considerably in regard to how long it can train for because of how quickly it can over-fit the data. Tweaking parameters to try to slow the rate at which TD3 learns was performed extensively. Although reducing the learning rate to 5e-6 did slow the model, the most impactful change was to restrict the number of training steps. *Max steps* was reduced to 1000 and the target network was evaluated every 10 training steps, which equates to evaluating the target network 100 times on 10 trajectories each time over the course of 1000 training steps. Table 2 demonstrates the network generally converges to the optimal policy, the next section details summary statistics of the action distribution.

Relative P&L	TD3
Mean	.0001
Standard Deviation	.0083

Table 2: Relative P&L Experiment 1

First Half of Training Action Distribution

Recall, a Martingale price process means that the future value of the stock is expected to be the current price. This means that the optimal policy for this price process is to sell off equal amounts across all time steps, in other words, each of the 5 actions should be to sell off 20% of starting inventory. Table 3 reports summary statistics for the actions as a percentage of starting inventory for the first half of training.

Statistic	Step 1	Step 2	Step 3	Step 4	Step 5	Averages
Count	50	50	50	50	50	50
Mean	.29	.20	.14	.09	.26	0.2
Std	.19	.04	.03	.03	.17	0.09
Min	.11	.14	.08	.03	.02	0.08
25%	.14	.16	.14	.07	.07	0.12
50%	.18	.20	.15	.11	.30	0.19
75%	.45	.24	.17	.12	.40	0.28
Max	.65	.27	.20	.15	.50	0.35

Table 3: First Half Action Distribution Experiment 1

Second Half of Training Action Distribution

Table 4 reports summary statistics for the second half of training.

Statistic	Step 1	Step 2	Step 3	Step 4	Step 5	Averages
Count	50	50	50	50	50	50
Mean	.18	.23	.22	.14	.23	0.2
Std	.03	.01	.01	.01	.04	0.02
Min	.12	.2	.19	.11	.14	0.15
25%	.14	.22	.21	.13	.20	0.18
50%	.18	.23	.22	.13	.20	0.19
75%	.21	.24	.23	.14	.26	0.22
Max	.24	.27	.26	.17	.32	0.25

Table 4: Second Half Action Distribution Experiment 1

Notice how the standard deviation has decreased roughly 80% as the model starts to converge to the optimal policy. Further, notice how the means of all steps are very close to selling 20% of starting inventory, which is the optimal policy in the case of a martingale price process. Finally, notice how the extremes have narrowed significantly. The shift in the summary statistics towards the optimal policy is a strong indication the TD3 algorithm can identify the optimal policy when presented with a Martingale price process.

Experiment 2: Agent Performance in Different Degrees of Bias

The second experiment's analysis and results are presented in 3 parts, with each part consisting of 2 or 3 sub-parts: first, we show the TD3's performance compared to TWAP and DDPG when we bias the price process by .1%. The inclusion of this bias means that the price process is no longer a Martingale, therefore, TWAP is *not* the optimal policy. This implies that the model will have to find the inventory schedule that maximizes the

return from exploiting the bias while minimizing the impact from the quadratic penalty. The Figure 12 illustrates the relative P&L of the evaluation iterations during training. We discuss the properties of how the algorithm converged by providing a visual representation of the incremental improvements with every additional evaluation iteration. The following two sub-experiments consist of the same analysis but with 1% and 10% bias respectively. To highlight the degree to which this bias affects the distribution from which the price changes are drawn, we provide a comparative illustration in the case of 1% and 10% bias in Figures 13 and 15. We chose not to include the .1% comparative distribution because the difference was too faint to be visually discernible from the Martingale example. After each analysis, summary statistics on relative returns are reported in tables 5, 6 and 7 respectively.

Brownian Motion Price Process with Bias +.001

Recall, for experiments 2, 3 and 4, we scale up the number of training steps to 10k. To maintain the same number of times we evaluate the agent, we scale up the number of training steps in-between each evaluation iteration to 100. This means that every 100 training steps, the target network is tested on 10 trajectories drawn from the price process. Below are the reported results of average relative P&L of every evaluation iteration over the course of training:

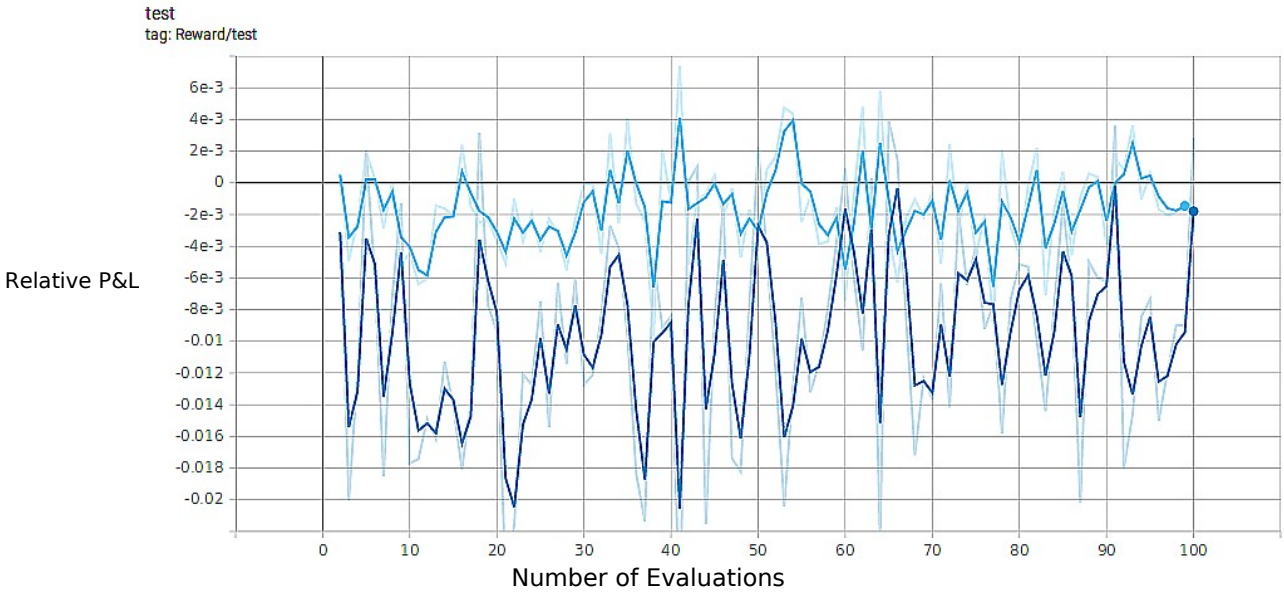

Figure 12: Target Network Relative P&L for TD3 vs DDPG Price Process with .1% Bias

TD3 (light blue), DDPG (dark blue)

Figure 12 depicts the *smoothed* comparative returns to TWAP for both TD3 and DDPG. Smoothing the returns makes for a more interpretable graph. The less pronounced lines are the true return values whereas the more pronounced lines are the smoothed returns. Both are consistently below TWAP, although TD3 comes quite close to converging, it is still considered to be roughly 1 basis point below TWAP. TWAP is an optimal policy for mitigating market impact costs incurred by the quadratic penalty, but it does not exploit the newly added bias by design. DDPG seems to consistently under-perform. It cannot exploit the bias while simultaneously managing the quadratic penalty. After taking into account the differences in scale from the first experiment's results, the DDPG has similar performance to the TD3 before it converges to TWAP, while the TD3 is closer in performance to the first experiment depicted in table 2 (for experiment 2 metrics, please refer to Table 5). A potential reason for TD3 not surpassing TWAP consistently could be that the profitable area of the solution space that falls between the action of fully exploiting the bias or fully minimizing the impact of the quadratic penalty

through TWAP may be very narrow. This perhaps could be improved with further tuning of the hyper-parameters. The improvements to TD3 over DDPG are notable, however. Delaying its update, clipping its value estimate and injecting noise into the target action all seem to dramatically stabilize the algorithm when operating in a noise-heavy environment with a challenging solution space. For instance, the DDPG is known to be overly-responsive to erratic state changes. For this reason, delaying the update, as seen in TD3, would help regularize the input to avoid pre-mature adjustments to its parameters.

Relative P&L	**DDPG**	**TD3**
Mean	-.0095	-.0016
Standard Deviation	.0069	.0031

Table 5: Relative P&L Experiment 2

Brownian Motion Price Process with Bias +1%

For comparison, we provide an example of 10 episodes of a price process with 1% skew in figure 13. There does seem to be a slight upward bias, but in all fairness, we can

