Physics-Informed Neural Networks and Option Pricing

Andreas Louskos

This paper could not have been completed without the help of my advisor, Yoonsang Lee, and Joseph DiPalma. I am grateful for Professor Lee's tutelage and support throughout the project and allowing me to bridge my interests of pure and applied mathematics. Moreover, I am thankful for Joseph DiPalma's incredible help in systematically structuring the code and developing an easily accessible and understandable package for a physics-informed machine.

Abstract

We apply a physics-informed approach to the Black-Scholes-Merton equation for pricing American options. In this work, we use a data-driven solution approach proposed by Raissi et. al [20] to derive numerical solutions for the Black-Scholes-Merton equation. We compare our results with the values derived using the Brennan-Schwartz algorithm for pricing American options [6, 18, 27] . Broadly, the physics-informed models report mean squared error training losses lower than 10^{-4}, but the intricate non-linearity of the Black-Scholes-Merton equation still provides the model with difficulty in pricing American options. We vary the architecture and learning process for our original model to provide more understanding of convergence and stability issues that exist with physics-informed neural networks. Lastly, we address issues that occur with optimizers and learning rates in the original TensorFlow implementation by Raissi et. al [20], alongisde limitations in our own approach.

1 Introduction

Partial-differential equations (PDEs) describe the behavior of a plethora of dynamical systems in the world around us. While PDEs are extremely common and useful, they are equally hard to solve. Not all PDEs are guaranteed to have simple analytical solutions and we require the use of numerical analysis to approximate the solution values at various points in the parameter space. Traditional numerical analysis methods include standard forward and backward finite difference methods, finite element methods, and Runge-Kutta methods. While these conventional methods provide adequate solutions, they are computationally expensive. With the developments in our understanding of deep neural networks and improvements in automatic differentiation, researchers have looked to physics-informed neural networks (PINNs) to derive numerical solutions to PDEs.

1.1 Finite Difference and Element Methods

Before beginning a detailed discussion of a physics-informed approach for the Black-Scholes-Merton equation, it is important to understand typical methods for deriving numerical solutions to PDEs. The most used methods are the finite difference and element methods. Both methods approximate numerical solutions to PDEs by reducing the problem into smaller components that when solved will numerically approximate the solution to the PDE. Finite difference and element methods differ in their approach because the former utilizes Taylor's series approximation while the latter uses an integral approach to approximating the function [15, 29].

Consider a one-variable function, $f(x)$. When developing a finite difference approach, we are required to approximate the derivatives of f. Before we begin providing estimates for the derivatives of f, we need to divide the domain into uniform - also possible to be non-uniform - intervals of space. In higher-dimension PDEs, when the result of dividing the space, time, and other dimensions' domain is known as the *mesh* [14]. The interval size in each dimension is known as the step-size or mesh width. It is extremely important to properly define step-sizes when dividing the domain because, as will be discussed later, it can affect the implementation and accuracy of the numerical method [15].

Let h be our mesh width. Using this we can approximate the derivatives of f. To calculate the derivatives of f we can use three different finite difference methods: forward, backward, and centered approximations. Each method refers to whether we are moving forward or backward in each dimension or whether we are using a centered approach that does not imply any forward or backward "movement". Essentially, if we are interested in calculating the derivative of f at a point $\tilde{x}$, the centered approach will not use $f(\tilde{x})$ in each calculations but will use a value of f to the right and left of $\tilde{x}$. Consequently, the three different methods for approximating the first derivative of f are as follows [15],

$$\frac{\partial f}{\partial x}_{+} = \frac{f(x+h) - f(x)}{h} \tag{1}$$

$$\frac{\partial f}{\partial x}_{-} = \frac{f(x) - f(x-h)}{h} \tag{2}$$

$$\frac{\partial f}{\partial x}_{0} = \frac{f(x+h) - f(x-h)}{h}. \tag{3}$$

(1), (2), and (3) refer to the forward, backward, and centered approaches respectively. Notice how (1) and (2) use the value of $f(x)$ in their calculations while (3) does not, that is the biggest difference between the centered finite difference method and the other two methods. (1), (2), and (3) are known as *first order accurate* approximations and more accurate approximations exist, however, these are generally accepted if the step-size is sufficiently small [15]. Similar formulas can be derived for higher-order derivatives that can be used to numerically solve the PDE. It is important to understand that finite difference methods cannot approximate the function value at all points in the domain but only on the grid - mesh - points defined by our mesh width(s). Ultimately, substituting the derivative approximations into our PDE will provide us with a recursive system of equations that when solved using boundary and initial/final time information, provides us with our mesh values.

Finite difference methods are traditionally used to solve PDEs and have also been applied to options pricing [9, 10, 23]. Simple finite difference methods such as (1), (2), and (3) and their higher-level equivalents can be used for approximation but researchers have continuously sought to improve these basic numerical methods for more complicated PDEs, such as free boundary problems [10]. Moreover, while finite difference methods have been used to price American options, stability and convergence issues can arise from improper mesh generation [15] or domain definition [10]. The issues that arise from finite difference methods contributed to the development of a similar method known as the finite element method.

The finite element method is more abstract than the finite difference method because it uses calculus and linear algebra to transform the PDE into a more digestible problem. In general, the finite element method subdivides the continuum - the system defined by the PDE - into a finite set of smaller systems (elements) whose behavior is dictated by a fixed set of parameters [29]. Furthermore, it is known that the solution to the total system - the assembly of its individual components - operates under the rules as *standard discrete problems* [29]. Using this process and these properties, the finite element method divides the problem into smaller components whose solutions, when brought together, assist in solving the original problem. While the finite element method is equally as standardized as the finite difference method, for the purposes of this paper it is not necessary to provide a detailed overview of the generalized finite element method. Similar to the finite difference method, the finite element method and its variants have also been used to price American options [17, 28]. However, they also require a mesh, are more computationally expensive given their additional flexibility, and are more complicated approaches than traditional finite difference methods [29]. While both methods provide extremely accurate results, the need for mesh construction is limiting because if done improperly can lead to aforementioned stability and convergence issues. These limitations have motivated researchers to study physics-informed machines because they do not suffer from similar structural problems.

1.2 Deep Neural Networks and Automatic Differentiation

1.2.1 Perceptrons and Deep Neural Networks

The increase use of neural networks as function approximators [11] and numerical solvers provides a new frontier for machine learning and numerical analysis. Thus, for the purposes of this paper it is important to form a basic understanding of neural networks. With knowledge of how a single perceptron - single artificial neuron - makes predictions and is trained, we can develop intuition for the inner-workings of a deep neural network.

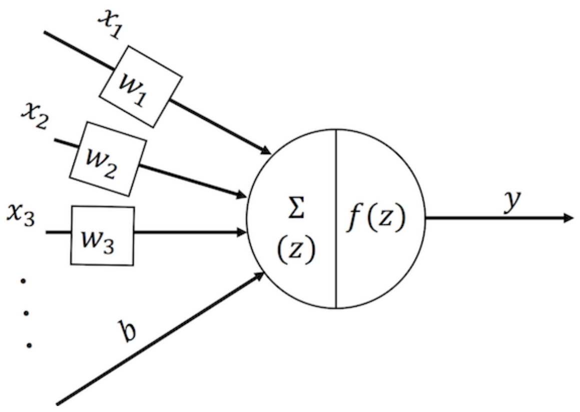

Figure 1: The structure of the perceptron. We compute the dot product between the inputs and their corresponding weights, add the bias, and then apply the activation function to the ouput to provide the final prediction.

The perceptron is the most basic neural network architecture. It consists of a single neuron and activation function [2]. Let $\boldsymbol{x}$ be the inputs to the perceptron, $\boldsymbol{w}$ be the corresponding weights, b be the corresponding bias, and g be the corresponding activation function. As Figure 1 illustrates, before applying g, the perceptron receives input $\boldsymbol{x}$, computes the weighted sum $\boldsymbol{w}\boldsymbol{x}^{\intercal}$, and adds the bias. Thus, for inputs $\boldsymbol{x}$, our perceptron predicts the following,

$$h(\boldsymbol{x}) = g\left(\boldsymbol{w}\boldsymbol{x}^{\intercal} + b\right), \tag{4}$$

where $h(\boldsymbol{x})$ represents the perceptron's output for input $\boldsymbol{x}$. The process for building more complicated neural networks is not considerably different.

The outline of a feed-forward neural network is provided in Figure 2. A layer in a neural network is defined as a collection of neurons that each individually process the inputs and provides a singular output. Consequently, the output of a layer is a vector whose dimensionality is equal to the number of neurons in that given layer. While only one layer can be used to make predictions, researchers often opt to make their neural networks "deep" to improve the flexibility and modeling capabilities of the network. When a neural network is referred to as "deep" it implies that there are multiple layers between the input and output layer [2]. Given that each neuron has a weight vector used to calculate its output, we create a "weight matrix" to mathematically describe the collection of weight vectors in each layer. The intuition behind using deep neural networks rather than rudimentary architectures is that lower layers identify low-level patterns in the data and as information is propagated through the neural network, higher layers are able to identify the high-level patterns in the data that are necessary for accurate predicting [2]. Equally important to the structure of the neural network is the process it undergoes to be trained and determine the optimal collection of weights for our problem.

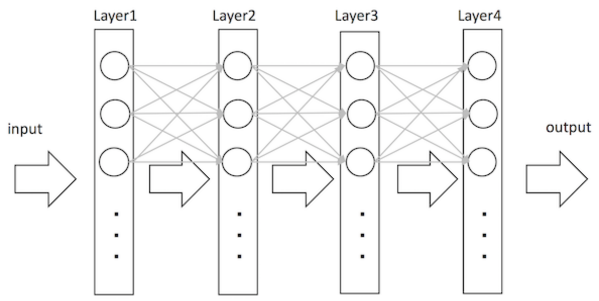

Figure 2: The standard structure for a sequential neural network. The inputs are provided to the first layer - collection of perceptrons - and each perceptron processes the inputs individually. After all perceptrons in the first layer have processed the inputs according to (4), then this output vector of the first layer becomes the input for the second layer. This process is repeated until the output layer.

1.2.2 Automatic Differentiation and Training

To train a neural network we need to provide it with a training set and a loss function. The training set is composed of training inputs and possibly corresponding target values for each corresponding input. If a training set includes target values for its inputs, the training process is known as *supervised learning* else it is referred to as *unsupervised learning* [2]. For the purposes of our paper, we will elaborate on the optimization process when target values are provided.

Since each task has a different set of appropriate loss functions, we aim to provide an overview of how an arbitrary loss function is optimized. The process for optimizing the weights on a neural network requires two steps: a forward and backward pass. In the forward pass, the inputs are provided to the neural network and propagated through the network, leading to a final prediction. After the prediction is made, the error is measured with respect to the true (target) value using the designated loss function. Next, the gradient of the loss function is computed with respect to each weight and the weights are adjusted according to the direction dictated by the gradient. The gradient dictates the "direction" in which the weights should be adjusted to minimize our loss function [2]. As this process is repeated, the weights are continuously updated and the loss function's gradient with respect to the weights is recalculated. Consequently, we expect to ultimately arrive to a collection of weights that sufficiently minimizes our loss function. The process of calculating gradients and adjusting the weights according to the gradient of the loss function is known as *stochastic gradient descent*. In machine learning it is common to scale the gradient by a factor η, also known as the learning rate, before the weights are adjusted. The learning rate controls the rate at which the model adapts to the problem [2]. While implementing stochastic gradient descent is not difficult, the process for calculating the gradient of the loss function with respect to the weights is cumbersome. To do so, we utilize automatic differentiation [3].

