Efficient Machine Learning Methods for Solving Hamilton-Jacobi-Bellman Equations in Finance

by

Andrew S. Na

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

© Andrew S. Na 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

Supervisor(s):	Justin W.L. Wan Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal Member:	Yuying Li Professor, David R. Cheriton School of Computer Science, University of Waterloo
	Christopher Batty Associate Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	James R. Thompson Associate Professor, School of Accounting and Finance, University of Waterloo
External Examiner:	Christoph Reisinger Professor, Mathematical Institute, University of Oxford

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Recent developments in machine learning have allowed for the solution of high-dimensional partial differential equations overcoming the curse of dimensionality. In our work, we utilize deep neural networks to our advantage to solve Hamilton-Jacobi-Bellman (HJB) equations that arise in financial applications efficiently. Two prevalent methods are commonly used to solve high-dimensional HJB equations. One method uses physics-based neural networks to approximate the unknown solution while regularizing the solution with boundary conditions. Another method, and the focus of this work, represents the HJB equation as a backward-stochastic differential equation (BSDE). This allows us to solve the HJB equation using a reinforcement learning framework.

A major bottleneck is present in both methods. Both methods require a neural network to be trained at each timestep. This means that a new network with large number of parameters need to be stored and trained at each timestep which requires a lot of memory, time and energy. In this work we propose different methods to solve the HJB equation in specific financial applications. To do this, we design specific neural network architectures that can fit the problem and derive loss functions to allow us to train efficient networks.

In this work we explore four different problems. Firstly, we solve the HJB equation that arises in high-dimensional American options pricing. We present a method that uses two separate neural network that learns to approximate the price and delta. We improve efficiency by reducing the number of neural networks that we need to train. Secondly, we extend and solve the optimal trade execution problem for multiple agents and assets under the Markowitz criteria in a time-consistent way.

Lastly, we solve the inverse problem with minimal arbitrage violations. The computation of implied volatilities from data is a challenging task that is necessary to accurately compute the price and hedging positions of over-the-counter (OTC) products.

Acknowledgements

I would like to thank my PhD advisor Dr. Wan, for all the guidance, patience and enlightening discussions we had throughout the years of working together. It has been one of the most memorable periods of my life.

I would also like to thank Dr. Li and Dr. Batty on the great feedback they gave me at each step of my research. Finally, I would like to thank my parents and Alicia who supported me through my doctoral studies with great patience, understanding and love.

Dedication

I dedicate this work to my parents and Alicia Chung.

Table of Contents

Examining Committee ii

Author's Declaration iii

Abstract iv

Acknowledgements v

Dedication vi

List of Figures xii

List of Tables xvi

1 Introduction 1

1.1 Hamilton-Jacobi Bellman Equations 3

1.1.1 Backward stochastic differential equation 5

1.2 Preliminary on Machine Learning 7

1.2.1 Supervised learning 7

1.2.2 Reinforcement learning 8

1.2.3 Neural networks 9

1.2.4 Generative adversarial networks 11

1.3 Applications in Finance . . . 12
1.3.1 American option pricing . . . 12
1.3.2 Optimal trade execution . . . 13
1.3.3 Computing volatility surfaces with minimal arbitrage violations . . . 14
1.4 Contributions . . . 14
1.5 Outline . . . 16

2 Efficiently Solving High-Dimensional American Option Pricing using Deep Recurrent Networks 17

2.1 Introduction . . . 17
2.2 HJB Equation from the American Option Problem . . . 20
2.2.1 American options . . . 20
2.2.2 HJB formulation . . . 22
2.3 Loss Function Formulation from the BSDE equation . . . 22
2.3.1 Discretization of BSDE . . . 23
2.3.2 Least squares formulation of the BSDE . . . 24
2.4 Continuation price and delta at timestep n . . . 26
2.5 Deep Recurrent Neural Network Formulation . . . 27
2.5.1 Sequence of neural networks . . . 27
2.5.2 Activation function selection for output and embedding layers . . . 31
2.5.3 Training and evaluation . . . 31
2.6 Computational Cost of Deep Recurrent Network . . . 34
2.6.1 Time cost . . . 35
2.6.2 Memory cost . . . 36
2.7 Numerical Results . . . 37
2.7.1 American style geometric average call option . . . 39
2.7.2 American style max call option . . . 47
2.8 Conclusion . . . 54

3 Solving High-Dimensional HJB Equations Arising from Multi Agent Optimal Trade Execution using Residual Networks and Self Attention 56

3.1 Introduction . . . 56

3.2 Mean-Variance Optimal Trade Execution . . . 59

3.2.1 Problem formulation . . . 60

3.2.2 Extension to d assets and K agents . . . 62

3.2.3 Mean variance criteria of K agents . . . 63

3.3 Backward Stochastic Differential Equation (BSDE) Representation of the HJB Equation . . . 65

3.3.1 HJB Formulation . . . 65

3.3.2 BSDE formulation . . . 68

3.3.3 Discretization of BSDE . . . 69

3.4 Neural network approach to the optimal trade execution problem . . . 72

3.4.1 Neural network formulation . . . 72

3.4.2 Network Training and Prediction . . . 75

3.5 Numerical Results . . . 78

3.5.1 Experiment 1: validation of neural network solution against existing solutions . . . 78

3.5.2 Experiment 2: sensitivity of parameters . . . 82

3.5.3 Experiment 3: optimal execution in advantageous and disadvantageous market conditions for 2 agents and the effects of correlation . 86

3.5.4 Experiment 4: optimal execution for 2 agents with different price impact . . . 92

3.6 Conclusion . . . 96

4 Efficiently Computing Implied Volatility Surfaces using GANs with Minimal Arbitrage Violations 97

4.1 Introduction . . . 97

4.2 Implied Volatility Surfaces and Static Arbitrage . . . 100

4.2.1 Implied volatility . . . 100
4.2.2 Local volatility . . . 101
4.2.3 Static arbitrage . . . 101
4.3 GAN Framework for Computing Implied Volatility Surfaces . . . 103
4.3.1 GAN formulation of the inverse problem . . . 103
4.3.2 Loss function and proposed model . . . 105
4.3.3 Training the GAN . . . 110
4.3.4 Generating data for training, validation and testing . . . 110
4.4 Numerical Results . . . 112
4.4.1 Experiment 1: comparison of GAN models, the effects of soft constraints and MSE loss . . . 114
4.4.2 Experiment 2: comparison of Implied Volatility with IV-ANN . . . 117
4.4.3 Experiment 3: computing market consistent implied volatility surfaces 120
4.4.4 Experiment 4: comparison of local volatility computation with VAE 120
4.5 Conclusion . . . 122

5 Conclusion and Future Work **123**
5.1 Summary . . . 123
5.2 Future Work . . . 125
5.2.1 Extensions to pricing high-dimensional American options using recurrent networks . . . 125
5.2.2 Extensions to optimal trade execution problem . . . 125
5.2.3 Extensions to computing volatility surfaces using generative methods 126

References **127**

APPENDICES **139**

A Additional Details **140**

A.1 Additional Details and Results for Chapter 2 140

A.1.1 Time and memory cost table 140

A.1.2 Feature selection and network summary 140

A.2 Additional Details and Results for Chapter 3 142

A.2.1 Almgren-Chriss model 142

A.2.2 Generating data for optimal trade execution 143

A.2.3 Experiment: performance of multiple sellers and buyers in different economies 144

A.3 Additional Details and Results for Chapter 4 148

A.3.1 Heston model 148

A.3.2 Surface stochastic volatility inspired (SSVI) model 149

A.3.3 No-arbitrage conditions imposed on option price surface 149

List of Figures

1.1 A simplified diagram of reinforcement learning. This diagram illustrates the interaction between the agent and environment 8

1.2 A 2 layer ANN model. Each circle called a *node* represents a perceptron and each layer is made up of multiple perceptrons [68]. 9

2.1 Deep recurrent network for price rolled out through time. Our network is composed of a recurrent unit, an embedding linear layer and an output linear layer. Takes in a matrix of inputs X, hidden information h and outputs $\mathcal{F}$. 28

2.2 Stacked RNN units within the deep RNN 29

2.3 Logarithm of the total time, T(N), against the number of time steps, N, for the DRL method (blue line) and the proposed method (orange line) for dimensions (a) $d = 2$, (b) $d = 5$ and (c) $d = 10$. 35

2.4 Logarithm of total peak memory versus the number of time steps, N, for the DRL method (blue line) and the proposed method (orange line) for dimensions (a) $d = 2$, (b) $d = 5$ and (c) $d = 10$. 37

2.5 Multi-dimensional geometric call options. Comparison of the exercised sample points between the proposed method ((a) and (c)) and the DRL method ((b) and (d)). Points exercised (blue) and points continued (red); points misclassified as continued (bold dark blue) and points misclassified as exercised (bold dark red). 44

2.6 Multidimensional geometric call options. Density of the relative P&Ls computed by the proposed approach, subject to 250 hedging intervals at $s_0 = 100$ for dimensions (a) $d = 2$, (b) $d = 5$, (c) $d = 30$ and (d) $d = 100$. 46

2.7 2-dimensional max call options. Comparison of exercised sample points between the proposed method ((a) and (c)) and the DRL method ((b) and (d)) computed at times $t = 0.5$ ((a) and (b)) and $t = 0.74$ ((c) and (d)) and initial price $s_0 = 110$. Exercised points(blue) and continued points (red); points misclassified as continued(bold dark blue) and points misclassified as exercised (bold dark red). 51

2.8 2-dimensional max call option. Comparison of the densities of the relative P&Ls computed by the proposed approach (blue) versus the relative P&Ls computed by the finite difference method (red) at $s_0 = 1.1K$. 55

3.1 U-net with self attention architecture. It is composed of a multi-head attention layer, convolution layers and decoding transpose convolution layers with layer normalization. The input data is added to the output of each layer creating a residual structure. 74

3.2 Data flow illustration of the residual U-net with self-attention rolled out through time. 74

3.3 The efficient frontier interpolated from the neural network approximation (blue) and interpolated values from [55] (orange). Interpolation was performed using a polynomial of degree 4. 80

3.4 The sensitivity of of α^* with respect to $\phi = [0.0, 0.2, 0.4, 0.6, 0.8]$ at $t = 1/250$. The sensitivity of agent 1 (left) and the agent 2 (right) 83

3.5 The sensitivity of of α^* with respect to $\gamma = [3.0, 3.75, 4.5, 5.25, 6.0]$ at $t = 1/250$. The sensitivity of agent 1 (left) and the agent 2 (right). 83

3.6 The effect of $\kappa_t = [0.0, 2.5\times10^{-8}, 5\times10^{-8}, 7.5\times10^{-8}, 1\times10^{-7}]$ (top) and $\kappa_p = [0.0, 0.000125, 0.00025, 0.000375, 0.0005]$ (bottom) on α^* for agent 1 (left) and agent 2(right). We used the average of 10 networks for this simulation. 84

3.7 α^* for agent 1 (top left) and agent 2 (top right). v^* for agent 1 (bottom left) and agent 2 (bottom right) with $\kappa_t = 1 \times 10^{-7}$. We fix $M = 1000$ and $N = 100$. We used the average of 100 networks for this simulation. 85

3.8 α^* for agent 1 (top left) and agent 2 (top right) with low correlation. α^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation.We used the average of 100 networks for this experiment. 87

3.9 v^* for agent 1 (top left) and agent 2 (top right) with low correlation. v^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment. 88

3.10 α^* for agent 1 (top left) and agent 2 (top right) with low correlation. α^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment. 89

3.11 v^* for agent 1 (top left) and agent 2 (top right) with low correlation. v^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment. 90

3.12 α^* for agent 1 (top left) and agent 2 (top right) with low correlation. α^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment. 91

3.13 v^* for agent 1 (top left) and agent 2 (top right) with low correlation. v^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment. 93

3.14 The optimal portfolio (top) and trade rate (bottom) for Agents 1 (blue) and 2 (orange). For a good economy (right) and bad economy (left) when $\phi = 0.0$. 94

3.15 The optimal portfolio (top) and trade rate (bottom) for Agents 1 (blue) and 2 (orange). For a good economy (right) and bad economy (left) when $\phi = 0.3$. 95

4.1 Schematic of discriminator network during training. The discriminator is trained with the Black-Scholes implied volatility (BS-IV) as the true labels and the generator output with noise as the fake labels. The generator weights are fixed while the discriminator is being trained. 104

4.2 Schematic of generator network during training. The ground truth is given by the Black-Scholes implied volatility. The discriminator weights are fixed during the training of the generator network. 104

4.3 Pipeline used to generate data for training and testing our model. 111

4.4 The implied volatility computed by GAN-1 with soft constraints (red) and GAN-2 with soft constraints (green) compared to the Black-Scholes implied volatility (blue). 115

4.5 The implied volatility approximated by GAN-1 with no constraints (orange) and GAN-1 with soft constraints (green) compared to the Black-Scholes implied volatility (blue). 115

4.6 The implied volatility approximated by GAN-2 with no constraints (orange) and GAN-2 with soft constraints (green) compared to the Black-Scholes implied volatility (blue). 116

4.7 The implied volatility computed by GAN-1 without MSE (red) and GAN-2 without MSE (green) compared to the Black-Scholes implied volatility (blue). 118

4.8 Implied volatility cross section generated by the IV-ANN for different maturities. 119

4.9 GAN-2 implied volatility (top) and IV-ANN implied volatility (bottom repricing error heatmap. The warmer colors (yellow) respresent higher error values and colder colors (blue) represent lower error values. 119

4.10 Implied volatility generated by the generator (left) and the market implied volatility computed by Brent's method (right). 120

4.11 Local volatility surface generated by trained VAE using a shallow network (left), VAE using a deep network (middle), and computed from implied volatility using Equation (4.4) (right). 121

A.1 The optimal portfolio (top) and trade rate (bottom) for Agents 1 (blue) and 2 (orange). For a good economy (right) and bad economy (left) when $\phi = 0.7$. 145

List of Tables

2.1 2-dimensional geometric average call option: computed prices and deltas at $t = 0$ . . . 41

2.2 5-dimensional geometric average call option: computed prices and deltas at $t = 0$ . . . 41

2.3 30-dimensional geometric average call option: computed prices and deltas at $t = 0$ . . . 42

2.4 100-dimensional geometric average call option: computed prices and delta at $t = 0$ . . . 43

2.5 F1-score calculated using (2.28) for multi-dimensional American style geometric average call option under time constraint. . . . 43

2.6 Multi-dimensional geometric average call options. Computed (a) portfolio mean and (b) portfolio standard deviation of the relative P&Ls, subject to 250 hedging intervals. . . . 47

2.7 2-dimensional Max call: computed option prices and deltas at $t = 0$. . . . 48

2.8 2-dimensional Max call: Deep OTM and ITM computed option prices and deltas at $t = 0$. . . . 49

2.9 5-dimensional max call option: computed prices and deltas at $t = 0$ . . . 52

2.10 30-dimensional max call option: computed prices and deltas at $t = 0$ . . . 53

2.11 F1-score calculated using (2.28) for 2-dimensional American style max call option under time constraint. . . . 53

2.12 2-dimensional Max call option: Means and standard deviations of the relative P&Ls by finite difference versus the proposed method, subject to 100 hedging intervals . . . 54

3.1 Residual U-net with self-attention approximation for the solution to the optimal value $u(x_0, 0)$ over different number of time-steps N. The FDM values are results from [3]. The relative errors are computed using (3.21) . 79

3.2 Residual U-net with self-attention approximation for the solution to the mean-variance criterion $u(x_0, 0)$ over different number of simulations M. In the FDM solution M represents the grid size used for the wealth [3]. The relative error is computed using (3.21) 80

3.3 The residual U-net with self attention approximation to the solution of the optimal value $u(s_0, b_0, \alpha_0, 0)$ for different maturities T in years. The analytical solution is given by (3.22). We calculate the relative error using (3.21). 81

3.4 The residual U-net with self-attention approximation to the optimal value, $u(s_0, b_0, \alpha_0, 0)$, for different maturities T in years. The analytical solution is given by (3.23). We calculate the relative error using (3.21). 82

4.1 Model Parameters used in GAN for GAN-1 and GAN-2. 110

4.2 Domain of parameters used in the simulation of training and testing data, all parameters were drawn randomly from the parameter space. 111

4.3 Domain of parameters used in out-of training volatility surface generation, all parameters were drawn uniformly from the domain. 112

4.4 We compare the performance of GAN-1 and GAN-2 using training time, MAE and MAPE. 114

4.5 Number of butterfly arbitrage violations detected in implied volatility surface. 117

4.6 Number of calendar arbitrage violations detected in implied volatility surface. 117

4.7 Timing and error comparison between GAN-2 and IV-ANN 118

A.1 Time and memory comparison between methods. The American style geometric average call option with $d = 2, 5$, and 10, priced using $M = 50000$. We used a Recurrent layer size of $L = 7$ for the proposed method. We used a timestep skip, $J = 4$ and $L = 7$ for the DRL method. 141

A.2 Average Sharpe ratio of sellers and buyers in good market conditions, poor market conditions and mixed market conditions. 146

A.3 Average Sharpe ratio of sellers and buyers in good market conditions, poor market conditions and mixed market conditions. 147

A.4 Average Sharpe ratio of sellers and buyers in good market conditions, poor market conditions and mixed market conditions. 147

Chapter 1

Introduction

Machine learning (ML) has been shown to be an effective framework to solve many different types of data driven problems. It has been used in applications that are driven by real data for pattern recognition and classification tasks with known labels. It is still unclear as to what extent ML can be applied in Finance. Recent developments have been pushing to apply machine learning to solve more quantitative problems such as pricing, optimal trade execution and model calibration in high dimensions. These problems are often represented as high-dimensional partial differential equations (PDEs) and methods have continue to be developed to solve them using machine learning. Methods such as deep backward stochastic differential equation (BSDE) methods [71, 49, 86] and the deep Galerkin method (DGM) [121] have shown remarkable capabilities in solving high-dimensional PDEs. The deep BSDE method and the deep residual learning method have been applied to solve Hamilton-Jacobi-Bellman (HJB) equations in finance [34].

However, current applications of the deep BSDE are inefficient in training. The inefficiencies are a result of learning an unknown function that satisfies the HJB equation. To make the problem more difficult, the control is dependent on the unknown function that satisfies the HJB equation. On top of this, at each timestep the unknown function must be approximated by a neural network. In addition, the labels to train the neural network are unknown. To overcome this, the labels need to be generated by a trained neural network from the previous timestep. In order to ensure that the approximation is accurate, we need to train and sample over many epochs. This naturally takes a long time since a problem may require 100 to 200 timesteps, which means we need to train 100 to 200 different neural networks. It also means the memory required to store the neural network model can be costly as each network may have large number of parameters. Ultimately, the goal of this work is to develop efficient methods using machine learning methods to increase efficiency.

In this thesis, we develop methods to overcome the inefficiencies in current machine learning methods to solve HJB equations that arise in finance. One method to improve efficiency is to use recurrent neural networks [36] and time embedding [132]. This means we decouple the dependencies through time and treat it as a feature of the model. To solve the HJB equation, we can approximate labels at each timestep using knowledge about the problem.

To illustrate this approach, we look at two different problems in finance. We use the high-dimensional American option pricing problem to demonstrate the inefficiencies in the deep BSDE scheme [86, 34]. We propose approximating the labels at each timestep by using knowledge about the previous option price. This allows us to use a recurrent network to approximate the solution of the HJB equation. We show that our proposed method reduces the computational time and memory requirements.

Sometimes, we cannot use this method since we may not be able to estimate the value function at each timestep. To overcome this we propose to use a self-attention network with time-embedding coupled with a neural network to learn the labels at each timestep. Self-attention coupled with neural networks has been shown to learn spatial and temporal dependence [132] overcoming the need for recurrent networks. This allows us to take advantage of the recursive nature of our problem and allows us to train the network sequentially without the need to estimate the value function at each timestep.

One application that illustrates this is the optimal trade execution for multiple unique agents. In this application, the value function at each timestep is unknown. By using a deep neural network with self-attention we can estimate the value function at each timestep. Using the gain in efficiency from machine learning allows us to extend the current scope of applications in finance, which is limited in dimensions due to time and memory constraints. We present a new time-consistent approach to the optimal trade execution problem for multiple-agents and assets under mean-variance criteria. The extension to multiple agents and assets allows us to address the short comings of the original Almgren-Chriss model [7]. Classical methods to solve the Almgren-Chriss model can currently solve the problem for one asset one agent with a 4^{th} order complexity [55]. Mean-field game has been explored for optimal trade execution in a high frequency setting for multiple agents. This framework, assumes there is one representative major investor with multiple minor investors, our ML framework allows us to overcome this issue by defining each agent uniquely by their own HJB equation. The mean-field game is again restricted to one asset [139].

Another method to improve the efficiency of ML methods to solve problems arising in finance is by aiding the training of the neural network. More specifically, in this thesis, we explore two methods to improve training time. In one application, we use a GAN

framework to compute implied volatility surfaces with minimal arbitrage violations. There are many works that address the computation of implied volatility surfaces using machine learning [97, 78, 111, 128, 43, 84, 98, 81]. However, none of these methods address the potential arbitrage violations present when computing implied volatility points outside of training data. We propose to use a discriminator neural network to restrict the possible outputs of the generator neural network such that the output volatility surface has minimal arbitrage violations. Restricting the output of the generator allows us to compute implied volatility surfaces with minimal arbitrage violations using shallow networks. Using shallow networks allows us to increase efficiency without sacrificing accuracy.

Another application, we investigate to improve training efficiency is in image denoising. Even though, image denoising is not directly related to financial applications, diffusion models are becoming prevalent in finance literature. Diffusion models are being explored to generate financial tabular data for stress testing and economic studies [117]. We propose a pre-training computation step where the numerical solution to the KFP equation is used as a label embedding into the image data. This has been shown to speed up the training of score-based diffusion generative models.

In this chapter, first we provide a formal background on the HJB equation. We explore the connection between the HJB equation and the backward stochastic differential equation (BSDE). Then we review some fundamentals of machine learning and summarize our contributions. Finally, we discuss three applications in finance and current machine learning methods used to solve them.

1.1 Hamilton-Jacobi Bellman Equations

An HJB equation is a non-linear controlled PDE and originates from optimal control theory in dynamical systems. Let X be a random variable at time t. We consider a continuous dynamical system in space-time starting at a *state* (X, t) with a *control variable* v that controls the future evolution of the state. The aim is to optimize the cumulative *objective function* over a simulated trajectory.

An example of an optimal control problem that arises in finance is the optimal asset allocation problem. Consider a market agent looking to optimize their investment in a risky asset. The agent can choose the quantity (control variable) to buy/sell when they observe the asset price (state) that will maximize their wealth (objective function) at the end of their investment horizon. This can be modelled using the HJB equation.

The HJB equation can be solved using classical numerical methods such as finite differences. However, due to the curse of dimensionality, these methods become inefficient when the dimensionality is greater than 3. Statistical methods can be used for dimensions greater than 3 up to 20, but they degrade in quality as dimensionality increases [121, 34]. Many frameworks to overcome the curse of dimensionality using neural networks have been proposed, with two methods that are most widely adopted [71, 49, 121, 86, 98]. One method is to formulate the HJB equation into a BSDE [71, 86]. Another method is to construct a neural network basis function to learn the PDE solution [121]. In this thesis, we use the BSDE approach to derive problem specific loss functions and design machine learning architectures that efficiently solve our HJB equation.

We define the HJB equation formally here. For an underlying stochastic process $X(t)$, we construct the HJB equation using Ito's Lemma. Before we present Ito's Lemma we need to define a *martingale*, *quadratic variation* and *covariation*.

Definition 1.1.1. (Martingale). A martingale is an adapted stochastic process $X(t)$ such that $\forall\ 0 \leq s < t < \infty$, $\mathbb{E}[X(t)|\mathcal{F}] = X(s)$ where $\mathcal{F}$ is the filtration on the probability space $(\Omega, \mathcal{F}, \mathbb{P})$. The expectation is taken over the random variable X conditional on the filtration $\mathcal{F}$. A martingale is said to be square integrable if $\mathbb{E}[X(t)^2] < \infty$, $\forall\ t \geq 0$.

Definition 1.1.2. (Quadratic Variation). Let $0 \leq t_1 \leq ... \leq t_{n-1} \leq t$ be the partition of $[0, t]$. Then the quadratic variation is given by:

$$\lim_{\max_{1\leq k\leq n}|t_k - t_{k-1}|\to 0} \sum_{k=1}^{n} (X(t_k) - X(t_{k-1}))^2.$$

Definition 1.1.3. (Covariation). For square integrable $X(t)$ and $Y(t)$, the covariation is given by:

$$\lim_{\max_{1\leq k\leq n}|t_k - t_{k-1}|\to 0} \sum_{k=1}^{n} (X(t_k) - X(t_{k-1}))(Y(t_k) - Y(t_{k-1})).$$

We state Ito's Lemma as:

Lemma 1.1.4. *(Ito's Lemma). Let $X_1(t), ..., X_d(t)$ be d continuous martingales, and let $u : \mathbb{R}^+ \times \mathbb{R}^d$ be a $C^{1,2}$ function. Then*

$$u(X,t) = u(X,0) + \int_0^t \frac{\partial u}{\partial t}(X(s), s)ds + \sum_{i=1}^{d} \int_0^t \frac{\partial u}{\partial X_i}(X_1(s), ..., X_d(s), s)dX_i(s)$$
$$+ \frac{1}{2} \sum_{i,j=1}^{d} \int_0^t \frac{\partial^2 u}{\partial X_i \partial X_j}(X_1(s), ..., X_d(s), s)d < X_i, X_j > .$$

Where $< X_i, X_j >$ *is the covariation between* $X_i(t)$ *and* $X_j(t)$.

In this framework, we are interested in the optimal objective function which we call the *value function* $u(X,t)$ and the optimal control $v^*(X,t)$. The optimal control is the control variable that optimizes the objective function. The HJB equation couples the value function and the optimal control such that

$$\begin{aligned} \partial_t u(X,t) + \mathcal{L}u(X,t;v^*(X,t)) &= 0, \\ u(T,X) &= g(X), \end{aligned} \tag{1.1}$$

subject to

$$v^*(X,t) = \arg\max_v \mathcal{L}(X,t;v,u(X,t)), \tag{1.2}$$

where $\mathcal{L}$ is the generator on $u(X,t)$ given $v^*(X,t)$ with terminal condition $g(X(T))$. In financial applications, equation (1.1) typically has the following form

$$\begin{aligned} \partial_t u + \mathcal{L}u(X,t;v^*(X,t)) &= \partial_t u + \mu(X,t,v^*(X,t))\partial_X u(X,t) \\ &\quad + \frac{1}{2}\sigma(X,t,v^*(X,t))\sigma^T(X,t,v^*(X,t))\partial_{XX}u(X,t) - F(X,t), \end{aligned}$$

where ∂_t is the first derivative with respect to t, ∂_X is the first derivative with respect to X, ∂_{XX} is the second derivative with respect to X, $\mu(X,t,v^*(X,t))$ is the drift coefficient, $\sigma(X,t,v^*(X,t))$ is the diffusion coefficient and $F(X,t)$ is some growth condition. Optimizing (1.2) is equivalent to optimizing the objective function under dynamic programming [109]. The HJB equation is formed from two subproblems; the PDE in (1.1) and the optimization problem (1.2). Solving the HJB equation yields the solution to both equations simultaneously.

1.1.1 Backward stochastic differential equation

There is a deep connection between HJB equations and BSDEs. We can reformulate the HJB equation into a BSDE by using the Feynman-Kac formula, which forms a relationship between the PDE and the expected value of the value function. We state the Feynman-Kac formula as the following theorem:

Theorem 1.1.5. *Let* X *be some underlying stochastic process,* v^* *be the optimal control,* $g : \mathbb{R}^d \mapsto \mathbb{R}^d$ *and* $F : [0,T) \times \mathbb{R}^d \mapsto \mathbb{R}^d$ *be continuous functions. Let* $u : \mathbb{R}^d \times [0,T] \mapsto \mathbb{R}^d$ *be a* $C^{1,2}$ *function on* $[0,T) \times \mathbb{R}^d$*, such that*

$$\begin{aligned} \partial_t u + \mathcal{L}u(X,t,v^*) &= F(X,t), \text{ on } \mathbb{R}^d \times [0,T) \\ u(X,T) &= g(X(T)), \text{ on } \mathbb{R}^d. \end{aligned}$$

Suppose that

$$\max_{0\leq t\leq T}|u(X,t)|+\max_{0\leq t\leq T}|F(X,t)|\leq Ke^{a|X|^2}$$

$\forall\ X\in\mathbb{R}^d$ *and some* $K>0$ *and* $0<a<\frac{1}{2dT}$. *Then*

$$u(X,t)=\mathbb{E}[\int_t^T e^{-\int_t^T v^*(X,s)ds}F(X,s)ds+e^{-\int_t^T v^*(X,s)ds}g(X(T))].$$

Proof. The proof of Theorem 1.1.5 is given in [109]. □

We can construct the BSDE for a diffusion process. Let $X(t)$ be a diffusion process that satisfies the stochastic differential equation (SDE)

$$dX(t)=\mu(X,t)dt+\sigma(X,t)dW(t),$$

where $dW(t)$ is the increment of the Wiener process. The Feynman-Kac theorem states that the solution to our HJB equation may be represented (probabilistically) by [50]:

$$u(X,t)=\mathbb{E}[\int_t^T F(X,s,v^*(X,s))ds+g(X(T))]. \tag{1.3}$$

By applying Ito's formula (1.3) can be equivalently formulated as the integral form of the BSDE [50]

$$Y(t)=g(X(T))+\int_t^T f(X,s,v^*(X,s))ds-\int_t^T Z(X,s,v^*(X,s))dW(s),$$

with Z an adapted process. We let [109]

$$Y(t)=u(X,t)$$

and

$$Z(X,t,v^*(X,t))=\sigma(X,t,v^*(X,t))^T\partial_x u(X,t),$$

which gives us the BSDE

$$du(X,t)=-F(X,t,v^*(X,t))dt+\sigma(X,t,v^*(X,t))^T\partial_x u(X,t)dW(t). \tag{1.4}$$

1.2 Preliminary on Machine Learning

The analytical solution to the HJB equation (1.1)-(1.2) is usually infeasible and typically solved using numerical methods. There are numerous numerical methods to solve HJB equations using classical methods, including finite differences and Monte-Carlo methods [57, 58, 50, 100, 109].

More recently, machine learning methods have been proposed to solve problems arising in finance in order to address the curse of dimensionality [71, 49, 121, 34, 98]. However, there are still challenges in solving HJB equations using neural networks as mentioned before. In this section we will review the background of machine learning and how it can be applied to our applications. In this thesis, we propose supervised and reinforcement learning methods to improve efficiency through machine learning architecture design.

1.2.1 Supervised learning

In machine learning, the model learns a mapping of the input to the label. Suppose we have M samples. The input data is called a feature given by the vector $\mathbf{X} = [x_1, ..., x_M]^\top$ that gives us some information about the targets, $\mathbf{Y} = [y_1, ..., y_M]^\top$. Then the pair $(\mathbf{X}, \mathbf{Y})$, represents our dataset made of features and targets. The neural network model can be seen as a function $q(\mathbf{X}; \theta)$, where θ contains the parameters of the model. More discussion on neural network can be found in Section 1.2.3. Supervised learning computes the optimal parameters θ^* such that the difference between $q(\mathbf{X}; \theta)$ and $\mathbf{Y}$ are minimized according to some loss function, denoted by $L(q(\mathbf{X}; \theta), \mathbf{Y})$. More specifically, the supervised learning problem is given by:

$$\theta^* = \arg\min_{\theta} L(q(\mathbf{X}; \theta), \mathbf{Y}).$$

Once the optimal θ^* is found, the training is complete and we can test the model on a set of new unseen samples of inputs, X', and targets, Y'. If $q(X'; \theta^*)$ is approximates Y' successfully under some metric, i.e., accuracy, then the neural network is said to perform well out of training. It is not immediately clear that supervised learning can be used to solve the HJB equation. This is because we need explicit labels for each timestep. In this thesis, we develop appropriate approximations to allow us to solve HJB equations using supervised learning.

Supervised learning has seen success in financial applications with known inputs as shown in training deep neural networks to compute volatility surfaces using both real market data and synthetic data [98, 97, 2, 84, 32, 31].

1.2.2 Reinforcement learning

Reinforcement learning does not rely on explicit labels. Instead, reinforcement learning learns the best action an agent should take based on the interaction between an agent and its environment that optimizes an agent's utility function. We present a simplified diagram of the interaction between agent and environment in Figure 1.1.

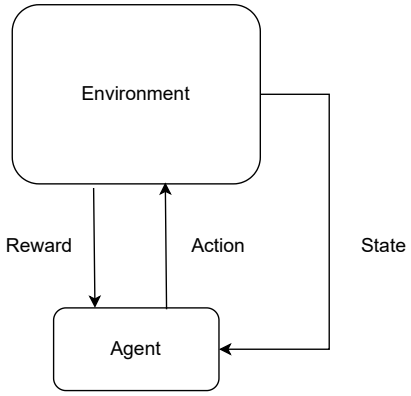


Figure 1.1 A simplified diagram of reinforcement learning. This diagram illustrates the interaction between the agent and environment

In reinforcement learning, the agent observes the state (X, t) from the environment. From the state the agent chooses an action to take based on their objective function. Then a reward based on the agents actions is computed and the environment is updated. A neural network can be used to approximate the value function, or the policy function of the agent. Iterative methods such as value iteration or policy iteration are often performed to learn the optimal value or policy. In our work, we focus more on value iteration. This is because solving the value function gives us an optimal solution which implies the optimality of the control. We also do not need to perform an additional update step, which saves on computation, especially since the HJB equation gives us a model to solve for the unknown value function.

The HJB equation can be reformulated as a reinforcement learning framework using approximate dynamic programming (ADP) to solve high-dimensional stochastic control problems [71]. This is done in two steps. The first step is to replace the true value function using an approximation. The second step is to advance forward in time with a backward sweep to update the value function. A rigorous formulation is given in [71]. Note that the forward approach can get stuck in poor local minima and potentially diverge from the true solution [86].

Standard RL algorithms like deep Q-networks (DQNs) have been used to solve HJB equations [76, 136, 96]. It has also been used for optimal trade execution with trade books to solve controlled partial integro-differential equation (PIDE) [103].

1.2.3 Neural networks

The basic unit of neural networks is the perceptron (a model of a single neuron) given by the non-linear function [68]:

$$\text{perceptron}(\mathbf{X};\theta) = a(\theta^T\mathbf{X}),$$

where $a(\cdot)$ is the activation function, e.g. ReLU, sigmoid [68]. The activation function gives the perceptron flexibility to fit different functions. An artificial neural network (ANN) is made from multiple layers of perceptrons and is also known as a multi-layer perceptron (MLP) or feedforward neural network (FNN). A schematic of the ANN is shown in Figure 1.2.

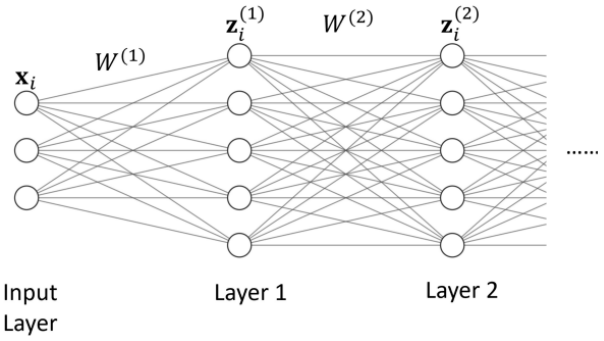


Figure 1.2 A 2 layer ANN model. Each circle called a *node* represents a perceptron and each layer is made up of multiple perceptrons [68].

An ANN has $\ell = 1, ..., l$ layers. We let $\ell = 1$ be the input layer, $\ell = 2, ..., l-1$ be the hidden layers and $\ell = l$ be the output layer. Let $W_\ell, b_\ell \in \theta_\ell$ be the weights and biases of the ANN. More specifically, the deep ANN is given by:

$$ANN(\mathbf{x};\theta) = a_l(a_{l-1}(\cdots a_2(a_1(\mathbf{x};\theta_1);\theta_2)\cdots;\theta_{l-1});\theta_l).$$

An ANN is considered deep when there are at least 4 hidden layers. The ANN has been shown to be a universal function approximator [68]. This property allows us to approximate

the unknown value function of the HJB equation as a neural network at each time period. The standard ANN can be used in combination with supervised and reinforcement learning to solve HJB equations. As example, we will briefly discuss two commonly used ANNs, namely, convolutional neural network and recurrent neural network.

Convolutional neural networks

Convolutional neural networks (CNN) are specialized neural networks that are designed to learn grid-like topology [68]. This is accomplished using the convolution operator between layers of ANNs. CNNs use the convolution operator instead of general matrix multiplication in at least one of their layers . More specifically the CNN takes a grid-like input X, i.e., a timeseries. The output called a *feature map* of a one layer convolution of X and the *kernel* W is given by [68]:

$$S(t) = (W * X)(t) = \sum_{s=-\infty}^{\infty} W(s)X(t-s),$$

where $*$ is the standard convolution operator. In practice, the infinite sum is truncated to a finite sum. Convolution is so powerful because of two properties: sparse weights between layers and parameter sharing between nodes [68].

The benefits of using CNNs in solving HJB equation is not immediately clear. This is because most timeseries data can be approximated by ANNs. However, a CNNs allow for more efficient neural network design through weight sharing [68]. In this thesis, we propose a model using CNNs to solve optimal trade execution problems in high-dimensions. It allows us to solve high-dimensional HJB equations with high-dimensionality in the control space.

Recurrent neural networks (RNN)

Recurrent neural network (RNN) were designed to approximate dynamic functions and sequences. It embeds temporal information using a *hidden state*. This hidden state acts like memory and ensures past information is carried into the future. The main advantage of using a RNN over traditional feedforward networks is that the RNN is developed to learn time dependent dynamic problems [51]. However, the hidden state may bias the training and lead to poor prediction performance. Over long sequences, there is a risk of forgetting, where the hidden state is dropped due to vanishing gradients [132].

The benefits for using RNN over ANNs and CNNs is that RNNs can present a solution for all time using a single network. ANNs and CNNs require multiple networks to solve time dependent problem because they are built on individual neurons that map specific inputs to outputs. RNNs introduce gates that augment the input based on past information (hidden state). This means each input may not result in the same output if the hidden states are different. This mechanism removes the need to train and save weights at each timestep. Solving the HJB equation using RNNs can be challenging because RNNs require a label for each timestep, but it may not be always available. Note, RNNs are shown to be universal approximators in [118].

1.2.4 Generative adversarial networks

Generative-adversarial network (GAN) are based on a game in which the generator network, G, must compete against an adversary or discriminator network, D. Let $\mathbf{Z} = [z_1, ..., z_M]^\top$ where $z_m \sim N(0, 1)$. In a standard GAN model, the learning objective is to ensure the distribution of $G(\mathbf{Z})$ is similar to the distribution of the labels $\mathbf{Y}$. The standard loss function for the GAN is given by the minimax problem [68]:

$$\min_G \max_D L(G, D) = \min_G \max_D \{\mathbb{E}[\log(D(\mathbf{Y}))] + \mathbb{E}[1 - \log(D(G(\mathbf{Z})))]\}$$

Solving the standard GAN loss function is difficult. In practice, we reformulate the standard loss function as two minimization problems given by:

$$\begin{aligned} \max_D L(D) &= \max_D\{\mathbb{E}[\log(D(\mathbf{Y}))] + \mathbb{E}[1 - \log(D(G(\mathbf{Z})))]\} \qquad (G \text{ is fixed}) \\ \min_G L(G) &= \min_G\{\mathbb{E}[1 - \log(D(G(\mathbf{Z})))]\} \quad (D \text{ is fixed}). \end{aligned}$$

The loss function for the GAN can be solved iteratively to maximize $L(D)$ first, and then $L(G)$. Solving iteratively can lead to vanishing gradients, but this can be mitigated by the choice of activation function and gradient clipping. Vanishing gradients can occur for some activation functions, when performing backpropagation because the derivative of the activation function with respect to the error can go to zero. Existing implementations of optimizers in python can also train both networks at once, but convergence in this case can be an issue. Minimizing the loss function drives the discriminator to learn to classify samples as real or fake and at the same time, trains the generator to evade the discriminator [68].

GAN is becoming popular approach in finance. GANs have been used to directly generate volatility surfaces [120]. However, directly applying GAN to financial problems

results in unrealistic outputs that are inconsistent with simulated data. If used correctly it can be a powerful tool to generate robust simulated data sets that satisfy the underlying distribution. GANs have also been used to aid training other neural network models used in finance [41]. They have also been used to generate dynamic implied volatility surfaces [39, 135]. In Chapter 4, we show how GAN can be used to aid in efficiently solving the inverse problem.

1.3 Applications in Finance

In this section, we present an overview of the three applications in finance. We provide background knowledge on three applications: American option pricing, optimal trade execution of an investment position and computing volatility surfaces. We discuss how machine learning is used to solve them and the improvements we propose to address the inefficiencies in current methods.

1.3.1 American option pricing

Pricing and hedging options is one of the fundamental problems in mathematical finance as it allows traders to price contracts fairly and hedge against risk. The most commonly traded options in the market are American options. An American option is a financial contract that gives a contract holder the right but not the obligation to buy or sell the underlying asset, i.e., stock, foreign currencies, interest rates, at an agreed upon price (*strike* price) at any time before the contract matures or expires. The act of executing the option, i.e., buying or selling the underlying asset, is called *exercising* the option. If the option can only be exercised at maturity, that is called an European option. Two common payoff structures for options are *call* and *put*. Mathematically, for an underlying asset with price S and a strike K, a call option has the payoff:

$$Call = \max\{S - K, 0\}$$

and the put option has the payoff:

$$Put = \max\{K - S, 0\}.$$

The holder of the American option wants to choose the best time to exercise their position during the life of the option in order maximize their gain. The additional exercise condition of the American option introduces a non-linearity which cannot be captured by

the traditional Black-Scholes option pricing formula. The Black-Scholes option price is the *continuation price* of the American option and we refer to the derivative of the value of the American option with respect to the price of the underlying as the *delta* of the option. There are many mathematical formulations for the price of the American option which includes the HJB formulation, linear complementarity problems, free boundary problems, and least squares formulations, etc. [25, 26, 27, 100, 54, 129]. The HJB equation couples the price and control. This means solving the HJB equation gives us both quantities simultaneously [34].

In this thesis, we propose to use recurrent networks to address the inefficiencies of the deep backward scheme. Using recurrent networks allows us to reduce the number of neural networks that need to be trained to solve the problem. In order to train the network, we derive and present a custom loss function using the continuation price of the American option. Our proposed method allows us to solve the American option pricing problem more efficiently than existing methods.

1.3.2 Optimal trade execution

The HJB equation also appears in the optimal trade execution problem. The optimal trade execution problem studies the rate at which a trading agent should liquidate their portfolio position of risky assets such that they maximize their returns, and minimize risk while keeping asset price fluctuations low [7]. The rate at which the agent liquidates is known as the trade rate. This is different from the asset allocation problem, as in the optimal trade execution problem the trading agent is trying to exit a specific position.

