MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

# Accurate Function Approximation with Neural Networks

**Scientific Students' Association Report**

Author:

Bálint Takáts

Advisor:

dr. Balázs Renczes

2023

# Contents

**Kivonat** **i**

**Abstract** **ii**

**1 Introduction** **1**

**2 Microsecond Calibration** **4**

2.1 Model Calibration . . . . 4

2.2 Stochastic Models . . . . 4

2.3 Pricing . . . . 6

2.4 Evaluation and Comparison . . . . 8

2.5 Intuition in One-Dimension Using the Sinusoidal Function . . . . 10

2.6 Efficient Training . . . . 11

**3 Pricing with Physics Informed Neural Networks** **13**

3.1 General Setup . . . . 13

3.2 The Black-Scholes PDE . . . . 15

3.3 Automatic Differentiation . . . . 16

3.4 Gradient Pathologies . . . . 16

3.5 Efficient Training . . . . 19

3.6 Results . . . . 20

**4 Conclusion and Future Work** **23**

**Acknowledgements** **25**

**Bibliography** **26**

# Kivonat

A neurális hálózatok sokoldalú eszközökké váltak a függvények közelítésére különböző tudományterületeken. A hasznos alkalmazások egyre bővülő területei a biológia, a fizika, a mérnöki és a pénzügyi tudományok. A kutatásokat jelenleg a struktúra közelítési képességeivel kapcsolatos, folyamatosan bővülő szakirodalom motiválja. Ezek az elméleti eredmények azt mutatják be, hogy a neurális hálózatok képesek megtanulni a függvények széles körének tetszőlegesen jó közelítését. Ebben a tanulmányban két olyan modern módszert vizsgálok, amelyek a neurális hálózatokat függvények közelítésére használják fel: a fizikával informált neurális hálózatokat (PINN) és az inverz függvényapproximációt.

Az első részben olyan módszert mutatok be, amelyet részben invertálható függvények közelítésére lehet alkalmazni. Ezt a részt a pénzügyi modellek kalibrálása motiválta, ahol az a feladat, hogy olyan modellparamétereket találjuk, amelyek reprodukálják a volatilitási felületet. Egy kétlépéses tanítási módszert javaslok, amelyet az autoencoder architektúra ihletett. A bemutatott algoritmus célja, hogy megoldást adjon egy olyan domén automatikus felfedezésére, ahol egy részleges inverz pontosan közelíthető, miközben felgyorsítja a nehezen reprezentálható vagy időigényes inverz függvények kiszámítását. Bemutatom, hogy a sztochasztikus pénzügyi modellek kalibrálási ideje jelentősen felgyorsul, miközben a pontosság csak kis mértékben csökken.

A második részben bemutatom a fizikával informált neurális hálózatok tanításával kapcsolatos munkámat. A PINN-ek a neurális hálózatok egy olyan osztálya, ahol a tanítási eljárásra egy puha megkötést teszünk. Ebben az esetben ez a megkötés egy parciális differenciálegyenlet, amelynek teljesítésére betanítjuk a hálózatot. Ebben az elméletibb részben áttekintem a terület legújabb eredményeit, és megmutatom, hogyan alkalmazhatóak ezek az elméleti eredmények egy hálózat tanítására, amely közelíti a híres Black-Scholes differenciálegyenletet. Bemutatom, hogy a PINN keretrendszer sikeresen alkalmazható nem differenciálható peremfeltételű differenciálegyenletekre. Kísérletezem olyan algoritmusokkal, amelyeknek célja a tanítási folyamat stabilizálása és felgyorsítása. A hatékonyság bemutatása mellett megvizsgálom a gradiens patológiákat, hiperparaméter keresést végzek és tesztelek különböző optimalizálási módszereket. A hálózatok optimalizálásának további segítésére egy fejlett eljárást is adok.

Ebben a tanulmányban a pénzügyi területen való alkalmazhatóságra fogok összpontosítani. Azonban a bemutatott megoldások további problémák széles körére is alkalmazhatóak. Emellett átfogó, lépésről lépésre haladó magyarázatokat adok, kísérő Jupyter notebookokon keresztül, megkönnyítve a gyors megértést és kísérletezést.

# Abstract

Neural networks have become versatile tools for approximating involved functions across various scientific disciplines. The ever-growing domain of useful applications includes biology, physics, engineering and finance. Research is currently motivated by the constantly growing literature on the approximation capabilities of this structure. These theoretical results show that neural networks can learn to approximate a wide range of functions arbitrarily well. In this study, I explore two recent approaches that leverage neural networks to approximate functions: physics-informed neural networks (PINNs) and inverse function approximation.

In the first part, I will present a neural network training method that can be used to approximate partially invertible functions. This part was motivated by financial model calibration, where the task is to find a set of model parameters that can match the volatility surface. I propose a two-step training method that is inspired by the autoencoder architecture. The presented algorithm aims to provide a way to automatically discover a domain where a partial inverse can be accurately approximated while also speeding up the calculation of difficult-to-represent or time-consuming-to-calculate inverse functions. I will demonstrate a significant speedup in the calibration time of the stochastic financial models while accuracy decreases only marginally.

In the second part, I will present my work on training physics-informed neural networks. PINNs are a class of neural networks where a soft constraint is placed on the training procedure. In this case, this constraint is a partial differential equation that the network is trained to satisfy. In this more theoretic part, I review recent advances in the field and put them to use by training a network to approximate the famous Black-Scholes differential equation. I demonstrate that the PINN framework can be successfully applied to differential equations with a non-differentiable boundary condition. I also conduct experiments with the proposed algorithms aimed at stabilizing and speeding up training. Besides showing the effectiveness, I investigate the gradient pathologies, conduct a hyperparameter search and experiment with different optimization methods. I also give advanced procedures to further aid the training of these networks.

In this study, I will focus on the applicability and effectiveness of the proposed methods in financial applications. However, the presented solutions can be applied to a wide range of problems. Additionally, I provide comprehensive, step-by-step explanations through accompanying Jupyter notebooks, facilitating quick comprehension and experimentation.

# Chapter 1

# Introduction

The following work focuses on financial derivatives, financial models and how to approximate functions that are used to price these products. The most basic financial derivatives are options. These are instruments that derive their value from an underlying security like stocks or exchange-traded funds (ETFs). The simplest kind of option is a European call option, which offers the buyer the opportunity to buy the underlying asset at a predetermined price at a predetermined time. This raises a question: how can one calculate the fair price of this contract in an ever-changing market? It took three brilliant minds to answer this question: Robert Merton, Fischer Black and Myron Scholes. They were awarded the Nobel Prize in Economics in 1997 for their outstanding work [4]. To better understand the impact of their findings, one must consider the role of financial institutions in the market. In essence, they are the guardians of risk management. By offering derivatives as insurance, they allow various entities to hedge their market exposures, effectively shielding them from potential adverse shifts in market conditions. Take the airline industry, for instance. Here, fuel costs often constitute a large chunk of operational expenses. As a protective strategy against spiralling oil prices, airlines often resort to purchasing derivatives tethered to oil price fluctuations. In this scenario, a call option on oil serves as a financial cushion, insulating airlines from abrupt and potentially crippling fuel cost hikes. The market is responsible for providing these instruments at their fair price. The three Nobel laureates discovered a way to determine the fair market price while also devising a method for the banks to hedge the risk of the derivatives positions. This led to a liquid and well-functioning derivatives market and a new branch of mathematics, the theory of finance.

Despite decades of research, the Black-Sholes pricing formula is still widely used within the industry. This simple model only requires five parameters to describe a European call option: strike price, spot price, time to expiry, risk-free rate of interest and volatility. Strike price and time to expiry are set by trading parties before the purchase. The owner of the option can purchase the underlying at the previously specified strike price. The time when this purchase can occur is determined by the time of expiry. The risk-free rate of interest is what an investor can expect to earn on an investment that carries zero risk. An example of this would be a zero-coupon bond issued by a central bank. Spot price refers to the price of the underlying security. The most interesting among them is volatility. It is an inherent phenomenon in the market, a statistical measure of the change in the price of a security. Alternatively, it can be thought of as risk or uncertainty.

The ease of use and simplicity of the Balck-Scholes pricing formula lead to cases where it fails to model aspects of the market. For example, the Black-Scholes model comes with the assumption of constant volatility, an inherent flaw of the model since empirical evidence shows that volatility is time-dependent. The volatility smile stems from the observation that far-out-of-the-money (OTM), where the strike ($K$) is significantly lower than the underlying's price ($S$), options usually command higher prices, hence higher volatilities. Volatility is often implied from liquid vanilla options thus the name implied volatility (IV). The IV surface is a two-dimensional market description constructed by observing available volatilities for different strikes and expiration dates. Due to its flaws, the Black-Scholes model is insufficient to model the market accurately. Addressing this issue has led to increasingly sophisticated models. One industry standard is the SABR model which has become commonplace in most fixed-income desks [26]. Other notable models are the Heston model [28] and the Heath–Jarrow–Morton model [12].

Financial models usually come in the shape of stochastic differential equations (SDE). However, finding analytic formulas for these types of equations can be challenging. Moreover, a wide variety of derivatives are available in the market, many of them are so-called exotic options. Arbitrage-free pricing of complex path-dependent options requires expensive to compute Monte Carlo methods and model calibration. Model calibration, in this case, amounts to finding a set of model parameters that match the observed volatilities of the market as closely as possible. In the absence of an analytical solution to the underlying SDE, this problem usually boils down to least squares minimization using iterative optimizers such as Levenberg-Marquard [35, 38]. Running iterative optimization can be challenging since it requires the calculation of the Jacobian matrix of the volatility surface with respect to the model parameters. Calibration can take up to 20 minutes due to the relative slowness of the Monte Carlo method. Several solutions have been proposed to solve this issue. The method proposed by Hernandez [27] describes the direct calibration of model parameters from the IV surface. The two-step calibration introduced by Horváth et al. [5] trains a network in the opposite direction and calibrates using approximate derivatives. The common theme in these works is the offline training of the networks to speed up online calibration. One contribution of this paper is a method to speed up the online calibration from a few hundred milliseconds to a few microseconds. I also provide a framework to accurately and quickly train a neural network to approximate non-injective functions. In this context, the proposed technique opens the door to real-time calibration of intricate models with a multitude of parameters. Beyond its immediate application, this method also holds promise for broader engineering disciplines. Notably, it could offer a viable solution for tackling ill-posed inverse problems, which involve inferring causative factors from noisy measurements.