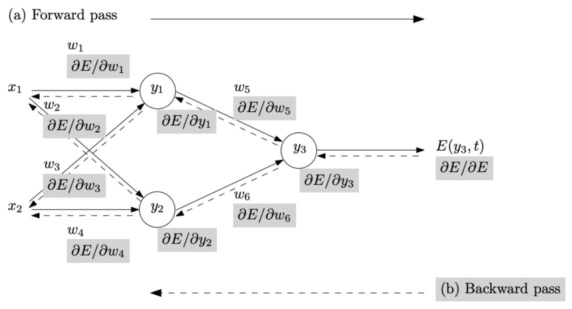
(a) Forward pass
w_1
$\partial E/\partial w_1$
x_1
w_2
$\partial E/\partial w_2$
w_3
$\partial E/\partial w_3$
x_2
w_4
$\partial E/\partial w_4$
y_1
$\partial E/\partial y_1$
y_2
$\partial E/\partial y_2$
w_5
$\partial E/\partial w_5$
w_6
$\partial E/\partial w_6$
y_3
$\partial E/\partial y_3$
$E(y_3, t)$
$\partial E/\partial E$
(b) Backward pass

Figure 3: Automatic differentiation numerically implements the properties of the chain rule for differentiation. Using these mathematical properties we are able to calculate the gradient of the loss function (E in Figure 3) with respect to the weights and carry out stochastic gradient descent.

Figure 3 accurately captures the aforementioned process of stochastic gradient descent. As was discussed, after the forward pass, we are required to calculate the gradient of the loss function with respect to the weights, known as the backward pass. The backward pass is dependent on the chain rule for differentiation. Let $f(y)$ be an arbitrary one-variable function and let y be a function of x. The chain rule for differentiation states that,

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial x}. \tag{5}$$

The entirety of automatic differentiation is based on (5). Consider an example from Figure 3. Suppose we are interested in calculating $\frac{\partial E}{\partial w_6}$. In order to do so, automatic differentiation proceeds by calculating $\frac{\partial E}{\partial y_3}$ and $\frac{\partial y_3}{\partial w_6}$ and applying (5) yields $\frac{\partial E}{\partial w_6}$. This process of automatic differentiation is propagated till the derivatives with respect all weights are calculated. This backwards propagation from the output layer to the input layer defines this process as the "backward pass" [3]. Having covered all necessary material needed for the purpose on this paper we can proceed to discuss physics-informed machines in detail.

1.3 Physics-Informed Neural Networks

PINNs were introduced by Raissi et. al [20, 21] and Lagaris et. al. [13] as alternatives for deriving numerical solutions to PDEs. This newly developed approach stems from deep neural network's capacity to be universal function approximators [11, 15] and continuous improvements in automatic differentiation [3] to calculate gradients of the neural network's outputs with respect to its inputs, help us model PDEs. PINNs improve our ability to derive numerical solutions for PDEs because by design they are required to respect any conservation principles or invariances that arise from the physical laws that dictate the observed data. As we briefly mentioned, PINNs extend our abilities in computational science and provide further resources for numerical solvers with improved data-efficiency [20, 21] that are *mesh-free* [1].

To begin our conversation on and analysis of PINNs, consider the following arbitrary partial-differential equation,

$$u_t + \mathcal{N}[u] = 0, \quad x \in \Omega, \quad t \in [0, T] \tag{6}$$

where u is the hidden solution of the system, $\mathcal{N}[\cdot]$ is a nonlinear differential operator parameterized by model parameter λ, and Ω is a subset of $\mathbb{R}^D$ [20]. PINNs can address two problems: data-driven solutions and data-driven discovery of PDEs. In the former case, the model parameter λ is fixed and we are interested in inferring the behavior of the hidden state $u(t,x)$. On the other hand, the latter case tries to determine what parameter λ best describes the observed data [20, 21]. Both approaches offer valuable insights, but this paper is motivated by the data-driven solution approach rather than its counterpart.

Before we proceed by explaining our approach and results, it is important to discuss the data-driven solutions approach and general implementation. Using (6), we define $f(t,x)$, as follows,

$$f \coloneqq u_t + \mathcal{N}[u] \tag{7}$$

where u is our neural network's prediction. To calculate the derivatives of u with respect to t and x, we will use automatic differentiation [3]. Both f, defined in (7), and u receive the same input values but differ in their activation functions given the nature of the differential operator $\mathcal{N}$ [20]. Additionally, a well-posed PDE, alongside providing a formulation of the form of (6), will provide information about initial or final time (possibbly both) and boundary conditions. To train our neural network, we will construct a loss function around the information provided by the PDE to be minimized.

During training, our goal is to minimize the following mean squared error loss,

$$MSE = MSE_{t_b} + MSE_{x_b} + MSE_f \tag{8}$$

where t_b and x_b are the boundary values for time and space, respectively, and f is defined in (7). Moreover, we calculate the individual terms of (8) as follows,

$$MSE_{t_b} = \frac{1}{N_{t_b}} \sum_{i=1}^{N_{t_b}} \left(u\left(t_b, x_{t_b}^i\right) - u_{t_b}^i\right)^2 \tag{9}$$

and

$$MSE_{x_b} = \frac{1}{N_{x_b}} \sum_{i=1}^{N_{x_b}} \left(u\left(t_{x_b}^i, x_b\right) - u_{x_b}^i\right)^2 \tag{10}$$

and

$$\mathrm{MSE}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(f\left(t_i, x_i\right)\right)^2 . \tag{11}$$

In (9), $\{x_{t_b}^i, u_{t_b}^i\}_{i=1}^{N_{t_b}}$ refer to the spacial inputs and function values on the time boundary t_b. Likewise, $\{t_{x_b}^i, u_{x_b}^i\}_{i=1}^{N_{x_b}}$ indicate the time inputs and function values on the space boundary x_b. Lastly, $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ denote the collocation points for $f(t,x)$ [20, 21]. Thus, from (9), (10), and (11) we can understand that by minimizing (8), we are improving the accuracy of our predictions on the time and space boundary and at collocation, indirectly helping us to correctly predicting the hidden state of the PDE, $u(t,x)$. This is why PINNs are classified as unsupervised learning. While we are improving our predictions of the hidden state we are not doing so by directly training on the function values. Using this information we can proceed to apply the theory to our specific problem.

2 Problem Setup

2.1 Background Information

The motivation of this paper comes from personal experience and interest in trading American options. Given that numerical solutions for American options are hard to derive, I was driven to apply the Raissi et. al's [20, 21] approach to the Black-Scholes-Merton equation for American option pricing. A call option is a financial derivative that provides the buyer the right, but not the obligation, to purchase an asset at a specific price within a given time period [25]. The opposite of a call option is called a put option. Both call and put options have two different types: European and American options. European options guarantee the buyer the option to purchase/sell the asset at the exercise price only at the expiration date, while American options allow the buyer to purchase/sell the asset at the exercise price at any time prior to and including the expiration date [25]. This discrepancy plays a significant role in whether we can derive analytical solutions to the Black-Scholes-Merton equation for either option type. Analytical solutions have been derived for both European call and put options, but none are known for American call or put options [25]. The biggest challenge for finding analytical solutions for American options is the ability to exercise the contract at any point in time within the given time period. While it can be argued that it is not optimal to exercise an American option prior to expiration, the option to do so results in the Black-Scholes-Merton equation for American options being harder to solve analytically [10].

Since the analytical solutions to the Black-Scholes-Merton equation for American options are unknown, applied mathematicians require the use of numerical analysis to derive solutions for the PDE. A common approach to deriving numerical solutions for pricing American options involves conventional finite difference and element methods alongside newly developed variants such as that proposed by Han et. al. [10]. However, traditional methods lack versatility and are computationally expensive to undertake. Moreover, when constructing a grid for a finite difference and element method, parameters and step-sizes need to be selected accordingly so no stability issues arise [14]. Collectively this suggests that PINNs are a possible avenue for deriving numerical solutions to the Black-Scholes-Merton PDE. To develop a PINN that would derive solutions to the Black-Scholes-Merton equation, it is necessary to understand the final time and boundary conditions alongside the residual (collocation) function in order to construct the loss function.

2.2 Black-Scholes-Merton Equation

Let S be the asset price, t be the time, C be the call option value, r be the risk-free rate, σ be the volatility, and D_0 be the continuous dividend yield. Given that the dividend yield does not drastically affect the numerical solutions of the PDE [25], this paper, for simplicity, will focus its effort on non-dividend paying assets, with modeling dividend paying assets left for an auxiliary further study. The Black-Scholes-Merton equation for an American call option states,

$$\begin{gathered}\frac{\partial C}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} + (r - D_0)\, S \frac{\partial C}{\partial S} - rC = 0 \\ 0 < S < S_f(t), \quad 0 \le t < T\end{gathered} \tag{12}$$

where $S_f(t)$ is the free boundary of early exercise. The final time and boundary conditions are given as follows,

$$C(T, S) = \max(S - E, 0), \quad 0 \le S \le S_f(T) = \max\left(E, \frac{rE}{D_0}\right), \tag{13}$$

$$C\left(t, S_f\left(t\right)\right) = \max\left(S_f\left(t\right) - E, 0\right), \quad \frac{\partial C}{\partial S}\left(t, S_f\left(t\right)\right) = 1, \quad 0 \leq t \leq T, \tag{14}$$

$$C\left(t, S\right) \to 0 \quad \text{as} \quad S \to 0, \quad 0 \leq t \leq T. \tag{15}$$

T denotes the time of expiration and recall that D_0 is the continuous dividend yield. If $D_0 = 0$ - as will be in the cases considered in this paper - then there is no free boundary (it becomes infinity) and it is not optimal to exercise the option prior to expiry [25]. However, for our PINN approach, the free-boundary is not considered, and the problem is approached as a traditional PDE with the Dirichlet boundary conditions described in (13) and (15). Since we are focusing our research to non-dividend paying assets, disregarding the free boundary should not limit our approach considerably.

To construct our deep neural network, we will construct our loss functions using conditions (12), (13), and (15) following the setup described in (8), (9), (10), and (11). However, we will also use a finite element method to derive the derivative value at each collocation point and measure the mean squared error loss of our predictions to the "true"[1] option values. As was discussed in Section 1.3, we expect that by minimizing the mean squared error loss on (12), (13), and (15), we will consequently develop accurate predictions for the hidden state $C\left(t, S\right)$ [20, 21]. The PINN approach does not differ for the Black-Scholes-Merton equation because instead of operating in the time and space dimensions, we are operating in the time and asset price dimensions. Raissi et. al. [20, 21] used a 2-dimensional problem with time and space coordinates, which we translate to a different 2-dimensional problem, in which space represents the asset's price.

It is important to mention that previous literature has applied neural networks to the Black-Scholes-Merton equation using more traditional supervised learning approaches [8, 16, 26] with few publications focusing on an unsupervised physics-informed approach [1, 22, 24] and its implication for future numerical solvers. The aim of this paper is to provide more detail on a Black-Scholes-Merton physics-informed approach while also looking into possibly stability or learning issues as the architecture and training method are varied. We aim to better understand physics-informed models and how training and model construction affects data-efficiency and accuracy. This will improve our understanding of future implications for and applications of these models.

2.3 Constructing the Loss Function

Let $u\left(t, S\right)$ represent our neural network's function predictions for inputs t and S. Likewise, let f represent our neural network's prediction for the constraint dictated by (12). To begin our PINN approach to a Black-Scholes, we need to construct our loss function using conditions (8), (9), and (10). Recall that the loss used for optimization will be the total of three individual mean squared losses based on those conditions. The first loss we focus on involves the collocation points. We define our function $f\left(t, S\right)$ using our definition in (7) and information from (12) and subsequently define our mean squared error loss on collocation as,

$$\text{MSE}_f = \frac{1}{N_f} \sum_{i=1}^{N_f} \left(f\left(t_i, S_i\right)\right)^2, \tag{16}$$

[1]We refer to the values derived from the finite difference method as the "true" values because we do not have an analytical solution for American options to use for reference.

where $\{t_i, S_i\}_{i=1}^{N_f}$ is the set of randomly sampled collocation points. Similarly, we now enforce condition (13) for the derivative value at expiration,

$$\text{MSE}_{exp} = \frac{1}{N_{exp}} \sum_{i=1}^{N_{exp}} \left(u\left(T, S_i\right) - C\left(T, S_i\right)\right)^2, \tag{17}$$

where $C\left(T, S_i\right)$ is previously defined. $\{S_i\}_{i=1}^{N_exp}$ denotes the set of points of randomly selected asset prices at expiration. The final step to completing the loss involves implementing boundary condition (15) for points when $S = 0$. Thus, it follows that we define our loss on the boundary as,

$$\text{MSE}_b = \frac{1}{N_b} \sum_{i=1}^{N_b} \left(u\left(t_i, 0\right)\right)^2. \tag{18}$$

Putting (16), (17), and (18), we define our total loss as,

$$\text{MSE}_{total} = \text{MSE}_f + \text{MSE}_{exp} + \text{MSE}_b. \tag{19}$$

We expect that by minimizing (19), our neural network will be able to accurately predict numerical solutions to the Black-Scholes-Merton equation [20, 21]. Before proceeding to implementing our method to an example and discussing the process, it is important to examine the use of a verification function before implementing our code on the specific problem.