The value function of a trading agent is based on the agent's *utility* of their wealth. The utility function is a real-valued smooth function which describes the risk preferences of an agent. The optimal control can be given by the trading rate at which the trading agent should exit their position. Solving the HJB equation gives us the value function and the optimal trade rate. The advantage of using an HJB equation to solve the optimal trade execution is that it gives us a continuous framework which can be solved using classical numerical methods [130, 30, 59, 55, 137, 131]. There are other methods to solve the optimal trade execution problem such as dynamic programming and mean-field games [95, 140, 14, 133, 139] .

In this thesis, we extend the classic optimal trade execution problem to model multiple agents. Our framework allows us to couple each agent implicitly through the price of the asset. This creates an environment that can change depending on the decisions made by individual agents. We also propose a machine learning architecture to efficiently solve the

resulting system of HJB equations that govern the dynamics of each trader and a custom loss function that can be used to train the neural network that represents each agent.

1.3.3 Computing volatility surfaces with minimal arbitrage violations

To compute implied volatility surfaces for all maturities and strikes we need to solve the inverse problem. The inverse problem looks at estimating a parameter that allows the model to match observed data, i.e., the implied volatility which matches the Black-Scholes PDE to the market option price. This is done to compute a volatility surface, which is an important parameter that needs to be matched to market data as it is a signal to gauge price and sentiment in the market [94, 38]. The implied volatility is often used with other methods, i.e., local volatility to compute the prices of exotic options and ensures the exotic options prices are consistent with market prices.

Solving the inverse problem can introduce arbitrage opportunities in the implied volatility surfaces. This is because any interpolated point cannot guarantee that they are arbitrage-free. An arbitrage opportunity is a trading strategy where we earn a profit without taking any risk, i.e., if a market participant can purchase a good in one location for a lower price and sell it in another location for a higher price. Arbitrage introduces instability in the market and violates the assumptions of risk-neutral pricing. It is important for the implied volatility surface to be arbitrage-free as it is often used to price and hedge options.

In this thesis, we present a GAN framework to efficiently compute implied volatility surfaces using simulated data. This is in line with other works in this field to study the viability of an ML technique for this task [98, 97, 88, 32]. We show that GAN allows us to use shallow networks without losing accuracy.

1.4 Contributions

The main contribution of this thesis is twofold. Firstly, we design appropriate machine learning architectures to efficiently solve three different financial applications using supervised and reinforcement learning. This allows us to speed up computation and extend the current scope of models in finance. Secondly, we derive application specific neural networks and loss functions for our three applications. More specifically:

(I) Solve high-dimensional HJB equations more efficiently using machine learning. We demonstrate this for pricing high-dimensional American options, computing volatility surfaces and for optimal trade execution. In particular:

- We extend the framework of [34] and propose a deep Recurrent neural network (RNN) framework that solves the American option problem. By using deep RNNs the number of networks that need to be trained is reduced. This means we can speed up the computation time of high dimensional option pricing. We show that our proposed framework achieves a better time and memory complexity for ($d > 20$). Our method computes option prices and deltas faster and with less memory compared to the work presented in [34].
- We extend the optimal trade execution problem to K-agent setting. We use a residual network with CNN layers and self-attention to avoid using RNNs which are known to exhibit vanishing gradients over long sequences. Our architecture allows us to pass past approximations to the current approximation, effectively learning a feature map between current inputs and past outputs. The use of a CNN allows us to handle the high-dimensional nature of our control space and maps the inputs to the control grid.
- We utilize the dynamics between generator and discriminator network to increase the efficiency of our solution by allowing us to train shallow networks.

(II) Derive loss functions for specific applications in finance. We derive loss functions based on the HJB equation that governs the dynamics of each problem.

- In the American option pricing problem, we derive a loss function for the unknown value function and its derivative. We use the pathwise approximation of the continuation price and delta as initial approximations to the option value and delta. We solve the deltas of the American option pricing problem in all spacetime using our deep RNN framework. This has not been done by any other RNN option pricing work to the best of our knowledge at this time.
- In the optimal trade execution problem, we derive a loss function for each $k = 1, ..., K$ agents based on the extended model. This allows us to extend the scope of numerical approximations and allows for more accurate modelling of trading agents.
- We incorporate GAN dynamics to compute the implied volatility surface using a discriminator and generator. This allows us to train a shallow generator to efficiently compute implied volatility from data. We also show how our generator can be used to compute the local volatility.

1.5 Outline

This work is organized as follows. Firstly, in Chapter 2, we study in depth the machine learning solution for the HJB equation arising from the pricing and hedging of American options. We present a deep recurrent neural network (RNN) framework to solve the high-dimension problem using supervised learning by approximating the labels at each timestep. We derive the loss function to utilize the deep RNN formulation.

Secondly, in Chapter 3, we study the solution of the time-consistent mean-variance optimal trade execution problem for multiple agents. We reformulate the mean-variance optimal trade execution problem such that the objective function becomes a set of coupled equations: a dual representation and an auxiliary HJB equation. We propose a residual network (resnet) with CNN layers and self-attention in order to solve the auxiliary HJB equation using reinforcement learning. We also propose extensions to the optimal trade execution framework to incorporate interactions between multiple agents. We validate our results with analytical results where possible.

Thirdly, in Chapter 4, we investigate the solution of the inverse problem of implied volatility surfaces using supervised learning with minimal arbitrage violations. In this chapter, we propose a GAN framework to compute the implied volatility surface with minimal arbitrage violations. Finally, we conclude the thesis in Chapter 5 with a summary of our work and a discussion on future work.

Chapter 2

Efficiently Solving High-Dimensional American Option Pricing using Deep Recurrent Networks

2.1 Introduction

In this chapter, we study the efficient machine learning solution of HJB equations for pricing and hedging high-dimensional American options. The pricing and hedging of American options presents interesting practical and academic challenges, such as accurate and efficient computation of prices and the delta. In practical applications of hedging, we need to compute both the price and delta of the option across the entire spacetime [85, 73, 34]. The objective of our work is to compute option prices and deltas on the entire spacetime accurately and efficiently for high-dimensional American options.

Neural network solutions to the American option pricing problem are some of the most studied topics in neural networks applications in finance. This is because the capabilities of neural networks to overcome the curse of dimensionality is attractive to practitioners who price and hedge American options. The pricing and hedging of American options presents interesting practical and academic challenges, such as accurate and efficient pricing. However, many papers that look at neural networks in option pricing do not account for the extended training time required to train a network for each timestep to achieve good price estimates. This becomes more important when the trader on the trading floor needs to purchase an American option but needs an acceptable price for the contract. It is also turns out the trader will require the derivative of the option price with respect to

the underlying asset, called the delta of option, in order to hedge their position [85]. This is often done within a span of a few minutes and with limited computing resources. This is what motivated our research into recurrent neural network solutions to the American options problem.

There are numerous classical approaches to solve the American option problem such as the binomial tree method [85], numerical solutions to partial differential equations (PDE) with free boundaries or penalty terms [59], [1], [44], regression based methods [58], [100], [90], stochastic mesh methods [27], etc. The PDE method is the most accurate method to compute option prices and deltas throughout the options horizon. However, when the dimension of an American option, i.e. the number of underlying assets, is too large numerical solution of PDEs is practically infeasible as the complexity grows exponentially with the dimension. When the dimension, d, is moderate (e.g. $3 < d \leq 20$), the regression-based Longstaff-Schwartz method [100] is widely considered as the state-of-the-art approach for computing option prices. However, Figure 1 in [22] shows that using such regressed values as the spacetime solution is inaccurate. This is reinforced in Figure 8 in [34], which shows the clear advantages of using a neural network framework for pricing and hedging.

Outside of classical methods, there is a growing number of applications of neural networks on the American option problem. Some of the early approaches that solves the American option problem with neural networks use simple feedforward networks to approximate the price of the option for restricted number of assets ($d \leq 10$) as seen in [90] and [72]. Later works extend the scope and incorporate chained deep neural networks to solve the American option problem which we refer to as the deep BSDE method [49], [71], and [60]. The work of [70], follows the line of work of [71] and [49], but adds a regularizing term for added stability.

The work of [116] uses a neural network to learn option prices of the linear complementary problem of option pricing in an unsupervised manner. More recently, authors of [79] proposed a randomized reinforcement learning using recurrent networks to solve the American option problem. However, the methods mentioned do not provide methodology to determine the delta of an American option using neural networks.

Delta and price approximation for all spacetime using a deep residual network was proposed by [34]. In our work we refer to the method of [34] as the deep residual learning (DRL) method. The work of [86] provides theoretical convergence results for universal approximators. RNNs are shown to be universal approximators in [118]. The DRL method solves many of the issues of some earlier work. One major drawback is that it is both computationally expensive in time and memory because the weights need to be trained and stored at each timestep.

Deep optimal stopping method of [16] uses a neural network to learn the optimal stopping rule from Monte Carlo samples. This method uses a neural network to learn the stopping time decision function. This is an interesting approach however, comparison to this approach is not appropriate because the method can only compute prices over spacetime and does not offer methodology to compute deltas.

A physics-inspired neural network (PINN) approach called the deep Galerkin method (DGM) has been proposed [121]. The deep BSDE method and DGM are both methods to solve high-dimensional PDEs and are closely related through the PDE itself. The deep BSDE method re-represents the PDE as a BSDE through the Feynman-Kac formula. The PDE itself is used to construct the loss function for the DGM, whereas the BSDE is used to construct the loss function for the deep BSDE method. The DRL method makes extensive comparisons the the DGM method and shows that it is more accurate and computationally more efficient [34]. It also makes comparisons to Longstaff-Schwartz method with the same conclusion when dimensionality grows larger [34].

In this chapter, we use two deep recurrent neural networks to solve high dimensional American style options problems. One of our networks is used to approximate the continuation price and the other is used to approximate the delta. This is in line with the general formulation outlined in [86]. In our proposed method, we introduce a least squares formulation that approximates the solution to the underlying backward stochastic differential equation (BSDE). Our method solves for price and delta in all of space and time.

We outline the contributions of this paper:

- We extend the framework of [34] and propose a deep Recurrent neural network (RNN) framework that solves the American option problem. Using deep RNNs the number of networks that need to be trained is reduced, which means we can speed up the computation time of high dimensional option pricing. We show that our proposed framework achieves a better time and memory complexity for ($d > 20$). Our method computes option prices and deltas faster and with less memory compared to the work presented in [34].

- We introduce the approximation of deltas in all spacetime using our deep RNN framework, this has not been done by any other RNN option pricing paper to the best of our knowledge. The delta is computed using the pathwise derivative approach which is extended to general time t. We incorporate the domain knowledge of American options into our deep RNN framework to approximate accurate, time and memory efficient prices and deltas.

In contrast to the approach in [34], our approach presents a few key differences.

- *Different Neural Network Architecture*: The most obvious difference in our approach is the use of *two* deep recurrent network. One deep recurrent network learns the continuation price and the other learns the delta. We use deep recurrent networks to approximate the continuation price and delta of American option through time. The deep RNN allows us to effectively reduce the number of networks that needs to be trained and stored to two networks as opposed to multiple networks.

- *Different Optimization criteria*: Our loss function learns the continuation price and delta functions at each timestep to determine the price and delta of the option.

This chapter is organized as follows. Section 2.2, we formally introduce the American option problem and the resulting HJB equation. In Section 2.3, we discuss the BSDE associated with the American option, the discretization of the BSDE, and the resulting loss function. In Section 2.4, we present a pathwise approximation to the continuation price and delta which are used as labels for training the deep recurrent network. Section 2.5, we describe the deep recurrent neural network proposed to solve the American option problem. Section 2.6, we analyze the computational cost for time and memory. Section 2.7, we compare our network to previous neural network designs to illustrate the advantages of our framework. Section 2.8 concludes the chapter.

2.2 HJB Equation from the American Option Problem

2.2.1 American options

We provide an initial discussion on the American option in Section 1.3. We now review the American option problem more formally. Suppose we have a basket of d stocks; with price processes $\mathbf{S}(t) = [S_1(t), ..., S_d(t)]^\top \in \mathbb{R}^d$. Let $t \in [0, T]$ be the time up to maturity T, and let r be the interest rate. Let δ_i, $\mu_i = (r - \delta_i)$ and σ_i be the dividend, drift and volatility of stocks $i = 1, ..., d$. Let $\rho \in \mathbb{R}^{d\times d}$ be the correlation matrix and we define the correlated random variable as:

$$dW_i(t) = \sum_{j=1}^{d} L_{i,j}\psi_j(t)\sqrt{dt},$$

where $\psi_j(t) \sim N(0, 1)$ are i.i.d, and L is the Cholesky factor of ρ, i.e., $\rho = LL^\top$. Given a vector of initial prices, $\mathbf{s}(0) \in \mathbb{R}^d$, the price vector $\mathbf{S}(t)$ follows the dynamics given by

geometric Brownian motion. For each element i we have:

$$dS_i(t) = \mu_i S_i(t)dt + \sigma_i S_i(t)dW_i(t), \tag{2.1}$$

with initial condition $S_i(0) = s_i(0)$. For simplicity, we assume that the market has sufficient liquidity and we do not consider market frictions. Let $g(\mathbf{S}, t)$ be the payoff of the American option at the underlying price $\mathbf{S}(t)$. The payoff function is a function of $q(\mathbf{S}, t)$, which varies depending on the payoff style, i.e., the commonly seen max call option has a payoff of the form $q(\mathbf{S}, t) = \max_{i=1,...,d}\{S_i(t) - K\}$, with the payoff:

$$g(\mathbf{S}, t) := \max\{q(\mathbf{S}, t), 0\}. \tag{2.2}$$

The American option is more complicated to price than the European option with the same payoff because it can be exercised at any point t. To determine if we want to exercise or not, we need the continuation price.

Let $u(\mathbf{S}, t)$, be the continuation price of the American call option, i.e., the discounted payoff of the option when it is not exercised at time t at price $\mathbf{S}(t)$. We denote the optimal stopping time by $\tau \in [t, T]$, which is the best time to exercise the option between time t and maturity T. Then the continuation price can be written as:

$$u(\mathbf{S}, t) = \sup_{\tau\in[t,T]} \mathbb{E}[e^{-r(\tau-t)}g(\mathbf{S}, \tau); \mathbf{S}(t) = \mathbf{s}(t)]. \tag{2.3}$$

Note that we exercise the option when we determine that the payoff is greater than or equal to the continuation price. Then the value of the American style option is given by:

$$\begin{aligned} V(\mathbf{S}, t) &= \max\{g(\mathbf{S}, t),\ u(\mathbf{S}, t)\} \\ &= \begin{cases} u(\mathbf{S}, t) & u(\mathbf{S}, t) > g(\mathbf{S}, t) \\ g(\mathbf{S}, t) & \text{otherwise} \end{cases}, \end{aligned} \tag{2.4}$$

The optimal exercise boundary is the price of the asset such that:

$$u(\mathbf{S}, t) = g(\mathbf{S}, t). \tag{2.5}$$

In practical applications of hedging, we are interested in evaluating the delta of the American option. We denote the vector of delta as $\nabla_S V(\mathbf{S}, t) = [\frac{\partial V}{\partial S_1}(\mathbf{S}, t), ...\frac{\partial V}{\partial S_d}(\mathbf{S}, t)]^\top$. We compute the delta of the American style option as:

$$\nabla_S V(\mathbf{S}, t) = \begin{cases} \nabla_S u(\mathbf{S}, t) & u(\mathbf{S}, t) > g(\mathbf{S}, t) \\ \nabla_S g(\mathbf{S}, t) & \text{otherwise} \end{cases}. \tag{2.6}$$

Our goal is to compute both the option price $V(\mathbf{S}, t)$ and delta $\nabla_S V(\mathbf{S}, t)$ on the entire space-time in a time and memory efficient way.

2.2.2 HJB formulation

Let $\mathcal{L}u(\mathbf{S},t)$ be the differential generator arising from the Black-Scholes equation for the value function u defined as:

$$\mathcal{L}u(\mathbf{S},t) = -\partial_t u(\mathbf{S},t) - \frac{1}{2}\sum_{i,j=1}^{d}\sigma_i\sigma_j\rho_{i,j}\frac{\partial^2 u}{\partial S_i \partial S_j}(\mathbf{S},t) - \sum_{i=1}^{d}(r-q_i)S_i\frac{\partial u}{\partial S_i}(\mathbf{S},t) + ru(\mathbf{S},t).$$

Then the HJB equation that arises for the American option pricing problem is given by:

$$\begin{aligned}\min\{\mathcal{L}u(\mathbf{S},t), u(\mathbf{S},t) - g(\mathbf{S},t)\} = 0,\ &\in \mathbb{R}^d \times [0,T)\\ u(\mathbf{S},T) = g(\mathbf{S},T),\ &\in \mathbb{R}^d.\end{aligned}$$

We can write the HJB equation in an equivalent form given by:

$$(1 - v^*(\mathbf{S},t))\mathcal{L}u(\mathbf{S},t) + v^*(\mathbf{S},t)(u(\mathbf{S},t) - g(\mathbf{S},t)) = 0$$

subject to

$$v^*(\mathbf{S},t) = \arg\min_{v\in\{0,1\}}\{(1-v(\mathbf{S},t))\mathcal{L}u(\mathbf{S},t) + v(\mathbf{S},t)(u(\mathbf{S},t) - g(\mathbf{S},t))\}.$$

The control space is binary in this application where $v(\mathbf{S},t) = 1$ (or 0) indicates the option is exercised (or not exercised). The optimal control exercise decision is denoted $v^*(\mathbf{S},t)$ is chosen to maximize the option holders gain.

2.3 Loss Function Formulation from the BSDE equation

The American option pricing problem is often formulated as a Hamilton-Jacobi-Bellman (HJB) equation. This non-linear PDE is difficult to solve as we need to solve the PDE and the optimal control. Instead of working with the HJB equation, we reformulate the American option pricing problem into a BSDE with the following theorem:

Theorem 2.3.1. *(BSDE formulation) Assume that an American option is not exercised at time $[t, t+dt]$. Then the continuation price of the American option at time t for asset price $\boldsymbol{S}$ satisfies the following BSDE*

$$du(\boldsymbol{S},t) = rudt + \sum_{i=1}^{d}\sigma_i S_i\frac{\partial u}{\partial S_i}dW_i(t), \tag{2.7}$$

with terminal condition $u(\boldsymbol{S}, T) = g(\boldsymbol{S}, T)$*,* $S_i(t)$ *satisfies* (2.1) *and* r*,* σ_i *and* $dW_i(t)$ *are defined as in* (2.1)*.*

Proof. Readers interested in the complete proof are referred to [50, 20]. The proof is constructed using Ito's lemma. We can see that for one asset $i = 1$, the drift coefficient of the BSDE (1.4) $-F(S_i, t) = \partial_t u(S_i, t) \frac{1}{2}\sigma_i^2 \frac{\partial^2 u}{\partial S^2}(S_i, t) + (r - \delta_i) S_i \frac{\partial u}{\partial S_i}(S_i, t) = r u(S_i, t)$ and if the price u is smooth enough then we know that the diffusion coefficient $Z(S_i, t, \frac{\partial u}{\partial S_i}) = \sigma_i S_i \frac{\partial u}{\partial S_i}$ [109]. □

The significance of the BSDE formulation is twofold; one, it allows us to solve $u(\mathbf{S}, t)$, and $\nabla_S u(\mathbf{S}, t)$ simultaneously for all spacetime. If the approximation for the price is correct, then the approximation for the delta will also be correct, which allows us to perform a complete hedging process [34]. The second significance of the BSDE formulation is that it allows us to design a less expensive neural network compared to the HJB PDE [121]. This is because using the HJB PDE requires us to compute and store the Hessian tensor, which is an $\mathcal{O}(Md^2)$ tensor, where M is the number of samples used to train the neural network. Compared to the HJB formulation, the BSDE formulation only requires the Jacobian tensor of size $\mathcal{O}(Md)$.

2.3.1 Discretization of BSDE

In this chapter, we use the Euler method as in [34] and [86] to simulate SDE (2.1). Let $m = 1, .., M$ be the indices of M simulation paths and $n = 1, ..., N$ be the indices of the discrete timesteps from 0 to T. Let $\Delta t = T/N$ be the timestep size and let $t^n = n\Delta t$. We discretize $dW_i(t)$ as:

$$(\Delta W_i)_m^n = \sum_{j=1}^{d} L_{i,j} (\psi_j)_m^n \sqrt{\Delta t}.$$

We discretize (2.1) as:

$$(S_i)_m^0 = (s_i)_m^0, \qquad i = 1, .., d \tag{2.8}$$

$$(S_i)_m^{n+1} = (1 + (r - \delta_i)\Delta t)(S_i)_m^n + \sigma_i (S_i)_m^n (\Delta W_i)_m^n, \qquad i = 1, ..., d. \tag{2.9}$$

We can discretize the payoff function at $n + 1$ as:

$$g(S_m^{n+1}) = \max\{q(S_m^{n+1}), 0\}.$$

For $n = N-1, ..., 0$, we discretize (2.7) as:

$$u_m^{n+1} = (1 + r\Delta t)u_m^n + \sum_{i=1}^{d} \sigma_i (S_i)_m^n \frac{\partial u_m^n}{\partial S_i} (\Delta W_i)_m^n, \tag{2.10}$$

where u_m^n is the discrete approximation to the continuation price, and $\frac{\partial u_m^n}{\partial s_i}$ is the discrete approximation to the delta.

We can then determine the control, v_m^{n+1}, defined as:

$$v_m^{n+1} = \begin{cases} 1, & u_m^{n+1} > g(S_m^{n+1}) \\ 0, & \text{otherwise.} \end{cases}$$

Finally, the value of the American option is discretized as:

$$V_m^n = v_m^n u_m^n + (1 - v_m^{n+1}) g(S_m^n),$$

with terminal condition:

$$V_m^N = g(S_m^N). \tag{2.11}$$

For high dimensions when d is large, solving (2.10) directly can be time consuming or even intractable. Instead, we use (2.10) to construct a least squares formulation which will be presented in Section 2.3.2. This is then used to compute the option price (2.4) and delta (2.6).

2.3.2 Least squares formulation of the BSDE

Consider the n-th timestep. For easier exposition, we will drop the dependence on S_m^n in order to simplify notation. We present the loss function we derived to train the deep RNN in the following proposition.

Proposition 2.3.2. *Let u_ϕ^n and $\nabla_S u_\theta^n$ be the approximation for the discrete continuation price u^n and the discrete delta $\nabla_S u^n$, respectively. We let $\nabla_S u_\theta^n = [\frac{\partial u_\theta^n}{\partial S_1}, ..., \frac{\partial u_\theta^n}{\partial S_d}]^\top$, then the loss function to train the neural network approximation u_ϕ^n and $\nabla_S u_\theta^n$ are given by:*

$$L_p(\phi, \theta) = \mathbb{E}[|u_\phi^n - u^n|^2] + \frac{\Delta t}{(1 + r\Delta t)^2} \mathbb{E}\left[\left| \sum_{i=1}^{d} \sum_{j=1}^{d} L_{i,j} \sigma_i S_i \left(\frac{\partial u_\theta^n}{\partial S_i} - \frac{\partial u^n}{\partial S_i} \right) \right|^2 \right], \tag{2.12}$$

where $\mathbb{E}[|(\psi_j)_m^n|^2] = 1$. Note that the authors of [86] show that the loss function (2.12) converges to the solution of the BSDE (2.7).

Proof. To derive the least-squares formulation of the BSDE (2.10), we first define the approximation error as:

$$e^{n+1} = u_\phi^{n+1} - u^{n+1}.$$

To approximate u_ϕ^{n+1}, y^{n+1} must satisfy (2.10). In other words, u_ϕ^{n+1} can be defined as:

$$u_\phi^{n+1} = (1 + r\Delta t)u_\phi^n + \sum_{i=1}^{d} \sigma_i S_i \frac{\partial u_\theta^n}{\partial S_i} (\Delta W_i)_m^n.$$

Now the error can be written as:

$$e^{n+1} = (1 + r\Delta t)(u_\phi^n - u^n) + \sum_{i=1}^{d} \sigma_i S_i \left(\frac{\partial u_\theta^n}{\partial S_i} - \frac{\partial u^n}{\partial S_i} \right) (\Delta W_i)_m^n.$$

Taking the expected value, the error equation becomes:

$$\mathbb{E}[e^{n+1}] = \mathbb{E}\left[(1 + r\Delta t)(u_\phi^n - u^n) + \sum_{i=1}^{d} \sigma_i \left(\frac{\partial u_\theta^n}{\partial S_i} - \frac{\partial u^n}{\partial S_i} \right) (\Delta W_i)_m^n \right].$$

Discounting the process, we have that $\mathbb{E}[e^{n+1}] = (1 + r\Delta t)\mathbb{E}[e^n]$. Consequently, we have:

$$\mathbb{E}[e^n] = \mathbb{E}\left[(u_\phi^n - u^n) + \frac{1}{1 + r\Delta t} \sum_{i=1}^{d} \sigma_i \left(\frac{\partial u_\theta^n}{\partial S_i} - \frac{\partial u^n}{\partial S_i} \right) (\Delta W_i)_m^n \right].$$

Then the mean squared error at n is given by:

$$\begin{aligned}
\mathbb{E}[|e^n|^2] &= \mathbb{E}\left[\left| (u_\phi^n - u^n) + \frac{1}{(1 + r\Delta t)} \sum_{i=1}^{d} \sigma_i \left(\frac{\partial u_\theta^n}{\partial S_i} - \frac{\partial u^n}{\partial S_i} \right) (\Delta W_i)_m^n \right|^2 \right] \\
&= \mathbb{E}\left[\left| (u_\phi^n - u^n) + \frac{\sqrt{\Delta t}}{(1 + r\Delta t)} \sum_{i=1}^{d} \sum_{j=1}^{d} L_{i,j} \sigma_i \left(\frac{\partial u_\theta^n}{\partial S_i} - \frac{\partial u^n}{\partial S_i} \right) (\phi_j)_m^n \right|^2 \right] \\
&\approx \mathbb{E}\left[|(u_\phi^n - u^n)|^2 + \frac{\Delta t}{(1 + r\Delta t)^2} | \sum_{i=1}^{d} \sum_{j=1}^{d} L_{i,j} \sigma_i \left(\frac{\partial u_\theta^n}{\partial S_i} - \frac{\partial u^n}{\partial S_i} \right) (\phi_j)_m^n |^2 \right],
\end{aligned}$$

□

We remark that we use the loss (2.12) over the loss of [34] for two reasons:

- The loss (2.12) does not require integration over time steps, reducing the need to store the results of each time step. This also allows us to train the recurrent network for all time in one step.
- In our approach, the delta is approximated by a neural network, which allows us to compute delta at $t = 0$ for any stopping time τ using the pathwise derivative method without having to worry if $\partial S/\partial S_0$ is evaluable. The pathwise continuation price is derived as shown in Section 2.4.

Given the loss function $L_p(\phi, \theta)$ we compute the optimal weights θ and ϕ by minimizing (2.12):

$$\{\phi^*, \theta^*\} = \arg\min_{\phi,\theta}[L_p(\phi, \theta)]. \tag{2.13}$$

Since the BSDE is a backward process, we start our computation at $n = N$, where the value of the option is given by the payoff (2.2). This means that the risk-neutral price of the option at $n = N-1$, must be the discounted value of the option. To train our network, we must find the continuation price and delta at time n such that the value of the option at n is given by (2.4).

2.4 Continuation price and delta at timestep n

In order to compute the optimal weights, we need to determine the continuation price, u^n, and delta, $\frac{\partial u^n}{\partial S_i}$, in (2.12). To determine u^n, we will solve the optimization problem (2.3). To determine $\frac{\partial u^n}{\partial S_i}$ is not as straightforward. Our derivation follows the work of [25] which shows a framework for estimating the delta of a European call options using the pathwise derivative. This was later extended to the American options in [129] and applied to neural networks in [34].

We see in [25] that when the underlying asset evolves following the process (2.1), $\frac{\partial S_i}{\partial S_0} = \frac{S_i}{S_0}$. Let $z_i \sim N(0, 1)$, we can extend this notion to the increment $[t, \tau]$ under (2.1) since:

$$\frac{\partial S_i(\tau)}{\partial S_i(t)} = e^{(\mu - \frac{\sigma^2}{2})(\tau - t) + \sigma\sqrt{(\tau - t)}z_i} = \frac{S_i(\tau)}{S_i(t)}. \tag{2.14}$$

Finally, given τ from (2.3), we can define the delta of the continuation price as:

$$\frac{\partial u}{\partial S_i}(S_i, t) = e^{-r(\tau - t)} \frac{\partial g}{\partial S_i(\tau)} \frac{\partial S_i(\tau)}{\partial S_i(t)}, \tag{2.15}$$

where $\frac{\partial S_i(\tau)}{\partial S_i(t)}$ is determined from (2.14). The pathwise derivative approach may not be applicable if the derivative $\partial S_i(\tau)/\partial S_i(t)$ cannot be evaluated. This may occur if the underlying asset does not follow the process (2.1). However, we are always able to evaluate (2.15). This is because our framework uses a deep RNN to approximate the continuation price of the option, and using autodifferentiation on the computation graph for the input variable $S_i(\tau)$ with respect to the realised asset price $S_i(t)$ we can always evaluate the derivative $\partial S_i(\tau)/\partial S_i(t)$.

Note that if we denote τ as $\tau = \tilde{n}\Delta t$ then the continuation price can be discretized using (2.3) as:

$$u_m^n \approx e^{-r(\tilde{n}-n)\Delta t} g(S_m^{\tilde{n}}). \tag{2.16}$$

Using (2.14) the delta can be discretized as:

$$\nabla_S u_m^n \approx e^{-r(\tilde{n}-n)\Delta t} \nabla_S g(S_m^{\tilde{n}}) \frac{S_m^{\tilde{n}}}{S_m^n}. \tag{2.17}$$

2.5 Deep Recurrent Neural Network Formulation

To solve (2.13) is non-trivial. This problem can be solved using some parameterized function, such as polynomials, to approximate u^n, which allows us to solve the optimization problem in the parameter space. Classical methods such as the Longstaff-Schwartz method are built on this principle. More modern methods as seen in [79] can be used to learn such functions. We note that methods such as the Longstaff-Schwartz method and the method proposed by [79] do not solve the BSDE problem outlined in (2.7).

A main drawback of the deep neural network approach in [34] is that they require a feedforward neural network for every time step. It may be somewhat improved by skipping every J timsteps, but it still requires to train N/J networks where N is the total number of timesteps. When N is large, training time can be very costly.

Our proposed approach is to use two deep recurrent networks to compute the approximate continuation price u_θ^n and delta $\nabla_S u_\phi^n$. The advantage of our deep Recurrent network approach is that we only require two networks. This leads to tremendous savings in time and memory for large N.

2.5.1 Sequence of neural networks

The main advantage of using a Recurrent network over traditional feedforward networks is that the network is able to learn time dependent dynamic problems [51]. There exist many

different types of RNNs and we refer the reader to [36] and [83]. In our approach, we use the gated recurrent unit (GRU) of [36] to solve the American option pricing problem.

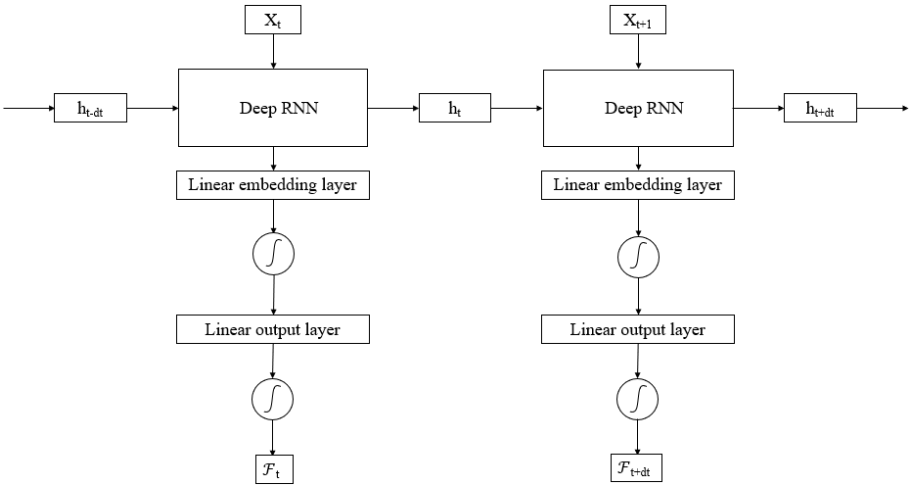


Figure 2.1 Deep recurrent network for price rolled out through time. Our network is composed of a recurrent unit, an embedding linear layer and an output linear layer. Takes in a matrix of inputs X, hidden information h and outputs $\mathcal{F}$.

In this chapter, we present two recurrent networks, rolled out through time, $\{\mathcal{F}^n(\mathbf{s};\phi)|n = N-1,...,0\}$ and $\{\mathcal{G}^n(\mathbf{s};\theta)|n = N-1,...,0\}$, where $\{\phi,\theta\}$ are trainable set of parameters. We will first present our recurrent network used to approximate the price, $u^n_\phi(\mathbf{s}) = \{u^n(\mathbf{s};\phi)|n = N-1,...,0\}$.

The arbitrage-free price of an option should differ from the previous price and delta by a function with magnitude $\mathcal{O}(\Delta t)$. As shown in [34], using a linear combination of the neural network output and the discounted value provides an estimate of the continuation price with lower variation, which improves the accuracy of regression.

We note that our problem is solved backwards in time, and we use "previous", "current" and "next" to refer to the $(n+1)^{\text{th}}$, n^{th}, and $(n-1)^{\text{th}}$ timesteps respectively. We define the user-defined coefficient $\alpha \in [0,1]$. The coefficient α determines how much of the approximation is given by the discounted value and the deep RNN approximation $\mathcal{F}^n(s;\phi)$.

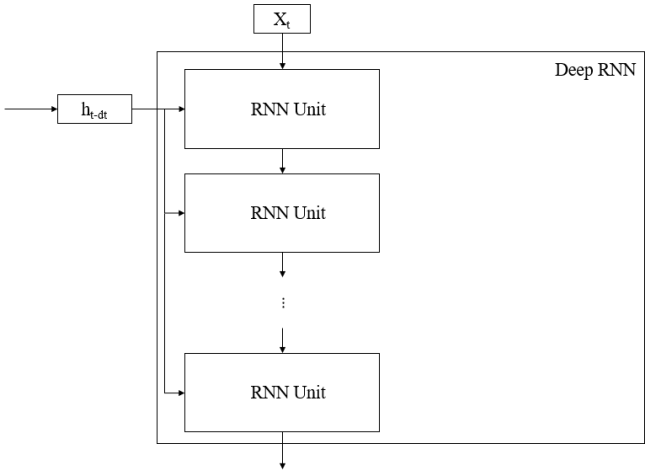


Figure 2.2 Stacked RNN units within the deep RNN

Mathematically, we approximate the continuation price as:

$$
\begin{aligned}
u_\phi^N(\mathbf{s}) &= g(\mathbf{s}^N) \\
u_\phi^n(\mathbf{s}) &= (1-\alpha)e^{-r(N-n)\Delta t} g(\mathbf{s}^N) + \alpha \mathcal{F}(\mathbf{s};\phi), \qquad n = N-1, ..., 0.
\end{aligned} \tag{2.18}
$$

The set of parameters ϕ includes all the weights and biases used in Algorithm 5.

Our deep RNN, $\mathcal{F}(\mathbf{s};\phi)$, is shown in Figure 2.1. We parameterize it by an L-layer deep RNN with GRU units with a linear embedding layer and output layer, as shown in Figure 2.2. The parameters $\phi_r^n \in \mathbb{R}^{d\times d}, \phi_z^n \in \mathbb{R}^{d\times d}, \phi_h^n \in \mathbb{R}^{d\times d}$ are trainable weights.

In the following part, we temporarily use a square bracket subscript for the layer index $l = 1, ..., L$. We construct the L-layer deep RNN for a continuation price as

- For layers, $l = 1$:

$$
\begin{aligned}
\mathbf{r}_{[l]}^n &= \hat{\sigma}(\phi_r^{n\top}[\mathbf{h}^{n+1}, \mathbf{X}^n]) \\
\mathbf{z}_{[l]}^n &= \hat{\sigma}(\phi_z^{n\top}[\mathbf{h}^{n+1}, \mathbf{X}^n]) \\
\hat{\mathbf{h}}^n &= \tanh((\phi_{\hat{h}}^{n\top})_{[l]}[\mathbf{r}_{[l]}^n \odot \mathbf{h}^{n+1}, \mathbf{X}^n]) \\
\mathbf{h}_{[l]}^n &= (1-\mathbf{z}_{[l]}^n) \odot \mathbf{h}^{n+1} + \mathbf{z}_{[l]}^n \odot \hat{\mathbf{h}}_{[l]}^n.
\end{aligned}
$$

- For layers, $l = 2, ..., L$:

$$\begin{aligned}
\mathbf{r}^n_{[l]} &= \hat{\sigma}(\phi_r^{n\top}[\mathbf{h}^{n+1}, \mathbf{h}^n_{[l-1]}]) \\
\mathbf{z}^n_{[l]} &= \hat{\sigma}(\phi_z^{n\top}[\mathbf{h}^{n+1}, \mathbf{h}^n_{[l-1]}]) \\
\hat{\mathbf{h}}^n &= \tanh((\phi_{\hat{h}}^{n\top})_{[l]}[\mathbf{r}^n_{[l]} \odot \mathbf{h}^{n+1}, \mathbf{h}^n_{[l-1]}]) \\
\mathbf{h}^n_{[l]} &= (1 - \mathbf{z}^n_{[l]}) \odot \mathbf{h}^{n+1} + \mathbf{z}^n_{[l]} \odot \hat{\mathbf{h}}^n_{[l]}.
\end{aligned}$$

- For the embedding layer:

$$\mathbf{e}^n = \text{swish}(\phi_e^{n\top}\mathbf{h}^n_{[L]}),$$

  where $\phi^n_e \in \mathbb{R}^{d\times d}$ are trainable weights.

- For the output layer:

$$\mathcal{F}(\mathbf{s};\phi) = \text{softplus}(\phi_{\mathcal{F}}^{n\top}\mathbf{e}^n),$$

  where $\phi^n_{\mathcal{F}} \in \mathbb{R}^{d\times d}$ are trainable weights. The swish activation function is given in (2.20)

We construct the deep RNN to compute the deltas $\mathcal{G}(\mathbf{s};\theta)$, in a similar way to $\mathcal{F}(\mathbf{s};\phi)$. The main difference between the two networks is in the output layers, where

$$\mathcal{G}(\mathbf{s};\theta) = \hat{\sigma}(\theta_{\mathcal{G}}^{n\top}\mathbf{e}^n),$$

where $\theta^n_{\mathcal{G}} \in \mathbb{R}^{d\times d}$ are trainable weights. Note that the sigmoid function is used here as it is the derivative of the softplus function with respect to the input.

To compute deltas, we define the coefficient $\beta \in [0, 1]$. Like α, β, determines how much of the approximation is given by the discounted delta and the deep RNN approximation $\mathcal{G}(\mathbf{S};\theta)$. Mathematically, we approximate the option delta as

$$\begin{aligned}
&\nabla_S u^N_\theta(\mathbf{S}) = \nabla_S g(\mathbf{S}^N) \\
&\nabla_S u^n_\theta(\mathbf{S};\theta) = (1-\beta)e^{-r(N-n)\Delta t}\nabla_S g(\mathbf{S}^N)\frac{\mathbf{S}^N}{\mathbf{S}^n} + \beta\mathcal{G}(\mathbf{S};\theta), \qquad n = N-1, ..., 0. \qquad (2.19)
\end{aligned}$$

The set of parameters θ includes all the weights and biases used in Algorithm 6.

Our deep Recurrent networks are structured as one-way stacked recurrent unit as shown in Figure 2.2. For $n = N-1, ..., 0$ our input, $\mathbf{X}^n$ passes through the first Recurrent unit. The subsequent Recurrent units use the output of the previous unit as the input of the next, while the previous hidden state, $\mathbf{h}^{n+1}$, is passed to all the recurrent units. The output of the deep RNN is passed through a linear embedding layer, an activation function, then a linear output layer, and a final activation function which is chosen based on the domain information of the problem.

2.5.2 Activation function selection for output and embedding layers

The choice of suitable activation functions leads to better predictive capabilities for any network [68]. As shown in Figure 2.1, our network adds two linear layers to the output of the deep recurrent network, a linear embedding layer and a linear output layer. We denote the embedding layer as $\mathbf{e}^n$, $\mathbf{o}_p^n$ for the output layer of the price approximation, and $\mathbf{o}_\Delta^n$ for the output layer of the delta approximation. For continuation prices the natural activation function used for the output layer is the popular rectilinear unit (ReLU) activation function [68]. The ReLU function has the form:

$$\text{ReLU}(x) = \max\{x, 0\},$$

which has the same form as our payoff function $f(x)$. However, the ReLU activation function suffers from vanishing gradients if a network is trained for too many iterations. Instead, we use the softplus activation function given by:

$$\text{softplus}(x, \kappa) = \frac{1}{1 + e^{-\kappa x}}.$$

This activation function approaches the ReLU activation as $\kappa \to \infty$. This gives us a good smoothing of the payoff and its exponential behaviour as it evolves through time. For the output layers of the network that approximate the option deltas, we use a sigmoid activation function, $\hat{\sigma}(\cdot)$ [68]. This selection is natural as the sigmoid function is the derivative of the softplus function.

For the embedding layer, we use the swish activation function given by:

$$\text{swish}(x) = x\hat{\sigma}(x). \tag{2.20}$$

The swish activation function is empirically shown to perform better than the ReLU function, as shown in [112] and helps our method reach more accurate solutions.

2.5.3 Training and evaluation

Consider training a recurrent network. Recurrent networks can be trained with a full sequence of events at once. For clarity, in this chapter, we will look at how to construct the data required for training at each timestep, n and all simulations m.

Case: $\mathbf{n = N}$. At the final timestep N, we let $u_\phi^N(\mathbf{S}^N) = g^N(\mathbf{S}^N)$ and $\nabla_S u_\theta^N(\mathbf{S}^N) = \nabla_S g^N(\mathbf{S}^N)$. We consider a smoothed payoff function given by using a scaled softplus activation function with a user defined parameter κ given by

$$u_\phi^N(\mathbf{S}^N) = g_\kappa^N(\mathbf{S}^N) \equiv \frac{1}{\kappa}\ln(1 + e^{\kappa q(\mathbf{S}^N)}), \tag{2.21}$$

where $q(S_m^N)$ is the same as in (2.2), let $\nabla_S g(\mathbf{S}^N) = [\frac{\partial g(\mathbf{S}^N)}{\partial S_1^N}, \cdots, \frac{\partial g(\mathbf{S}^N)}{\partial S_d^N}]$, then the derivative of the smoothed payoff is

$$\nabla_S u_\theta^N(\mathbf{S}^N) = \nabla_S g_\kappa^N(\mathbf{S}^N) \equiv \hat{\sigma}(\kappa g(\mathbf{S}^N))\nabla_S q(\mathbf{S}^N). \tag{2.22}$$

The hidden states are initialized as $g_\kappa^N(\mathbf{S}^N)$ for $\mathcal{F}(\mathbf{S}^N;\phi)$ and $\nabla_S g_\kappa^N(\mathbf{S}^N)$ for $\mathcal{G}(\mathbf{S}^N;\theta)$. The initial value of the continuation price is given by $u^N(S_m^N) = g_\kappa^N(S_m^N)$, which is the payoff at $t = T$. Similarly, the initial delta is given by $\nabla_S u^N(\mathbf{S}^N) = \nabla_S g_\kappa^N(\mathbf{S}^N)$. The initial hidden states are given by

$$\mathbf{h}^N = [g_\kappa^N(\mathbf{S}^N)],$$

for the pricing network and

$$\mathbf{h}^N = [\nabla_S g_\kappa^N(\mathbf{S}^N)],$$

for the delta network.