In the second part of this work, I show how to train physics-informed neural networks (PINNs) to approximate the Black-Scholes pricing function. The PINN framework aims to address one fundamental problem of deep learning [39, 40, 41]. Recent advances in deep learning have yielded exciting results in language modelling, image generation and object detection [21, 6, 16, 10]. One core ingredient that made their success possible is limitless high-quality data. Neural networks work best in data-rich domains but their effectiveness remains subpar in settings where data is sparsely available. The over-parametrization allowed for flexibility but generalization requires prior knowledge or inductive bias. Achieving this requires clever network design like the convolutional neural network (CNN). This architecture bakes translational invariance into the network's architecture [33]. Another notable architecture is the attention mechanism of large language models (LLMs). Their architecture is invariant to the ordering of input tokens [13]. Clever network design is a tedious process and more an art than an exact science. This work and many others address the problem of prior knowledge by exploring the construction of training regimes that allow for better generalization with the modification of the loss function in the following way:

$$L(\mathbf{w}) := \frac{1}{N_b}\sum_{i=1}^{N_b}[u_i - N_{\mathbf{w}}(x_i)]^2 + \lambda R[N_{\mathbf{w}}(x)] \; . \tag{1.1}$$

In this case, the loss $L(\mathbf{w})$ consists of two parts: the mean squared error over the given input-output pairs $x_i$ and $u_i$, for $i = 0, 1, \ldots, N_b$, plus a penalty or soft constraint in the form of $R[N_{\mathbf{w}}(x)]$, sometimes also called the residual. Furthermore, $\mathbf{w}$ denotes the parameters or the neural network, $N_{\mathbf{w}}$ represents the output of the neural network, $N_b$ is the batch size, and $\lambda$ is the scaling factor for the residual loss.

Soft constraints have been successfully applied in other domains of deep learning, most notably, the weight decay using $L_1$ or $L_2$ norms [25]. The penalty in the PINN case will force the network to obey a partial differential equation. Recent advances have hinted at possible uses in chaotic flow simulations and accurate weather dynamics modelling [15, 18]. I describe PINNs as a deep learning method, but they can be viewed as an alternative to classical PDE solvers. An added benefit over previously developed methods, like finite elements, is the possibility to train the network with extra input and get a family of solutions as a single neural network [8]. My contributions will be the design and training of a neural network that satisfies the Black-Scholes differential equation. I further expand on previous work by augmenting the learning rate annealing algorithm proposed by Wang et al. [20, 19].

Following the principles of Open Science, my results are available on GitHub: **NN-Approximation** to facilitate understanding and further experimentation.

# Chapter 2

# Microsecond Calibration

## 2.1 Model Calibration

In Chapter 1, I introduced the concept of model calibration. Calibration is the process of finding parameters such that the model reproduces the observed market reality as closely as possible. It is important to distinguish model parameters from the weights of the neural networks. Hence, $\boldsymbol{\theta}$ will refer to the parameters of the pricing model, and $\mathbf{w}$ will refer to the parameters of the neural network. To formalize this, let's define the pricing model as $M := M(\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} \in \Theta$ in the set $\Theta \subset \mathbb{R}^n$, for some $n \in \mathbb{N}$. Once $M(\boldsymbol{\theta})$ is fixed, meaning the model and its parameters, all corresponding derivatives prices are also fully specified. Using this definition, we can introduce a pricing map: $P := P(M(\boldsymbol{\theta}), \phi) \to \mathbb{R}^m$ for some $m \in \mathbb{N}$ where $\phi$ specifies the products we aim to price. The specified products are usually liquid options for different strikes and expiries that can be observed in the market. I denote the real market price of these options with $P_{\text{mkt}}(\phi)$.

Parameter calibration solves the following problem: given $M$ and $P_{\text{mkt}}(\phi)$ find a set of parameters $\boldsymbol{\theta}$ so that the difference between $P_{\text{mkt}}(\phi)$ and $P(M(\boldsymbol{\theta}), \phi)$ is minimized:

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \|P(M(\boldsymbol{\theta}), \phi) - P_{\text{MKT}}(\phi)\|_2^2 \ . \tag{2.1}$$

Where $\|.\|_2$ represents the Euclidean norm. In practice, it is not possible to calibrate with zero error. Noise in the market data and imperfect calibration schemes introduce errors in a given calibration. Hedge funds and investment banks use sophisticated optimization algorithms to find the optimal values of $\boldsymbol{\theta}$ like Levenberg-Marquard [35, 38]. It is also possible to use Gradient Descent or any other iterative optimization algorithm. The main idea of Gradient Descent is the calculation of the Jacobian of the error with respect to $\boldsymbol{\theta}$ and then the update of the parameters:

$$\boldsymbol{\theta_{n+1}} = \boldsymbol{\theta_n} - \alpha \frac{\partial}{\partial \boldsymbol{\theta_n}} \|P(M(\boldsymbol{\theta_n}), \phi) - P_{\text{MKT}}(\phi)\|_2^2 \ . \tag{2.2}$$

Where $\alpha$ is the learning rate.

## 2.2 Stochastic Models

In this section, I will describe two notable models that are widely used in industry.

**The Black-Scholes model**

The model is described by the following stochastic differential equation [4]:

$$\frac{dS}{S} = \mu\, dt + \sigma\, dW\ . \tag{2.3}$$

The symbols mean the following:

- $S$ represents the price of the underlying. The price changes with time.
- $\sigma$ represents the volatility, it is a constant.
- $\mu$ is the expected rate of return.
- $W$ is a stochastic variable following a Wiener process.

$dS$, $dt$ and $dW$ represent an infinitesimally small change in $S$, $t$ and $W$ respectively. The source of uncertainty in the model is $W$: its exact movement is unpredictable. However, statistical description can be given, as it follows the following few rules.

The expectation is zero, it is a martingale:

$$\mathrm{E}[W_t] = 0\ . \tag{2.4}$$

The variance is $t$:

$$\mathrm{Var}(W_t) = t\ . \tag{2.5}$$

Increments have a normal distribution, centered at zero:

$$W_t - W_0 \sim N(0, t)\ . \tag{2.6}$$

**The Heston model**

The following SDE describes the Heston model [28]:

$$dS_t = \mu S_t\, dt + \sqrt{\nu_t} S_t\, dW_t^S, \tag{2.7}$$

$$d\nu_t = \kappa(\theta - \nu_t)\, dt + \xi\sqrt{\nu_t}\, dW_t^\nu, \tag{2.8}$$

$$dW_t^S \cdot dW_t^\nu = \rho\, dt\ . \tag{2.9}$$

The model has the following parameters:

- $\rho$ is the correlation of $W_t^S, W_t^\nu$.
- $\nu_t$, the volatility at time $t$.
- $\theta$, the long variance, or long-run average variance of the price; as $t$ tends to infinity, the expected value of $\nu_t$ tends to $\theta$.
- $\kappa$, the rate at which $\nu_t$ reverts to $\theta$.

- $\xi$, the volatility of the volatility, or 'vol of vol', determines the variance of $\nu_t$.

The model Heston incorporates two distinct random processes: one governing the evolution of the underlying asset and another controlling its volatility. Given that the volatility itself is subject to stochastic behaviour, the model falls under the category of stochastic volatility models. Figure 2.1 provides a visual comparison that highlights the significant differences between the two models discussed in this section.

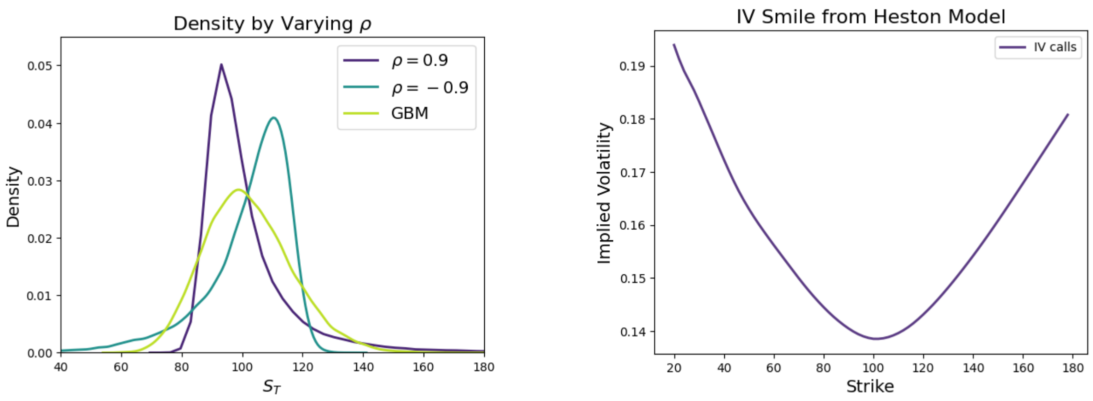

**Figure 2.1:** On the left, two probability density functions were created using the Heston model and different correlation parameters $\rho$ compared with geometric Brownian motion (GBM). On the right, is the volatility smile produced by the model. The strike and $S_T$ is measured in dollars.

