Option Pricing with Physics-informed Neural Networks

Nichume Zamxaka
Supervisor: Dr Ralph Rudd

A dissertation submitted to the Faculty of Commerce, University of Cape Town, in partial fulfillment of the requirements for the degree of Master of Philosophy.

February 10, 2023

MPhil in Mathematical Finance,
University of Cape Town.

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Philosophy in the University of the Cape Town. It has not been submitted before for any degree or examination in any other University.

September 29, 2023

Signed by candidate

Type text here

Abstract

We investigate the application of physics-informed neural networks (PINNs) to option pricing. PINNs are neural networks that are trained to numerically solve partial differential equations (PDEs) by obeying the dynamics induced by the PDE as well as the initial/terminal conditions of the PDE. They are mesh-free to an extent and compute the derivatives of the PDE through backward-propagation. We construct a PINN toy example to solve the Black-Scholes-Merton PDE for a vanilla European option. The numerical solutions from the PINN are compared against the true analytical solution – the Black-Scholes-Merton equation. The problem is also extended by incorporating a local volatility model. Here, we derive the PDE of a vanilla European option under the constant elasticity of variance (CEV) model. We then construct and train a PINN to solve the PDE and compare it to the true analytical solution of a special case of the CEV model, the square-root process.

Keywords: physics-informed neural networks, partial differential equation, option pricing, mesh-free, local volatility.

Acknowledgements

I thank my supervisor Doctor Ralph Rudd for his patience and direction throughout the process of this dissertation as well as Professor David Taylor for the rewarding opportunity.

I thank my parents, Nonela and Thembinkosi Zamxaka for the constant reassurance and supporting me in all my endeavours. I thank my two aunts, Felicia Mangisa and Nontlantla Zamxaka and my uncle, Thembelani Zamxaka for helping me to work on the dissertation during the holidays.

Finally, I would like to thank Absa Capital for the financial assistance they provided me with during the 2022 academic year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this dissertation, we are particularly concerned with investigating the use of physics-informed neural networks (PINNs) to solve the Black-Scholes-Merton - (BSM) partial differential equation for pricing options. PINNs are deep neural networks that are trained to learn the differential operators and initial and boundary conditions of PDEs. Problems in traditional applied mathematics, such as PDEs, are increasingly being solved using deep neural networks. (Cuomo *et al.*, 2022).

High-dimensional partial differential equations are important as they describe the behaviour of variables in a system (Evans, 2010). They are used in almost all fields including physics, engineering, and finance (Guo *et al.*, 2020) but solving them remains a challenge. Most PDEs have no analytical solutions (Guo *et al.*, 2020). As a result, solving PDEs numerically has been an active area of research. Traditional numerical methods include the finite-difference method, finite-element methods and Runge-Kutta methods. However, these methods have challenges of their own.

The finite-difference method suffers from the curse of dimensionality. This means that as the dimension of the PDE increases, it may become computationally expensive for the method to arrive at a solution (Sirignano and Spiliopoulos, 2018). This is mainly because of the explosion of the number of grid points and the need to sometimes reduce the step size as the dimension of the PDE increases (Sirignano and Spiliopoulos, 2018). Additionally, these approaches might be limiting in that they demand an a priori specification of the grid points, which, if done incorrectly, could cause stability and convergence problems. These techniques are also less appealing because the PDE solutions are only given for points that are on the grid.

More recently, studies have shown that neural networks have great potential in providing numerical solutions to PDEs (Guo *et al.*, 2020). In their paper, Sacchetti *et al.* (2022) reveal that neural networks have two fundamental features that make them suitable for solving PDEs. Firstly, neural networks satisfy the univer-

sal approximation theorem. The universal approximation theorem states that there exists a network that can approximate any continuous function with arbitrary accuracy (Bishop and Nasrabadi, 2006). They also mention that neural networks are infinitely-differentiable functions with the derivatives being computed using back-propagation and the chain rule for differentiation. This feature makes it inexpensive to compute derivatives required to formulate PDEs and is a massive improvement from traditional numerical methods which numerically approximate these derivatives.

This dissertation presents the physics-informed neural network first proposed by Raissi *et al.* in 2017. This particular kind of neural network uses the dynamics captured by the PDE to inform the construction of the objective function. It leverages the structure of the neural network to be able to compute derivatives easily. These derivatives are used to construct the PDE that features in the objective function. We now provide an overview of the paper.

In chapter 2, a brief literature review on PINNs is provided along with a background on the structure of basic neural networks and PINNs. In chapter 3, we give a background on option pricing and derive the PDEs that are solved in the dissertation. Chapter 4 follows with the methodology of the construction of the PINN along with the numerical results of the BSM PDE and the CEV PDE. Chapter 5 concludes the paper.

Chapter 2

Background

2.1 Literature review

Using neural networks to approximate the solution to PDEs has been an active area of research for many years (Sirignano and Spiliopoulos, 2018). According to studies, deep neural networks are gradually performing well when it comes to solving PDEs numerically. This is attributable to developments in fast and robust computing. In this section, we briefly present literature on machine learning methods that have been explored in the past to numerically solve PDEs.

Lagaris *et al.* (1998) presented a model to provide a solution to PDEs where they decomposed the problem into two parts. The first part involves satisfying the initial/boundary conditions and the second part involves a feed-forward neural network that is trained to obey the laws induced by the PDE. The results showed good convergence properties. However, as the dimension increases they acknowledged that the training sets become large – placing pressure on memory and computation time.

In 2018, Han *et al.* (2018) developed a strategy for solving high-dimensional non-linear parabolic PDEs using neural networks. The strategy involved reformulating the PDE into a backward stochastic differential equation (BSDE) and then using neural networks to approximate the unknown gradient of the solution (Han *et al.*, 2018). The disadvantage of this method is the discretization error that comes from approximating the integrals of the BSDE using the Euler scheme. Dockhorn (2019) in his paper also showed that small neural networks with just one hidden layer were successful in learning the solution to Poisson's equation as well as the Navier-Stokes equations.

Although these papers use neural networks to approximate the solutions, most still rely on defining a fixed mesh before training commences. Most of them use the supervised learning approach which requires a large volume of training data as they do not consider the physical laws embedded in the PDE (Guo *et al.*, 2020).

In contrast, physics-informed neural networks introduce the information provided by the PDE during the training process. This reduces the need for large data in the training process and leads to convergence much quicker (Guo *et al.*, 2020).

The key advantage of PINNs is the inexpensive computation of derivatives. This property makes it more attractive to use for solving PDEs. It is also an improvement from particularly the finite-difference method that numerically approximates these derivatives. PINNs are also mesh-free. During the training process, the inputs can be generated randomly within a bounded space in every *epoch*[1]. In this way, PINNs overcome the need for defining a fixed grid and fixed step sizes for the training inputs(Guo *et al.*, 2020). PINNs are also more flexible than classical neural networks in that they can be trained to solve even more challenging PDEs. Classical neural networks must be re-adjusted to the particular characteristics of different PDEs. They also overcome the curse of dimensionality but at an expense of an increase in the training time (Raissi *et al.*, 2019). As the dimension of the PDE increases, the training data increases as well which may decrease the speed of the training process.

PINNs can be trained to solve PDEs in two different ways. The first method is called the data-driven solutions of partial differential equations and the second is the data-driven discovery of partial differential equations (Raissi *et al.*, 2019). The first method tries to deduce the behaviour of the hidden solution from the data given the fixed *model parameters* [2]. The second method tries to determine the model parameters which best describe the observed data. In this dissertation, we will focus on the data-driven solutions of PDE. To begin our discussion on how PINNs work, we first provide an explanation of how neural networks work in general.

2.2 Structure of Neural Networks and Backward Propagation

2.2.1 Perceptrons and Deep Neural Networks

A feedforward neural network is made up of an input layer, hidden layers, and an output layer. The number of hidden layers in a neural network is usually referred to as the depth of the neural network. Neural networks with a large number of hidden layers are called deep neural networks. Each layer is made up of a number of perceptrons. A perceptron is the most basic element of a neural network (Bishop and Nasrabadi, 2006). Figure 2.1 shows that a perceptron consists of a

[1] Epoch is one complete run of the training data during the training process of the neural network
[2] Model parameters, here, refer to the coefficients of the PDE

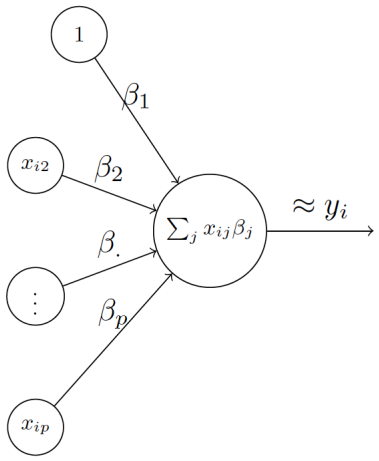

Fig. 2.1: The structure of the perceptron.

node/neuron representing the activation function as well as the edges representing the weights/parameters of the network. The perceptron receives the inputs, $\mathbf{x}$ and computes the dot product of the inputs and the weights, β to get $\mathbf{x}^\top\beta$. This is sent to the neuron where the activation function is applied. In figure 2.1, the activation function is the identity function. The activation function could be any function with popular functions including sigmoid, ReLu and the hyperbolic tangent. Once the activation function is applied, the perceptron returns the result as an output. In mathematical terms, the output can be expressed by the following equation,

$$\mathbf{y} = f(\mathbf{x}^\top\beta)$$

Other texts also include a bias term, $\mathbf{b}$ in which case the output would change into the following,

$$\mathbf{y} = f(\mathbf{x}^\top\beta + \mathbf{b})$$

The idea behind constructing a deep neural network is to use the output of perceptrons of the previous layer as input into the perceptrons of the next layer. This is how deep neural networks are constructed. Below is a figure that ultimately shows the structure of a deep neural network.

