Comparison Between Different Numerical Methods in the Applications of Option Pricing

by

Lidan Chen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Jun Liu

Waterloo, Ontario, Canada, 2020

© Lidan Chen 2020

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

Driven by the increasing popularity of neural networks, we implement a new neural network-based method Physics-Informed Neural Networks in the option pricing model of financial markets. We consider another four traditional methods: the binomial tree, Monte Carlo, finite difference, along with artificial neural network as references, and show the pros and cons of these different methods in different scenarios.

Acknowledgements

I would express gratitude for my supervisor, Dr. Jun Liu, for his guidance and support.

Also, I would like to thank Dr. Xinzhi Liu, who spent precious time reading this paper as a second reader.

Great thanks go as well to CM group, Hybrid Systems Lab, especially Mengyao Zhang, Yan Liu and Zhibing Sun, all of whom contributed to the research.

I would also like to thank Dr. Hans De Sterck and Dr. Jeff Orchard again for serving as my committee members.

Dedication

This is dedicated to my family.

Table of Contents

1 Introduction **1**

2 Option Pricing Theory **3**

2.1 Introduction to Options 3

2.1.1 European Options 4

2.1.2 American Options 4

2.2 Option Pricing Theory 5

2.2.1 The Black-Scholes-Merton model and its connection to PDEs 5

3 Numerical Methods **9**

3.1 Binomial Tree 9

3.2 Monte-Carlo Method 11

3.3 Finite Difference Methods 12

3.4 Artificial Neural Networks 13

3.5 PINN 17

4 Implementation Results and Discussions **20**

4.1 Binomial Tree 20

4.2 Monte-Carlo Method 22

4.3 Finite Difference Methods 27

4.4 Artificial Neural Network 29

4.5 PINN with DeepXDE 32

5 Conclusion and Future Outlooks **41**

Chapter 1

Introduction

Financial derivatives are widely used in today's market. In fact, the markets for options, futures, forwards, swaps and other types of derivatives are much bigger than the original market for their underlying assets. Options are one of the most popular and well-studied derivatives. They are extremely volatile, and thus it is important to accurately price them as fast as possible. Hence, we choose to study option pricing. In this paper, we not only use traditional methods to solve option pricing models, such as the binomial tree, Monte Carlo, finite difference (FD) and artificial neural networks (ANNs) but also use the Physics-Informed Neural Networks (PINN) method based on DeepXDE: a deep learning library for solving different equations [1].

A brief introduction to options is given in Chapter 2. This chapter includes basic concepts, such as the arbitrage-free assumption, and a description of the geometric Brownian motion that the underlying asset follows. Based on these assumptions, the famous Black-Scholes-Merton model is introduced. Furthermore, by introducing the Feynman-Kac theorem, the relationship between the pricing model and its corresponding partial differential equation (PDE) is described. Explicit solutions of the Black-Scholes-Merton equation to certain options, as well as the corresponding PDE, will be used as a reference for the following chapters.

In Chapter 3, we briefly introduce the theoretical foundations of various computational tools used for option pricing, including traditional methods such as the binomial tree, Monte-Carlo simulation and FD. Two machine learning methods are also introduced: using ANNs to directly approximate the price of the option, and using PINN to solve the corresponding PDE.

In Chapter 4, we implement the above numerical methods and have a short discussion at the end of each section. We also discuss how to reduce the variance of large volatility and strike price in the Monte-Carlo methods, and discuss the results of the explicit, implicit and Crank-Nicholson scheme in FD setting. We use the true price of Black-Scholes as a reference to approximate the price of a European call option using ANNs. Finally, we implement the PINN scheme with the DeepXDE library and discuss the impact of the number of training points, the neural network (NN) depth, the NN width and the learning rate on the model. In addition, we also discuss some extra

features in this scheme.

In Chapter 5, we briefly summarize the results and discuss some possibilities for future research.

Chapter 2

Option Pricing Theory

In this chapter, we introduce some basic financial concepts and formulate and discuss the main assumptions behind the standard option pricing theory. Most importantly, we will introduce the Black-Scholes-Merton model on the basis of these assumptions, and its corresponding PDE by introducing the Feynman-Kac theorem.

2.1 Introduction to Options

An option in finance is a contract that gives the buyer the right, but not the obligation, to buy or sell an underlying asset at a specified strike price before or on a specified date [2]. The strike price itself or the rule of calculating it is usually set beforehand. It might be set by reference to the market price of the underlying asset, or fixed to be a certain amount, or a discount at a premium, etc. When the buyer chooses to buy/sell his corresponding option, it is called “exercise”. An option that gives the right to buy the underlying asset is called a “call”, and an option that gives the right to sell the underlying asset is called a “put”. In the next section, we would discuss the deterministic relationship between European call and put options. In most cases, it suffices to discuss one of them.

There are many rules about when and how the options should be exercised. One of the most common ones is by time. A European option is an option that can only be exercised on a preset specific date in the future, while the holder of an American option can exercise his option at any time before that date. The date on which the option is exercised is called the expiry date. The mentioned option types are called “vanilla” options as they are highly standardized and well studied.

2.1.1 European Options

The value of the option expiry date, as a function of the value of the underlying asset, is called the payoff function. As discussed above, the holder of a European option has the right to exercise the option at the expiry date [3], and the payoff he gets depends on the price of the underlying asset at this date. For example, for a call option, if the price of the asset is greater than the strike price, the holder will exercise; otherwise, he will not exercise. Let us denote the price of the underlying asset at expiration by X_T, and the strike price by K. Then, the payoff function of the long position (buying) a European call option can be written as:

$$\max(X_T - K, 0).$$

Similarly, the payoff of the long position of a put option can be written as $\max(K - X_T, 0)$. If we were at the short position (sell) of a call option, the payoff can be written as $\min(K - X_T, 0)$, and the payoff when we short a put is $\min(X_T - K, 0)$ [3]. Figure 2.1 shows the payoff diagrams:

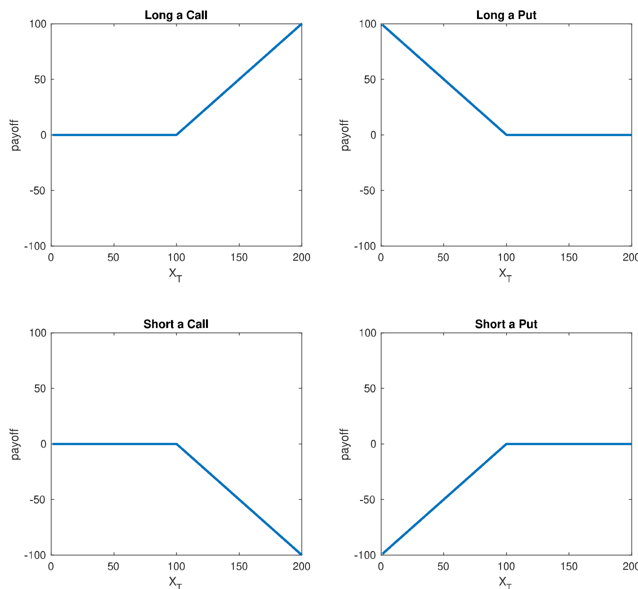


Figure 2.1: Payoff functions with $K = 100$ for European options

2.1.2 American Options

An option can be named as American option if it can be exercised at any time from now to the expiry date [4]. When there exists an opportunity to exercise at any time, things get complicated. The payoff function now, in contrast to the European option, cannot depend solely on the price of the asset at maturity. In practice, the most advantageous strategies for options with convex payoff functions, like stock call options that do not pay dividends, are exercised at expiry. In other

cases, most options, covering put options, also have an optimal exercise strategy, which, however, cannot be simply or easily calculated. In addition, in the case of the American call option, when the maturity is limited, there is an analytic formula for the price. However, it does not exist for the corresponding put option.

2.2 Option Pricing Theory

There are two assumptions when pricing an option. The first one assumes that in the market, bonds and stocks can be used to replicate each return payoff structure of the contract. The second one is the arbitrage-free principle in the risk-free measure.

Moreover, suppose that there are only two types of assets in the market: the risk-free assets B, and the risk assets S. Let us define a trading strategy (x, ϕ), which represents that we purchase x unit of the risk-free asset and ϕ units of the risk asset at time 0. Let $V_t(x, \phi)$ represent the value of the trading strategy at time t. Throughout the process, we keep this ratio constant. Let us denote the above market as (B, S).

Definition 2.2.1. *A trading strategy (x, ϕ) in the market (B, S) is said to be an arbitrage opportunity (or arbitrage) if*

- $V_0(x, \phi) = 0$ *(no initial investment),*
- $V_1(x, \phi) \geq 0$ *(no risk of loss),*
- $\mathbb{P}(V_1(x, \phi) > 0) > 0$.

Note that this assumption provides the foundation of option pricing. Based on the above assumptions, we can discuss a widely used model for the pricing and hedging of European contingent claims - the Black-Scholes-Merton model.

2.2.1 The Black-Scholes-Merton model and its connection to PDEs

Assume that the price of the cash bond at time t, B_t, satisfies the following differential equation

$$dB_t = rB_t dt,$$

where $r > 0$ is the risk-free interest rate. Then, if we assume $B_0 = 1$, we will get the unique solution

$$B_t = e^{rt}.$$

Now suppose that the price of the risk asset at time t is S_t. Then, S_t follows a stochastic differential equation of the form

$$dS_t = \mu S_t dt + \sigma S_t dW_t,$$

where S_0 is the initial price, $\mu > 0$ and $\sigma > 0$ represent the drift and volatility, respectively, and W_t is a standard Brownian motion.

Proposition 2.2.1. *The above equation has a unique solution with given by*

$$S_t = S_0 \exp((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t).$$

Proof. Apply Ito's formula and use the ansatz

$$Y_t := \log S_t.$$

We can easily get the desired expression. □

Remark 2.2.1. *In a risk-free market, $r = \mu$. This means that, under the risk-neutral measure, all the assets grow with the risk-free interest rates.*

Obviously, the European call option for asset stocks with strike K and expiry date T is a contract allowing owners to buy the stock share at the price of K at time T. So, the value of the call at time T is $\max(S_T - K, 0)$. Therefore, the price of the European call option at time 0 should be the discounted expectation, under the risk-neutral measure, of the value at time $t = T$, which is

$$\mathbb{E}[e^{-rT} \max(S_T - K, 0)].$$

Similarly, the price of the European put option at the time 0 is

$$\mathbb{E}[e^{-rT} \max(K - S_T, 0)].$$

Note that this provides the basic idea of the Monte-Carlo method that will be discussed in the next chapter.

Theorem 2.2.1. *(The Black–Scholes Formula [5]) Assume that the risk-free interest rate is r, and the underlying asset S_t has volatility $\sigma > 0$ with current price S_0. Then the price of a call option with strike K and maturity T can be written in the following form:*

$$c = S_0 \mathcal{N}(d_1) - Ke^{-rT}\mathcal{N}(d_2),$$

and the price of a put option with the same strike and maturity can be written as

$$p = Ke^{-rT}\mathcal{N}(-d_2) - S_0\mathcal{N}(-d_1),$$

where

$$d_1 = \frac{\ln(\frac{S_0}{K}) + (r + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}},$$
$$d_2 = \frac{\ln(\frac{S_0}{K}) + (r - \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T},$$

and $\mathcal{N}(\cdot)$ is the cumulative probability distribution function for the standardized normal distribution.

Proof (sketch). We will not perform a complete calculation of the proof. The idea is that the standard Brownian motion W_t has independent increments and it is normally distributed with 0 mean and variance t. This implies that the price of the asset price S_t is a log-normally distributed random variable. Some standard argument in probability yields the result. □

It is quite straightforward to derive a closed-form solution for the simple European call and options. However, these cases are too simple. There are numerous different types of contracts on the market. For example, the Asian options, the barrier options and the Bermuda options, etc. The following two theorems provide a general deterministic way of solving such problems.

Theorem 2.2.2. *(Feymann-Kac Theorem with Discounting [6]) Consider the stochastic differential equation*

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t,$$

and let $h(x)$ be a function and $r \geq 0$ be a constant. Fix $T > 0$ and let $t \in [0, T]$. Define the function

$$v(t, x) = \mathbb{E}[e^{-r(T-t)}h(X_T)|X_t = x].$$

Then, v satisfies the following partial differential equation:

$$v_t(t, x) + \mu(t, x)v_x(t, x) + \frac{1}{2}\sigma^2(t, x)v_{xx}(t, x) = 0,$$
$$v(T, x) = h(x).$$

for all x.