Case: $\mathbf{n = N - 1, ..., 0}$. We use the deep RNN framework to compute the continuation price and delta of the option at timesteps $n = N-1, ..., 0$. We construct the training input as

$$X^n = [\mathbf{S}^n, g(\mathbf{S}^n)]^T, .$$

Then we determine the optimal stopping time $\tilde{n}$ as:

$$\tilde{n} = \max_{\tilde{n}\in[N,n]} \{e^{-r(N-n)\Delta t}\mathbb{E}_t[q(\mathbf{S}^N)]\}$$

Finally, we update the continuation price and delta of the option using (2.16) and (2.17).

Minimizing the loss functions gives us the optimal set of parameters. We can express this as

$$\{\phi^*, \theta^*\} = \arg\min_{\{\Phi,\Theta\}} \{L(u_\phi, \nabla_S u_\theta)\}. \tag{2.23}$$

To solve (2.23) an optimization algorithm like Adam [92] is used. The set of optimal parameters naturally gives us ϕ^*, and θ^* which solves the problem (2.12).

The full algorithm for training is presented in Algorithm 1. We let $\mathbf{h}_\Delta^N$ be the hidden state of the delta approximating network. The max function is the element-wise maximum

Algorithm 1 Training the recurrent network

Require: $S^n, Optimizer$
  initialize u,$\nabla_S u$,u_ϕ, and $\nabla_S u_\theta$ of shape ($N+1$,d,batch size)
  for S^n in Batch **do**
    $n = N$
    u^N,$\nabla_S u^N \leftarrow g^N(S^N)$,$\nabla_S g^N(S^N)$
    u_ϕ^N,$\nabla_S u_\theta^N \leftarrow g^N(S^N)$,$\nabla_S g^N(S^N)$
    $\{\mathbf{h}^N, \mathbf{h}_\Delta^N\} \leftarrow \{g^N(S^N), \nabla_S g^N(S^N)\}$
    for $n = N-1, ..., 0$ **do**
      $\mathbf{X}^n \leftarrow [S^n, g(S^n)]^\top$
      $\tilde{n} \leftarrow \max_{\tilde{n} \in [N-1,\dots,n+1]} \mathbb{E}_t[g(\mathbf{S})]$
      u^n,$\nabla_S u^n \leftarrow e^{-r(\tilde{n}-n)\Delta t} g^{\tilde{n}}(S^{\tilde{n}})$,$e^{-r(\tilde{n}-n)\Delta t} \nabla_S g^{\tilde{n}}(S^{\tilde{n}}) \frac{S^{\tilde{n}}}{S^n}$
    $u_\phi^{N-1,\dots,0} \leftarrow$ Algorithm 5
    $\nabla_S u_\theta^{N-1,\dots,0} \leftarrow$ Algorithm 6
    $\{\phi^*, \theta^*\} \leftarrow Optimizer(L(u_\phi, \nabla_S u_\theta))$

between its inputs. Once training is complete, we evaluate the trained network with a forward pass of the network. Similar to the training phase, the recurrent networks can evaluate the full sequence of events at once. The initial setup at time $n = N$ is the same and will not be repeated.

In the evaluation phase, we directly construct the evaluation input (2.5.3). With the hidden state at $n = N$ we use the trained network Algorithm 5 to approximate u_ϕ^{n*} and we use the trained network Algorithm 6 to approximate $\nabla_S u_\theta^{n*}$ for $n = N-1, ..., 0$ [1]. For the evaluation phase, we use the learned price, u_ϕ^{n*} to construct the optimal control given by

$$v^{n*} = \begin{cases} 1 & \text{if } u_\phi^{n*} > g_\kappa^n(\mathbf{S}^n) \\ 0 & \text{otherwise.} \end{cases}$$

Then the value update is given by

$$V^n \approx \max\{g^n(\mathbf{S}^n), u_\phi^{n*}\} \tag{2.24}$$

and the delta is updated as

$$\nabla_S V^n \approx (1 - v^{n*}) \nabla g^n(\mathbf{S}^n) + v^{n*} \nabla_S u_\theta^{n*} \tag{2.25}$$

The full algorithm for evaluation is presented in Algorithm 2.

[1] Algorithms 5 and 6 can be found in Appendix A.1.2

Algorithm 2 Evaluation of Recurrent network

Require: S^n, ϵ
  initialize V,$\nabla_S V$,u^*_ϕ, and $\nabla_S u^*_\theta$ of shape (N+1,d,batch size)
  for S^n in Batch **do**
    $n = N$
    $V^N, \nabla_S V^N \leftarrow g^N(S^N), \nabla_S g^N(S^N)$
    $(u^*_\phi)^N$,$(\nabla_S u^*_\theta)^N \leftarrow g^N(S^N)$,$\nabla_S g^N(S^N)$
    $\{\mathbf{h}^N, \mathbf{h}^N_\Delta\} \leftarrow \{g^N(S^N), \nabla_S g^N(S^N)\}$
    for $n = N-1, ..., 0$ **do**
      $\mathbf{X}^n \leftarrow [S^n, g(S^n)]^\top$
    $u_\phi^{[N-1,...,0]*} \leftarrow$ Algorithm 5
    $\nabla_S u_\theta^{[N-1,...,0]*} \leftarrow$ Algorithm 6
    $v^* \leftarrow (u^*_\phi \geq g(S^{[N-1,...,0]}))$
    $V \leftarrow \max\{g(S^{[N-1,...,0]}), u_\phi^{[N-1,...,0]*}\}$
    $\nabla_S V \leftarrow (1-v^*)\nabla_S g(S^{[N-1,...,0]}) + v^* \nabla_S u_\theta^{[N-1,...,0]*}$

2.6 Computational Cost of Deep Recurrent Network

In this section, we analyze the computational time and memory cost of our algorithm and compare the proposed method with the DRL method. The DRL method and other feedforward methods require the neural networks to be initialized and trained at each time step. The method in [34] attempts to reduce the RAM load by pricing at each time step and storing the results. However, this means that they must write to a hard drive at each time step, which introduces additional time inefficiencies and large memory overhead.

The time complexity of training and pricing for the DRL method has been shown to be $\mathcal{O}((\frac{c_1 N}{J} + c_2)NMLd^2)$ [34], where N is the number of time steps, M is the number of simulations, L is the number of neural network layers, d is the dimensions, J is the number of timestep skips, and $\{c_1, c_2\}$ are constants related to the batch size and number of epochs. Note that J is a small constant, typically around $2-5$. Thus the complexity is quadratic in N.

The memory complexity of the DRL method is calculated to be $\mathcal{O}(NMd)$ [34], for storing the asset prices, price approximation u^n_ϕ and the delta approximation $\nabla_S u^{n+1}_\theta$. However, this does not count the number of network weights and trainable parameters that must be computed and stored at each timestep. Taking into account all network parameters, the total memory complexity is $\mathcal{O}(NMLd^2)$. Our proposed method allows for

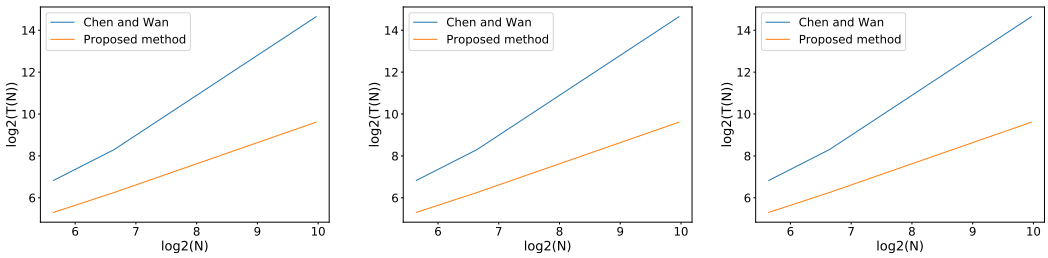
time and memory savings since we only need to train one set of weights rather than N/J sets of weights. In our analysis, we drop small constant factors on the time and memory complexities.

2.6.1 Time cost

We will separate the analysis of two cases; one at the final time and one for the other time steps.

Case 1: $\mathbf{n = N}$. At final time when $n = N$, the proposed method needs to compute v^{N*}, V^N and $\nabla_S V^N$, each of which is an $M \times d$ matrix. This is performed twice, once for Algorithm 1 and once for Algorithm 2. For each Algorithm 1 and Algorithm 2, the time complexity for v^N is of constant order. The evaluation of $g^N(S^N)$ and $\nabla_S g^N(S^N)$ over an array of size $M \times 2d$ both require addition which is $\mathcal{O}(Md)$. Thus, the time complexity requirement for N is $\mathcal{O}(Md)$.

Case 2: $\mathbf{n = N-1, ..., 0}$. At timesteps $n = N-1, ..., 0$, we need to construct the input X^n an $M \times 2d$ matrix, initialize and optimize the Recurrent networks y^n and ∇y^n. The construction of the input X^n and the initialization of the Recurrent network is done in constant time. Most of the operations required in Algorithm 1 are performed during optimization. For input size $M \times 2d$ our network has weights of size $2d \times 2d$ and an output weight of size $2d \times d$ for each of the L layers. The dominant operation in Algorithms 1 and 2 is multiplication of weights and inputs which requires $\mathcal{O}(MLd^2)$ operations. Thus, over N timesteps the total time complexity for Algorithms 1 and 2 is bounded by $\mathcal{O}(NMLd^2)$.



(a) $d = 2$ (b) $d = 5$ (c) $d = 10$

Figure 2.3 Logarithm of the total time, T(N), against the number of time steps, N, for the DRL method (blue line) and the proposed method (orange line) for dimensions (a) $d = 2$, (b) $d = 5$ and (c) $d = 10$.

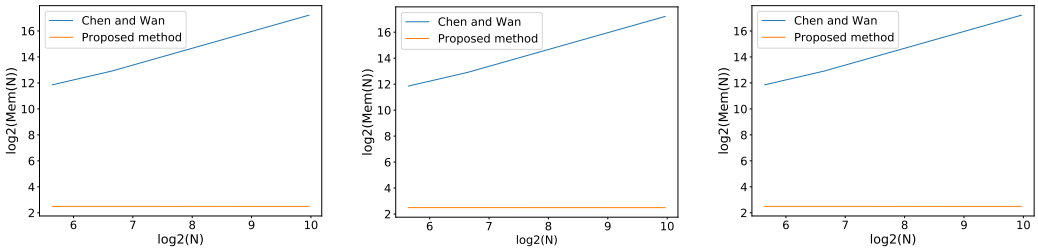
Using the Python library time, we can track the time requirements of the DRL method and compare it to our method; the total times used by both methods are presented in Table A.1. The complexity of the methods was measured by taking the logarithm of the timing results from Table A.1 and presented in Figure 2.3. Figure 2.3 shows that the DRL method has a worse than linear growth with respect to N and a near linear growth in time in our method. As shown in Table A.1, we can clearly see the absolute advantage we achieve by reducing the complexity by a factor of N, and our method at $N = 100$ is already 10 times faster than the DRL method. Also, the time results did not vary significantly between dimension $2, 5$ and 10, but showed more variation when the number of timesteps was increased.

2.6.2 Memory cost

Another contribution of our work is the reduction of memory complexity by a factor of N due to the use of a Recurrent network. This is a large reduction, since N is the dominant number in our complexity, since a large N is required for more accurate solutions.

The initialized asset prices, option payoff $g^n(\mathbf{S}^n)$ and delta $\nabla_S g^n(\mathbf{S}^n)$ require $\mathcal{O}(NMd)$ floating points of storage, which is the same as the DRL method [34]. Our method differs in terms of the network; thus, we only need to compare the storage requirements of the network. Since the proposed method only stores the training parameters in the final timestep, it is not required to store the training weights and gradient information in each timestep. Thus, for the input of size $M \times 2d$, hidden layers of size $2d \times 2d$ and output layer of size $2d \times d$, the Recurrent networks requires $\mathcal{O}(Md^2)$ floating points. This is performed over L layers, thus the final memory complexity is $\mathcal{O}(MLd^2)$ which is constant in N.

As for comparison, using the python library tracemalloc we measured the total peak memory requirements of the DRL method and compared it to our method. This was done by measuring the peak memory of each function and adding the accumulated peak memories. The total peak memory used by both methods is presented in Table A.1. The memory complexity of the methods were measured by taking the logarithm of the peak memory measured for a given number of timesteps N and presented in Figure 2.4. Figure 2.4 shows the near linear growth of memory in N for the DRL method and a constant growth in our method. Furthermore, the memory results did not vary significantly between dimensions $2, 5$ and 10, but showed more variation when the number of time steps was increased.



(a) $d = 2$ (b) $d = 5$ (c) $d = 10$

Figure 2.4 Logarithm of total peak memory versus the number of time steps, N, for the DRL method (blue line) and the proposed method (orange line) for dimensions (a) $d = 2$, (b) $d = 5$ and (c) $d = 10$.

2.7 Numerical Results

In this section, we solve the high-dimensional American option problem using our Algorithms 1 and 2. We compute the price $V(\mathbf{s}_0, 0)$ and the delta $\nabla V(\mathbf{s}_0, 0)$ at $t = 0$ for given $\mathbf{s}_0 = [s_0^1, \ldots, s_0^d]^\top$ where $s_0^1 = \cdots = s_0^d = 0.9K, K$ or $1.1K$. We use two separate types of experiments. In the first type we constrain all computational runtime such that the runtime for our proposed method and the runtime for the DRL method are the same. In the second type of experiment, we run the experiment until the pricing errors produced by our method matches the pricing errors of the DRL method. In this thesis, we do not test the neural network and loss function separately as the loss function was developed to train our RNN. The main focus of our work was to develop a method to solve high-dimensional American option pricing problem efficiently. However, if we consider using a traditional ANN instead of an RNN we would need a node for each grid point in space time and it is unclear whether any efficiency would be gained.

In our experiments we set the strike $K = 100$. The smoothing parameter in (2.21), is given by $\kappa = 2/\Delta t$. We use the parameters $\alpha = 0.5$ and $\beta = 0.5$. The network used in our Experiments is a $L = 5$ deep RNN with GRU units. The batch size for our proposed method was given as 10000. Our computation was performed using a 6GB-NVIDIA GTX 2060 GPU, a 6 core AMD Ryzen 5 3600X processor and 16GB of RAM.

In Experiment 1, we look at the American style geometric average option. We compare the accuracy of the proposed method to the DRL method for both price and delta at time $t = 0$ for $S_0 = s_0$ for all simulations, while under a similar runtime constraint. In the Experiments 2, we compare the price at $t = 0$, delta at $t = 0$ for a max call option under

similar runtime constraints. Experiments 3 and 4 evaluate how long the proposed method needs to run in order to match the accuracy of the DRL method. For Experiments 1 and 2 , finite difference solutions with very fine grids are used as exact solutions. We note that this is feasible only if $d \leq 3$. We make comparison results to the unsupervised method of [116], and also look at the deep out-the-money(OTM) and deep in-the-money(ITM) performance of our method, the DRL method and the Longstaff-Schwartz method in Experiment 2. In the case of ITM, at-the-money (ATM) and OTM, we do not make a comparison with the Longstaff-Schwartz method. This is because the Longstaff-Schwartz method has been compared with the DRL method in [34]. It has been shown in [34] that the DRL method is more efficient than the Longstaff-Schwartz method. In addition, we remark that the comparison is not made with the other methods referenced in the introduction, such as [70], [116], [121], [49], [15] and [71]. This is because the work of [121] was compared in [34]. The works of [49], [15] and [71] compute European option problems using neural networks based on BSDEs, these works do not discuss the more challenging American option problems and these methods have not been adapted to solve for deltas over spacetime. The work of [70] looks at solving the Black-Scholes and HJB equations using a BSDE approach however they look at solving the general PDE and not the American option problem. Their approach is also simulated with very limited number of timesteps, simulation size and the dimensions are limited to $d = 2$. We do not compare with methods that directly learn the boundary condition, such as the deep optimal stopping method shown in [10]. This is because this method uses a method similar to the Longstaff-Schwartz method to compute the option value, which has comparisons shown in [34] and in our method. It also requires a neural network to be trained at each timestep to get accurate optimal stopping decisions resulting in efficiency worse than the DRL method.

We note that when finite difference solutions are available, we can evaluate the absolute and percent errors of computed prices and deltas. More specifically, denote the finite difference solutions by v_{exact}. Then the percent error of the price at $t = 0$ is

$$\frac{|V(\mathbf{s}_0, 0) - V_{\text{exact}}|}{|V_{\text{exact}}|} \times 100\%, \tag{2.26}$$

and the percent errors of the delta at t=0 is

$$\frac{\|\nabla V(\mathbf{s}_0, 0) - \nabla V_{\text{exact}}\|_{L_2}}{\|\nabla V_{\text{exact}}\|_{L_2}} \times 100\%. \tag{2.27}$$

In addition, we evaluate the quality of computed exercise boundaries. In the first set of experiments, we look at the exercise boundary, i.e., a point (S^n, t^n) is 'exercised' or

'continued', computed by our proposed method and the DRL method, under run time constraint. The true exercise point is determined by finite difference method without any run time constraints. Let the 'exercised' class be the positive class and denote the numbers of true positive, true negative, false positive, and false negative samples as TP, TN, FP, FN, respectively. Then the quality of the exercise boundaries can be evaluated as

$$\text{f1-score} = \frac{TP}{TP + 0.5(FP + FN)}. \tag{2.28}$$

The best (or worst) case of the f1-score is 1 (or 0), respectively. Another common metric to evaluate the quality of classification problems is accuracy; however, since all of our experiments are skewed to the positive class, the F1 score would be a better metric than the accuracy [34]. We also show that our method can be used for delta hedging across spacetime, by demonstrating our hedging positions in a 2-dimensional case.

2.7.1 American style geometric average call option

We consider a d-dimensional American style 'geometric average' call option, where $\rho_{ij} = \rho$ for $i \neq j$, $\sigma_i = \sigma$ for all i's, and the payoff function is given by $g(\mathbf{S}, t) = \max\{(\prod_{i=1}^{d} S_i)^{1/d} - K, 0\}$. Although, geometric average options are rarely seen in practice, they have semi-analytical solutions for benchmarking the performance of our algorithm in high dimensions, as shown in [67], [121] and [34]. More specifically, it is shown in [67], that the d-dimensional can be reduced to a one dimensional American call option with the variable $s' = (\prod_{i=1}^{d} s_i)^{1/d}$, where the effective volatility is $\sigma' = \sqrt{(1 + (d-1))/d}\sigma$, and the effective drift, $\mu' = r - \delta + \frac{1}{2}(\sigma'^2 - \sigma^2)$. Thus, by solving the equivalent one-dimensional option using finite difference method, one can compute the d-dimensional option prices and under special circumstances, deltas[2] accurately.

In the following Experiments 1 and 2, we evaluated the American style geometric average call option under run time constraints. For each $s_0 = 90, 100, 110$ and $d = 2, 5, 30, 100$ we run our method and the DRL method for a fixed run time in seconds. Through this experiment we show that under time constraints, our method approximates the solution more accurately than the DRL method. We computed price of the American style geometric average call option as described in [121], with the parameters $\{T = 2, r = 0, \sigma = 0.25, \rho = 0.75\}$.

[2]We note that solving the equivalent one-dimensional option is not sufficient for computing the d-dimensional delta except at the symmetric points $s_1 = \cdots = s_d$. Interested readers can verify this by straightforward algebra.

Experiment 1: American style geometric average call option

In the first experiment we compare the computed prices at $t = 0$ between our proposed method and the DRL method under a run time constraint; see Tables 2.1 - 2.4. Each of our results Tables includes:

- The exact prices computed by the Crank-Nicolson finite difference method with 1000 timesteps and $16,385$ space grid points.
- The computed prices, the corresponding percent errors, and the run time of the DRL method.
- The computed prices, the percent errors and the run time of our proposed method.

For the proposed method, the percent errors range from $0.29\% - 2.29\%$ for the calculated prices. Our method performs noticeably better than the DRL method at lower dimensions as total sample sized used for $d = 2, 5, 30, 100$ was 200000. For comparison, for the DRL method, the percent errors range from $2.04\% - 21.01\%$. To ensure the computation did not exceed the runtime limit, the timestep was set to $N = 10$ and the skip parameter $J = 8$. This led to a serious degradation of accuracy for the DRL method.

The DRL method gives us deltas in all spacetime. Tables 2.1 - 2.4[3] also shows the deltas at $t = 0$ computed by our proposed method, along with the ones computed using the DRL method. The DRL method produces percent errors of the deltas ranged from 1.01% to as high as 37.71%; this is due to the limited sampling available under the run time constraint. The limited sampling leads to a poorly trained network and a degradation of the exercise boundary as shown in Figure 2.5. We see better performance from our proposed model under our time constraints, as we consistently see errors below 2.519%. Furthermore, we compare the exercise boundaries computed by our approach with the ones computed using the DRL method.

In Table 2.5 we evaluate the F1 score of the exercise boundary classification using (2.28). For the proposed method, the F1 score ranges from $0.8945 - 0.9545$. For the DRL method, the F1 score ranges from $0.6193 - 0.6651$. This illustrates the more consistent performance of our proposed approach under time constraints.

In Figure 2.5 we visualize the exercise boundaries computed by the proposed method and the DRL method. The results of our computation shows more points because the

[3]Only a single value of delta is reported for the geometric average, since our experiment was run for $S_1 = \cdots = S_d$ all deltas are equal as shown in [34].

Table 2.1 2-dimensional geometric average call option: computed prices and deltas at $t = 0$

(a) Finite Difference method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	6.7000	–	19.3136	–	–
100	11.2502	–	25.8041	–	–
110	16.7708	–	31.3886	–	–

(b) Proposed method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	6.6520	0.71348	19.0308	1.46425	36.39
100	11.0801	1.5112	25.3053	1.93303	36.74
110	16.8600	0.5313	31.1031	0.90957	36.21

(c) DRL method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.2920	21.0117	23.0260	19.2220	36.01
100	10.0960	10.2594	30.5032	18.2110	36.25
110	16.2260	3.24612	35.3606	12.6540	36.09

Table 2.2 5-dimensional geometric average call option: computed prices and deltas at $t = 0$

(a) Finite Difference method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	6.0685	–	7.3985	–	–
100	10.3173	–	10.0894	–	–
110	16.1580	–	12.8505	–	–

(b) Proposed method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	6.0820	0.2241	7.3345	0.86504	37.19
100	10.4070	0.8733	9.9467	1.41436	38.69
110	15.9818	1.08926	12.3168	4.153145792	37.54

(c) DRL method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.6789	6.41345	7.9648	7.6543	37.06
100	9.1800	11.0281	12.6274	25.155	38.23
110	13.1910	18.3589	17.9543	32.713	37.07

Table 2.3 30-dimensional geometric average call option: computed prices and deltas at $t = 0$

(a) Finite Difference method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.7740	–	1.2028	–	–
100	9.9660	–	1.6593	–	–
110	15.3390	–	2.0697	–	–

(b) Proposed method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.7200	0.93701	1.1944	0.69837	52.49
100	9.8960	0.70136	1.6307	1.72362	52.37
110	15.4680	0.8449	2.0405	1.41083	53.26

(c) DRL method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.6557	2.04375	1.1907	1.00599	52.44
100	8.3430	16.2839	1.9863	19.707	52.44
110	14.8900	2.92532	2.3296	12.557	53.17

Table 2.4 100-dimensional geometric average call option: computed prices and delta at $t = 0$

(a) Finite Difference method

s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.6352	–	0.3548	–	–
100	10.0658	–	0.4967	–	–
110	15.2953	–	0.6272	–	–

(b) Proposed method

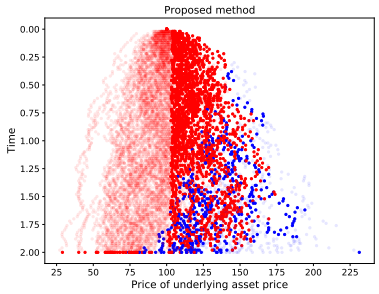
s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.6960	0.93701	0.3566	0.5073	212.74
100	9.9118	1.52993	0.4903	1.2885	205.08
110	15.3499	0.3570	0.6114	2.51913	206.21

(c) DRL method

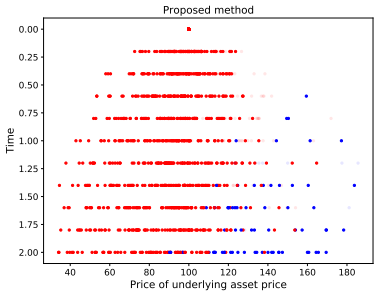
s_0	Price	Percent Error (%)	Delta	Percent Error (%)	Run Time (s)
90	5.9140	4.9404	0.4046	14.036	212.46
100	8.9989	10.5993	0.6195	24.723	205.59
110	16.2755	6.4085	0.6789	8.243	206.43

Table 2.5 F1-score calculated using (2.28) for multi-dimensional American style geometric average call option under time constraint.

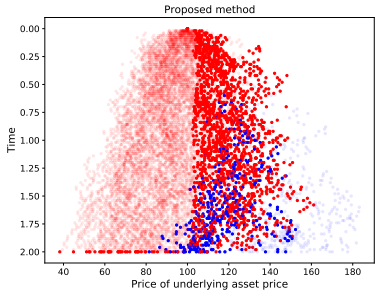
	Proposed method				DRL method			
s_0	$d = 2$	$d = 5$	$d = 30$	$d = 100$	$d = 2$	$d = 5$	$d = 30$	$d = 100$
90	0.9465	0.9486	0.9394	0.9545	0.6563	0.6550	0.6504	0.6565
100	0.9241	0.9193	0.9390	0.9213	0.6644	0.6651	0.6453	0.6550
110	0.8945	0.8953	0.9004	0.8956	0.6248	0.6403	0.6324	0.6193



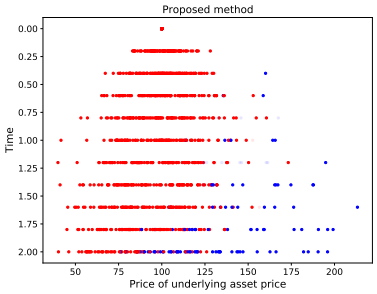
(a) Proposed method: $d = 5$



(b) DRL method: $d = 5$



(c) Proposed method: $d = 30$



(d) DRL method: $d = 30$

Figure 2.5 Multi-dimensional geometric call options. Comparison of the exercised sample points between the proposed method ((a) and (c)) and the DRL method ((b) and (d)). Points exercised (blue) and points continued (red); points misclassified as continued (bold dark blue) and points misclassified as exercised (bold dark red).

efficiency of our method allows us to use more timesteps for our simulation. Sample points are generated over the entire spacetime, i.e., $\{(S_n^m, t_n)|n = 0, \ldots, N; m = 1, \ldots, M\}$. Each sample point is classified using the proposed method or the DRL method. We use bold dark blue crosses to symbolize the sample points that are misclassified as continued (false positives), and bold dark red crosses to symbolize the points that are misclassified as exercised (false negatives). The plots show that under runtime constraints, the exercise boundary of our proposed method are more precise than the DRL method.

Experiment 2: delta hedging of American style geometric average call option

In this section we compare the delta hedging results calculated using the price and delta computed by the proposed method and the DRL methods, we present the portfolio mean and standard deviations for dimensions $[2, 5, 30, 100]$. We perform delta hedging simulations over the period $[0, T]$ using the proposed method for the geometric call option. We evaluate the quality visually using the density plotted of the relative profit and loss (P&L) as described in [59] and [73]. Relative P&L is given by:

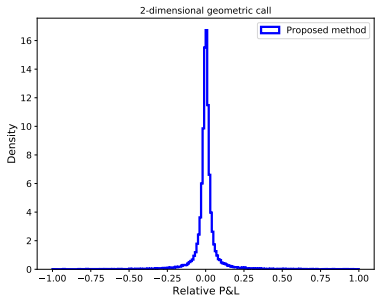
$$\text{Relative } P\&L \equiv \frac{e^{-rT}\Pi_T}{V_0},$$

where Π_T is the replicating portfolio at expiry T. The relative P&L for perfectly hedged portfolios should be a Dirac delta function [59].

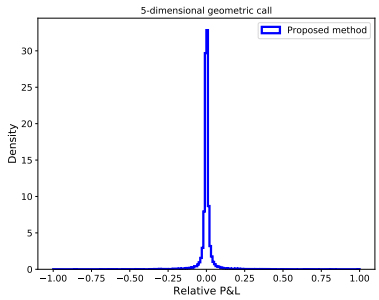
Due to the discretization of time, the relative P&L in numerical computation is a normal distribution with mean zero and small standard deviation proportional to Δt [34]. We did not compare the hedging results with the DRL method in this experiment as the hedging intervals were different. The computation of relative P&L requires both prices and deltas for the entire spacetime.

In Table 2.6 we show the calculated mean and standard deviation of the relative P&Ls for all 200000 simulation paths, using the proposed method. We computed values that are close to zero and standard deviations are also close to zero.

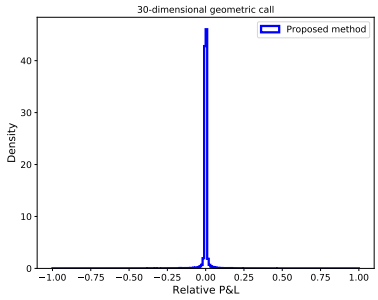
In Figure 2.6 we can see the densities of the relative P&Ls. The computed densities are normally distributed and approach the Dirac function with zero means. This further illustrates the capabilities of our approach in determining prices and deltas across spacetime. These results confirm that our methods provide a good hedge of assets across the spacetime.



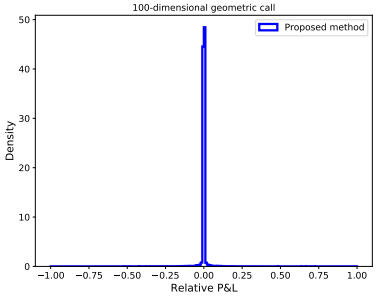
(a) $d = 2$



(b) $d = 5$



(c) $d = 30$



(d) $d = 100$

Figure 2.6 Multidimensional geometric call options. Density of the relative P&Ls computed by the proposed approach, subject to 250 hedging intervals at $s_0 = 100$ for dimensions (a) $d = 2$, (b) $d = 5$, (c) $d = 30$ and (d) $d = 100$.

Table 2.6 Multi-dimensional geometric average call options. Computed (a) portfolio mean and (b) portfolio standard deviation of the relative P&Ls, subject to 250 hedging intervals.

(a) Portfolio mean

s_0	$d = 2$	$d = 5$	$d = 30$	$d = 100$
90	-8.928e-8	-1.064e-8	-2.544e-9	7.430e-11
100	-9.805e-8	-5.697e-9	-1.347e-8	-1.931e-9
110	-1.774e-7	-4.014e-8	-1.426e-8	-4.312e-10

(b) Portfolio std dev

s_0	$d = 2$	$d = 5$	$d = 30$	$d = 100$
90	1.718e-4	5.711e-5	1.107e-5	3.329e-6
100	2.789e-4	9.691e-5	1.809e-5	5.446e-6
110	3.9398e-4	1.458e-4	2.656e-5	8.035e-6

2.7.2 American style max call option

We consider a d-dimensional American style 'max' call option, where $\rho_{ij} = \rho$ for $i \neq j$, $\sigma_i = \sigma$ for all i's, and the payoff function is given by $g(\mathbf{S}, t) = \max\{\max_{i=1,\ldots,d}\{S_i\} - K, 0\}$. Multi-dimensional max options are common in practical applications.

In the following Experiments 3, 4 and 5, we evaluate the American style max call option. We perform Experiment 3 such that the runtime between our method and the DRL method matches. This is done to show that our method performs more efficiently with other payoff structures. In this experiment, we also compare our method to the unsupervised method and investigate the performance of our method far ITM and far OTM, i.e., $s_0 = 80, 120$. Experiment 4 and 5 are run until the accuracy of our proposed method matches the accuracy of the DRL method. In Experiment 5, we also look at the timing results of the Longstaff-Schwartz method that matches the accuracy of the DRL method. For $d = 2$, we computed price of the American style max call option as described in [25], we set $\{T = 1, r = 0.05, \sigma = 0.2, \rho = 0.3\}$. For $d = 5, 30$ we computed the price of the American max call option as described in [54]. We set $\{T = 3, r = 0.05, \sigma = 0.2, \rho = 0\}$.

Experiment 3: 2-dimensional American style max call option.

The purpose of this example is to show that our method can be used to get the price and delta of different payoff functions. More specifically, we compare our method with

the DRL method, using the the finite difference method solution as a ground truth. We approximate the exact prices and deltas by the Crank-Nicolson finite difference method with 1000 timesteps and 2049×2049 space grid points as in [34]. The DRL method for this experiment was run with $J = 8$, with sample size 200000, for $N = 10$ timesteps. Our experiments compare the runtime of our proposed method and the DRL method given the same runtime. For $s_0 = 90, 100, 110$, we do not compare our method to the Longstaff-Schwartz method. This comparison has been performed in [34]. We present additional results for $s_0 = 80, 120$, where we make the comparison to the Longstaff-Schwartz method with a polynomial of degree 4.

Comparison with DRL Method

Using our proposed method, the percent errors of the computed prices at $t = 0$ are less than 2.062% (Table 2.7), the percent errors of the computed deltas at $t = 0$ are less than 2.3665% (Table 2.7). For the same runtime, our method produces smaller errors than the DRL method, which had errors between $4.2139\% - 27.6173\%$ for price and $4.3196\% - 66.1151\%$ for delta.

Table 2.7 2-dimensional Max call: computed option prices and deltas at $t = 0$.

(a) Finite Difference method

s_0	Price	Std	% Error	Delta	% Error	Run Time (s)
90	4.1941	–	–	[20.2762, 20.2353]	–	–
100	9.6309	–	–	[32.8168, 32.8905]	–	–
110	17.3235	–	–	[42.4883, 42.4207]	–	–

(b) Proposed method

s_0	Price	Std	% Error	Delta	% Error	Run Time (s)
90	4.2806	0.00162	2.062421	[20.7641, 20.7061]	2.3665	150.3
100	9.7198	0.00177	0.923071	[33.3599, 33.3998]	1.6016	156.39
110	17.4483	0.00168	0.72041	[43.3701, 43.1945]	1.9499	150.61

(c) DRL method

s_0	Price	Std	% Error	Delta	% Error	Run Time (s)
90	3.0358	0.3946	27.61737	[27.4312, 27.3825]	35.3041	150.4
100	8.0745	0.3856	16.16048	[39.4164, 39.3535]	66.1151	156.43
110	16.5994	0.3761	4.2139	[40.5587, 40.6825]	4.3197	150.85

American style max call option price and delta Deep OTM and ITM

Next we looked at testing the performance of our proposed method in the case where the max call option was far ITM, $s_0 = 80$, and far OTM, $s_0 = 120$. We compared this with the DRL method and the Longstaff-Schwartz method and summarize the results in Table 2.8. Using our proposed method, the percent errors of the computed prices at $t = 0$ are less than 2.7671% (Table 2.7), the percent errors of the computed deltas at $t = 0$ are less than 3.8461% (Table 2.8). For the same runtime, our method produces smaller errors than the DRL method, which had errors 13.5132% and 43.3926% for prices and 44.32096% and 6.3112% for deltas. However, this is not the case for the Longstaff-Schwartz method. We see that for small dimensions, the price approximated by the Longstaff-Schwartz method has errors of 1.6020% and 0.6170%. This outperforms both our proposed method and the DRL method under time constraint. However, our proposed method still outperforms the Longstaff-Schwartz method when it comes to computing deltas. The Longstaff-Schwartz method had relative errors of 8.6042% and 3.1955%.

Table 2.8 2-dimensional Max call: Deep OTM and ITM computed option prices and deltas at $t = 0$.

(a) Finite Difference method

s_0	Price	Std	% Error	Delta	% Error	Run Time (s)
80	1.3046	–	–	[8.9656, 8.9176]	–	–
120	26.5793	–	–	[47.9348, 47.7876]	–	–

(b) Proposed method

s_0	Price	Std	% Error	Delta	% Error	Run Time (s)
80	1.3407	0.00163	2.7671	[9.3101, 9,2609]	3.8461	151.82
120	26.826	0.00167	0.9281	[49.3198, 49.1329]	2.8523	153.67

(c) DRL method

s_0	Price	Std	% Error	Delta	% Error	Run Time (s)
80	0.7385	0.3891	43.3926	[12.9289, 12.8604]	44.2096	151.78
120	22.9876	0.3742	13.5132	[50.8435, 50.9202]	6.3112	153.56

(d) Lonstaff-Schwartz

s_0	Price	Std	% Error	Delta	% Error	Run Time (s)
80	1.2837	0.01369	1.6020	[9.5721, 9.8498]	8.60416	151.14
120	26.4153	0.01334	0.6170	[49.0682, 49.713]	3.19549	153.13

For pricing under limited runtime, Longstaff-Schwartz method performs better than both neural network approaches when $d = 2$, however, we see that the delta performance is not as good as our proposed method.

Price comparison with unsupervised method

We also compare the output of the proposed method to the unsupervised method using the L-BFGS optimizer and the loss function outlined in [116]. We implemented the unsupervised method as in [116], where an $101 \times 101 \times 75$ spacetime grid was used to run the unsupervised method. As in the other comparisons, we will limit the total run time to be the same as those given in Table 2.7. Then, for the initial prices of $s_0 = 90, 100, 110$, the resulting values output by the unsupervised method are $V_{unsupervised} = 25.72, 25.00, 24.59$, respectively.

Pricing errors are calculated based on FDM prices in Table 2.7 for each s_0. The relative errors of the unsupervised method for each s_0 are $[513.481\%, 167.160\%, 48.527\%]$, respectively. The inaccurate prices of the unsupervised method can be attributed to the prescribed limited training time. Note that if the method is allowed to run until convergence, the method is accurate and produces good results. In other words, given the same amount of time, our method is able to produce more accurate results than the unsupervised method. We also note that the unsupervised method is not designed to compute deltas so no delta comparison has been made.

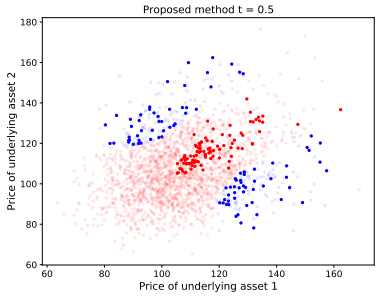
Experiment 4: 5-dimensional American style max call option.

We note that unlike Experiment 2, exact solutions are not available. Therefore, in this section, we run our proposed method until it reaches results close to those produced by the DRL method. We do not compare with the Longstaff-Schwartz method as this has been done in [34]. The DRL method for this experiment was run with $J = 2$, with sample size 720000, for $N = 100$ timesteps.

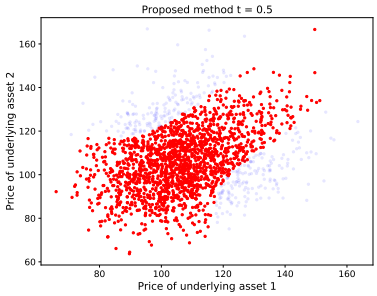
Table 2.9 reports the runtime, option prices and deltas at $t = 0$ computed by the proposed method and the DRL method. We observe in this experiment that, for our method, prices that have an absolute difference of order 10^{-2} to the price results from the DRL method following the setup in [34]. For $s_0 = 90, 100, 110$ the DRL method requires 10026.46 seconds, 10592.07 seconds and 10171.40 respectively. Our method requires 939.86 seconds, 953.41 seconds and 944.85 seconds respectively. Our method shows about a 10 times faster run time to achieve similar results.

Experiment 5: 30-dimensional American style max call option

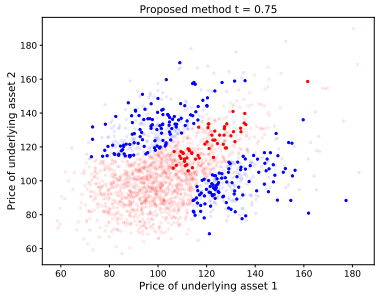
In this experiment, we extend Experiment 4 to 30 dimensions, with a simulation size of 270000. We report the runtime, the mean option price, the option price standard deviation,



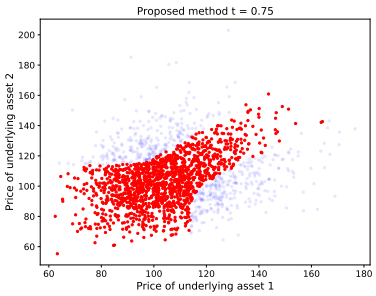
(a) Proposed method: $t = 0.5$



(b) DRL method: $t = 0.5$



(c) Proposed method: $t = 0.75$



(d) DRL method: $t = 0.75$

Figure 2.7 2-dimensional max call options. Comparison of exercised sample points between the proposed method ((a) and (c)) and the DRL method ((b) and (d)) computed at times $t = 0.5$ ((a) and (b)) and $t = 0.74$ ((c) and (d)) and initial price $s_0 = 110$. Exercised points(blue) and continued points (red); points misclassified as continued(bold dark blue) and points misclassified as exercised (bold dark red).

Table 2.9 5-dimensional max call option: computed prices and deltas at $t = 0$

(a) Proposed method

s_0	Price	Std	Delta	Run Time (s)
90	16.9152	0.000851	[18.196,18.060,18.142,18.173,18.206]	939.86
100	26.5210	0.000999	[19.100,19.117,19.087,19.511,19.072]	953.41
110	37.1389	0.000813	[20.475,20.646,20.678,20.443,20.601]	944.85

(b) DRL method

s_0	Price	Std	Delta	Run Time (s)
90	16.8896	0.000709	[17.280, 17.320, 17.540, 17.380, 17.470]	10026.46
100	26.4876	0.000728	[20.170, 20.040, 19.980, 20.710, 20.410]	10592.07
110	37.0996	0.000721	[21.570, 21.980, 21.900, 21.490, 22.020]	10171.40

the mean delta and the delta standard deviation that the methods took to achieve prices within the order of 10^{-2} absolute difference from the DRL method. In this example, we also make comparison with the Longstaff-Schwartz method. In addition, we also report standard deviation for price and delta, and report mean price and delta. The experimental results are summarized in Table 2.10.