## 2.3 Pricing

The use of neural networks for representing functions is underpinned by universal approximation theorems. In 1989, Hornik, Stinchcombe, and White showed that multilayer feed-forward networks, even with just a single hidden layer, can act as universal approximators [30]. This foundational result establishes neural networks as robust tools for high-dimensional function approximation. Subsequent research extended these findings to reveal that neural networks not only have the ability to approximate a given function $f$ but also all of its derivatives up to order $n$ [29]. This capability for accurate derivative approximation is a critical feature that enables the implementation of two-step calibration methods or, in my case, a two-step training procedure. The precise estimation of derivatives ensures that the network can be effectively employed for optimization.

Computationally slow-to-evaluate stochastic models are difficult to integrate in industrial applications. This might limit the model's applicability even if it possesses modelling advantages. To fix this, various solutions have been proposed. Hernandez's work proposes a network $N(w, \theta)$ that learns to represent the inverse pricing function [27]. In deep learning terms a dataset $D = \{(x_i, u_i) : x_i \in \mathbb{R}^m, u_i \in \mathbb{R}^n, x_i = P(M(u_i), \phi), i = 1 \ldots N_d\}$ is created where $x_i$ is one sample of volatilities produced by a model $M$ with parameters $u_i$. The parameters $u_i$ are sampled from a predetermined domain. It is then used to train a simple neural network with the following objective:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \frac{1}{N_d} \sum_{i=1}^{N_d} \left(N(x_i; \mathbf{w}) - u_i\right)^2 \ . \tag{2.10}$$

However, there is no guarantee that two sets of parameters $u_i$ and $u_j$, $i \neq j$, produce the same volatilities. A slight change in one parameter could be offset by slight changes in other parameters. It is not reasonable to expect that for one set of volatilities, only one set of parameters exists that can produce it. Mathematically speaking, we can not expect the inverse pricing map to be injective in the modelled domain [3]. We can not even be sure if an inverse exists. Many functions do not have an inverse but by restricting the domain a partial inverse may exist.

Realizing this problem Horváth et al. proposed a two-step calibration procedure [5]. The first step is learning the pricing function $P$ instead of the inverse of $P$. To achieve this only flipping of the dataset $D$ is required. Now the input of the network $N$ are parameters of the stochastic model and outputs are volatilities. Hence, the new training objective is the following:

$$\mathbf{w}^* = \arg\min_{w} \frac{1}{N_d} \sum_{i=1}^{N_d} \left(N(u_i; \mathbf{w}) - x_i\right)^2 \ . \tag{2.11}$$

The second step is calibration, where they replace the expensive-to-evaluate pricing function $P$ with a neural network $N$:

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \|N(\boldsymbol{\theta}, w) - P_{\mathrm{MKT}}(\phi)\|_2 \ . \tag{2.12}$$

I propose a two-step training procedure to learn a partial inverse pricing function. It is a direct approach like the one proposed by Hernandez but with a remedy to the original paper's shortcomings. First, we normalize the previously described dataset $D$. This means the SDE parameters are transformed using MinMax scaling into the range of $[-1, 1]$:

$$u_i := 2 \times \left(\frac{u_i - \min(\mathbf{u})}{\max(\mathbf{u}) - \min(\mathbf{u})}\right) - 1 \ . \tag{2.13}$$

A network $N : \mathbb{R}^n \to \mathbb{R}^m$ is then trained to represent the pricing function as previously described. Before the second training step, the weights of $N$ are frozen, meaning they will not change during the backward pass. We introduce a secondary network, denoted as $N' : \mathbb{R}^m \to \mathbb{R}^n$, which is initialized randomly. Subsequently, we construct a composite function $N^*(x) := N(N'(x))$. The optimization objective in this context is to approximate the identity function via $N^*$. Given the universal approximation capabilities of neural networks for continuous functions, achieving this goal should be feasible. As the training progresses, $N'$ is tuned to map volatilities to their corresponding parameters. When training converges, applying $N'$ followed by $N$ yields the identity function, confirming that $N'$ serves as a partial inverse of $N$. This approach is particularly effective as $N'$ is capable of autonomously discovering a domain where it can successfully construct this partial inverse. The approximated inverse will be one of several possible partial inverses depending on the random initialization of $N'$. Finally, we train $N^*$ with the following loss function:

$$L(\mathbf{w}) = \frac{1}{N_d} \sum_{i=1}^{N_d} \left(N^*(x_i; \mathbf{w}) - x_i\right)^2 + \frac{1}{N_d} \sum_{i=1}^{N_d} \max\left(0, |N'(x_i; \mathbf{w})| - 1\right) \ . \tag{2.14}$$

To better understand how this network functions, we can draw parallels between it and the autoencoder neural architecture [2]. Autoencoders usually take an image as input, which is then fed into two connected networks called the encoder and the decoder. After the encoder, a bottleneck is introduced by restricting information flow to a few hundred activations. The decoder then tries to decode this hidden state back into the original image. The better the reconstruction the more information is stored in the hidden state. However, this hidden state is usually random and not easily interpretable. We are trying to achieve something similar but we constrain the hidden state by first training on the pricing map. One important consideration is the out-of-sample accuracy of the pricing network $N$. It is a well-known fact that networks do not generalize well. Thus, we must restrict the outputs of the network $N'$ to the domain where we trained $N$. To achieve this a second term is added to the loss in Equation (2.14). Where $|.|$ represents the absolute value function. It is trivial to see that if the hidden state is within the range $[-1, 1]$ then the second term is zero, otherwise some positive number. In simple terms, this soft constraint forces the outputs of $N'$ to be within $[-1, 1]$.

## 2.4 Evaluation and Comparison

Evaluation can be tricky in this problem. It is infeasible to compare the calibration with the parameters we have in the training data as we know that the pricing map might not be injective. To verify the validity of a given calibration, the original pricing function must be rerun. With the implied volatility produced by the Heston model, we can calculate the evaluation metric. I used mean squared and mean absolute error to compare the calibration with the market volatility. The results achieved by two-step training are competitive with the original two-step calibration method proposed in [5]. Moreover, both techniques use the same neural network as pricing map representation their accuracy should be equivalent in most cases. I observed slightly worse calibration with my method. The results of my calibration can be seen in Figure 2.2 and 2.3.

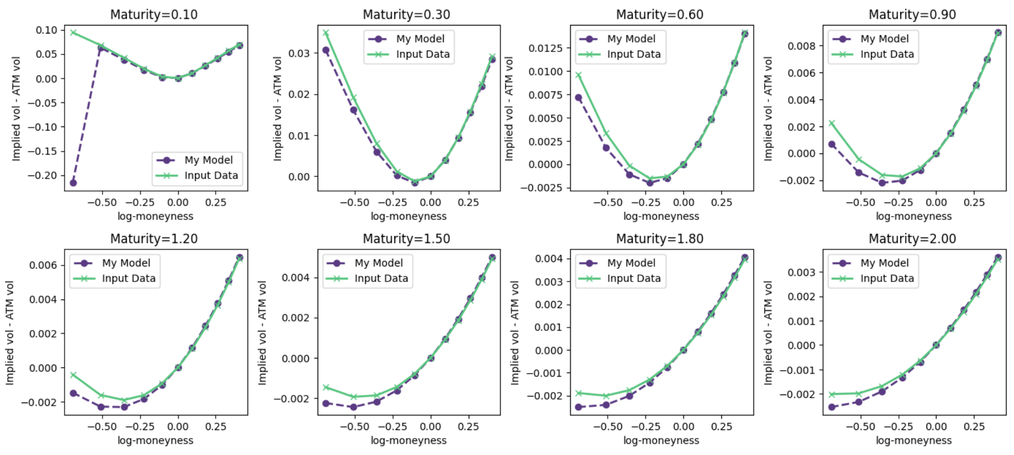

**Figure 2.2:** Calibration for a random sample of the test set calculated with the Heston model. Input data refers to the observed volatilities that we are trying to calibrate to.

The main advantage of the method described here is the speedups achieved by the reduced need for only a single forward pass of the network. In contrast, the prior method necessitated repeated evaluations of the pricing map, coupled with the computation of the Jacobian matrix, often resulting in dozens of iterations before achieving successful

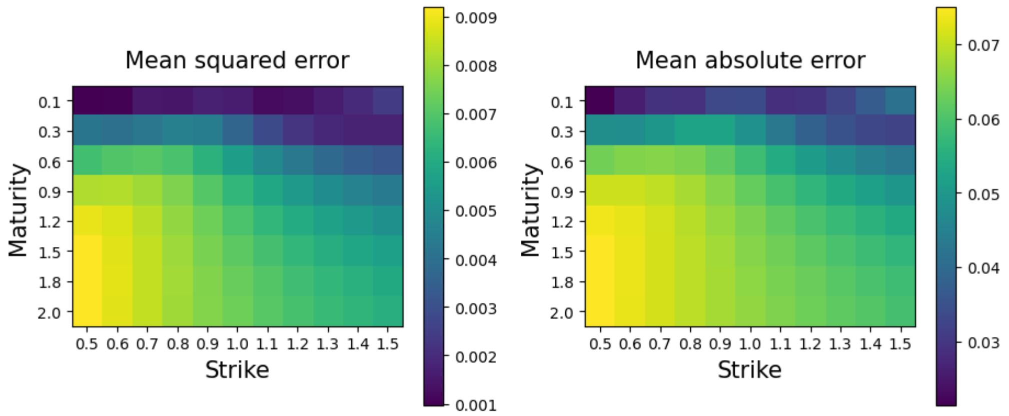

**Figure 2.3:** Distribution of the mean squared and mean absolute errors plotted against different strikes and expiration dates.

calibration. Although automatic differentiation has made the differentiation of networks relatively efficient, the multiple passes intrinsic to these traditional methods remain a significant computational bottleneck. Elimination of the need for multiple evaluations of the network, we can achieve ten to twenty-fold speedup. Moreover, neural networks are highly optimized for parallel computation enabling the calibration of multiple surfaces at the same time. By setting the batch size optimally, say at 1800 in our instance, and processing all test samples concurrently, we can realize calibration speeds that clock in at the microsecond range per sample. This is shown in Figure 2.4.