2.2.2 Back-Propagation and Gradient Descent

Now that there is a basic understanding of how a neural network arrives at a prediction we can proceed to show how it computes the optimal weights and bias parameters. This is called the training process. Suppose that we have a function

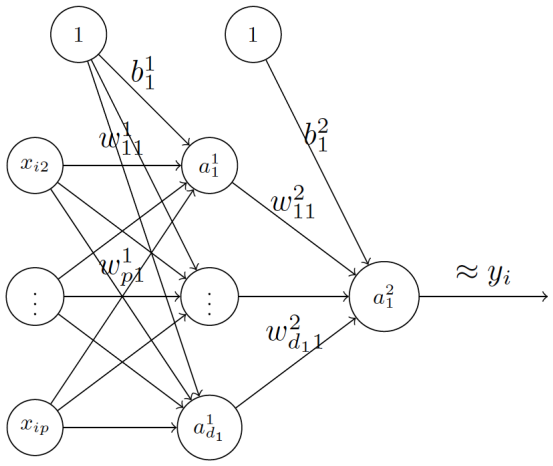

Fig. 2.2: The structure of the feed-forward deep neural network.

$g : \mathbb{R}^2 \to \mathbb{R}$ that we are trying to approximate and our neural network predicts $G : \mathbb{R}^2 \to \mathbb{R}$. To test how well the algorithm approximates the function we can construct an error/objective function that measures the difference between the predicted and the true values of the function. We can feed the network a set of inputs $\mathbf{x}_i = [a, b]^T$ for some $i = 1, 2, .., N$ of whose function values $g(\mathbf{x}_i)$ we know. This set is usually referred to as the training set.

The objective function would look as follows,

$$O := \frac{1}{N} \sum_{i=1}^{N} (g(\mathbf{x}_i) - G(\mathbf{x}_i))^2$$

and the goal would be to find the function $G : \mathbb{R}^2 \to \mathbb{R}$ such that the objective function is minimized. This is what a neural network essentially does in the background of the training process. It seeks the optimal parameters that minimize the objective function. The process through which it minimizes the objective function is via an optimization process. The gradient descent is the commonly used optimization algorithm in training neural networks.

Next, we provide a heuristic explanation of how the gradient descent works. Suppose that we are on a surface in $\mathbb{R}^2$ defined by a function $g(x, y)$ and we wish to find $\mathbf{x}^* = [x^*, y^*]^\top$ at which the surface is at a minimum. Suppose we choose an initial point $\mathbf{x}_0 = [x_0, y_0]^\top$, we can decide on which direction we want to move by computing the gradient of the surface where we are now and find which direction is downward from our present position and take a step in that direction. The downward direction would be informed by the negative of the gradient computed at that point. This is because the gradient at a point provides the direction in which the

function g would experience its steepest ascent. Thus, the negative gradient points in the direction of the fastest decrease and would eventually lead to the minimum of the surface. This is equivalent to the following mathematical expression,

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \lambda \nabla g(\mathbf{x}_i)$$

for some $i = 1, 2, 3....$ till a maximum number of steps or predefined tolerance on the gradients is reached.

The $\nabla g(\mathbf{x}_i)$ denotes the partial derivative of $g(\cdot)$ with respect to every element of the vector $\mathbf{x}$. It is also known as the gradient of $g(\cdot)$. The λ denotes the *learning rate*[3]. So, we take a downward step that is proportional to the size of the gradient and this proportion is determined by the learning rate. This is the process that happens in every epoch during the training process until the weights arrive at the minimum of the objective function.

The gradients of the loss function with respect to the weights are calculated elegantly via a process called back-propagation. Back-propagation depends entirely on the chain rule for differentiation (Bishop and Nasrabadi, 2006). Looking at figure 2.2, suppose we want to evaluate $\frac{\partial y_i}{\partial w_{11}^1}$. In order to do so, back-propagation starts by evaluating $\frac{\partial y_i}{\partial w_{11}^2}$ then also $\frac{\partial w_{11}^2}{\partial w_{11}^1}$ and applying the chain rule leads to $\frac{\partial y_i}{\partial w_{11}^1}$. The derivatives are computed from the output layer to the input layer ensuring that all derivatives with respect to each of all the weights have been evaluated. Understanding the basic elements of a neural network is important for the discussion of how PINNs work.

2.3 Physics-informed Neural Networks (PINNs)

In this section, we provide an analysis of how PINNs learn the solution to a partial differential equation. We adopt the notation that was used in (Dockhorn, 2019). Let $\Omega \subset \mathbb{R}^D$ be the domain and let $\partial\Omega \subset \mathbb{R}^{D-1}$ be the boundary of Ω and consider the following PDE for some non-linear differential operator, $\mathcal{N}$,

$$\begin{aligned} \mathcal{N}(u(\mathbf{x})) &= 0, \quad \text{in} \quad \Omega \\ u(\mathbf{x}) &= g(\mathbf{x}), \quad \text{on} \quad \partial\Omega \end{aligned} \tag{2.1}$$

where D denotes the number of space dimensions, $u(\mathbf{x})$ denotes the hidden solution of the PDE and $\mathbf{x}$ denoting the space variables. The function $g(\mathbf{x})$ corresponds to some known initial/boundary condition. To find the solution $u(\mathbf{x})$ we proceed

[3] The learning rate is a tuning parameter in an optimization algorithm that informs the step size in every iteration when minimizing the objective function

by constructing a deep neural network $\hat{u}(\mathbf{x};\theta)$ where θ denotes the parameters of the network.

The next step is to construct the PDE as represented by equation 2.1 using the derivatives of the neural network with respect to its inputs. As discussed before, these derivatives are computed through backward propagation. We define $\hat{f}(\mathbf{x})$ to be the resulting PDE given by the following expression,

$$\hat{f}(\mathbf{x}) = \mathcal{N}(\hat{u}(\mathbf{x}))$$

and once we have this we can successfully construct the objective function. The objective function is the weighted sum of the mean squared error loss relating to the boundary training data and the mean squared error loss relating to the *collocation points*[4] (Raissi *et al.*, 2019). Both Raissi *et al.* (2019) and Dockhorn (2019) propose an objective function of this form,

$$O_{\text{Total}}(\theta) = \frac{1}{N_{\text{f}}} \sum_{i=1}^{N_{\text{f}}} \left(\hat{f}(\mathbf{x}_i)\right)^2 + \frac{1}{N_{\text{bou}}} \sum_{j=1}^{N_{\text{bou}}} \left(u(\hat{\mathbf{s}}_j) - g(\mathbf{s}_j)\right)^2 \tag{2.2}$$

where $\{\mathbf{x}_i\}_{i=1}^{N_{\text{f}}}$ and $\{\mathbf{s}_j\}_{j=1}^{N_{\text{bou}}}$ are randomly sampled input data from Ω and $\partial\Omega$ respectively. The sample can be changed in every epoch to avoid the need to ever create a fixed mesh (Sirignano and Spiliopoulos, 2018). After understanding the construction of the objective function, the next step is to find the optimal parameters θ^* such that the function is minimized. By doing this we are indirectly uncovering the hidden solution $u(\mathbf{x})$. From 2.2 it can be seen that no knowledge of the solution to the PDE was required. This is why PINNs are said to follow the unsupervised learning approach (Louskos, 2020).

[4] Collocation points are the points at which we are trying to solve for the PDE. Some other texts refer to this as the internal domain.

Chapter 3

The Black-Scholes-Merton PDE

3.1 Theory on financial options

The goal of this dissertation is to investigate the application of PINNs to option pricing, similarly to Louskos (2020). This entails constructing a PINN to solve the Black-Scholes-Merton PDE for European vanilla options. We start with a brief analysis of option pricing. A European option is a financial derivative that gives the holder of the option the right but not the obligation to buy/sell an asset at a fixed price at some future time (Hull, 1993). This particular asset is called the underlying. The time when the holder exercises their right is called the maturity time of the option. For a European option, the holder can only exercise their right at maturity. In contrast, an American option allows the flexibility of exercising at any time before maturity.

Calculating the price of a European option requires modelling the price of the underlying asset at maturity. The price of the underlying asset from the inception of the contract till maturity is usually volatile and the evolution thereof can never be known with certainty. Models such as the binomial model and the Black-Scholes-Merton model were popularly used to price these options in the past Hull (1993). The binomial model makes use of a 'tree-like' formulation that approximates the discrete evolution for the underlying asset prices. The underlying assumption of the model is that the asset price follows a random walk. Together with risk-neutral valuation, it provides the price of an option. Alternatively, the Black-Scholes-Merton model assumes a continuous evolution for the underlying asset prices.

3.2 The option pricing PDE under the Black-Scholes-Merton model

The Black-Scholes-Merton model assumes that the underlying asset S at some time t has dynamics governed by the following stochastic differential equation,

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{3.1}$$

where μ and σ denote the risk-free return and constant volatility of the underlying asset price (Black and Scholes, 1973). The price of the option can then be calculated by either using the PDE approach or the martingale approach. The martingale approach was later developed after the PDE approach. The martingale approach involves evaluating the expectation of the payoff as denoted by 3.4 under the risk-neutral measure. In mathematical terms the price of the option at time t under this approach is given by the following expression:

$$V_t = \mathbb{E}^{\mathbb{Q}}\left[\Phi(S_T)e^{-r(T-t)}|\mathcal{F}_t\right] \tag{3.2}$$

The $\mathcal{F}_t$ represents the history of the underlying asset price process up to time t and the expectation is under the risk-neutral measure as represented by $\mathbb{Q}$.

The solution to 3.2 provides the analytical solution popularly known as the Black-Scholes-Merton equation. Using the Feynman-Kac formula or evaluating the expectation using the density of $S_T|S_t$ it can be shown that this analytical solution is

$$D(t, S_t) = S_t N(d1) - Ke^{-r(T-t)}N(d2), \tag{3.3}$$

where

$$d1 = \frac{\ln(\frac{S_t}{K}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$
$$d2 = d1 - \sigma\sqrt{T-t}.$$

Alternatively, the price of the option can be found by first deriving the Black-Scholes-Merton PDE. The solution to the PDE then provides the actual price of the option with the boundary condition of the PDE being the payoff of the respective option. This is the PDE approach. We now illustrate the derivation of the Black-Scholes-Merton PDE.

