A Differential Machine Learning Approach for Calculating Deltas under the Heston Model

Elain Balderas Nina McClure Akash Yadav

June 4, 2023

Barcelona School of Economics
Master Program in Data Science Methodology
June 4, 2023

Contents

List of Figures

List of Tables

Abstract

This project introduces a novel technique called "differential machine learning" to estimate option prices and their deltas under the Heston model. Although option pricing under the Heston model has been done in the past, predicting sensitivities like deltas is not as straightforward, as it involves computing the derivatives of an estimated price function, which are historically not good approximates. Differential machine learning aims to remedy this issue with the aim of approximating the shape of the underlying function, by training on differentials of the target variable with respect to the input variables. Using a twin network architecture, we find that training on pathwise differentials and option price labels yields significantly better results compared to training on price labels alone. Moreover, the performance of the differential twin network surpasses standard training whether we use a train-test split on a dataset generated by Monte Carlo simulations, or a noisier training set generated via the least squares Monte Carlo method. Notably, our network is capable of computing both prices and deltas much faster in a single pass.

Acknowledgements

We would like to thank our advisor, Professor Eulalia Nualart, and our TA, Mr. Makar Pravosud for their guidance. We would also like to acknowledge Mr. Antoine Savine for providing invaluable insights for our project and possible future improvements.

1 Introduction

Estimating prices and price sensitivities of financial derivatives is an important topic in quantitative finance. Although simple models with closed-form solutions exist, such as under the canonical Black-Scholes model, more realistic models and those applicable to a wide range of instruments generally do not have solutions that can be computed analytically. The use of deep learning models to solve such problems computationally has become popular in recent decades.

We use a novel machine learning method, differential machine learning, attributed to Huge and Savine (2020)[9]. We predict option prices and option price sensitivities, known as *Greeks*, for European call options under Heston's model of stochastic volatility via a differential deep learning model. In particular, our main goal is to accurately predict *delta*, which is arguably the most important of the Greeks, and represents the sensitivity of the option price with respect to the underlying asset price. In essence, differential machine learning involves training a model with differentials of the target variable with respect to the input variable/s in order to better estimate the shape of the unknown pricing function that one is trying to approximate.

Estimating price sensitivities is not a straightforward task, even with the aid of sophisticated models such as neural networks. Option prices are easier to estimate and have been successfully estimated in the past, which is why they are not the focus of our research. But, due to the nature of our model, prices are predicted along with the *deltas* simultaneously. A previous attempt to estimate the Greeks under the Black-Scholes model via a neural network was unsuccessful [5]. Since then, *delta* under Black-Scholes has been successfully estimated using differential machine learning applied to a twin neural network in Huge and Savine (2020). We extend the work of Huge and Savine (2020) to estimate *delta* under a more complex and realistic pricing model: Heston's model of stochastic volatility. While there are pseudo closed-form solutions for option prices under Heston, there are no computationally efficient solutions for the Greeks.

Option prices and Greeks can be estimated with a high degree of accuracy via Monte Carlo methods. However, such methods are computationally costly which can render them useless in the context of financial risk management, where price sensitivities may need to be computed in close to real-time. Although neural networks can be fast to train and produce predictions, in this context of computational finance they are typically trained using data simulated via slow Monte Carlo methods, which may prevent their use in an online learning setting. We also approach this problem in a different way, learning from data generated by the least squares Monte Carlo (LSM) method of Longstaff-Schwartz (2001), which speeds us computation time by orders of magnitude.

This paper proceeds as follows: we first provide a discussion of the relevant literature in Section 2 followed by a brief overview of the Heston model of stochastic volatility in Section (3). Next, we delve

into the theoretical underpinnings of our work, including the LSM method and Monte Carlo simulations used to generate our data in Sections 4 and 5. Section 6 describes differential machine learning and the architecture of the twin neural network.[1] We present our results in Section 7 and provide a discussion of limitations and extensions in Section 8. Section 9 concludes.