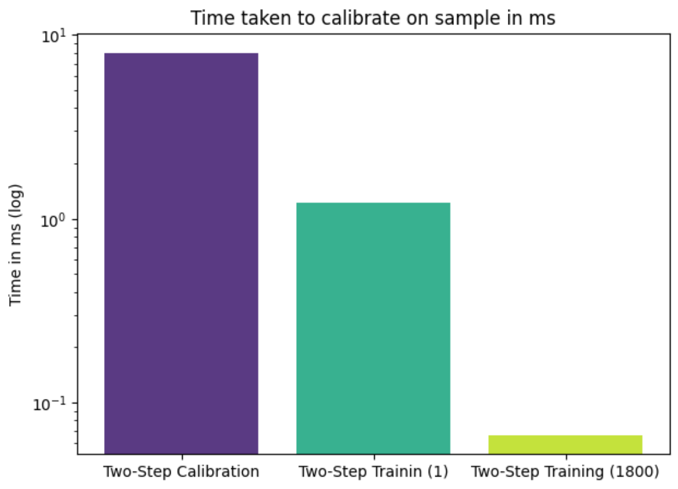

**Figure 2.4:** Time required to calibrate one sample. The batch size that was used for calibration is in the parentheses.

Choosing between the two-step calibration and two-step training methods necessitates the thorough evaluation of one's objectives. If the primary goal is to achieve swift, real-time calibration, particularly in scenarios reminiscent of high-frequency computational tasks, the two-step training approach appears more favourable. However, if you value accuracy and adaptability more, two-step calibration becomes the method of choice. An important technical nuance to remember regarding two-step training is that any modifications to the minimization objective can be intricate, primarily due to the need for model retraining.

Therefore, when implementing a model, these considerations should be the centre of the decision-making process.

## 2.5 Intuition in One-Dimension Using the Sinusoidal Function

It is difficult to visualize the learning using the Heston model since the parameters are in a five-dimensional space. To illustrate the power of this training paradigm we can first look at how it learns the familiar inverse of $f(x) = \sin(x)$. We know that if we restrict the domain of $f$ to $[-\pi/2, \pi/2]$ we can construct the inverse $f^{-1}(x) = \arcsin(x)$. Let's construct a dataset $D = \{(x_i, u_i) : x_i \in [0, 10], u_i = \sin(x_i), i = 1 \dots n\}$. It is trivial to learn the mapping from $x_i$ to $u_i$ but the function is not invertible in the specified domain. If we simply flip the dataset the network fails to learn the inverse. When we apply two-step training we get a random partial inverse. All the described behaviours can be observed in Figure 2.5. The top left image shows the approximation of sine, and the top right image depicts the naive approximation and how it fails to approximate arcsin. The bottom left image shows the random partial inverse found by the training. To better showcase the accuracy, I shifted the partial inverse down so that it lines up with the arcsin function.

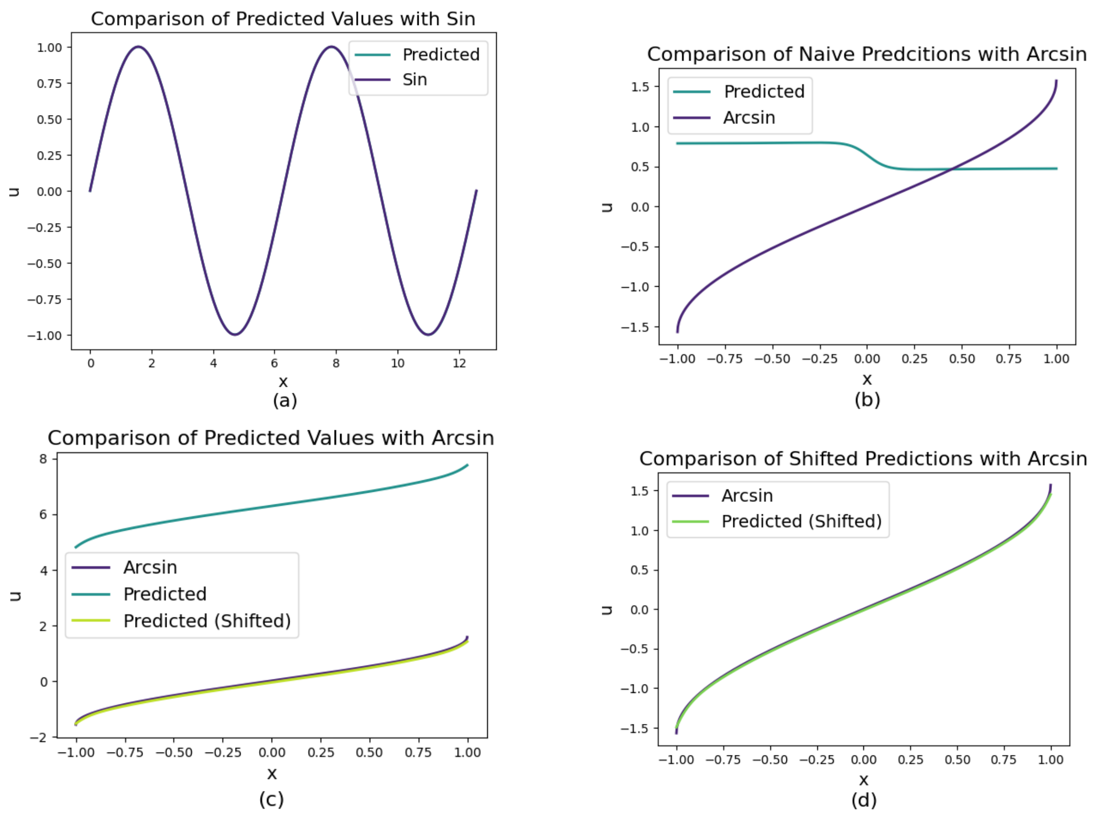

**Figure 2.5:** Neural network representation of sine on $(a)$. The naive solution to the inverse problem fails to learn the function on $(b)$. A random partial inverse was found by the training on $(c)$. The partial inverse found shifted down so that a comparison with arcsin is possible on $(d)$.

## 2.6 Efficient Training

Training neural networks with high precision and efficiency remains a complex task, despite significant advancements in deep learning over recent years. Balancing architecture, hyperparameters, and the vastness of data have driven researchers and practitioners to continuously seek optimal strategies. In light of this, I've assembled a list of best practices. These guidelines focus on accelerating training while improving model accuracy. Additionally, recommendations on architectural choices and hyperparameter settings are provided, based on empirical findings and industry benchmarks.

**Architecture selection** stands as a foundational step in the process of training a neural network. Evidence suggests that deep networks often outperform their wide and shallow counterparts. Corroborating this, Zhou Lu et al. established that there exists a certain category of networks that cannot be replicated by any shallow network, provided its size does not surpass a certain exponential bound [23]. Stemming from this, a deep yet compact network is used in this work. Nevertheless, the construction of deep networks introduces unforeseen complexities. The vanishing and exploding gradient problems emerge as the most pronounced among these challenges just to name a few. To counteract these issues, it is advisable to utilize a deep residual network [14]. This well-known architecture facilitates efficient gradient propagation during the backward pass. Additionally, convolutional networks have been empirically proven to boost accuracy while demanding fewer parameters. Through convolution, these networks adeptly extract spatial information embedded within the volatility surface [7, 43].

**Network initialization** is one of the main innovations that makes training modern deep networks stable and reliable. Glorot et al. described a network initialization scheme for ReLU networks that prevents the magnification and explosion of activations and gradients [24]. On the other end of the spectrum is the vanishing gradient problem that plagued networks that required tanh or sigmoid activations. If not managed carefully, the activations of a neuron could take on extreme values. The derivative of the mentioned activation functions is almost zero if the input is sufficiently large or small. This can lead to neurons getting stuck and unable to learn. An efficient solution is the He initialization [22].

**The Choice of Optimizer** can also lead to significant improvements in performance. Currently, two of the most widely-used optimizers are Adam [34] and AdamW [37]. Both are highly effective for function approximation tasks. The efficiency of these optimizers can be further enhanced by implementing strategies such as learning rate decay or increasing the batch size as training approaches completion.

**Regularization** is a cornerstone of most contemporary neural networks. Owing to their overparameterized nature and the unclear processes they operate by, neural networks are prone to overfitting. Implementing regularization compels them to generalize, subsequently strengthening their robustness and capabilities. Methods such as dropout [17], layer normalization [1], batch normalization [32], weight decay [25], and early stopping [42] are frequently employed to mitigate overfitting. It is important to note that some of these methods serve dual purposes, functioning both as regularization and normalization techniques.

Notably, batch normalization offers advantages beyond mere regularization, such as maintaining a consistent internal distribution of activations. However, based on my personal observations, refraining from regularization often yields superior outcomes. Allowing the network to pointwise converge to the underlying smooth function typically proves to be a better approach. This observation is noteworthy because it challenges prevailing assumptions about the necessity of regularization for effective neural network training, although it should be emphasized that this finding is specific to the context discussed in this work. Conversely, the objective often revolves around aligning with real-world market data, which might not always be immaculate. Monte Carlo pricing can introduce another layer of imperfection; at times, the simulation may not generate a sufficient number of paths to determine a credible implied volatility for far out-of-the-money options. This dilemma is showcased in Figure 2.6.