Proof (sketch). The idea of the proof is the following: Conditional expectations are martingales by the tower property of conditional expectations. Suppose $r = 0$. Then $\mathbb{E}[h(X_T)|X_t = x]$ is the price of any derivative security, where X is the underlying asset and h is the intermediate cash flow. Applying Ito's formula, the drift term must be 0, which yields the result. □

The PDE in the above theorem describes the flow of a time-dependent probability distribution. The stochastic process describes individual realizations. The Feynman-Kac Theorem provides the link between stochastic processes and partial differential equations. Thus, we can solve numerous problems with stochastic nature in a deterministic way. Owing to its physical applications, the method of solving PDEs is widely studied in numerical analysis. This helps us understand and calculate different types of financial derivatives. Note that the multivariate version of Feynman-Kac theorem works similarly. This also provides another way to look at the Black–Scholes Formula.

If the contract only depends on the asset value on the maturity date$T$, it is called a financial contract, that is, a simple contingent claim. The payoff can be written as $\phi(S_T)$.

Theorem 2.2.3. *(Black-Scholes PDE [6]) Let $v(t, S_t)$ be the value of the contingent claim ϕ at time $t \leq T$ and assume $v \in C^{1,2}$. Then $v(t, s)$ satisfies the following PDE*

$$v_t(t, x) + rxv_x(t, x) + \frac{1}{2}\sigma^2 x^2 v_{xx}(t, x) = 0,$$
$$v(T, x) = \phi(x).$$

Proof (sketch). The proof comes directly from the Feynman-Kac theorem. Note that here in the infinitesimal generator we use the risk-free interest rate r for the sake of no-arbitrage assumption. $\square$

The idea of the Black-Scholes-Merton model is to create a simple model for pricing options. Note that these assumptions are not always correct, because the volatility in the market is not always constant. However, for simplicity, we adopt the Black-Scholes hypothesis.

Chapter 3

Numerical Methods

Usually in finance, the calculation of a large number of prices of options has to be done within a short time. Therefore, fast and accurate calculation of option prices is crucial. Furthermore, as the asset models or the payoff structures become more complex, closed-form solutions as in the call/put option case are usually not available, or way too complicated to implement. Hence, we should search for efficient and accurate numerical solutions. In this chapter, we introduce the ideas of some of the most popular numerical methods in option pricing: the binomial tree, Monte-Carlo method, FD, ANNs and PINN. This forms the basis for the implementations in Chapter 4.

3.1 Binomial Tree

The binomial tree is a simple and commonly used numerical procedure of option pricing. In particular, we will study the Cox, Ross, Rubinstein tree [7].

The binomial tree is a discrete version of the Black-Scholes constant volatility process. The asset price can only be changed at certain predefined time points $0 = t_0 < t_1 < \cdots < t_N = T$. As shown in Figure 3.1, assume that

$$S_{t_{i+1}} = \begin{cases} S_{t_i}u, & \text{with probability } p, \\ S_{t_i}d, & \text{with probability } 1-p, \end{cases}$$

where $u > 1, d < 1$, and it is usually assumed that $u = \frac{1}{d}$. This means the asset price moves up with probability p and down with probability $1-p$ at each step. Risk-neutral assumption asks

$$e^{rdt} = pu + (1-p)d,$$

which yields

$$p = \frac{e^{rdt} - d}{u - d}.$$

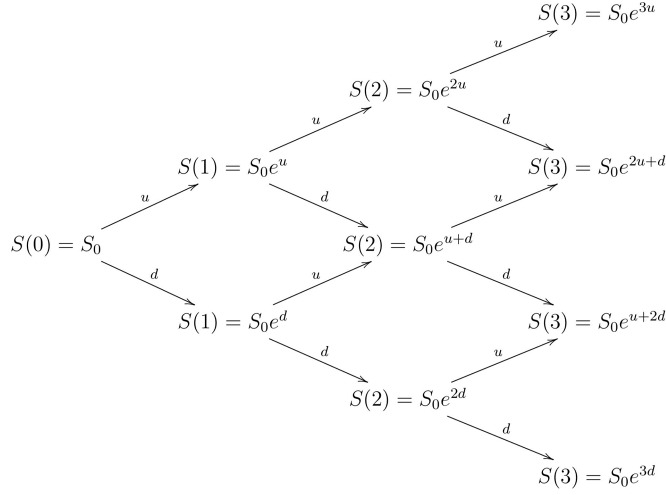


Figure 3.1: Graphical representation of the binomial tree

Theorem 3.1.1.

$$\sum_{i\in\{u,d\}^N} \mathbb{P}(S^i) = 1.$$

Proof. The sum is

$$\begin{aligned}
\sum_{i\in\{u,d\}^N} \mathbb{P}(S^i) &= \sum_{i\in\{u,d\}^N} p^{N_u(i)}(1-p)^{N_d(i)} \\
&= (1-p)^N \sum_{i\in\{u,d\}^N} (\frac{p}{1-p})^{N_u(i)} \\
&= (1-p)^N \sum_{i\in\{u,d\}^N} \binom{N}{k} (\frac{p}{1-p})^k \\
&= (1-p)^N (1 + \frac{p}{1-p})^N \\
&= 1,
\end{aligned}$$

by the binomial theorem. □

With the stock price tree, by walking backward in the tree, it is easy to get an approximation value of the option price at time 0. At the end step of the tree, i.e., at maturity T, the price of the option equals its intrinsic value. For a European type contingent claim, the model then works by taking expectation backwards at each time interval, and calculating the option value at each step. Let $v(n, k)$ denote the option price of the kth node at step n. At maturity:

$$v(N, j) = \phi(S_0 u^j d^{N-j}), \quad j = 0, 1, \ldots, N.$$

Working backwards in time, we have

$$v(i, j) = e^{-rdt} \left(pv(i+1, j+1) + (1-p)v(i+1, j)\right), \quad j = 0, 1, \ldots, N-1.$$

The implementation of the binomial tree is straight forward, and we will discuss some results in the next chapter.

3.2 Monte-Carlo Method

As a commonly used data processing tool, Monte-Carlo simulation is irreplaceable for some situations. For example, when the underlying stochastic process is not Markovian. As mentioned in the previous section, the idea lies in constructing the trajectories of the underlying asset by simulating a Brownian motion. And note again that the option price is the discounted expectation of the payoff at expiry under the risk-neutral measure. Simply, Monte-Carlo option pricing simulation can be divided into several steps as follows:

- Simulate a trajectory of the underlying asset under the risk-neutral measure.
- Force the payoff function at expiry.
- Repeat the simulation for sufficiently many times.
- Calculate average payoff for all simulations.
- Discount the average payoff to time 0 to get the option price.

This discrete way of simulating the time series is also regarded as the Euler method. It depends on the assumption that the underlying process is Markovian, i.e., the current value of the process can be simulated using the previous step. The Monte-Carlo method referring to here depends on the normality of ΔW_t, or, lognormality of the underlying process. Under the risk-neutral measure, we have:

$$S(t + \Delta t) = S(t)\exp((r - \frac{1}{2}\sigma^2)\Delta t + \sigma\sqrt{\Delta t}z),$$

where z is a $\mathcal{N}(0, 1)$ random variable. Usually, it does not matter to use super small steps, as the formula is exact. The key here is the number of simulations.

In many cases, the variance reduction can be used to improve the estimation accuracy of the Monte Carlo method, which can be obtained through a given simulation or calculation. One way to variance reduction is called importance sampling. We will briefly experiment with importance sampling in the next chapter. One of the biggest advantages of the Monte Carlo method is that it can handle European-style options depending on many variables. Of course, one can write down the corresponding PDE for multi-dimensional problems. Solving it, however, is very difficult, and it is what we call "the curse of dimensionality". The equation for higher dimensions is basically the same as the one-dimensional case, and correlated random variables can be calculated using the Cholesky decomposition.

3.3 Finite Difference Methods

Finite difference methods (FDMs) can be used to solve partial differential equations. It uses a discrete format to approximates the required derivative. FDM can handle a small number of dimensions well, since FDM finds the solution of the differential equation through numerically approximating each partial derivative. After finding a solution at each point, it can be plugged back into the grid and give us an approximation over the whole domain.

Assume that $f : \mathbb{R} \to \mathbb{R}$ is a function of x, then the derivative in discrete form is defined to be

$$\frac{\partial f(x)}{\partial h} = \frac{f(x+h) - f(x)}{h},$$

where h is the differentiation step. By Taylor's formula, we have

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \cdots .$$

Combining the two equations, we will get an error $O(h)$ for the derivative. To get an error $O(h^2)$, define:

$$\frac{\partial f(x)}{\partial h} = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + O(h^2).$$

The idea of this section is to use FDM to approximate the solution of the Black-Scholes PDE:

$$f_t(t,x) + rxf_x(t,x) + \frac{1}{2}\sigma^2 x^2 f_{xx}(t,x) = 0,$$
$$f(T,x) = \phi(x),$$

where $\phi(x)$ is the payoff function of the stock price at maturity. To approximate the solution f, we first need to create a grid. Taking the European put option as an example. Let N, M represent the total steps of discretization in time and space, and let $S_{i,j}$ represents the stock price for time $i\Delta t$ and space $j\Delta S$. There will also exist three boundary conditions for the mesh, and it is with these that all the calculations can be made. The terminal condition (at T) can be written as

$$f_{N,j} = K - \max(S_{N,j}, 0), \;\; j = 0, 1, \ldots, M.$$

Apart from the initial conditions (terminal conditions in this case), boundary conditions need to be enforced on the scheme. The lower boundary condition is $S = 0$. Consequently, the price of the put option would be equal to K. For the upper boundary condition, which is the option price at expiry, then the option price would be 0. Applying the finite-difference on the derivatives in the Black-Scholes PDE on the space-time grid gives the following approximations:

- Forward approximations:

$$\frac{\partial f}{\partial t} = \frac{f_{i+1,j} - f_{i,j}}{\Delta t}.$$

- Backward approximations:

$$\frac{\partial f}{\partial t} = \frac{f_{i,j} - f_{i-1,j}}{\Delta t}.$$

- Central approximations:

$$\frac{\partial f}{\partial t} = \frac{f_{i+1,j} - f_{i-1,j}}{2\Delta t}, \quad \frac{\partial f}{\partial S} = \frac{f_{i,j+1} - f_{i,j-1}}{2\Delta S}.$$

- Second derivative:

$$\frac{\partial^2 f}{\partial S^2} = \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{\Delta^2 S}.$$

These approximations would yield the three different schemes of finite difference methods: the explicit (backward approximation), implicit (forward approximation) and Crank-Nicholson (central approximation). The explicit method is the easiest finite difference method to implement and it has the fastest algorithm, but it is also the most unstable one. The method calculates the option prices for each time step using known quantities from the previous time step. The implicit method is more stable. However, it requires larger number of computations. This method does not depend on the quantities from the previous state. Instead, it uses the current state and the next state.

The Crank-Nicholson method is a weighted sum between the explicit method and the implicit method. Applying the Crank-Nicholson idea to the Black-Scholes equation, we get the following grid equation:

$$\begin{aligned} &\frac{f_{i,j} - f_{i-1,j}}{\Delta t} + \frac{rj\Delta S}{2} + \frac{f_{i-1,j+1} - f_{i-1,j-1}}{2\Delta S} + \frac{rj\Delta S}{2}\frac{f_{i,j+1} - f_{i,j-1}}{2\Delta S} \\ &+ \frac{r^2 j^2 \Delta^2 S}{4}\frac{f_{i-1,j+1} - 2f_{i-1,j} + f_{i-1,j-1}}{\Delta^2 S} + \frac{r^2 j^2 \Delta^2 S}{4}\frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{\Delta^2 S} = \frac{r}{2}f_{i-1,j} + \frac{r}{2}f_{i,j}. \end{aligned}$$

One can write the above equation in a compact matrix form and solve it efficiently.