2.4 Using a Verification Function

Before applying the physics-informed approach to the Black-Scholes-Merton equation, we want to evaluate whether our model can predict numerical solutions for a "known" solution with the same nonlinear differential operator but differing boundary and final time conditions. It is within our discretion to choose the "known" solution. For the purposes of this paper, we chose the function $h\left(t, S\right) = S^2 + t^2$, where S and t have been previously defined. The first step to verifying our algorithm is applying the nonlinear differential operator of (12) to the verification function. Doing so yields,

$$\mathcal{N}\left[h\right] = \left(\sigma^2 + r - 2D_0\right) S^2 - rt^2.$$

Putting everything together, results in,

$$h_t + \mathcal{N}\left[h\right] = 2t + \left(\sigma^2 + r - 2D_0\right) S^2 - rt^2. \tag{20}$$

Let $g\left(t, S\right)$ be the function defined in (20). To define our loss function for our verification function, we are required to have target values for the boundary, final time, and collocation. The boundary and final time target values can be directly computed using $h\left(x\right)$. In a similar manner, our collocation MSE loss is not measured with respect to 0 as the target variable for all collocation points but it is measured with respect to the value of g at every collocation point. The fomurlae losses are derived similar to (16), (17), and (18). Using these losses for our algorithm, we proceed to train the model. If the results on our verification function reach satisfying accuracy, then we will proceed to approach the required problem. If the results on our verification function do not converge and offer adequate results after thorough optimization, this is indicative of arbitrary problems in our algorithm that we will need to address. While the process of using a verification function does not eliminate future need for optimization and debugging, it offers a preliminary approach to addressing significant issues in our algorithm. Having discussed verification functions, we can approach the Black-Scholes-Merton equation.

3 Implementation Example on an American Call Option

For our implementation example we focus on an American call option on a non-dividend paying asset. The derivative has an exercise price of \$100, annual volatility of 40%, 3 years till expiration and the risk-free rate 3%. To develop a PINN to approach this problem we need to generate training inputs alongside target variables that will be used for evaluation but not for training. While the physics-informed PDE solvers are mesh-free [1], we will construct a mesh for the purpose of evaluation.

3.1 Data Initialization

To construct a mesh we are required to bound both time and space. For American call options, time is already bounded between $t = 0$ and $t = T$. However, the same is not true for the asset's price. Clearly $0 \leq S$ but we need to develop intuition for creating an artificial upper bound for S. Enforcing a theoretical maximum on the asset price does limit our approach, however, by setting the maximum bound sufficiently high, we would be operating within reasonable probabilistic outcomes. In our case, we set the maximum for the asset's price to \$500. Next, we define our time increment $\Delta t = 5 \times 10^{-3}$ and $\Delta S = 1$. To derive the asset value at each point, we have used code for implementing finite difference methods for American options from an external source [27]. The code implements the Brennan-Schwartz algorithm for pricing American options. The convergence and accuracy of the algorithm have been discussed in previous literature [6, 18]. We will use the values calculated using this algorithm as our "true" values for evaluation. As we previously mentioned, our approach involves unsupervised learning and these target values are only used for evaluation of convergence.

The next step in creating a training set for our neural network, involves generating the set of points that will be used for each loss function. We begin by sampling to create our set of collocation points. Given that we have bounded both t and S, we used PyTorch's builtin function for generating random integers. For the asset price we generate random integers in the range $[0, 500]$ while for t we have to marginally vary our approach to sampling. For t we uniformly select integer values in the interval $[0, 300]$ and then downscale the values by a factor of 100. The data sampling process is constructed such that all generated collocation points are located on our mesh. Similarly, for sampling asset prices at expiration and times at the boundary, we used the same methods as we used for generating collocation points. Similarly, from (13) and (15), we were able to calculate the true values of the function at each boundary. Figure 4 provides a visualization for the collocation points sampling distribution. It is clear that the distribution is not biased towards one area of the mesh and the samples are evenly distributed across the mesh. Having accurately sampled our inputs and simultaneously drawing our corresponding "true" values we can proceed to create, train, and evaluate our model on this specific problem.

3.2 Model Architecture and Loss

In this section we offer our an initial approach to the problem, with further data analysis and evaluation occurring in the following sections. Our model is initialized with 10 layers of 50 neurons in each. Every linear layer, excluding the output layer, is followed by the ReLU activation function. The weights for the neural network are initialized using normal He (Kaiming in Pytorch) initialization preserving the weights variance in the backward pass and with the gain adjusted for

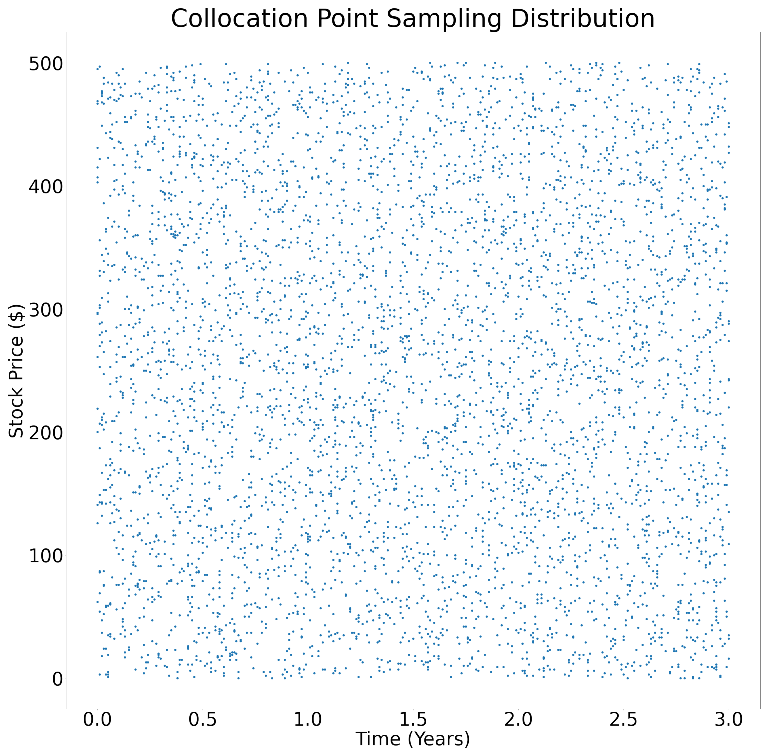

Figure 4: The collocation point sampling distribution for the American call with exercise price of $100, annual volatility of 40%, 3 years till expiration and the risk-free rate 3%. To generate the points, we used Pytorch's builtin function for uniformly generating integers within a configured range and scaled accordingly.

the ReLU activation function. Moreover, the bias vectors for each layer are initialized to the zero vector.

For training we use an Adam optimizer with two separate learning rates. In the first round of training we train the model using a learning rate of $\eta = 8 \times 10^{-3}$ for 710 epochs. In the second round of training we train the model using a learning rate of $\eta = 1 \times 10^{-3}$ for 4700 epochs. During both rounds of training, we use the loss functions defined in (16), (17), and (18), however, we overweight (16) by a factor of 100. We do so because during experimentation, we noticed that the magnitude of losses (17) and (18) were much larger than that of (16) and the neural network did not focus on and consequently not improve the collocation loss. Thus, as is commonly accepted [2], we decided to overweight the collocation loss.

Using this process we trained a model to develop a preliminary approach to a physics-informed Black-Scholes-Merton model. It is noteworthy to mention that the training data was shuffled during every epoch to prevent the model from improving predictions based on the data order rather than the underlying patterns that govern the data. The training set consisted of 100 instances for the asset price boundary, 1,000 for expiration, and 10,000 for collocation. After this process, we trained an initial physics-informed model to predict solutions to the Black-Scholes-Merton equation.

3.3 Initial Training Loss and Results

Epoch	MSE Expiration	MSE Boundary	MSE Collocation	MSE Function
0	2.43×10^5	2.00×10	1.86	2.39×10^5
250	5.30×10^{-3}	5.30×10^{-3}	7.86×10^{-1}	5.47×10^2
500	1.04×10^2	2.00×10^{-4}	02.46×10^{-2}	6.80×10
700	2.37	$10^{-4} >$	7.40×10^{-2}	2.12×10

Table 1: MSE Results from the first round of training. The losses at certain epochs are provided to show convergence. After the first round of training, the model underwent a second round of training with a lower learning rate to further optimize the model. All values have been rounded to 4 decimal places.

Epoch	MSE Expiration	MSE Boundary	MSE Collocation	MSE Function
0	8.01	$10^{-4} >$	7.60×10^{-3}	3.60×10
1000	6.61×10^{-2}	$10^{-4} >$	4.50×10^{-3}	2.60×10
2000	3.65×10^{-1}	$10^{-4} >$	5.10×10^{-3}	2.79×10
3000	2.28	$10^{-4} >$	5.00×10^{-3}	3.08×10
4000	10^{-2}	$10^{-4} >$	4.90×10^{-3}	2.57×10
4700	5.69×10^{-2}	$10^{-4} >$	4.70×10^{-3}	2.67×10

Table 2: MSE Results from the second round of training. As is evident from the data, the model converges to an acceptable MSE on all three training functions alongside the MSE on the function that we used for evaluation.

From Table 1 and 2, it is clear that the model converges to an acceptable minimum. The stopping points for each round of training were determined after running the model for an extensive amount of epochs and deciding whether added epochs significantly improved model convergence or not. To further evaluate the model and develop a better understanding of its predictive behavior, we also looked at the behavior of our neural network at $t = 0.0$, $t = 1.5$, and expiration. Evaluating the data at $t = 0.0$ in Figure 5, it is noticeable that for asset prices near the exercise price, the model has more difficulty accurately predicting the option value as opposed to prices that are farther from the exercise price. This pattern is also observed in Figure 6, for the option prices at $t = 1.5$. Given that for a call option to be in the money the asset price must be at least as large as the exercise price, for asset prices near the exercise price the value of the option will behave more erratically given that small changes in the asset prices can result in a change in the moneyness[2] of the option. However, for $t = 1.5$, it is evident that the interval of asset prices for which the neural network is having difficulty pricing the option is tighter. That is most likely a result of the difference in the intrinsic time value of the two options. The option at $t = 0.0$ has more added time than the option at $t = 1.5$. This added time value and its relationship with the option value could be making it more difficult for the neural network to correctly price the option.

[2]Moneyness refers to the relationship of the asset price and the option's exercise price. For example, if the asset price is higher than the exercise price for the call option, the option is considered to have high moneyness.

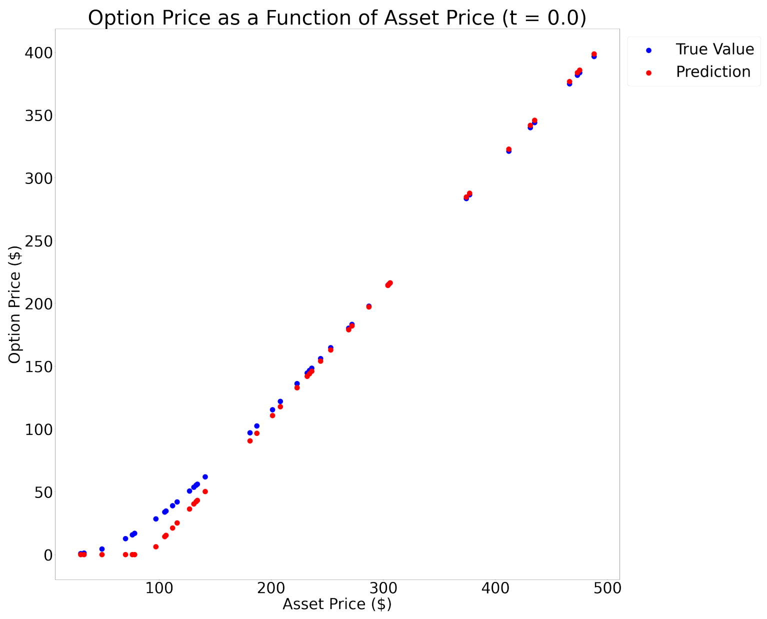

Figure 5: The model predictions for the option price as a function of asset price at $t = 0.0$.