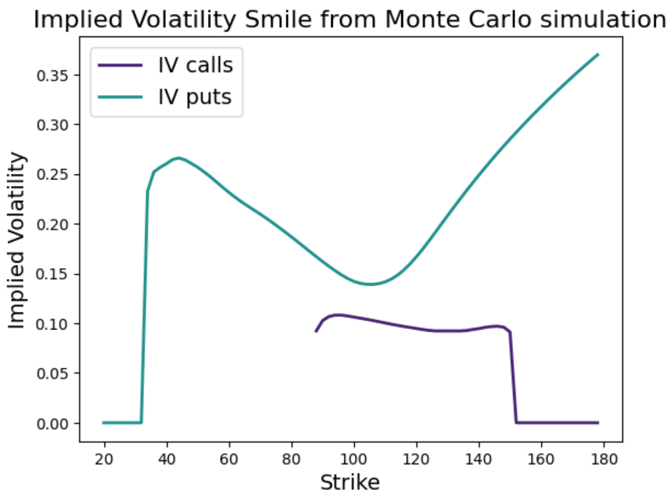

**Figure 2.6:** Broken implied volatilities produced by a Monte Carlo simulation.

# Chapter 3

# Pricing with Physics Informed Neural Networks

Most physical phenomena can be represented using partial differential equations (PDEs). This includes the dynamics of a pendulum, Maxwell's equations governing electromagnetism, and the intricate Navier-Stokes equations that describe fluid flow. Despite the fundamental nature of these equations, solving them remains challenging; for instance, the Navier-Stokes equation has eluded solutions for over a century. Historically, numerous techniques have been proposed to address such PDEs, with Physics-informed Neural Networks (PINNs) emerging as a modern approach to this age-old challenge.

PINNs serve as versatile function approximators designed especially for situations where data is scarce or of dubious quality. By incorporating prior knowledge of the physical system into the network, often in the form of a PDE, it serves as a regularizing force, steering the optimization towards a more universal solution. Essentially, the primary objective of the PINN framework is to emulate the inherent solution dictated by the PDE.

Various domains, be it engineering, biology, chemistry, or meteorology, are underpinned by precise physical laws. The fusion of neural networks with these domains holds the promise of intriguing outcomes. Nonetheless, the area of applying neural networks to these problems is developing, with several challenges hindering the training processes. Addressing these limitations could truly harness the capabilities of neural networks. Conventional methods for solving PDEs without analytical solutions often resort to numerical strategies like the finite element, finite difference, or finite volume methods. Yet, once adeptly trained, neural networks promise significant computational acceleration.

In the subsequent sections of this study, I explore the use of PINNs to approximate the renowned Black-Scholes PDE continuing the theme of financial mathematics. This PDE was derived by Hull [31] from the SDE discussed in Chapter 2. The unique advantage of this equation is that it has a known analytical solution, permitting precise accuracy assessments.

## 3.1 General Setup

The data-driven solution of PDEs starts with the problem definition. Let $u(t, x)$ represent the underlying solution to a system of non-linear partial differential equations. In general, the problem can be described in the following way:

$$u_t + N_x[u] = 0, \quad x \in \Omega, \quad t \in [0, T] \ . \tag{3.1}$$

Here $N_x$ is a non-linear differential operator that can contain second or higher-order derivatives, $u_t$ is the underlying (hidden) solution of the PDE and $\Omega$ is the domain. Without boundary and initial value conditions, there are infinitely many solutions to the PDE. The following equations describe them:

$$u(x, 0) = h(x), \quad x \in \Omega, \tag{3.2}$$

$$u(x, t) = g(x, t), \quad t \in [0, T], \quad x \in \partial\Omega \ . \tag{3.3}$$

Here $\partial\Omega$ denotes the boundary of $\Omega$. Furthermore, $u(x, 0)$ corresponds to the initial value condition that is described by $h(x)$, and $g(x, t)$ describes the behaviour of the function $u$ on the boundary of $\Omega$. Raissi et al. proposed to represent $u(x, t)$ with a neural network $f_{\mathbf{w}}(x, t)$ [39]. We can now define the residual:

$$R_{\mathbf{w}}(x, t) := \frac{\partial}{\partial t} f_{\mathbf{w}}(x, t) + N_x[f_{\mathbf{w}}(x, t)] \ . \tag{3.4}$$

The residual gauges the ability of the network to satisfy the PDE. While manually differentiating intricate functions can be daunting, automatic differentiation offers a streamlined way to handle the differentiation of neural networks with efficiency [11]. Following this, a dataset must be formulated. This is done by randomly selecting points from both the domain and its boundary. With the dataset established, the subsequent step involves defining a loss function. This loss encompasses the combined values from the boundary condition, initial value condition, and the residual. Based on the work done by Raissi et al. we finally arrive at the minimization objectives. For the boundaries, we know the behaviour of the underlying function exactly so the minimization objective is trivial:

$$L_b(\mathbf{w}) := \frac{1}{N_b} \sum_{j=1}^{N_b} \Big[ f_{\mathbf{w}}(x^j, t^j) - g(x^j, t^j) \Big]^2 , \tag{3.5}$$

$$L_i(\mathbf{w}) := \frac{1}{N_i} \sum_{j=1}^{N_i} \Big[ f_{\mathbf{w}}(x^j, 0) - h(x^j, 0) \Big]^2 , \tag{3.6}$$

$$L_r(\mathbf{w}) := \frac{1}{N_r} \sum_{j=1}^{N_r} \left[ \left( \frac{\partial}{\partial t} f_{\mathbf{w}}(x^j, t^j) + N_x[f_{\mathbf{w}}(x^j, t^j)] \right)^2 \right] \ . \tag{3.7}$$

The combined minimization objective can be written down as:

$$L(\mathbf{w}) := \lambda_b L_b(\mathbf{w}) + \lambda_i L_i(\mathbf{w}) + \lambda_r L_r(\mathbf{w}) \ . \tag{3.8}$$

The loss is the weighted sum of the residual, initial, and boundary losses, each scaled by their respective weighting coefficients $\lambda_b$, $\lambda_i$ and $\lambda_r$. These coefficients allow us to control the relative importance of each loss component in the overall optimization process. We can then minimize this loss function using stochastic gradient descent like Adam. One advantage of this method is the ability to freely sample as many points from the domain as needed. Typical millions to hundreds of millions of points are sampled through thousands to hundreds of thousands of epochs [8, 19]. I want to stress that learning is not supervised as there is no exact solution to learn from. The points are randomly sampled and then the constraints are minimized.

## 3.2 The Black-Scholes PDE

In this section, I discuss the details of how to implement the previously described algorithms in the Black-Scholes case. I will denote the underlying solution to the Black-Scoles PDE as $C(S, t)$ since calculations will be done for European call options. The initial value condition and the boundary conditions have very intuitive and easy-to-grasp meanings so I will describe these first. At time $T$, the value of the option can be calculated from the payout function:

$$C(S, T) = \max\{S - K, 0\} \; . \tag{3.9}$$

A call option gives the holder the option to buy the underlying security at the predetermined time and price. In this work, I will consider the underlying security to be a stock. If at $t = T$ the price of the underlying $S$ is above the strike price $K$ the investor can buy it for $K$ and sell it for $S$, hence they made a profit of $S - K$. If the price is below the strike, they will choose not to buy the security and the value of the option is 0. To summarize, the payout of a European call is its initial value condition. The price of the underlying $S$ can take on values $x \in [0, \infty)$. If the underlying's price hits 0 the price of the call is also 0. This follows from the fact that if the price hits zero the company is bankrupt and has zero value left for the shareholders:

$$C(0, t) = 0 \; . \tag{3.10}$$

The last boundary condition is the case when $S \to \infty$. We know that the price follows a Brownian motion with drift $r$ and standard deviation $\sigma$ based on Equation 2.3. Given the known price $S$ at time $t$, the expected value of $S$ at time $T$, denoted as $E[S(T)]$, can be found using the formula:

$$E[S(T)] = S(t)e^{r(T-t)} \; . \tag{3.11}$$

where $r$ is the continuously compounded rate. If the option is sufficiently in the money, the probability $P(S(0) > K)$ approaches 1. So we can calculate the option price without taking into account the non-linear part of the payout function at time $T$. We can then discount the price back to the present value. Putting all this together the call option value for deep-in-the-money options is the following:

$$C(S(T), T) = E[S(T)] - K = S(t)e^{r(T-t)} - K, \tag{3.12}$$

$$C(S(t), t) = S(t) - Ke^{-r(T-t)} \; . \tag{3.13}$$

In financial mathematics the term used to describe $E[S(T)]$ in (3.11) is the forward price. Moreover, the price calculated in (3.12) is the intrinsic value of the option.

Finally, the Black-Scholes PDE can be derived from the stochastic differential equation [31]. The following equation describes the resulting PDE:

$$\frac{\partial C}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 C}{\partial S^2} + rS\frac{\partial C}{\partial S} - rC = 0 \tag{3.14}$$

## 3.3 Automatic Differentiation

Calculation of the constraint seems like a difficult challenge as it involves the calculation of first and second-order derivatives. This would be true for a method like finite differences. Luckily, pytorch [9] allows for automatic differentiation. This makes it possible to calculate the partial derivatives of a function specified by a computer program. The following code snippet describes the calculation of the three partial derivatives that are required to compute the PDE constraint.

**Algorithm 1** Algorithm for Calculating Derivatives

```
1: Input: $input, t, s$
2: $outputs \leftarrow \text{NetworkForwardPass}(input)$
3: $outputs \leftarrow \text{RescaleOutputs}(outputs)$
4: $dVdt \leftarrow \text{Grad}(outputs, t)$
5: $dVdS \leftarrow \text{Grad}(outputs, s)$
6: $d2VdS2 \leftarrow \text{Grad}(dVdS, s)$
```

## 3.4 Gradient Pathologies

Early deep neural networks encountered issues during training, such as the exploding and vanishing gradient problem, mentioned in Chapter 2. PINNs also face these hitches. While they've shown potential in some scenarios, there are times they don't derive the intended solution.

Research by Wang et al. revealed a specific challenge with PINNs [19]. They found that the gradients of $L_r$ can be notably larger than those of $L_b$ or $L_i$. This imbalance in gradient magnitudes can cause the training process to undervalue the boundary and initial conditions. They call this effect stiffness in the training process. Addressing these gradient disparities is vital to ensure PINNs work effectively and provide reliable results.