3.4 Artificial Neural Networks

ANNs are data processing entities that have similar information processing properties and functions as the biological neural networks, i.e., human or animal brains. There has been a lot of research done on option pricing by computer scientists in the field of neural networks. The neural networks are not required to fulfill any economic assumptions or axioms as they yield results based on the universal approximation theorem. As discussed before, the Black-Scholes formula is based on some crucial assumptions such as no-arbitrage and lognormality of the underlying process. However, the study on neural networks in option pricing has largely been carried out using the market data. Some analysis has been done by Hutchinson et al. (1994), which show that for American options, the neural network does a better job than the Black-Scholes model. Malliaris Salchenberger (1993), on the other hand, showed the Black-Scholes formula performs better in the case of in-the-money options, whereas the neural network approach is dominant in the outlier prediction. Later on, some researchers also showed that including the economic assumptions in the neural network helps describe the market better. These studies are highly based on market data, which we will not discuss in this project. We will now introduce the basic concepts in the study of neural networks, and two different approaches based on the assumption of the Black-Scholes-Merton model.

Artificial Neural Networks Approximation based on the Black-Scholes-Merton Model

We consider neural networks whose elements are arranged into separate layers, where each layer can control only its output into the next layer. This is called a feedforward neural network since it guarantees the one-way transfer of data [9]. The layers between the input and output layers are called hidden layers and each one of them can have any number of neurons. The number of hidden layers depends on the nature of the approximation problem and there are various empirical techniques on how to choose it.

In the hidden layers and the output layer, we have a continuous output by using a so-called "activation function," which is commonly taken as the sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

since its derivative has a closed form. Other commonly used activation functions are $\sigma(x) = \ln(1+e^x)$, $\sigma(x) = \sin(x)$, the heaviside function and the ReLu-function.

In multi-layer neural networks, we usually use an algorithm called backpropagation to calculate the error distribution after the data go through the network. Given a set of data points $\{(x_n, y_n) : n = 1, \dots, N\}$, where $(x_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$ are independent samples from an unknown probability measure ν, we define the best neural network function $\alpha \in \mathcal{N}_K$, which is defined by the minimized parameter set θ

$$\min_{\alpha \in \mathcal{N}_K} \mathbb{E}[g(y, \alpha(x, \theta))]$$

for a given loss function g. Assume that g is convex in its second argument α. Use the function

$$g(y, \alpha(x, \theta) := (y - \alpha(x, \theta))^2$$

as a general loss function. The goal is to minimize the expectation of the loss function over all possible α's. As discussed before, the backpropagation algorithm works in the minimization procedure by calculates the differences between the training pattern target and the actual value. Based on the differences, the weights are updated. After several steps, it might end up in a steady-state where the weights at each layer are fixed. The most common way to optimize these weights is the gradient descent scheme. For example, consider the simple case where we have one hidden layer in the network, then we write the neural network functions as

$$\mathcal{N}_k := \{\alpha : \alpha(x, \theta) = \sum_{k=1}^{K} \theta_k^1 \sigma(\theta_k^2 x + \theta_k^3)\},$$

and use an iterative scheme to determine

$$\bar{\theta}(n+1) = \bar{\theta}(n) - \nabla_\theta g(y_n, \alpha(x_n, \bar{\theta}(n))).$$

Note that the neural network using several hidden layers works similarly. An example of the neural network structure is shown in Figure 3.2.

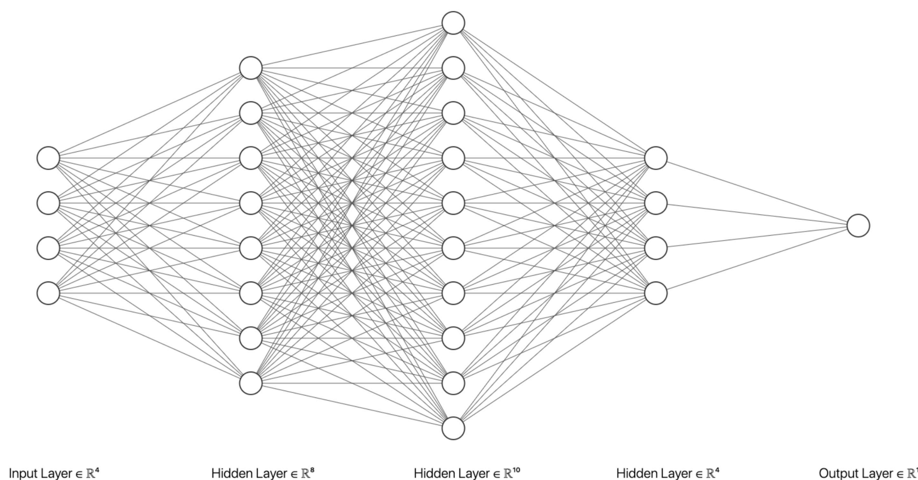


Figure 3.2: An example of the neural network with three hidden layers

Now consider the price of a European call option,

$$C = \phi(S, K, T),$$

and normalize the equation by defining the moneyness $m = S/K$. Then we have

$$C/K = K\phi(S/K, 1, T) \implies c = C/K = \phi(m, T).$$

Some underlying assumptions are imposed on the neural network to meet the behaviour of traders on options markets. For call options, they easily carry to the puts due to put-call-parity. The first condition below ensures a non-increasing price with respect to the strike, and this is because the difference between the underlying price and the strike would be smaller in the case of the in-the-money case:

$$\frac{\partial C}{\partial K} \leq 0.$$

The second assumption ensures the convexity of the price in terms of the strike:

$$\frac{\partial^2 C}{\partial K^2} \geq 0.$$

The third assumption says that the longer the maturity, the larger the probability of having a positive price:

$$\frac{\partial C}{\partial T} \geq 0.$$

The last assumption ensures that when the event of a large enough strike has probability 0, we will make profit through the following option:

$$\lim_{K \to \infty} C(S, K, T) = 0.$$

Use Artificial Neural Networks to the Solve Heat Equation

Another approach of using the neural network to price options is to solve the corresponding PDE. Traditional tools based on discrete domains, like finite element, finite difference and finite volume, perform weak solutions on them on this grid. In general, these methods are usually effective, but the solutions obtained are discrete or have finite differentiability. Another possibility is to use neural networks. 10. The effectiveness of this method depends greatly on the function approximation ability of the feedforward neural network, which finally contributes to the solutions. As already introduced in Chapter 1, we can use the PINN method for example to solve the corresponding PDE. Choosing the suitable discretization and structure of the neural network, the goal reduces again to finding a trial solution that "nearly satisfies" the PDE by minimizing a related error over all hidden nodes. Once the optimal parameters are obtained numerically, we can use this trial solution as a smooth approximation to the true solution that can be evaluated continuously over the domain. One would expect that the solution accuracy would increase with a finer network, but at the expense of computation and possible overfitting. Therefore, we hope to obtain an approximation of sufficient accuracy by using a minimum of computational cost.

We will solve the Black-Scholes PDE analytically by transforming our problem into the heat equation. The heat equation has a well-known solution and has been studied in detail in physics and is easy to implement. The idea is to do two steps of change of variables, to convert the Black-Scholes PDE with the terminal condition into a standard heat equation with the initial condition.

Let define a new random variable τ such that $t = T - \frac{2\tau}{\sigma^2}$, then $t = T$ corresponds to $\tau = 0$. Since the underlying stock price is log-normally distributed, we define $x = \log(S)$. Using these new definitions, the terminal condition can be converted to an initial condition, and get rid of the independent variable S in the dynamics. Finally, a substitution of the form

$$u = \exp(\alpha x + \beta\tau)V$$

can be used to get rid of the unwanted constants and first order in the x terms. With the above construction, we can get a new function u that solves the following standard heat equation:

$$\begin{aligned} u_t = u_{xx}, \;\; x \in \mathbb{R}, 0 \le t \le \frac{\sigma^2 T}{2}, \\ u(x,0) = e^{-\alpha x}\phi(e^x), \end{aligned}$$

where ϕ is the payoff function. If it is a European call option, then

$$u(x,0) = e^{-\alpha x}\max(e^x - K, 0).$$

Finally, to convert the solution of the heat equation back to the option price, we have

$$\begin{aligned} V(t,S) &= Kv(\frac{\sigma^2(T-t)}{2}, \log(S)) = Kv(\tau, x) = Ke^{\alpha x + \beta\tau}u(x,\tau), \\ \alpha &= \frac{\sigma^2 - 2r}{2\sigma^2}, \\ \beta &= -(\frac{\sigma^2 + 2r}{2\sigma^2})^2. \end{aligned}$$

To assign the appropriate boundary conditions on the function u, recall that

$$V(T,S) = \max(S-K,0), \forall S,$$
$$V(t,0) = 0, \forall t.$$

Therefore, let $\alpha := \frac{1-\kappa}{2}$, we have boundary conditions for the call option:

$$u(\tau,x) \to 0, \quad \text{as } x \to -\infty,$$
$$u(\tau,x) \to e^{\frac{(\kappa+1)x}{2}}, \quad \text{as } x \to \infty.$$

Similarly, the boundary conditions for a put option is

$$u(\tau,x) \to e^{-\frac{(\kappa-1)^2}{4}}, \quad \text{as } x \to -\infty,$$
$$u(\tau,x) \to 0, \quad \text{as } x \to \infty.$$

3.5 PINN

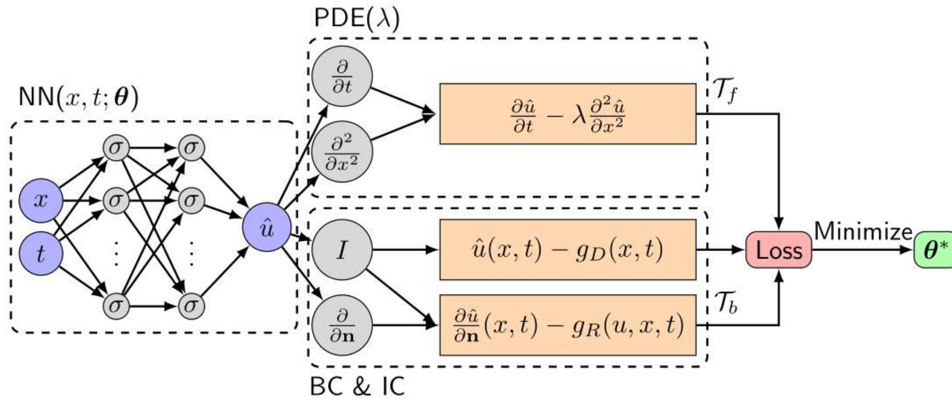


Figure 3.3: Structure of the PINN method with the heat equation as an example [1]

Compared with traditional neural network methods, PINN solving PDE has three advantages. PINN uses automatic differentiation (AD) to embed the PDE into the loss function of the neural network, so the effect of the mesh size on the result can be ignored without defining the mesh. PINN can solve different types of differential equations, and its algorithm is relatively simple, especially for solving inverse problems. Not only that, but a new residual-based adaptive refinement (RAR) method is also proposed, which can improve the training efficiency of PINN.

DeepXDE is a Python library for PINNs. DeepXDE is not only a teaching tool but also a research tool. It is based on constructive solid geometry (CSG) and can support complex domains. At the same time, DeepXDE defines a callback function, through which users can easily supervise and modify the solution process.

PINN evolved from feed-forward neural network (FNN). FNN is the simplest neural network, and it can also be called a multilayer perceptron (MLP). In addition, another neural network based

on residuals, ResNet (residual neural network), is easy to train for deep networks. The library DeepXDE we use for implementation supports both FNN and ResNet. In the implementation of the heat equation next chapter we use FNN since it is sufficient for most of the PDE's.

PINN has four steps. The first step is to construct a neural network $\widehat{u}(\theta; x)$ with parameter θ. Where $\theta = \{w^l, b^l\}$ is the set of trainable weight matrices and bias vectors and σ is a nonlinear activation function. The input of this neural network is x and t, and the output is vector. Using set instead of solution u. Here $\widehat{u}$ is used to approximate the final PDE solution u. In the second step, since $u(x)$ is unknown, we use PDE to train $\widehat{u}(\theta; x)$. Since it is difficult to constrain the neural network in the entire domain, the goal is achieved by constraining two training sets T_f and T_b. T_f is the residual set of the PDE function, and T_b is the residual set on the boundary. I in T_b refers to identity, which is the value of u obtained. $\frac{\partial}{\partial n}$ is the derivative of the vertical vector n. g_D and g_R are two functions that we define according to the different boundary conditions required by PDE, which means that g_D is taken from the set D and g_R is taken from R. The third step, use the L^2 norm to select the loss function for the weighted summation of the PDE equation, and the residuals of the boundary conditions:

$$\mathcal{L}(\theta; \mathcal{T}) = w_f \mathcal{L}_f(\theta; \mathcal{T}_f) + w_b \mathcal{L}_b(\theta; \mathcal{T}_b),$$

where

$$\mathcal{L}_f(\theta) = \frac{1}{|\mathcal{T}_f|} \sum_{x \in \mathcal{T}_f} ||\mathcal{F}(\widehat{u}, x)||^2,$$

$$\mathcal{L}_b(\theta) = \frac{1}{|\mathcal{T}_b|} \sum_{x \in \mathcal{T}_b} ||\mathcal{B}(\widehat{u}, x)||^2,$$

$\mathcal{F}(\widehat{u}, x)$ and $\mathcal{B}(\widehat{u}, x)$ are solved by the AD method. Finally, we find a suitable θ^* through training to minimize the loss function.