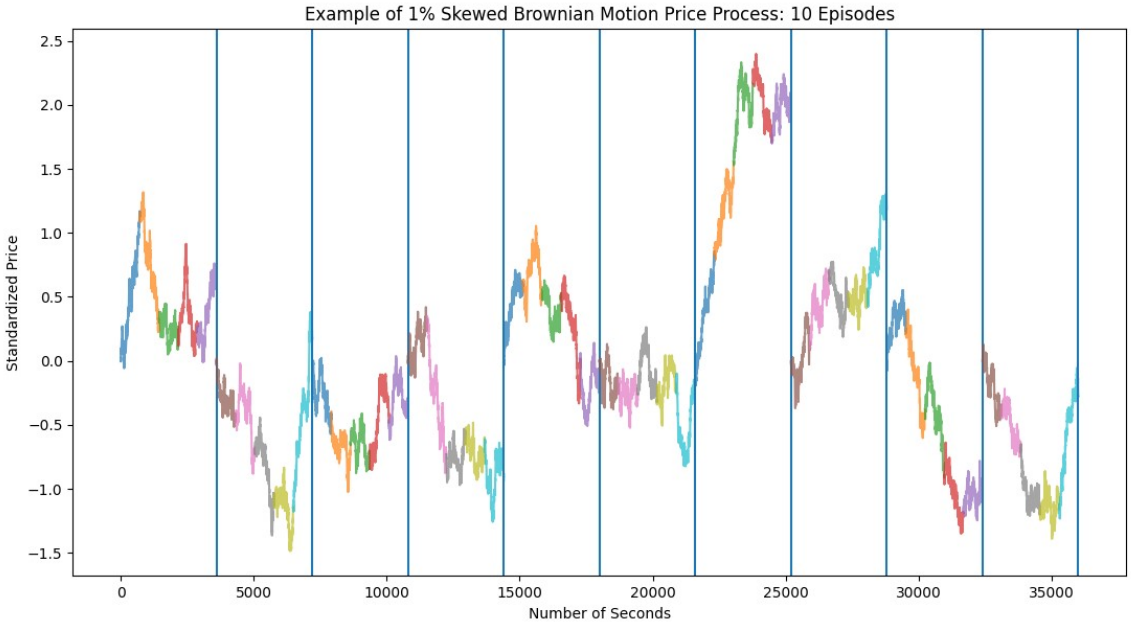

Figure 13: Standardized Price Process of 1% Skewed Brownian Motion

only be sure there is a bias because we have synthetically created it; the figure is provided simply for comparative purposes to highlight differences when we increase the bias.

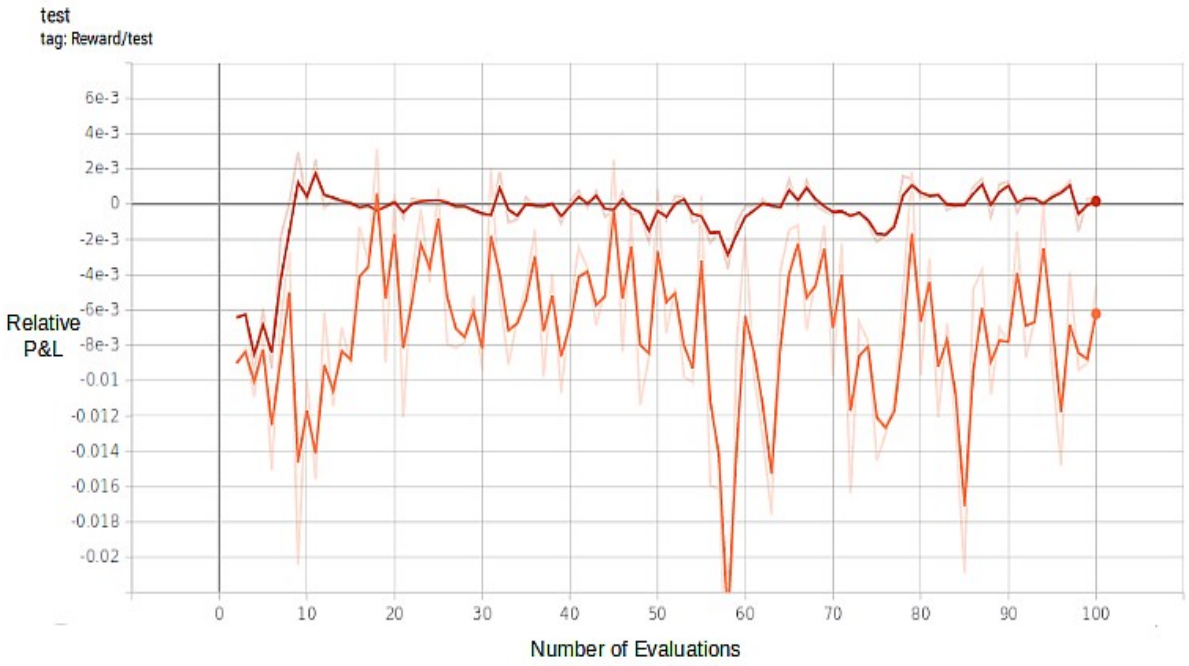

Figure 14: Target Network Relative P&L for TD3 vs DDPG Price Process with 1% Bias TD3 (red) DDPG (orange)