However, looking at Figure 7, we notice that the model accurately captures the behavior of the Black-Scholes-Merton equation. The MSE reported at expiration was less than 1×10^{-4}. The robustness of the method appears in constraint (12). Many functions can satisfy conditions (13) and (15), however, our loss function penalizes functions that do not satisfy the Black-Scholes-Merton equation, acting like a regularization mechanism [20]. This allows us to use small data sets to train our model, which is favorable considering the difficulty of obtaining large amounts of data for physical systems [19]. Using this model as a starting point, we will proceed by varying the size of the model's architecture, training sets,and other parameters to provide a more detailed assessment on the behavior of our neural network. From Tables 1 and 2 and Figures 5, 6, and 7, it is evident that the model is able to capture the intricate non-linear behavior of the Black-Scholes-Merton equation. However, it may require added complexity to more effectively learn the behavior of the solution near the exercise price given that is where the model had the most difficulty offering accurate predictions. Further experimentation will make more clear what parameters contribute the most to the model's ability to capture the complex nature of option prices.

3.4 Optimization and Evaluation

3.4.1 Number of Layers and Neurons

One of the most common approaches to improving the complexity of the neural network is increasing the number of layers. While we can also increase the number of neurons per layer, it is commonly accepted that increasing the number of layers rather than the number of neurons per layer will be more rewarding [2]. However, given the complexity of physics-informed machines, we

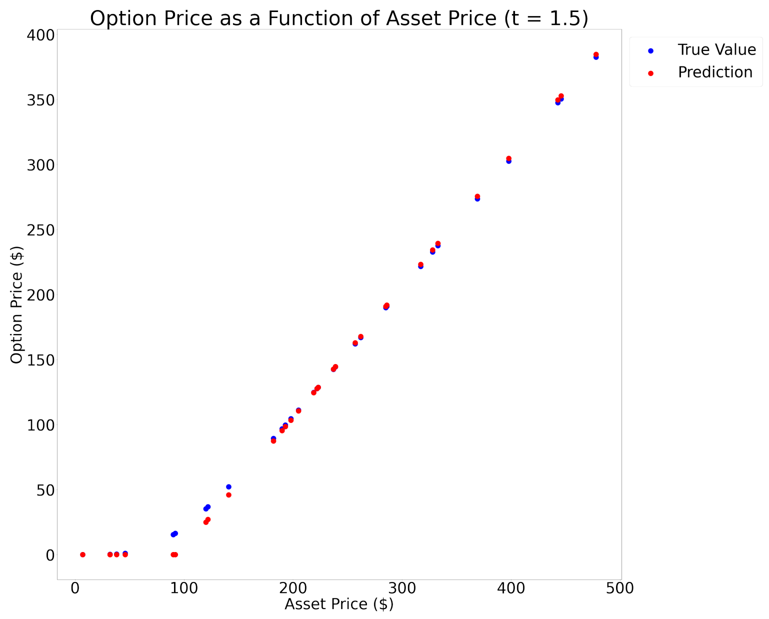

Figure 6: The model predictions for the option price as a function of asset price at $t = 1.5$.

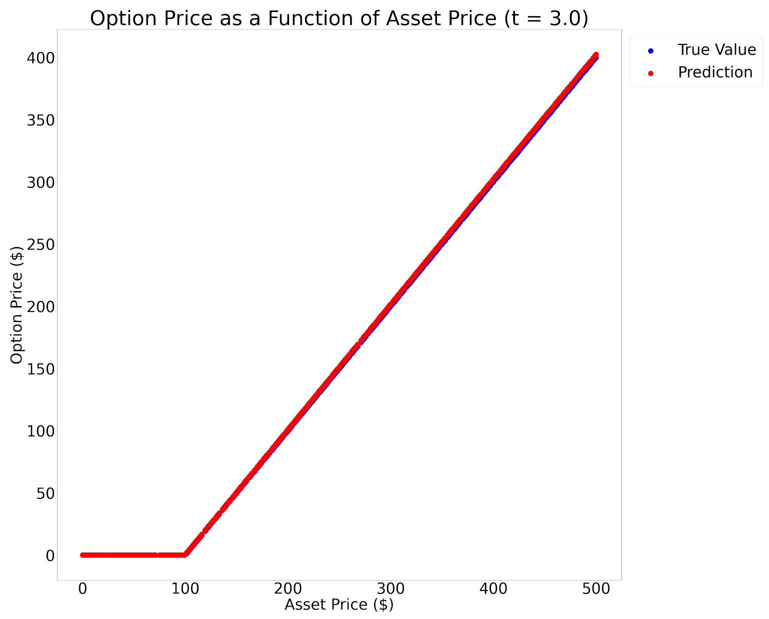

Figure 7: The model predictions for the option price as a function of asset price at expiration. From Table 2 we know that the MSE at expiration is less than 1×10^{-4} after our two rounds of training.

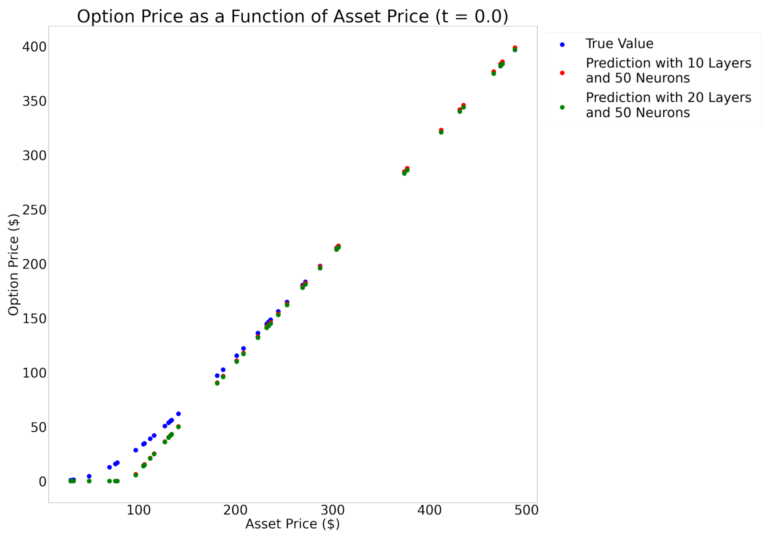

Figure 8: The models predictions for the option price as a function of asset price at $t = 0.0$. We are comparing the predictions made by our 10 layer iteration and 20 layer iteration of the model. Based on the figure, there appears to be no significant change in the model's predictions with the added complexity.

will vary both architectural components and reevaluate the model. In this section our goal is to determine the optimal number of layers and neurons for which the model achieves high accuracy but not at the cost of high complexity.

For the first step in our evaluation, we increased the number of layers to 20 and retrained the model. The model's training schedule[3] was slightly modified from the previous iteration with the first round of training running for 650 epochs with a learning rate of $\eta = 8 \times 10^{-3}$. In the second round of training the model ran for 2000 epochs with a learning rate of $\eta = 10^{-3}$. The training schedule modification does not limit our comparison of the two iterations of the model because varying the model's architecture varies the stochastic gradient process. Therefore, when discussing and comparing our models, we are considering the final predictions each model makes after its respective training schedule. After its training schedule, the 20 layer iteration reports 5.60×10^{-2} loss at expiration, less than 10^{-4} loss at expiration, 3.60×10^{-3} loss at collocation, and 25.54 loss on the function values. This is comparable with our initial model, but provides marginally lower values for all losses. While the model offers added accuracy, the improvement in accuracy is not significant, as can be assessed in Figure 8, to justify the added complexity of doubling the number of layers. For more a detailed comparison of the 20 and 10-layer model for the other time value refer to Figure 9. We exclude expiration from our analysis because the original model performs well on the expiration dataset with the new models performing negligibly different at expiration. To further assess how the added complexity contributes to the model's behavior, we further increased the number of layers from 20 to 30.

The results from the 30-layer model are not noticeably different from the previous two models' results. The 30-layer model reports 5.7×10^{-3} loss at expiration, less than 10^{-4} loss on the

[3]During experimentation, we determined that 8×10^{-3} was an appropriate learning rate for the first round of training and, likewise, 10^{-3} was an adequate learning rate for the second round of training. We will separately investigate the learning rate and its relationship to the model's training stability.

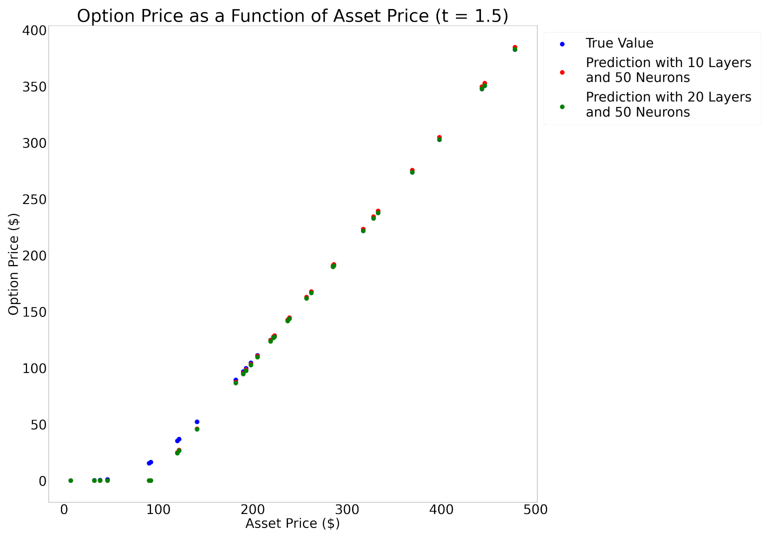

Figure 9: The models predictions for the option price as a function of asset price at $t = 1.5$. We are comparing the predictions made by our 10 layer iteration and 20 layer iteration of the model. It clear that similar to Figure 8, the added complexity of the model does not offer serious improvements to predictions.

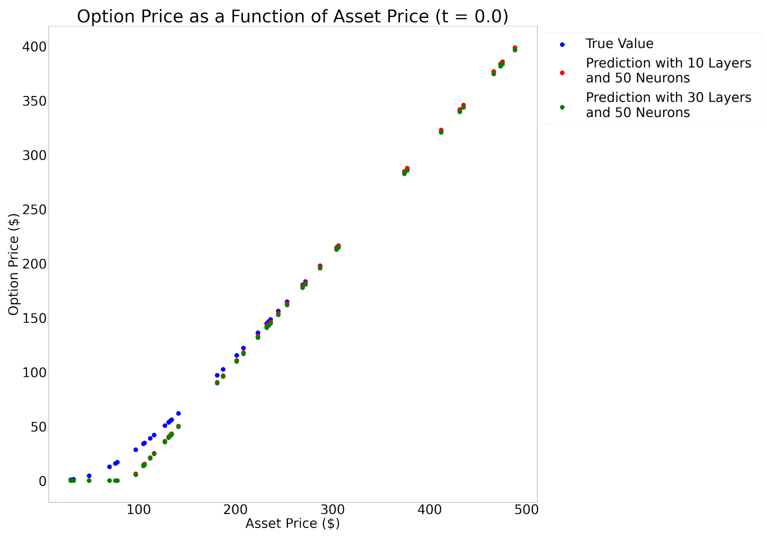

Figure 10: The models predictions for the option price as a function of asset price at $t = 0.0$. We are comparing the predictions made by our 10 layer iteration and 30 layer iteration of the model.