2 Literature Review

The use of deep learning in the approximation of option prices and their Greeks has been a point of interest in the field of computational finance since the 1990s (Ruf and Wang, 2020) [14]. As emphasized by Huge and Savine (2020) [9], neural networks approximate prices more effectively than traditional machine learning methods. This is because the neural network "resolves the curse of dimensionality" by learning the essential basis functions which best approximate the target function *from the data itself*. This is in stark contrast with the way classic regression techniques break down when parameters exceed the sample size, as issues of overfitting may arise. A neural network, on its own, finds the appropriate low-dimensional regression space from the data as a dimensionality reduction feature. Its data-driven method of extracting basis functions from the data outperform features that were chosen from prior knowledge. At the same time, the architecture can remain unchanged even if the input dimension is increased, and the neural networks' connection weights grow linearly (and not polynomially) as the sample size increases. It is therefore no surprise that several studies have explored different approaches to explore the benefits of neural networks to accelerate European option pricing.

These ideas can be broadly categorized into two main categories. The first approach focuses on neural networks for model calibration, specifically utilizing neural network regression to fit model-free non-parametric pricing functions or retrieve parameters from observed data [13] [15] [12]. Cuchiero et al. (2020)[6] introduces a fully data-driven approach for calibrating local stochastic volatility models. They parametrize the leverage function using feed-forward neural networks, enabling direct learning of network parameters from market option prices. This approach leverages the power of neural stochastic differential equations (SDEs) and generative adversarial networks (GANs) to generate volatility surfaces. Meanwhile, Liu et al. (2019) [11] propose the calibration neural network (cANN) for calibrating financial asset price models using an artificial neural network (ANN). The approach utilizes a machine learning framework that combines an offline training phase for the ANN's weights with an online evaluation phase, facilitating efficient and accurate calibration of high-dimensional stochastic volatility models. Horvath et al. (2019) [8] propose a neural network-based calibration method that achieves rapid calibration of the full implied volatility surface across various volatility models and derivative contracts.

The second approach focuses on accelerating classical solvers for partial differential equations (PDEs) through deep learning techniques. In this regard, the main objective is to estimate option prices and their respective Greeks using financial models. The exploration of estimating Greeks, especially in models

[1] The code used for this paper can be found on GitHub via this link: https://github.com/elainbrianne/DSMMasterProject23.

lacking closed-form solutions, remains an area of substantial untapped potential within the existing literature. Liu et al.'s work (2018) [10] proposes the use of ANNs to expedite the valuation of financial options and computation of implied volatilities. By training an optimized ANN on a data set generated by the Black-Scholes and Heston models, the method leverages the universal function approximation capability of ANNs to serve as an efficient agent of the original solver. Their experimental results demonstrate that the ANN solver significantly reduces computing time across various solvers, including the Black-Scholes equation, and the Heston stochastic volatility model using the COS method.

Our paper aims to build upon Liu et al (2018)[10]'s work by proposing the use of differential machine learning for pricing options and estimating Greeks. We address a limitation of ANNs, where their performance may be hindered when predicting both prices and Greeks. ANNs compute the derivatives of value predictions to generate Greeks, but since value predictions are only estimates, the quality of these predictions affects the accuracy of the derivatives. Differential machine learning presents a solution by approximating prices from the Greeks, which generally provides good predictions of the values. By training on differentials and utilizing a twin network, we can construct the Greeks directly and obtain accurate approximations of option prices. In theory, a twin network can still approximate prices from differentials, yielding better value predictions for more complex pricing models with challenging derivatives.

Furthermore, the twin network architecture for differential machine learning proposed by Huge and Savine (2020)[9] offers a faster method for computing Greeks in higher-dimensional models without closed-form solutions. This approach is more efficient and convenient, as it trains and outputs option prices and Greeks simultaneously using a single architecture. The twin network combines feedforward and backpropagation equations, allowing for the computation of both price predictions and their differentials (the Greeks) in a single pass.

3 The Heston Model

The Heston model is defined by the two-dimensional stochastic differential equation (SDE):

$$dSt = \sqrt{v_t} S_t dW_t \tag{1}$$

$$dv_t = k(\theta - v_t)dt + \epsilon\sqrt{v_t} dB_t \tag{2}$$

where k, θ, ϵ are strictly positive constants, W and B are scalar Brownian motions in some probability measure. We assume that $d\langle W, B\rangle_t = \rho dt$ for the correlation parameter $\rho \in (-1, 1)$. In this case, S_t represents the asset price process that is assumed to be martingale. Moreover, $\sigma = \sqrt{v_t}$ is the square root of a Cox-Ingersoll-Ross (CIR) process, and represents the instantaneous variance (volatility) of relative changes to S_t.