To derive the PDE, we will make use of a no-arbitrage argument and some key assumptions. It is assumed that the underlying price process follows the dynamics

provided by 3.1, the asset pays a continuous dividend yield of q, and that the risk-less interest rate, r is constant. Consider now a European call option D on S with payoff,

$$\Phi(S_T) = (S_T - K)^+ \tag{3.4}$$

at maturity T. We construct a portfolio V at time t that consists of one call option, D, and a negative position in the underlying asset, S of Δ_t units. At the time t the value of the portfolio is

$$V_t = D_t - \Delta_t S_t \tag{3.5}$$

and after an instantaneous change in time, the value of the portfolio changes by

$$dV_t = dD_t - \Delta_t(dS_t + qS_t dt). \tag{3.6}$$

Using Ito's lemma we derive the dynamics of the option for a small change in t which results in the following equation:

$$dD_t = \frac{\partial D}{\partial t}dt + \frac{\partial D}{\partial S}dS_t + \frac{1}{2}\frac{\partial^2 D}{\partial S^2}d[S]_t, \tag{3.7}$$

where $d[S]_t = \sigma^2 S_t^2 dt$. If we substitute 3.7 into 3.6 then we get

$$dV_t = \left(\frac{\partial D}{\partial t} + \frac{1}{2}\sigma^2 S_t^2 \frac{\partial^2 D}{\partial S^2} - \Delta_t q S_t\right) dt + \left(\frac{\partial D}{\partial S} - \Delta_t\right) dS_t. \tag{3.8}$$

To make this a risk-less portfolio we need to re-balance continuously such that $\frac{\partial D}{\partial S} = \Delta_t$. This ensures that we lose the term that carries the stochastic component in 3.8. Following the no-arbitrage argument, if the portfolio is risk-less then it must earn the risk-less rate r over the period $[t, t + dt]$. Hence,

$$rV_t dt = \left(\frac{\partial D}{\partial t} + \frac{1}{2}\sigma^2 S_t^2 \frac{\partial^2 D}{\partial S^2} - \Delta_t q S_t\right) dt \tag{3.9}$$

and after re-arranging and substituting for $\Delta_t = \frac{\partial D}{\partial S}$ into 3.9 and 3.5 into 3.9 gives the following equation:

$$\frac{\partial D}{\partial t} + (r - q)S_t \frac{\partial D}{\partial S} + \frac{1}{2}\sigma^2 S_t^2 \frac{\partial^2 D}{\partial S^2} - rD = 0 \tag{3.10}$$

with boundary condition $D(T, S) = \Phi(S)$. The equation, 3.10 is known as the Black-Scholes-Merton PDE for a European call option.

In the next chapter, we will attempt to construct a toy example that will provide numerical solutions to equation 3.10. The solutions will then be benchmarked against the true analytical solution as represented by 3.3. As depicted by equation 3.1, the Black-Scholes-Merton model assumes that the volatility of the underlying asset price process is constant. This assumption will later be relaxed and we will consider incorporating a local volatility model under the same context.

3.2.1 Constructing the loss function

In this section, we derive the loss function that will be used to construct the BSM PINN. The notation in 2.3 will be maintained. To construct the PDE loss we will make use of 2.3 and 3.10. Define f to be the neural network's prediction of 3.10 and subsequently, the PDE loss on the collocation points is

$$\text{MSE}_{\text{PDE}} = \frac{1}{N_f}\sum_{i=1}^{N_f}\left(\hat{f}(t_i, S_i)\right)^2,$$

where $\{t_i, S_i\}_{i=1}^{N_f}$ is the set of randomly sampled collocation points. Similarly, the boundary loss is constructed by making use of 3.4 and it is given by the following expression:

$$\text{MSE}_{\text{bou}} = \frac{1}{N_{\text{bou}}}\sum_{j=1}^{N_{\text{bou}}}\left(\hat{u}(T, S_j) - \Phi(S_j)\right)^2$$

where $\{S_j\}_{j=1}^{N_{\text{bou}}}$ is a set of randomly sampled asset prices at the maturity of the option. The loss function that will be minimized during the training phase of the PINN to numerically solve 3.10 is then finally

$$\text{MSE}_{\text{total}} = \text{MSE}_{\text{PDE}} + \text{MSE}_{\text{bou}}.$$

We now discuss the local volatility model which is an alternative to the geometric Brownian motion that is assumed by the Black-Scholes-Merton model.

3.3 The option pricing PDE under the CEV model

The *constant elasticity of variance* (CEV) model is a local volatility model that was originally introduced by Cox (1975) as one of the first alternatives to the geometric Brownian model for modeling asset prices. The main characteristic of the CEV model is that it allows volatility to change with both time and asset prices. It also helps to capture the *leverage effect* [1] and volatility smiles which are usually observed in options markets. The asset price process is modelled as the solution to

$$dS_t = \mu S_t dt + \sigma S_t^{\alpha} dW_t, \tag{3.11}$$

with the instantaneous local volatility function given as

$$\sigma(t, S_t) = \sigma S_t^{\alpha-1}.$$

[1] The leverage effect refers to the tendency of the asset price volatility to be negatively correlated to the returns of the asset price.

Here α represents the elasticity parameter of the local volatility and σ is the scale parameter. The elasticity parameter controls the severity of the steepness of the volatility skew and the scale parameter fixes the at-the-money volatility level (Linetsky and Mendoza, 2009).

For $\alpha = 1$ the CEV model reduces to the geometric Brownian motion, for $\alpha = 0$ it reduces to a process similar to the Ornstein-Uhlenbeck process without mean reversion and the square-root model of Cox and Ross (1976) for $\alpha = 0.5$. These are the only cases for which analytical solutions are available for the stock price evolution. Cox (1975) originally studied the model for $\alpha < 1$ and under this range, the stock model produces results consistent with the leverage effect.

In the extension of this dissertation, we are particularly concerned with constructing a PINN to numerically solve the option pricing PDE under the CEV model. Using the same methodology that was used to derive the BSM PDE and replacing 3.1 with 3.11 we get the option pricing PDE

$$\begin{aligned} \frac{\partial D}{\partial t} + (r-q)S_t\frac{\partial D}{\partial S} + \frac{1}{2}\sigma^2 S_t^{2\alpha}\frac{\partial^2 D}{\partial S^2} - rD = 0 \\ \Phi(S_T) = (S_T - K)^+. \end{aligned} \tag{3.12}$$

We will also compare the solutions predicted by the PINN to the true analytical solution of equation 3.12. For the European call option, the analytical solution appearing in McWalter (2022) and also derived in earlier works such as in Glasserman (2004) is

$$C(S_t, K, \sigma, \alpha, T) = S_t\left[1 - \mathcal{P}(y; z, x)\right] - Ke^{-rT}\mathcal{P}(x; z-2, y),$$

where $\mathcal{P}(., d, \lambda)$ is the non-central chi-square distribution with d degrees of freedom and the non-centrality parameter λ and

$$x = \kappa S_t^{2(1-\alpha)}e^{2r(1-\alpha)T}, \qquad y = \kappa K^{2(1-\alpha)}, \qquad \text{and} \qquad z = 2 + \frac{1}{1-\alpha}$$

where

$$\kappa = \frac{2r}{\sigma^2(1-\alpha)(e^{2r(1-\alpha)T} - 1)}.$$

Chapter 4

Numerical results

We consider a European call option on a non-dividend paying asset as the underlying for our numerical implementation example. The goal is to develop a PINN that will numerically solve the BSM PDE as depicted in 3.10. The parameters of the market model and the underlying asset are as follows: a constant risk-free rate of 10%, annual volatility of 20%, and $q = 0$. The call option has an exercise price of R100 and it expires in one year. Additional parameters that will be used also include $S_{\min} = 80, S_{\max} = 120$. This is the bounded space for the underlying asset for which we will present the BSM PDE solutions from our PINN. We make use of Julia's SciML library Zubov *et al.* (2021) to develop our PINN as well as the training data that will be used in the training process [1].

4.1 Model Architecture of PINN

The first thing that was done was to create bounds for both space and time. We restricted the space domain between R80 and R120. This was done to particularly focus on asset prices around the strike to address the complexities that were discovered by Louskos (2020) in their paper. They discovered that the PINN produced inaccurate results for asset prices near the strike. The time domain was set between $t = 0$ and $t = T$. Next, we describe the process that was used to set up the PDE and generate the training data.

The PDE in 3.10 was set up on Julia using the Neural.jl and ModellingToolkit.jl packages. The Neural.jl package made it possible for us to specify the differential operators, the boundary condition, the parameters of the PDE as well as the variables under which we are trying to solve the PDE. The latter package allowed us to specify the domain of the space variable as well as time. The domain specification was important to ensure that the training data is generated within the bounds

[1] The code for all experiments conducted under this section can be found at https://github.com/ZAMXAKA/PINNs-dissertation-code---NZ.git

specified by the domain (Ma *et al.*, 2021). Next, we explain how the training data was generated.

We made use of the quadrature training technique to generate the training data as well as for evaluating both the PDE loss and the boundary loss terms. This training strategy was chosen as it is the only training strategy which produced a minimal loss function. The method uses *adaptive quadrature techniques*[2] to decide on a number of training points to be generated given an error estimate that was specified (Zubov *et al.*, 2021). In this instance, the integration refers to the evaluation of the PDE loss function and boundary loss function. Hence, the training strategy chooses pairs $(t_i,\ S_i)$ that result in an integral value less than the absolute tolerance/error specified by the user and this is how the training sets are generated. We now discuss other training strategies that are available in the Neural.jl package.