There are three methods for selecting residual points: the first is to randomly select points on the grid as the residual points before training, and use the same residual points during the training process. The second is to randomly select different residual points in each iteration during the training process. The third is to adaptively improve the location of the residual points during the training process.

The RAR method was proposed because the more residual points selected, the slower the training time. Therefore, it is necessary to find a balance and use a faster speed to achieve more accurate results. Since the residual points are usually randomly distributed, good results can be obtained in most cases, but for some PDE equations with steep gradient solutions, this may not be so effective, so add more at the place where the PDE residual is greater than residual points. The specific steps are: the first step, select an initial distribution of residual set $\mathcal{T}$, then train the neural network for some number of iterations. The second step is to let $\mathcal{S}$ be randomly take a large number of points in the domain and compute the mean residual ε_r:

$$\varepsilon_r = \frac{1}{|S|} \sum_{x \in S} ||\mathcal{F}(\widehat{u}, x)||^2.$$

The third part is to stop adding residual points when the average residual of the entire domain is smaller than the custom threshold. Otherwise, Select m points with the largest residuals in S and add them to T, and repeat step 2.

Chapter 4

Implementation Results and Discussions

In Chapter 3, we described some of the methods and the different approaches for pricing European options. In this chapter, we perform these algorithms with a European call option as an example and compare the resulting option prices to the true solution provided by the closed-form solution. Furthermore, we will implement the PINN method with the help of the DeepXDE library, by using the partial differential equation introduced in Chapter 3. Lastly, we will provide a discussion and possible modifications for each method.

4.1 Binomial Tree

We implement the binomial tree model on an (at the price) European call option with

$$K = 100, r = 0.05, \sigma = 0.25,$$

and 20 steps and plot the price surface and the tree of the stock price under the risk-neutral measure as follows.

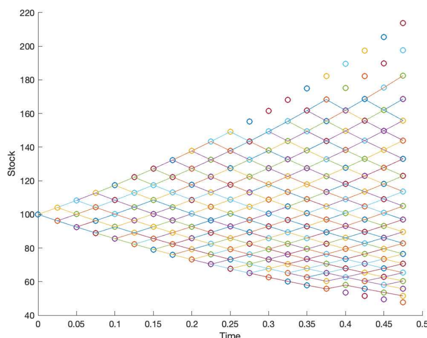
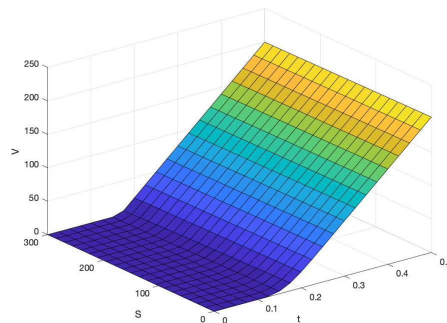


Figure 4.1: Binomial tree illustration



bib2

Figure 4.2: European call option price surface

One can see that the binomial tree is very visible, direct, and deterministic. In order to improve accuracy, using the binomial tree for pricing a European call option, we fix the current stock price to be $S = 100$ and maturity $T = 1$. We run a test for different time steps (up to 2000 steps) and compare the error with respect to the true price of the option, the result is illustrated graphically as in Figure 4.3. The true price of the option is calculated using the Black-Scholes formula.

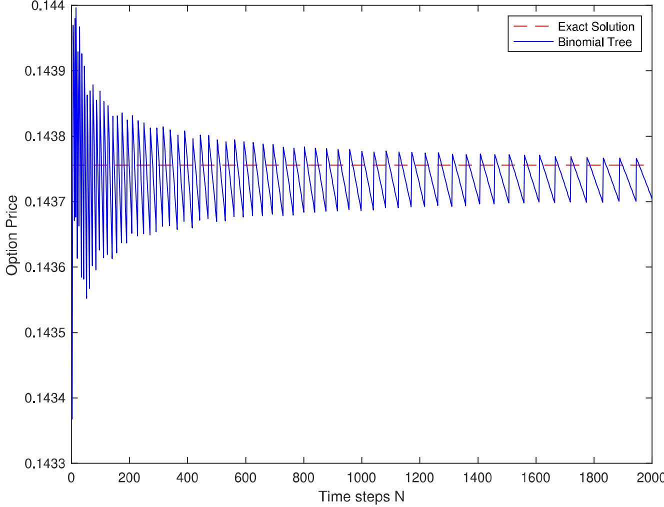


Figure 4.3: Option price by binomial tree against number of steps

One can easily see in the above figure that the more branches we have, the more accurate the model. A variation of the tree model is called the trinomial tree. It is also an efficient method for option pricing, which uses 3 nodes instead of 2 at each step. Trinomial trees are more useful when we want to ensure nodes lie on a given level such as a barrier since this gives the better convergence. Its convergence property is also similar to that of the binomial tree. The following figure illustrates the structure of the stock price evolution under a trinomial tree.

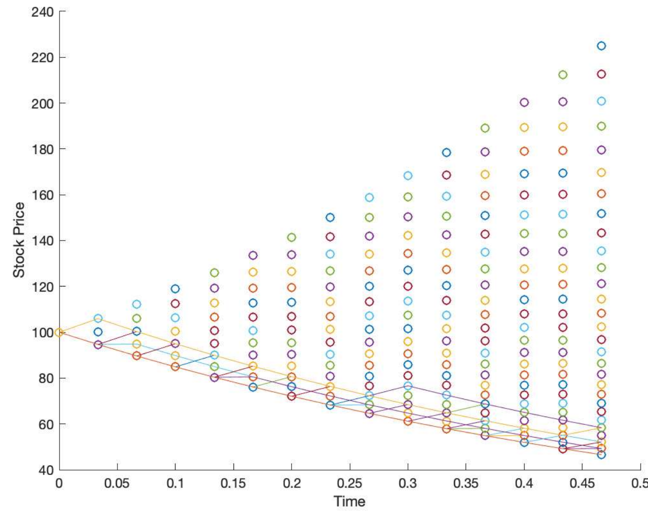


Figure 4.4: Stock price under the trinomial tree

One thing to note in this section is that the binomial method (or the trinomial method) consumes a lot of memory to store the trajectory. For a European type of option, one solution is to directly generate the price distribution at time T. However, we cannot do that if the option is path-dependent. The advantage of the binomial tree method is that it is very easy to understand and implement. We will see later that it is actually just a special case of the explicit finite difference scheme.

4.2 Monte-Carlo Method

The Monte-Carlo method, as explained in the previous chapter, is basically simulating different trajectories and take the expectation of the desired function of the underlying process, in order to get rid of the randomness as much as possible. First simulate five trajectories of a geometric Brownian motion with $S_0 = 100, \mu = 0.05, \sigma = 0.25$, as shown in Figure 4.5 below.

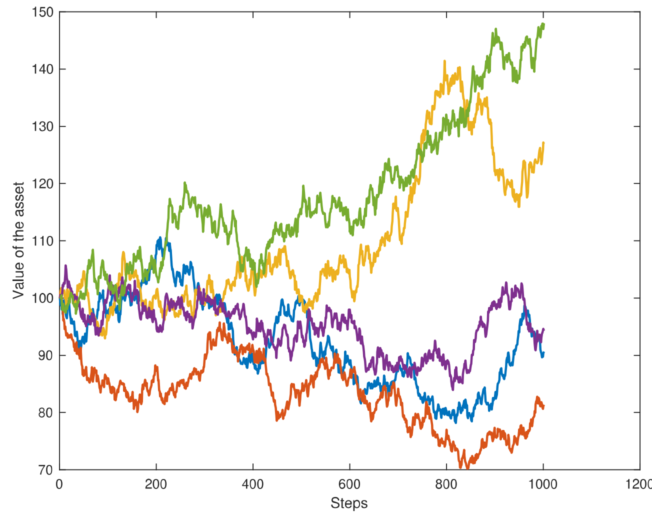
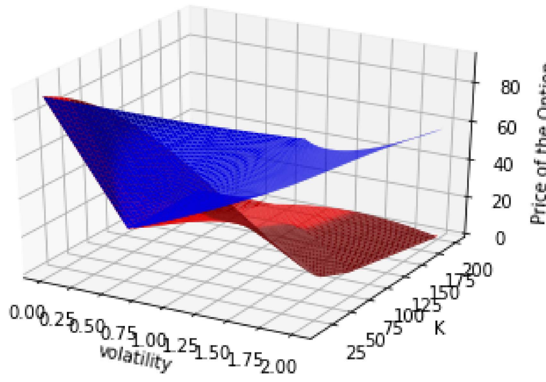


Figure 4.5: Five realizations of a geometric brownian motion by Monte-Carlo against number of steps

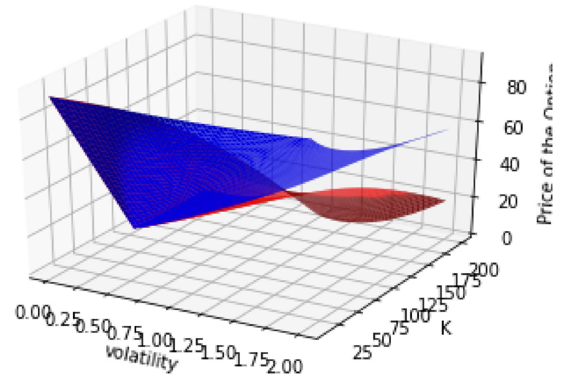
Using the Monte-Carlo simulation method explained in the previous chapter, we implement the pricing of a European call option with the following parameters:

$$S_0 = 100, r = 0.05, T = 1.$$

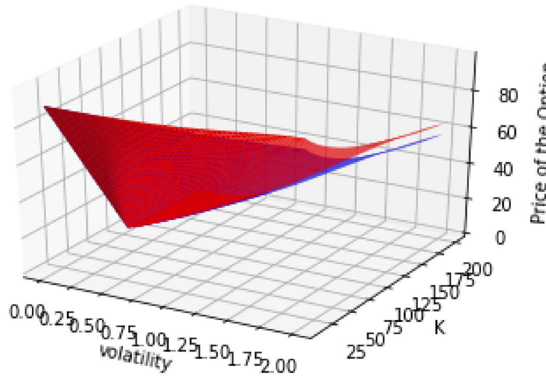
With $10^2, 10^3, 10^4$ simulations respectively, setting the strike K to be varying from 40 to 200, and the volatility σ from 0.01 to 2, we get the following price surface for the corresponding options shown in Figure 4.6.



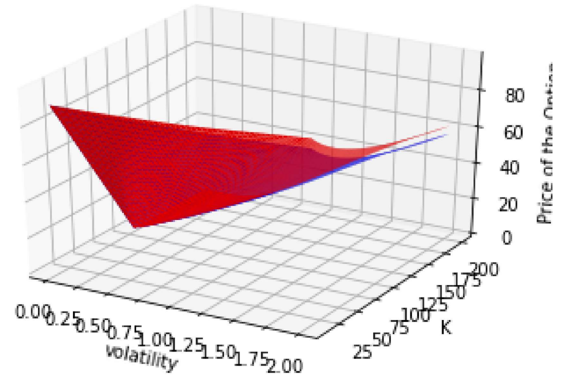
(a) 10 simulations



(b) 10^2 simulations



(c) 10^3 simulations



(d) 10^4 simulations

Figure 4.6: Monte-Carlo pricing surface for a European call option with different numbers of simulations

In the above figures, the blue surfaces represent the Black-Scholes “true” value, and the red surfaces represent the Monte-Carlo results. In terms of a call option, when volatility or the strike is large, the result is extremely volatile. This is because larger randomness in the underlying process brings the larger variance in the Monte-Carlo result, which is the expectation of a function of this randomness. Bigger volatility parameter enlarges the randomness driven by the Brownian motion. When K is large, where $S_T > K$, i.e., the probability such that we have a Monte-Carlo value is small. Therefore, there are a lot of meaningless trials where the stock price does not meet the strike. In addition, the larger the number of simulations, the smaller the error. Let $\sigma = 0.25$ and $K = 100$, and plot the absolute error of the Monte-Carlo simulated price compared to the true price, which is shown in shown in Figure 4.7.