Table 2.10 reports the runtime, option prices and deltas at $t = 0$ computed by the proposed method, the DRL method and the Longstaff-Schwartz method. This experiment shows that, for our method, prices have an absolute difference of order 10^{-2} to the price computed from the DRL method. For $s_0 = 90, 100, 110$ the DRL method requires 15763.37 seconds, 15783.14 seconds and 15743.56 seconds respectively. Our method required 1384.35 seconds, 1388.06 seconds and 1389.65 seconds, respectively. The Longstaff-Schwartz method required 47034.76 seconds, 47094.21 seconds and 47099.81 seconds, respectively. Our method shows about a 10 times faster run time to achieve similar accuracy over the DRL method and over 35 times faster than the Longstaff-Schwartz method. We can also see that the delta of the option computed for the Longstaff-Schwartz method has a larger variation over the deltas computed using the DRL method and the Proposed method.

Experiment 6: delta hedging of American style max call option

In this section we compare the delta hedging results calculated using the price and delta computed by the proposed method and the DRL methods. We also present the portfolio mean and standard deviations for dimension $d = 2$ at time $t = 0.5, 0.75$ with hedging

Table 2.10 30-dimensional max call option: computed prices and deltas at $t = 0$

(a) Proposed method

s_0	Price	Std	Delta	Std	Run Time (s)
90	41.6933	0.00175	4.9568	0.0230	1384.35
100	57.382	0.00195	4.9354	0.021	1388.06
110	72.3044	0.00146	4.9315	0.0232	1389.65

(b) DRL method

s_0	Price	Std	Delta	Std	Run Time (s)
90	41.7125	0.00152	5.0139	0.0117	15763.37
100	57.4205	0.00148	5.0121	0.0108	15775.14
110	72.3504	0.00186	5.0133	0.0137	15743.56

(c) Longstaff-Schwartz

s_0	Price	Std	Delta	Std	Run Time (s)
90	41.6844	0.00259	4.8842	0.03386	47034.76
100	57.3705	0.00283	4.8221	0.03459	47094.21
110	72.5003	0.00309	4.9585	0.03614	47099.81

intervals $\{100, 500\}$. In addition, we compute the relative P&Ls of the delta hedging portfolio using the finite difference method and compare them with the values computed by our approach. Table 2.11 shows the F1 score of the two methods. The F1 score of our method ranges from $0.8011 - 0.9564$. The F1 score of the DRL method ranges from $0.6667 - 0.7965$.

Table 2.11 F1-score calculated using (2.28) for 2-dimensional American style max call option under time constraint.

	Proposed method		DRL method	
s_0	$t = 0.5$	$t = 0.75$	$t = 0.5$	$t = 0.75$
90	0.9564	0.9405	0.7187	0.7163
100	0.8983	0.8780	0.7965	0.7471
110	0.8362	0.8011	0.6667	0.6667

Table 2.12 and Figure 2.8 show the means, standard deviations and distributions of the relative P&Ls computed by the proposed approach versus by finite difference methods. The variation computed by our proposed approach decreases as more timesteps are used

Table 2.12 2-dimensional Max call option: Means and standard deviations of the relative P&Ls by finite difference versus the proposed method, subject to 100 hedging intervals

(a) 100 hedging intervals

	Proposed method		Finite difference method	
s_0	mean	std dev	mean	std dev
90	-0.0018	0.0899	0.022	0.1932
100	-0.0022	0.0434	0.0016	0.0990
110	-0.0023	0.0072	0.0016	0.0614

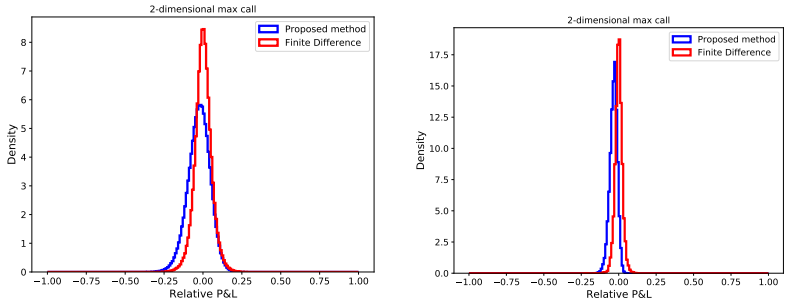
(b) 500 hedging intervals

	Proposed method		Finite difference method	
s_0	mean	std dev	mean	std dev
90	-0.0018	0.0011	0.022	0.1932
100	-0.0028	0.0017	0.0016	0.0990
110	-0.0029	0.0021	0.0016	0.0614

in the training of the model, however, we can observe that the proposed method produces a skewed normal P&L distribution. The right skewness of the resulting P&L distribution shows that the replicating portfolio of our methodology is biased toward overestimating the risk of the portfolio.

2.8 Conclusion

In this chapter, we propose a deep Recurrent network framework for pricing and hedging high-dimensional American option. Our proposed method achieves a better runtime efficiency than the DRL method. We reformulate the BSDE as a least-squares problem that is solved using a neural network. We present a loss function (2.12), which allows us to train our deep RNN framework in all spacetime for the continuation price and delta with theoretical convergence results as shown in [86]. The theoretical time complexity of our proposed method is linear in the number of timesteps N which is better than the quadratic results of [34]. Interestingly, we found that as N decreases there is a constant increase of the quadratic complexity term.



(a) 100 hedging intervals (b) 500 hedging intervals

Figure 2.8 2-dimensional max call option. Comparison of the densities of the relative P&Ls computed by the proposed approach (blue) versus the relative P&Ls computed by the finite difference method (red) at $s_0 = 1.1K$.

Chapter 3

Solving High-Dimensional HJB Equations Arising from Multi Agent Optimal Trade Execution using Residual Networks and Self Attention

3.1 Introduction

As introduced in Section 1.3, the optimal trade execution model was developed as a framework that allows traders to control the rate of how much quantity of an asset they wanted to sell/buy in their portfolio. Models that focus on limit-order transactions view the problem locally and cannot capture the risk aversion of investors. It was found that incorporating risk aversion allowed the trading to reflect real investors more accurately and allowed for the construction of an efficient frontier [7]. By trading optimally a large investment agent can limit the fluctuation of prices due to large movements in quantity. But a challenge in the original model is that the investor that is being modelled is assumed independent and does not interact with other investors [7]. We use the term agent and investor interchangeably in this chapter.

Another problem is that the framework for optimal trade execution under the mean variance criteria is time inconsistent due to the variance term which is non-monotonic with respect to wealth [14]. If the optimal control given at time t is time-inconsistent, then this means the optimal control at time t is no longer optimal at a later time. The problem with time inconsistency is that absence of time consistency also prevents us from using

dynamic programming directly. Conversely, a time-consistent solution will result in an optimal control computed at time t and some other time t' to be the same at a future time s. A common approach to overcome time-inconsistency is to solve for the optimal control at time t and the portfolio is not changed until the portfolio matures at T. This strategy is referred to as the *pre-commitment* strategy.

The pre-commitment solution to the mean-variance problem often reformulates the original mean-variance problem to an equivalent formulation [95, 55, 19]. Pre-commitment strategy works well within stable economic regimes and is shown to outperform time-consistent strategies [56, 133]. However, the pre-commitment assumes that there is complete information from t to T but in reality the state of the economy might change drastically within a given time period due to exogenous factors.

There has been increasing development in continuous time consistent mean variance asset allocation in both reinsurance and investments [69]. Most of the current work solves the continuous time optimal portfolio allocation problem [140, 4, 3, 137], with one paper looking at the control of impulse controls in jump processes [131]. The analytical solution to the continuous time consistent mean variance asset allocation problem has been studied for an agent with d-assets [140] and in the 1 asset reinsurance and investment setting using K-agents [69]. The optimal asset allocation problem can be viewed as a special case of the optimal trade execution problem when there is no price impact, i.e., $\kappa_t = 0$ and $\kappa_P = 0$ for agents who can only sell/buy the asset. The numerical solutions to the optimal trade execution problem has been presented for the time-inconsistent strategy using finite difference methods (FDM) [55, 137]. The FDM solution to the time-consistent mean variance asset allocation problem, i.e., control on α, and its extension to impulse control has also been solved [131]. A mean-field game approach is explored in [139], however this framework limits the problem to one large representative institutional investor and multiple high frequency traders. This framework also only looks at one asset being traded by all traders.

Recent advances in reinforcement learning (RL) uses model free methods to solve the optimal trade execution on limit-order-books and real data [103, 96, 33]. However, the optimal trade execution on the trading book dynamics alone does not capture the preference of traders [6]. A RL framework that solves the continuous time mean-variance portfolio optimization problem derives the optimal policy iteration [136]. A RL method has been used to approximate the asset allocation under mean variance criteria in the discrete time setting [76]. However, the issue with using the discrete time model in the optimal trade execution is that the problem is solved locally and does not guarantee the solution given satisfies the HJB equation [14].

In this Thesis, we do not make comparisons to the solver from [86] and the deep BSDE approach of [49]. This is because both of the methods cannot solve the optimal trade execution problem efficiently. These are approaches that use ANNs thus that need to be trained at every timestep. For the optimal trade execution problem this approach a single ANN is not enough to reduce the dimensionality of the problem. This is because we have multiple assets, multiple control variables and time. Thus we cannot use these methods for comparison. Our solution is an extension on the BSDE solver proposed by [86]. We use a self-attention layer coupled with a Residual U-net to train a single network to solve the optimal trade execution problem. The self-attention layer captures the temporal dependencies between the inputs. The residual U-net captures the relationship between the controls and other input features. Our method allows us to improve efficiency by training one network rather than one for each timestep.

Another difficulty of this problem is that none of the existing numerical solutions have addressed the problem of solving the time-consistent mean variance optimal trade execution problem in continuous time for multiple assets and agents. The analytical solutions are restricted to the control on the quantity, α, and cannot be extended to the control of the trade rate, v, because the trade rate is a non-linear control. This motivates the need for numerical solutions to the continuous time-consistent mean variance optimal trade execution problems in higher dimensions. In Chapter 2, we discussed how machine learning can be used to solve the high-dimensions HJB equation more efficiently. This method can be used to solve the time-consistent optimal trade execution problem to multiple agents and assets presented in this chapter.

In this chapter, we extend the optimal trade execution formulation to the time-consistent framework and the use of neural networks to solve the high-dimensional optimal trade execution problem for d-assets and K-agents. In this chapter:

- We present a time-consistent mean-variance optimal trade execution framework for multiple assets and multiple agents. This framework allows us to extend the current framework and study a more complex game between agents. We reformulate the mean variance problem into its dual representation and an auxiliary HJB equation. To extend this framework into higher dimensions we reformulate the HJB equation as a backward stochastic differential equation (BSDE). The auxiliary HJB equation has been shown to have a viscosity solution [4].

- We present a neural network solution for the d-dimensional K-agent and mean-field optimal trade execution problem using neural networks, which extends the numerical solution of the time consistent optimal trade execution problem to high dimensions.

We use techniques based on neural network pricing of high dimensional options to formulate a deep residual self-attention network that overcomes the need to save weights at each timestep [49, 34, 71, 86].

- We show numerical experiments on the interactions of K-agent optimal trade execution under different market conditions in a finite horizon setting.

This chapter is organized as follows. In Section 3.2, we describe the optimal trade execution problem and extend it to the d-asset K-agent setting. We explore agents that may be concerned about their own performance relative to other agents. In Section 3.3, we formulate the HJB PDE representation of the d-asset K-agent optimal trade execution problem, its equivalent BSDE formulation and its approximate discretization. In Section 3.4, we discretize the BSDE to the least squares problem and introduce the neural network models used to solve them. In Section 3.5, we present numerical results and conclude in Section 3.6.

3.2 Mean-Variance Optimal Trade Execution

In this section, we present the optimal trade execution problem with $i = 1, ..., d$ assets and K agents. This is an extension of the classical framework [55]. Suppose each agent k, where $k = 1, ..., K$, can observe a basket of d stocks in the market, with price process $\mathbf{S}_k(t) = [S_{1,k}(t), ..., S_{d,k}(t)]^\top \in \mathbb{R}^d$. Then the number of shares of the d stocks owned by agent k is given by $\boldsymbol{\alpha}_k(t) = [\alpha_{1,k}(t), ..., \alpha_{d,k}(t)]^\top$ and the quantity invested in a risk free bank account is given by $b_k \in \mathbb{R}$. Let $t \in [0, T]$ be the time up to maturity T, then at any time t, the market agent k has a wealth $X_k(t)$ given by:

$$X_k(t) = b_k(t) + \sum_{i=1}^{d} \alpha_{i,k}(t) S_{i,k}(t). \tag{3.1}$$

To handle buying and selling cases symmetrically, we start off with $b_0 \in \mathbb{R}$ in cash, $\alpha_{i,k,0} > 0$ shares if we are selling and $\alpha_{i,k,0} < 0$ shares if buying. This means that we are trying to liquidate a long (short) position if we are selling (buying). More precisely

$$b_k(0) = b_0, \quad S_{i,k}(0) = s_{i,0}, \quad \alpha_{i,k}(0) = \alpha_{i,k,0}, \tag{3.2}$$

$$b_k(T) = b_T, \quad \alpha_{i,k}(T) = 0, \tag{3.3}$$

where $b_T \in \mathbb{R}$ is the cash at terminal time T and $b_T - b_0$ is the cash generated by selling/buying in $[0, T)$. The objective is to maximize the wealth at maturity, T, and simultaneously minimize the risk as measured by the variance. Note that due to the non-monotonicity of the variance term, this problem as presented is time inconsistent [14, 130].

A natural optimization criteria to consider for optimal trade execution is the mean-variance (MV) strategy of Markowitz. The MV strategy looks for the optimal strategy that maximizes an investor's expected gain, and at the same time minimizes their risk [137].

In this section we define the mean-variance criteria for optimal trade execution. Let $\mathbb{E}[X_k(T)|v_k(t)]$ be the conditional expected wealth given the trade rate $v_k(t)$. We denote $\mathbb{E}_v[X_k(T)] = \mathbb{E}[X_k(T)|v_k(t)]$. The agent k with trade rate $v_k(t)$ maximizes the expected wealth and minimizes the variance defined as

$$Var_v[X_k(T)] = \mathbb{E}_v[X_k(T)^2] - (\mathbb{E}_v[X_k(T)])^2.$$

This can be reduced to a single objective function given by

$$J(X_k(T)) = \mathbb{E}_v[X_k(T)] - \frac{\gamma_k}{2} Var_v(X_k(T)), \tag{3.4}$$

where $\gamma_k > 0$ is the risk preference of the investor k. Then the mean-variance optimal execution problem is to determine the strategy $v(t)$ such that:

$$\arg\max_v J(X_k(T)) = \arg\max_v \left\{ \mathbb{E}_v[X_k(T)] - \frac{\gamma_k}{2} Var_v(X_k(T)) \right\}. \tag{3.5}$$

Let $v_k^*(t)$ be the optimal control that maximizes (3.5), then the optimal objective function is given by

$$J^*(X_k(T)) = \mathbb{E}_{v^*}[X_k(T)] - \frac{\gamma_k}{2} Var_{v^*}(X_k(T)).$$

In general, the solution for the mean-variance problem is time inconsistent. Many approaches have been explored to overcome this limitation by solving an auxiliary problem which is referred to as the time consistent mean-variance problem [140, 55, 130, 3, 131, 56, 69].

3.2.1 Problem formulation

The dynamics of the optimal trade execution problem can be formulated by the Almgren-Chriss model for the case of one asset and one agent [7]. For the one asset, one agent case,

the instantaneous trade rate is given by:

$$v(t) = \frac{d\alpha(t)}{dt},$$

where $\alpha(t)$ is the quantity of stock held by the investor. Let μ be the drift, r be the risk free rate and σ be the volatility of the asset. The price process of the risky asset follows the Geometric Brownian motion (GBM) [55]:

$$dS(t) = (\mu - r + g(v(t)))S(t)dt + \sigma S(t)dW(t),$$

where $g(v(t))$ is the level of permanent price impact. We use the following linear form for the permanent price impact [66, 55, 7]:

$$g(v(t)) = \kappa_p v(t),$$

where the constant κ_p is a constant permanent price impact factor. A linear permanent price impact is chosen to be consistent with the formulation of [55]. It has been shown that a linear permanent price impact is required to prevent roundtrip arbitrage [130].

The stock price we receive from trading is given by [55]:

$$\bar{S}(t, v(t)) = S(t)f(v(t)),$$

where $f(v(t))$ is the level of temporary price impact. The temporary price impact is used in our model to capture liquidity effects on the trade [6]. The function $f(v(t))$ is assumed to have the form

$$f(v(t)) = [1 + \kappa_s \text{sign}(v(t))]\exp\{\kappa_\tau v(t)^\beta\},$$

where κ_s is the bid-ask spread for the asset, κ_τ is a constant that represents the temporary price impact factor of the investor and β is the price impact exponent [55].

The cash position is then given by [55]:

$$\frac{db(t)}{dt} = rb(t) - v(t)S(t)f(v(t)).$$

Given states $\{S(T^-), b(T^-), \alpha(T^-)\}$ at the instant $t = T^-$ before the end of the trading horizon, we have one final liquidation (if $\alpha(T^-) \neq 0$) to ensure that the number of shares $\alpha(T) = 0$. The liquidation value is computed as follows [130]:

$$\begin{aligned} b(T) &= b(T^-) + \lim_{v \to -\infty} \{\alpha(T^-)\bar{S}(T^-, v(T^-))\} \\ &= \int_{t=0}^{T^-} -e^{r(T-t)} v(t)\bar{S}(t, v(t))dt + \lim_{v \to -\infty} \{\alpha(T^-)\bar{S}(T^-, v(T^-)\}. \end{aligned}$$

3.2.2 Extension to d assets and K agents

In this section, we describe how we extend the dynamics of the optimal trade execution problem to d-assets and K-agents. First, we consider the d-assets and one agent case. Let the trade rate for the agent with d-assets be given by $\mathbf{v}(t) = [v_1(t), ..., v_d(t)]^\top$. As with the one dimensional case, we assume the process $S_i(t)$ follows the GBM, with a modification due to the permanent price impact. Let μ_i and σ_i be the drift and volatility of stocks $i = 1, ..., d$. Let $\rho \in \mathbb{R}^{d\times d}$ be the correlation matrix and we define the correlated random variable $dW_i(t) = \sum_{j=1}^{d} L_{i,j}\psi_j\sqrt{dt}$, where $\psi_j \sim N(0, 1)$ are i.i.d, and L is the Cholesky factor of ρ, i.e., $\rho = LL^\top$.

Given a the initial prices, $s_i(0) \in \mathbb{R}$, for each asset i, we have:

$$dS_i(t) = (\mu_i - r + g_i(v_i(t)))S_i(t)dt + \sigma_i S_i(t)dW_i(t), \tag{3.6}$$

with initial condition $S_i(0) = s_i(0)$.

For simplicity, we assume that the market has sufficient liquidity and we do not consider market frictions. A multi-asset optimal trade execution problem has been modelled by a Orstein-Uhlenbeck (OU) process [17]. In our model, we assume a constant μ_i, r and σ_i and we introduce a permanent price impact function $g_i(v_i(t))$ in (3.6). We use the following linear form for the permanent price impact [55]:

$$g_i(v_i(t)) = \kappa_{p,i}v_i(t),$$

where the constant $\kappa_{p,i}$ is the permanent price impact factor for asset i. Given a vector of stock prices $\mathbf{S} = [S_1, ..., S_d]^\top$, the bank account b is assumed to follow

$$\frac{db(t)}{dt} = rb(t) - \sum_{i=1}^{d} v_i(t)S_i(t)f_i(v_i(t)), \tag{3.7}$$

where $f_i(v_i(t))$ is the temporary price impact of the investor as they sell/buy the asset i. The function $f_i(v_i(t))$ is assumed to have the form

$$f_i(v_i(t)) = [1 + \kappa_{s,i}\text{sign}(v_i(t))]\exp\{\kappa_{\tau,i}v_i(t)^\beta\},$$

where $\kappa_{s,i}$ is the bid-ask spread for each asset, $\kappa_{\tau,i}$ is the temporary price impact factor of the investor for each asset and β is the price impact exponent [55].

Next, we extend the optimal trade execution problem to K-agents. Let $k = 1, ..., K$, then the trading rate for each agent k is given by $\mathbf{v}_k = [v_{1,k}, ..., v_{d,k}]^\top$. Let $S_i(t)$ be the price of asset i that follows the GBM. For multiple agents the dynamics of asset i is given by:

$$dS_i(t) = (\mu_i - r + \sum_{k=1}^{K} g_{i,k}(v_{i,k}(t)))S_i(t)dt + \sigma_i S_i(t)dW_i(t),$$

where $g_{i,k}(v_{i,k}) = \kappa_{p,i,k} v_{i,k}$. Our formulation incorporates the trade actions of each agent in the asset price model through the price impact that each agent k has on the asset i . The actions of agents do not affect each other directly but their interactions occur through asset dynamics. Any agents not explicitly modelled or exogenous factors that affects the asset price is captured by the diffusion term.

Our state space is modelled through asset prices, which tells us that the asset dynamics has full information about each agent. The asset price is driven by the trade activity of each agent. This is consistent with the observation that in a multiple agent setting the total market impact on an asset i is the sum of permanent price impact of all agents and the temporary price impact [119]. In the mean-field game framework the coupling is done through the average trade rate as shown in [139].

The cash dynamic for each agent k is given by

$$\frac{db_k(t)}{dt} = rb_k(t) - \sum_{i=1}^{d} v_{i,k}(t)S_i(t)f_{i,k}(v_{i,k}(t)),$$

where

$$f_{i,k}(v_{i,k}(t)) = [1 + \kappa_{s,i}\text{sign}(v_{i,k}(t))]\exp\{\kappa_{\tau,i,k} v_{i,k}(t)^\beta\}.$$

The bid-ask spread $\kappa_{s,i}$ is only dependent on the asset i. The temporary price impact factors of each agent k only affects the execution price of agent k for each asset i and gives us a measure of how each agent perceives the cost of making a trade on the market as well as transaction costs. Note that in our framework we do not consider the case of different time horizons, this is because it introduces additional costs such as illiquidity of an asset over a certain period of time. This introduces complexities that are currently not captured by our framework.

3.2.3 Mean variance criteria of K agents

We also consider an investor k that is aware about their performance relative to other investors. This allows each agent to track their performance against the average of K

market agents. In our framework each agent can only view the average position of all agents through the average of their wealth in dollar amounts. This represents a money-manager who wants to outperform the average among a select funds to attract investment. We refer to these agents as performance conscious agents of degree ϕ. More formally, let the average dollar wealth of agents be given by [69]:

$$\bar{X}(t) = \frac{1}{K}\sum_{l=1}^{K} X_l(t).$$

Then for $\phi \in [0,1)$ we let $\hat{X}_k(t) = X_k(t) - \phi\bar{X}(t)$ in dollars. This measures the the wealth of individual investor k compared to the wealth of the average of all $k = 1, ..., K$ investors. If $\hat{X}_k(t) > 0$ then agent k is outperforming the average and $\hat{X}_k(t) < 0$ means agent k is underperforming the average. The mean-variance criteria becomes [69]:

$$J(\hat{X}_k(T); v, \phi) = \mathbb{E}_v[\hat{X}_k(T)] - \frac{\gamma_k}{2} Var_v[\hat{X}_k(T)], \tag{3.8}$$

where $\gamma_k > 0$ is the risk aversion parameter of each agent k. Note that by setting $\phi = 0$ in (3.8), the cost function reduces to (3.4). The solution to the K-agent problem results in an optimal control [69]. The Nash equilibrium for the multi-agent problem under time-consistent MV is given by the following definition [69]:

Definition 3.2.1. Optimal control of time consistent mean-variance problem for K agents [69]. Let $v_k \in \mathcal{V}_k$ be the admissible control $\forall k = 1, ..., K$. A vector $(v_1^*, v_2^*, ..., v_K^*)$ is said to be a time consistent optimal strategy, for any fixed initial state $(t, x, y) \in [0, T] \times \mathbb{R} \times \mathbb{R}$. For a fixed number $h > 0$, the new vector $(v_1^h, v_2^h, ..., v_K^h)$ is defined by:

$$v^h(\tau, x, y) = \begin{cases} v_k(\tau, x, y) & \text{for } (\tau, x, y) \in [t, t+h) \times \mathbb{R} \times \mathbb{R} \\ v_k^*(\tau, x, y) & \text{for } (\tau, x, y) \in [t+h, T] \times \mathbb{R} \times \mathbb{R} \end{cases}$$

satisfies:

$$\liminf_{h \to 0} \frac{J_k(t, x, y; v_1^*, v_2^*, ...v_k^*, ..., v_K^*) - J_k(t, x, y; v_1^*, v_2^*, ...v_k^h, ..., v_K^*)}{h} \geq 0,$$

for $k = 1, 2, \ldots, K$. In addition, the optimal value function is defined as:

$$u_k(t, x, y) = J_k(t, x, y; v_1^*, v_2^*, ..., v_K^*).$$

Remark 1. The time consistent solution satisfies the conditions of a subgame perfect Nash equilibrium [131]. We remark that in this paper, we use the term *optimal* control instead of *equilibrium* control in order to be consistent with other numerical approaches in the literature [131, 137, 19] .

3.3 Backward Stochastic Differential Equation (BSDE) Representation of the HJB Equation

In this section, we present the optimal trade execution problem using an auxiliary HJB formulation and its BSDE. The HJB formulation has been shown to have a viscosity solution [4]. We also show that when there is no price impact, the formulation reduces to the optimal investment problem. We reformulate the HJB to the BSDE in order to solve the optimal trade execution problem as a stochastic optimization problem which is more suitable for applications of neural networks.

3.3.1 HJB Formulation

We derive the general HJB equation for each performance-conscious agent k. We first look at the HJB equation for an agent k with a single asset. To simplify the exposition, we drop the dependence on t when it is clear. We define the value function such that:

$$u_k(S, b_k, \alpha_k, t; \phi) = \sup_{v_k}\{J_k(X_k, T; v_k, \phi)\}.$$

The value function u can be solved by reformulating the problem as an auxiliary problem [137]. However, the auxiliary problems do not permit a viscosity solution. To overcome this, we can rewrite the variance term such that [4]:

$$Var(X_k) = \mathbb{E}[X_k]^2 - \sup_{\psi\in\mathbb{R}}\{-\psi^2 - 2\psi\mathbb{E}[X_k]\}.$$

Then the original mean-variance criteria becomes:

$$u_k(S, b_k, \alpha_k, t) = \sup_{\psi\in\mathbb{R}}\{U_k(S, b_k, \alpha_k, t) - \frac{\gamma}{2}\psi^2\}, \tag{3.9}$$

where $U_k(S, b_k, \alpha_k, t)$ is the value function of the Markovian control problem given by [4]:

$$U_k(S, b_k, \alpha_k, t) = \sup_{v}\{(1-\gamma\psi)X_k(T) - \frac{\gamma}{2}(X_k(T))^2\}. \tag{3.10}$$

Note that $\psi^* = -\mathbb{E}^{v_k^*}[X_k(T)]$ solves (3.9). Let $\mathcal{L}U_k = (\mu - r)S\frac{\partial U}{\partial S} + \frac{(\sigma S)^2}{2}\frac{\partial^2 U_k}{\partial S^2}$. Then the HJB equation that solves (3.10) is given by:

$$\frac{\partial U_k}{\partial t} + \mathcal{L}U_k + r\frac{\partial U_k}{\partial b_k} + \sup_{v_k\in\mathcal{V}}\left\{g(v_k)S\frac{\partial U_k}{\partial S} + v_k\frac{\partial U_k}{\partial \alpha_k} - v_k S f(v_k)\frac{\partial U_k}{\partial b_k}\right\} = 0, \tag{3.11}$$

with terminal condition

$$U_k(T) = (1 - \gamma\psi)X_k(T) - \frac{\gamma}{2}(X_k(T))^2.$$

We extend this formulation to a performance conscious agent k [69] by replacing $X_k(T)$ with $\hat{X}_k(T) = X_k(T) - \phi\bar{X}(T)$. We formally extend (3.11) to K agents and d assets in the following Lemma:

Lemma 3.3.1. *Let the generator $\mathcal{L}U_k$ be defined as:*

$$\mathcal{L}U_k = \sum_{i=1}^{d}(\mu_i - r)S_i\frac{\partial U}{\partial S_i} + \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\rho_{i,j}\sigma_i\sigma_j S_i S_j\frac{\partial^2 U}{\partial S_i^2}.$$

For each k agent with assets $i = 1, \ldots, d$, the extended HJB equation is given by:

$$\frac{\partial U_k}{\partial t} + \mathcal{L}_k U_k + rb_k\frac{\partial U_k}{\partial b_k} + \sup_{v\in\mathcal{V}}\left\{\sum_{i=1}^{d} g(\sum_{k=1}^{K} v_{i,k})S_i\frac{\partial U_k}{\partial S_i} + \sum_{i=1}^{d} v_{i,k}\frac{\partial U_k}{\partial \alpha_{i,k}} - v_k\sum_{i=1}^{d}(S_i f(v_{i,k}))\frac{\partial U_k}{\partial b_k}\right\} = 0, \tag{3.12}$$

with terminal condition

$$U_k(S_1, ..., S_d, b_k, \alpha_{1,k}, ..., \alpha_{d,k}, T) = (1 - \gamma\psi)X_k(T) - \frac{\gamma}{2}X_k(T)^2.$$

Proof. Let $(S_1, ..., S_d, b_k, \alpha_{1,k}, ..., \alpha_{d,k}, t)$ be denoted as $(S_i, b_k, \alpha_{i,k}, t)_{i=1,...,d}$. Let Δt be the timestep size such that $\Delta t \to 0$, then $\Delta S_i = S_i(t + \Delta t) - S_i(t), \Delta b_k = b_k(t + \Delta t) - b_k(t)$, and $\Delta\alpha_{i,k} = \alpha_{i,k}(t + \Delta t) - \alpha_{i,k}(t)$. For $i = 1, ..., d$ and for each k the value function U_k of the auxiliary problem is given by:

$$U_k = \sup_{v\in\mathcal{V}}\mathbb{E}_v[U_k(S_i, b_k, \alpha_{i,k}, T)_{i=1,...,d}].$$

By the optimality of the control and law of iterated expectation as shown in [130], we can rewrite U_k as:

$$U_k = \sup_{v\in\mathcal{V}}\mathbb{E}_v[U_k(S_i + \Delta S_i, b_k + \Delta b_k, \alpha_{i,k} + \Delta\alpha_{i,k}, t + \Delta t)_{i=1,...,d}]. \tag{3.13}$$

Let $\Delta U_k = U_k(S_i + \Delta S_i, b_k + \Delta b_k, \alpha_{i,k} + \Delta\alpha_{i,k}, t + \Delta t)_{i=1,...,d} - U_k(S_i, b_k, \alpha_{i,k}, t)_{i=1,...,d}$, then (3.13) can be written as

$$\sup_{v_k\in\mathcal{V}}\mathbb{E}_v[\Delta U_k] = 0. \tag{3.14}$$

By Ito's Lemma:

$$\mathbb{E}_v[\Delta U_k] = \mathbb{E}_v\left[\frac{\partial U_k}{\partial t}\Delta t + \sum_{i=1}^{d}\frac{\partial U_k}{\partial S_i}\Delta S_i + \frac{1}{2}\sum_{i=1}^{d}\frac{\partial^2 U_k}{\partial S_i^2}(\Delta S_i)^2 + \frac{\partial U_k}{\partial b_k}\Delta b_k + \sum_{i=1}^{d}\frac{\partial U_k}{\partial \alpha_{i,k}}\Delta\alpha_{i,k}\right],$$

where

$$\begin{aligned}
\mathbb{E}_v[\Delta S_i] &= (\mu_i - r + \sum_{k=1}^{K} g_{i,k}(v_{i,k}))S_i\Delta t,\\
\mathbb{E}_v[\Delta b_k] &= (rb_k - \sum_{i=1}^{d} v_{i,k}S_i f_{i,k}(v_{i,k}))\Delta t,\\
\mathbb{E}_v[\Delta\alpha_{i,k}] &= v_{i,k}\Delta t.
\end{aligned}$$

The expectations of $\Delta S_i, \Delta b_k$ and $\Delta\alpha_{i,k}$ are straightforward. The expectation of $(\Delta S_i)^2$ requires more thought. Applying Ito's lemma on $(\Delta S_i)^2$, the expected value is given by:

$$\mathbb{E}_v[(\Delta S_i)^2] = \sum_{j=1}^{d}\rho_{i,j}\sigma_i\sigma_j S_i S_j\Delta t.$$

Let $\mathcal{V} := [v_{\min}, v_{\max}]$ be the space of bounded trade rates v, such that there exists an optimal control $v^*_{i,k}$. Expanding (3.14) and evaluating $\mathbb{E}_v[\Delta S_i], \mathbb{E}_v[(\Delta S_i)^2], \mathbb{E}_v[\Delta b_k], \mathbb{E}_v[\Delta\alpha_{i,k}]$ gives us:

$$\begin{aligned}
\Bigg[\frac{\partial U_k}{\partial t} + \mathcal{L}U_k + rb_k\frac{\partial U_k}{\partial b_k} + \sup_{v\in\mathcal{V}}\Bigg\{&\sum_{i=1}^{d}\sum_{k=1}^{K} g_{i,k}(v_{i,k})S_i\frac{\partial U_k}{\partial S_i}\\
&+\sum_{i=1}^{d} v_{i,k}\frac{\partial U_k}{\partial\alpha_{i,k}} - \sum_{i=1}^{d}(v_{i,k}S_i f_{i,k}(v_{i,k}))\frac{\partial U_k}{\partial b_k}\Bigg\}\Bigg]\Delta t = 0.
\end{aligned}$$

Dividing out the Δt and bringing the supremum inside the expectation gives us the extended HJB equation with terminal condition

$$U_k(S_1, ..., S_d, b_k, \alpha_{1,k}, ..., \alpha_{d,k}, T) = (1-\gamma\psi)X_k(T) - \frac{\gamma}{2}X_k(T)^2.$$

□

Remark 2. The K HJB equations of Lemma 3.3.1 are coupled through the optimal control.

Remark 3. *Special Case:* $\kappa_\tau = 0, \kappa_p = 0$

For the special case of $\kappa_\tau = 0, \kappa_p = 0$, i.e., there is no price impact, the optimal trade execution reduces to the optimal asset allocation problem. The analytical solution to the optimal asset allocation problem for time consistent mean-variance problem is known and has been solved in the linear-quadratic (LQ) sense [140, 95].

3.3.2 BSDE formulation

A stochastic approach has an advantage in that the HJB can be extended to include multiple assets and multiple agents. We cannot solve the multiple agent problem using a PDE approach since it is computationally infeasible, even with the dimensional reduction techniques [55]. This is because each PDE for variance and wealth requires a three dimensional mesh even with one asset. We drop the explicit dependence on t to keep the notation concise. To solve (3.12), we formulate the BSDE for U_k using the following lemma.

Lemma 3.3.2. *If there exists the optimal control $v^*_{i,k}$, the BSDE that satisfies viscosity solution for (3.12) is given by:*

$$dU_k = \left(\frac{\partial U_k}{\partial t} + \mathcal{L}_k U_k\right) dt + \sum_{i=1}^{d} \sigma_i \frac{\partial U_k}{\partial S_i} dW_i, \tag{3.15}$$

where

$$\begin{aligned}\frac{\partial U_k}{\partial t} + \mathcal{L}_k U_k = -r b_k \frac{\partial U_k}{\partial b_k} + \sum_{i=1}^{d} v^*_{i,k} f_{i,k}(v^*_{i,k}) \frac{\partial U_k}{\partial b_k} \\ - \sum_{i=1}^{d} (v^*_{i,k}) \frac{\partial U_k}{\partial \alpha_{i,k}} - \sum_{i=1}^{d} \sum_{k=1}^{K} (g_{i,k}(v^*_{i,k})) S_i \frac{\partial U_k}{\partial S_i}.\end{aligned}$$

Proof. The proof is from the direct application of the BSDE formulation given in [109]. Let $x_k = b_k + \sum_{i=1,\dots,d} S_i \alpha_{i,k}$, for each agent k. Given the optimal control $v^*_{i,k}$ and a semi-linear PDE given by:

$$\frac{\partial U_k}{\partial t} + \mathcal{L} U_k + \sum_{i=1}^{d} F\left(x_k, v^*_{i,k}, U_k, \frac{\partial U_k}{\partial x_k}, t\right) = 0,$$

with terminal condition $U_k(T) = \sum_{i=1}^{d} G(x_k)$. The corresponding BSDE is given by:

$$dU_k = -\sum_{i=1}^{d} F\left(x_k, v^*_{i,k}, U_k, \frac{\partial U_k}{\partial x_k}, t\right) dt + \sum_{i=1}^{d} \sigma_i \frac{\partial U_{i,k}}{\partial x_k} dW_i,$$

from the semi-linear PDE we can express $-F\left(x_k, v^*_{i,k}, U_k, \frac{\partial U_k}{\partial x_{i,k}}, t\right)$ as:

$$-\sum_{i=1}^{d} F\left(x_k, v^*_{i,k}, U_k, \frac{\partial U_k}{\partial x_k}, t\right) = \frac{\partial U_k}{\partial t} + \mathcal{L} U_k.$$

From (3.12) for each k agents:

$$-\sum_{i=1}^{d} F\left(x_k, v^*_{i,k}, U_k, \frac{\partial U_k}{\partial x_k}, t\right) = -rb_k\frac{\partial U_k}{\partial b_k} + \sum_{i=1}^{d} v^*_{i,k} f_{i,k}(v^*_{i,k})\frac{\partial U_k}{\partial b_k} - \sum_{i=1}^{d}(v^*_{i,k})\frac{\partial U_k}{\partial \alpha_{i,k}} - \sum_{i=1}^{d}(g_{i,k}(\sum_{k=1}^{K} v^*_{i,k}))S_i\frac{\partial U_k}{\partial S_i}.$$

□

The significance of the system of BSDEs is twofold. One, the system of BSDEs does not require us to compute the Hessian of the extended HJB equation. This a big advantage as it allows us to avoid computing the second derivative on U since computation of the second derivative can be very costly for high dimensions. This reduces the computational complexity of the problem. The second significance of the system of BSDEs is that it allows us to solve (3.15) in a least squares sense which is a recent technique used in high-dimensional option pricing [71, 49, 86, 34, 102]. This allows us to avoid using numerical solutions to the PDE and overcome the issue with dimensionality. We can replace X in (3.9) and (3.10) with $\hat{X}$ for the case where the agent is performance conscious, i.e., $\phi > 0$.

3.3.3 Discretization of BSDE

We use the Euler's method to simulate the forward SDE (3.6) through time. Let $m = 1, .., M$ be the indices of simulation paths and $n = 0, ..., N-1$ be the indices of the discrete time-steps from 0 to T. Let $\Delta t = T/N$ be the timestep size and let $t^n = n\Delta t$. We discretize $dW_i^{t^n}$ as $(\Delta W_i)_m^n = \sum_{j=1}^{d} L_{i,j}(Z_j)_m^n\sqrt{\Delta t}$ and (3.6) as:

$$\begin{aligned}(S_i)_m^0 &= (s_i)_m^0,\\ (S_i)_m^{n+1} &= (1 + (\mu_i - r + \sum_{k=1}^{K} g_{i,k}((v_{i,k})_m^n))\Delta t)(S_i)_m^n + \sigma_i(S_i)_m^n(\Delta W_i)_m^n,\end{aligned} \tag{3.16}$$

for $i = 1, \ldots, d$. Next we discretize the position in the bank account, $b(t)$, using (3.7):

$$(b_k)_m^{n+1} = (1 + r\Delta t)(b_k)_m^n - \sum_{i=1}^{d}(v_{i,k})_m^n(S_i)_m^n f((v_{i,k})_m^n)\Delta t. \tag{3.17}$$

The number of shares in stocks, $\alpha_{i,k}$ is the discretization of $\alpha(t)$ and its dynamics are given by:

$$(\alpha_{i,k})_m^{n+1} = (\alpha_{i,k})_m^n + (v_{i,k})_m^n \Delta t. \tag{3.18}$$

The total wealth for each agent k at each timestep can be calculated as:

$$(X_k)_m^n = (b_k)_m^n + \sum_{i=1}^{d} (\alpha_{i,k})_m^n (S_i)_m^n.$$

When $\phi \neq 0$, we need to account for the cash account at each timestep n by:

$$(\hat{X}_k)_m^n = (X_k)_m^n - \phi(\bar{X}_k)_m^n.$$

Let $\hat{\mathcal{V}}$ be the discretization of $\mathcal{V}$ into $C \in \mathbb{Z}^+$ discrete points. Equations (3.16), (3.17), and (3.18) are simulated forward in time over a subset of discretized control values $\hat{\mathcal{V}} \subset \mathcal{V}$. The discrete set of control points does not need to be too dense and we restrict it to the discrete set shown in [55].

To discretize the system of BSDEs, we first let $\{D_t, D_S, D_b, D_\alpha\}$ be the approximations of the partial derivatives with respect to $\{t, S, b, \alpha\}$, respectively, defined as follows:

$$\begin{aligned}
D_t U &= \frac{U^{n+1} - U^n}{t^{n+1} - t^n}, \\
D_S U &= \frac{U^n(S + \Delta S) - U^n(S)}{\Delta S}, \\
D_b U &= \frac{U^n(b + \Delta b) - U^n(b)}{\Delta b}, \\
D_\alpha U &= \frac{U^n(\alpha + \Delta \alpha) - U^n(\alpha)}{\Delta \alpha}
\end{aligned}$$

Then we can discretize (3.15) as:

$$\begin{aligned}
(U_k)_m^{n+1} - (U_k)_m^n &= \left(D_t (U_k)_m^n + (\mathcal{L}_k U_k)_m^n\right) \Delta t + \sum_{i=1}^{d} \sigma_i D_{S_i} (U_k)_m^n \Delta (W)_m^n \\
&= (\Delta U_k)_m^n,
\end{aligned}$$

where

$$D_t(U_k)_m^n + (\mathcal{L}_k U_k)_m^n = -r((b_k)_m^n - \phi\bar{b}_m^n)D_b(U_k)_m^n + \sum_{i=1}^{d}(v_{i,k})_m^n f((v_{i,k})_m^n)(S_i)_m^n D_b(U_k)_m^n - \sum_{i=1}^{d}((v_{i,k})_m^n - \phi(\bar{v}_i)_m^n)D_\alpha(U_k)_m^n - \sum_{i=1}^{d}(g(\sum_{k=1}^{K} v_{i,k})_m^n))(S_i)_m^n D_S(U_k)_m^n.$$

The recursive system of equations to solve for $(U_k)_m^n$ at time n from $n+1$ is then given by:

$$(U_k)_m^n = (U_k)_m^{n+1} - (\Delta U_k)_m^n.$$

We can approximate U at timestep n by:

$$(U_k)_m^n = (U_k)_m^{n+1} - (\Delta U_k)_m^n,$$

and enforcing time-consistency through each discrete timestep the approximation of U is given by:

$$(U_k)_m^n = \mathbb{E}_{v^{*n+1}}[(U_k)_m^{n+1}] - (\Delta U_k)_m^n. \tag{3.19}$$

Note that the BSDE (3.19) is solved backward in time.