The GridTraining strategy discretizes the domain into a set of grid points. The disadvantage of this strategy is that the number of grid points scales as the dimension of the PDE increases. The strategy also suffers from the fact that the points that are not on the grid are not used during the training process. The StochasticTraining strategy allows for almost all points in the domain to be used during the training process. This strategy ensures that a fixed number of points are generated randomly in a given bounded interval. The QuasiRandomTraining improves the spacing between randomly sampled points by using low-discrepancy sequences. All these methods generate a fixed number of training points but unlike the Quadrature Training method, they do not ensure that the points generated produce a minimal integration approximation error (Zubov *et al.*, 2021).

The PINN was built using an input layer, one hidden layer, and an output layer. The idea was to build a simple toy example and illustrate that a simple PINN is capable of solving a PDE. The hidden layer made use of the sigmoid activation function in the respective neurons and the output layer made use of the linear activation function. The activation functions were chosen by trial and error and that was the combination which led to more palatable results. Each layer was constructed using 16 neurons. The next task was to initialize the parameters of the PINN. We made use of the Lux.jl package to set the parameters of the network to initial random values between -1 and 1.

Next, we used SciML's discretization method under ModelingToolkit.jl to transform the PDE that was specified into a problem ready to be solved numerically (Zubov *et al.*, 2021). It does this by transforming the PDE into a ModelingToolkit

[2] Adaptive quadrature technique is a numerical integration method which evaluates the integral of a function to a specified precision by continually dividing the integration into sub-intervals under which local quadrature methods are applied.

PDE interface. This is important so that the PDE is recognised as the loss function that which needs to be optimized under this environment. Before the application of the discretizer, the PDE could still be solved using other methods such as the finite-difference method or finite-element methods via DifferentialEquations.jl. We thus had to make the PDE ready to be solved by a PINN and this was taken care of by the PhysicsInformed NN discretizer which required the specification of the neural network structure and the training strategy. After the application of the discretizer which carries the neural network structure and the training strategy, the problem was now ready to be optimized.

The optimizer that was used in the training process was the Broyden-Fletcher-Goldfarb-Shanno(BFGS) optimizer (Zubov *et al.*, 2021). This optimizer was used in all rounds of training as it was the only optimizer that did not get stuck at a local minimum. Below we illustrate our first attempt and show the initial erroneous solution provided by the PINN. The error will later be fixed by implementing the methodology of Wang *et al.* (2021) that is explained in section 4.2.1. Additionally, it is important to note that the terms parametric and non-parametric will both be employed in a specific context. The parametric model will refer to the model under which σ and r are kept constant and the non-parametric model will refer to the model under which σ and r are variable.

4.2 Results and discussion

4.2.1 Parametric BSM PDE

In this section, we present the solution to 3.10 predicted by the PINN and compare it against the well-known analytical solution. The solution was calculated at distinct times as shown by the plots above in 4.1. The prediction of the PINN at all times is very inaccurate compared to the true solution of the PDE. This is also accompanied by a large total loss value of 899.121438 meaning that the convergence was very poor. Wang *et al.* (2021) in their paper provided insight into the reason why PINNs may fail to converge during the training process.

In their test example, they illustrated that the magnitudes of the gradients with respect to the weights of both the PDE loss and the boundary loss have scaling differences. The difference is caused by the more involved utilization of the chain rule in computing the gradient of the PDE loss as compared to that of the boundary loss (Wang *et al.*, 2021). In the example, they showed that the magnitude of the gradient of the PDE loss is much larger than that of the boundary loss and this causes the minimization of PDE loss to dominate the training process. The model essentially produces predictions that are biased in terms of reducing the error of the

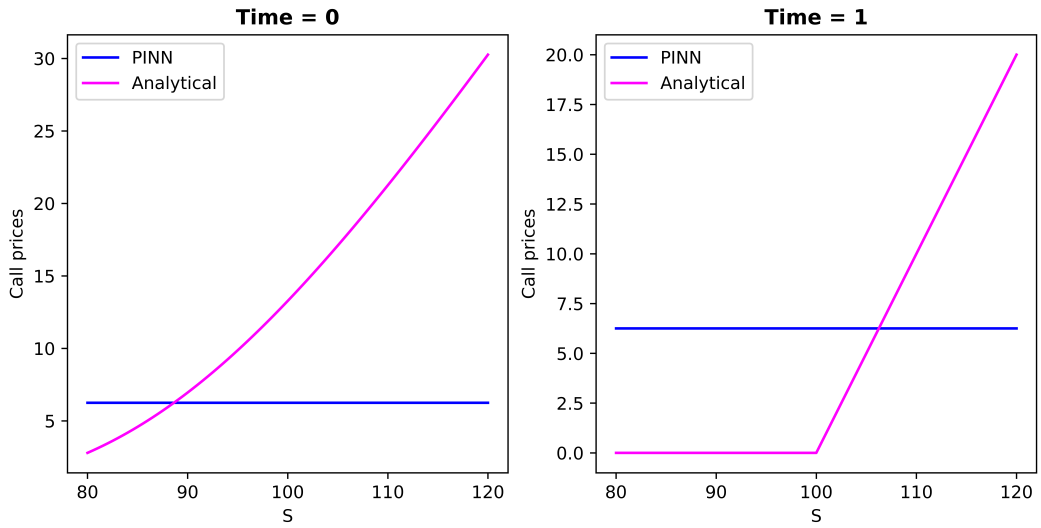

Fig. 4.1: Comparison of PINN call prices and analytical call prices over the range of S under the parametric model.

PDE at the expense of the boundary condition not being captured correctly (Wang *et al.*, 2021).

They propose a method to resolve the issue of imbalanced gradients by scaling the boundary loss by an arbitrary constant, λ. They further provided a detailed algorithm that calculates λ automatically using back-propagated gradient statistics during training. However, that is not the focus of the dissertation so we chose λ through trial and error. The table below shows how each scaling factor improved the convergence rate of the PINN during training.

Scaling factor, λ	**Total Loss**
1	899.12143
0.01	$2.447\text{x}10^{-3}$
0.05	$2.043\text{x}10^{-2}$
0.0555	$2.260\text{x}10^{-3}$

Tab. 4.1: Table of scaling factors vs total loss values.

Table 4.1 suggests that 0.0555 gives the least possible total loss value after the training process. This was the value that was then chosen to scale the boundary condition as represented by 3.4 in the previous section. This is similar to scaling the boundary loss function. Below, we illustrate how the scaling changed the predictions produced by the PINN.

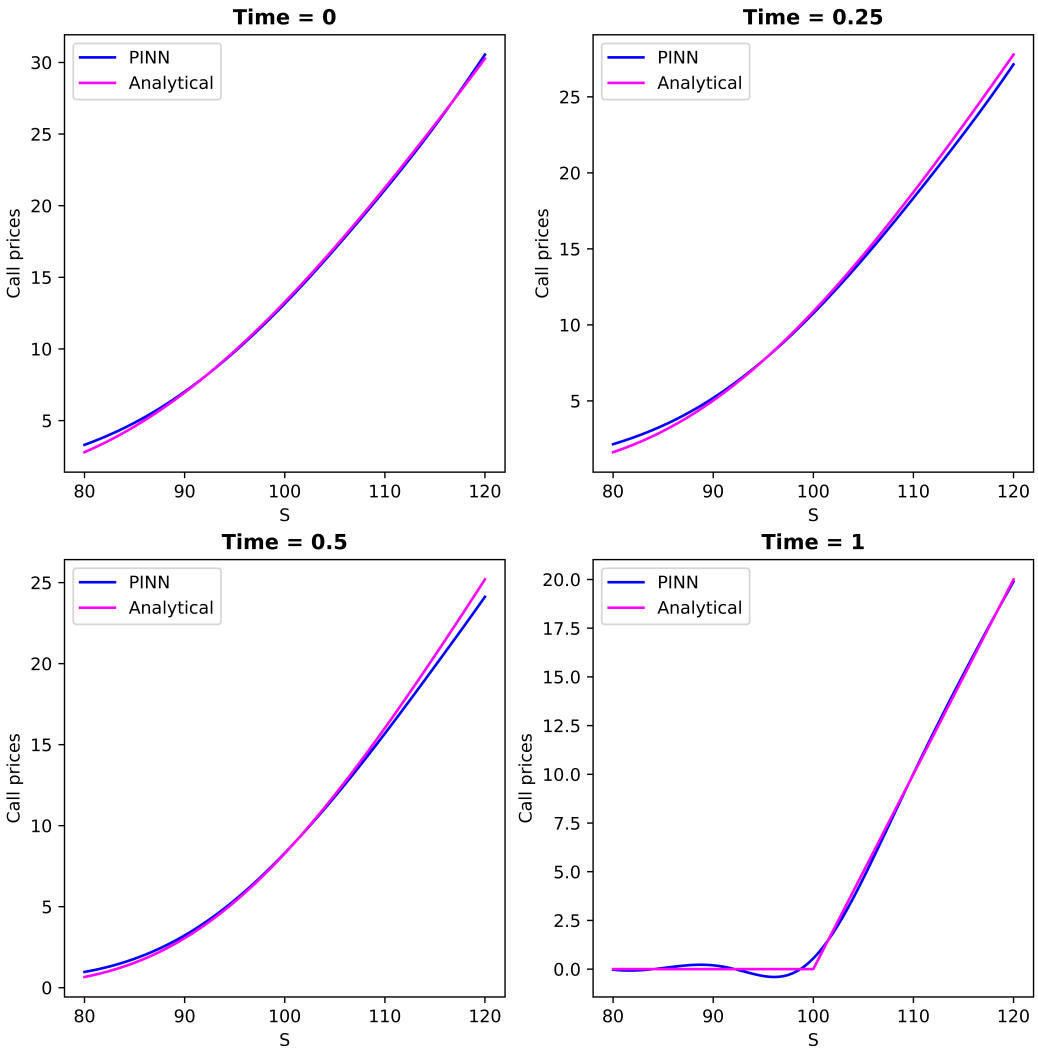

Fig. 4.2: Comparison of PINN call prices and analytical call prices over the range of S under the parametric model.