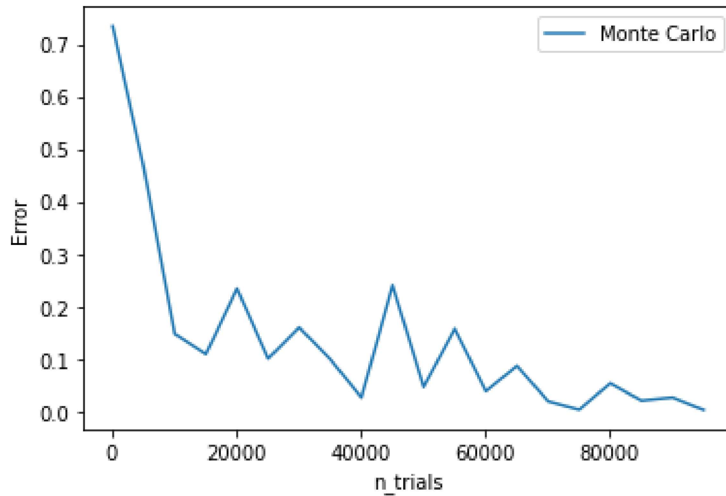


Figure 4.7: Monte-Carlo error against number of simulations

It is easy to see that the Monte-Carlo error decreases with the number of simulations. So we can simply use more trials to increase the accuracy of Monte-Carlo pricing. However, when K is large, this method will not solve the problem, and we will spend a lot of computational power in generating scenarios that we do not use. One way to solve this problem is to use the importance sampling method.

The importance sampling method is a variance reduction technique with the idea that f "important" values are emphasized by sampling more frequently, then the estimator variance can be reduced [8]. The idea is to generate values of S_T under a distribution that is more likely to exceed K. More specifically, we wish to estimate

$$\mathbb{E}^Q[e^{-rT}(S_0 e^Z_T - K)^+],$$

where $Z_T \sim (rT - \frac{1}{2}\sigma^2 T, \sigma^2 T)$. Here Q is a probability measure [6]. Set $K = 200$ to achieve the above variance reduction, and plot the standard error against the number of simulations as in Figure 4.9.

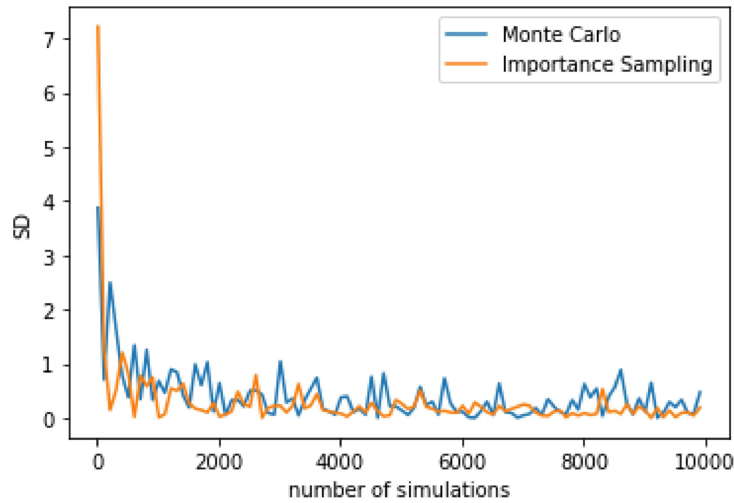


Figure 4.8: Monte-Carlo standard error against number of simulations

We can see that in both methods, the standard error reduces with the number of simulations. However, when simulating a small number of times, through importance sampling, the variance can be significantly reduced. When the simulation is performed multiple times, it is difficult to see the difference, but it may be related to the parameters. Furthermore, we list the computational times with 100 discretization steps of the two different Monte-Carlo methods as shown in Figure 4.9.

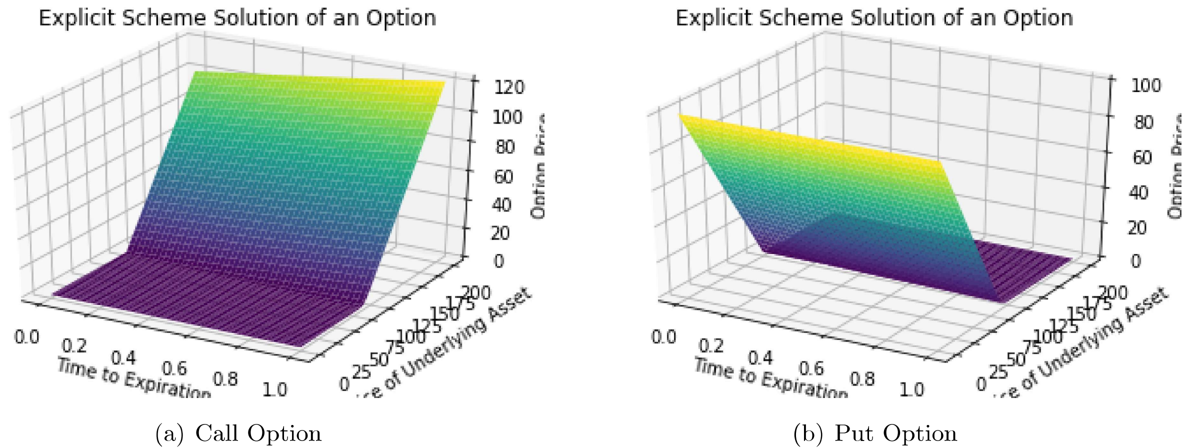
method	runtime	error	standard error	n_trials	n_steps
Black Scholes	0.000419	0.000000	0.000000	100	100
MC	0.001316	0.351944	2.916642	100	100
Importance Sampling	0.000796	1.212279	2.428221	100	100
Black Scholes	0.000385	0.000000	0.000000	600	100
MC	0.004568	0.769842	1.153938	600	100
Importance Sampling	0.002485	1.346573	1.013451	600	100
Black Scholes	0.000371	0.000000	0.000000	1100	100
MC	0.007065	0.226922	0.844478	1100	100
Importance Sampling	0.003939	0.046939	0.814968	1100	100
Black Scholes	0.000368	0.000000	0.000000	1600	100
MC	0.011800	0.596556	0.688706	1600	100
Importance Sampling	0.004743	0.363393	0.670496	1600	100

Figure 4.9: Monte-Carlo running times

We can see that the importance sampling method reduces the variance when the number of trials is limited, and it also reduces the running time. However, Monte-Carlo can take a long while to run when there are a lot of simulations. Monte Carlo is often used because the requirements for mathematics are usually very basic and it is especially easy to compute options depending on several assets.

4.3 Finite Difference Methods

In this section, we implement the explicit method, the implicit method and the crank Nicholson method according to the Black-Scholes PDE. Use the parameters $S_0 = K = 100, T \in [0, 1], r = 0.05, \sigma = 0.025$, we perform the PDE under the above three different methods with time discretization 5000 steps and space discretization 100 steps. Taking the explicit scheme as an example, we plot the price surface as Furthermore, we list the computational times with 100 discretization steps of the two different Monte-Carlo methods as shown in Figure 4.10.



(a) Call Option

(b) Put Option

Figure 4.10: Price surface for call and put options under an explicit scheme

Note that under appropriate discretization, the implicit and Crank-Nicholson schemes yield similar surfaces. During the implementation, it was found that the approximation of the solution sometimes grows exponentially as time increases with the explicit method, for example, with a small number of time steps. Let the time steps be proportionate to the square of space steps, in order to compare three methods. According to [10], the implicit method should require the most computing time, and the Crank-Nicholson method should have the best convergence property. We record the computing times for $N = 10, N = 100$, and $N = 1000$ for the three methods in the following table:

Computing times	Explicit Method	Implicit Method	Crank-Nicholson Method
N=10	0.0240s	0.2052s	0.0301s
N=100	0.1565s	2.1984s	0.2179s
N=1000	10.44s	1303s	23.182s

As we can see from the above table that the explicit method is the fastest method due to its nature of calculating new values from purely the old values, and the implicit method is the slowest. It stopped computing when we raised the number of space steps to 10000. This is because, at each step, it tries to solve an equation involving both the current state and the next one. It requires too much computational power. The Crank-Nicholson method is a combination of the explicit method

and the implicit method. In terms of computational times, it is very similar to the explicit method. We further plot the relative errors under the three methods, with a relatively small number of steps in space, as shown in Figure 4.11.

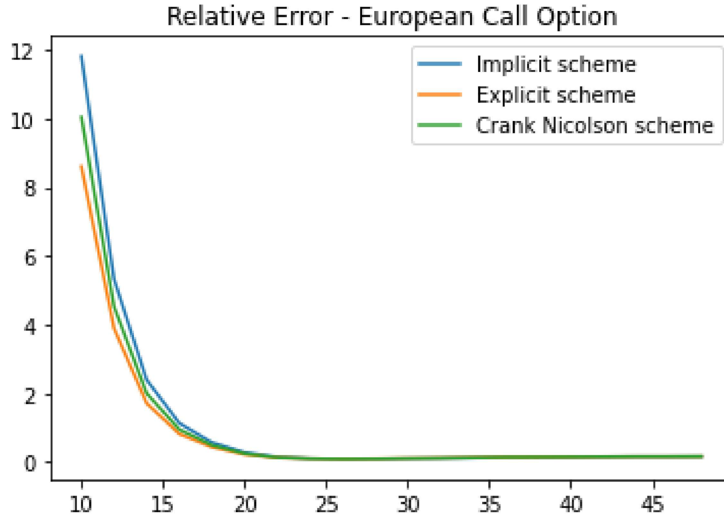


Figure 4.11: Different finite difference schemes error contrast

The above figure shows that the explicit method outperforms the other two methods, while the Crank-Nicholson method has a relatively small error, especially when there is not a very fine mesh.

The finite difference method is a purely deterministic method for option pricing based on the corresponding PDE of the underlying problem. Therefore, it does not consist of any randomness, one can easily see the similarities with the binomial tree method. Similar to the tree method, the finer the mesh, the more accurate the result. However, this also means that the approximated derivatives depend highly on the mesh we choose, which brings up stability and accuracy problems.

After trying the the vanilla call options, we also tried to implement the Heston model below to explore how the finite difference method behaves in higher dimensions:

$$\begin{aligned} dS_t &= rS_t dt + \sqrt{V_t} S_t dW_t^1, \\ dV_t &= a(V_0 - V_t)dt + b\sqrt{V_t} dW_t^2, \\ dW_t^1 dW_t^2 &= \rho dt, \end{aligned}$$

where r, a, b are positive constants. The Heston model assumes that the volatility is stochastic instead of a constant, and that its driving Brownian motion is positively correlated with the Brownian motion which drives the underlying asset.

However, we came across some problems, of which one is to label the grid. We tried to use the Cantor's mapping theorem to solve it but found it very complicated and easy to mess up with. In terms of higher dimensional problems, the Monte-Carlo method works better in terms of simplicity so far. But of course, it is also very slow and depends largely on the number of simulations.

4.4 Artificial Neural Network

If we plot the solution of an European call option as a function of the underlying stock price, it looks like a polynomial in terms of the stock price S. Therefore, we use the polynomial regression in this section. Polynomial regression is a common form of linear regression, where the relationship between x and y (dependent variable) is a polynomial of degree n. Polynomial regression fits the value of x and the conditional mean of y into a nonlinear relationship, shown as $E[y|x]$. First, generate 50000 training data points for a European call option, as shown in Figure 4.15. The training set is generated by simulating 50000 different paths of the geometric Brownian motion from time 0 to maturity.