Building on the observations of Wang et al., my own research on the Black-Scholes PDE further corroborates their results. In my experiment, I subjected a basic neural network to training over 8000 epochs and subsequently examined the gradient statistics for its initial three layers. As depicted in Figure 3.1, there's a clear representation of gradient distributions associated with every component of the loss function. The spread of the histogram offers an insight into the gradient's variance — a wider spread indicates a greater variance in the gradients.

To address this challenge, a straightforward learning rate annealing algorithm was introduced [19]. My experiments validate the efficacy of this method for the specific problem. The algorithm evaluates the gradients for every segment of the loss, subsequently adjusting the individual loss components using the derived statistics using the formula described by Equations (3.16), (3.15) and (3.17). Consistent with their observations, Figure 3.2 distinctly illustrates that the gradients exhibit a normal distribution where all loss components have roughly the same standard deviation.

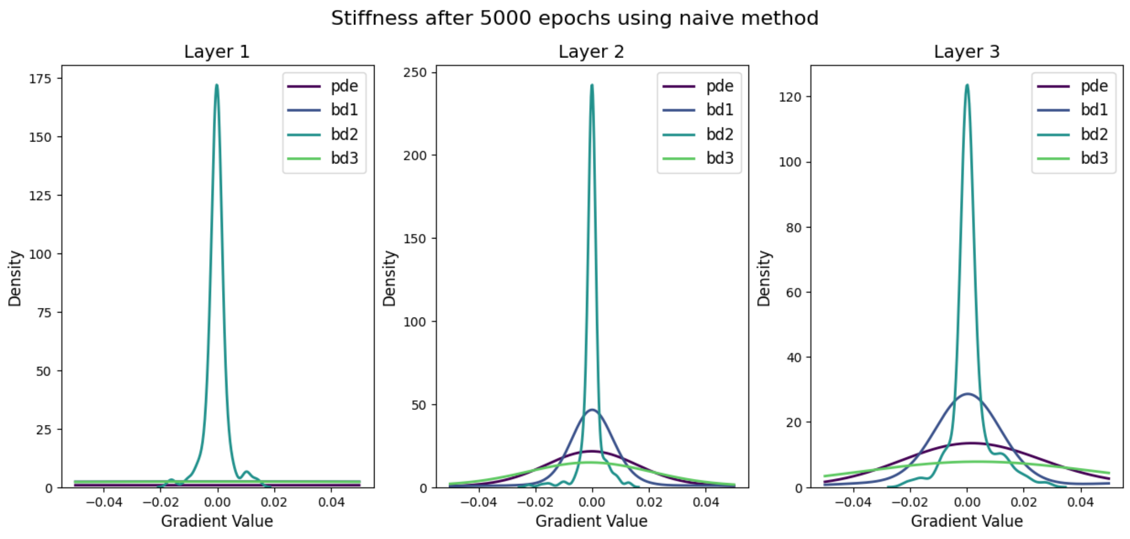

**Figure 3.1:** Stiffness in the training dynamic using a naive approach. $bd_1$, $bd_2$ and $bd_3$ refer to the initial and boundary conditions described in Section 3.2 and *pde* refers to the residual.

$$\lambda_b = \frac{\|\nabla_{\mathbf{w}} L_i(\mathbf{w})\|_2 + \|\nabla_{\mathbf{w}} L_b(\mathbf{w})\|_2 + \|\nabla_{\mathbf{w}} L_r(\mathbf{w})\|_2 + 2}{\|\nabla_{\mathbf{w}} L_b(\mathbf{w})\|_2}, \tag{3.15}$$

$$\lambda_i = \frac{\|\nabla_{\mathbf{w}} L_i(\mathbf{w})\|_2 + \|\nabla_{\mathbf{w}} L_b(\mathbf{w})\|_2 + \|\nabla_{\mathbf{w}} L_r(\mathbf{w})\|_2 + 2}{\|\nabla_{\mathbf{w}} L_i(\mathbf{w})\|_2}, \tag{3.16}$$

$$\lambda_r = \frac{\|\nabla_{\mathbf{w}} L_i(\mathbf{w})\|_2 + \|\nabla_{\mathbf{w}} L_b(\mathbf{w})\|_2 + \|\nabla_{\mathbf{w}} L_r(\mathbf{w})\|_2 + 2}{\|\nabla_{\mathbf{w}} L_r(\mathbf{w})\|_2}. \tag{3.17}$$

Here $\nabla_{\mathbf{w}} L_i(\mathbf{w})$ denotes the gradients of the loss $L_i$ with respect to $\mathbf{w}$.

The original authors suggested recalculating the training statistics every few hundred to a thousand epochs. This recommendation is made to mitigate performance degradation, as this computation can be resource-intensive. Additionally, they highlighted that the parameters $\lambda_b$, $\lambda_i$, and $\lambda_r$ might exhibit somewhat stochastic behaviour, oscillating during the training process. When utilizing an optimizer that incorporates gradient statistics, such as momentum or Adam, these constant and erratic fluctuations could potentially impede the training progress.

The aforementioned algorithm enabled the training of the network. However, I observed that the learning still tends to stagnate and fails to converge. The primary obstacle was identified to be the boundary conditions. In their absence, a PDE can have infinitely many solutions. Once the losses are rescaled, the boundary conditions face challenges in rivalling the residual loss. This results in a plateau in the boundary losses while the residual loss keeps reducing. Consequently, achieving convergence to the accurate solution becomes unattainable.

I propose introducing a correction term to their algorithm to hasten the convergence of the boundary and initial value conditions in the early phases of the training. Once these conditions begin to converge, the correction term diminishes in magnitude, allowing the learning rate annealing algorithm to take the forefront. My experiments indicate that this adjustment significantly boosts performance. The results of this modification are illustrated in Figure 3.3.

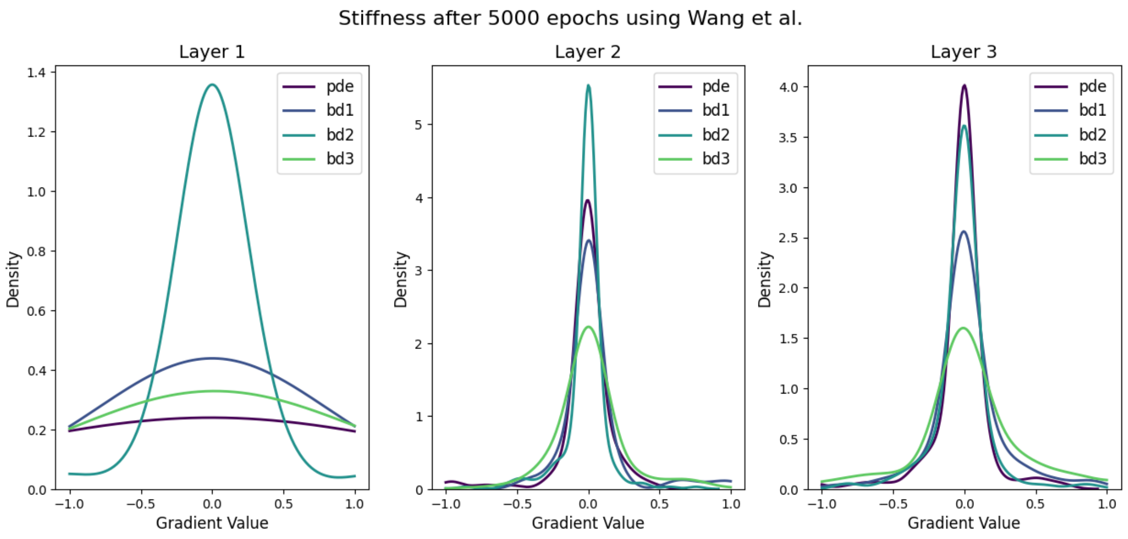

**Figure 3.2:** Stiffness in the training dynamic after normalization.$bd_1$, $bd_2$ and $bd_3$ refer to the initial and boundary conditions described in Section 3.2 and *pde* refers to the residual.

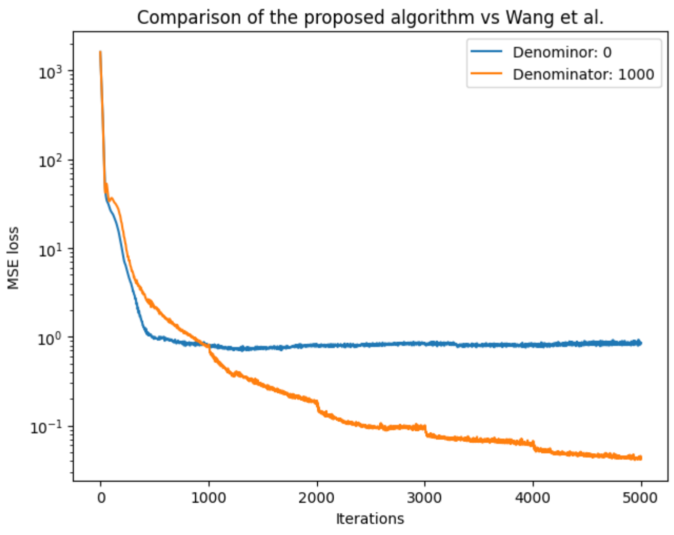

**Figure 3.3:** Comparison of the learning rate annealing algorithm proposed by Wang et al. in blue and my algorithm in orange.

$$\alpha_i := \theta/(\theta + \text{epoch}) + 1, \tag{3.18}$$

$$\alpha_b := \theta/(\theta + \text{epoch}) + 1, \tag{3.19}$$

$$L(\mathbf{w}) := \alpha_b \lambda_b L_b(\mathbf{w}) + \alpha_i \lambda_i L_i(\mathbf{w}) + \lambda_r L_r(\mathbf{w}) \; . \tag{3.20}$$

This straightforward extension necessitates a single parameter $\theta$. When $\theta$ is set to zero, the optimization process remains unchanged compared to the originally proposed algorithm. As the number of epochs progresses, the magnitude of the correction terms diminishes.