Figure 14 depicts the comparative returns to TWAP for both TD3 and DDPG. The TD3 algorithm starts to consistently surpass TWAP after a few hundred training steps (equivalently, roughly 8 evaluation iterations), then briefly drops to TWAP and finally closes above TWAP. DDPG, although seems to improve in performance from the last experiment after taking into account the differences in scale, still under-performs. It cannot exploit the bias while simultaneously managing the quadratic penalty, although it seems to do a slightly better job when the bias is more pronounced. Interestingly, there is a drop in performance for both algorithms at roughly the midway point of training. We believe this drop is important for highlighting both the similarities and differences between these algorithms. It seems as though the TD3 modifications are successfully

stabilizing the target network when the network is at risk of experiencing moments of high volatility. Indeed, the modifications could be greatly reducing the magnitude of volatility in returns. Both algorithms suffer losses, but the DDPG's performance degradation is much more pronounced than TD3's.

Relative P&L	DDPG	TD3
Mean	-.0072	-.0004
Standard Deviation	.0055	.0019

Table 6: Relative P&L Experiment 3

Brownian Motion Price Process +10%

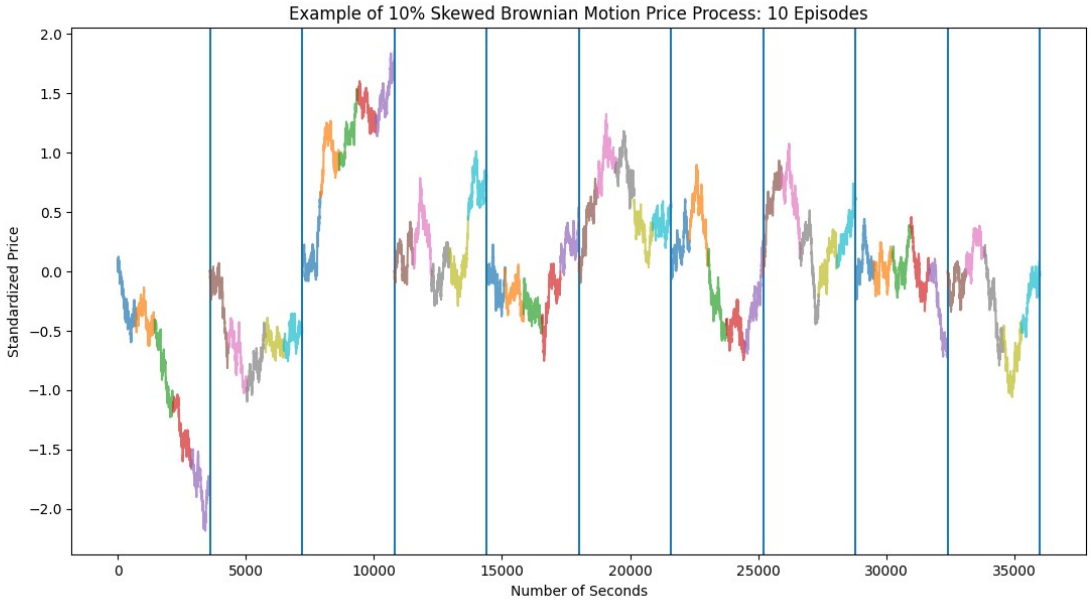

Figure 15: Example of 10% Skew Brownian Motion Price Process: 10 Episodes

For comparison, we provide an example of 10 episodes of a price process with 10% skew in figure 15. The bias is still faint, but does seem to be less negatively inclined.