Least squares formulation

To solve the BSDE (3.19), we approximate $(U_k)^n$ by $(\hat{U}_k)^n$ and reformulate the problem in a least squares formulation. Consider the n^{th} timestep from (3.19), the pointwise residual $(R_U)_m^n$ is given by:

$$(R_U)_m^n = (U_k)_m^{n+1} - ((\hat{U}_k)_m^n - \left(D_t(\hat{U}_k)_m^n + (\mathcal{L}\hat{U}_k)_m^n\right)\Delta t + \sum_{i=1}^{d}\sigma_i D_{S_i}(\hat{U}_k)_m^n \Delta(W)_m^n).$$

The objective becomes the minimization of the residuals such that $(R_U)_m^n \to 0$. Since our K BSDE equations are coupled by the optimal control and solved backwards in time, we satisfy the optimality condition which solves the multi-agent game as in Definition 3.2.1.

This gives us the criteria we need to form the following least squares problem:

$$L_U = \mathbb{E}[||(R_U)^n||_2^2]. \tag{3.20}$$

Minimizing (3.20) gives us a good approximation to $(U_k)^n$. The optimal control v^{*n} can then be determined by:

$$(v_{i,k}^*)^n = \arg\max_{v\in\hat{\mathcal{V}}} \mathbb{E}_v[(\hat{U}_k)^n].$$

Remark 4. *Computing gain and risk*
Given the optimal control $v_{i,k}^{*n}$ that maximizes (3.5), we can compute the expected gain and risk of the strategy by running a Monte Carlo simulation to compute the terminal wealth. The Monte Carlo follows the algorithm as shown in [130].

3.4 Neural network approach to the optimal trade execution problem

In this section, we propose a neural network approach to solve the system of least squares formulation (3.20). The method follows a general framework that has been used to solve high-dimensional American option pricing problems and has been shown to produce good results [71, 49, 86, 34, 102]. This framework has also been used to solve more general HJB equations as shown in [86].

3.4.1 Neural network formulation

We model the approximations $\hat{U}_m^n$ using a deep residual U-net with a self attention layer. U-net is a convolutional neural network architecture which has had great success in image processing and was first introduced for the application of biomedical image segmentation [113]. We chose this architecture to learn the relationship between the discretized trade rates and assets. This is because U-net has been shown to use and augment existing features more effectively [113]. We also explore the use of self-attention layer to increase the efficiency of the residual deep network by embedding temporal relationship between the previous estimate $(U_k)_m^{n+1}$ and the current states are given by $\{(S_1)_m^n, ..., (S_d)_m^n, (b_k)_m^n, (\alpha_{1,k})_m^n, ..., (\alpha_{d,k})_m^n, (v_{1,k})_m^n, ..., (v_{d,k})_m^n\}$. The use of a self-attention layer allows our network to determine which features are the most relevant in predicting the output [132]. This allows us to avoid using recurrent network structures which have a chance to exhibit vanishing gradient properties due to the activation functions used in their gate structures [132].

Self-attention residual network

The standard attention network was initially used to model predictive sequential language models [132]. They were initially implemented to learn the importance of features in predicting a language sequence. The self-attention architecture was introduced as a method to embed information inputs to create latent feature space for more efficient training [132].

In this paper, we use self-attention layer to overcome the limitation of deep residual network methods. Deep residual networks need to save weights at every timestep [71, 49, 34]; however, adding the self-attention layer allows us to capture the dependencies between features.

The attention mechanism relies on a query Q, and key-value pair $\{K, \tilde{V}\}$. We let $A(Q, K, \tilde{V})$ be the score function given by [132]:

$$A(Q, K, V) = softmax\left(\frac{QK^\top}{\sqrt{l}}\right)\tilde{V},$$

where the softmax function is a standard activation function [68] and l is the length of the sequence. Let $h = 1, ..., H$ be the number of heads where each head is given by $\mathcal{H}_h = A(Q, K, \tilde{V})$. It was found that constructing multiple attention layers in a bagging approach allowed for better performance than a single attention mechanism [132]. The multi-head attention mechanism is the concatenation of h heads and is defined as [132]:

$$\mathcal{M}_H = \mathcal{H}_1 \oplus ... \oplus \mathcal{H}_{H-1} \oplus \mathcal{H}_H, \ h = 1, ..., H,$$

where $\oplus$ is used to signify concatenation. It has been shown that using a linear layer to represent $\{Q, K, \tilde{V}\}$ is beneficial to learning [132]. The self-attention network is a special case of the multi-head attention mechanism where the inputs are passed through a linear layer to construct $\{Q, K, \tilde{V}\}$. More specifically, given an input X and weights W_Q, W_K, $W_{\tilde{V}}$ we can define $Q = W_Q X$, $K = W_K X$, $\tilde{V} = W_{\tilde{V}} X$.

The self-attention mechanism itself does not have much prediction power [132]. To add prediction power to the self-attention layer, we couple it with a convolution neural network (CNN). The benefit of using a CNN is that it captures the high dimensional structure across the discretized space of trade rates and our input features [68]. More specifically, we use a U-net architecture coupled with a residual structure. The U-net we use has 4 convolution layers and 4 transposed convolution layers with a converging-diverging geometry as shown in Figure 3.1 [113]. The convergent-divergent structure of the U-net allows for the network to capture the content in the data and allows for localization [113]. It also allows us to limit the effects of features that may not contribute to predictions.

In between each network layer, we normalize the input X. Normalization allows our network to learn more effectively and prevent cross correlation between network layers [68]. For learnable parameters φ and η to assist convergence of the deep residual network, we use group normalization, $\hat{G}$, given by [74]:

$$\hat{G}(X, \eta, \varphi) = \eta\frac{X - \mathbb{E}[X]}{\sqrt{Var(X) + \varepsilon}} + \varphi,$$

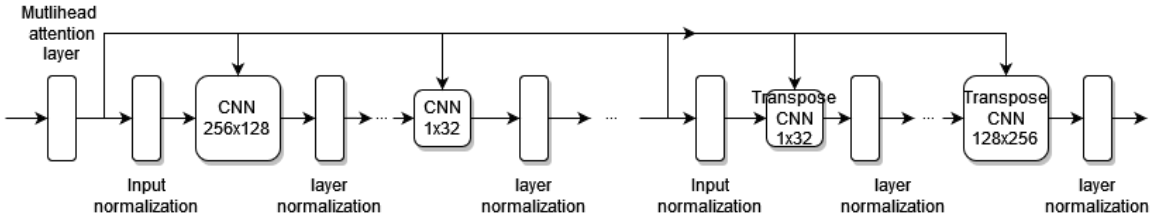


Figure 3.1 U-net with self attention architecture. It is composed of a multi-head attention layer, convolution layers and decoding transpose convolution layers with layer normalization. The input data is added to the output of each layer creating a residual structure.

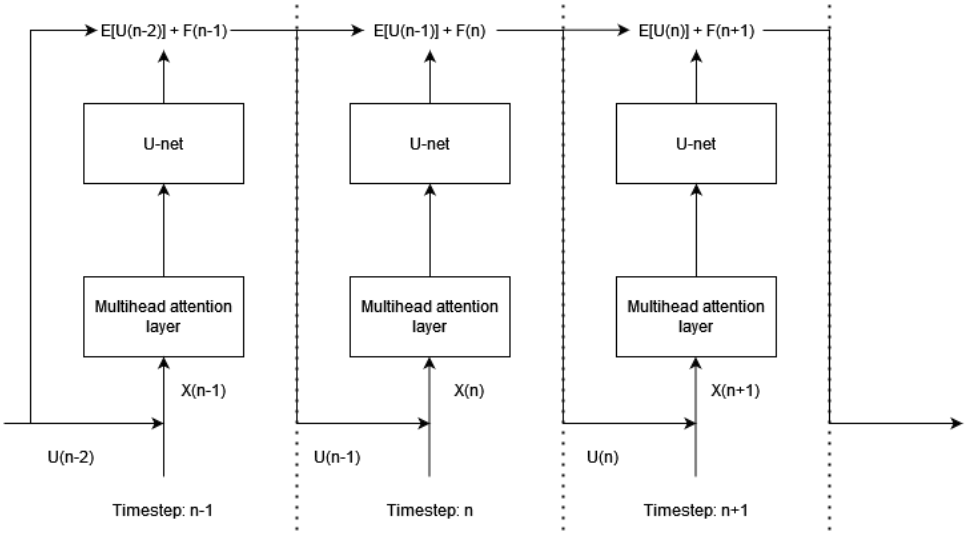


Figure 3.2 Data flow illustration of the residual U-net with self-attention rolled out through time.

where $\varepsilon \ll 1$ is some small number to aid in numerical stability.

For a given input, X^n, at timestep n, the residual U-net for each timestep has the following form [74]:

$$Res_U^n(X^n) = \text{U-net}(X^n) + W_R Res_U^{n-1}(X^{n-1}),$$

where W_R is the weight of a linear layer. The residual U-net with self-attention used in this work is shown in Figure 3.2. The network becomes a deep network as we unroll each network through time. By adding $(U_k)^{n+1}$ to the output, it gives our network a residual network structure through time.

To solve (3.19), we first approximate the function U_k using the deep residual network with self attention. Given the set of weights Θ, let $\hat{U}_k(\Theta)$ be the neural network approximation to U_k. The initial input X is used to construct the self-attention mechanism $\mathcal{M}_H$, where each head is given by $\mathcal{H}_h = A(W_Q X, W_K X, W_{\tilde{V}} X)$. The output X_a is then added to the original input and normalized. More precisely, X_a is given by:

$$X_a = \hat{G}(\mathcal{M}_H + W_a X).$$

This new input is then passed through our residual U-net. This allows us to learn spatial relationship between our features and the discretized trade rates.

In our network, we use the swish activation function over the ReLU activation function. This is because the ReLU activation function may cause information loss when training over long sequences due to the negative values being zeroed out [102]. The output of CNN plus some weighted X_a gives us the final output:

$$\mathcal{F}(\Theta) = Res_U^n(X_a;).$$

For timestep n and sample m, the neural network approximation to $(U_k)_m^n$ is given by:

$$(\hat{U}_k)_m^n(\Theta) = \mathbb{E}[(U_k)^{n+1}; (v_{i,k}^*)^{n+1}] + \mathcal{F}(\Theta)\Delta t.$$

Note that the number of time-steps N represents the depth of the deep residual self-attention network.

3.4.2 Network Training and Prediction

Let $(\hat{U}_k)_m^n(\Theta_k)$ be the deep self-attention residual network approximation of $(U_k)_m^n$ with parameters $\{\Theta_k\}$ for agent k. The parameters $\{\Theta_k\}$ are the weights and biases of the

deep self-attention residual network. The input data of $(\hat{U}_k)_m^n(\Theta_k)$ is given by $(X_{input})_m^n = \{(S_1)_m^n, ..., (S_d)_m^n, (b_k)_m^n, (\alpha_{1,k})_m^n, ..., (\alpha_{d,k})_m^n, (v_{i,k})_m^n, (U_k)_m^{n+1}, n\Delta t, m, c\}$. We concatenate the output from the previous time-step into our input. This information is used by the self-attention mechanism to embed previous timestep information into the model. The last three inputs give positional data for time, simulation path, and control index on the discretized set of control points $\hat{\mathcal{V}}$. The label data used to train the network is given by the value function from the previous timestep and is given by:

$$(Y_{label})_m^n = (U_k)_m^{n+1}.$$

This is because the BSDE is solved backward in time and we must satisfy the terminal condition (3.10). We let

$$F_m^n = \left(D_t(\hat{U}_k)_m^n + (\mathcal{L}\hat{U}_k)_m^n\right),$$

where

$$\begin{aligned} D_t(\hat{U}_k)_m^n + (\mathcal{L}\hat{U}_k)_m^n &= -r((b_k)_m^n)D_b(\hat{U}_k)_m^n \\ &+ \sum_{i=1}^{d}(v_{i,k})_m^n f_{i,k}((v_{i,k})_m^n)(S_i)_m^n D_b(\hat{U}_k)_m^n \\ &- \sum_{i=1}^{d}((v_{i,k})_m^n)D_\alpha(\hat{U}_k)_m^n - \sum_{i=1}^{d}\sum_{k=1}^{K}(g_{i,k}(v_{i,k})_m^n))(S_i)_m^n D_S(\hat{U}_k)_m^n, \end{aligned}$$

and let

$$Z_m^n = \sum_{i=1}^{d}\sigma_i D_S(\hat{U}_k)_m^n \Delta(W_i)_m^n.$$

We rewrite the losses (3.20) such that

$$(L_U)^n(\Theta_k) = \mathbb{E}[((R_U)^n(\Theta_k))^2] = \mathbb{E}[((Y_{label})^n - ((\hat{U}_k)^n(\Theta) - F^n\Delta t + Z^n))^2].$$

Minimizing $L_U(\Theta_k)$ gives us the optimal set of parameters for the self-attention network $(\hat{U}_k)^n(\Theta_k)$ given by:

$$(\Theta_k^*)^n = \arg\min(L_U)^n(\Theta_k).$$

We summarize the training of the proposed model in Algorithm 3. The gradients are computed using autodifferentiation. Data generation is dicussed in Appendix A.2.2.

To compute $(v_{i,k}^*)^n$ for agent k we run an inference of the network. We perform a linear search for the optimal expected trade rate for each asset i over the grid of $\hat{\mathcal{V}}$. We summarize the inference step of the proposed model in Algorithm 4. Optimization is performed using the Adam algorithm [92] with a learning rate of 1×10^{-4}.

Algorithm 3 Training a deep residual network with self-attention

Require: $(X_{input})_m^n, (Y_{label})_m^n, (U_k)_m^N, \mathcal{F}$
  $loss_k \leftarrow []$
  for $k = 1, ..., K$ **do**
    if $n = N$ **then**
      Compute ψ_k^* according to [4]
      $U_k^N \leftarrow (1 - \gamma_k \psi_k^*) X_k^N - \frac{\gamma_2}{2}(X_k^N)^2$
    $loss_n \leftarrow []$
    for $n = N - 1, ..., 0$ **do**
      $(\hat{U}_k)_m^n(\Theta_k) \leftarrow \mathbb{E}[(U_k)_m^{n+1}] + \mathcal{F}(\Theta_k)\Delta t$
      Compute F_m^n, Z_m^n
      $loss_n \leftarrow (L_{MSE})_k^n(\Theta_k)$ ▷ accumulate losses
    $loss_k \leftarrow (loss_n)$ ▷ accumulate losses
  $\Theta^* \leftarrow \arg\min\{loss_k\}$

Algorithm 4 Inference of value function from a deep residual network with self-attention.

Require: $X_m^n, (U_k)_m^N, \mathcal{F}$
  for $k = 1, ..., K$ **do**
    if $n = N$ **then**
      Compute ψ_k^* according to [4]
      $U_k^N \leftarrow (1 - \gamma_k \psi_k^*) X_k^N - \frac{\gamma_2}{2}(X_k^N)^2$
      $(v_{i,k}^*)^N \leftarrow \arg\max_{v \in \hat{\mathcal{V}}}\{\mathbb{E}_v\{U_k^N\}\}$
    for $n = N - 1, ..., 0$ **do**
      $(\hat{U}_q)_m^n \leftarrow \mathbb{E}[(U_k)_m^{n+1} | (v_{i,k}^*)^{n+1}] + \mathcal{F}(\Theta_k^*)\Delta t$
      $(v_{i,k}^*)^n \leftarrow \arg\max_{v \in \hat{\mathcal{V}}}\{\mathbb{E}_v\{\hat{U}_k^n\}\}$

3.5 Numerical Results

In this section, we outline the numerical experiments to validate the proposed method. We also study the effects of different parameters on the proposed method. Using our method we can observe a novel scenario where the price impact of two agents differs greatly and study how each agent would sell off their assets in different economic conditions. Then we look at groups of sellers and buyers and the average performance of each group in different economic conditions.

We do not test the neural network and the loss function separately as our network is designed to account for the high dimensionality of the controls and the temporal dynamics of the problem. We use a self-attention network to map temporal dynamics to input features and we use a CNN to map the control to other input features. A feedforward neural network implementation would yield similar results, but we know that CNNs are more efficient due to weight sharing [68]. The goal of this work was to show that we can extend the current scope of existing problems by using efficient machine learning techniques.

In Experiment 1, we look at a special case when the trade impact factors $\kappa_p = 0, \kappa_\tau = 0$ to validate the proposed method. In Experiment 2 we observe the effects of different parameters on α^*. In Experiment 3, we study 2 agents, one with a large permanent price impact trading in an environment with another agent that has no impact. In this setting, we also vary $\phi = 0.0, 0.3, 0.7$ and consider when economic conditions are good ($\mu - r > 0$) and bad ($\mu - r < 0$). In Experiment 4 we study the neural network solution to the optimal execution problem with 2 agents on different strategies in different market conditions. We also look at the effects of correlation when agents are trading $d = 2$ assets. We perform additional experiments where we study the neural network solution with 10 agents and 2 assets in the Appendix A.2.3.

3.5.1 Experiment 1: validation of neural network solution against existing solutions

In this experiment, we compare the approximation of the neural network solution to the results of existing work [55, 4, 69, 140]. This experiment verifies that the outputs of our neural network solution coincide with known solutions wherever they are available. In this experiment, we fix the parameter $\gamma = 6, \phi = 0.0$ and assume no price impact, i.e., $\kappa_p = 0, \kappa_\tau = 0$ and $\kappa_s = 0$.

Single agent, single asset

In the case of a single agent with 1 asset we compare the results to the FDM solution [55, 3]. First, we look at the performance of the proposed method for different N. This is done to determine the appropriate number of timesteps we need for an accurate solution to the problem. We fix $M = 3000$, $T = 1$ year and run the neural network until $L_U(\Theta) < 5 \times 10^{-6}$ and summarize the results in Table 3.1. We also record the FDM solution for $N = 10, 100, 200, 300, 400$ with relative error given by:

$$Err_{rel} = \frac{|actual - approx.|}{|actual|} \times 100\%, \tag{3.21}$$

we assume the FDM solution is the actual solution and the NN solution is the approximation.

N	FDM	Proposed method	Relative error
10	1.0565641	1.051620	0.4679%
100	1.0565812	1.056400	0.017%
200	1.0565820	1.056500	0.0078%
300	1.0565823	1.056309	0.026%
400	1.0565824	1.056879	0.028%

Table 3.1 Residual U-net with self-attention approximation for the solution to the optimal value $u(x_0, 0)$ over different number of time-steps N. The FDM values are results from [3]. The relative errors are computed using (3.21)

We see that the proposed method has the smallest relative error to the FDM solution at $N = 200$. We found that a network depth $N > 200$ tends to overfit the data, which is shown by the decreases in accuracy. Next, we look for the sample size M that minimizes the relative error. This is done to determine the appropriate batch size we need to accurately compute our results. We fix $N = 300$ and run the neural network until convergence. The results are summarized in Table 3.2. In Table 3.2, we record the FDM solution for the number of space grids $100, 1000, 2000, 3000$. We observe that the proposed method has the smallest relative error to the FDM solution at $M = 1000$.

We also make a comparison to the FDM solution to the optimal trade execution for $\kappa_\tau = 2 \times 10^{-6}$, $T = 1/250$, $\mu = 0$ and $r = 0$ [55]. To make further comparisons we compute the efficient frontier as shown in Figure 3.3. The efficient frontier is computed as described in [130]. For an initial price of $s(0) = 100$ and $\alpha(0) = 1.0$, the expected gain from selling

M	FDM	Proposed method	Relative Error
100	1.0553888	1.053110	0.216%
1000	1.0565197	1.056694	0.016%
2000	1.0565823	1.056762	0.017%
3000	1.0566032	1.056879	0.027%

Table 3.2 Residual U-net with self-attention approximation for the solution to the mean-variance criterion $u(x_0, 0)$ over different number of simulations M. In the FDM solution M represents the grid size used for the wealth [3]. The relative error is computed using (3.21)

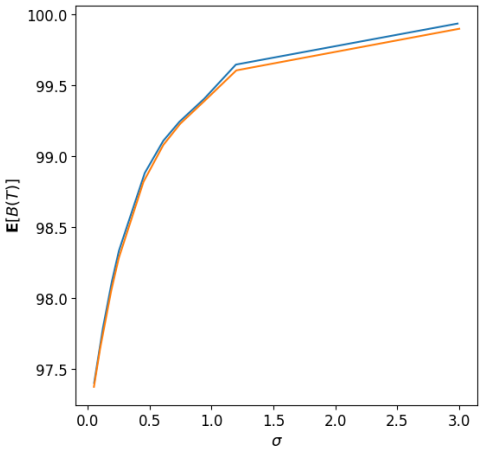


Figure 3.3 The efficient frontier interpolated from the neural network approximation (blue) and interpolated values from [55] (orange). Interpolation was performed using a polynomial of degree 4.

is 99.295 with a standard deviation of 0.7469 [55]. For our neural network approximation we get an expected gain of 99.271 with a standard deviation of 0.7402.

Single agent, 2 assets

We compare the neural network solution with analytical solutions. For the special case of no price impact, an analytical solution exists for the case of a single agent with multiple assets [140]. This experiment shows that our outputs are consistent with multidimensional results.

In the case of a single agent with multiple assets an analytical solution exists [140]. Let $\mathbf{X}(t) = [X_1(t), ..., X_d(t)]^\top$, $\mathbf{A} = [\mu_1 - r, ..., \mu_d - r]^\top, \mathbf{B} = [\sigma_1, ..., \sigma_d]^\top$ be the wealth of the investor, vector of drift rates and the covariance matrix for each assets $i = 1, ..., d$. This framework assumes independence between the assets letting us represent the volatilities as a vector. The time-consistent mean variance optimal asset allocation for multiple assets is given by [140]:

$$\text{Minimize } J(\mathbf{X}(T); \mu, \lambda) \equiv \mathbb{E}[\mu \mathbf{X}(T)^2 - \lambda \mathbf{X}(T)],$$

where parameter $\mu > 0$ and $-\infty < \lambda < \infty$. Let $\mu = 1$, $\lambda = \gamma$ and let $\alpha^*(t)$ be the quantity of assets held by the agent. Then the optimal asset allocation for a single agent with multiple assets is given by [140]:

$$\alpha^*(t) = [\mathbf{B}\mathbf{B}^\top]^{-1}\mathbf{A}^\top(\frac{\gamma}{2}e^{-r(T-t)} - \mathbf{X}(t)). \tag{3.22}$$

Our model can be used to compute $\alpha^*(t)$ by solving the equation $\alpha^*(t) = \alpha^*(t-dt)+v^*(t)dt$. We fix $N = 100$, and $M = 1000$ and compute $\mathbb{E}[J(X(T)); \alpha^*(t), t = 0, \Delta t, ..., N\Delta t]$ using Monte Carlo simulation [140]. Given $\{\alpha_1(0) = 1.0, \alpha_2(0) = 0.5\}$ the optimal value for $d = 2$ at $T = 1/250, 1/52, 1/12, 1$ years are summarized in Table 3.3. In Table 3.3 we see the highest relative error occurs at $T = 1$ with a relative error of 0.138%.

T	Analytical Solution	Proposed method	Relative Error
1/250	1.500340	1.500390	0.0033%
1/52	1.501912	1.501507	0.027%
1/12	1.506883	1.506248	0.042%
1	1.583452	1.585642	0.138%

Table 3.3 The residual U-net with self attention approximation to the solution of the optimal value $u(s_0, b_0, \alpha_0, 0)$ for different maturities T in years. The analytical solution is given by (3.22). We calculate the relative error using (3.21).

2 *agents, single asset*

We compare the neural network solution with analytical solutions. For the special case of no price impact, an analytical solution exists for multiple agent with 1 asset [69]. For the case of K-agents with a single asset we assume the agents are only trading 1 asset. The mean-variance optimal portfolio problem is given by [69]:

$$\text{Minimize } J(X(T)) \equiv \mathbb{E}[X(T)] - \frac{\gamma}{2} Var(X(T)),$$

The analytical solution to the time-consistent auxiliary problem is given by [69]:

$$\alpha_k^* = \frac{\mu_k}{\gamma_k \sigma_k^2} + \frac{\phi_k}{\sigma_k} \frac{1}{K} \sum_{k=1}^{K} \frac{\mu_k}{\gamma_k \sigma_k (1 - \phi_k)}. \tag{3.23}$$

We use Monte Carlo Simulation to compute $\mathbb{E}[J(X(T)); \alpha^*(t), t = 0, \Delta t, ..., N\Delta t]$ [69]. Given $\alpha_k(0) = 1.0$ the optimal value for $K = 2$ are summarized in Table 3.4. We present results for only one agent as the agents are assumed to be symmetric.

T	Analytical Solution	Proposed method	Relative Error
1/250	1.000221	1.000216	0.0005%
1/52	1.001059	1.001032	0.002%
1/12	1.004577	1.004224	0.0351%
1	1.0565823	1.056694	0.0122%

Table 3.4 The residual U-net with self-attention approximation to the optimal value, $u(s_0, b_0, \alpha_0, 0)$, for different maturities T in years. The analytical solution is given by (3.23). We calculate the relative error using (3.21).

In Table 3.4 we see the highest relative error occurs at $T = 1/12$ with a relative error of 0.0351%. The proposed model performs well in both multi-agent and multi-dimensional cases.

3.5.2 Experiment 2: sensitivity of parameters

In this experiment we study the effects of the relative performance factor ϕ, the risk preference γ, the permanent price impact factor κ_p and the temporary price impact factor κ_t on the optimal portfolio α^*. We vary each of these factors independently and observe

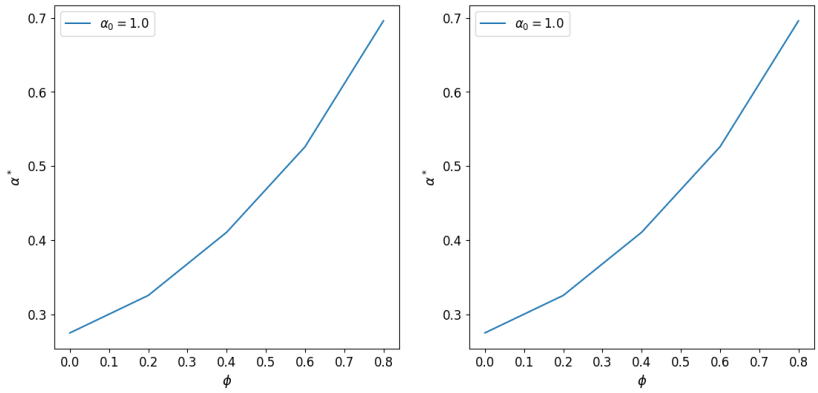


Figure 3.4 The sensitivity of of α^* with respect to $\phi = [0.0, 0.2, 0.4, 0.6, 0.8]$ at $t = 1/250$. The sensitivity of agent 1 (left) and the agent 2 (right)

the portfolio at different time-steps for 2 agents with 1 asset. For the sensitivity study we assume that agents want to liquidate long positions (sell only). We set $\alpha_0 = 1.0$, $s_0 = 1.0$, $b_0 = 0.0$, $\mu = 0.1$, $r = 0.05$, $\sigma = 0.2$ and $T = 1/250$.

To analyze the sensitivity of ϕ, we let $\gamma = 6$, $\kappa_p = 0$ and $\kappa_\tau = 0$. This is to show that the neural network approach captures the cross sectional behaviour of α^* as we increase ϕ. From existing work, we know that as ϕ increases, α^* should also increase. This is due to the agent taking riskier positions to increase their expected return [69].

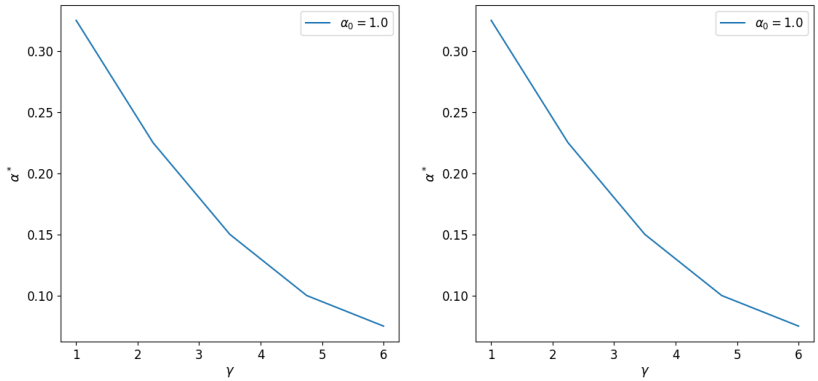


Figure 3.5 The sensitivity of of α^* with respect to $\gamma = [3.0, 3.75, 4.5, 5.25, 6.0]$ at $t = 1/250$. The sensitivity of agent 1 (left) and the agent 2 (right).

Figure 3.4 shows α^* increases as ϕ increases. This is expected as the agent invests more into risky assets as they become more aware of other agents [69]. The neural network

approach can be seen to capture this behaviour. We also want to ensure that α^* decreases as γ increases. This behaviour is an important behaviour to observe in agents as it is commonly known that as the risk aversion increases the agent will hold a smaller quantity of risky assets [140]. First we fix $\phi = 0.0$, $\kappa_p = 0.0$, and $\kappa_\tau = 0.0$. This comparison is done to show that the neural network approach captures the cross sectional behaviour of α^* as we increase γ.

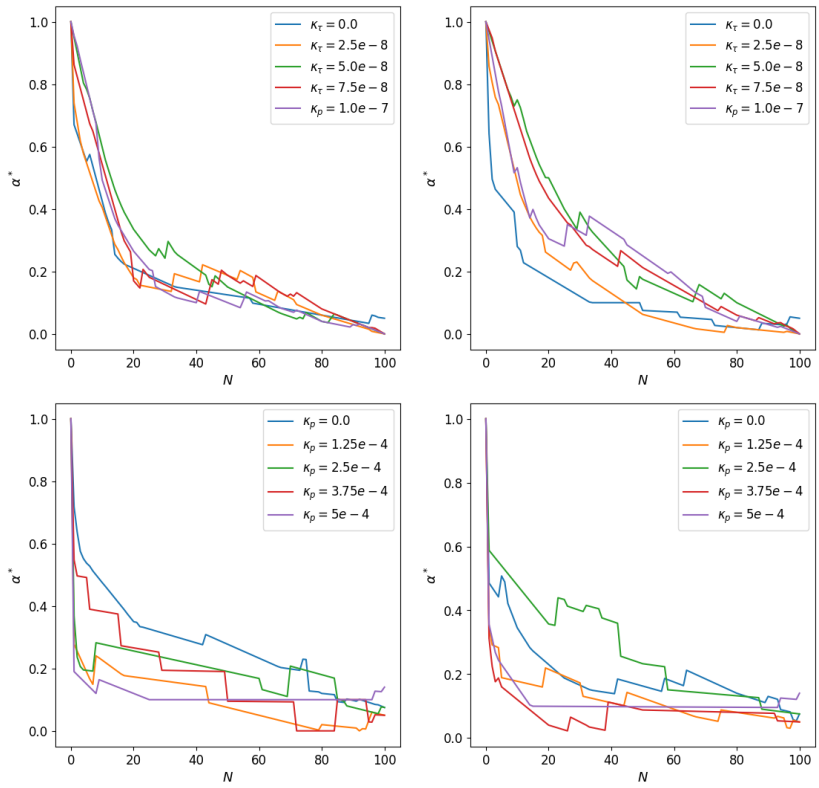


Figure 3.6 The effect of $\kappa_t = [0.0, 2.5 \times 10^{-8}, 5 \times 10^{-8}, 7.5 \times 10^{-8}, 1 \times 10^{-7}]$ (top) and $\kappa_p = [0.0, 0.000125, 0.00025, 0.000375, 0.0005]$ (bottom) on α^* for agent 1 (left) and agent 2(right). We used the average of 10 networks for this simulation.

Figure 3.5 shows α^* decreases as γ increases. This is expected as the agent invests less into risky assets as the agent becomes more risk averse (γ increasing). To observe the effects of κ_p and κ_t we fix $\phi = 0.0$ and $\gamma = 6$. We choose to fix $\phi = 0.0$ to avoid influence of other agents to the dynamics. In Figure 3.6 we plot α^* over different time-steps N for different κ_t (top) and κ_p (bottom) for both agents.

The temporary price impact, κ_τ, is the local impact each agent has on the asset price

due to the velocity of trade. To maximize the trade criteria the agents take a less aggressive approach and prefer to liquidate their positions in a more gradual manner. We observe that when the permanent price impact, κ_p, is low the agent trades more actively and the activity decreases as κ_p increases. This is expected because as the permanent price impact increases, the more impact the agent has on the asset price. Since the goal of the optimal execution problem is to trade the asset without affecting the price as much as possible the agent will be more hesitant to trade as their permanent price impact increases.

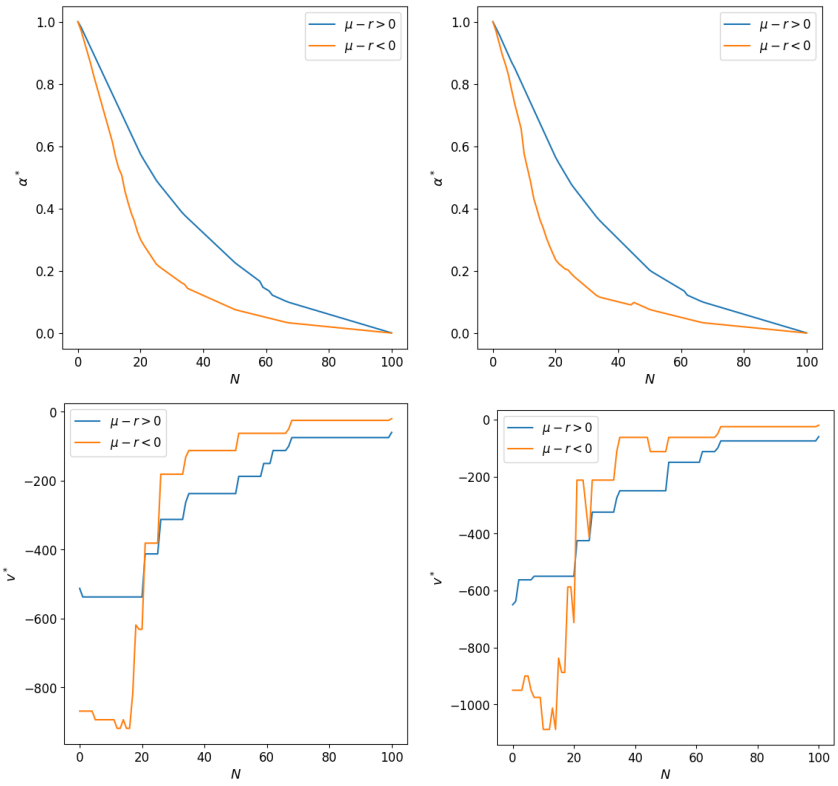


Figure 3.7 α^* for agent 1 (top left) and agent 2 (top right). v^* for agent 1 (bottom left) and agent 2 (bottom right) with $\kappa_t = 1 \times 10^{-7}$. We fix $M = 1000$ and $N = 100$. We used the average of 100 networks for this simulation.

3.5.3 Experiment 3: optimal execution in advantageous and disadvantageous market conditions for 2 agents and the effects of correlation

In this experiment, we study the effects of macroeconomic pressure on the asset. We consider $K = 2$ that sell off $d = 1$ asset and consider two cases. To simulate good market conditions we set $\mu - r > 0$. In contrast, to simulate poor market conditions we set $\mu - r < 0$. We assume there is no permanent price impact, i.e., $\kappa_p = 0.0$, $\phi = 0.2$ and $\gamma = 6$. For good market conditions we set $\mu = 0.1$ and $r = 0.05$. For poor market conditions we set the $\mu = 0.0$. Figure 3.7 shows the sell only strategy of agent 1 (left) and agent 2 (right). The portfolio is presented in the top figures and the controls are presented in the bottom figures.

In Figure 3.7 we observe that the agents liquidate their position faster if they expect the price of the asset to decrease. This is expected but how they choose to sell is important to observe. We can see the optimal trade speed for $\mu - r > 0$ is more gradual. This means the sell off is more gradual and we do not observe large jump in behaviour. This is not the case for $\mu - r < 0$ as we see fast sell off in the first 20 timesteps. Then any residual amount of the asset is sold off more gradually until T. Note that the difference in optimal paths is due to randomization of the training data during the batch training.

We can extend the experiment to $d = 2$ to study the affects of correlation with the proposed method. Since we directly consider the number of shares of an asset held instead of the proportion, we do not need to worry about adding additional constraints to the problem. In experiment 2 we only look at the selling agents and consider 4 cases to study the affects of correlation in the order execution problem. In case 1 we simulate good market conditions ($\mu_1 - r > 0$, $\mu_2 - r > 0$). In case 2 we simulate poor market conditions ($\mu_1 - r < 0$, $\mu_2 - r < 0$). In case 3 we simulate mixed market conditions ($\mu_1 - r > 0$, $\mu_2 - r < 0$). We study the affects of correlation between assets 1 and 2. We set weak positive correlation as $\rho = 0.3$ and strong positive correlation as $\rho = 0.7$. We set weak negative correlation as $\rho = -0.3$ and strong negative correlation as $\rho = -0.7$.

Case $\mathbf{1}$*:* $\boldsymbol{\mu_1 - r > 0, \mu_1 - r > 0}$

In case 1 we set $\mu_1 = \mu_2 = 0.1$ and $r = 0.05$. Figure 3.8 shows the optimal portfolio for each asset with high and low correlations. We also explore the case of positive and negative correlation. The top row shows the optimal portfolio for agent 1 (left) and agent 2 (right) with weak correlation. The bottom left and right figures show the optimal portfolio of the agents for strong correlation. Due to the non-uniqueness of the optimal trade rate, there

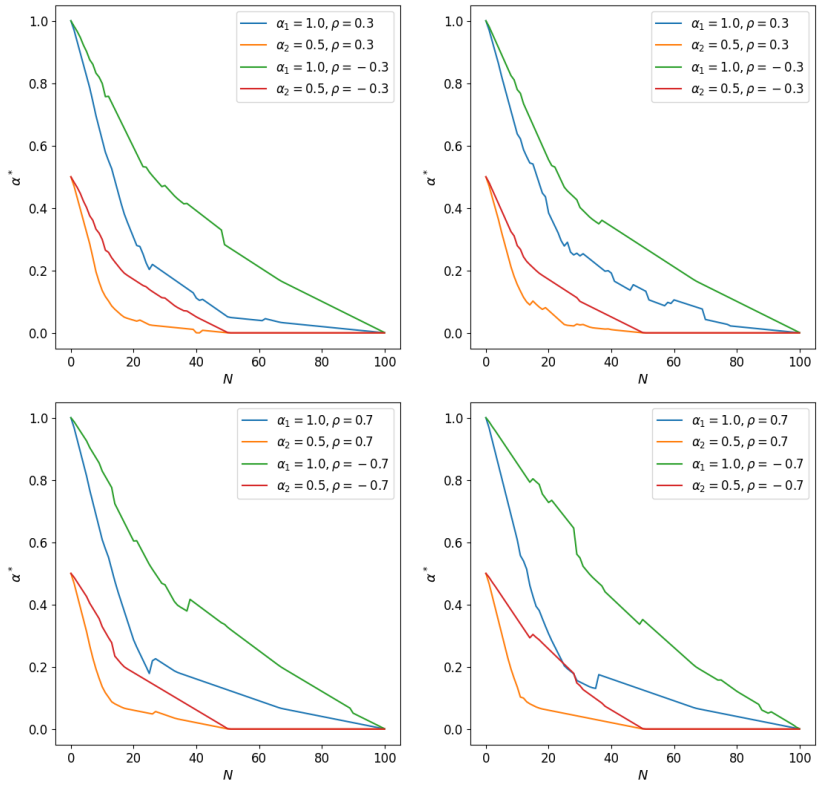


Figure 3.8 α^* for agent 1 (top left) and agent 2 (top right) with low correlation. α^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation.We used the average of 100 networks for this experiment.

is no uniqueness in the optimal portfolios.

We observe that when assets are negatively correlated the agents tend to hold more risky assets than when they are positively correlated. Positively correlated assets sell more quickly because when one asset is sold, the price impact increases the value of correlated assets, prompting the agent to sell off assets faster. Asset 1 is held longer to reduce the overall impact when the agent liquidates their position. Figure 3.9 shows the optimal control for each agent in assets with low correlation (top left and right) and high correlation (bottom left and right).

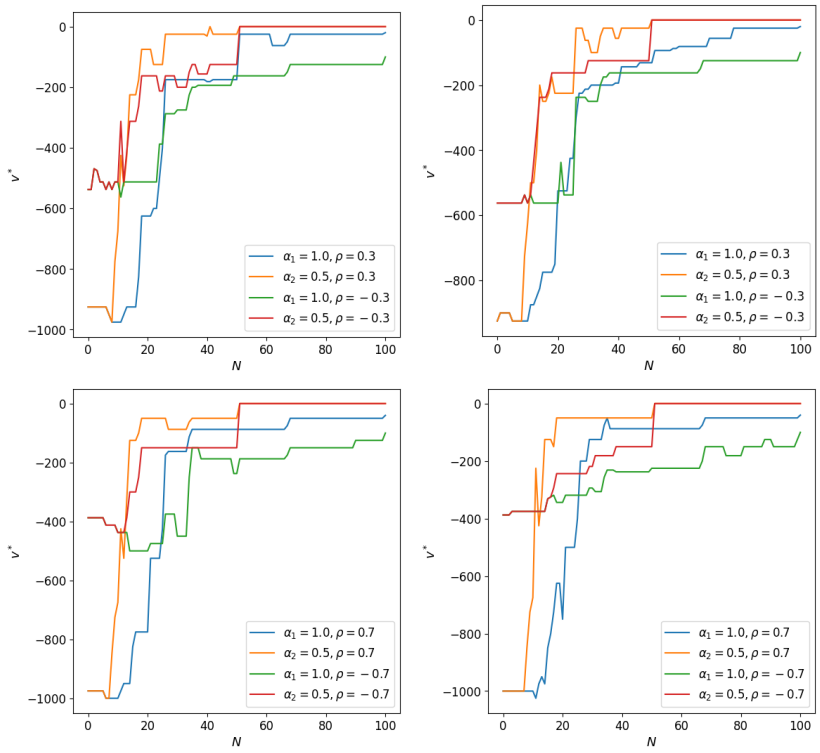


Figure 3.9 v^* for agent 1 (top left) and agent 2 (top right) with low correlation. v^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment.

In Figure 3.9 we observe that the trade rates of each agent are more negative for positively correlated assets which implies that they are selling at a faster rate. In contrast, for negatively correlated assets the trade rate is less negative resulting in slower liquidation of the risky assets. This results in a larger position in the risky asset held over time. Figure

3.9 shows the same trade rate for assets 1 and 2 at $N = 0$. This implies that the agents do not have a strong preference on holding asset 1 or 2 when they first begin to trade.

Case **2***:* $\mu_1 - \mathbf{r} < \mathbf{0}, \mu_2 - \mathbf{r} < \mathbf{0}$

In case 2, we set $\mu_1 = \mu_2 = 0.0$ and $r = 0.05$. Case 2 assumes that both assets are under poor market conditions. Figure 3.10 plots the optimal portfolio for agents 1 and 2 when the assets are weakly and strongly correlated. The corresponding optimal controls are plotted in Figure 3.11.