Integrating these methodologies led to rapid, efficient, and stable training. As demonstrated in Figure 3.4, the learning curves are outlined for both training and test losses. Given that we are working with the Black-Scholes PDE, an analytical solution exists. This permitted the computation of exact prices for the test set, facilitating the derivation of a mean squared error loss. In contrast, for the training data, the boundary and residual losses are displayed.

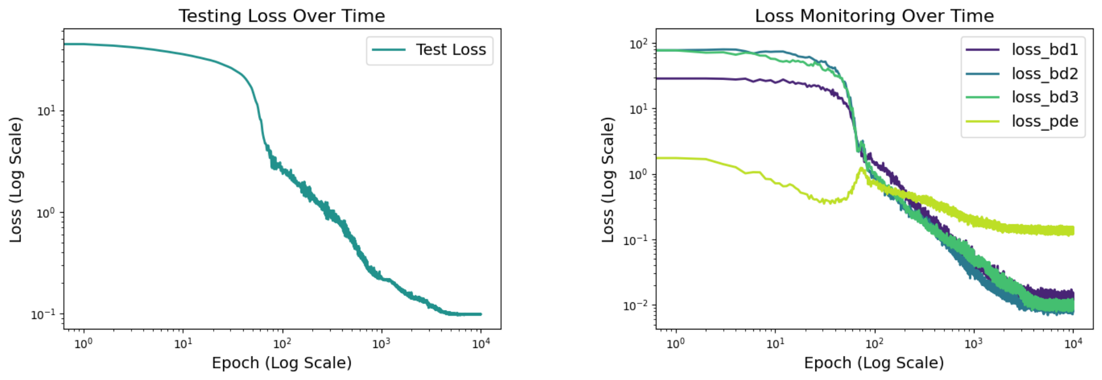

**Figure 3.4:** On the left: mean squared test error. On the right: the training loss by components. The sum of each individual component would be the total training loss.

## 3.5 Efficient Training

As explained in Chapter 2, effective training hinges on architecture design. In the scope of this research, I employed a residual network equipped with ReLU activations. This simple feed-forward architecture was subsequently trained using the Adam optimizer, learning rate decay, and the methodology introduced in this study. A valuable tactic to boost performance, especially when Adam's convergence stalls, is to resort to a quasi-Newton algorithm instead of Adam for optimization purposes. Torch offers an optimizer, L-BFGS [36], which can be leveraged to further reduce the training error. I must note, however, that this algorithm is inherently crafted for full batch learning, thereby preventing the feasibility of employing mini-batches for network training. Furthermore, based on my empirical observations, the learning process occasionally diverges owing to numerical instabilities, leading to non-numeric values.

To further optimize performance, I conducted a hyperparameter search with the sole goal of reducing the test error. The results of which are detailed in Table 3.1.

The selection of the appropriate domain is key for the training of neural networks. It is important to ensure that input values lie within the range of [-1, 1]. This normalization is vital as it prevents activation values from becoming excessively large, which can lead to erratic training dynamics. One common approach to achieve this normalization is by employing MinMax or some other scaling on the inputs.

Beyond normalization, there are other considerations linked to the nature of the problem at hand. For example, working with large spot prices can pose challenges. Even though

| **Hyperparameter** | **Value** |
|---|---:|
| Width | 16 |
| Depth | 14 |
| Residual blocks | 6 |
| Epochs | 20000 |
| Learning rate (start) | 0.001 |
| Learning rate decay step size | 8000 |
| Learning rate decay rate ($\gamma$) | 0.4 |
| Batch size | 512 |
| $\theta$ | 1000 |

**Table 3.1:** Hyperparameters and their corresponding values for the model training.

high spot prices can be representative of real-world scenarios, they can introduce complexities in training. Specifically, call values derived from these high spot prices can be quite large, potentially hindering the network's convergence. Therefore, to ensure both representativeness and computational efficiency, it is advisable to be careful in choosing spot prices for neural network training. This applies to the other parameters as well to a lesser extent.

Thus, it is not merely about representing the real-world scenario but also about ensuring that the model's learning process remains efficient and stable. Striking this balance, I have confined my experiments within the domain specified in Table 3.2. This domain offers a reasonable mix of practical relevance and computational efficiency.

| **Parameter** | **Range** |
|---|---:|
| Spot Price ($S$) | [0, 20] |
| Strike Price ($K$) | 5 |
| Expiration ($T$) | [0.1, 10] |
| Interest Rate ($r$) | [0.005, 0.1] |
| Volatility ($\sigma$) | [0.001, 0.2] |

**Table 3.2:** Parameters and their corresponding ranges.

## 3.6 Results

Evaluating the results proved to be challenging due to the significant influence of the neural network's initialization on learning. For a credible comparison between distinct techniques, it was essential to maintain consistency in initialization. With an optimized architecture and carefully calibrated hyperparameters, the effectiveness of the learning rate annealing method was evident. This is exhibited in Figure 3.5, where an impressive decrease in the test error can be observed.

One notable drawback of the proposed method is the sudden spiking in test error when gradient statistics are recalculated for the learning rate annealing algorithm, as illustrated in Figure 3.5. This surge in error can be attributed to sudden changes in the backpropagated gradients. While the error usually subsides after a few iterations, as the optimizer adjusts to the updated optimization objective, this transient instability presents a significant challenge to the method's reliability.

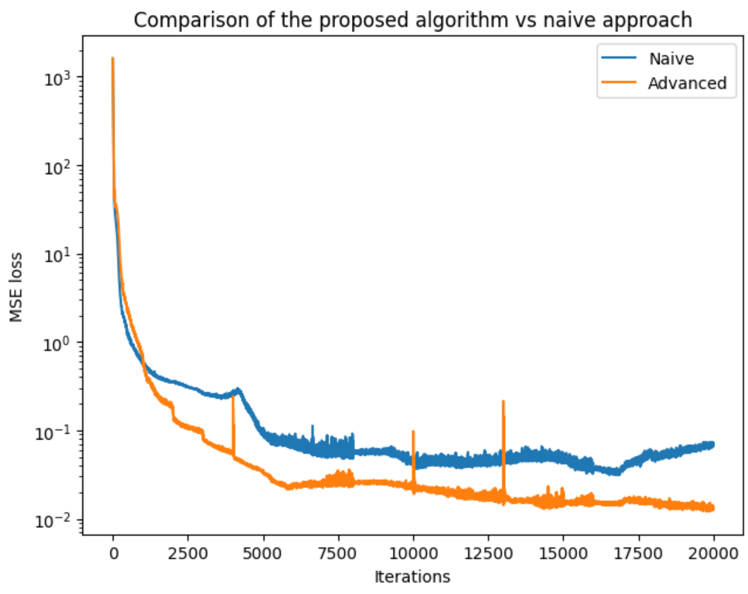

**Figure 3.5:** Comparison of my work (Advanced) and no learning rate annealing (Naive). Note that the y-axis is a log scale.

This is particularly true for optimizers that are dependent on gradient statistics, such as Adam and AdamW. In Figure 3.6, the probability density of the training process reaching a specific loss level for both methods is shown. This graph reveals that although the advanced method can achieve higher accuracies, the highly variable distribution indicates instability is still present. Future research is required to mitigate this limitation of the algorithm. One potential approach is to employ a learning rate warm-up following each update of the parameters $\lambda_b$, $\lambda_i$, and $\lambda_r$. Another strategy could involve synchronizing the learning rate decay steps with these parameter updates. Such synchronization would ensure that a reduced learning rate is in effect when gradient statistics are recalculated, thereby minimizing the risk of test loss spiking.

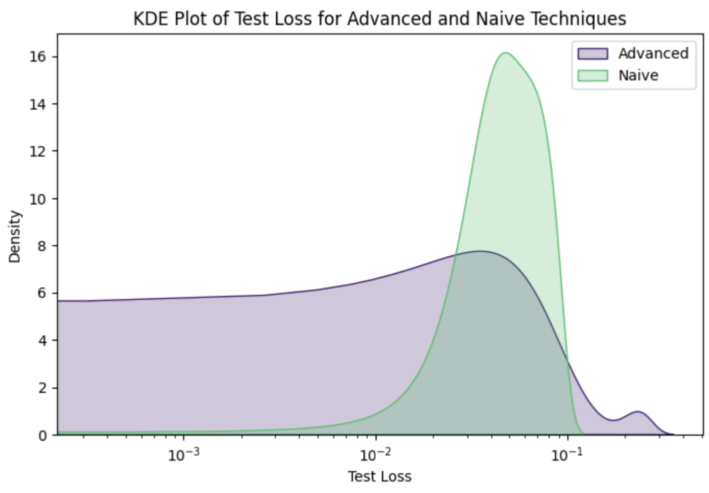

**Figure 3.6:** Comparison of expected loss after 20,000 iterations of the two methods.

Afterwards, I showed that the model's learning genuinely converges across the entire training domain. To accomplish this, I computed the mean absolute error across a two-dimensional cross-section of the domain. This data was then visualized as a heat map in Figure 3.7. While the accuracy appears to diminish for elevated spot prices, it's important to stress that with a fixed $K$ (strike price), the associated call option's price will naturally be more significant.