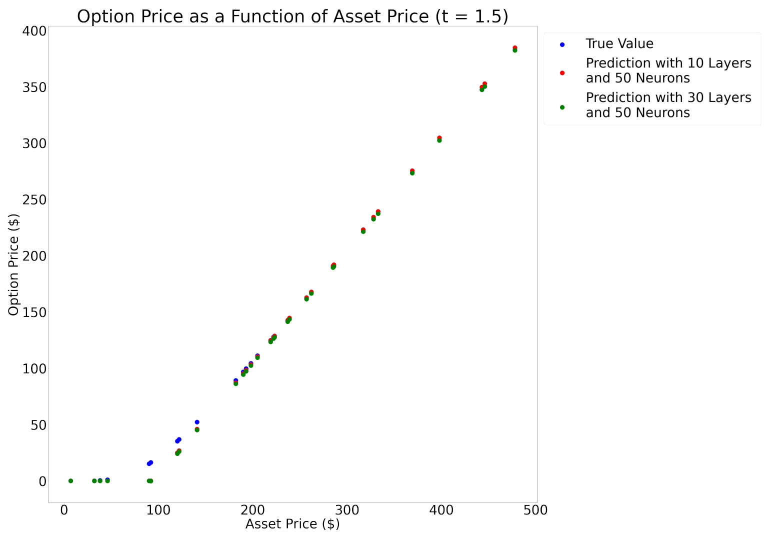

Figure 11: The models predictions for the option price as a function of asset price at $t = 1.5$. We are comparing the predictions made by our 10 layer iteration and 30 layer iteration of the model.

boundary, 4.60×10^{-3}, and 26.19 loss on the function values. The model trained for one round for 1510 epochs with a learning rate $\eta = 8 \times 10^{-3}$. These metrics remain on par with the previous two models further validating that the added complexity of the model does not significantly improve the model's predictive power. Referring to Figure 11, we compare the original and deeper model furthering the argument against added complexity. For added assessment refer to Figures 12 and 13. Given our results with the 20 and 30-layer models, it is unnecessary to make the model deeper because there will be no noticeably added benefit to its predictive capabilities. To proceed with our analysis, we will continue by varying the number of neurons alongside the number of layers to see if a two-dimensional complexity increase may prove more fruitful.

We proceed by increasing the number of neurons per layer from 50 to 75 while maintaining the original 10 layer structure. This neural network architecture reports 2.20×10^{-2} loss at expiration, less than 10^{-4} loss at expiration, 6.50×10^{-3} loss at expiration, and 26.21 loss on the function values. The model only underwent one round of training for 1000 epochs with learning rate $\eta = 8 \times 10^{-3}$. Similar to the previous models, this iteration does not offer additional predictive capacity and is similar to the previous models. A more comprehensive analysis is offered in Figures 12 and 13. Similar to when we were varying the number of layers in the model, we do not provide the plots for expiration because there are marginal improvements in the model's predictions. To continue our study of our physics-informed machine, we use a 20 layer architecture with 75 neurons per layer. This new iteration, similar to previous ones, does not offer a significant improvement to the model. It reports 8.00×10^{-3} at expiration, 4.00×10^{-4} at the boundary, 9.00×10^{-3} at collocation, and 25.74 loss on the function values. This is extremely similar to previous results offering minimal difference between this iteration of the model and previous iterations. This implies that the added complexity both with respect to the number of neurons and layers is not conducive to improving the model. Even though a deeper model does not offer better predictions, the convergence to acceptable predictions of multiple models further validates the potential of physics-informed neural networks. Our next step in evaluation of the physics-informed machines will involve varying the size of the

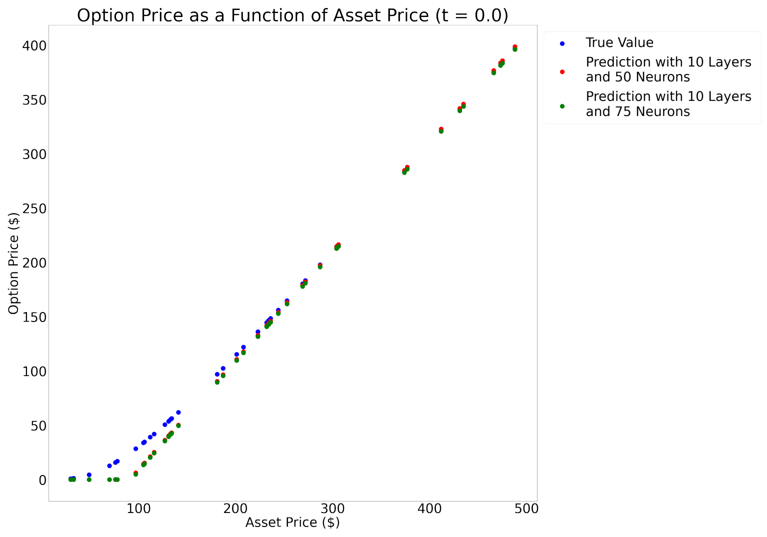

Figure 12: The models predictions for the option price as a function of asset price at $t = 0.0$. We are comparing the predictions made by our 50 neuron iteration and 75 neuron iteration of the model.

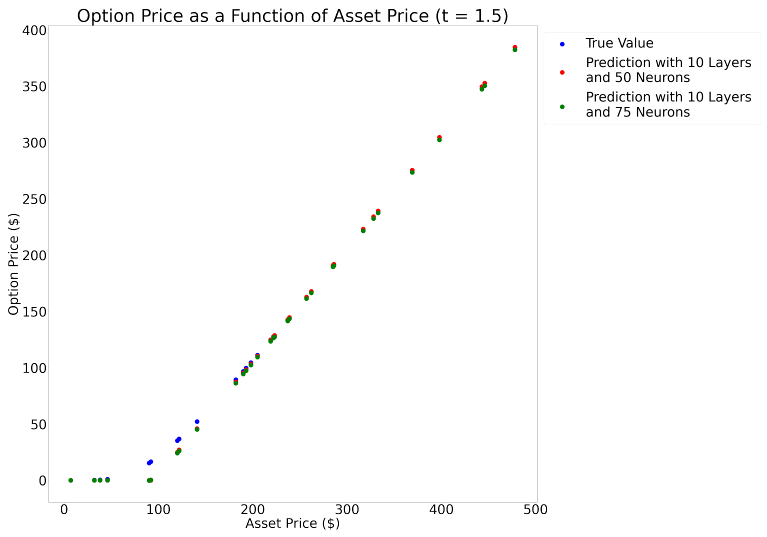

Figure 13: The models predictions for the option price as a function of asset price at $t = 1.5$. We are comparing the predictions made by our 50 neuron iteration and 75 neuron iteration of the model.

training sets and other training parameters.

3.4.2 Training Sets and Auxiliary Parameters

The model that will be used for comparison will be our original 10-layer model with 50 neurons in each layer. Our original model is trained with 100 points at the boundary, 1,000 at expiration, and 10,000 at collocation. We will vary the size of the collocation dataset given that it is the term that offers the most important penalty to the predictions. Given that we are increasing the size of the training set we cannot directly compare the two models as in previous figures, since they are not trained on the same data, but we can compare their MSE since it is an average loss per prediction which is a more appropriate metric. When our model is trained on 15,000 collocation points the model does not improve noticeably. The model used two rounds of training. The first round lasted for 2,050 epochs using a learning rate of $\eta = 8 \times 10^{-3}$ while the second round lasted for 3,800 epochs using a learning rate of $\eta = 10^{-3}$. After training, the model reports 1.90×10^{-3} loss at expiration, less than 10^{-4} loss at the boundary, 4.60×10^{-3} loss at collocation, and 26.72 loss on the function values. Compared to previous model iterations, this iteration does not provide starkly different results, not justifying the added training time arising from a larger training set. We continued our investigation on how the size of the training set affects the model, we further increased the training set to include 20,000 collocation points. The model was trained for one round of 2,050 epochs with a learning rate of $\eta = 8 \times 10^{-3}$. The model trained on 20,000 collocation points achieves 2.50×10^{-3} loss at expiration, less than 10^{-4} loss at the boundary, 4.60×10^{-3} loss at collocation, and 26.75 loss on the function values. Having doubled the number of collocation points with no noticeable improvements, we continue our optimization by investigating the learning rate, and other parameters.

3.4.3 Further Optimization

Since adjusting the number of layers, neurons, and training instances did not significantly contribute to the long-term convergence of the mode, we will look to differing our optimization approach.

To begin, we will use our original model's architecture with 10 layers and 50 neurons in each layer and vary the learning rate and train the model for 10,000 epochs. Moreover, in this section we will focus on improving our function loss even at a cost of increasing any of the other three losses. If we can improve our function loss and still manage to keep the MSE of the other three losses small relative to the range of true values then this neural network may be better equipped for use since it is better at generalizing the function at collocation at which the solution is not known. Given that a larger collocation dataset did not significantly improve the model, we will use our original setup with 10,000 collocation points to require less computational effort from our system.

Figure 14 shows how the MSE on the function values varies over 10,000 epochs as we change the training algorithm's learning rate. We notice that for learning rate values 10^{-3} and 10^{-2}, the MSE on the function values has long-term convergence to levels comparable with our previous results. However, when the learning rate is increased to 5×10^{-2}, the MSE on function values diverges to unacceptable values, implying that 5×10^{-2} is a value near the maximum learning rate. The maximum learning rate refers to the learning rate value for which the training algorithm diverges [2]. Being able to derive an estimate for the maximum learning is useful because it is commonly accepted that the optimal learning rate is approximately half the maximum learning rate. Thus,

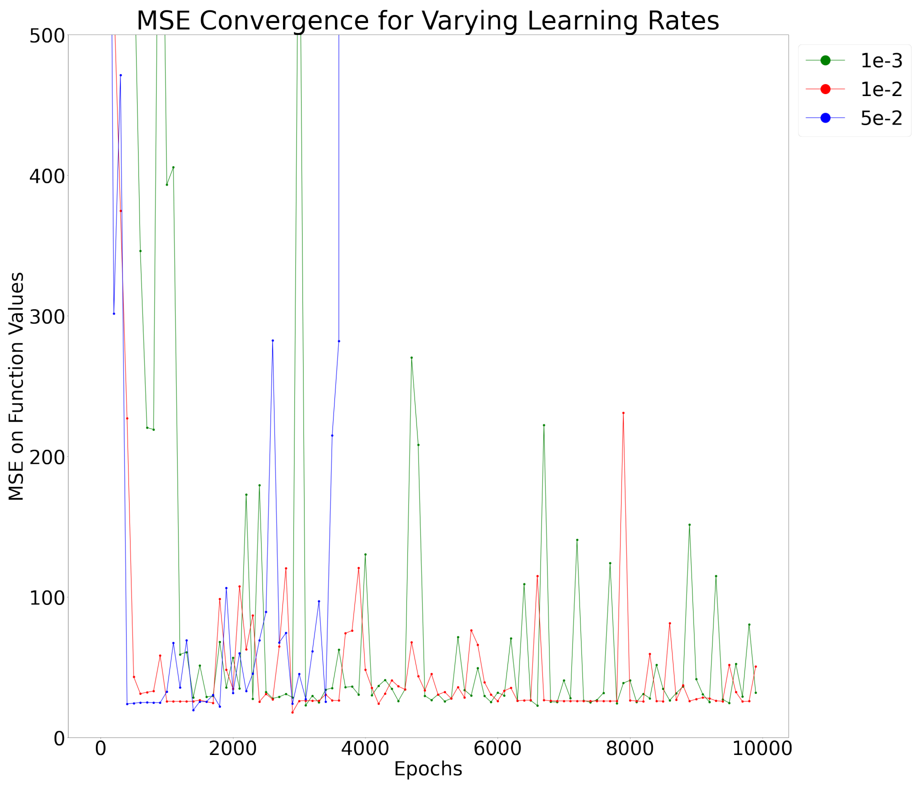

Figure 14: Long-term behavior of the MSE on the function values for varying learning rates. From the figure we can understand 5×10^{-2} is a value near the maximum learning rate - the learning rate for which the training algorithm diverges [2].

knowing that the training algorithm diverges for a learning rate of 5×10^{-2}, we analyze the model's convergence for 10,000 epochs of training using a learning rate of 2.5×10^{-2}.

From Figure 15, it is clear that 2.5×10^{-2} is still an inappropriate learning rate considering that the training still diverges. Consequently, we understand that our maximum learning is much smaller than 5×10^{-2}. To continue our investigation, we further lower the learning rate to 2×10^{-2} and train the model 10,000 epochs. The results are also provided in Figure 15. Once again, the training algorithm diverges, leading us to further reducing the learning rate to two different values - 1.25×10^{-2} and 1.5×10^{-2} - and retraining the model for both values. The final results are provided in Figure 15. Clearly, from Figure 15, the maximum learning rate is between 1.25×10^{-2} and 1.5×10^{-2}. Using this information, we will conclude our investigation into the learning rate by training model using a learning rate of 7.5×10^{-3}. Given that from Figure 15 we can assume that 1.5×10^{-2} is an upper-bound for the maximum learning rate, then 7.5×10^{-3} is a sufficient approximation for the optimal learning rate.