Figure 4.2 shows an improvement from figure 4.1. The solution to the BSM PDE predicted by the PINN seemingly matches the analytical solution for all stock prices between $S_{\min}$ and $S_{\max}$. It is also important to note that no fixed grid was specified a priori the training process. This shows that PINNs are able to predict accurate solutions by just training on random numbers within the specified range. Sirignano and Spiliopoulos (2018) proposed that PINNs are mesh-free and we have managed to show that in just a single round of training. Next, we illustrate the non-parametric solutions.

4.2.2 Non-parametric BSM PDE

In this section, we discuss the results obtained from the non-parametric model. We again developed a PINN to solve equation 3.10 but instead of using a fixed σ and r like how it was done in section 4.2.1, we defined a range for these parameters. This meant that the PINN needed four training sets, the additional sets being for both σ and r. The same range was chosen for these parameters. These sets were generated similarly to the training sets of both S and t. Thus r $\in (0.05, 0.15)$ and $\sigma \in (0.05, 0.15)$.

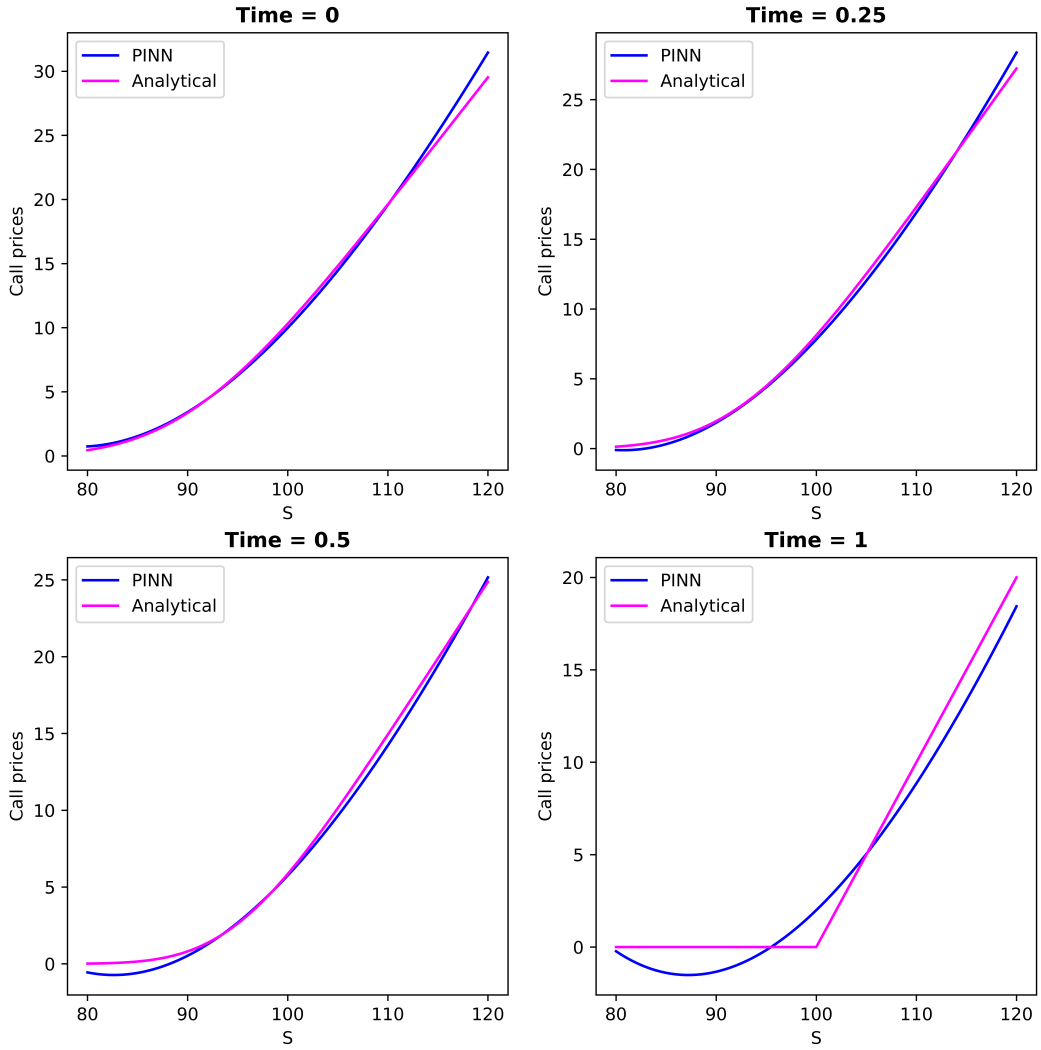

Fig. 4.3: Comparison of the PINN solution and the true analytical solution at chosen times for $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$ and $r = \frac{r_{\min}+r_{\max}}{2}$ under the non-parametric model.

Figure 4.3 demonstrates the comparison between the numerical solution of the PINN and the true analytical solution for specific times before maturity and at maturity. These solutions were calculated at $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$ and $r = \frac{r_{\min}+r_{\max}}{2}$. The PINN is now a function that takes in four variables being time, stock price, volatility, and the risk-free rate. This is why we had to decide on a level of σ and r to compute the solutions. The largest error was picked up at maturity as can be seen from figure 4.3 while the errors of the solutions at other times were minimal. We had again arbitrarily chosen a scaling factor of 0.001 and following the intuition of Wang *et al.* (2021), the chosen scaling factor may have led to a compromise of the boundary condition for increased accuracy of the PDE calculations. Figure 4.4 below shows the option prices predicted by the PINN as well as the analytical solution for all times before maturity and at maturity. Figure 4.4 further demonstrates

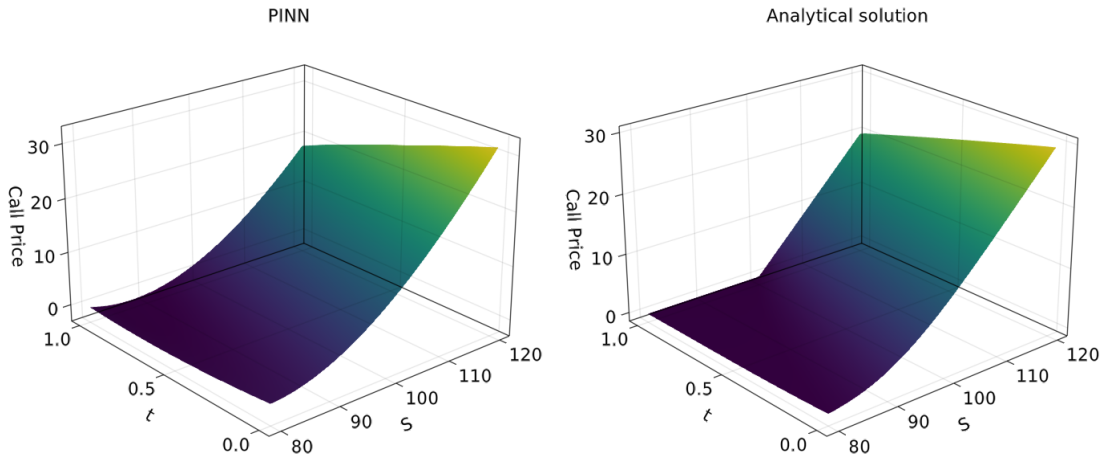

Fig. 4.4: Comparison of the PINN solution and the true analytical solution at all times for $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$ and $r = \frac{r_{\min}+r_{\max}}{2}$ under the non-parametric model.

the error at maturity as the shape of the two plots is different at $t = 1$. The right panel has a kinked shape at $t = 1$ to demonstrate that the call option is at-the-money at $S = 100$ and the left panel does not correctly capture this. Figure 4.5 shows the absolute error between the true analytical solution and the trial solution of the PINN. It can be seen that the errors are minimal at time zero but begin to increase as we approach maturity for all the stock prices in the chosen range.

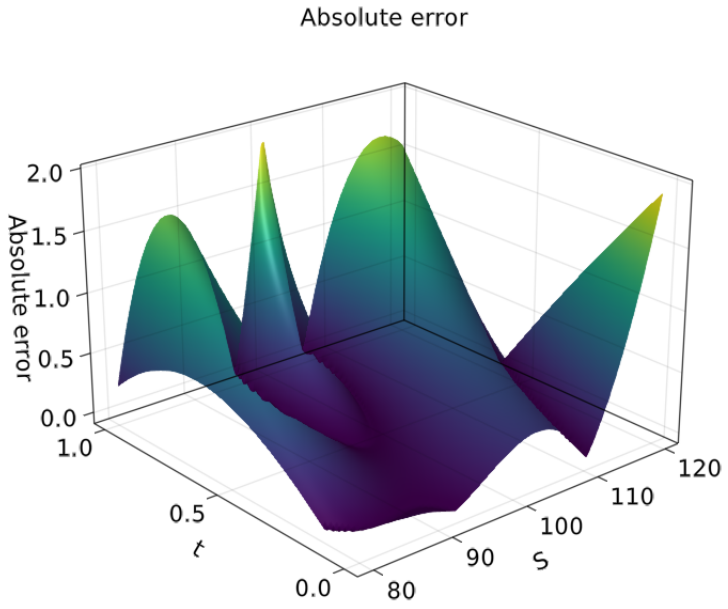

Fig. 4.5: Absolute error at all times for $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$ and $r = \frac{r_{\min}+r_{\max}}{2}$ under the non-parametric model.