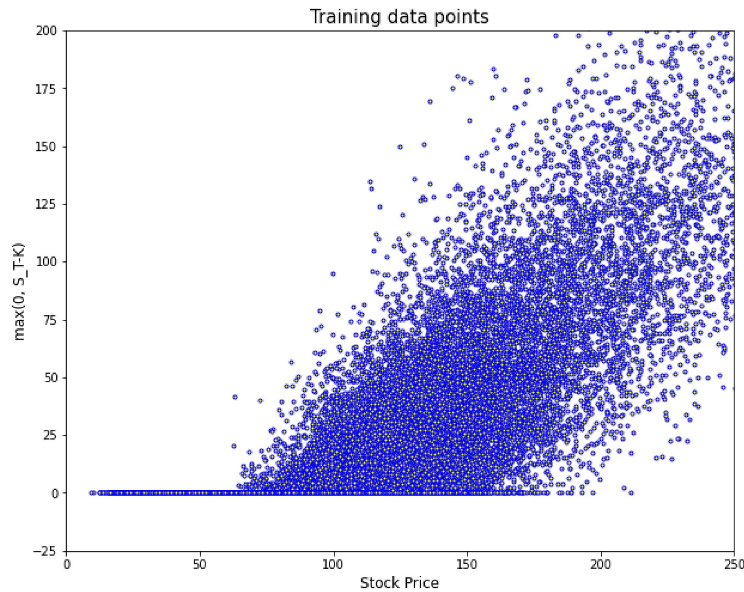


Figure 4.12: Training data sets for a European call option

Now, we propose to use a 7th degree polynomial to approximate the value function. We use the library scikit-learn to achieve this:

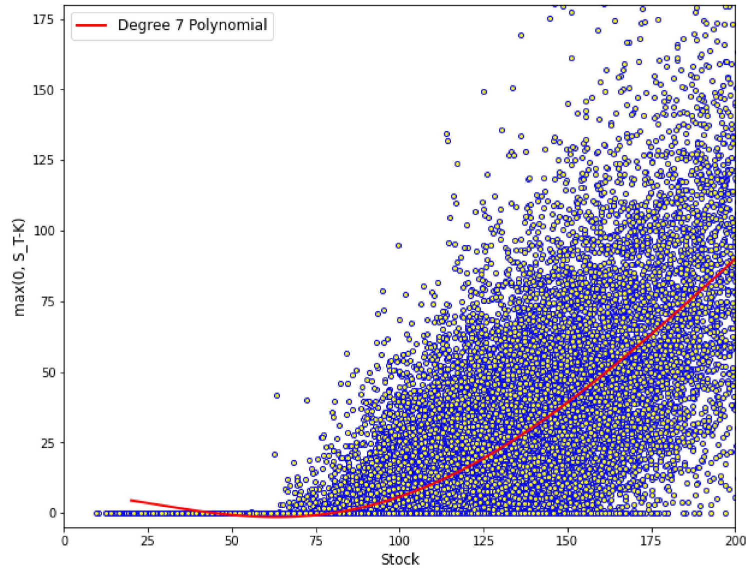


Figure 4.13: Polynomial regression with degree 7

As can be seen from the above figure that when the stock price is low, this regression does not show the monotonicity of the price well. We tried regression with degree 6, but it further drives the error up when the stock price is low. We then performed the regression with degree 8, and in order to access the accuracy, we use the Black-Scholes-Merton price introduced in the previous chapter as the true solution to the value function.

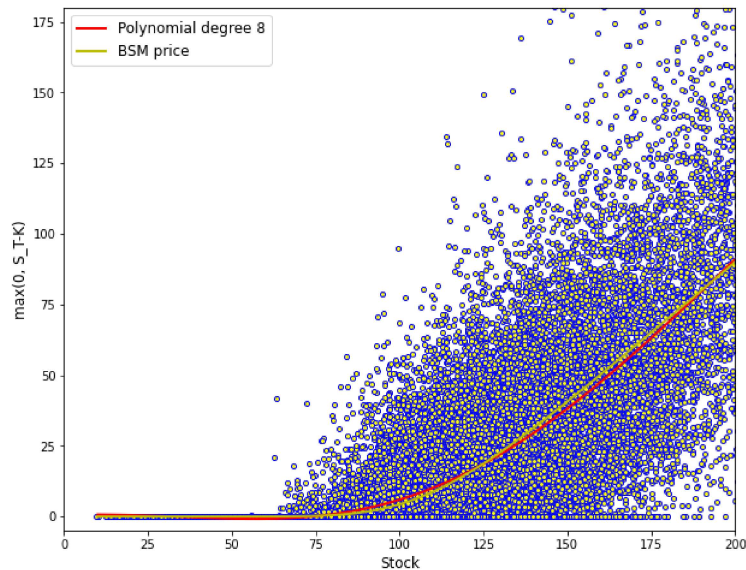


Figure 4.14: Polynomial regression with degree 8

As is shown in the above figure, this polynomial regression works very well. We want to see in the rest of this session if ANN performs just as well. Now, we implement the following vanilla diamond shape neural network, and use the sigmoid function as the activation function:

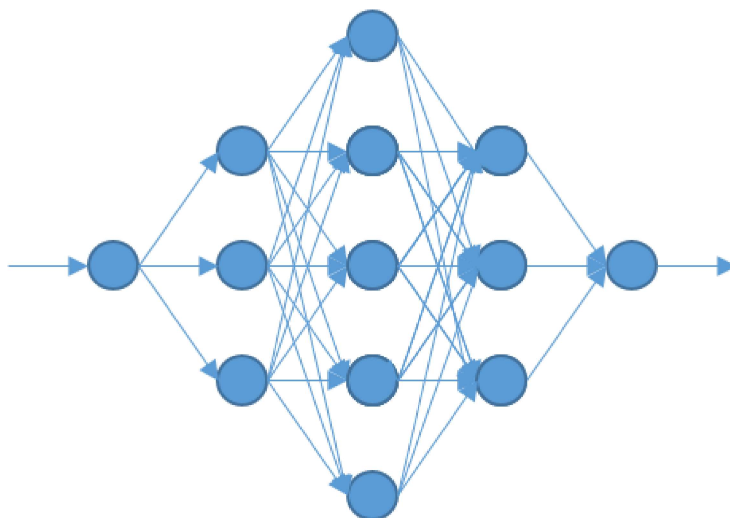
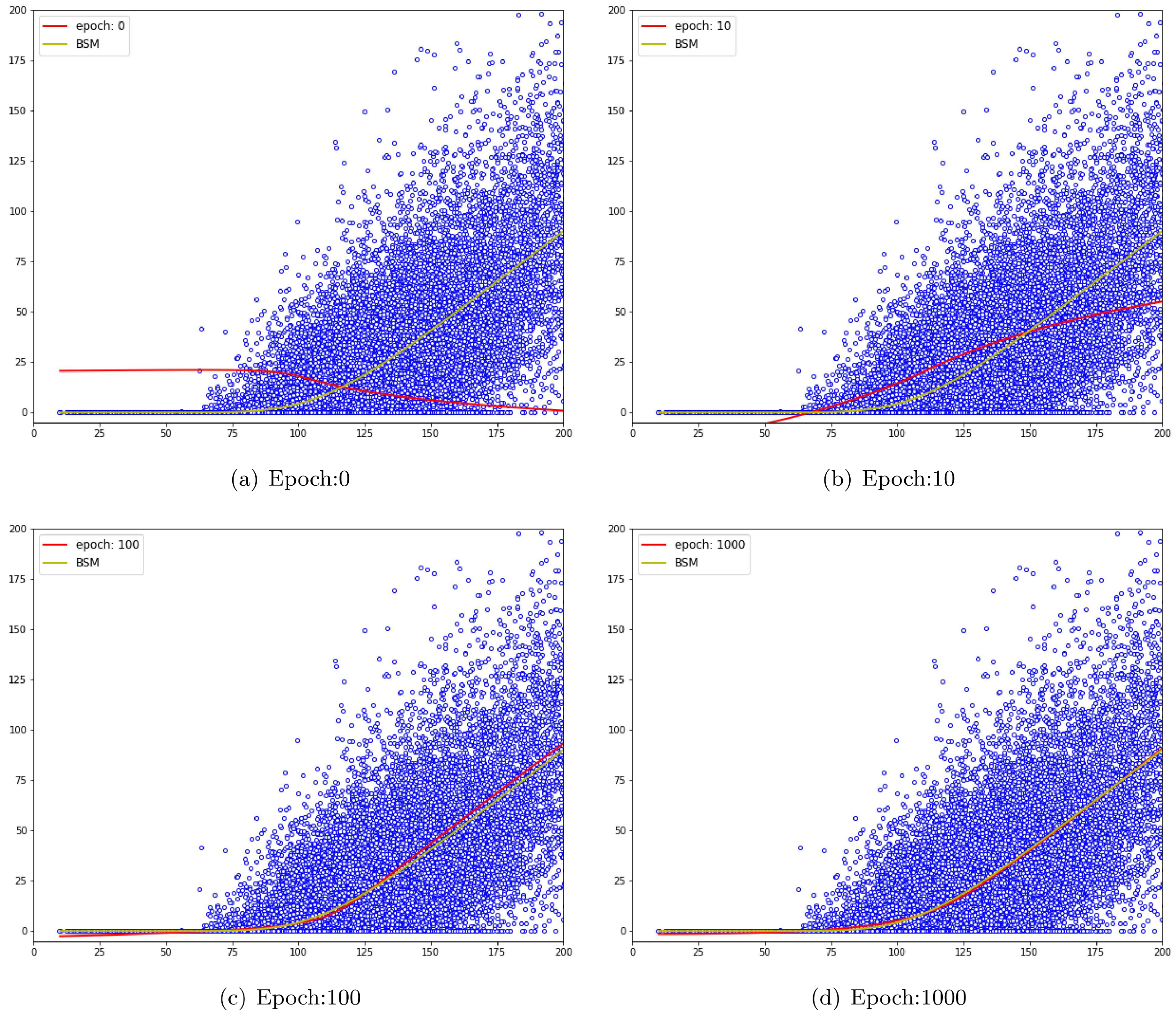


Figure 4.15: Structure of the neural network

We use the gradient descent method, and its general idea is to tweak parameters iteratively in order to minimize a cost/loss function.

The learning rate is an important parameter in gradient descent. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time. Conversely, if the learning rate is too high, it might end up quickly diverging. Let the learning rate to be 0.05, the different performance with different epochs:



(a) Epoch:0

(b) Epoch:10

(c) Epoch:100

(d) Epoch:1000

Figure 4.16: Approximation of ANNs with different epochs

It is shown that the performance with 1000 epochs is very similar to polynomial regression with degree 8. However, the polynomial regression depends a lot on the pre-assumed degree, while the neural network works automatically without any assumption like that. We have achieved a similar result with ANN without pre-assumptions.

4.5 PINN with DeepXDE

In this section, we implement the PINN method to solve the standard heat equation with Black-Schole's initial and boundary conditions using the DeepXDE library. We follow the following procedure:

- Specify the computational domain using the **geometry** module. In our case, it is an interval.

- Specify the PDE using the grammar of **TensorFlow**.
- Specify initial and boundary conditions.
- Specify the training data (at random) into **data.TimePDE** module.
- Construct a neural network using **maps**.
- Call **Model.compile** to set the learning rate.
- Call **Model.train** to train the network.
- Call **Model.predict** to predict the PDE solution at different locations.