Using the learning rate of 7.5×10^{-3}, we trained the model for 10,000 epochs, with the results provided in Figure 16. From Figure 16, it is evident that the long-term behavior of the loss function when using the optimal learning rate does not vary significantly when compared to the behavior of the loss function when other learning rates are used. The most noticeable difference between

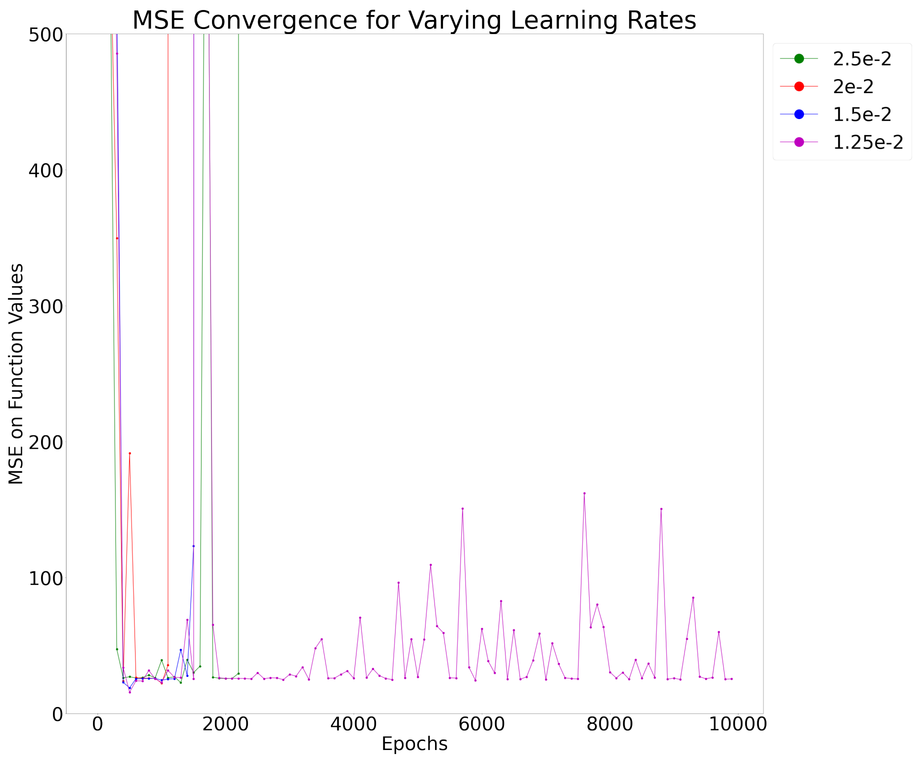

Figure 15: Long-term behavior of the MSE on the function values for varying learning rates. From the figure we can understand that the maximum learning rate value is between 1.25×10^{-2} and 1.5×10^{-2}, given that the training algorithm converges with the former learning rate but diverges with the latter value.

the suggested optimal learning rate is that, at certain epochs, the larger learning rates provide us with MSE values on the function targets lower than 20. A larger learning rate causes the model to move with greater magnitude in the direction of the gradient, possibly allowing the model to traverse to more optimal MSE levels. Overall, as expected, the size of the learning rate affects the convergence and stability of the training process. We also noticed that certain learning rates allow us to achieve better accuracy on the function values because they allow for larger adjustments to the neural network's weights. Having observed that a learning rate of 1.25×10^{-2} allows us to achieve an MSE score on the function values below 20, we will use the results from that learning rate to determine whether for that MSE score on the function values also sufficiently minimizes the remaining three losses to conclude our investigation.

Using our previous results, we determined that after 500 epochs of training with the learning rate of 1.25×10^{-2}, the model reported 17.62 loss at expiration, 3.00×10^{-4} loss at the boundary, 1.20×10^{-2} loss at collocation, and 15.67 loss on the function. A more detailed assessment is offered in Figures 17, 18, and 19. While the first two losses are on par with the results from our previous iterations of the model, the loss at expiration has increased significantly while the loss on the function values has also been reduced. However, the MSE at expiration remains small relative to the size of the true values at expiration. Even though for this iteration of the model, the MSE at expiration is higher than that of other models, when comparing the results from Figure 19 to

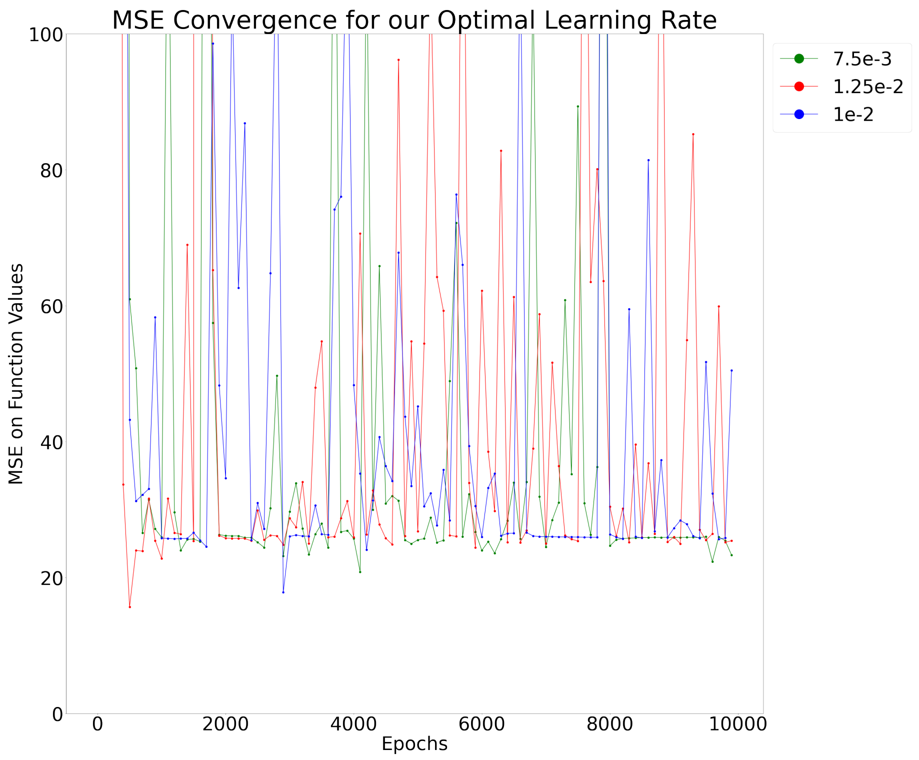

Figure 16: Long-term behavior of the MSE on the function values for varying learning rates. We compare the results from using the suggested optimal learning rate with those from other learning rate values that have shown to converge.

those from Figure 7, it is clear that the model's ability to predict the option prices at expiration has not decreased drastically. Moreover, Figures 17 and 18 when compared to the previous model predictions at $t = 0.0$ and $t = 1.5$, there is a noticeable improvement in the model's predictions for asset prices near the exercise price, which was our goal. Therefore, we have seen that allowing for more error at expiration, the boundary, and collocation, can help the model improve its overall predictive capacity.

4 Further Study and Limitations

While our models have exhibited great ability to numerically solve the Black-Scholes-Merton equation, a lot remains that can be done to improve them. In Section 3.4 we noticed that all iterations of our model had difficulty developing accurate predictions for the options price for asset prices near the exercise price. Thus, to better our model, we can construct our loss function such that the MSE penalty is amplified for points within a prescribed range from the exercise price. Similarly, the MSE penalty could be implemented such that it varies inversely as the asset price diverges in either direction from the exercise price. The amplified losses for points near the exercise price will force the algorithm to focus more on these points during training and consequently

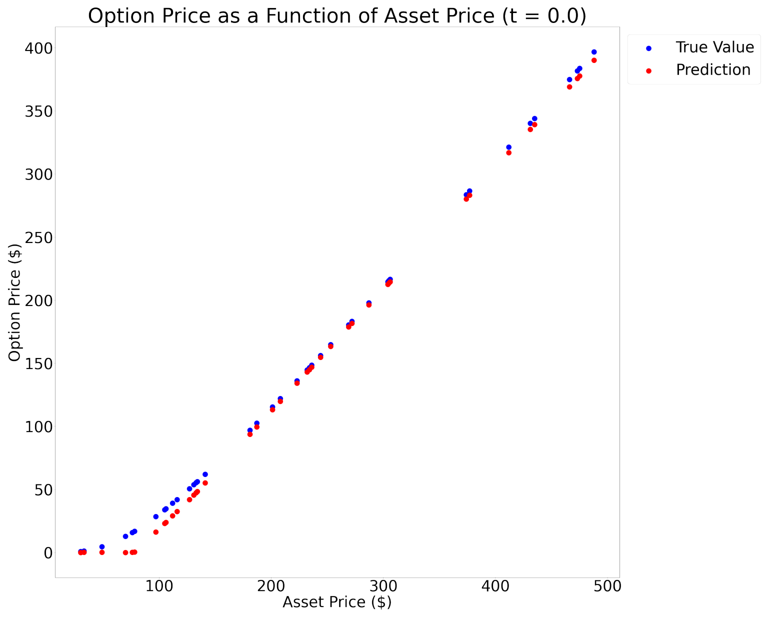

Figure 17: The model predictions for the option price as a function of asset price at $t = 0.0$. The iteration of the model was trained for 500 epochs with a learning rate of 1.25×10^{-2}.

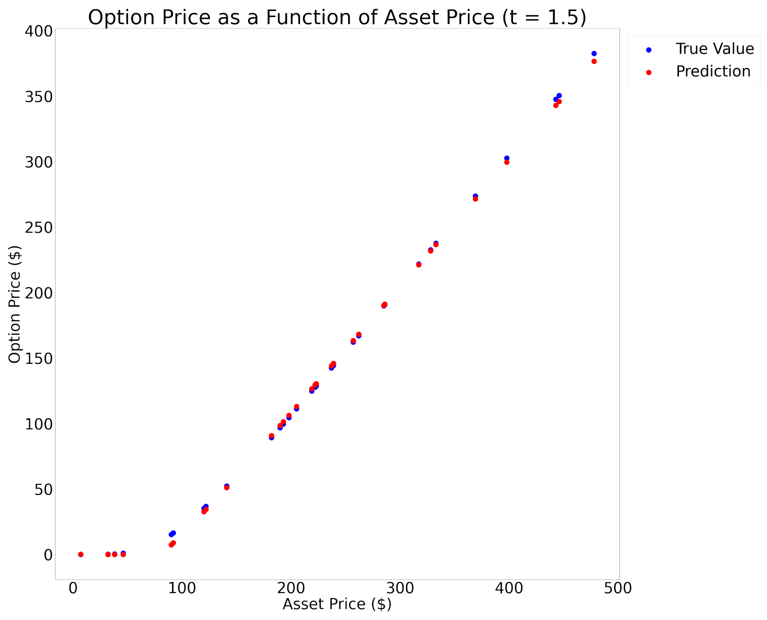

Figure 18: The model predictions for the option price as a function of asset price at $t = 1.5$. The iteration of the model was trained for 500 epochs with a learning rate of 1.25×10^{-2}.