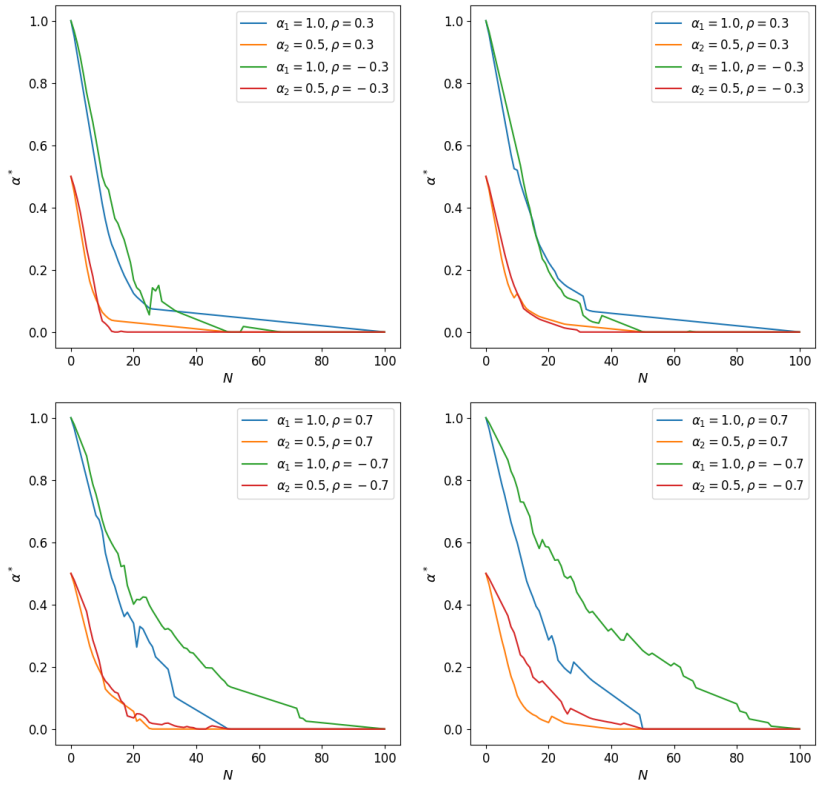


Figure 3.10 α^* for agent 1 (top left) and agent 2 (top right) with low correlation. α^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment.

In Figure 3.10, we observe that for weakly correlated assets the agents actively liquidated assets regardless of correlation. However, under strong correlation negatively correlated assets are liquidated more gradually. This is in line with what was observed in

case 1. Since the two assets are negatively correlated, the agents expect asset 1 to increase when asset 2 decreases, and vice versa. This factor reduces the negative drift experienced by each asset resulting in the agents liquidating the assets more gradually.

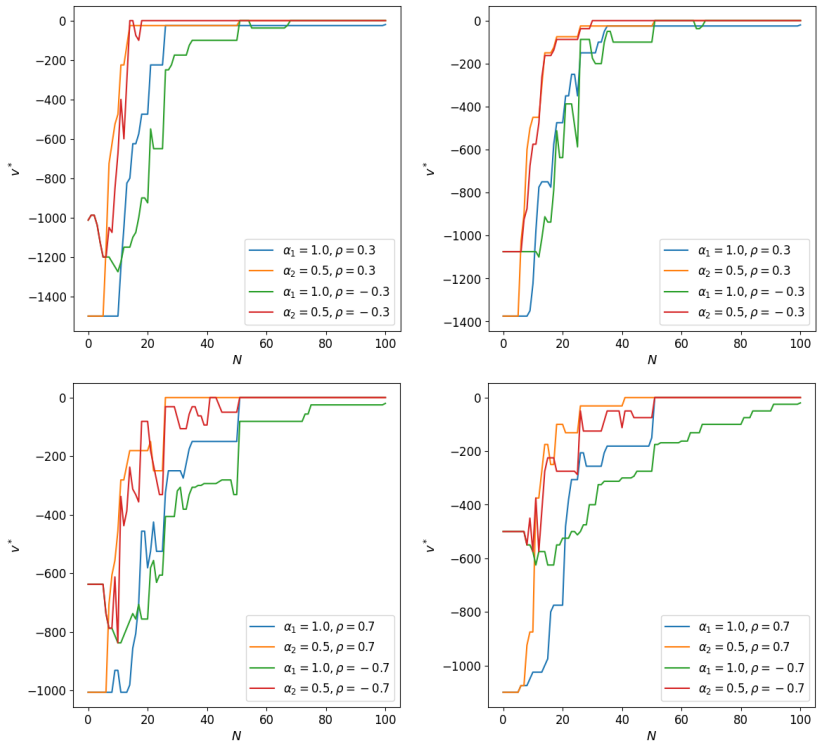


Figure 3.11 v^* for agent 1 (top left) and agent 2 (top right) with low correlation. v^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment.

In Figure 3.11 the agents liquidate positively correlated assets more quickly at $N = 0$. This reflects the initial steeper slope shown for positively correlated assets. The path of the controls is not unique in our problem, as the HJB equation only guarantees uniqueness of the value function.

Case **3***:* $\mu_1 - \mathbf{r} > \mathbf{0}, \mu_2 - \mathbf{r} < \mathbf{0}$

In case 3, we set $\mu_1 = 0.1$, $\mu_2 = 0.0$ and $r = 0.05$. Figure 3.12 plots the optimal portfolio of agent 1 and agent 2 when $\mu_1 - r > 0$ and $\mu_2 - r < 0$. The optimal control of agent 1 and agent 2 are shown in Figure 3.13.

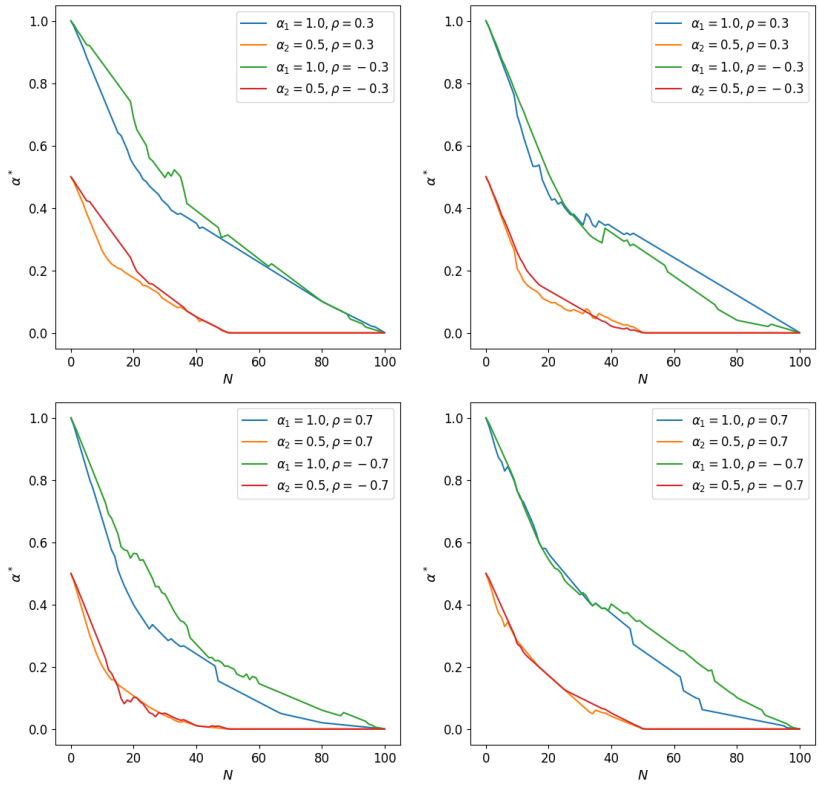


Figure 3.12 α^* for agent 1 (top left) and agent 2 (top right) with low correlation. α^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment.

In Figure 3.12 we observe that when there are mixed market conditions, the agents try to liquidate the assets in a similar manner regardless of correlation. Both negative and positive correlation encourages the agents to gradually liquidate asset 1 as asset 1 is expected to increase its price over time.In Figure 3.13 we see the agents quickly liquidating asset 2. We also observe that the agents actually do not liquidate asset 1 completely until the assets are sold off at T.

3.5.4 Experiment 4: optimal execution for 2 agents with different price impact

In this experiment, we study the interaction of two agents with different permanent price impact factors, κ_{P1}, κ_{P2}. Agent 1 has $\kappa_{P1} = 5 \times 10^{-4}$ and agent 2 has $\kappa_{P2} = 0.0$. We study the interaction between the agents for 1 asset and observe how they interact in a good economy, i.e., $\mu - r > 0$ and in a bad economy, i.e., $\mu - r < 0$. We also consider the different levels of performance awareness, $\phi = 0.0, 0.3, 0.7$. We set $r = 0.05$, $\mu = 0.1$ or $\mu = 0.0$, $s_0 = 1.0$, $\alpha = 0.0$, $\kappa_\tau = 0.0$, and $\kappa_s = 0.0$ for $T = 1/250$, $M = 1000$, $N = 100$.

First, we consider the case when $\phi = 0.0$, we see the optimal portfolio and trade rate for agent 1 and agent 2 in Figure 3.14. In Figure 3.14, we observe that when $\mu - r > 0$ agent 1 is very inactive in their trading activity and holds a large proportion of wealth in risky assets. Agent 2 on the other hand sells off their position consistently until around $N = 20$. Where they see a change in price due to the activity from agent 1.

This tells us that agent 2 perceives the market to be more risky than agent 1. This can be seen in the trade rates, where agent 2 sells off more quickly than agent 1. When $\mu - r < 0$, we observe an interesting behaviour. Agent 2 immediately sells off their position. There is no penalty for agent 2 to do this as they have no impact on the asset price. However, agent 1 cannot do the same and actually holds a large position in the risky asset longer before liquidating fully.

Next, we consider the case when $\phi = 0.3$, we see the optimal portfolio and trade rate for agent 1 and agent 2 in Figure 3.15. In Figure 3.15, we observe that both agents are aware of the general performance of the other agent they both sell quickly. When $\mu - r > 0$, both agents want to hold more positions in the risky asset until the moment they want to liquidate after $N = 100$, this is expected as the payoff from holding the asset until $N = 100$ outweighs the penalty from failing to liquidate in the time horizon when $\mu - r > 0$. We see that agent 1 trades faster than agent 2 until around $N = 20$, but the trading eases as N goes to 100. When $\mu - r < 0$, we see a similar behaviour to the good economy case for

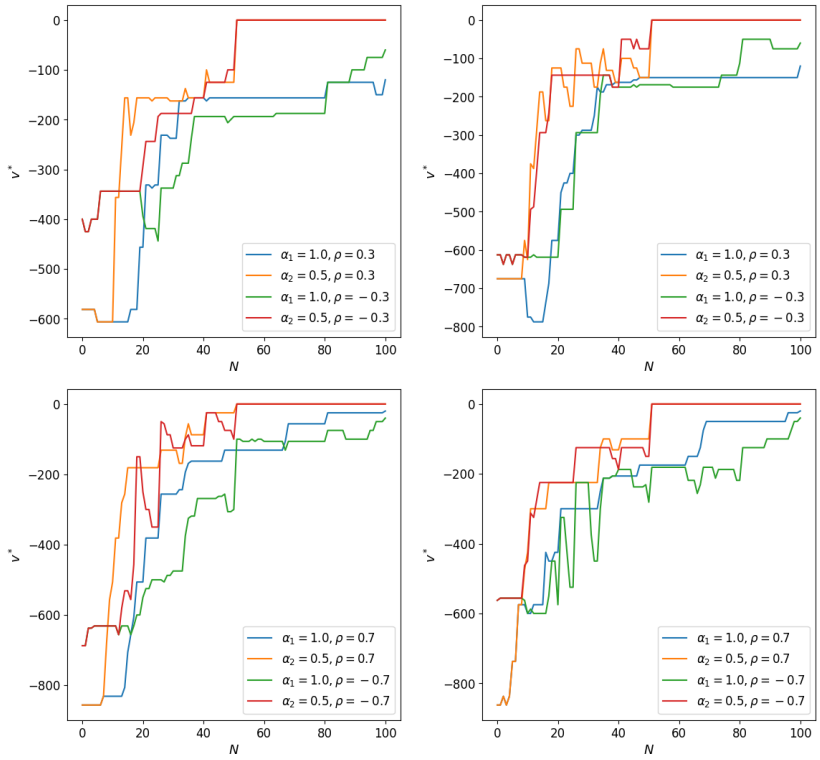


Figure 3.13 v^* for agent 1 (top left) and agent 2 (top right) with low correlation. v^* for agent 1 (bottom left) and agent 2 (bottom right) with high correlation. We used the average of 100 networks for this experiment.

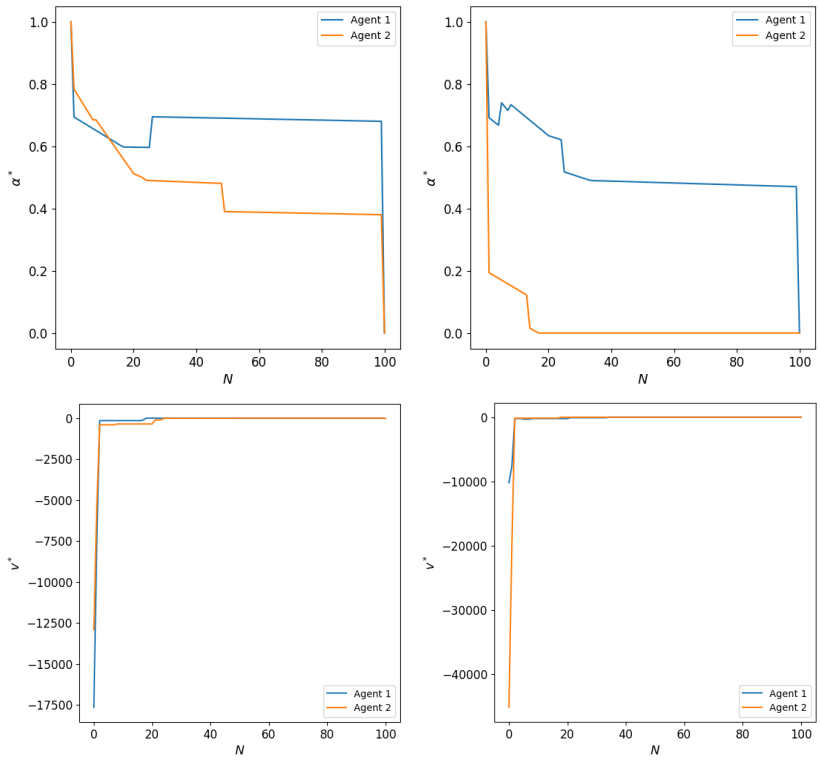


Figure 3.14 The optimal portfolio (top) and trade rate (bottom) for Agents 1 (blue) and 2 (orange). For a good economy (right) and bad economy (left) when $\phi = 0.0$.

agent 1 except that agent 1 trades faster until around $N = 10$. We see agent 2 trade more gradually until $N = 10$ as it would see the asset price drop due to the actions of agent 1.

Finally, we consider the case when $\phi = 0.7$, we see the optimal portfolio and trade rate for agent 1 and agent 2 in Figure A.1. In Figure A.1, we observe that agent 1 is willing to hold more position in the risky asset but not as quickly as when $\phi = 0.0$. This is because both agents now weigh the average performance of the market almost as much as their own individual performance. When $\mu - r > 0$, again both agents want to hold more positions in the risky asset until the moment they want to liquidate after $N = 100$. We see that agent 2 trades faster than agent 1 at all timesteps. When $\mu - r < 0$, we see agent 1 selling more actively than agent 2 which in turn lowers the asset price. In response we see agent 2 reducing their position in the risky asset drastically from the start.

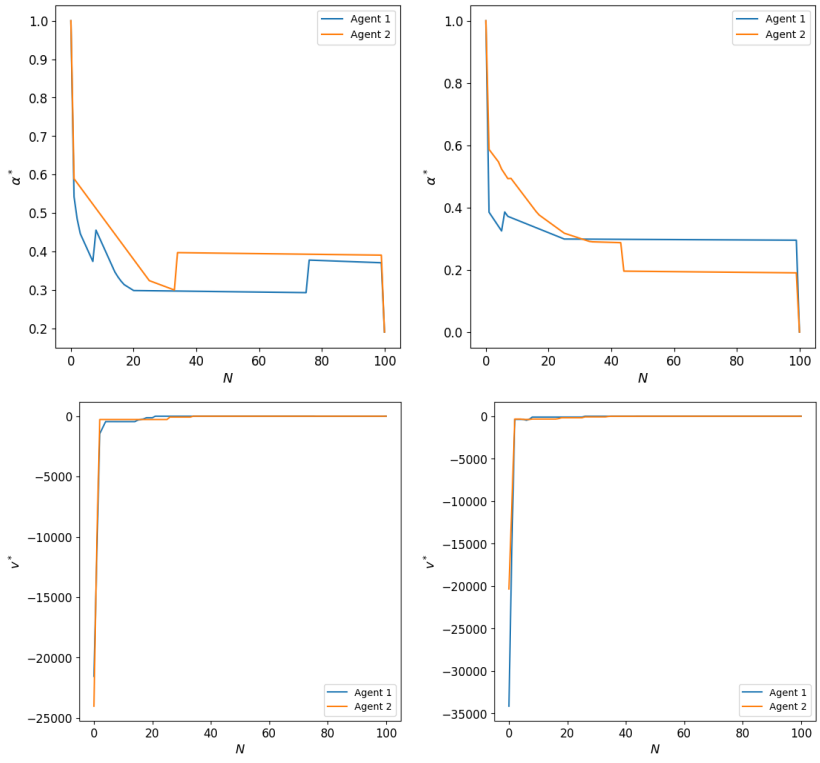


Figure 3.15 The optimal portfolio (top) and trade rate (bottom) for Agents 1 (blue) and 2 (orange). For a good economy (right) and bad economy (left) when $\phi = 0.3$.

3.6 Conclusion

In this chapter, we extended the optimal trade execution problem under the Markowitz criteria to a time-consistent optimal trade execution problem for multiple assets and multiple agents setting. This is done by formulating the original mean-variance problem into its dual representation coupled with an auxiliary HJB equation [4]. To extend this into higher dimensions, we reformulate the auxiliary HJB equation into a BSDE that can be solved using a neural network. We present a deep residual self-attention network to solve the BSDE presented in (3.19). This allows us to use a deep residual network that learns temporal dependencies. This overcomes the issue of saving weights for every timestep N and overcomes the vanishing gradient problem of recurrent networks [102].

Chapter 4

Efficiently Computing Implied Volatility Surfaces using GANs with Minimal Arbitrage Violations

4.1 Introduction

As introduced in Section 1.3, a standard inverse problem in finance is the computation of implied volatility surfaces. The Black-Scholes equation assumes that the volatility parameter σ is constant under geometric Brownian motion (GBM) with no market frictions [38, 94]. It is well documented that the observed market volatility is not constant across moneyness and maturity [94]. To account for this the implied volatility, $\sigma_{IV}(K, T)$, is solved from the Black-Scholes equation that matches the market price at each T and K. Because of this mapping, market traders often interpret implied volatility is a representation of option prices [38].

Given market option prices, the implied volatility can be computed using classical methods such as the Brent method. We assume the market price is arbitrage-free at each T and K. However, no-arbitrage can be violated when we extrapolate or interpolate from one point to other maturities and strikes. The computation of arbitrage-free implied volatility surfaces is important because an arbitrage opportunity in the options market allows trading strategies that are implemented with no cost and has positive returns. Such an opportunities create inefficiencies in the market which adds cost to market participants.

In this thesis, we use the implied volatility to directly compute the local volatility using Gatheral's formula [31]. If the implied volatility surface we compute violates arbitrage then

the resulting local volatility will also violate arbitrage. The deep local volatility method computes the local volatility directly from the option price surface using Dupire's equation [32]. The authors use soft constraints to mitigate arbitrage violations in the option price surface learned by the neural network before using Dupire's Equation to compute the local volatility [32, 2, 88].

Stochastic volatility models and local volatility models were developed by [94]. Stochastic volatility models assume the variance follows a stochastic process which is coupled with the asset price process. Stochastic volatility models are attractive to use as they can successfully model key features of volatility observed empirically such as curvature in large maturities. Stochastic volatility models have more parameters to calibrate and with limited data and the absence of market data points makes it difficult to compute stochastic volatility models that are consistent with market prices [37, 23]. Local volatility, $\sigma_{LV}(K,T)$, can be computed by solving Dupire's equation using numerical PDE methods [47]. Currently, practitioners also use parametric models such as stochastic surface volatility inspired (SSVI) models to compute local volatility. Parametric models have been devleoped to ensure arbitrage is not violated and ensure that it is consistent with the Heston model in the limits [61, 62]. However, it has been shown that the exclusion of large-moneyness behaviour still exposes Parametric models to arbitrage violations [104]. In this work, we propose to use a GAN framework to compute arbitrage free implied volatility surface which can be used to compute the local volatility.

Recently, neural networks and machine learning models have been developed to compute the Black-Scholes implied volatility in real and synthetic markets [97, 78, 111, 128, 43, 84, 98, 81]. Methods using convolutional neural networks (CNNs) has been explored in [78] and [43]. They showed that CNNs performed well; however, they needed to be redesigned for specific models. Deep artificial neural networks (ANNs) such as the rough volatility method [84], the implied volatility ANN (IV-ANN) method [98], the calibrating neural network (CaNN) method [97], and the method in [81] are used to learn the solution of the option pricing function and construct the implied volatility surface by applying an inverse method or another neural network. The methods mentioned above do not account for the possible arbitrage violation of the neural network as there is no guarantee that the no-arbitrage conditions are not violated with generic neural networks. We also note that the inversion network of CaNN is model dependent as it uses the parameters of the model it is trying to calibrate as inputs used to learn the option price.

Variational autoencoder (VAE) has been used in extrapolation and interpolation of implied volatility surfaces [18]. The VAE model was used to complete local volatility when some of the local volatility points are known. The use of regularization terms to limit arbitrage violations is explored using VAEs using a similar approach to the DNN [18].

GAN for generating implied volatility with no-arbitrage penalty has been explored [120]. However, the authors use the standard GAN loss function which results in irregular surfaces [39]. We show in our numerical experiment that this results in implied volatility surfaces that are inconsistent with the market. VolGAN, a method to generate dynamic implied volatility and covariation from financial timeseries has been proposed [39]. The authors also use the standard GAN loss with arbitrage penalties but use the log volatility to generate the surface which results in better performance. The inclusion of the mean-squared error (MSE) term in the loss function has been explored in the Fin-GAN framework [135].

In Chapters 2 and 3, we looked at efficiently solving the HJB equation arising from American option problem and we extended the optimal trade execution problem to multiple agents. These applications illustrated how HJB equation could be reformulated to a BSDE and solved efficiently using neural networks. In this chapter, we explore methods to improve efficiency by using specific ML architecture. More specifically, we use a generative adversarial network (GAN) framework to restrict the output of the a generator neural network. Our proposed method uses the discriminator as a classifier to determine whether the generated output comes from the an implied volatility surface that is arbitrage free or not. We found that this behaviour improves efficiency by reducing the need to use deep networks to model the generator network.

In this chapter, we propose using a GAN to generate implied volatility surfaces. The generator network is assisted in training by a discriminator that evaluates whether the generated implied volatility matches the targets distribution or not. We derive a GAN framework from the inverse problem that is consistent with the target volatility using an MSE loss function [135]. Our framework also trains our network to satisfy the no-arbitrage constraints by introducing penalties as regularization terms [114]. Although training a generator and a discriminator involves two networks, our proposed GAN model allows the use of shallow networks which results in much lower computational cost.

The contribution of this chapter is as follows:

- We show that the use of a generator-discriminator pair allows us to train shallow networks to achieve greater efficiency without losing accuracy.
- We propose a GAN-based framework to compute implied volatility surfaces from synthetic market option prices generated by the Heston model. Our framework allows us to use the trained generator network to generate the implied volatility with minimal arbitrage violations out-of training, which is important for pricing and hedging options.

- We also show that our method can be used to generate market consistent local volatility with minimal tuning of the generator on limited sample data by using a pre-trained discriminator. The local volatility is calculated from the implied volatility surface and is used to price exotic options.

The chapter is organized as follows. In Section 4.2, we provide a formal introduction to implied and local volatility. We also discuss the static arbitrage conditions and how they are applied to implied volatility surfaces. In Section 4.3, we show the construction of the loss function in terms of volatility for implied and local volatility and connect our loss function to the standard GAN framework. We also describe our architecture and the training of our GAN. In Section 4.4, we go over our numerical results to show the advantage of using a GAN to solve the inverse problem over deep neural networks and conclude in Section 4.4.

4.2 Implied Volatility Surfaces and Static Arbitrage

In the following section, we discuss implied volatility and local volatility. We also discuss static arbitrage and present penalty terms that impose no-arbitrage as regularization on loss functions. One of the biggest challenges in computing implied volatility surfaces comes from the lack of available data. To overcome this we follow [98, 97, 41, 18, 2] and generate synthetic data. We assume the market call option prices are generated from an underlying asset modelled by the Heston model. We denote the market call option prices generated by the Heston model as $V_{mkt}(K,T)$. Readers interested in learning more about the Heston model is directed to Appendix A.3.

4.2.1 Implied volatility

Given the initial stock price s_0, the strike price K, the time to maturity T, the risk-free interest rate r. Let $V_{mkt}(K,T)$ be the market call option price at T and K. The implied volatility, $\sigma_{IV}(K,T)$, is found by solving the inverse problem

$$V(S_0,K,T,r,\sigma_{IV}(K,T)) = V_{mkt}(K,T), \tag{4.1}$$

where the Black-Scholes call price, V, is found by solving the Black-Scholes equation [98, 63]. Note that the solution to the inverse problem, (4.1), can be reformulated as

$$V(S_0,K,T,r,\sigma_{IV}(K,T)) - V_{mkt}(K,T) = 0. \tag{4.2}$$

Computing implied volatility is then reduced to the root finding problem given by (4.2). Two common root finding methods are Newton's method and Brent's method [24]. In this chapter, we will use Brent's method for solving (4.2). In order to construct the implied volatility surface, we need to perform interpolation which may introduce arbitrage violations.

4.2.2 Local volatility

Given the stock price S, under local volatility the call option price follows the dynamics given by:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma_{LV}(S,t)^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

with terminal condition at T is $V = \max(S-K, 0)$. To change the dependent variables of the local volatility to $\sigma_{LV}(K,T)$ we can form the equivalent forward equation called the Dupire equation [47]

$$\frac{\partial V}{\partial T} + \frac{1}{2}\sigma_{LV}(K,T)^2 K^2 \frac{\partial^2 V}{\partial K^2} + rK\frac{\partial V}{\partial K} = 0 \tag{4.3}$$

with initial condition $V(K,0) = \max(S_0 - K, 0)$ [94, 63, 64]. Let $\partial_T = \frac{\partial}{\partial T}$, $\partial_K = \frac{\partial}{\partial K}$, and $\partial_{KK} = \frac{\partial^2}{\partial K^2}$. Given the option price surface $V(K,T)$, in terms of K, and T we can define the local volatility as

$$\sigma_{LV}(K,T) = \left(\frac{\partial_T V + rK\partial_K V}{0.5K^2\partial_{KK}V}\right)^{1/2}.$$

We can compute the local volatility directly from implied volatility [65]. Formally, Let $k = \log(K/s_0)$ be the moneyness and let $w = \sigma_{IV}^2(k,T)T$ be the total variance, then the local volatility can be computed by [65]

$$\sigma_{LV}(k,T) = \frac{\partial w}{\partial T} \Big/ \left(1 - \frac{k}{w}\frac{\partial w}{\partial k} + \frac{1}{4}(-\frac{1}{4} - \frac{1}{w} + \frac{k^2}{w^2})(\frac{\partial w}{\partial k})^2 + \frac{1}{2}\frac{\partial^2 w}{\partial k^2}\right) \tag{4.4}$$

4.2.3 Static arbitrage

The absence of static arbitrage in option prices ensures the fair valuation of the option for a fixed T. In this chapter, we refer to static arbitrage as arbitrage. We can say an option price surface or an implied volatility surface is free of arbitrage if and only if [62]:

1. it is free of calendar spread arbitrage (monotonicity condition);

2. it is free of butterfly spread arbitrage (convexity condition).

The absence of calendar spread implies that monotonicity constraints of the option price/implied volatility surface is satisfied over T. The absence of butterfly arbitrage ensures that there exists a non-negative probability measure [62]. This ensures that the option price/volatility is a martingale and arbitrage-free.

Generally, arbitrage violation are found by checking that the option price/implied volatility surface violates one or both of these conditions [29], which are often imposed by constraints. In this chapter, we distinguish two types of constraints. Hard constraints are constraints that must be satisfied to be a feasible solution. Alternatively, soft constraints are constraints that can be violated, but incur a penalty. This penalty term is then added as a regularization term to the objective function as shown in [88] and [2]. Note that this method does not guarantee that all arbitrage opportunities will be eliminated.

There are two methods to impose soft constraints for no-arbitrage. The first method is to impose it on the option price surface directly [88]. The second method is to impose the soft constraints through the implied volatility surface [2]. In this thesis we work directly with the implied volatility surface. The condition imposed on the price surface is presented in Appendix A.3.3.

No-arbitrage conditions imposed on implied volatility surfaces

Let the scaled implied volatility surface of k be denoted by $I(k,T) : \mathbb{R} \times [0,\infty) \mapsto [0,\infty)$. We can build on the no-arbitrage conditions on $V(k,T)$, and extend it on to $I(k,T) = \sigma_{IV}(k,T)\sqrt{T}$. The sufficient conditions for $I(k,T)$ to be arbitrage-free are [114, 2]:

- (Smoothness): $I(k,T)$ is twice differentiable for every $T > 0$;
- (Positivity): for every $k \in \mathbb{R}$ and $T > 0$, $I(k,T) > 0$;
- (Durrleman's condition): for every $k \in \mathbb{R}$ and $T > 0$,

$$0 \leq (1 - \frac{k\partial_k I}{I})^2 - \frac{1}{4}(\partial_k I)^2 + I\partial_{kk} I, \tag{4.5}$$

  where $I = I(k,T)$;

- (Monotonicity in T): for every $k \in \mathbb{R}$, $I(k,\cdot)$ is non-decreasing;

- (Large moneyness behaviour): for every $T > 0$, $\sigma_{IV}^2(k, T)$ is linear as $k \to \pm\infty$; and
- (Value at maturity): for every $k \in \mathbb{R}$, $I(k, 0) = 0$.

Note that the butterfly spread no-arbitrage conditions are satisfied from (4.5) and calendar spread no-arbitrage conditions are satisfied by the monotonicity condition.

Extending this approach, we can show the constraints to eliminate calendar and butterfly spread can be expressed as soft constraints when calibrating for the implied volatility surface $\sigma_{IV}(k, T)$ [2].

4.3 GAN Framework for Computing Implied Volatility Surfaces

In this section, we present our proposed model used to compute implied volatility surfaces. We propose a generative-adversarial network (GAN) model to compute no-arbitrage implied volatility surfaces. The GAN model is composed of two neural networks, the generator network and the discriminator network. The generator learns to generate implied volatility surfaces that have minimal arbitrage violations. The discriminator network is trained on implied volatility surfaces over different time to maturities and interest rates. The discriminator learns to classify the given data as true if the data is from the distribution of the implied volatility surface and false if it is not. Concurrently, the generator learns to generate data that is consistent with the distribution of the labels by minimizing detection from the discriminator network. This competition between the generator and discriminator allows the GAN to generate out-of-training samples that closely mimic the target in distribution [68]. Our GAN framework allows us to use smaller more efficient networks for the generator. Our proposed model can be used to compute both the local volatility and the implied volatility. We remark that our method is not dependent on the Heston model. To allow our model to train on limited number of observations, we engineer two additional features. We use an adjusted log-moneyness to enrich the feature space and the at the money (ATM) implied volatility as a form of target encoding.

4.3.1 GAN formulation of the inverse problem

We formally formulate the inverse problem that will be solved by our proposed GAN framework to compute the complete implied volatility surfaces. Given a grid of strikes and

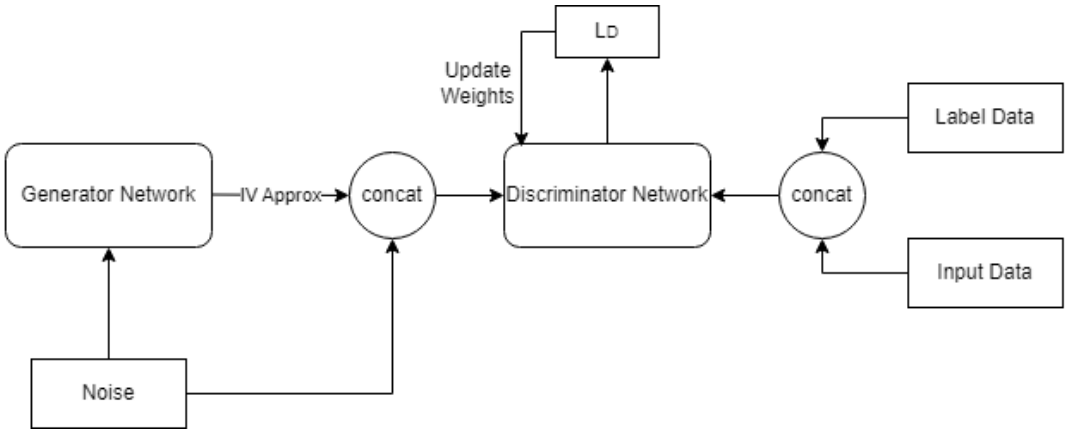


Figure 4.1 Schematic of discriminator network during training. The discriminator is trained with the Black-Scholes implied volatility (BS-IV) as the true labels and the generator output with noise as the fake labels. The generator weights are fixed while the discriminator is being trained.

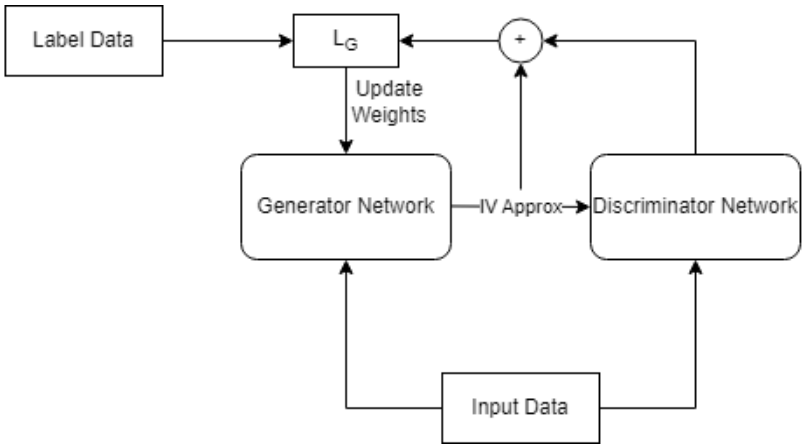


Figure 4.2 Schematic of generator network during training. The ground truth is given by the Black-Scholes implied volatility. The discriminator weights are fixed during the training of the generator network.

maturities, the objective function of the inverse problem requires us to solve

$$\arg\min_{\sigma_{IV}}\{\|V(S_0,K,T,r,\sigma_{IV}(K,T))-V_{mkt}(K,T)\|_2^2\}=\sigma_{IV}^*. \tag{4.6}$$

We formulate the inverse problem for the implied volatility directly, as it is given by definition. Let $\sigma_{mkt}(K_i,T_j)$ be the market implied volatility calculated from $V_{mkt}(K_i,T_j)$, we can reformulate the inverse problem as

$$\min_{\sigma_{IV}}\{\|\sigma_{IV}(K,T)-\sigma_{mkt}(K,T)\|_2^2\}=\sigma_{IV}^*. \tag{4.7}$$

To solve the implied volatility problem, it suffices to solve (4.7). We can approximate σ_{IV} using the generator network. However, approximating the implied volatility surface using a neural network may lead to values of σ_{IV} that are not be arbitrage-free. The main idea we propose is to use adversarial learning. We propose to use a discriminator to restrict the output of the generator such that the generator outputs a σ_{IV}^* that has a similar distribution to σ_{mkt}.

4.3.2 Loss function and proposed model

In this section, we present our GAN model more formally. Our proposed model can be used for computing the implied and local volatility. The only difference is in the input to the model and the target used. For the implied volatility, the input of our proposed model is composed of the option price V_{mkt}, the risk free rate r, the moneyness $k=\log(K/s_0)$, and the time to maturity T.

First, we construct the loss function to approximate the implied volatility. Let $\mathcal{K}$ and $\mathcal{T}$ be the training set for the log moneyness and time to maturity, respectively. For each $k\in\mathcal{K}$ and $T\in\mathcal{T}$, we draw M different r from the range $[0.0,0.05]$ and corresponding option prices $\mathbf{V}=[(V_{mkt})_i,...,(V_{mkt})_M]^\top$. We let $\mathbf{Y}=[y_1,...,y_M]^\top$ be the labels of the data, i.e. $y_i=(\sigma_{mkt})_i$. Let X be the input to the GAN, $X=\{r\mathbf{1},\mathbf{V},(K/s_0)\mathbf{1},k\mathbf{1},T\mathbf{1}\}$, where $\mathbf{1}$ is a vector of ones of dimension M. The probability distribution of X is denoted by $\mathbb{P}_X$.

Let $G:\mathbb{R}^{5M}\mapsto\mathbb{R}^M$ be the generator and $D:\mathbb{R}^{6M}\mapsto[0,1]^M$ be the discriminator. Let $Z=\{\mathbf{Z}_1,\mathbf{Z}_2,\mathbf{Z}_3,\mathbf{Z}_4,\mathbf{Z}_5\}$, where $\mathbf{Z}_i=[z_i,...,z_M]^\top$ and $z_i\sim N(0,1)$ is the standard normal distribution. The probability distribution of Z is denoted by $\mathbb{P}_Z$. In a standard GAN model, the learning objective is to ensure the distribution of $G(Z)$ is similar to the

distribution of Y. We let $\hat{X} = \{X, \mathbf{Y}\}$ and let $\hat{Z} = \{Z, G(Z)\}$, then the standard loss function for the GAN is given by the minimax problem [68]:

$$\min_G \max_D \mathcal{L}(G, D) = \min_G \max_D \left\{ \mathbb{E}^{\mathbb{P}_X}[\log(D(\hat{X}))] + \mathbb{E}^{\mathbb{P}_Z}[1 - \log(D(\hat{Z}))] \right\}$$

Solving the standard GAN loss function is difficult, so in practice we reformulate the the standard loss function as two problems. The reformulation of the standard GAN loss function is commonly done in practice in order to avoid vanishing gradients of the generator [12]. However, it has been shown that the gradients $-log(D(\cdot))$ can be unbounded, so a small learning rate should be used for training, [12]. The reformulated GAN problem is given by

$$\begin{aligned} \max_D \mathcal{L}(D) &= \max_D \{\mathbb{E}^{\mathbb{P}_X}[\log(D(\hat{X}))] + \mathbb{E}^{\mathbb{P}_Z}[1 - \log(D(\hat{Z}))]\} \qquad (G \text{ is fixed}) \\ \min_G \mathcal{L}(G) &= \min_G \{-\mathbb{E}^{\mathbb{P}_Z}[\log(D(\hat{Z}))]\} \quad (D \text{ is fixed}). \end{aligned}$$

These two loss functions are solved iteratively such that G is fixed when minimizing $\mathcal{L}(D)$ and D is fixed when minimizing $\mathcal{L}(G)$.

In our framework, we do not use the standard GAN loss function. We use the conditional GAN which introduces the labels as a feature to the model. To minimize the number of arbitrage violations from we formulate our generator loss function similarly to [2]. We also use the mean squared error (MSE) to learn the shape of the implied volatility surface [84]. For M simulations, the MSE is given by

$$MSE(\mathbf{Y}, G(Z)) = \frac{1}{M} \sum_{i=1}^{M} (y_i - G(z_i))^2. \tag{4.8}$$

One important aspect of our GAN model is to incorporate the no-arbitrage conditions. However, instead of imposing the conditions on the volatility $\sigma_{IV}(k, T)$ directly, we will construct constraints on the total variance, [2]. Let ℓ_{cal} be the risk in the total variance from the calendar spread arbitrage, and ℓ_{but} be the risk from the butterfly and vertical spread arbitrage. They can be written as:

$$\begin{aligned} \ell_{cal}(k, T) &= \partial_T w(k, T) \\ \ell_{but}(k, T) &= \left(1 - \frac{k\partial_k w(k, T)}{2w(k, T))}\right)^2 - \frac{\partial_k w(k, T)}{4}\left(\frac{1}{w(k, T)} + \frac{1}{4}\right) + \frac{\partial^2_{kk} w(k, T)}{2}. \end{aligned}$$

Note that we want $\ell_{cal} \geq 0, \ell_{but} \geq 0$ to eliminate arbitrage violation [62]. The proof of why $\ell_{but} \geq 0$ ensures the butterfly spread arbitrage is not violated is shown in [62].

In our method we enforce the constraints by imposing penalty terms on the generator. We define the penalty terms to satisfy the no-arbitrage conditions more formally, [2]. Let $\mathcal{I}_{cal} = \{(k,T) : k \in [k_{\min}, k_{\max}], T \in [0,\infty)\}$, $\mathcal{I}_{but} = \{(k,T) : k \in [k_{\min}, k_{\max}], T \in [0,\infty)\}$, and $\mathcal{I}_{\infty} = \{(k,T) : k \in \{6k_{\min}, 4k_{\min}, 4k_{\max}, 6k_{\max}\}, T \in [0,\infty)\}$ [2] then the no-arbitrage penalty terms are given by:

- **(Monotonicity in T):**

$$\mathcal{L}_c = \frac{1}{|\mathcal{I}_{cal}|} \sum_{k,T \in \mathcal{I}_{cal}} \max(0, -\ell_{cal}(k,T)).$$

- **(Durrleman's condition):**

$$\mathcal{L}_b = \frac{1}{|\mathcal{I}_{but}|} \sum_{k,T \in \mathcal{I}_{but}} \max(0, -\ell_{but}(k,T)).$$

- **(Large moneyness limit):**

$$\mathcal{L}_{\infty} = \frac{1}{|\mathcal{I}_{\infty}|} \sum_{k,T \in \mathcal{I}_{\infty}} |\partial^2_{kk}\omega(k,T)|.$$

To generate arbitrage-free implied volatility, we incorporate the no-arbitrage conditions $\mathcal{L}_c, \mathcal{L}_b$ and $\mathcal{L}_{\infty}$ in the loss function of the generator. Furthermore, to ensure that $G(Z)$ has a similar distribution to y, we also want to minimize the negative log-likelihood loss function given by

$$\mathcal{L}_{D_G} = -\frac{1}{M} \sum_{i=1}^{M} \log(D(\hat{Z}_i)),$$

where $\hat{Z}_i = \{z_i, (G(Z))_i\}$. Putting everything together, the loss function for the generator is given by

$$\mathcal{L}_G = MSE(\mathbf{Y}, G(Z)) + \lambda_1 \mathcal{L}_c + \lambda_2 \mathcal{L}_b + \lambda_3 \mathcal{L}_{\infty} + \lambda_4 \mathcal{L}_{D_G}. \tag{4.9}$$

The parameters $\lambda_1 > 0$, $\lambda_2 > 0$, and $\lambda_3 > 0$ are used to determine the amount of calendar arbitrage, butterfly arbitrage and the limit behaviour that are penalized. $\lambda_4 \in [0,1]$ is a hyperparameter that controls how much the generator will fool the discriminator. Note that the additional terms act as regularizers to the standard MSE loss function.