Calculating the Greeks in the Heston model, including the *delta*, remains a computationally challenging

task due to the model's complexity.

4 Pricing Function Approximation

Let us restate our goal of pricing approximation in mathematical terms:

Approximate asset pricing function $f(x)$ of a set of inputs x (instrument parameters, model and market variables) with a function $\hat{f}(x; w)$ subject to adjustable weights w, learned from a set of m instances of inputs $x^{(i)}$ (each with dimension n) paired with labels $y^{(i)}$ (in our case $\in \mathbb{R}$) by minimizing a cost function (also called a loss function).

To this end, we will focus on learning more efficiently, in terms of computation times, by training approximations on *sampled payoffs* in place of ground truth option prices [1].

4.1 Least Squares Monte Carlo

Regular Monte Carlo schemes to obtain option prices and deltas require simulating multiple asset prices and volatility paths. This process is extremely slow. We propose an alternative method using Longstaff-Schwartz' (2001) prescription of Least Squares Monte Carlo (LSM). In this method, sample datasets are produced for the computation cost of one Monte Carlo path, where each example is not a ground truth price, but one sample of the payoff. This produces intermediate and potentially noisy approximations of the payoff which are then fed into a regression model (or, in our case, the twin neural network).

From the fundamental theorem of asset pricing, we know that true option price is the discounted sum of its expected future payoffs, i.e.

$$V^{(i)} = E[\ payoff\ |X^{(i)}]$$

where $X^{(i)}$ is the input example. Traditional Monte Carlo involves averaging payoffs over a number of simulations all identically seeded based on $X^{(i)}$. This is also called *nested simulations* because a set of simulations is necessary to compute the value of each example, with the initial states being sampled as well.

Instead, for each instance i, we draw one single payoff from its distribution conditional on $X^{(i)}$. The labels $Y^{(i)}$ in our training dataset correspond to these random draws. Notice that labels are unbiased, albeit noisy, estimates of true prices:

$$E[Y^{(i)}] = E[\ payoff\ |X^{(i)}] = V^{(i)}$$

[1] Note: We will also present results learning from option prices to demonstrate the logic that training with differential labels will improve our results. However, we will not be using the full computational benefits of our twin neural network.

This is why universal approximators trained on LSM datasets converge to true prices despite having never seen one.

In our implementation of LSM, we need to use proper discretization scheme for our Heston model SDEs. Here, we deploy the 4th order Runge-Kutta (RK4) method where we combine a weighted sum of several estimates of the change over one time step. Details of this discretization scheme are in Appendix A.2 and a detailed review is presented by Butcher (2015). [4]