We include the script as follows:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
import matplotlib.pyplot as plt
import numpy as np
import deepxde as dde
from deepxde.backend import tf

def xde_call():
    def pde(x, y):
        dy_x = tf.gradients(y, x)[0]
        dy_x, dy_t = dy_x[:, 0:1], dy_x[:, 1:]
        dy_xx = tf.gradients(dy_x, x)[0][:, 0:1]
        return dy_t-dy_xx

    def ini(x):
        r = 0.05
        sigma = 0.25
        k = r/(0.5*sigma*sigma)
        a = (1-k)/2
        return np.exp(-a*x[:, 0:1])*np.max(np.exp(x[:, 0:1])-1,0)

    def func(x):
        r = 0.05
        sigma = 0.25
        k = r/(0.5*sigma*sigma)
        a = (1-k)/2
        b = (k-1)/4-k
        Lmax = 1
```

```
        return np.exp(-a*Lmax-b*x[:, 1:])*(np.exp(Lmax)-np.exp(-k*x[:, 1:]))

    def boundary_l(x, on_boundary):
        return on_boundary and np.isclose(x[0], -3)

    def boundary_r(x, on_boundary):
        return on_boundary and np.isclose(x[0], 0.5)

    geom = dde.geometry.Interval(-3, 0.5)
    timedomain = dde.geometry.TimeDomain(0, 1) # T = 1
    geomtime = dde.geometry.GeometryXTime(geom, timedomain)

    bc_l = dde.DirichletBC(geomtime, lambda X: np.zeros((len(X), 1)), boundary_l)
    bc_r = dde.DirichletBC(geomtime, func, boundary_r)
    ic = dde.IC(geomtime, ini, lambda _, on_initial: on_initial)

    data = dde.data.TimePDE(
        geomtime,
        pde,
        [bc_l, bc_r, ic],
        num_domain=400,
        num_boundary=200,
        num_initial=100,
        num_test=2000,
    )

    layer_size = [2] + [20] * 3 + [1]
    activation = "tanh"
    initializer = "Glorot uniform"
    net = dde.maps.FNN(layer_size, activation, initializer)

    model = dde.Model(data, net)