For the discriminator network, we maximize the binary cross-entropy (BCE) loss function [68] defined as:

$$\mathcal{L}_D = \frac{1}{M}\sum_{i=1}^{M}\Big(\log(D(\hat{X}_i)) + \log(1 - D(\hat{Z}_i))\Big), \tag{4.10}$$

where $\hat{X}_i = \{r, V_i, (K/s_0), k, T, y_i\}$. The discriminator loss is chosen to maximize the likelihood that the discriminator classifies the target values as true given some noisy inputs.

In our framework, we model G and D using feedforward neural networks. We define the generator neural network as $G_\phi = G(\cdot;\phi)$ with a set of parameters ϕ. We let $\hat{Z}_\phi = \{Z, G_\phi(Z)\}$ and define the discriminator neural network as $D_\theta = D(\cdot;\theta)$ with a set of parameters θ. The generator network we use is the ANN with $l = 1, 2$ layers. We did not use a deep neural network as other methods on volatility computation such as [98] and [32]. We found that increasing the depth past $l = 2$ did not improve the accuracy of the generator network. We observe that if the depth past $l = 1$ in the discriminator network, it may actually lead to a degradation of performance as it was over-fitting the trained data. Thus D_θ is always constructed with $l = 1$.

Let $\theta_l \in \theta$ and $\phi_l \in \phi$ be the weights of the $l = 1, .., L$ layers in the discriminator and generator neural network. Then $D_\theta(\hat{X})$ with parameters θ is represented by

$$\begin{aligned} h_1 &= \text{softplus}(\theta_1^\top \hat{X}; \beta), \\ D_\theta(\hat{X}) &= \text{sigmoid}(\theta_2^\top h_1). \end{aligned}$$

We denote GAN-1 as one layer $(G_\phi^1)(Z)$ (i.e. $l = 1$), with parameters ϕ_1 where

$$\begin{aligned} q_1 &= \text{softplus}(\phi_1^\top Z; \beta) \\ (G_\phi^1)(Z) &= \text{softplus}(\phi_2^\top q_1; \beta), \end{aligned}$$

and denote GAN-2 as two layer generator network $(G_\phi^2)(Z)$ (i.e. $l = 2$). The two layer GAN is given by:

$$\begin{aligned} q_1 &= \text{softplus}(\phi_1^\top Z; \beta) \\ q_2 &= \text{softplus}(\phi_2^\top q_1; \beta) \\ (G_\phi^2)(Z) &= \text{softplus}(\phi_3^\top q_2; \beta). \end{aligned}$$

We use a scaled version of the softplus activation function and sigmoid activation function defined as [45, 68]

$$\text{softplus}(X;\beta) = \frac{1}{\beta}\log(1+e^{\beta X}),$$
$$\text{sigmoid}(X) = \frac{1}{1+e^{-X}}.$$

We remark that the softplus activation function is used for the generator network output since we want a smooth enough function to learn no-arbitrage soft constraints [45]. The sigmoid activation function for the discriminator output is standard for classification problems [68]. We train the discriminator network first using $G_{\phi^*}(Z)$ with ϕ^* fixed. An illustration of the fixed discriminator is shown in Figure 4.1. Similarly, we train the generator network using the $D_{\theta^*}(\hat{X})$ with θ^* fixed. An illustration of the fixed generator is shown in Figure 4.2. The loss function for the discriminator neural network is composed of a reward for correctly classifying Y as σ_{mkt} and penalized when it incorrectly classifies $G_{\phi^*}(Z)$ as σ_{mkt}:

$$\mathcal{L}_{D_\theta} = \frac{1}{M}\sum_{i=1}^{M}\left(\log(D_\theta(\hat{X}_i)) + \log(1 - D_\theta((\hat{Z}_{\phi^*})_i))\right). \tag{4.11}$$

The generator is composed of the MSE loss which solves (4.7), the penalty terms $\mathcal{L}_c, \mathcal{L}_b, \mathcal{L}_\infty$ and a reward for successfully evading the discriminator as:

$$\mathcal{L}_{G_\phi} = MSE(\mathbf{Y}, G_\phi(Z)) + \lambda_1\mathcal{L}_c + \lambda_2\mathcal{L}_b + \lambda_3\mathcal{L}_\infty + \lambda_4\mathcal{L}_{D_G}, \tag{4.12}$$

where

$$\mathcal{L}_{D_G} = -\frac{1}{M}\sum_{i=1}^{M}\log(D_{\theta^*}((\hat{Z}_\phi)_i)).$$

The optimal set of discriminator network parameters is denoted by θ^* and is found by

$$\theta^* = \arg\min_\theta\{\mathcal{L}_{D_\theta}\}.$$

The optimal set of generator network parameters is denoted by ϕ^*, which is found by minimizing (4.9) this gives us

$$\phi^* = \arg\min_\phi\{\mathcal{L}_{G_\phi}\}.$$

Our loss function allows us to train a shallow generator to compute implied volatility surfaces. This makes computation efficient and accurate. Our loss function introduces

no-arbitrage penalty terms that penalize the generator network when the output violates arbitrage. We also reward the generator network when the output evades the detection of the discriminator network. This improves the quality of the generator output compared to a generator with only mean squared error loss.

4.3.3 Training the GAN

Training the GAN requires two steps. The first step is to train the discriminator. However, loss (4.11) is not straightforward to compute directly as it involves both the generator and the discriminator. Instead we use an alternating training schedule. In the alternating schedule we first train the discriminator as a classifier such that we maximize (4.11) using the Adam optimizer [92]. In the second step, given X, Z and $\mathbf{Y}$ we train the generator by minimizing (4.12) using Adam. Training is performed for 50 epochs and to mitigate the vanishing gradient issues found in GAN training, we use the softplus activation function in our hidden layers. We use the weights of the generator neural network and the discriminator neural network from the previous epoch to initialize the weights of the next epoch. Once training is complete, we evaluate the GAN with a forward pass using the testing data.

The architecture and training parameters are detailed in table 4.1.

Table 4.1 Model Parameters used in GAN for GAN-1 and GAN-2.

Parameters	**Options**
Neurons(each layer)	100
Activation function	softplus ($\beta = 1$)
Dropout rate	0.0
Batch-normalization	No
Optimizer	Adam
Batch size	128

4.3.4 Generating data for training, validation and testing

We outline the data generation used to train, validate and test our model. Our proposed model can be used to compute implied volatility surfaces using a discriminator neural network and a generator neural network. The two neural networks require different input features for training and testing.

In this thesis, our market option price is generated using the Heston model with different combination of parameters. We use synthetic data over real data as it is difficult to obtain good quality real data for all T and K. We found that this is a common approach used in literature [98, 97]. In this section we provide more details on the training and testing data used in this chapter. We generate three separate sets of data. One set used for training and validating our model. A second set for testing volatility surfaces and a third set for testing price errors.

Generating synthetic volatility data

We generate synthetic market data for training our GAN framework. The characteristic function of the Heston model was used with the COS method to generate the European call price, of an underlying asset following the geometric Brownian motion [53]. Then the Black-Scholes implied volatility was computed using Brent's method [98].

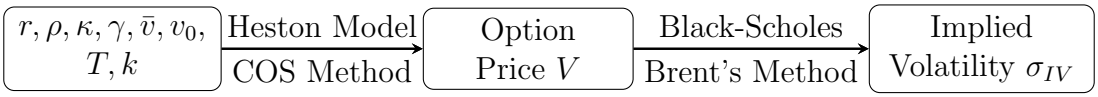


Figure 4.3 Pipeline used to generate data for training and testing our model.

For training and validating, we use the parameters shown in table 4.2 in our simulations. Each parameter was sampled randomly from the uniformly distributed parameters.

Table 4.2 Domain of parameters used in the simulation of training and testing data, all parameters were drawn randomly from the parameter space.

Parameter	**Range**
r, risk-free interest rate	$(0.0, 0.05)$
κ, reversion speed	$(0.0, 3.0)$
ρ, correlation	$(-0.9, 0.0)$
γ, volatility of variance	$(0.01, 0.5)$
$\bar{v}$, long-run mean variance	$(0.01, 0.5)$
v_0, initial variance	$(0.05, 0.5)$
K/s_0, moneyness	$(0.5, 2.5)$
T, time to maturity	$(0.5, 2.0)$

For training and validating, we generated 10 different combinations of parameters above. For each set of parameters, we generated the option price and implied volatil-

ity for 75 different maturity dates, and 50 different strikes. For testing volatility surface we use 1 set of parameters over 11 different maturities and 157 different strikes with parameters shown in table 4.3. For testing pricing error we use 50 different sets of parameters, 8 different maturities and 11 different strikes with parameters shown in table 4.2.

Table 4.3 Domain of parameters used in out-of training volatility surface generation, all parameters were drawn uniformly from the domain.

Parameter	**Range**
r, risk-free interest rate	0.02
κ, reversion speed	2.7
ρ, correlation	-0.4
γ, volatility of variance	0.2
$\bar{v}$, long-run mean variance	0.4
v_0, initial variance	0.4
K/s_0, moneyness	$(0.3, 2.8)$
T, time to maturity	$(0.3, 2.0)$

Volatility data from call options on the S&P 500

We test our GAN framework on market index options built on the S&P 500 index. The options dataset has the following features. The strike, moneyness, bid price, mid price, ask price, last price, volume, implied volatility and time traded. We gathered our data from *Barchart*. On June 15^{th}, 2023, we gathered monthly options data for 15^{th} of September, December, March and June, which we set as the time to maturity $T = 0.25, 0.5, 0.75, 1$. The interest rate r was set to the 1-year treasury bond rate of 5.463%. We collected a total of 821 data points of $K/s_0 \in [0.24, 1.17]$ and time-to-maturity $\in \{0.25, 0.5, 0.75, 1.0\}$.

4.4 Numerical Results

In this section, we present our experimental results. Our experiment is divided into two main categories. First we show the that our proposed method can compute the implied volatility surface. We simulate $\mathcal{M}_{train} = 24,000$ training points and $\mathcal{M}_{test} = 6000$ testing points. To measure the accuracy of our method we use the mean absolute percent error

(MAPE) as a measure of performance. The MAPE is given by:

$$MAPE = \frac{1}{\mathcal{M}_{test}} \sum_{i=1}^{\mathcal{M}_{test}} \frac{|\sigma_{mkt,i} - \sigma_{IV,i}|}{\sigma_{mkt,i}}.$$

We also evaluate another measure of performance, the mean absolute error(MAE), which is given by:

$$MAE = \frac{1}{\mathcal{M}_{test}} \sum_{i=1}^{\mathcal{M}_{test}} |\sigma_{mkt,i} - \sigma_{IV,i}|.$$

We also use repricing error as a method to measure the quality of our method [84, 32, 31]. We compute the average relative pricing error (ARPE) of the European option given by:

$$ARPE = \frac{1}{\mathcal{M}_{test}} \sum_{i=1}^{\mathcal{M}_{test}} \frac{|V_{mkt,i} - V_i|}{|V_{mkt,i}|},$$

where V is the Black-Scholes European call option price computed using σ_{IV}. We use two other metrics to measure the performance of our proposed method. We use the maximum relative price error (MRPE) given by:

$$MRPE = \max_{i=1,\ldots,\mathcal{M}_{test}} \frac{|V_{mkt,i} - V_i|}{|V_{mkt,i}|},$$

and the standard deviation of relative price error. Note that all three measures are presented as error heatmaps [84]. In all of our experiments, we use generated data as described in Section 4.3 to train, validate and test our model. We use the following values for $\lambda_1 = 1, \lambda_2 = 1, \lambda_3 = 1, \lambda_4 = 0.5$ for our experiments.

In experiment 1, we observe the effectiveness of our model with soft constraints and without. We also compare our model with a GAN that does not use the MSE loss for computing local volatility [120]. In experiment 2, we compare our proposed methods with the IV-ANN method of [98], we chose not to compare with the deep calibration [84] approach as the network architecture used in this model is captured by the IV-ANN method. We compute the implied volatility and present a qualitative and quantitative analysis. We also look at the repricing error of our method and the IV-ANN method to replicate V_{mkt}. In experiment 3, we use a pre-trained discriminator to train a generator to compute volatility surfaces on data. We generate market volatility surfaces using limited market data. In experiment 4, we compare our method to a VAE local volatility implementation. In this experiment, we compute the local volatility from (4.4). Both methods are given a set computation time and evaluate the performance of both methods.

4.4.1 Experiment 1: comparison of GAN models, the effects of soft constraints and MSE loss

In this experiment, we compare the two GAN models proposed in our chapter. We use MAE and MAPE to measure the performance of both models for implied volatility. For the local volatility we use the ARPE, MRPE and the standard deviation of repricing error heatmaps. To begin our comparison, we compare the training performance of GAN-1 and GAN-2. The training procedures for GAN-1 and GAN-2 are identical and are described in section 4.3.

We summarize the training performance of GAN-1 and GAN-2 in table 4.4, which shows the MAE and MAPE of both models and their respective training times. We notice that GAN-1 trains faster than GAN-2 by 10 seconds but GAN-2 tends to yield better accuracy than GAN-1 in both MAE and MAPE. However, only looking at error does not give us a full picture of the performance of our proposed method. Thus we also compare more qualitative results.

Table 4.4 We compare the performance of GAN-1 and GAN-2 using training time, MAE and MAPE.

Model	**Training Time**	**MAE**	**MAPE**
GAN-1	$198.689732s$	$4.2680e^{-5}$	0.07898%
GAN-2	$208.098838s$	$2.1376e^{-5}$	0.03981%

We compare the qualitative performance of our proposed models GAN-1 (dashed red) and GAN-2 (dashed green) for different maturities as shown in Figure 4.4. We show a cross section of the implied volatility surface for different maturities to highlight the tail behaviour of our generated surface.

Our target value shown by the solid blue line was constructed from the implied volatility. The red dashed lines are the implied volatilities output from GAN-1 and the green dashed lines are the implied volatilities output from GAN-2. We can see that the implied volatility approximated by GAN-2 follows the implied volatility curve but does not match it completely. The implied volatility approximated by GAN-1 fails to capture the curvature of the implied volatility curve in maturities further out. Though GAN is a powerful tool, the generator network still requires some hidden layers to perform well. However, the training time required for GAN-2 is only 10 seconds more than GAN-1 showing that it is still efficient.

We compare the generated implied volatility from our models with soft constraints

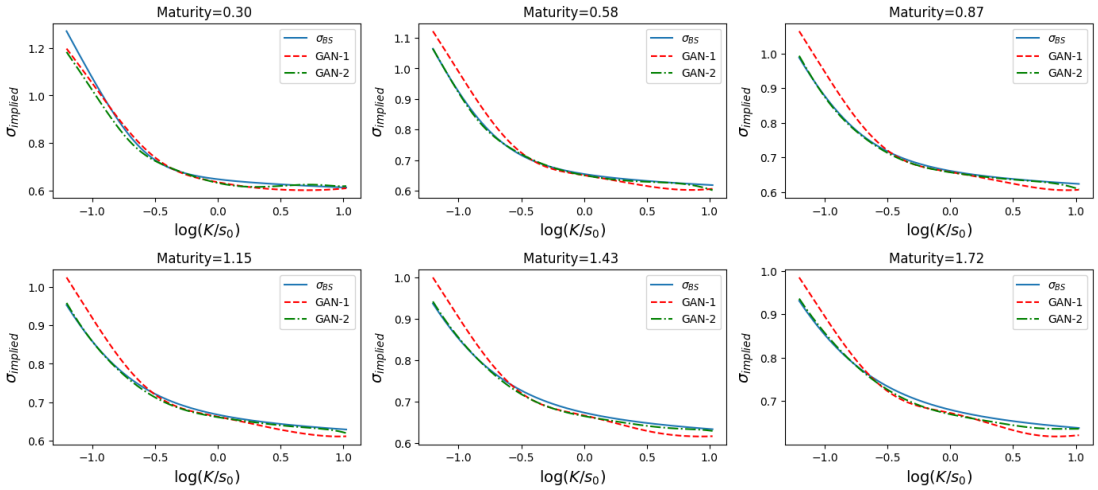


Figure 4.4 The implied volatility computed by GAN-1 with soft constraints (red) and GAN-2 with soft constraints (green) compared to the Black-Scholes implied volatility (blue).

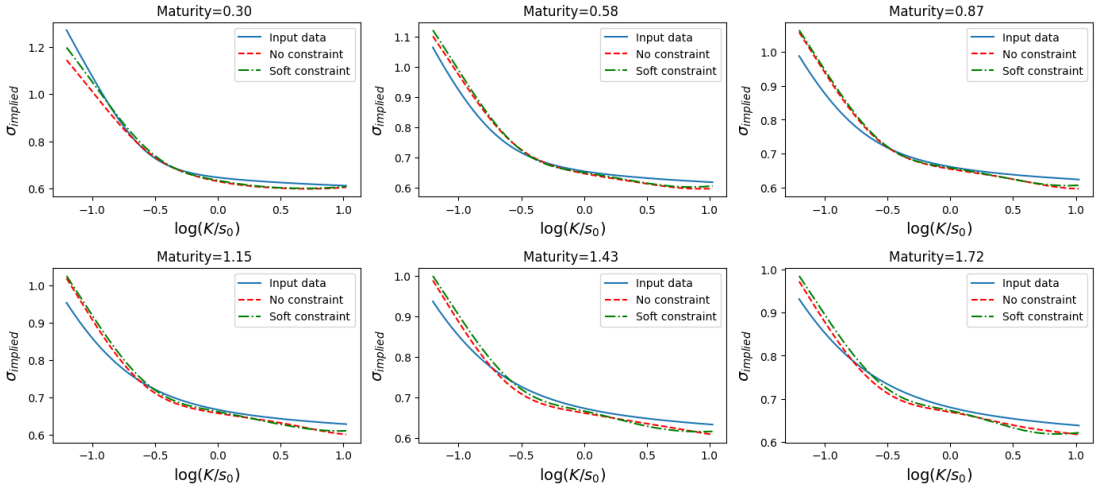


Figure 4.5 The implied volatility approximated by GAN-1 with no constraints (orange) and GAN-1 with soft constraints (green) compared to the Black-Scholes implied volatility (blue).

(dashed green) vs. our models without any constraints (dashed red) for GAN-1 in Figure 4.5 and for GAN-2 in Figure 4.6. As with Figure 4.4, we look at the cross section of the implied volatility surface at different maturities. We observe the role of the regularizer as maturity increases. We see that the no-arbitrage penalty terms flatten the curve and allow for better fitting with the implied volatility curve. In the case with no constraints we can qualitatively observe some concavity in the implied volatility curves which suggests that arbitrage has been violated under the Heston model. In general, implied volatility can remain arbitrage-free with concavity in the presence of jumps.

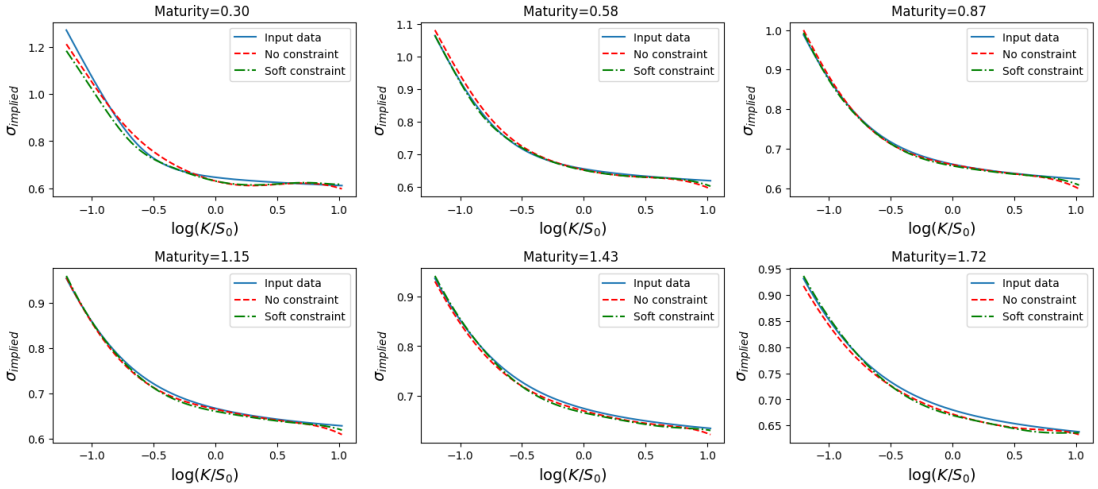


Figure 4.6 The implied volatility approximated by GAN-2 with no constraints (orange) and GAN-2 with soft constraints (green) compared to the Black-Scholes implied volatility (blue).

We show the number of arbitrage violations during training and testing for both GAN models when soft constraints are used and when they are not used in tables 4.5 and 4.6. We found that adding the soft constraints greatly assisted the generator in avoiding arbitrage butterfly and vertical spread violations. From testing, we see that GAN-1 and GAN-2 without soft constraints performs considerably worse, as we had 4.90% and 11.15% arbitrage violations respectively. From table 4.5 we see a reduction in butterfly and vertical spread arbitrage violations when soft constraints are used. The soft constraints do not guarantee that arbitrage violations do not occur.

Table 4.5 Number of butterfly arbitrage violations detected in implied volatility surface.

	GAN-1		**GAN-2**	
	Soft Constraint	No Constraint	Soft Constraint	No Constraint
Training	0/28306	1049/28306	0/28306	2143/28306
Testing	0/4995	245/4995	0/4995	557/4995

Table 4.6 Number of calendar arbitrage violations detected in implied volatility surface.

	GAN-1		**GAN-2**	
	Soft Constraint	No Constraint	Soft Constraint	No Constraint
Training	0/28306	0/28306	0/28306	0/28306
Testing	0/4995	0/4995	0/4995	0/4995

Comparison to GAN without MSE Loss

We construct GAN-1 and GAN-2 without MSE loss function to compare our formulation with the formulation presented in [120]. In our experiment we found that without the MSE loss term the GAN is unable to fully capture the implied volatility surface. More specifically, it undershoots the volatility value resulting in a surface that is inconsistent with the market. This is likely due to the synthetic data being difficult to segment. In Figure 4.7 we show the GAN-1 and GAN-2 network without MSE.

In Figure 4.7, we see that the implied volatility generated by the GAN without MSE cannot fit the ground truth implied volatility at any point and fails to estimate the market implied volatility surface. This highlights the importance in the inclusion of the MSE term in the loss function.

4.4.2 Experiment 2: comparison of Implied Volatility with IV-ANN

In this experiment, we compare the performance of our proposed methods to the IV-ANN method [98] with 4 hidden layers with the ReLU activation function. We design the GAN network such that the MAE of the two approaches are similar. Doing this we find that our proposed GAN-2 is roughly two times faster than the IV-ANN approach. The performance is gauged by the mean absolute error of the predicted values. The MAE for the IV-ANN was reported as $9.73e^{-5}$ in [98]. We summarized the training time, mean absolute error and the mean absolute percent error for each method in table 4.7. As seen in table 4.7,

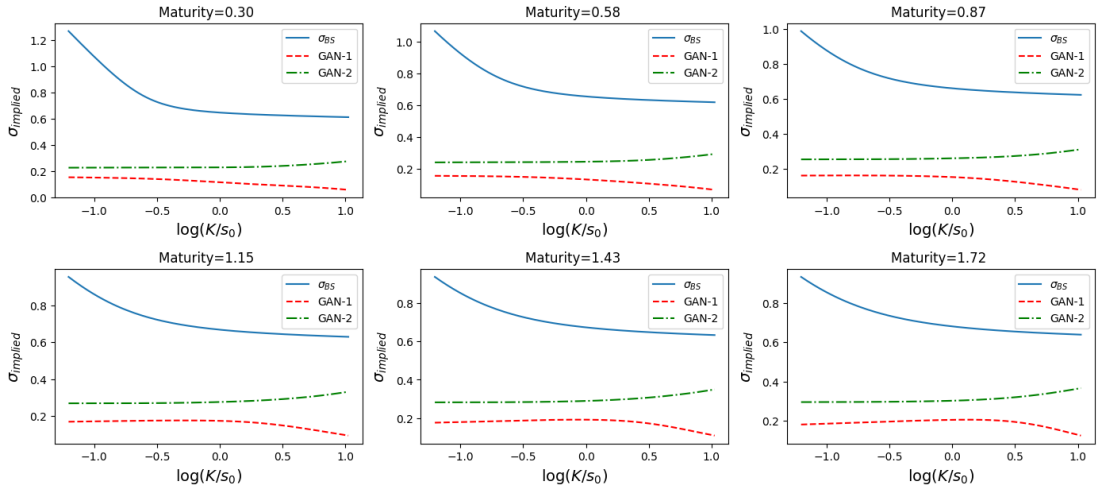


Figure 4.7 The implied volatility computed by GAN-1 without MSE (red) and GAN-2 without MSE (green) compared to the Black-Scholes implied volatility (blue).

using the same training data and testing set we can see that GAN-2 outperforms IV-ANN in runtime when they are both trained to a similar mean absolute error.

Table 4.7 Timing and error comparison between GAN-2 and IV-ANN

Model	**Training Time**	**MAE**	**MAPE**
IV-ANN	446.204831	$2.3235e^{-5}$	0.044106%
GAN-2	209.194824	$2.1376e^{-5}$	0.039810%

Next we evaluate the quality of our proposed method vs. the IV-ANN method. We show the implied volatility learned by the IV-ANN method in Figure 4.8. Figure 4.8 looks at the cross section of the implied volatility surface generated by the IV-ANN method for different maturities. From Figure 4.8 we see that the implied volatility learned by IV-ANN has difficulty fitting the Black-Scholes implied volatility qualitatively.

We compare the repricing performance of our proposed method vs. the IV-ANN method. The repricing error heatmap of the IV-ANN method and GAN-2 is show in Figure 4.9. We see that our proposed GAN-2 network outperforms the IV-ANN method. The IV-ANN method has a maximum ARPE of $0.080\% \pm 0.12\%$ and a MRPE of 0.70%. Our proposed method has a maximum ARPE of $0.08\% \pm 0.1\%$, and a MRPE of 0.50%.

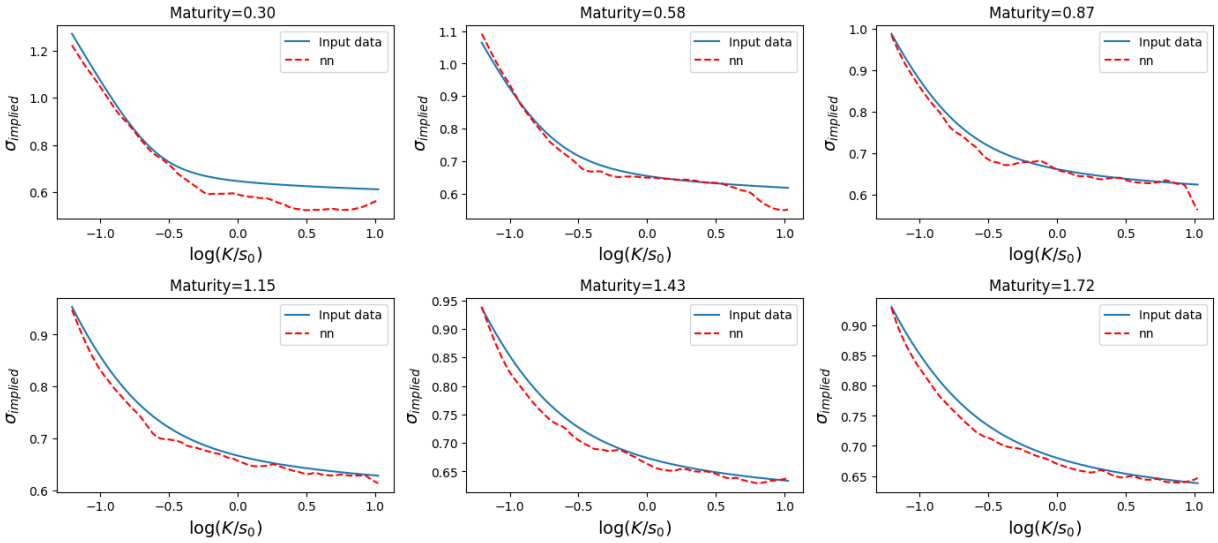


Figure 4.8 Implied volatility cross section generated by the IV-ANN for different maturities.

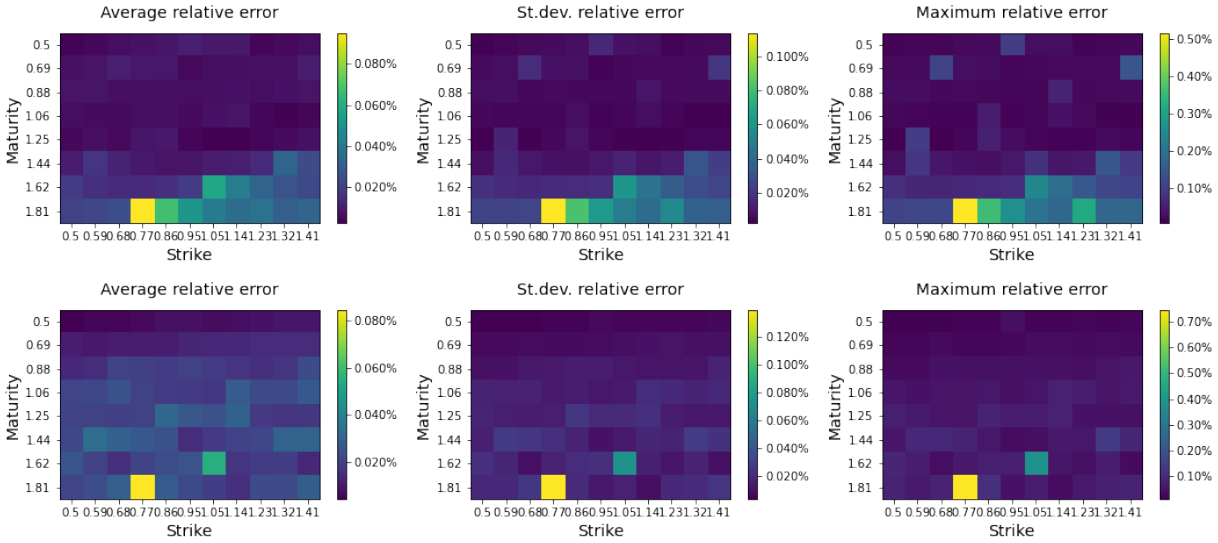


Figure 4.9 GAN-2 implied volatility (top) and IV-ANN implied volatility (bottom repricing error heatmap. The warmer colors (yellow) respresent higher error values and colder colors (blue) represent lower error values.

4.4.3 Experiment 3: computing market consistent implied volatility surfaces

In this experiment we show that our method can be adapted to be used on market data with minimal additional training with limited sample data. This is achieved using a pre-trained a 2-layer discriminator with synthetic data. We fine-tune the 2-layer generator which is retrained on a limited subset of the market data over 200 epochs with a batch size of 32. The GAN-2 model is tested on a separate test set that was not used for training the generator.

We use GAN-2 to generate a market consistent implied volatility surface, this shows the capability of our network to generate market consistent volatility surfaces just by retraining the generator. We present the market consistent implied volatility surface in Figure 4.10.

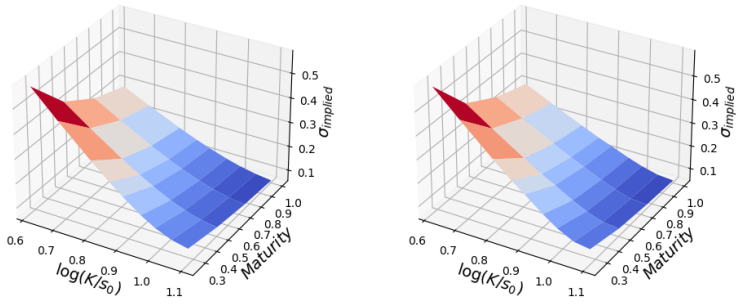


Figure 4.10 Implied volatility generated by the generator (left) and the market implied volatility computed by Brent's method (right).

Figure 4.10 shows the implied volatility surfaces from our generator and the market implied volatility. We see they are near identical qualitatively. Quantitatively, we measured the MAE score for GAN-2 as 0.007 and the MAPE score as 7.28%.

4.4.4 Experiment 4: comparison of local volatility computation with VAE

In this experiment, we compare the local volatility computed by our proposed method with the VAE method. We compute the local volatility using equation (4.4). We implemented the VAE using 2 hidden layers with 200 nodes [18]. The number of hidden layers and

nodes were chosen such that the training time for the VAE and our GAN methods were the same. The encoder was constructed with a ReLU activation function for the hidden layers and a sigmoid activation function for the output. The decoder was constructed with ReLU activation function for all layers. The VAE was trained over the same training set as our proposed method.

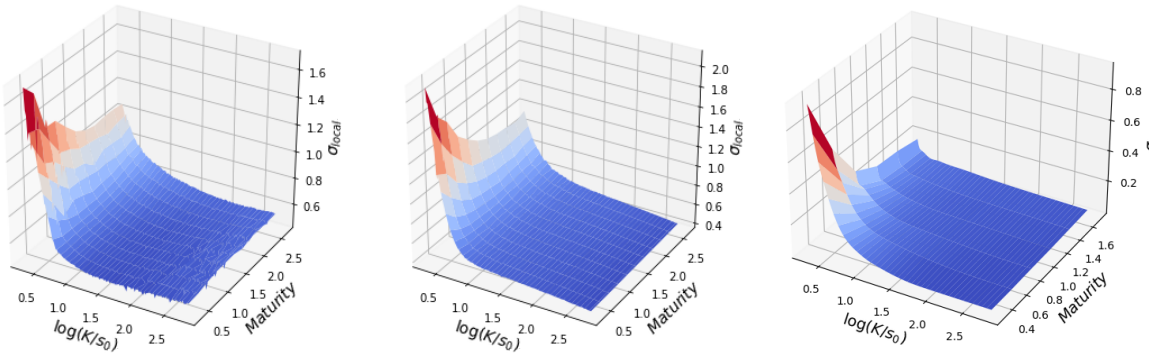


Figure 4.11 Local volatility surface generated by trained VAE using a shallow network (left), VAE using a deep network (middle), and computed from implied volatility using Equation (4.4) (right).

First, we observe that the resulting local volatility surface computed from our method is qualitatively different. This is because we use Equation (4.4) to compute our local volatility with autodifferentiation. The VAE implementation is trained on local volatility computed from a finite difference approximation of Equation (4.3). Given the same training time, the local volatility surface by our proposed method is smooth as shown in Figure 4.11, the right most figure. The smoothness of our result is from the autodifferentiation of the the implied volatility surface. This is in contrast to the noisy surface generated from the VAE as shown in the left hand side of Figure 4.11. Noise is present in VAE because the encoder network is trained to generate the parameters of a normal distribution that fits the data and data can be sampled using these learned parameters [91]. This adds noise to the output if there is not a sufficient amount of samples generated. We note that the VAE method can produce smooth local volatility surfaces as shown in the right hand side of Figure 4.11 however it requires a deeper network for the encoder or larger batch sizes. Adding the extra layers to smoothen the local volatility surface resulted in a training time of 1036.027 seconds. Since the VAE is trained on an approximation of the local volatility, there is approximation error in the output.

4.5 Conclusion

In this chapter, we present a framework to generate implied volatility surfaces efficiently using GAN. By using a generator and discriminator together, we are able to create a model framework that is efficient and accurate by using shallow-narrow networks. We use no-arbitrage penalty terms with the MSE loss function to penalize arbitrage opportunities generated by the generators. The log-likelihood estimation of the discriminator adds to the generator by allowing the posterior generator to successfully output a valid volatility surface that is consistent with the option price. The discriminator is trained as a classifier to classify the implied volatility as true or false.

Chapter 5

Conclusion and Future Work

5.1 Summary

In this thesis, we propose machine learning solutions for solving HJB equations efficiently with applications in pricing and hedging American options, optimal trade execution, and solving the inverse problem to compute volatility surfaces. We present methods to solve specific problems in an efficient way, using supervised and reinforcement learning. This is achieved through the appropriate derivation of loss functions and different architectures to capture important domain information. We also show that by using machine learning we can extend classical problems in finance to study higher-dimensional problems that used to be difficult to solve.

Firstly, in Chapter 2 we propose a deep Recurrent network framework for pricing and hedging high-dimensional American option. Our proposed method achieves a better runtime efficiency than the DRL method. We reformulate the BSDE as a least-squares problem that is solved using a neural network. We present a loss function (2.12), which allows us to train our deep RNN framework in all spacetime for the continuation price and delta with theoretical convergence results as shown in [86]. The theoretical time complexity of our proposed method is linear in the number of timesteps N which is better than the quadratic results of the DRL method [34]. Interestingly, we found that as N decreases there is constant increase of the quadratic complexity term. Our Algorithms 1 and 2 yields prices and deltas at all points in space and time and it is more accurate than the DRL method under time constraints.

Secondly, in Chapter 3 we solve the optimal trade execution problem under the Markowitz criteria. We formulate an auxiliary time-consistent optimal trade execution problem for

multiple assets and multiple agents setting. This is accomplished by formulating the original mean-variance problem into its dual representation coupled with an auxiliary HJB equation [4]. To extend this into higher dimensions, we derived the high-dimensional auxiliary HJB equation and reformulated it into a BSDE that can be solved using a neural network. We present a deep residual U-net with self-attention network to solve the BSDE presented in (3.19). Our framework overcomes the issue of saving weights for every timestep N and overcomes the vanishing gradient problem of recurrent networks [102]. We show that the proposed method matches analytical methods where applicable. We also show sensitivities of the proposed method with respect to $\phi, \gamma, \kappa_t, \kappa_p$. When an agent is more conscious of other agents in the market, i.e., ϕ increases, agents tend to invest more in risky assets. When an agent is more risk averse, i.e., γ increases, the agent invests less in risky assets. We saw that as the temporary price impact of the agent increases they tend to liquidate more gradually and as the permanent price impact increases the agents liquidates less frequently.

Lastly, in Chapter 4, we present a framework to generate volatility surfaces efficiently using GAN. By using a generator and discriminator together we are able to create a model framework that is efficient and accurate with shallow-narrow networks. We used no-arbitrage penalty with the MSE loss function to penalize arbitrage opportunities generated by the generators. The log-likelihood estimation of the discriminator adds to the generator by allowing the posterior generator to successfully output a valid volatility surface that is consistent with the option price. The discriminator was trained as a classifier to classify the volatility as true or false.

Through our numerical results, we demonstrate that GAN can be used to restrict the output space of a generator network, allows us to train shallow networks without losing accuracy. This ultimately improves efficiency by training smaller networks. We also demonstrate that regularization using no-arbitrage soft constraints helps mitigate arbitrage violations for implied volatility computation. We further show the capability of our network to compute the local volatility and compared our model with a deep neural network and SSVI. Our method produces a maximum ARPE of $0.1\% \pm 0.117\%$ which is better than both the deep neural network and SSVI method with maximum ARPE of $0.8\% \pm 0.2\%$. Our proposed method also has a MRPE comparable to the SSVI at 1%. Finally, we show that our model can be generalized to market data with limited data samples using a pre-trained discriminator and re-training the generator.

5.2 Future Work

In this section we discuss future research directions that can extend our work in Chapters 2-4.

5.2.1 Extensions to pricing high-dimensional American options using recurrent networks

In Chapter 2, we present a deep recurrent network to solve the high-dimensional American option. The main drawback of the deep recurrent network is that, when there are no time constraints, our method is not as accurate. This can be explained by the difference in accuracy between different loss functions used to solve (2.7) as shown in [86]. A potential future work is to improve the accuracy of the method by improving the Recurrent network architecture, or an extension to transformer networks or sequence models.

Another avenue of work can also explore the use of neural network sample generators to generate better sample data for training. This can greatly help in the use of machine learning in finance. A generator that can distribute an even grid of random numbers, i.e., quasi-random number generators, can create dense simulation paths to encourage better learning as quasi-Monte Carlo has stable convergence to the mean.

5.2.2 Extensions to optimal trade execution problem

In Chapter 3, we extended the optimal trade execution problem into K agents and d assets. We proposed a CNN approach coupled with the self-attention mechanism to overcome the issues of reinforcement learning framework. However, there are many different aspects left to explore. The model we used can be extended to include models for limit order books and trade volumes. This changes the dynamics completely since currently we assume there are no volume constraints on assets [5, 30]. These additional policy constraints can be incorportated using a policy gradient method. Another interesting aspect would be to look at different investment horizons. Different investment horizons would introduce additional complexity to the model in terms of illiquidity of certain assets, which are not currently captured by our framework.

Another extension could be to explore the induced time-consistency approach to solving the mean-variance criteria. This is because by formulating an auxiliary problem of the mean-variance criteria, we are changing the objective function. Even though it is done in

such a way that it can be mapped to the original problem, the controls are sub-optimal solutions to the original problem.

5.2.3 Extensions to computing volatility surfaces using generative methods

In Chapter 4, we discuss how to formulate the computation of volatility surface as a machine learning problem. However, we assume that only one price exists for an option. It would be interesting to explore how generative models behave in a market setting with more than one offered price and how it learns the volatility surfaces from dynamic market.

This has been explored by VolGAN [135]. It showcases the applicability of GANs to generate data for financial applications. However, the flaw of using GANs in this line of work is that GANs are not dynamic models and fail to capture the time dependent dynamics of the distributions. One way to overcome this issue would be to transform the time-series data such that each increment is independent identically distributed. Another way to overcome this issue would be to consider a dynamic generative model such as diffusion models [126].

References

[1] Y. Achdou and O. Pironneau. Computational methods for option pricing. *Frontiers in Applied Mathematics*, 30, 2005.

[2] D. Ackerer, N. Tagasovska, and T. Vater. Deep smoothing of the implied volatiliity surface. *34th Conference on Neural Information Processing System*, 2020.

[3] G. Aivaliotis and J. Palczewski. Investment strategies and compensation of a mean–variance optimizing fund manager. *European Journal of Operational Research*, 234(2):561–570, 2014. 60 years following Harry Markowitz's contribution to portfolio theory and operations research.

[4] G. Aivaliotis and A.Y. Veretennikov. An hjb approach to a general continuous-time mean-variance stochastic control problem. *Random Operators and Stochastic Equations*, 26(4):225–234, 2018.

[5] A. Alfonsi, A. Fruth, and A. Schied. Optimal execution strategies in limit order books with general shape functions. *Quantitative Finance*, 10(2):143–157, 2010.

[6] Robert Almgren. Optimal trading with stochastic liquidity and volatility. *SIAM Journal on Financial Mathematics*, 3(1):163–181, 2012.

[7] Robert Almgren and Neil A Chriss. Optimal execution of portfolio trans-actions. 2000.