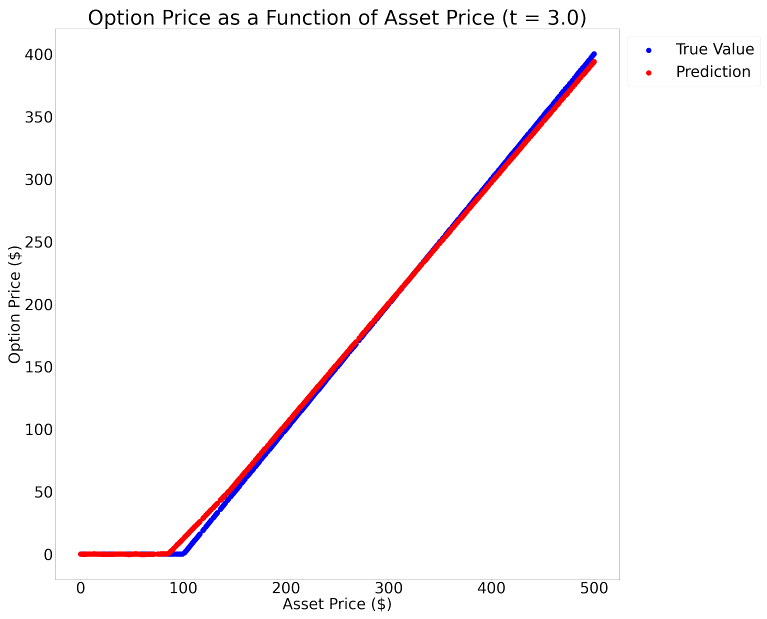

Figure 19: The model predictions for the option price as a function of asset price at expiration. The iteration of the model was trained for 500 epochs with a learning rate of 1.25×10^{-2}.

improve its predictions at those points. Similarly, consistently throughout our results, we noticed that our predictions for times farther from expiration were not as accurate as those for times closer to expiration. Thus, as was aforementioned, we could create a system such that the points closer to expiration would be underweighted in the loss and the points farther away from expiration would be overweighted. These more detailed loss functions will allow the algorithm to improve on its weaknesses while hopefully maintaining its strengths during training.

To further improve these models we could also force the model to train on an auxiliary task to act as a regularization term, a common practice in machine learning [2]. We could require the neural network to also learn the binary task of classifying the option as “in-the-money” or “out-of-the-money”. In previous results, we have seen the model, near the exercise price, predict that certain options would be worthless when in truth they had positive value. Thus, our model is implying that an option is not in the money when it actually is. By requiring the model to also predict the moneyness of the option, it could help the model better its predictions for asset prices near expiration.

A last noteworthy approach that can be taken is implementing curriculum learning. Curriculum learning involves creating a progressively harder training process for the algorithm to use. It requires forcing the algorithm to learn easier aspects or sub-tasks of the broader problem at first and then increasing the difficulty as training progresses [4]. Curriculum training could be implemented by requiring the algorithm to first learn to predict option prices for asset prices within a given range from the exercise price and then building up by making it necessary for the model to predict options prices for asset prices that get progressively farther from the exercise price. We could also apply a similar method that begins by training the algorithm on options price farther from expiration and then gradually build up by predicting option prices closer expiration. Lastly, these two methods could be combined to initially focus the model on option prices farther away from expiration and near the asset price and then progressively broaden the range of what is being learned in two-dimension: time and asset price. The mentioned approaches are some amongst a plethora of optimization, learning, and regularization techniques that can be implemented to improve the model's accuracy. Other practices can include but are not limited to constructing new loss functions more suitable for the task, varying optimizers, and implementing learning schedules [2].

While these physics-informed machines have proved fruitful in learning numerical solutions to the Black-Scholes-Merton equation, they have limitations that require addressing when considering further study. Firstly, the models have had difficulty training using various optimizers. As was mentioned, we used Adam given that it is commonly considered to have good convergence quality and speed [2]. However, when we attempted to train the models using PyTorch's built-in traditional stochastic gradient descent, it was not possible. PyTorch was not able to implement traditional stochastic gradient descent and successfully calculate the gradients of the weights of each layer. It is important to understand why the model is not able to train with rudimentary stochastic gradient descent, because if this issue is resolved, it is possible that the model can improve given the better convergence quality of traditional stochastic gradient descent as opposed to the Adam algorithm [2]. Other issues arose when we were evaluating Raissi et.al 's original code when implementing physics-informed neural networks in TensorFlow.

We previously discussed that increasing the learning rate past a specific value could lead to divergence rather than convergence in training. Nevertheless, when evaluating the TensorFlow code, we noticed that, at times, increasing the learning rate by 10^{-2} could result in the model's inability to calculate gradients and execute training. This was extremely unusual given that the learning rate was of order of magnitude 10^{-1}. While the learning rate can cause stability and convergence issues, minor adjustments such as those we made are not expected to hinder the model from training. Similar with our algorithm in PyTorch, Raissi et. al's algorithm [20] in TensorFlow was also unable to train using regular stochastic gradient descent even though it was able to train using the Adam optimizer. While our algorithm is able to train and provide rewarding results, these stability and numerical limitations need to be explored to develop a better understanding of how the black box of physics-informed machines works. Moreover, resolving the issues mentioned would allow us to use optimizers with better convergence quality and offer more opportunities for optimization.

5 Conclusion

We have shown that physics-informed neural networks are capable of learning the nonlinear behavior described by the Black-Scholes-Merton equation. In this work, we have provided the framework for developing a physics-informed approach to the Black-Scholes-Merton equation. The results from our original model validate that there exists a neural network that - to a given degree of accuracy - can provide numerical solutions to the Black-Scholes-Merton equation. Moreover, the repeated convergence of other iterations of the model reaffirms the applicability of physics-informed neural networks to the Black-Scholes-Merton equation. With the continuing developments in deep learning technology and theory, we believe that our contributions can offer a new interdisciplinary foundation for pricing existing financial derivatives. Applying physics-informed machines to financial derivative pricing provides researchers in both disciplines with a new method for modeling and understanding these assets.

It is necessary to mention that the contributions put forth in this paper do not replace existing methods for pricing American options. Even though they show promising results with limited hyperparameter tuning, previous methods inspired from the binomial tree, finite difference, or finite element method have ripened over the last 50 years and at times can provide better results than our physics-informed approach. Both methods can coexist, with deep neural networks offering an additional perspective to viewing numerical analysis for American options.

While our results are reason for optimism, in Section 3.4 we showcased that to achieve a high level of accuracy, the optimization process is more arduous than conventional hyperparameter tuning in Raissi et. al's [20, 21] and Lagaris et. al's [13] work. We will be required to better understand how moneyness and time value affect the predictive capacity of the model but also why given optimizers and learning rates may prevent successful training. Moreover, we will need to develop a more thorough background on how the network's activation and loss functions contribute to its overall accuracy. In this paper, we have aimed to address some of these issues by comparing shallower models with their deeper equivalents and varying our training process to test convergence and stability.

As a whole, we believe this paper offers a foundation for a new area in mathematical finance that will bridge three disciplines: computer science, economics, and mathematics. Through this continuous experimentation and learning process there is potential to enrich all three fields and constantly offer improvements.