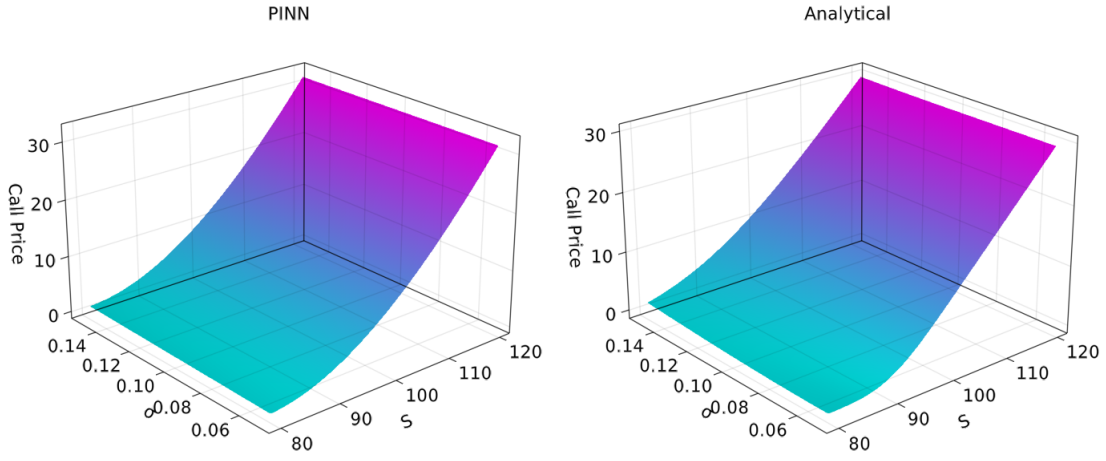

Fig. 4.6: Comparison of the PINN solution and the true analytical solution for $t = 0$ and $r = \frac{r_{\min}+r_{\max}}{2}$ under the non-parametric model.

Nevertheless, with a faster machine, errors could further be reduced by utilizing a variety of techniques to improve convergence. These tasks are not limited to increasing the training rounds, incorporating the automatic method of calculating the scaling factor proposed by Wang *et al.* (2021) as well as regularization techniques. Regularization is an algorithm added to the learning phase of a neural network to prevent it from over-fitting (Bishop and Nasrabadi, 2006).

Under the non-parametric model, we also explored comparing the solutions over the whole range of σ and r. Figures 4.6 and 4.8 demonstrate these comparisons over both these ranges. In figure 4.7, we see that the absolute error gets large as the volatility parameter σ increases. The error also increases as the *moneyness*[3] increases with the errors particularly large around the in-the-money range.

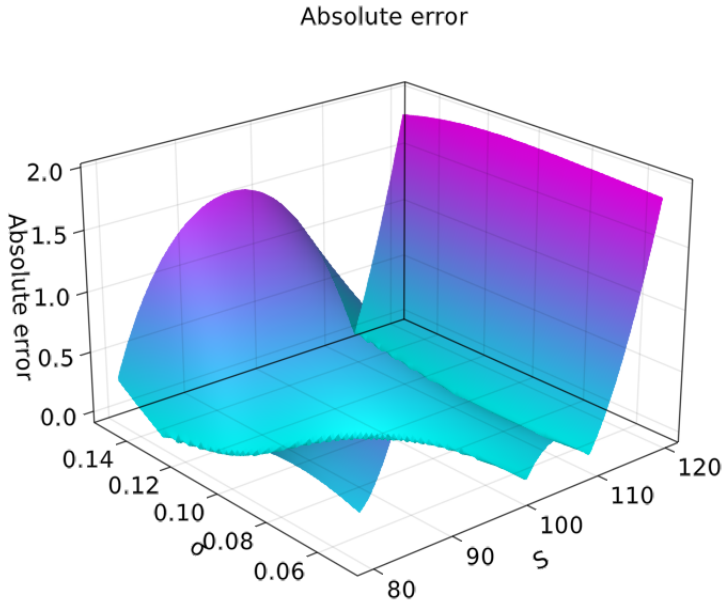

Fig. 4.7: Absolute error of the solutions for $t = 0$ and $r = \frac{r_{\min}+r_{\max}}{2}$.

Similarly, both figures 4.8 and 4.9 seemingly produce the same trend as figures 4.6 and 4.7. The errors are large around the in-the-money range as well as for r closer to $r_{\max}$. The largest error can also be seen to have occurred at $S = 120$. It is worth investigating the effect of moneyness on the PDE calculations made by the PINN. However, this is beyond the scope of the dissertation. We now proceed to the results of the CEV model.

[3] Moneyness in options refers to whether the intrinsic value of an option will lead to a profit or not.

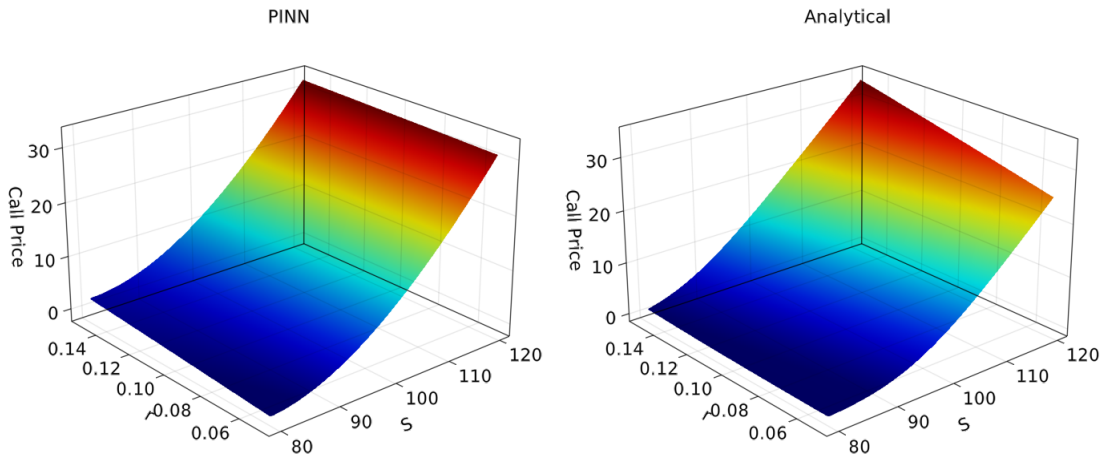

Fig. 4.8: Comparison of the PINN solution and the true analytical solution for $t = 0$ and $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$ under the non-parametric model.

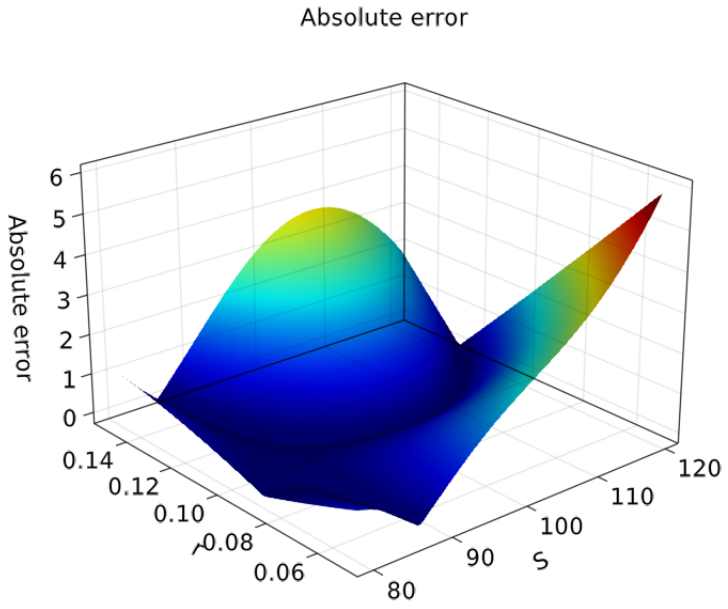

Fig. 4.9: Absolute error of the solutions for $t = 0$ and $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$.

4.2.3 Parametric CEV PDE

In this section, we present the numerical results of the option pricing PDE under the CEV model that was discussed in 3.3. Here, we attempted to construct a PINN example to numerically solve 3.12. The parameters and the structure of the model remain unchanged as in 4.1 with the exception of σ and the new parameter α. In the CEV model, σ is the scale parameter. In deciding to maintain the BSM volatility at time zero, we had to solve for σ in equation 3.3 and this resulted in $\sigma = 2$. The elasticity parameter α was set to be equal to 0.5 and this represents the square-root process of Cox and Ross (1976).