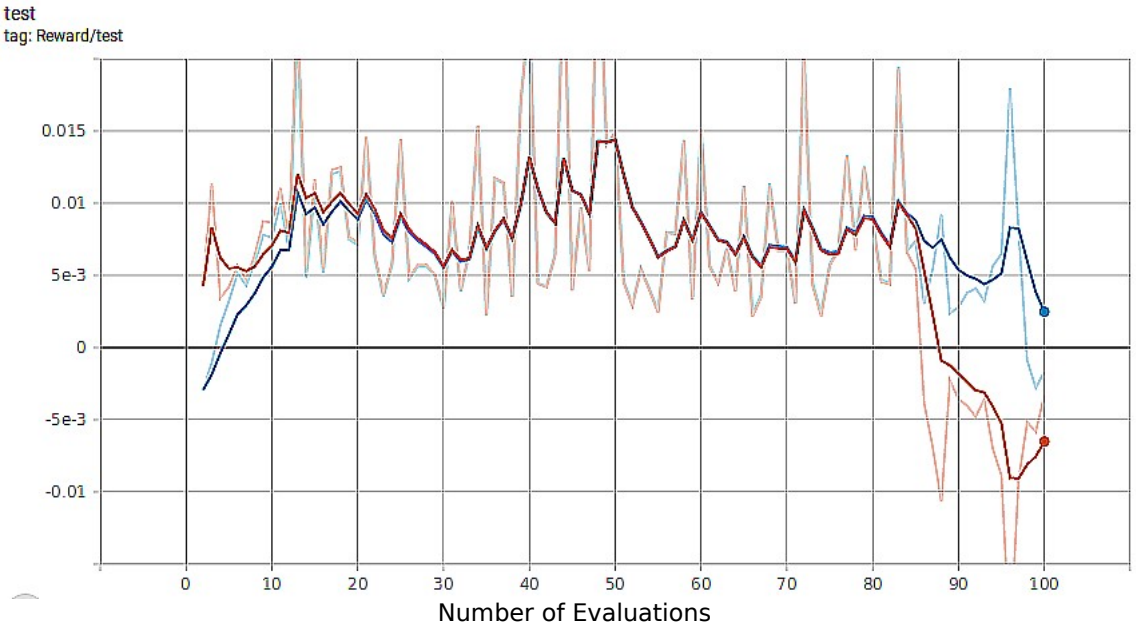

Figure 16: Target Network Relative P&L for TD3 vs DDPG Price Process with 10% Bias TD3 (blue), DDPG (red)

Finally, we use a price process with 10% bias to compare the TD3 and DDPG to TWAP and we visualize the test performance during training in Figure 16. Interestingly, the DDPG converges to a profitable policy faster than the TD3, although both stabilize for thousands of training steps until degradation. Another interesting observation is how the TD3's clear dominance from previous experiments doesn't seem to be present for most of the training period. However, notice how the DDPG suffers from significant performance degradation at the 80th evaluation iteration, whereas the TD3's drop is far less pronounced and largely still profitable. This is another example of how TD3 can help reduce

performance volatility by restricting large changes in the target network from one update to the next. We report summary statistics of the test performance in Table 7.

Relative P&L	**DDPG**	**TD3**
Mean	.0061	.0075
Standard Deviation	.0078	.0059

Table 7: Relative P&L Experiment 4

Discussion

Summary of Problem

Optimal order execution is an important problem faced by large market participants, as it requires the participant to balance the risk of holding the asset through price volatility with the market impact risk of exiting a large position. If a participant places too large an order, the risk of market impact increases, thereby affecting the price of the asset through standard supply-demand dynamics in the order book. If the participant holds the position too long, he/she increases the risk of exposure to unfavorable price movements.

Given a set number of shares at the beginning of an episode, the agent must find the optimal sell-off schedule that will optimize its returns when compared to a steady sell-off schedule, known as steady-sell off TWAP, where no adjustment decisions can be made throughout. OOE consists of a trader being given X total inventory to sell at the beginning of an episode, each episode being of a fixed length in time T number of seconds. There are N decision points over the course of an episode, we refer to these as k *steps* where $k \in [0, 1, .., N]$ and N represents the terminal step. Each k is separated by a discrete time interval t where $t \in [0, 1, .., N]$. Each t spans $\frac{T}{N}$ seconds, or referred to as M seconds. At each step k, we must determine the amount of inventory to sell x_k such that $\sum x_k$ equates to X to ensure full liquidation when the episode terminates. Rather than submitting one large order at the beginning of each step and negatively impacting available prices in the order book, we would like to equally disperse x_k throughout the time interval t by following a TWAP schedule at 1 second time-increments. To obtain the size of the per second order

increment, we must compute $\frac{x_k}{M}$. Once the order is complete at the end of time interval t we move to step $k+1$ where a new amount x_{k+1} must be determined to then be sold over the course of $t+1$.

Summary of Model

The TD3 algorithm is nearly identical to that of the DDPG. Three modifications were introduced, however, to better address approximation errors in the DDPG algorithm (Fujimoto 2018). First is to smooth the action selection of the target network, second is to select the minimum value estimate produced by the critic networks and last is to delay the update for the actor and target network. These improvements were introduced to reduce over-estimation in the critic output and stabilize training.

Summary of Results

According to the results from the previous section, increases in bias positively correlate with the TD3 algorithm successfully identifying an exploitable pattern. Likewise, DDPG also experiences an improvement, albeit not to the same degree. If the bias is too weak or non-existent, as would be the case in the first price processes of Brownian motion, the TD3 algorithm converges to a TWAP steady sell-off schedule. This is consistent with previous research that demonstrates that the optimal policy is to sell-off shares in equal increments throughout the period in the event of a Martingale process. Prior to starting the experiments, two hypotheses were made:

Hypothesis 1: In the first experiment, the TD3 converges to the optimal policy, which is a steady-sell off in the case of a Martingale price process.