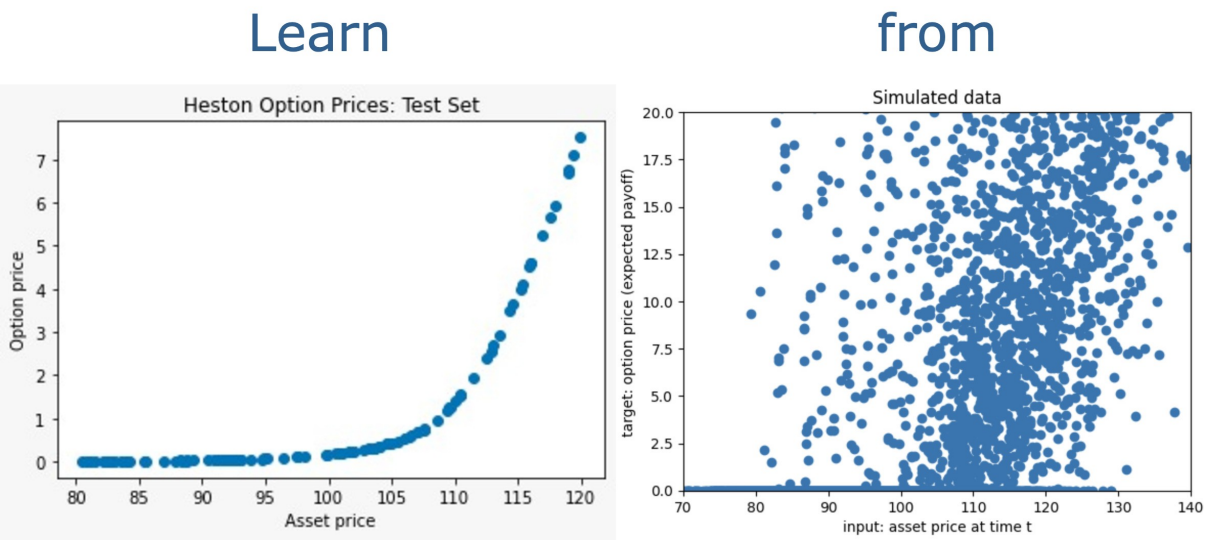

Figure 1: Learn the correct pricing function from a noisy LSM dataset

4.2 Universal approximation theorem

For the sake of completeness, we begin with the definitions of an approximator and capacity of the approximator.

An **approximator** is defined as functions $\hat{h}(x, w)$ of the input vector x, parameterized by a vector w of *learnable weights* of dimension d. For example, in classic regression, regression weights are often denoted by β. In neural networks, all connection matrices and bias vectors in the multiple layers of the network constitute w.

The **capacity** of the approximator can be understood as the measure of its computational complexity and its ability to approximate functions by matching discrete sets of data points. The classic formal definition of capacity is the Vapnik-Chervonenkis dimension. In our case, we use a weaker definition as the number of learnable parameters.

The theorem then states:

A universal approximator $\hat{f}$ trained by minimization of mean squared error (MSE) over a training set $X^{(i)}, Y^{(i)}$ of independent instances of states coupled with conditional payoffs, converge to the true pricing function when the size of the training set and capacity both grow to infinity.

Using the LSM approach, our training set of sampled payoffs consists of realizations $(x^{(i)}, y^{(i)})$ of the random variable (X, Y) where X is the initial state and Y is the final payoff. Informally, we have

$$\begin{aligned}
\text{price} &= E[Y|X] && \text{expectation of payoff given state} \\
&= \arg\min_f E_X\left\{\left[f(X) - Y\right]^2\right\} \\
&\approx \arg\min_w E_X\left\{\left[\hat{f}(X;w) - Y\right]^2\right\} && \text{for a universal approximator } \hat{f}, \text{asymptotically in capacity} \\
&\approx \arg\min_w \mathbf{MSE}(x^{(i)}, y^{(i)})(X)
\end{aligned}$$

Universal approximators, like neural networks, trained on datasets of sampled payoffs by minimization of the MSE, therefore, converge to the correct pricing function.

4.3 Training with derivatives

In the previous subsections, we found that we can train our model on sampled payoffs obtained in computationally tractable time and we have the theoretical guarantee that a universal approximator like ANN will converge to the correct pricing function. However, this still leaves us with many important questions, i.e., how many training data points will be enough? In practice, training on noisy payoffs is prone to overfitting and unrealistic training dataset sizes are necessary. In addition, risk sensitivities converge considerably slower than option prices. This can be resolved by training our models on datasets which include differentials of the labels with respect to inputs.

Since we will be focusing on the Heston model for our analysis, let us motivate this by considering its example. For the Heston model, $x^{(i)}$ would be the spot price sampled on some present or future date $T_1 \geq 0$, label $y^{(i)}$ would be the payoff of a call expiring on a later date T_2, sampled on the same path number i. We want to learn a function of S_{T_1} approximating the value of T_2 call measured at T_1. Differential labels are the pathwise derivatives of the payoff at T_2 with respect to the state at T_1 on path i, given by $\partial y^{(i)}/\partial x^{(i)}$. For the Heston model, and also Black-Scholes (since the payoff is the same under both models), we have:

$$\frac{\partial y^{(i)}}{\partial x^{(i)}} = \frac{\partial (S_{T_2}^{(i)} - K)^+}{\partial S_{T_1}^{(i)}} = \frac{\partial (S_{T_2}^{(i)} - K)^+}{\partial S_{T_2}^{(i)}} \frac{\partial S_{T_2}^{(i)}}{\partial S_{T_1}^{(i)}} = \mathbb{1}_{\{S_{T_2}^{(i)} > K\}} \frac{S_{T_2}^{(i)}}{S_{T_1}^{(i)}}$$