Appendix A. PyTorch Implementation Code for Original Model

```
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt

torch.manual_seed(123)

class BlackScholesMertonModel1(nn.Module):
    def __init__(self):
        super(BlackScholesMertonModel1, self).__init__()

        self.bn1 = nn.BatchNorm1d(2)
        self.fc1 = nn.Linear(2, 50)
        self.act1 = nn.ReLU()

        self.bn2 = nn.BatchNorm1d(50)
        self.fc2 = nn.Linear(50, 50)
        self.act2 = nn.ReLU()

        self.bn3 = nn.BatchNorm1d(50)
        self.fc3 = nn.Linear(50, 50)
        self.act3 = nn.ReLU()

        self.bn4 = nn.BatchNorm1d(50)
        self.fc4 = nn.Linear(50, 50)
        self.act4 = nn.ReLU()

        self.bn5 = nn.BatchNorm1d(50)
        self.fc5 = nn.Linear(50, 50)
        self.act5 = nn.ReLU()

        self.bn6 = nn.BatchNorm1d(50)
        self.fc6 = nn.Linear(50, 50)
        self.act6 = nn.ReLU()

        self.bn7 = nn.BatchNorm1d(50)
        self.fc7 = nn.Linear(50, 50)
        self.act7 = nn.ReLU()

        self.bn8 = nn.BatchNorm1d(50)
        self.fc8 = nn.Linear(50, 50)
        self.act8 = nn.ReLU()

        self.bn9 = nn.BatchNorm1d(50)
        self.fc9 = nn.Linear(50, 50)
        self.act9 = nn.ReLU()

        self.bn10 = nn.BatchNorm1d(50)
        self.fc10 = nn.Linear(50, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act1(x)

        x = self.fc2(x)
        x = self.act2(x)

        x = self.fc3(x)
        x = self.act3(x)

        x = self.fc4(x)
        x = self.act4(x)

        x = self.fc5(x)
        x = self.act5(x)

        x = self.fc6(x)
        x = self.act6(x)

        x = self.fc7(x)
        x = self.act7(x)
        x = self.fc8(x)
        x = self.act8(x)

        x = self.fc9(x)
        x = self.act9(x)

        x = self.fc10(x)
        return x

from ExplicitEu import ExplicitEu
from ImplicitEu import ImplicitEu
from ImplicitAm import ImplicitAmBer
from ImplicitAm import ImplicitAmBre

S0 = 100
exercise_price = 100
sigma = 0.4
r = 0.03
dividend = 0.00
tau = 3
M = 500  # S
N = 600 # t
Smax = 500
is_call = True

N_b = 100
N_exp = 1000
N_f = 10000

lb = [0, 0]
ub = [500, tau]

t, S = np.meshgrid(np.linspace(0, 1, N+1), np.linspace(0, Smax, M+1))
option = ImplicitAmBer(S0, exercise_price, r, tau, sigma, Smax, M, N, is_call)
option.price()
option_fde_prices = option.grid

def initialize_data(N_b, N_exp, N_f, lb, ub, exercise_price, tau):
    # data for collocation
    stock_price_collocation = torch.randint(low = 0, high = ub[0] + 1, size =
    ↪  (N_f, 1)).type(torch.FloatTensor)
    time_collocation = (torch.randint(low = 0, high = 100 * ub[1] + 1, size =
    ↪  (N_f, 1)) / 100).type(torch.FloatTensor)

    stock_price_mean = torch.mean(stock_price_collocation)
    stock_price_std = torch.std(stock_price_collocation)

    time_mean = torch.mean(time_collocation)
    time_std = torch.std(time_collocation)

    X_f = torch.cat((time_collocation, stock_price_collocation), 1)

    time_collocation = (time_collocation - time_mean) / time_std

    stock_price_collocation = (stock_price_collocation - stock_price_mean) /
    ↪  stock_price_std

    X_f_norm = torch.cat((time_collocation, stock_price_collocation), 1)
    # data for boundary
    time_boundary = (torch.randint(low = 1, high = 100 * ub[1] + 1, size = (N_b,
    ↪  1)) / 100).type(torch.FloatTensor)

    X_b = torch.cat((time_boundary, 0 * time_boundary), 1)

    # data for initial time
    stock_price_exp = torch.randint(low = 0, high = ub[0] + 1, size = (N_exp,
    ↪  1)).type(torch.FloatTensor)

    option_price_exp = stock_price_exp - exercise_price

    u_exp = torch.Tensor([[max(instance, 0)] for instance in
    ↪  option_price_exp]).type(torch.FloatTensor)

    X_exp = torch.cat((0 * stock_price_exp + tau, stock_price_exp), 1)

    return X_f, X_f_norm, X_b, X_exp, u_exp

X_f, X_f_norm, X_b, X_exp, u_exp = initialize_data(N_b, N_exp, N_f, lb, ub,
↪  exercise_price, tau)

u_collocation = []

for instance in X_f:
    time = instance[0]
    stock_price = instance[1]

    stock_price = int(stock_price.item())
    time = int(time.item() * 200)
    u_collocation_val =
    ↪   torch.Tensor(np.array([(np.round(option_fde_prices[stock_price, time],
    ↪   3))])).type(torch.FloatTensor)
    u_collocation.append(u_collocation_val)

u_collocation = np.array(u_collocation)
u_collocation = np.reshape(u_collocation, (-1, 1))
u_collocation = torch.Tensor(u_collocation).type(torch.FloatTensor)
print(u_collocation.shape)
f_collocation = torch.zeros(N_f, 1)
u_boundary = torch.zeros(N_b, 1)

# initializing the pde solver
model1 = BlackScholesMertonModel1()

# Original weight initialization didn't change the weights.
# Initialize the weights
for m in model1.modules():
    if isinstance(m, nn.Linear):
        torch.nn.init.kaiming_normal_(m.weight, mode = 'fan_out',
        ↪   nonlinearity='relu')
        torch.nn.init.constant_(m.bias, 0)

# perform backprop
MAX_EPOCHS_1 = int(710)
LRATE = 8e-3

# use Adam for training
optimizer = torch.optim.Adam(model1.parameters(), lr=LRATE)
# scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 210, gamma=0.5,
↪   last_epoch=-1, verbose=False)

X_f.requires_grad = True

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model1.to(device)

# send everything to GPU.
X_f = X_f.to(device)
X_b = X_b.to(device)
X_exp = X_exp.to(device)
u_boundary = u_boundary.to(device)
u_exp = u_exp.to(device)
u_collocation = u_collocation.to(device)
f_collocation = f_collocation.to(device)
```

```
loss_history_function_1 = []
loss_history_f_1 = []
loss_history_boundary_1 = []
loss_history_exp_1 = []

print("Learning Rate for this Round of Training : ", LRATE)
print("First Round of Training")
for epoch in range(MAX_EPOCHS_1):

    # boundary loss
    with torch.no_grad():
        rand_index = torch.randperm(n = len(X_b), device = device)
    X_b_shuffle = X_b[rand_index]
    u_boundary_shuffle = u_boundary[rand_index]
    u_b_pred = model1(X_b_shuffle)
    mse_u_b = torch.nn.MSELoss()(u_b_pred, u_boundary_shuffle)

    # expiration time loss
    with torch.no_grad():
        rand_index = torch.randperm(n = len(X_exp), device = device)
    X_exp_shuffle = X_exp[rand_index]
    u_exp_shuffle = u_exp[rand_index]
    u_exp_pred = model1(X_exp_shuffle)
    u_exp_pred = u_exp_pred.to(device)
    mse_u_exp = torch.nn.MSELoss()(u_exp_pred, u_exp_shuffle)

    # collocation loss
    with torch.no_grad():
        rand_index = torch.randperm(n = len(X_f), device = device)

    X_f_shuffle = X_f[rand_index]

    f_collocation_shuffle = f_collocation[rand_index]
    u_collocation_shuffle = u_collocation[rand_index]

    u_pred = model1(X_f_shuffle)

    stock_price = X_f_shuffle[:, 1:2]

    u_pred_first_partials = torch.autograd.grad(u_pred.sum(), X_f_shuffle,
    ↪  create_graph = True, allow_unused = True)[0]
    u_pred_dt = u_pred_first_partials[:, 0:1]
    u_pred_ds = u_pred_first_partials[:, 1:2]

    u_pred_second_partials = torch.autograd.grad(u_pred_ds.sum(), X_f_shuffle,
    ↪  create_graph = True, allow_unused = True)[0]
    u_pred_dss = u_pred_second_partials[:, 1:2]

    f_pred = u_pred_dt + (0.5 * (sigma ** 2) * (stock_price ** 2) * u_pred_dss) +
    ↪  ((r - dividend) * stock_price * u_pred_ds) - (r * u_pred)
    f_true = f_collocation_shuffle
    mse_f = 100 * torch.nn.MSELoss()(f_pred, f_true)

    loss = mse_f + mse_u_exp + mse_u_b

    mse_function = torch.nn.MSELoss()(u_pred, u_collocation_shuffle).detach()

    # optimizer step
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    loss_history_f_1.append(mse_f / 100)
    loss_history_boundary_1.append(mse_u_b)
    loss_history_exp_1.append(mse_u_exp)
    loss_history_function_1.append(mse_function)

    if (epoch % 10) == 0:
        print("- - - - - - - - - - - - - - -")
        print("Epoch : ", epoch)
        print(f"Loss Residual:\t{loss_history_f_1[-1]:.4f}")
        print(f"Loss Boundary:\t{loss_history_boundary_1[-1]:.4f}")
        print(f"Loss Expiration:\t{loss_history_exp_1[-1]:.4f}")
        print(f"Loss Function:\t{loss_history_function_1[-1]:.4f}")
print("-----------------------------------------------------------")

MAX_EPOCHS_2 = int(4700)
LRATE = 1e-3

# use Adam for training
optimizer = torch.optim.Adam(model1.parameters(), lr=LRATE)

loss_history_function_2 = []
loss_history_f_2 = []
loss_history_boundary_2 = []
loss_history_exp_2 = []

print("Learning Rate for this Round of Training : ", LRATE)
print("Second Round of Training")
for epoch in range(MAX_EPOCHS_2):

    # boundary loss
    with torch.no_grad():
```

```
    rand_index = torch.randperm(n = len(X_b), device = device)
X_b_shuffle = X_b[rand_index]
u_boundary_shuffle = u_boundary[rand_index]
u_b_pred = model1(X_b_shuffle)
mse_u_b = torch.nn.MSELoss()(u_b_pred, u_boundary_shuffle)

# expiration time loss
with torch.no_grad():
    rand_index = torch.randperm(n = len(X_exp), device = device)
X_exp_shuffle = X_exp[rand_index]
u_exp_shuffle = u_exp[rand_index]
u_exp_pred = model1(X_exp_shuffle)
u_exp_pred = u_exp_pred.to(device)
mse_u_exp = torch.nn.MSELoss()(u_exp_pred, u_exp_shuffle)

# collocation loss
with torch.no_grad():
    rand_index = torch.randperm(n = len(X_f), device = device)

X_f_shuffle = X_f[rand_index]

f_collocation_shuffle = f_collocation[rand_index]
u_collocation_shuffle = u_collocation[rand_index]

u_pred = model1(X_f_shuffle)

stock_price = X_f_shuffle[:, 1:2]

u_pred_first_partials = torch.autograd.grad(u_pred.sum(), X_f_shuffle,
↪  create_graph = True, allow_unused = True)[0]
u_pred_dt = u_pred_first_partials[:, 0:1]
u_pred_ds = u_pred_first_partials[:, 1:2]

u_pred_second_partials = torch.autograd.grad(u_pred_ds.sum(), X_f_shuffle,
↪  create_graph = True, allow_unused = True)[0]
u_pred_dss = u_pred_second_partials[:, 1:2]

f_pred = u_pred_dt + (0.5 * (sigma ** 2) * (stock_price ** 2) * u_pred_dss) +
↪  ((r - dividend) * stock_price * u_pred_ds) - (r * u_pred)
f_true = f_collocation_shuffle
mse_f = 100 * torch.nn.MSELoss()(f_pred, f_true)

loss = mse_f + mse_u_exp + mse_u_b

mse_function = torch.nn.MSELoss()(u_pred, u_collocation_shuffle).detach()
```

```
    # optimizer step
    loss.backward()
    optimizer.step()
    for m in model1.modules():
        if isinstance(m, nn.Linear):
            print(m.weight.grad)

    optimizer.zero_grad()
    loss_history_f_2.append(mse_f / 100)
    loss_history_boundary_2.append(mse_u_b)
    loss_history_exp_2.append(mse_u_exp)
    loss_history_function_2.append(mse_function)

    if (epoch % 10) == 0:
        print("- - - - - - - - - - - - - - - -")
        print("Epoch : ", epoch)
        print(f"Loss Residual:\t{loss_history_f_2[-1]:.4f}")
        print(f"Loss Boundary:\t{loss_history_boundary_2[-1]:.4f}")
        print(f"Loss Expiration:\t{loss_history_exp_2[-1]:.4f}")
        print(f"Loss Function:\t{loss_history_function_2[-1]:.4f}")
print("----------------------------------------------------------")
```

References

[1] Al-Aradi, Ali et al. “Solving Nonlinear and High-Dimensional Partial Differential Equations via Deep Learning.” (2018). Web.

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O’Reilly Media, Inc, 2019. Web.

[3] Baydin, Atilim Gunes et al. “Automatic Differentiation in Machine Learning: a Survey.” (2015). Web.

[4] Bengio, Yoshua et al. “Curriculum Learning.” *Proceedings of the 26th Annual International Conference on Machine Learning*. Vol. 382. ACM, 2009. Web.

[5] Black, Fischer, and Myron Scholes. “The Pricing of Options and Corporate Liabilities.” *The Journal of political economy* 81.3 (1973): 637–654. Web.

[6] Brennan, Michael J, and Eduardo S Schwartz. “The Valuation of American Put Options.” *The Journal of finance* (New York) 32.2 (1977): 449–. Web.

[7] Cox, John C, Stephen A Ross, and Mark Rubinstein. “Option Pricing: A Simplified Approach.” *Journal of financial economics* 7.3 (1979): 229–263. Web.

[8] Eskiizmirliler, Saadet, Korhan Gunel, and Refet Polat. “On the Solution of the Black-Scholes Equation Using Feed-Forward Neural Networks.” *Computational economics (2020)*. Web.

[9] Geske, Robert, and Kuldeep Shastri. “Valuation by Approximation: A Comparison of Alternative Option Valuation Techniques.” *Journal of financial and quantitative analysis* 20.1 (1985): 45–71. Web.

[10] Han, Houde, and Xiaonan Wu. “A Fast Numerical Method for the Black-Scholes Equation of American Options.” *SIAM journal on numerical analysis* 41.6 (2003): 2081–2095. Web.

[11] Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. “Multilayer Feedforward Networks Are Universal Approximators.” *Neural networks* 2.5 (1989): 359–366. Web.

[12] Johnson, H. E. “An Analytic Approximation for the American Put Price.” *Journal of financial and quantitative analysis* 18.1 (1983): 141–148. Web.

[13] Lagaris, I.E, A Likas, and D.I Fotiadis. “Artificial Neural Networks for Solving Ordinary and Partial Differential Equations.” *IEEE transactions on neural networks* 9.5 (1998): 987–1000. Web.

[14] LeVeque, Randall J. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics, 2007. Web.

[15] Lin, Henry W, Max Tegmark, and David Rolnick. “Why Does Deep and Cheap Learning Work So Well?” *Journal of statistical physics* 168.6 (2017): 1223–1247. Web.

[16] Liu, Shuaiqiang, Kees Oosterlee, and Sander Bohte. “Pricing Options and Computing Implied Volatilities Using Neural Networks.” *Risks (Basel)* 7.1 (2019): 16–. Web.

[17] MacMillan, Lionel W. "Analytic Approximation for the American Put Option." *Advances in Futures and Options Research : A Research Annual* 1 (1986). 119-139. Web.

[18] Madi, Sofiane et al. "Pricing of American Options, Using the Brennan–Schwartz Algorithm Based on Finite Elements." *Applied mathematics and computation* 339 (2018): 846–852. Web.

[19] Milano, Michele, and Petros Koumoutsakos. "Neural Network Modeling for Near Wall Turbulent Flow." *Journal of computational physics* 182.1 (2002): 1–26. Web.

[20] Raissi, M, P Perdikaris, and G.E Karniadakis. "Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations." *Journal of computational physics* 378 (2019): 686–707. Web.

[21] Raissi, Maziar, and George Em Karniadakis. "Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations." *Journal of computational physics* 357 (2018): 125–141. Web.

[22] Salvador, Beatriz, Cornelis W Oosterlee, and Remco van der Meer. "European and American Options Valuation by Unsupervised Learning with Artificial Neural Networks." *Proceedings* 54.1 (2020): 14–. Web.

[23] Shahmorad, Sedaghat, Robab Kalantari, and Ahmad Assadzadeh. "Numerical Solution of Fractional Black-Scholes Model of American Put Option Pricing via a Nonstandard Finite Difference Method: Stability and Convergent Analysis." *Mathematical methods in the applied sciences* 44.4 (2021): 2790–2805. Web.

[24] Sirignano, Justin, and Konstantinos Spiliopoulos. "DGM: A Deep Learning Algorithm for Solving Partial Differential Equations." *Journal of computational physics* 375 (2018): 1339–1364. Web.

[25] Wilmott, Paul., Jeff. Dewynne, and Sam. Howison. *Option Pricing: Mathematical Models and Computation.* Oxford: Oxford Financial Press, 1993. Web.

[26] Zapart, Christopher A. "Beyond Black–Scholes: A Neural Networks-Based Approach to Options Pricing." *International journal of theoretical and applied finance* 6.5 (2003): 469–489. Web.

[27] Zhange, Quancheng. "Options Pricing using Finite Difference Methods." Github repository (2017). Web.

[28] Zhu, Song-Ping, and Wen-Ting Chen. "An Inverse Finite Element Method for Pricing American Options." *Journal of economic dynamics & control* 37.1 (2013): 231–250. Web.

[29] Zienkiewicz, O. C., R. L. Taylor, and J. Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals.* Seventh edition. Oxford, UK: Butterworth-Heinemann, 2013. Web.