Figure 11 demonstrates the TD3's eventual oscillation around 0, which is the baseline TWAP strategy and is theoretically optimal in the event of a Martingale price process. To demonstrate TD3 not only produces equivalent results to baseline TWAP but also matches its action strategy, tables 3 and 4 were included in the analysis. The tables clearly show a change in action preferences towards the optimal policy of selling off equal increments of 20% of starting inventory at the beginning of every step during the episode. Although the TD3 algorithm definitely converges, we had to aggressively scale back the number of iterations it could train for because of how erratic it could behave in a context where the problem is possibly overly-simplistic. We could have possibly added a decay to the learning rate to prevent this from happening. This served as an important reminder of how quickly DRL algorithms can over-fit the data, despite extensively tweaking their hyper-parameters in an attempt to restrict the rate at which it learns. Even when employing models like TD3, which are specifically designed to mitigate major shifts in the target networks upon update, the problem should be well suited for these types of powerful algorithms. Regardless, it was interesting to see just how quickly the model could adapt in the event of a Martingale price process.

Hypothesis 2: In the second experiment, the TD3 algorithm outperforms the DDPG algorithm and consistently surpasses the baseline TWAP returns.

The second experiment was to test whether the model would be able to identify the different degrees of bias across 3 different price processes when compared to TWAP and DDPG. The results proved to be very interesting, as they tended to highlight where TD3 shines over DDPG and where both algorithms struggle to identify and exploit the bias. Both algorithms are quite similar except for 3 modifications used in TD3 to help stifle overly aggressive target network updates. For more details, please refer to Figure 8 for an explanation of the pseudo-code. Unsurprisingly, the degree of bias was positively correlated with both algorithm's returns over the baseline TWAP schedule. In instances where the bias was faintest, both algorithms were sub-optimal, although the difference between TWAP and TD3 could be negligible. As the bias increases slightly, DDPG still struggles to be profitable, although it still clearly benefits from a more pronounced bias signal, while TD3 manages to start becoming profitable. Where the results become very interesting is when bias is at its highest and the algorithms perform nearly identically. The results diverge, however, when both are faced with a risk of high-volatility in target updates when the actor-critic networks are suddenly very different from their target networks. When analyzing these results, it appears that TD3 has greater resilience to volatility in its value estimate. Given that there are only 3 differences between TD3 and DDPG, it seems as though the modifications intended to stabilize TD3 could be serving their purpose. Naturally, DDPG and TD3 move very similarly when stable, it is when their on-line and off-line networks are highly divergent that their differences become apparent. The results on the skew experiments are compelling: they show TD3 nearly matches TWAP in no bias and extremely faint bias (suffering very little loss over TWAP) and greatly surpasses TWAP when the skew parameter is set to as low as 10%. Further, TD3 appears to consistently outperform DDPG, particularly when both networks are faced with large discrepancies between their target networks and their action and value networks.

Limitations

Although the results are intriguing, there are a few limitations that should be outlined. The chief limitation being that Brownian motion with bias is not a perfect predictor of future earning potential in a real-world setting, however, few if any back-testing methodologies are ever sufficient on their own (López de Prado, 2018). Indeed, this approach can be used as an important facet of a holistic back-testing process, where a combination of both simulated and historical data can be used to effectively gauge the performance of an algorithm.

Contribution

This thesis is the first known example of applying the TD3 or DDPG algorithm to the optimal order execution problem. It is also the first known work where RL is used to exploit the bias in a skew-normal Brownian motion price process in trading. We demonstrate that relatively quick convergence can be achieved when using a continuous action-space. It is the first example of the TD3 agent successfully identifying the known optimal policy in a Brownian motion price process with zero skew or drift. Further, we demonstrate how TD3 significantly outperforms the baseline TWAP schedule and proves to be more stable than the DDPG algorithm in a skew-normal Brownian motion price process with the skew parameter set to only 10%.

Future Research

It would be fascinating to see if the agent's performance is maintained when run in a market simulator, such as the ABIDES (Agent-Based Interactive Discrete Event Simulation) Market Simulator for AI Research (Byrd et al., 2019). The ABIDES market simulator uses the full limit order book and can simulate thousands of different agents that interact with the market via limit orders or market orders. These interactions can help us model the price impact of strategies, rather than using an approximate method like our quadratic penalty. Other future research could compare the agent's performance if it were to submit limit orders as opposed to market orders, or a combination of the two.

Conclusion

This work is the first known example of applying the TD3 or DDPG algorithm to the optimal order execution problem. Much of our approach to constructing the OOE problem was borrowed from Ning at al. (2018), where we differ in the algorithm selection, action-space and reward design. First we discuss the order execution problem as well as order book dynamics. We then summarize key concepts in RL and model approximation to showcase RL as a viable option for solving the OOE problem. Next, a literary review is done on historical applications of RL in OOE. We also start to explain why DRL could help circumvent some issues encountered by more traditional RL methods. Next, an in depth review of major algorithmic innovations in the DRL space is provided. This discussion naturally leads to our reasoning behind selecting TD3 as the central model in this thesis. Once the foundation laid, the experimentation methodology and hypotheses were outlined.