    model.compile("adam", lr=0.001)
    losshistory, train_state = model.train(epochs=5000)
    dde.saveplot(losshistory, train_state, issave=True, isplot=True)
```

Use the parameters $S_0 = K = 100, T = 1, r = 0.05, \sigma = 0.025$, implement the network for this heat equation and get the pricing curve in the transformed variable x in the figure below:

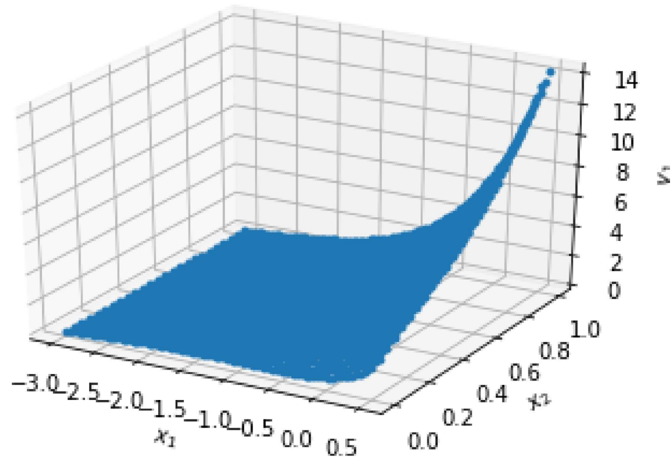
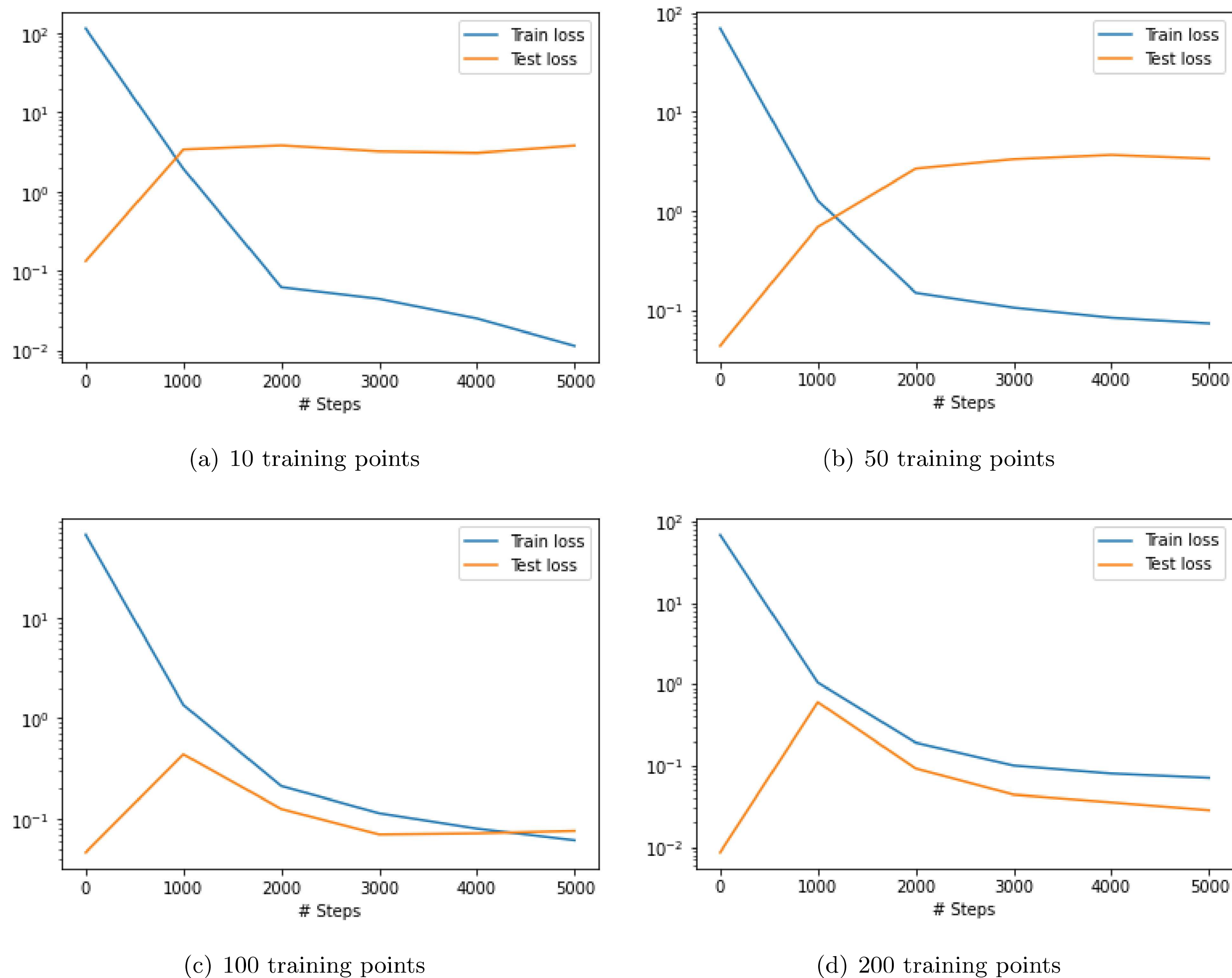


Figure 4.17: Solution surface by DeepXDE

One of the main advantages of using this method to solve PDE is that the procedure of taking the derivative is much more efficient than that of the finite-difference. In PINN, we need to compute the derivatives of the network outputs with respect to the network inputs. This is done by backpropagation, a special technique of automatic differentiation. AD applies the chain rule repeatedly to compute the derivatives. There are only two steps in AD: one forward pass to compute the values of all variables, and one backward pass to compute the derivatives, no matter what the input dimension is [1]. To compute the second-order derivatives, simply repeat AD twice, as we can see from the script above.

First, with the number of epoch=5000 and learning rate = 0.001, we test how the number of training points affects the loss of the model, and plot the results in Figure 4.18.



(a) 10 training points

(b) 50 training points

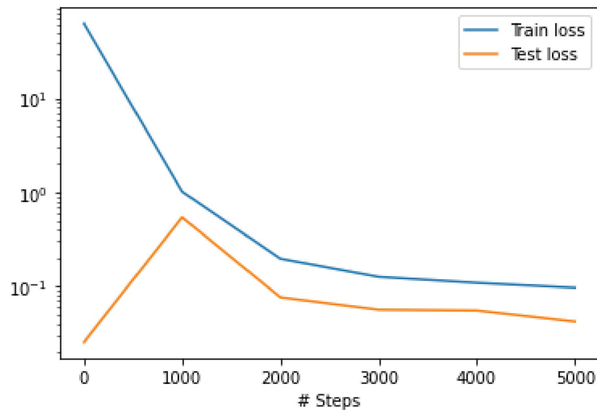
(c) 100 training points

(d) 200 training points

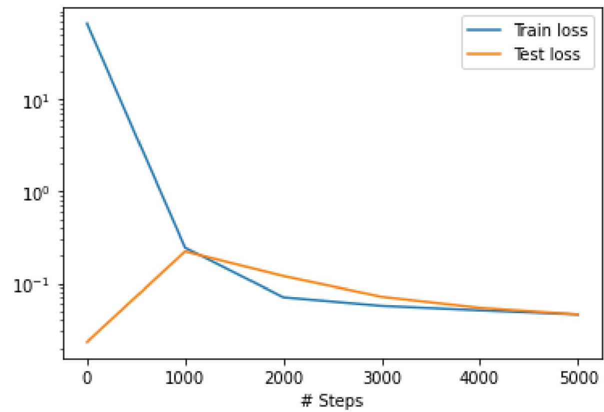
Figure 4.18: PINN training and validation loss with different numbers of training points

It is not difficult to see that with only 10 and 50 training points each time, the training loss is going down, while the test loss is going up. So we need more training points. With 100 and 200 training points, the absolute loss is decreasing with respect to the number of steps. Moreover, the more training points, the less the absolute loss. However, training loss still predominates the validation loss, which means the model is still under fitting for a small number of steps. As the number of epochs grows, the difference between the training loss and test loss gets smaller.

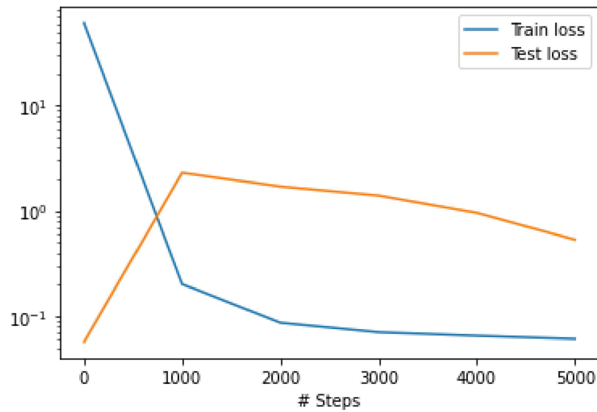
Then, we will test how the neural network depth affects the accuracy of the model. With the number of epochs = 5000, the number of training points = 200, and learning rate = 0.001, we plot the empirical loss for the NN with 2, 4, 8, 16 layers with 20 neurons per layer in Figure 4.19.



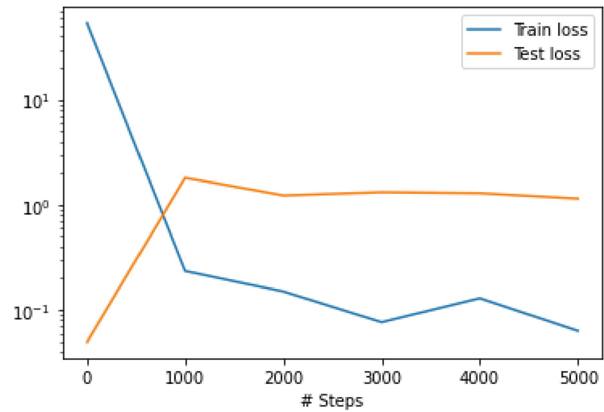
(a) 2 hidden layers



(b) 4 hidden layers



(c) 8 hidden layers



(d) 16 hidden layers

Figure 4.19: PINN training and validation loss with different NN depths

As can be seen from the above figures, with 2 and 4 layers, the absolute loss is reduced a bit. The training loss and test loss of 4 layers are very similar to 5000 steps. However, the absolute loss does of the 8 layers does not change much, but the test loss is larger, which indicates that the model is overfitting. This can be further confirmed by looking at the result with 16 layers. Furthermore, the training times are:

Unit (s)	2 layers	4 layers	8 layers	16 layers
Training time	10.57	24.53	54.67	109.03

We can see that the computational time grows linearly with the NN depth. Through this test, we know that the number of hidden layers does not contribute too much to the accuracy of the model, with a large number of layers, the model could appear to be overfitting, and the computational time is too long.

We will now test how the neural network width affects the accuracy of the model. With the

number of epochs = 5000, the number of training points = 200, and learning rate = 0.001, we plot the empirical loss for a 4-hidden-layer NN with 5, 10, 20, 40 neurons per layer in Figure 4.20:

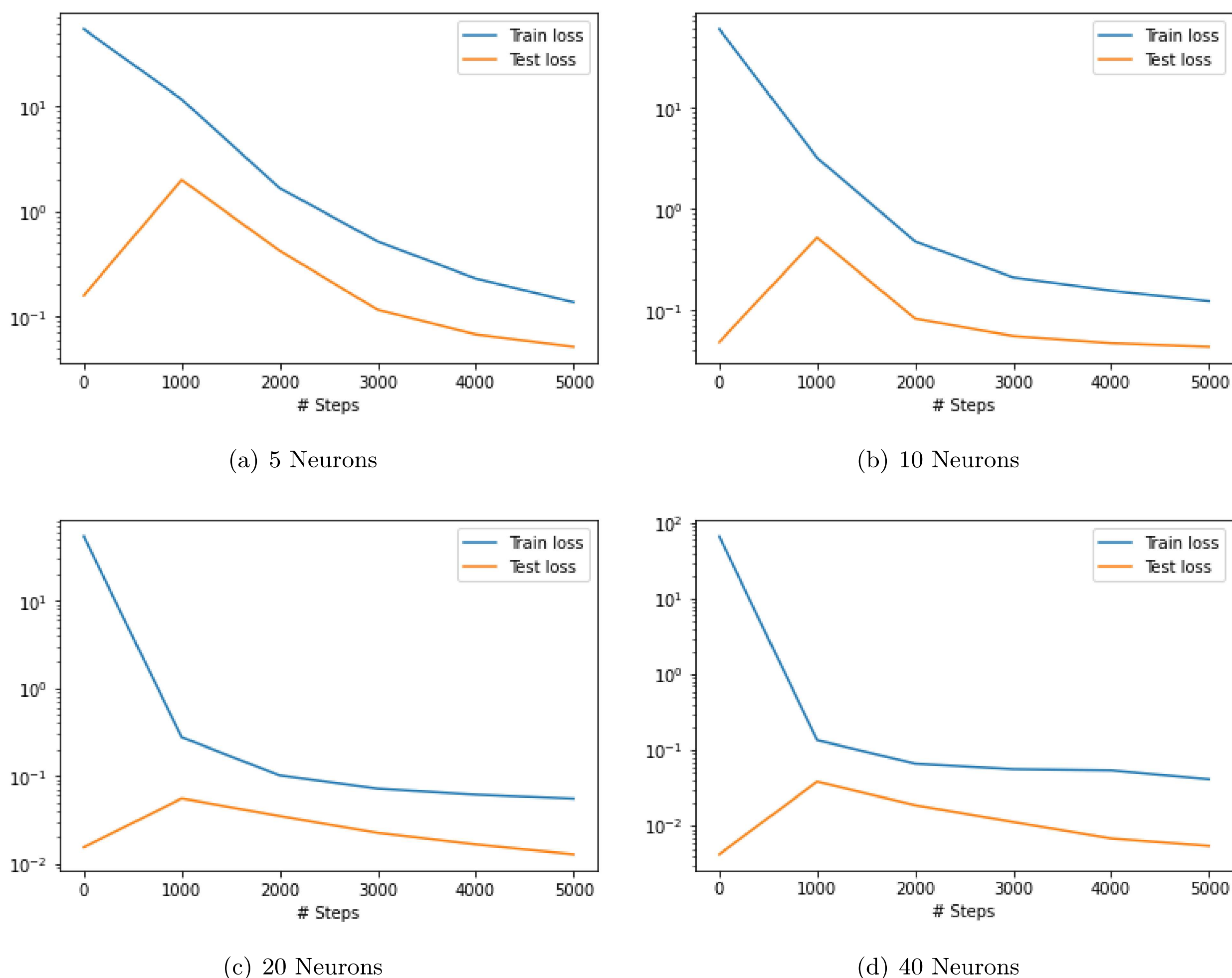


Figure 4.20: PINN training and validation loss with different NN width

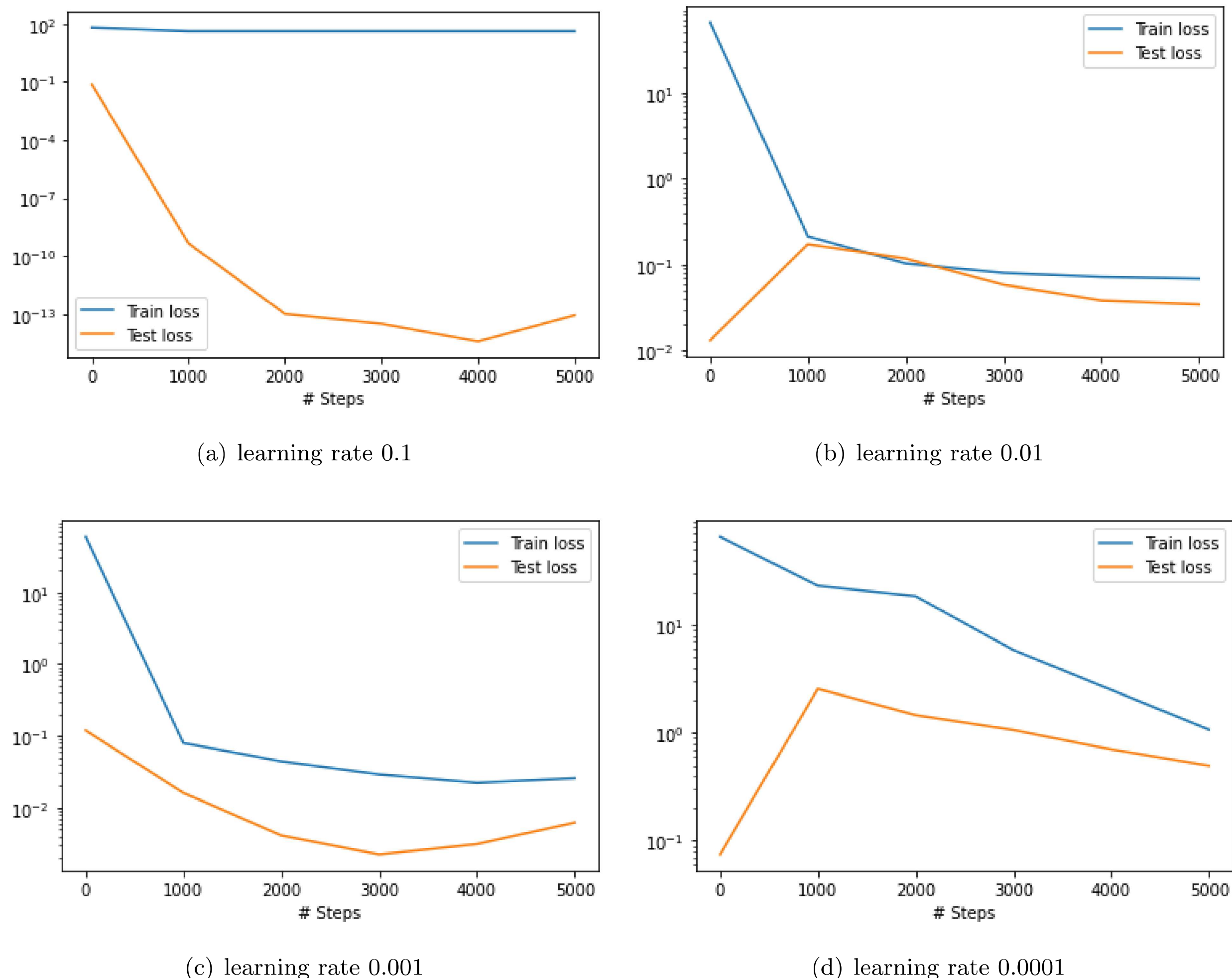
We can see from the above figures that, with 5 to 20 neurons, the absolute loss decreases. When it reaches 40 layers, the training loss and test loss do not change too much. Furthermore, the training times are:

Unit (s)	5 Neurons	10 Neurons	20 Neurons	40 Neurons
Training time	9.028	13.8976	21.1876	35.8706

It can be seen that the computational time grows sub-linearly. However, it does not help so much with too many neurons. In this example, the best NN width is 20. However, in general, we find that the training time for neural networks is longer than the finite difference method.

Finally, we will test how the learning rate affects the accuracy of the model. With the number of epochs = 5000, the number of training points = 200, NN width = 20 and NN depth = 4, we

plot the empirical loss for a model with learning rate 0.1, 0.01, 0.001 and 0.0001 in Figure 4.21:



(a) learning rate 0.1

(b) learning rate 0.01

(c) learning rate 0.001

(d) learning rate 0.0001

Figure 4.21: PINN training and validation loss with different learning rates

It can be seen from the above data that if the learning rate too big, the loss explodes, and if the learning rate too small, the loss increases again. Therefore, in this model, the best learning rate is around 0.001.

Another thing to try with the DeepXDE methods is the residual-based adaptive refinement (RAR) procedure. The residual points are usually randomly distributed in the domain. This works well for most cases, but it may not be efficient for certain PDEs that exhibit solutions with steep gradients. With the European call options, we don't have such a problem, but possibly with other possible payoffs. The RAR procedure works as follows:

1. Select the initial residual points $\mathcal{T}$, and train the neural network for a limited number of iterations.

2. Estimate the mean PDE residual by Monte-Carlo integrations.

3. Stop if the residual is smaller than some pre-defined threshold.
4. Add m new residual points with the largest residual to the current residual set, and go to step 2).

In this way, we make sure that the average residual is smaller than a threshold, by repeating adding points when needed. We did not implement this procedure, but it is also a possibility to try in the future.

Chapter 5

Conclusion and Future Outlooks

In this paper, we introduce the option pricing theory along with its various computational tools. Firstly, we modified the assumptions of arbitrage-free pricing, and the famous Black-Scholes-Merton model, and its connection with partial differential equations. We then discuss the theoretical foundations for different numerical methods, including the binomial tree, Monte-Carlo simulation, finite difference, ANNs and PINN.

We implement the numerical methods mentioned above, and some of the conclusions are as follows:

- The binomial method is the easiest to understand and implement. However, it requires a lot of memory if it is not a simple European option. And if the option is path-dependent, this method cannot be used.
- The Monte-Carlo method can be used when the dynamics of the underlying asset are known. Using importance sampling can solve the problem of large variance caused by large volatility and large strike.
- In the finite difference method, the finer the mesh, the more accurate the result. Therefore, the result highly depends on the mesh we choose. The implicit scheme is the most stable one, but also the most time-consuming. The explicit scheme is not stable in a lot of scenarios but it is the fastest. The Crank-Nicholson scheme behaves relatively better in both cases. But it is not easy to implement with higher dimensions.
- With the artificial neural networks, it can be compared with the true price to directly estimate the option price. And it does not require strong pre-assumptions or information of the model. The more epochs, the smaller the model error.
- Instead of approximating the price directly, we can also price the options by solving the corresponding partial differential equations with various deep learning methods. For example, the PINN scheme. We implement this scheme by using the DeepXDE library with TensorFlow

grammar. This library is easy to use and supports a lot of different features, like callback functions and different geometries.

- With the PINN scheme, the more training points, the smaller the absolute loss. As the number of steps grows, the model gets more and more accurate. The depth of the neural network does not affect the result too much. The error reduces with the width of the neural network, but the speed gets slower. The learning rate should not be too large or too small.

There are a lot of other possibilities with option pricing. For example, with the artificial neural network, instead of the Black-Scholes price, we can use the market price of the options. This is because the Black-Scholes model has an unrealistic assumption: the volatility is constant. In this case, we can get the suitable parameters for the model and reveal a different volatility curve. Another possibility is to look at American options, which can be regarded as an optimization problem. One can use the methods in reinforcement learning to solve these kinds of problems.

References

[1] Lu, L., Meng, X., Mao, Z., & Karniadakis, G. (2020, February 14). DeepXDE: A Deep Learning Library for Solving Differential Equations. Retrieved August 15, 2020, from https://arxiv.org/abs/1907.04502

[2] Option (Finance). (2020, July 15). Retrieved August 15, 2020, from https://en.wikipedia.org/wiki/Option_(finance)

[3] Hull, J. C. (2018). *Options, Futures, and Other Derivatives*. Pearson.

[4] Wilmott, P., Howison, S., & Dewynne, J. (2010). *The Mathematics of Financial Derivatives : A Student Introduction*. Cambridge University Press.

[5] Joshi, M. S. (2003). *The Concepts and Practice of Mathematical Finance*. Cambridge University.

[6] Björk, T. (2020). *Arbitrage Theory in Continuous Time*. Oxford University Press.

[7] Cox, J. C., Ross, S. A., & Rubinstein, M. (1979). Option pricing: A Simplified Approach. *Journal of Financial Economics*, 7(3), 229-263. doi:10.1016/0304-405x(79)90015-1

[8] Glynn, P. W., & Iglehart, D. L. (1987). Importance Sampling for Stochastic Simulations. *Management Science*, 35(11), 1367-1392. doi:10.21236/ada193585

[9] Dindar, Z. (2006, February 10). Artificial Neural Networks Applied to Option Pricing. Retrieved August 15, 2020, from http://wiredspace.wits.ac.za/handle/10539/181

[10] Brandimarte, P. (2006). *Numerical Methods in Finance and Economics : A Matlab−based Introduction*. Wiley-Interscience.