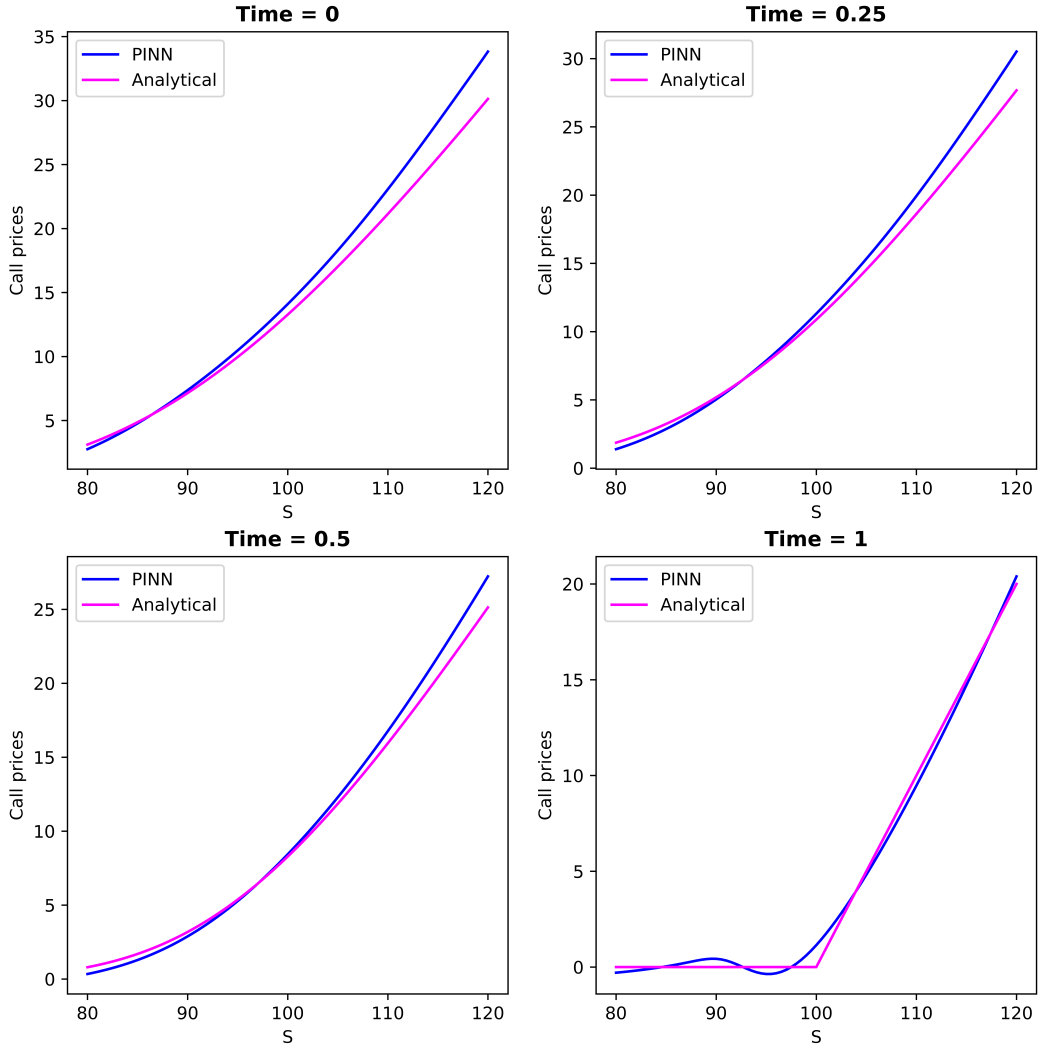

Fig. 4.10: Comparison of CEV PINN call prices and CEV analytical call prices over the range of S for scaling factor $= 0.04$.

In figure 4.10 we plot the solution to the option pricing PDE under the CEV model for both the analytical solution as well as predicted by the PINN. The solution of the PINN is more accurate at maturity. It means that the boundary condition was captured fairly well by the PINN at the expense of the accuracy of the PDE itself. Here, we used a scaling factor of 0.04. Below we also show the same results as depicted by 4.11 and this is when we tuned the scaling factor to 0.09.

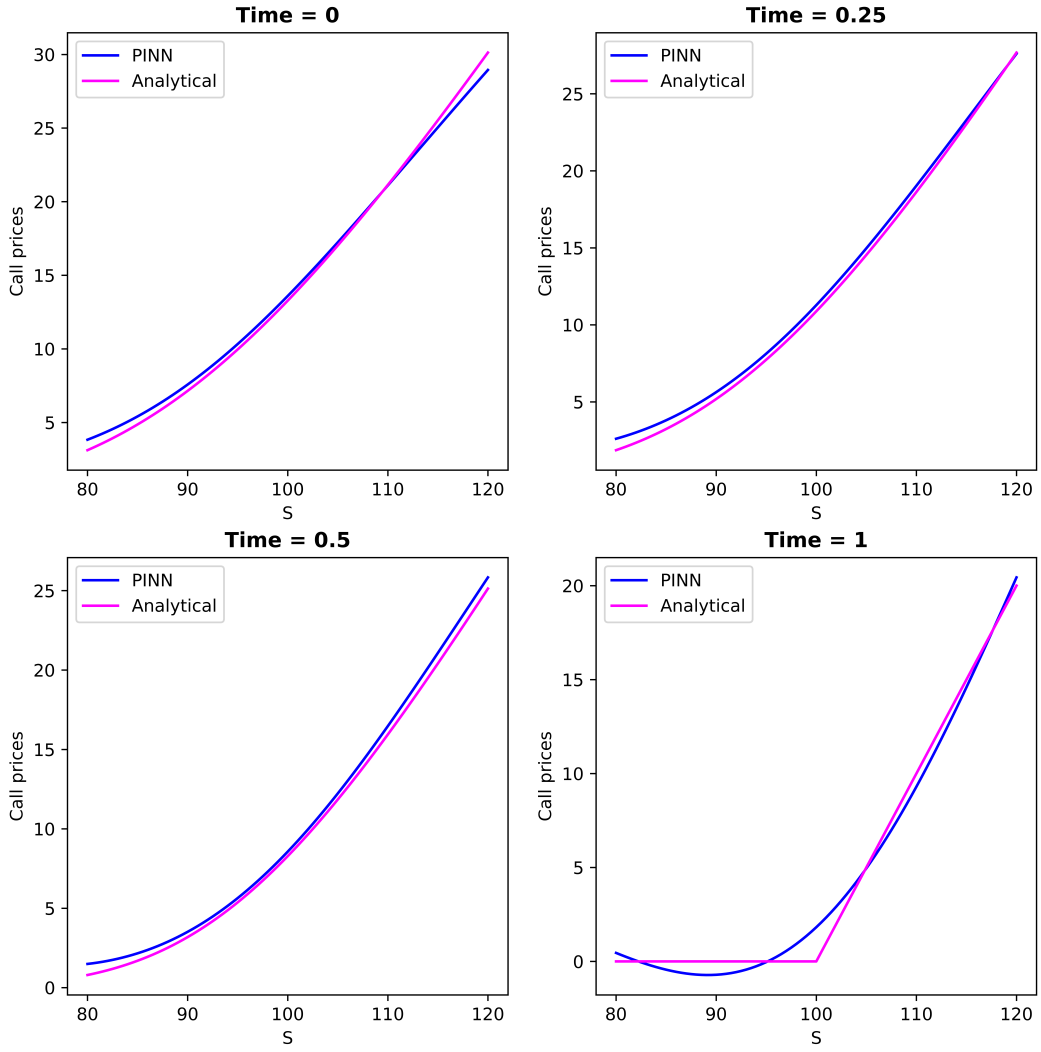

Fig. 4.11: Comparison of CEV PINN call prices and CEV analytical call prices over the range of S for scaling factor $= 0.09$.

Figure 4.11 shows an improvement at all the other times as compared to figure 4.10. However, the solution of the PINN at maturity is worse. This further shows that the theory developed by Wang *et al.* (2021) is worth investigating further as it seems to have an effect on the accuracy of the solutions produced by the PINN.

Nevertheless, we have shown the ability of the model to calculate the solutions in just one round of training with no difficulty. It was also conceptually easy to transition from the BSM PINN to the CEV PINN - all that had to be done was to introduce α and slightly change the PDE. In a traditional method such as the finite-difference method, the structure would have had to change more significantly. Next, we show the performance of the PINN under the non-parametric setting.

4.2.4 Non-parametric CEV PDE

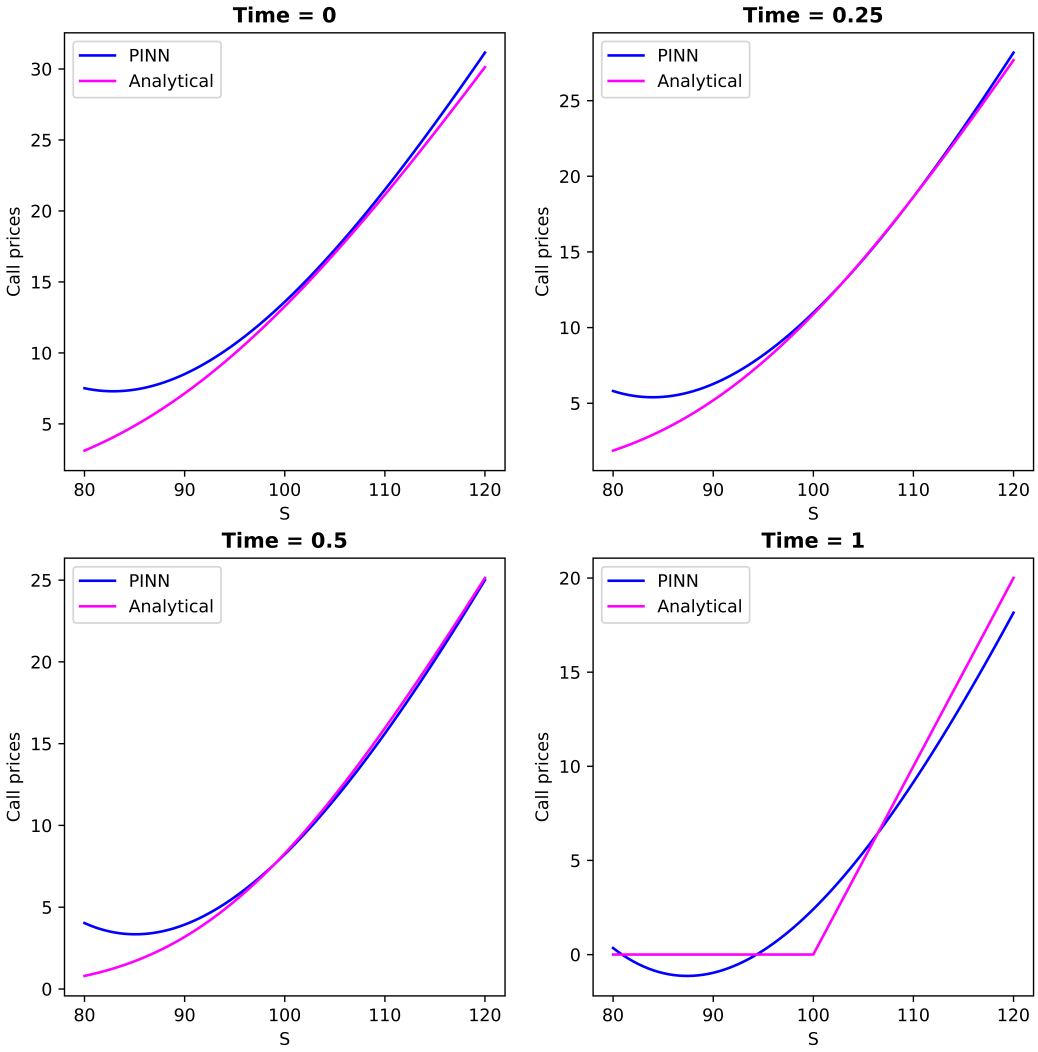

Fig. 4.12: Comparison of CEV PINN call prices and CEV analytical call prices over the range of S for $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$ and $r = \frac{r_{\min}+r_{\max}}{2}$ under the non-parametric model.