For the first experiment, the TD3 agent converged to the baseline TWAP, the optimal policy in the case of a Martingale price process. The skew experiments demonstrated the TD3's ability to detect varying degrees of faint bias in the presence of a skew-normal Brownian motion price process. We compared these results to that of the DDPG algorithm, where the TD3 algorithm consistently outperformed DDPG. Bias was created by skewing the normal distribution from which the errors were drawn to generate the price processes. It is evident that, although faint, larger degrees of bias lead to larger returns over the TWAP baseline execution strategy. This also demonstrates that the TD3 can converge to a profitable policy after only a thousand observations on a simulated price

process, despite the bias being very faint. It is important to note, if the bias is too faint, the TD3 considers the TWAP strategy to be the optimal policy, which intuitively seems to make sense. In conclusion, these results are promising for DRL applications in OOE and are notable given the paucity of research in the continuous action space for this particular problem. Future research demonstrating the viability of DRL in trading will be relevant to any firm looking to optimize order flows.

Bibliography

Almgren, R., & Chriss, N. (2001). Optimal execution of portfolio transactions. *The Journal of Risk*, *3*(2), 5–39. https://doi.org/10.21314/JOR.2001.041

Azzalini, A., & Capitanio, A. (1999). Statistical applications of the multivariate skew normal distribution. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, *61*(3), 579–602. https://doi.org/10.1111/1467-9868.00194

Bellman, R. (1952). On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences*, *38*(8), 716–719. https://doi.org/10.1073/pnas.38.8.716

Berkowitz, S. A., Logue, D. E., & Noser, E. A. (1988). The Total Cost of Transactions on the NYSE. *The Journal of Finance*, *43*(1), 97–112. https://doi.org/10.1111/j.1540-6261.1988.tb02591.x

Brownian Motion—SciPy Cookbook documentation. (2017). https://scipy-cookbook.readthedocs.io/items/BrownianMotion.html

Byrd, D., Hybinette, M., & Balch, T. H. (2019). ABIDES: Towards High-Fidelity Market Simulation for AI Research. *ArXiv:1904.12066 [Cs]*. http://arxiv.org/abs/1904.12066

Cartea, lvaro, & Jaimungal, S. (2015). Incorporating Order-Flow into Optimal Execution. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.2557457

Caspi, Leibovich, Novik, & Endrawis. (2019). *Twin Delayed Deep Deterministic Policy Gradient—Reinforcement Learning Coach 0.12.0 documentation*. https://nervanasystems.github.io/coach/components/agents/policy_optimization/td3.html

Dempster, M. A. H., & Leemans, V. (2006). An automated FX trading system using adaptive reinforcement learning. *Expert Systems with Applications*, *30*(3), 543–552. https://doi.org/10.1016/j.eswa.2005.10.012

Deng, Y., Bao, F., Kong, Y., Ren, Z., & Dai, Q. (2017). Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems*, *28*(3), 653–664. https://doi.org/10.1109/TNNLS.2016.2522401

Deng, Y., Kong, Y., Bao, F., & Dai, Q. (2015). Sparse Coding-Inspired Optimal Trading System for HFT Industry. *IEEE Transactions on Industrial Informatics*, *11*(2), 467–475. https://doi.org/10.1109/TII.2015.2404299

Dhesi, D. G., Emambocus, M. A. W., & Shakeel, M. B. (2012). Semiclosed Pricing Mechanism. *ArXiv:1112.0342 [q-Fin]*. http://arxiv.org/abs/1112.0342

Doostparast, M. (2017). Explicit expressions for European option pricing under a generalized skew normal distribution. *ArXiv:1707.09609 [q-Fin, Stat]*. http://arxiv.org/abs/1707.09609

Doria, D., Dawson, B., & Vindiola, M. (2015). *Enhanced Experience Replay for Deep Reinforcement Learning:* Defense Technical Information Center. https://doi.org/10.21236/ADA624278

Eling, M., Farinelli, S., Rossello, D., & Tibiletti, L. (2010). Skewness in hedge funds returns: Classical skewness coefficients vs Azzalini's skewness parameter. *International Journal of Managerial Finance*, *6*(4), 290–304. https://doi.org/10.1108/17439131011074459

Fama, E. F. (1965). The Behavior of Stock-Market Prices. *The Journal of Business*, *38*(1), 34–105. JSTOR.

Fujimoto, S. (2018). *Addressing Function Approximation Error in Actor-Critic Methods*. 10.