[8] Brian D.O. Anderson. Reverse-time diffusion equation models. *Stochastic Processes and their Applications*, 12(3):313–326, 1982.

[9] K. Andersson and C. Oosterlee. A deep learning approach for computations of exposure profiles for high-dimensional Bermudan options, 2020.

[10] Kristoffer Andersson and Cornelis W. Oosterlee. A deep learning approach for computations of exposure profiles for high-dimensional bermudan options. *Applied Mathematics and Computation*, 408:126332, 2021.

[11] M. Annunziato, A. Borzi, F. Nobile, and R. Tempone. On the connection between the Hamilton-Jacobi-Bellman and the Fokker-Planck control frameworks. *Applied Mathematics*, 5(16):2476–2484, 2014.

[12] Martín Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *ICLR*, abs/1701.04862, 2017.

[13] B. Bakker. Reinforcement learning with long short-term memory. volume 14, pages 1475–1482. International Conference on Neural Information Processing Systems: Natural and Synthetic, 2001.

[14] S. Basak and G. Chabakauri. Dynamic mean-variance asset allocation. *The Review of Financial Studies*, 23(8):2970–3016, 2010.

[15] C. Beck, S. Becker, P. Grohs, N. Jaafari, and A. Jentzen. Solving stochastic differential equations and Kolmogorov equations by means of deep learning, 2018.

[16] S. Becker, P. Cheridito, and A. Jentzen. Deep optimal stopping, 2018.

[17] P. Bergault, F. Drissi, and O. Guéant. Multi-asset optimal execution and statistical arbitrage strategies under ornstein-uhlenbeck dynamics. *SIAM Journal on Financial Mathematics*, 13(1):353–390, 2022.

[18] M. Bergeron, N. Fung, J. Hull, Z. Poulos, and A. Veneris. Variational autoencoders: A hands-off approach to volatility. *The Journal of Financial Data Science*, 4(2):125–138, 2022.

[19] T. Bjork, M. Khapko, and A. Murgoci. On time-inconsistent stochastic control in continuous time. *Finance and Stochastics*, 21(2):331–360, 2017.

[20] Tomas Björk. *Arbitrage Theory in Continuous Time*. Oxford University Press, 12 2019.

[21] Nicholas M. Boffi and Eric Vanden-Eijnden. Probability flow solution of the fokker–planck equation. *Machine Learning: Science and Technology*, 4, 2022.

[22] B. Bouchard and X. Warin. Monte-carlo valuation of american options: Facts and new algorithms to improve existing methods. *Numerical Methods in Fiannce*, 12:212–255, 2012.

[23] P. P. Boyle and D. Thangaraj. Volatility estimation from observed option prices. *Decisions in Economics and Finance*, 23(1):31–52, 2000.

[24] R. P. Brent. *Algorithms for minimization without derivatives*. Prentice Hall, Upper Saddle River, NJ, 1973.

[25] M. Broadie and P. Glasserman. Estimating security price derivatives using simulation. *Management Science*, 42(2):269–285, 1996.

[26] M. Broadie and P. Glasserman. Pricing American-style securities using simulation. *J. Econom. Dynam. Control*, 21:1323–1352, 1997.

[27] M. Broadie and P. Glasserman. A stochastic mesh method for pricing high-dimensional American options. *J. Comput. Finance*, 7:35–72, 2004.

[28] R. A. Carmona. *HJM: A Unified Approach to Dynamic Models for Fixed Income, Credit and Equity Markets*, pages 1–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[29] P. Carr and Madan D.B. A note on sufficient conditions for no arbitrage. *Finance Research Letters*, pages 125–130, 2005.

[30] A. Cartea and S. Jaimungal. Incorporating order-flow into optimal execution. volume 10, pages 339–364, 2016.

[31] M. Chataigner, A. Cousin, S. Crépey, M. Dixon, and D. Gueye. Short communication: Beyond surrogate modeling: Learning the local volatility via shape constraints. *SIAM Journal on Financial Mathematics*, 12(3):SC58–SC69, 2021.

[32] M. Chataigner, S. Crepey, and M. Dixon. Deep local volatility. *Risks*, 2020.

[33] D. Chen, Y. Zhu, M. Liu, and J. Li. Cost-efficient reinforcement learning for optimal trade execution on dynamic market environment. In *Proceedings of the Third ACM International Conference on AI in Finance*, ICAIF '22, page 386–393, New York, NY, USA, 2022. Association for Computing Machinery.

[34] Y. Chen and J. W. L. Wan. Deep neural network framework based on backward stochastic differential equations for pricing and hedging american options in high dimensions. *Quantitative Finance*, 21(1):45–67, 2021.

[35] Yangang Chen and Justin W. L. Wan. Numerical method for image registration model based on optimal mass transport. *Inverse Problems and Imaging*, 12:401–432, 2018.

[36] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics, Oct 2014.

[37] T. F. Coleman, Y. Li, and A. Verma. Reconstructing the unknown local volatility function. *Journal of Computational Finance*, 2(3):77–102, 2000.

[38] R. Cont and J. da Fonseca. Dynamics of implied volatility surfaces. *Quatitative Finanace*, pages 45–60, 2002.

[39] R. Cont and M. Vuletic. Simulation of arbitrage free implied volatility surfaces. 2022.

[40] A. Cousin and E. Di Bernardino. On multivariate extensions of value-at-risk. *Journal of Multivariate Analysis*, 119:32–46, Aug 2013.

[41] C. Cuchiero, W. Khosrawi, and J. Teichmann. A generative adversarial network approach to calibration of local stochastic volatility models. *Risks*, 8(4), 2020.

[42] Giannis Daras, Mauricio Delbracio, Hossein Talebi, Alexandros G. Dimakis, and Peyman Milanfar. Soft diffusion: Score matching for general corruptions. *ArXiv*, abs/2209.05442, 2022.

[43] G. Dimitroff, D. Roder, and C.P. Fries. Volatility model calbitration with convolutional neural networks. *SSRN*, 2018.

[44] D. J. Duffy. Finite difference methods in financial engineering. Chichester, 2006. John Wily & Sons, Ltd.

[45] C. Dugas, Y. Bengio, F. Elisle, and C. Nadeau. Incorporating second-order functional knowledge for better option pricing. *Cirano Working Papers*, 02 2001.

[46] R. Dunn, P. Hauser, T. Seibold, and H. Gong. Estimating option prices with Heston ' s stochastic volatility model. 2014.

[47] B. Dupire. Pricing with a smile. *Risk*, pages 18–20, 1994.

[48] V. Durrleman. Convergence of at-the-money implied volatilities to the spot volatility. *Journal of Applied Probability*, 45(2):542–550, 2008.

[49] W. E, J. Han, and A. Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in Mathematics and Statistics*, 5:349–380, Nov 2017.

[50] N. El Karoui, S. Peng, and M. C. Quenez. Backward stochastic differential equations in finance. *Mathematical Finance*, 7:1–71, Jan 1997.

[51] J. L. Elman. Finding structure in time. *Cognitive Science*, 1990.

[52] Guillaume F. and W. Schoutens. Calbration risk: Illustrating the impact of calibration risk under the Heston model. *Review of Derivatives Research*, pages 57–79, 2012.

[53] F. Fang and C. Oosterlee. A novel pricing method for European options based on Fourier-cosine series expansions. *SIAM Journal of Scientific Computing*, pages 826–848, 2009.

[54] N. P. Firth. *High Dimensional American Options*. PhD thesis, University of Oxford, 2005.

[55] P. A. Forsyth. A hamilton–jacobi–bellman approach to optimal trade execution. *Applied Numerical Mathematics*, 61(2):241–265, 2011.

[56] P. A. Forsyth. Multiperiod mean conditional value at risk asset allocation: Is it advantageous to be time consistent? *SIAM J. Fin. Math.*, 11(2):358–384, jan 2020.

[57] P. A. Forsyth and G. Labahn. Numerical methods for controlled Hamilton-Jacobi-Bellman PDEs in finance. *Journal of Computational Finance*, (11):1–44, 2007.

[58] P. A. Forsyth, Li Y., and K. R. Vetzal. Optimal stopping of markov processes: Hilbert space theory, approximation algorithms and an application to pricing high-dimensional financial derivatives. *IEEE Trans. Automat. Control*, 44:1840–1851, 1999.

[59] P. A. Forsyth, Li Y., and K. R. Vetzal. Quadratic convergence for valuing American options using a penalty method. *Siam J. Sci. Comput.*, 23:2095–2122, 2002.

[60] M. Fujii, A. Takahashi, and Takahashi M. Asymptotic expansion as prior knowledge in deep learning method for high dimensional BSDEs, 2017.

[61] J. Gatheral and A. Jacquier. Convergence of Heston to SVI. *Quantitative Finance*, 11(8):1129–1132, 2011.

[62] J. Gatheral and A. Jacquier. Arbitrage-free SVI volatility surfaces. *Quantitative Finance*, 14(1):59–71, 2014.

[63] J. Gatheral and M. Lynch. Lecture 1: Stochastic volatility and local volatility, 2001.

[64] J. Gatheral and M. Lynch. Lecture 2: Fitting the volatility skew, 2001.

[65] Jim Gatheral. The volatility surface: A practitioner's guide. 2006.

[66] Jim Gatheral. No-dynamic-arbitrage and market impact. *Quantitative Finance*, 10(7):749–759, 2010.

[67] P. Glasserman and B. Yu. Simulation for American options: Regression now or regression later? In *Monte Carlo and Quasi-Monte Carlo Methods 2002*, pages 213–226, Berlin, Heidelberg, 2004. Springer.

[68] I. Goodfellow, Y. Bengio, and A. Courville. Deep learning, adaptive computation and machine learning. 2016.

[69] G. Guan and X. Hu. Time-consistent investment and reinsurance strategies for mean–variance insurers in n-agent and mean-field games. *North American Actuarial Journal*, 26(4):537–569, 2022.

[70] B. Guler and P. P. Laignelet. Towards robust and stable deep learning algorithms for forward backward stochastic differential equations, 2019.

[71] J. Han and W. E. Deep learning approximation for stochastic control problems, 2016.

[72] M. B. Haugh and L. Kogan. Pricing American options: A duality approach. *Oper. Res.*, 52:258–270, 2004.

[73] C. He, J. S. Kennedy, P. A. Forsyth, Li Y., and K. R. Vetzal. Calibration and hedging under jump diffusion. *Review of Derivatives Research*, 9:1–35, Jan 2007.

[74] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[75] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 630–645, Cham, 2016. Springer International Publishing.

[76] D. Hendricks and D. Wilcox. A reinforcement learning extension to the almgren-chriss framework for optimal trade execution. In *2014 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr)*, pages 457–464, 2014.

[77] D. Hendrycks and K. Gimpel. Gaussian error linear units (GELUs). *arXiv: Learning*, 2016.

[78] A. Hernandez. Model calibration with neural networks. *SSRN*, 2016.

[79] C. Herrera, F. Krach, P. Ruyssen, and J. Teichmann. Optimal stopping via randomized neural networks, 2021.

[80] S. L. Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *Review of Financial Studies*, pages 327–343, 1993.

[81] A. Hirsa, T. Karatas, and A. Oskoui. Supervised deep neural networks (DNNs) for pricing/calibration of vanilla/exotic options under various different processes, 2019.

[82] Jonathan Ho, Ajay Jain, and P. Abbeel. Denoising diffusion probabilistic models. *ArXiv*, abs/2006.11239, 2020.

[83] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov 1997.

[84] B. Horvath, A. Muguruza, and M. Tomas. Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models. *Quantitative Finance*, 21(1):11–27, 2021.

[85] J. Hull. Options, futures and other derivatives. Upper Saddle River, NJ, 2003. Prentice Hall.

[86] C. Hure, H. Pham, and X. Warin. Deep backward schemes for high-dimensional nonlinear PDEs. *Mathematics of Computation*, 2020.

[87] Aapo Hyvärinen. Estimation of non-normalized statistical models by score matching. *J. Mach. Learn. Res.*, 6:695–709, dec 2005.

[88] A. Itkin. Deep learning calibration of option pricing models: some pitfalls and solutions. arXiv:1906.03507v1, 2019.

[89] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. *Artificial Neural Netowrks: Concept Learning*, page 112–127, 1990.

[90] M. Kholer, A. Kryzyzak, and Todorovic N. Pricing of high-dimensional American options by neural networks. *Math. Finance*, 20:383–410, 2001.

[91] D. P. Kingma and M. Welling. Auto-encoding variational bayes. Banff, AB, Canada, 2014. 2nd International Conference on Learning Representations, ICLR.

[92] D.P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[93] Chieh-Hsin Lai, Yuhta Takida, Naoki Murata, Toshimitsu Uesaka, Yuki Mitsufuji, and Stefano Ermon. Fp-diffusion: Improving score-based diffusion models by enforcing the underlying score fokker-planck equation. In *International Conference on Machine Learning*, 2022.

[94] R. W. Lee. *Implied Volatility: Statics, Dynamics, and Probabilistic Interpretation*, pages 241–268. Springer US, Boston, MA, 2005.

[95] D. Li and W. L. Ng. Optimal dynamic portfolio selection: Multiperiod mean-variance formulation. *Mathematical Finance*, 10(3):387–406, 2000.

[96] S. Lin and P. A. Beling. A deep reinforcement learning framework for optimal trade execution. In Yuxiao Dong, Georgiana Ifrim, Dunja Mladenić, Craig Saunders, and Sofie Van Hoecke, editors, *Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track*, pages 223–240, Cham, 2021. Springer International Publishing.

[97] S. Liu, A. Borovykh, L.A. Grzelak, and C. Oosterlee. A neural network-based framework for financial model calibration. *Journal of Mathematics in Industry*, 2019.

[98] S. Liu, C. Oosterlee, and S.M. Bohte. Pricing options and computing implied volatilities using neural networks. *Risks*, 2019.

[99] Y. Liu. Value-at-Risk model combination using artificial neural. Technical report, Emory University, 2005.

[100] F.A. Longstaff and E.S. Schwartz. Valuing American options by simulation: A simple least-squares approach. *Rev. Financ. Stud.*, 14:113–147, 2001.

[101] Cheng Lu, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps. *ArXiv*, abs/2206.00927, 2022.

[102] A. S. Na and J. W. L. Wan. Efficient pricing and hedging of high-dimensional american options using deep recurrent networks. *Quantitative Finance*, 23(4):631–651, 2023.

[103] B. Ning, F. Ho Ting Lin, and S. Jaimungal. Double deep q-learning for optimal execution. *Applied Mathematical Finance*, 28(4):361–380, 2021.

[104] Qiankun Niu. No arbitrage conditions and characters of implied volatility surface: A review for implied volatility modelers. *Capital Markets: Asset Pricing & Valuation eJournal*, 2015.

[105] Bernt Øksendal. Stochastic differential equations : an introduction with applications. *Journal of the American Statistical Association*, 82:948, 1987.

[106] Tianyu Pang, Kun Xu, Chongxuan Li, Yang Song, Stefano Ermon, and Jun Zhu. Efficient learning of generative models via finite-difference score matching. *ArXiv*, abs/2007.03317, 2020.

[107] B. A. Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, 1989.

[108] B. A. Pearlmutter. Dynamic recurrent neural networks. Technical report, Carnegie Melon University, 1990.

[109] H. Pham. Feynman-kac representation of fully nonlinear pdes and applications. *Acta Mathematica Vietnamica*, 40:255–269, 2014.

[110] F. J. Pineda. Generalization of backpropagation to recurrent and higher order neural networks. pages 602–611. International Conference on Neural Information Processing Systems, 1987.

[111] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *International Journal of Automation and Computing*, pages 503–519, 2017.

[112] P. Ramachandran, B. Zoph, and Q.V. Le. Swish: A self-gate activation function, 2017.

[113] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. cite arxiv:1505.04597Comment: conditionally accepted at MICCAI 2015.

[114] M. Roper. Arbitrage free implied volatility surfaces, 2010.

[115] Yousef Saad. Iterative methods for sparse linear systems. 2003.

[116] Be. Salvador, C. Oosterlee, and R. Van der Meer. Financial option valuation by unsupervised learning with artificial neural networks, 2019.

[117] Timur Sattarov, Marco Schreyer, and Damian Borth. Findiff: Diffusion models for financial tabular data generation. ICAIF '23, page 64–72, New York, NY, USA, 2023. Association for Computing Machinery.

[118] A. M. Schäfer and H. G. Zimmermann. Recurrent neural networks are universal approximators. In Stefanos D. Kollias, Andreas Stafylopatis, Włodzisław Duch, and Erkki Oja, editors, *Artificial Neural Networks – ICANN 2006*, pages 632–640, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[119] T. Schoneborn. *Trade execution in illiquid markets Optimal stochastic control and multi-agent equilibria*. PhD thesis, Technical University of Berlin, 2008.

[120] T. Sidogi, W. T. Mongwe, R. Mbuvha, and T. Marwala. Creating synthetic volatility surfaces using generative adversarial networks with static arbitrage loss conditions. In *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1423–1429, 2022.

[121] J. Sirignano and K. Spiliopoulos. DGM: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, Dec 2018.

[122] Jascha Narain Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. *ArXiv*, abs/1503.03585, 2015.

[123] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *ArXiv*, abs/2010.02502, 2020.

[124] Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. In *Neural Information Processing Systems*, 2019.

[125] Yang Song and Stefano Ermon. Improved techniques for training score-based generative models. *ArXiv*, abs/2006.09011, 2020.

[126] Yang Song, Jascha Narain Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *ArXiv*, abs/2011.13456, 2020.

[127] Yang Song, Jascha Narain Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *ArXiv*, abs/2011.13456, 2020.

[128] J.D. Spiegeleer, D. Madan, S. Reyner, and W. Schoutens. Machine learning for quantitative finance: fast derivative pricing, hedging and fitting. *Quantitative Finance*, 18(10):1635–1643, 2018.

[129] H. Thom. *Longstaff Schwartz Pricing of Bermudan Options and their Greeks*. PhD thesis, University of Oxford, 2009.

[130] S. Tse. *Numerical Methods for Optimal Trade Execution*. PhD thesis, David R. Cheriton School of Computer Science, University of Waterloo, 2012.

[131] P. M. Van Staden, D. Dang, and P. A. Forsyth. Time-consistent mean–variance portfolio optimization: A numerical impulse control approach. *Insurance: Mathematics and Economics*, 83:9–28, 2018.

[132] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[133] E. Vigna. On time consistency for mean-variance portfolio selection. *International Journal of Theoretical and Applied Finance*, 23(06):2050042, 2020.

[134] Pascal Vincent. A connection between score matching and denoising autoencoders. *Neural Computation*, 23(7):1661–1674, 2011.

[135] M. Vuletic, M. Cucuringu, and F. Prenzel. Fin-gan: Forcasting and classifying financial time series via generative adversarial networks. 2023.

[136] H. Wang and X. Y. Zhou. Continuous-time mean–variance portfolio selection: A reinforcement learning framework. *Mathematical Finance*, 30(4):1273–1308, 2020.

[137] J. Wang and P.A. Forsyth. Continuous time mean variance asset allocation: A time-consistent strategy. *European Journal of Operational Research*, 209(2):184–201, 2011.

[138] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[139] Sebastian Jaimungal Xuancheng Huang and Mojtaba Nourian. Mean-field game strategies for optimal execution. *Applied Mathematical Finance*, 26(2):153–185, 2019.

[140] X.Y. Zhou and D. Li. Continuous-time mean-variance portfolio selection: A stochastic lq framework. *Applied Mathematics and Optimization*, 42(1):19–33, 2000.

APPENDICES

Appendix A

A.1 Additional Details and Results for Chapter 2

A.1.1 Time and memory cost table

We present the full time and memory comparison between methods in the following table:

A.1.2 Feature selection and network summary

In this section we discuss the features used in Chapter 2 and summarize the GRU as an algorithm. Feature selection in neural network has a significant impact on the accuracy of the network model [68], the correct choice of inputs in any regression problem allows for good predictions and approximations. We use the asset price, and $g(\mathbf{s})$ from (2.2). [34] found that $g(\mathbf{s})$ performs better than using the payoff, $f(\mathbf{s})$ due to the additional information it carries. We concatenate the inputs, but we do not perform normalization. Our concatenated input matrix is

$$\mathbf{x} = [s, h(s)]^\top \in \mathbb{R}^{M\times 2d}.$$

Unlike a traditional feedforward network, the Recurrent network uses the hidden state to carry information through time. In practical applications of Recurrent networks the hidden state is initialized as zero or given some random numbers following a uniform or normal distribution. In our approach, we initialize the hidden state using g^N for $\mathcal{F}(\mathbf{s};\Omega)$ and $\nabla_S g^N$ for $\mathcal{G}(\mathbf{s};\Theta)$. We pad the initial hidden state with itself to match the number of GRU layers.

Table A.1 Time and memory comparison between methods. The American style geometric average call option with $d = 2, 5$, and 10, priced using $M = 50000$. We used a Recurrent layer size of $L = 7$ for the proposed method. We used a timestep skip, $J = 4$ and $L = 7$ for the DRL method.

(a) $d = 2$: Proposed method

N	Total Time (s)	Training Time (s)	Memory (MB)	Price ($)
50	39.48	29.2	5.62	6.6601
100	76.35	56.73	5.63	6.4266
1000	516.07	379.94	5.63	6.5961

(b) $d = 2$: DRL method

N	Total Time (s)	Training Time (s)	Memory (MB)	Price ($)
50	113.75	113.62	3701.59	6.6702
100	316.3	316.06	7705.53	6.4817
1000	25590.97	25586.84	151944.60	6.5822

(c) $d = 5$: Proposed method

N	Total Time (s)	Training Time (s)	Memory (MB)	Price ($)
50	39.32	28.91	5.62	5.9069
100	78.82	57.77	5.63	5.9524
1000	812.96	577.02	5.63	5.8701

(d) $d = 5$: DRL method

N	Total Time (s)	Training Time (s)	Memory (MB)	Price ($)
50	134.8	134.64	4548.33	5.9218
100	410.92	410.62	9408.30	5.9518
1000	36739.60	36734.08	168054.00	5.8697

(e) $d = 10$: Proposed method

N	Total Time (s)	Training Time (s)	Memory (MB)	Price ($)
50	38.55	28.15	5.62	5.9883
100	76.44	56.05	5.63	5.9087
1000	785.49	552.97	5.63	5.7964

(f) $d = 10$: DRL method

N	Total Time (s)	Training Time (s)	Memory (MB)	Price ($)
50	251.40	251.02	6847.64	5.9791
100	726.46	725.86	14124.60	5.9053
1000	65499.84	65491.55	215701.40	5.7364

Algorithm 5 Continuation price network using a single GRU unit

Require: $\mathbf{h}^0, \omega_e^n, \omega_o^n, g^N$, GRU
  for $n = N-1, ..., 0$ **do**
    $\mathbf{e}^n \leftarrow \text{swish}(\omega_e^{n\top}\text{GRU})$
    $\mathbf{o}_p^n \leftarrow \text{softplus}(\omega_o^\top \mathbf{e}^n)$
    $\mathcal{F}(S;\Omega) \leftarrow \mathbf{o}_p^n$
    $u_\omega^n \leftarrow (1-\alpha)e^{-r(N-n)\Delta t} g^N + \alpha\mathcal{F}(S;\Omega)$

We can express our price approximating network for $n = N-1, ..., 0$ by Algorithm 5 and similarly we approximate the delta of the option we use Algorithm 6.

Algorithm 6 Delta network using a single GRU unit

Require: $\mathbf{h}_\Delta^0, \theta_e^n, \theta_o^n, \nabla_S g^N$, GRU
  for $n = N-1, ..., 0$ **do**
    $\mathbf{e}^n \leftarrow \text{swish}(\theta_e^{n\top}\text{GRU})$
    $\mathbf{o}_\Delta^n \leftarrow \hat{\sigma}(\theta_o^\top \mathbf{e}^n)$
    $\mathcal{G}(s;\Theta) \leftarrow \mathbf{o}_\Delta^n$
    $\nabla_S u_\theta^n \leftarrow (1-\beta)e^{-r(N-n)\Delta t}\nabla_S g^N \frac{S^N}{S^n} + \beta\mathcal{G}(S;\Theta)$

A.2 Additional Details and Results for Chapter 3

A.2.1 Almgren-Chriss model

The dynamics of the optimal trade execution problem is modeled by the Almgren-Chriss model [7]. The instantaneous trade rate is modeled by:

$$v(t) = \frac{d\alpha(t)}{dt},$$

where $\alpha(t)$ is the amount of stock held by a liquidating agent. Let μ be the drift, r be the risk free rate and σ be the volatility of the asset. The price process of the risky asset follows the Geometric Brownian motion (GBM) and is given by [55]:

$$dS(t) = (\mu - r + g(v(t)))S(t)dt + \sigma S(t)dW(t),$$

where $g(v(t))$ is the level of permanent price impact. The cash position is then given by [55]:

$$\frac{db(t)}{dt} = rb(t) - v(t)s(t)f(v(t)),$$

where $f(v(t))$ is the level of temporary price impact. We can define the temporary price change due to the temporary impact as $S_{temp}(t,v) = v(t)s(t)f(v(t))$, $S_{temp}(v,t) \leq S(t)$. Given states $\{S(T^-), B(T^-), \alpha(T^-)\}$ at the instant $t = T^-$ before the end of the trading horizon we have one final liquidation (if $\alpha(T^-) \neq 0$) to ensure that the number of shares $\alpha(T) = 0$. The liquidation value is given by[130]:

$$\begin{aligned} B(T) &= B(T^-) + \lim_{v \to -\infty} \{\alpha(T^-) S_{temp}(T^-, v)\} \\ &= \int_{t=0}^{T^-} -e^{r(T-t)} v(t) S_{temp}(t,v) dt + \lim_{v \to -\infty} \{\alpha(T^-) S_{temp}(T^-, v)\}. \end{aligned}$$

A.2.2 Generating data for optimal trade execution

To construct the training data used in Chapter 3 we need to perform Monte-Carlo simulations for the dynamics of the underlying asset $(S_i)_m^n$. The cash position $(b_k)_m^n$, position in stocks $(\alpha_{i,k})_m^n$ and the control $(v_{i,k})_m^n$ are simulated forward in time by discretizing their governing dynamics. At $n = 0$ we initialize the variables according the initial conditions given by (3.2) for the variables $\{(S_i)_m^0, (b_k)_m^0, (\alpha_{i,k})_m^0\}$. The trade rate $(v_{i,k})_m^n$ is constructed as a uniform grid of discretized points over $c = 1, ..., C$ points. The trade rate is given by:

$$v_{i,k} = \{v_1/T, ..., v_C/T\},$$

where $v_1 = v_{min}$ and $v_c = v_{max}$. For $n = 1, ..., N$, the dynamics of (2.9) is simulated using Euler's method. The dynamics of $\{(b_k)_m^n, (\alpha_{i,k})_m^n\}$ follow the dynamics given by (3.17) and (3.18). To ensure that all assets are liquidated at time T we define the trade rate the instant before maturity denoted by v_T which is given by [55]:

$$v_T = \frac{-(\alpha_{i,k})_m^N}{dT}, \text{ where } dT \ll dt.$$

The cash position is updated one final time at $t = T - dT$, this is done to penalize all positions that have not been liquidated at the final trading step. This is done at the end of the forward simulation in order to compute the value function at terminal time $t = T$.

A.2.3 Experiment: performance of multiple sellers and buyers in different economies

In this experiment we compare the investment performance between sellers (long) and buyers (short) in different economic conditions. To compare different groups of agents we compute the average Sharpe ratio of the selling agents and the buying agents. The Sharpe ratio provides an insight into the relative performance of the portfolio return. A positive Sharpe ratio is considered an investment strategy that outperforms the risk free rate. A negative Sharpe ratio indicates the liquidating strategy , i.e., selling and buying, have lower return than holding the asset. In our problem a strategy with Sharpe ratio > 0 is acceptable. We divide the 10 agents into buyers and sellers and observe the performance of each group of liquidating agents and compare them to a strategy where they just hold the asset from $[0, T]$.

For each agent $k = 1, ..., K$ the return on liquidating the position is given by $R_k = (X_k(T) - X_k(0))/X_k(0)$ and the standard deviation on the return is $\sigma_{R_k} = \sqrt{Var(R_k)}$. We want to compare the strategy with the strategy of holding the asset from $t = 0$ to T. The no-trade return is given by $R_{0,k} = (\sum_i \alpha_{0,i} S_i(T) - X_k(0))/X_k(0)$ then the Sharpe ratio is given by:

$$SR_k = \frac{\mathbb{E}[R_k] - \mathbb{E}[R_{0,k}]}{\sigma_{R_k}}.$$

Note, we compare the portfolio with the expected value of the asset instead of the risk free rate of return. This compares the performance of different strategies versus holding the asset without liquidating. If the number of long agents is given by K_{long} and the number of short agents is given by K_{short}. Then we can compute

$$\sum_{k \in K_{long}} SR_k/K_{long} \text{ and } \sum_{k \in K_{short}} SR_k/K_{short}$$

to measure the average performance of the long agents (buyers) and short agents (sellers) respectively.

We simulate $K = 10$ agents and $d = 2$ assets with $\rho = 0$. We do not consider correlation in this example to ensure the results are easier to understand. In this experiment we set $r = 0.05$ for $i = 1, ..., 10$, $\kappa_\tau = 1e-7$, $\kappa_p = 0.000125$, $\gamma = 2$ and fix $\phi = 0.2$. By increasing the number of agents beyond $K > 2$ we can study more interesting dynamics between agents. In case 1, we let 7 agents be selling agents and 3 agents be buying agents. Selling agents are agents liquidating long positions and buying agents are agents liquidating short positions. In case 2, we look at the 5 selling and 5 buying agents. In case 3, we look at

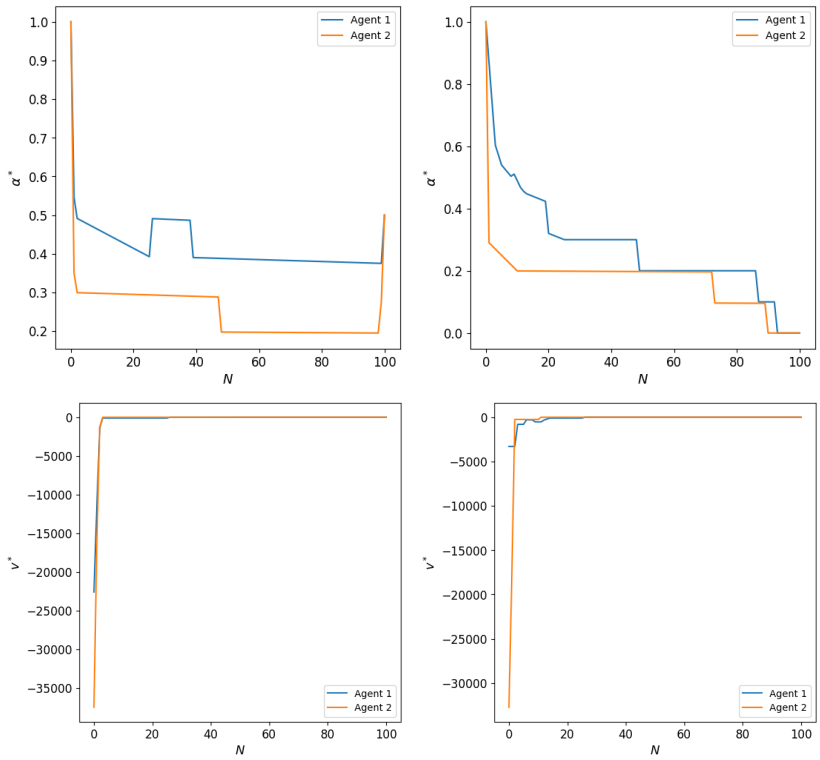


Figure A.1 The optimal portfolio (top) and trade rate (bottom) for Agents 1 (blue) and 2 (orange). For a good economy (right) and bad economy (left) when $\phi = 0.7$.

the case where we have 3 selling agents and 7 buying agents. For all three cases we look at different market conditions:

1. good market conditions ($\mu_1 - r > 0, \mu_2 - r > 0$),
2. poor market conditions ($\mu_1 - r < 0, \mu_2 - r < 0$),
3. mixed market conditions ($\mu_1 - r > 0, \mu_2 - r < 0$).

Case 1*: 7 selling agents, 3 buying agents*

In case 1, we look at the average Sharpe ratio of 7 selling agents and 3 buying agents in different market conditions. The average Sharpe ratio for sellers and buyers are summarized in Table A.2. We observe that under market conditions 2) and 3) it is advantageous for selling and buying agents to liquidate. The best Sharpe ratio we achieve is 0.35014 for selling agents under 2). This is because the agents liquidate the short positions gradually and the long positions quickly. There is an overall negative price impact due to the asymmetry in the number of sellers and buyers which leads to lower asset prices. Under market conditions 1), it is disadvantageous to liquidate short positions over holding the asset. This occurs from the overall negative price impact that decreases the amount of growth for both assets. Due to the imbalance of sellers and buyers we observe the selling agent always performing better than the holding strategy.

Market Condition	$\sum_{k \in K_{long}} SR_k / K_{long}$	$\sum_{k \in K_{short}} SR_k / K_{short}$
$\mu_1 - r > 0, \mu_2 - r > 0$	0.01456	-0.02949
$\mu_1 - r < 0, \mu_2 - r < 0$	0.35014	0.14856
$\mu_1 - r > 0, \mu_2 - r < 0$	0.06993	0.04475

Table A.2 Average Sharpe ratio of sellers and buyers in good market conditions, poor market conditions and mixed market conditions.

Case 2*: 5 selling agents, 5 buying agents*

In case 2, we look at the average Sharpe ratio of 5 selling agents and 5 buying agents in different market conditions. The average Sharpe ratio for sellers and buyers are summarized in Table A.3.

Market Condition	$\sum_{k\in K_{long}} SR_k/K_{long}$	$\sum_{k\in K_{short}} SR_k/K_{short}$
$\mu_1 - r > 0, \mu_2 - r > 0$	0.00909	-0.03696
$\mu_1 - r < 0, \mu_2 - r < 0$	0.15290	0.06123
$\mu_1 - r > 0, \mu_2 - r < 0$	0.1814	0.1080

Table A.3 Average Sharpe ratio of sellers and buyers in good market conditions, poor market conditions and mixed market conditions.

We observe that under market conditions 2) and 3) it is advantageous for long and short agents to liquidate. The best Sharpe ratio we achieve is 0.1814 for sellers under 3). This is because agents sell (buy) asset 1 more gradually (quickly) and asset 2 more quickly (gradually). Because $\alpha_{1,k} > \alpha_{2,k}$, the agents gain more from the increase in asset 1 price than the decrease in asset 2 price. This results in an advantage for gradual sellers and aggressive buyers. Under 1) sellers have an advantage since they sell off gradually and the price of both assets are expected to increase. Buyers are disadvantaged as the expected value of the asset increases under 1), even if they liquidate quickly.

Case 3*:* 3 *selling agents,* 7 *buying agents*

In case 3 we look at the average Sharpe ratio of 3 selling agents and 7 buying agents in market conditions $1), 2)$, and 3). The average Sharpe ratio for sellers and buyers are summarized in Table A.4. In Table A.4, we observe that under market conditions 2) it is advantageous for sellers and buyers to liquidate. The best Sharpe ratio we achieve is 0.3113 for selling agents under 2). There is an overall positive price impact since the number of buyers is greater than the number of sellers. We see that sellers and buyers generally do not out perform the hold strategy. This is due to the overall positive price impact adding to the drift term of both assets pushing up the asset price.

Market Condition	$\sum_{k\in K_{long}} SR_k/K_{long}$	$\sum_{k\in K_{short}} SR_k/K_{short}$
$\mu_1 - r > 0, \mu_2 - r > 0$	-0.06766	-0.14143
$\mu_1 - r < 0, \mu_2 - r < 0$	0.3113	0.1212
$\mu_1 - r > 0, \mu_2 - r < 0$	-0.02911	-0.06120

Table A.4 Average Sharpe ratio of sellers and buyers in good market conditions, poor market conditions and mixed market conditions.

A.3 Additional Details and Results for Chapter 4

A.3.1 Heston model

In Chapter 4, we model market asset prices using the Heston model. It is well known that the Heston model cannot replicate market prices for short expiries when parameters are time-consistent [64]. The Heston model was chosen because it has a closed form solution for European options [80]. In standard Black-Scholes model, it assumes the asset price follows a geometric Brownian motion (GBM) which leads to the well known Black-Scholes equation for the no-arbitrage option price. However, it assumes the volatility is constant across maturities and strikes which may not hold in practice.

We use the Heston model to price the European call option which is used as synthetic data for our model training. We denote the call option price as V. The Heston model has the nice property that the characteristic function can be derived analytically which can be used in fast efficient solvers. The asset price that follows the Heston model is given as the pair of SDEs [80]

$$\begin{aligned} dS_t &= rS_t dt + \sqrt{v_t} S_t dW_t^S \\ dv_t &= \kappa(\bar{v} - v_0)dt + \gamma\sqrt{v_t} dW_t^v \\ dW_t^S dW_t^v &= \rho dt, \end{aligned}$$

where S_t is the stock price at time t, r is the risk-free interest rate, κ is the mean reversion rate, ρ is the correlation between the stock price process and the variance process, W_t^S is the Wiener process driving the stock price dynamics, W_t^v is the Wiener process driving the variance process, γ is the volatility of the variance, $\bar{v}$ is the long-run average of the variance, v_t is the variance at time t and v_0 is the initial variance.

We used the cosine (COS) method of [53] to compute the European option price using the characteristic function of the Heston model. More precisely, let $i = \sqrt{-1}$, $\psi \in \mathbb{R}$ and let $\tau = T - t$ for any time $t \in [0, T]$. The variables $\{\kappa, \rho, \gamma, v_0, \bar{v}\}$ are the Heston parameters. The explicit form of the characteristic function of the Heston model given by [46]

$$f(i\psi) = e^{A(\tau)+B(\tau)v_t+i\psi S_t}$$

where

$$
\begin{aligned}
A(\tau) &= ri\psi\tau + \frac{\kappa\bar{v}}{\gamma^2}\left(-(\rho\gamma i\psi - \kappa - M)\tau - 2\ln\left(\frac{1-Ne^{M\tau}}{1-N}\right)\right) \\
B(\tau) &= \frac{(e^{M\tau}-1)(\rho\gamma i\psi - \kappa - M)}{\gamma^2(1-e^{M\tau})} \\
M &= \sqrt{(\rho\gamma i\psi + \kappa)^2 + \gamma^2(i\psi + \psi^2)} \\
N &= \frac{\rho\gamma i\psi - \kappa - M}{\rho\gamma i\psi - \kappa + M}.
\end{aligned}
$$

A.3.2 Surface stochastic volatility inspired (SSVI) model

Discretization can be a convenient method to compute local volatility, however, it has an issue with the regularity and availability of market data. To overcome the issue of limited market price data practitioners look at the parametric approximation given by the surface stochastic volatility inspired (SSVI) method [61, 62].

Definition A.3.1. Let $\theta_T = \sigma_{implied}(s_0, T)^2 T$ and let ϕ be a smooth function from $\mathbb{R}_+ \mapsto \mathbb{R}_+$ such that the limit of $\theta_T\phi(\theta_T)$ exists as $T \to 0$. Then the SSVI is the surface defined by [62]

$$
w(k, \theta_T) = \frac{\theta_T}{2}\left(1 + \rho\phi(\theta_T)k + \sqrt{(\phi(\theta_T) + \rho)^2 + (1-\rho^2)}\right). \tag{A.1}
$$

Note the parameter ρ is the same as the Heston parameter ρ. To solve for the local volatility surface we let $\lambda > 0$ and use the Heston-like parametric function given by [62]

$$
\phi(\theta) = \frac{1}{\lambda\theta}\left(1 - \frac{1-e^{-\lambda\theta}}{\lambda\theta}\right). \tag{A.2}
$$

A.3.3 No-arbitrage conditions imposed on option price surface

Let $V(K,T) : [0,\infty) \times [0,\infty) \to \mathbb{R}$ be the call price surface. For $V(K,T)$ to be arbitrage free it must have the following properties [29]:

$$
\frac{\partial V}{\partial T} > 0, \frac{\partial V}{\partial K} < 0, \frac{\partial^2 V}{\partial K^2} > 0. \tag{A.3}
$$

These conditions were formalized and summarized by the following Lemma (cite roper):

Lemma A.3.2. *Let $s > 0$ be a constant. Let $V(K,T)$ be a call price surface and satisfy the following conditions, there exists a non-negative Markov process X_t, $t = [0,T]$ such that $V(K,T) = \mathbb{E}[\max\{X_T - K, 0\}; X_0 = s]$.*

1. *(Convexity in K): $V(\cdot,T)$ is a convex function $\forall T > 0$;*
2. *(Monotonicity in T): $V(K,\cdot)$ is non-decreasing $\forall K > 0$;*
3. *(Large strike limit): $\lim_{K\to\infty} V(K,T) = 0,\ \forall T > 0$;*
4. *(Bounds): $\max\{s - K, 0\} \leq V(K,T) \leq s,\ \forall K > 0, T \geq 0$; and*
5. *(Expiry value): $V(K,0) = \max\{s - K, 0\},\ \forall K > 0$.*

All of the listed conditions of this theorem are necessary properties of $V(K,T)$ for it to be the conditional expectation of a call option under the assumption that X_t is a (non-negative) martingale.

Proof. The readers are referred to [114] for proof of this theorem. □

The first constraint of (A.3) ensures that calendar arbitrage (monotonicity) is not violated. The second and the third ensures that the butterfly arbitrage is not violated.

The constraints(A.3) are applied to the option price surface to ensure that the option price surface is arbitrage-free [29]. Neural networks are very difficult to train with hard constraints and tend to result in extremely large prediction errors [2]. [88] treats the constraints as soft constraints and derives penalty functions to limit arbitrage violations. For given constants $\delta_1 > 0, \delta_2 > 0, \delta_3 > 0$ and the approximate option price, $\hat{V}$, the regularization terms are given by [88]:

$$L_1 = \delta_1 \max\{-T^2 \frac{\partial \hat{V}}{\partial T}, 0\}, L_2 = \delta_2 \max\{-K^2 \frac{\partial^2 \hat{V}}{\partial K^2}, 0\}, L_3 = \delta_3 \max\{K \frac{\partial \hat{V}}{\partial K}, 0\}. \tag{A.4}$$

Another approach to ensure no-arbitrage via regularization is by treating the Dupire condition [47] as a soft constraint [32]. Let $k = \log(K/s_0)$, $\partial_k = \frac{\partial}{\partial k}$ and $\partial_{kk} = \frac{\partial^2}{\partial k^2}$ then the Dupire penalty is given by:

$$L_{dup} = \frac{\partial_T V}{k^2 \partial_{kk} V}.$$

It is clear that the Dupire penalty requires the conditions $\partial_T V \geq 0$ and $\partial_{kk} V > 0$. This may not hold true if relying on auto-differentiation. However, this may be remedied by

using a similar treatment on the put option surface as (A.4). We remark that for the DLV method the local volatility is computed as $\sigma_{local}(k,T) = (L_{dup})^{1/2}$. Note that the soft constraints can not guarantee that the calibrated implied volatility will be arbitrage free as it is not guaranteed to completely eliminate arbitrage in the option price surface.