This resulting formula is computationally efficient. The derivative is computed together with the payoff along the path and there is no need to regenerate the path (as done in differentiation by finite difference). Systematic application of the chain rule efficiently computes these pathwise differentials. This is also

known as *adjoint differentiation* or AD.

Moreover, since expectation and differentiation commute, given appropriate smoothing of discontinuous cash-flows, risk sensitivities are expectations of pathwise differentials. This means that pathwise differentials are unbiased estimates of true *deltas*.

We use the differential labels obtained from our training set created using LSM. The differentials are computed quickly and will be accurate or our optimizer might be off-target. Differentiation algorithms like finite difference, for example, fail on both accounts.

Adjoint differentiation in its automated form (AAD), with great analytical accuracy, computes differentials for a computation cost proportional to one evaluation of the price. This rings a bell! By leveraging the chain rule to calculate derivatives, backpropagation does the exact same thing in context of neural networks. We will use this to learn value and risk as functions of state, helping us compute prices and deltas very quickly.

5 Monte Carlo Data

To create a training and testing set for comparing the accuracy of our results, we utilize Monte Carlo simulations to estimate option prices, along with the Malliavin calculus to calculate the corresponding deltas. The strategy we use directly utilizes Example 4.2.5 of Alos and Lorite (2021) [1].

We employ Monte Carlo methods to generate a large number of random asset paths for the underlying asset and its volatility based on the Heston model. We then use these simulated asset price paths to estimate the option prices through averaging. Next, we use the Malliavin calculus, a mathematical framework that essentially allows us to compute derivatives with respect to random variables, to determine the deltas associated with our Monte Carlo-computed option prices. The Malliavin calculus was used in favor of finite differences and Fourier methods due to its reduction in computation time, and its robustness in computing the deltas even if the log-asset price functions are unknown.

5.1 Monte Carlo Technique for Simulating Asset Paths

We now provide a more detailed explanation of the mechanics underlying our Monte Carlo simulations. However, to ensure emphasis remains on our empirical strategy, we simplify the content to include only the most essential elements.

Our first task is to generate the pair of random paths for asset prices and volatilities. Given an arbitrary set of discrete times $T = \{t_i\}_{i=1}^{N}$, we aim to generate the pair of random paths $(S_t, V_t) \forall t \in T$. However, instead of directly generating (S_t, V_t) for each t, we instead generate a random sample of $(S_{(t+\Delta)}, V_{(t+\Delta)})$ given (S_t, V_t) for an arbitrary increment Δ. By repeatedly applying this single-period

scheme with varying Δ for each time period in T, we can generate the complete path $(S_t, V_t)\forall t \in T$.

To ensure convergence to a complete path, we need to utilize an appropriate numerical method to incorporate randomness in approximating the behavior of asset prices and volatilities. As time increments become smaller, the numerical method should ideally converge to the desired paths as Δ becomes smaller. For this process, we introduce the Euler discretization scheme to approximate asset prices, and the Quadratic-Exponential (QE) algorithm introduced in Andersen (2008) [2].

The basic Euler discrete-time approximation for S_t is given by:

$$ln\hat{S}_{(t+\Delta)} = ln\hat{S}_t - \frac{1}{2}\hat{V}_t\Delta + \sqrt{\hat{V}_t}Z_x\sqrt{\Delta} \tag{3}$$

where Z_x pertains to a standardized Gaussian variable with correlation ρ. To be more specific,

$$Z_X = \rho\Phi^{-1}(U_1) + \sqrt{1-\rho^2}\Phi^{-1}(U_2) \tag{4}$$

where U_1 and U_2 are independent samples from a uniform distribution, and where Φ^{-1} is the inverse cdf of a Gaussian.