In this section, we considered relaxing the assumptions of the constant scale parameter, σ, as well as the constant risk-free rate. Analogous to 4.2.2, we defined a range for both σ and r. The same range was maintained for r and we let $\sigma \in (1.5, 2.5)$. We had to add an extra hidden layer with 16 neurons to the existing PINN structure that was constructed for the parametric CEV PDE and that was the only change that was made. The extra layer was added to improve the convergence of the PINN. Figure 4.12 shows the solutions calculated at $\sigma = \frac{\sigma_{\min}+\sigma_{\max}}{2}$ and $r = \frac{r_{\min}+r_{\max}}{2}$ and it can be seen that the largest error is maturity. The errors that are also visible are around the out-the-money region and this is apparent at all times.

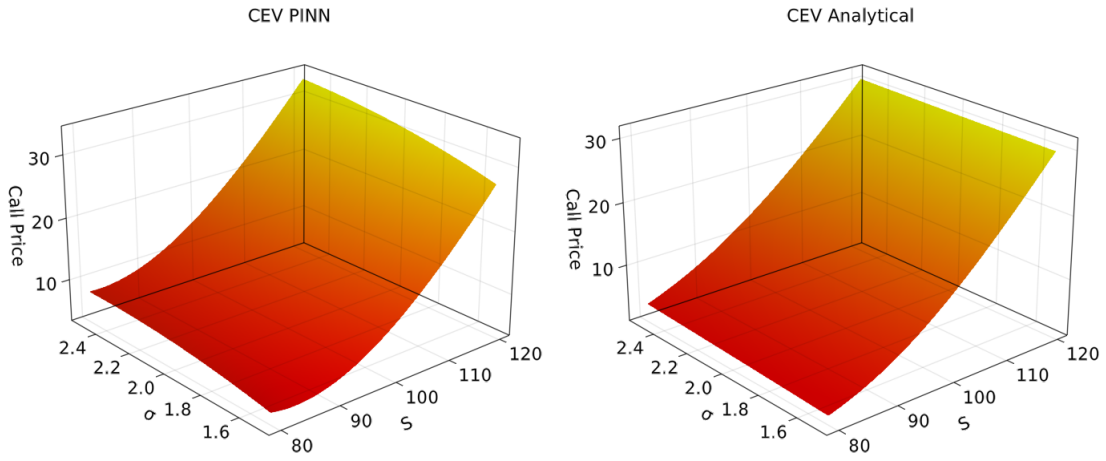

Fig. 4.13: Comparison of CEV PINN call prices and CEV analytical call prices over the range of S for $t = 0$ and $r = \frac{r_{\min}+r_{\max}}{2}$ under the non-parametric model.

The errors are much more visible around the out-the-money region in figure 4.13 and 4.14. Looking at 4.13, for all values of σ when the stock price is between 80 and 90, the two subplots look different. The analytical call prices in this region are far below 10 while the call prices of the PINN are much closer to 10. Figure 4.14 shows the absolute errors between the two subplots depicted by 4.13. It also corroborates the errors that were seen around the out-the-money region because the errors are very large at $S = 80$.

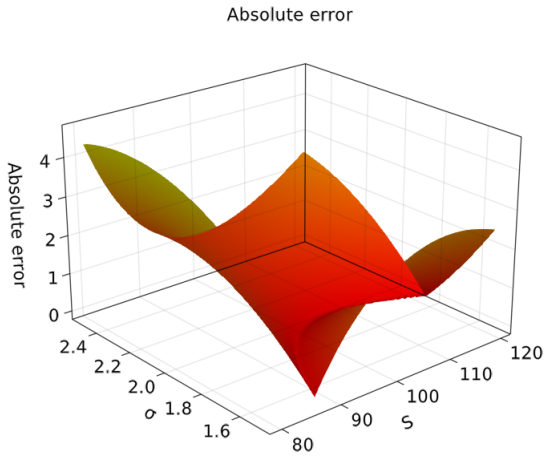

Fig. 4.14: Absolute error of the solutions for $t = 0$ and $r = \frac{r_{\min}+r_{\max}}{2}$.

Chapter 5

Conclusion

In this dissertation, we have formulated a physics-informed neural network (PINN) to solve the Black-Scholes-Merton PDE for a European call option. The PINN was constructed for both the parametric and non-parametric cases. We have shown that the PINN is mesh-free as no fixed grid and no step sizes were specified during the training phase. We have also shown that PINNs are more suitable for solving PDEs.

The partial derivatives were computed easily during the training phase and it was also easy to transition from the parametric to the non-parametric models. This showed that PINNs are able to overcome the curse of dimensionality. What remains to be done is to incorporate the algorithm proposed by Wang *et al.* (2021) to automatically generate the scaling factors of the boundary loss and PDE loss in every epoch. This will result in more accurate solutions and minimal errors.

Furthermore, we extended the option pricing problem by formulating a PINN to solve the option pricing PDE under the CEV model for a European call option. To our knowledge, this is the first dissertation to explore solving this particular PDE using the PINN methodology. It was reasonably easy to construct the PINN model to solve this PDE. This further enhanced the suitability of PINNs to be able to solve a wide range of PDEs without changing the whole structure of the network as opposed to classical neural networks.

Bibliography

Bishop, C. M. and Nasrabadi, N. M. (2006). *Pattern recognition and machine learning*, Vol. 4, Springer.

Black, F. and Scholes, M. (1973). The pricing of options and corporate liabilities, *Journal of political economy* **81**(3): 637–654.

Cox, J. (1975). Notes on option pricing i: Constant elasticity of variance diffusions, *Unpublished note, Stanford University, Graduate School of Business* .

Cox, J. C. and Ross, S. A. (1976). The valuation of options for alternative stochastic processes, *Journal of financial economics* **3**(1-2): 145–166.

Cuomo, S., Di Cola, V. S., Giampaolo, F., Rozza, G., Raissi, M. and Piccialli, F. (2022). Scientific machine learning through physics-informed neural networks: Where we are and what's next, *arXiv preprint arXiv:2201.05624* .

Dockhorn, T. (2019). A discussion on solving partial differential equations using neural networks, *arXiv preprint arXiv:1904.07200* .

Evans, L. C. (2010). *Partial differential equations*, Vol. 19, American Mathematical Soc.

Glasserman, P. (2004). *Monte Carlo methods in financial engineering*, Vol. 53, Springer.

Guo, Y., Cao, X., Liu, B. and Gao, M. (2020). Solving partial differential equations using deep learning and physical constraints, *Applied Sciences* **10**(17): 5917.

Han, J., Jentzen, A. and E, W. (2018). Solving high-dimensional partial differential equations using deep learning, *Proceedings of the National Academy of Sciences* **115**(34): 8505–8510.

Hull, J. (1993). *Options, futures, and other derivative securities*, Vol. 7, Prentice Hall Englewood Cliffs, NJ.

Lagaris, I. E., Likas, A. and Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations, *IEEE transactions on neural networks* **9**(5): 987–1000.

Linetsky, V. and Mendoza, R. (2009). The constant elasticity of variance model, *Unpublished note* .

Louskos, A. (2020). Physics-informed neural networks and option pricing, *Unpublished note* .

Ma, Y., Gowda, S., Anantharaman, R., Laughman, C., Shah, V. and Rackauckas, C. (2021). Modelingtoolkit: A composable graph transformation system for equation-based modeling.

McWalter, T. A. (2022). Numerical methods in finance ii, *Unpublished note, University of Cape Town, Finance and Tax Department* .

Raissi, M., Perdikaris, P. and Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, *Journal of Computational physics* **378**: 686–707.

Sacchetti, A., Bachmann, B., Löffel, K., Künzi, U.-M. and Paoli, B. (2022). Neural networks to solve partial differential equations: A comparison with finite elements, *IEEE Access* **10**: 32271–32279.

Sirignano, J. and Spiliopoulos, K. (2018). Dgm: A deep learning algorithm for solving partial differential equations, *Journal of computational physics* **375**: 1339–1364.

Wang, S., Teng, Y. and Perdikaris, P. (2021). Understanding and mitigating gradient flow pathologies in physics-informed neural networks, *SIAM Journal on Scientific Computing* **43**(5): A3055–A3081.

Zubov, K., McCarthy, Z., Ma, Y., Calisto, F., Pagliarino, V., Azeglio, S., Bottero, L., Luján, E., Sulzer, V., Bharambe, A. *et al. (2021). Neuralpde: Automating physics-informed neural networks (pinns) with error approximations,* arXiv preprint arXiv:2107.09443 .

Appendix A

Appendix A

A.1 Table of training times

Table A.1 below illustrates the training times of all four experiments that were conducted along with the total loss values after the training phase. It is important to note that the non-parametric models had a single round of training and the parametric models had two rounds of training. The times reported for the parametric models include both phases of training. We used 2500 epochs in every round of training and this was kept constant throughout the numerical experiments.

	Times	**Total Loss after training**
Non-parametric BSM PDE	98.68s	$2.23\text{x}10^{-3}$
Parametric BSM PDE	283.12s	$2.58\text{x}10^{-4}$
Non-parametric CEV PDE	86.19s	$9.88\text{x}10^{-3}$
Parametric CEV PDE	143.86s	$7.45\text{x}10^{-3}$

Tab. A.1: Table of training times in seconds.