Gatheral, J., & Schied, A. (2011). OPTIMAL TRADE EXECUTION UNDER GEOMETRIC BROWNIAN MOTION IN THE ALMGREN AND CHRISS

FRAMEWORK. *International Journal of Theoretical and Applied Finance*, *14*(03), 353–368. https://doi.org/10.1142/S0219024911006577

Hendricks, D. (n.d.). *An online adaptive learning algorithm for optimal trade execution in high-frequency markets*. 211.

Imperial, F. (2018). *Modelling Stock Prices and Stock Market Behaviour using the Irrational Fractional Brownian Motion: An Application to the S&P500 in Eight Different Periods*. https://doi.org/10.13140/RG.2.2.25164.92809

Kato, T. (2017). An Optimal Execution Problem in the Volume-Dependent Almgren-Chriss Model. *ArXiv:1701.08972 [q-Fin]*. http://arxiv.org/abs/1701.08972

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *ArXiv:1509.02971 [Cs, Stat]*. http://arxiv.org/abs/1509.02971

López de Prado, M. M. (2018). *Advances in financial machine learning*. Wiley.

Maeda, J., & Jacka, S. D. (2018). Modeling Technical Analysis. *ArXiv:1707.05253 [q-Fin]*. http://arxiv.org/abs/1707.05253

Malkiel, B. G., & Fama, E. F. (1970). EFFICIENT CAPITAL MARKETS: A REVIEW OF THEORY AND EMPIRICAL WORK*. *The Journal of Finance*, *25*(2), 383–417. https://doi.org/10.1111/j.1540-6261.1970.tb00518.x

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *ArXiv:1312.5602 [Cs]*. http://arxiv.org/abs/1312.5602

Moody, J., & Saffell, M. (2001). Learning to trade via direct reinforcement. *IEEE Transactions on Neural Networks*, *12*(4), 875–889. https://doi.org/10.1109/72.935097

Moreno-Vera, F. (2019). Performing Deep Recurrent Double Q-Learning for Atari Games. *ArXiv:1908.06040 [Cs, Stat]*. http://arxiv.org/abs/1908.06040

Ning, B., Ling, F. H. T., & Jaimungal, S. (2018). Double Deep Q-Learning for Optimal Execution. *ArXiv:1812.06600 [Cs, q-Fin, Stat]*. http://arxiv.org/abs/1812.06600

Reddy, K., & Clinton, V. (2016). Simulating Stock Prices Using Geometric Brownian Motion: Evidence from Australian Companies. *Australasian Accounting, Business and Finance Journal*, *10*(3). https://doi.org/10.14453/aabfj.v10i3.3

Ritchie Ng, J. F. (2018). *Deep Learning Fundamentals*. Zenodo. https://doi.org/10.5281/ZENODO.1480880

Roibu, A. C. (2019). Design of Artificial Intelligence Agents for Games using Deep Reinforcement Learning. *ArXiv:1905.04127 [Cs]*. http://arxiv.org/abs/1905.04127

Romero, J. M., & Bautista, J. (2016). Exact solutions for optimal execution of portfolios transactions and the Riccati equation. *ArXiv:1601.07961 [q-Fin]*. http://arxiv.org/abs/1601.07961

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Rummery, G. A., & Niranjan, M. (1994). *On-Line Q-Learning Using Connectionist Systems*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2018). High-Dimensional Continuous Control Using Generalized Advantage Estimation. *ArXiv:1506.02438 [Cs]*. http://arxiv.org/abs/1506.02438

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (Second edition). The MIT Press.

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *ICGA Journal*, *18*(2), 88–88. https://doi.org/10.3233/ICG-1995-18207

van Hasselt, H., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. *ArXiv:1509.06461 [Cs]*. http://arxiv.org/abs/1509.06461

Zhai, J., Liu, Q., Zhang, Z., Zhong, S., Zhu, H., Zhang, P., & Sun, C. (2016). Deep Q-Learning with Prioritized Sampling. In A. Hirose, S. Ozawa, K. Doya, K. Ikeda, M. Lee, & D. Liu (Eds.), *Neural Information Processing* (Vol. 9947, pp. 13–22). Springer International Publishing. https://doi.org/10.1007/978-3-319-46687-3_2

Zhou, T., Jia, C., & Li, H. (2017). The Simulation Analysis of Optimal Execution Based on Almgren-Chriss Framework. *DEStech Transactions on Computer Science and Engineering*, *cmsam*. https://doi.org/10.12783/dtcse/cmsam2017/16351

Zhu, S.-P., & He, X.-J. (2018). A new closed-form formula for pricing European options under a skew Brownian motion. *The European Journal of Finance*, *24*(12), 1063–1074. https://doi.org/10.1080/1351847X.2017.1339104