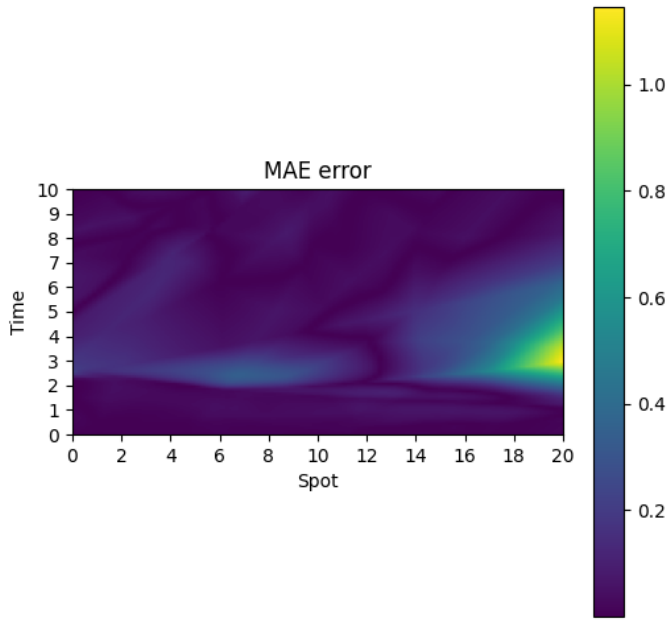

**Figure 3.7:** Mean absolute error heat map for fixed $r$, $\sigma$ and $K$.

In the end, the convergence of the network to the inherent solution is demonstrated in Figure 3.8. By holding all parameters constant, save for one, interpolation is carried out over the designated range. Within the plot, the neural network's approximation is labelled as "Predicted" while the true analytical solution is denoted as "Actual". This visually confirms the network's ability to closely match the genuine solution.

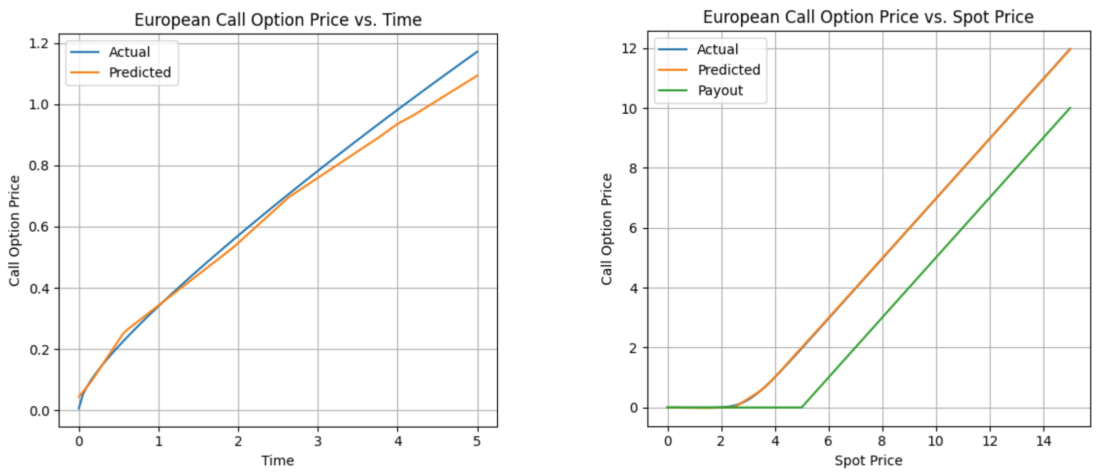

**Figure 3.8:** On the left: interpolation of time for a given set of $K$, $S$, $r$ and $\sigma$. On the right: interpolation of the spot price for a given set of $K$, $t$, $r$ and $\sigma$

# Chapter 4

# Conclusion and Future Work

In this comprehensive two-part study, I explored the capabilities of neural networks as universal function approximators for high-dimensional problems in the realm of finance. I introduced an innovative algorithm that promises a thousand-fold speed enhancement in the field of financial model calibration and inverse function approximation. I explored the use case and compared it to previous state-of-the-art solutions. Furthermore, I underscored the effectiveness of the PINN framework in accurately approximating the Black-Scholes equation. I validated the accuracy of my findings through comparison with the analytical solution. Additionally, I conducted an extensive hyperparameter tuning to optimize model performance. Building upon the learning rate annealing algorithm proposed by Wang et al., I introduced enhancements that demonstrated the algorithm's efficacy in enabling convergence of PINNs. I also showed that PINNs can converge even when the initial condition is not differentiable.

Building on this foundation, the research emphasizes the pivotal role of architecture selection, domain considerations, and normalization in ensuring efficient training. The experiments revealed that meticulous attention to these factors, coupled with the proposed algorithm, not only speeds up the training process but also ensures robust convergence, as evidenced by the comparative analyses. Through extensive experimentation and evaluations, this work makes a compelling case for the transformative potential of neural networks in addressing these intricate computational challenges.

Future work for both methodologies presented here holds immense potential. While the PINN framework might find some limitations in finance, given the sector's demand for ultra-precise outcomes and clear interpretability, other domains could benefit substantially from its application. Specifically, the realms of fluid dynamics and various physical systems stand to gain significantly. The capability of the PINN framework to model complex systems and capture intricate details places it as a promising tool in these fields. Simulating phenomena that have traditionally been computationally intensive or challenging could become more efficient and accurate, unlocking new possibilities and insights for researchers. There also exists a promising avenue for combining the two methodologies presented in this work. Specifically, this combination could offer a robust solution to ill-posed inverse problems. In such cases, the underlying physical system could be modelled using PINNs, while the inverse problem could be approximated using the method proposed herein.

The inverse approximation method proposed in this work is suitable for financial model calibration, however, the industry values precision above all else. Regulation and difficulty of interpretation might hinder its use cases. A more suitable environment would be where real-time calibration is essential. This could include rapidly changing systems where constant measurement is needed to control their evolution.

# Acknowledgements

Embarking on this journey would have been challenging without the unyielding support and invaluable contributions of numerous individuals who have provided not only their expertise but also unwavering encouragement.

Firstly, I express my heartfelt gratitude to Márton Sasvári and Viktoria Hunyadi, whose support and knowledge helped me greatly improve my understanding of this complicated subject. My sincere thanks extend to my great friend Máté Egri for his invaluable technical guidance and pragmatic insights into my research. His mathematical point of view helped steer this research in the right direction. I am profoundly grateful to Tamás Mészáros for his unyielding support and mentorship throughout this journey. A special thanks to my advisor, Balázs Renczes, who assisted me with the writing and technical details.

# Bibliography

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

[2] Dana H. Ballard. Modular learning in neural networks. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, page 279–284. AAAI Press, 1987.

[3] Christian Bayer and Benjamin Stemper. Deep calibration of rough stochastic volatility models. Technical report, Weierstrass Institute for Applied Analysis and Stochastics and Department of Mathematics, Technical University Berlin, 2018.

[4] Fisher Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[5] Aitor Muguruza Blanka Horvath and Mehdi Tomas. Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models. *Quantitative Finance*, 21(1):11–27, 2021.

[6] Prafulla Dhariwal and Alex Nichol. Diffusion models beat gans on image synthesis. In *Advances in Neural Information Processing Systems*, volume 34, pages 8780–8794. Curran Associates, Inc., 2021.

[7] Georgi Dimitroff, Dirk Röder, and Christian P. Fries. Volatility model calibration with convolutional neural networks. *SSRN Electronic Journal*, 2018.

[8] Michael Eidell, Rekha Mukund, and Sanjay Choudhry. Accelerating product development with physics-informed neural networks and NVIDIA modulus, 2021. URL `https://developer.nvidia.com/blog/accelerating-product-development-with-%physics-informed-neural-networks-and-modulus/`.

[9] Adam Peszke et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[10] Alhussein Fawzi et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 2022.

[11] Atılım Gunes Baydin et al. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(1):1–43, 2018.

[12] David Heath et al. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica*, 60(1):77–105, 1992.

[13] Juho Lee et al. Set transformer: A framework for attention-based permutation-invariant neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 3744–3753, 2019.

[14] Kaiming He et al. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[15] Karthik Kashinath et al. Physics-informed machine learning: Case studies for weather and climate modelling. *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences*, 379(2194), 2021.

[16] Kathryn Tunyasuvunakool et al. Highly accurate protein structure prediction for the human proteome. *Nature*, 596:590–596, 2021.

[17] Nitish Srivastava et al. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[18] Shengze Cai et al. Physics-informed neural networks (pinns) for fluid mechanics: a review. *Acta Mechanica Sinica*, 37:1727–1738, 2021.

[19] Sifan Wang et al. Understanding and mitigating gradient flow pathologies in physics-informed neural networks. *SIAM Journal on Scientific Computing*, 43(5):A3055–A3081, 2021.

[20] Sifan Wang et al. An expert's guide to training physics-informed neural networks. *arXiv preprint arXiv:2308.08468*, 2023.

[21] Tom Brown et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. More details needed on exact date.

[22] Yuhuang Hu et al. Overcoming the vanishing gradient problem in plain recurrent networks. *CoRR*, 2018. Institute of Neuroinformatics, University of Zurich and ETH Zürich, Zürich, Switzerland.

[23] Zhou Lu et al. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, volume 30, pages 6231–6239. Curran Associates, 2017.

[24] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*, volume 1. MIT Press, Cambridge, MA, 2016.

[26] P. Hagan, D. Kumar, A. Lesniewski, and D. Woodward. Managing smile risk. *Wilmott Magazine*, 2002.

[27] Andres Hernandez. Model calibration with neural networks. 2017.

[28] Steven L. Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *The Review of Financial Studies*, 6(2): 327–343, 1993.

[29] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[30] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(11), 1990.

[31] John C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, 2012.

[32] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, volume 37 of *ICML'15*, pages 448–456. JMLR.org, 2015.

[33] Eric Kauderer-Abrams. Quantifying translation-invariance in convolutional neural networks. *arXiv preprint arXiv:1801.01450*, 2018.

[34] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA*, 2015.

[35] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

[36] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45:503–528, Aug 1989.

[37] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA*. OpenReview.net, 2019. Conference dates: 06/05/2019 - 09/05/2019.

[38] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[39] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*, 2017.

[40] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017.

[41] Maziar Raissi, Paris Perdikaris, and George 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*, 378:686–707, 2019.

[42] Ruoqi Shen, Liyao Gao, and Yi-An Ma. On optimal early stopping: Over-informative versus under-informative parametrization. *arXiv preprint arXiv:2202.09885*, 2022.

[43] Henry Stone. Calibrating rough volatility models: A convolutional neural network approach. *arXiv preprint arXiv:1812.05315*, 2018.