For the instantaneous variance, denoted by V_t, we utilize the variance-discretization scheme introduced in Andersen (2008). This method aims to model the distribution of $V_{(t+\Delta)}$, taking into account the qualitative properties of the true distribution V_t and its behavior for different values of V_t.

The Quadratic scheme is utilized for sufficiently large values of $V_{(t+\Delta)}$. It assumes that $V_{(t+\Delta)}$ can be approximated by a non-central chi-square distribution with a non-centrality parameter, for which a standard Gaussian random variable would be a good proxy. Mathematically, it can be expressed as

$$\hat{V}_{(t+\Delta)} = a(b + Z_V)^2$$

Here, Z_v represents a standard Gaussian random variable, and the constants a and b are determined through moment-matching. The values of a and b depend on the time-step Δ, the estimated instantaneous variance $\hat{V}_t$, and the parameters for the SDE governing V_t.

However, for smaller values of $V_{(t+\Delta)}$, the Quadratic scheme is not applicable. In such cases, we employ the Exponential approach that incorporates an approximate density function with a probability mass at the origin and an exponential tail. The density function is given by:

$$Pr(\hat{V}_{(t+\Delta)} \in [x, x+dx]) \approx \Big(p\delta(0) + \beta(1-p)e^{-\beta x}\Big)dx \tag{5}$$

In this equation, x is a value less than or equal to 0, p is a constant within the range [0, 1], and β is a non-negative constant. The density function is then used for sampling by integrating Equation 5 and calculating its inverse. Consequently, we obtain the sampling scheme as follows:

$$\hat{V}_{(t+\Delta)} = \Psi^{-1}(U_V; p, \beta) \tag{6}$$

where U_v represents a uniform random variable.

To start the algorithm, we first compute $\Psi = \text{Var}(V_{(t+\Delta)}|v_t = \hat{V}_t)) \times \text{E}(V_{(t+\Delta)}|V_t = \hat{V}_t))^{-2}$.The decision to use the Quadratic or Exponential scheme is then dependent on the value of Ψ, at some critical level $\Psi_c \in [1, 2]$. We use the Quadratic Scheme if $\Psi \leq \Psi_c$ and the Exponential Scheme if otherwise.

A detailed explanation of the Quadratic-Exponential algorithm can be found in Appendix A.1.

5.2 Greek estimation using the Malliavin calculus

We then proceed to apply the Malliavin calculus to facilitate the efficient computation of the Greeks. Specifically, we apply the integration-by-parts formula introduced by Fournié et al (1999) [7] referenced by Alos and Lorite (2021) [1]. With this method, we can avoid computing the derivatives altogether by representing them as the expectation of the product of the payoff multiplied by an arbitrary weight.

By Proposition 4.2.16 in Alos and Lorite (1999) [1], we can calculate the Delta as the following:

$$\Delta = \frac{\partial V}{\partial S_0} = \frac{e^{-rT}}{T\sqrt{1-\rho^2}S_0} E\Big(f(S_T)\int_0^T \frac{1}{\sigma_s} dB_s\Big) \tag{7}$$

5.2.1 Generating the dataset

We initialize our parameters as $k = 0.5$, $\theta = 0.05$, $\epsilon = 1.1$, $\rho = -0.9$, with an initial volatility of $v_0 = 0.05$, and a strike price of 120. For simplicity, we assume a time-to-maturity $T = 2$. We then generate a grid of uniform spot prices within the range 70 to 180 according to the specified number of samples in our dataset (i.e. 1,000 and 10,000). We then perform the Monte Carlo simulations to generate 100,000 asset paths, which we use to calculate the price of a European option and its corresponding delta for our varying spot prices.

6 Differential Deep Learning

We use differential machine learning applied to a twin neural network in order to estimate option prices and deltas under the Heston model. In this section, we first explain the concept of differential machine learning in more detail, and then describe the architecture of our neural network.

The goal of many machine learning models is to approximate some unknown function that maps input variables to some target variable of interest. Differential machine learning aims to approximate the shape of the underlying function, by training on differentials of the target variable with respect to the input variables. The method was designed to allow for fast approximation of complex pricing functions in the context of financial risk management, where practitioners must estimate price sensitivities of financial derivatives in close to real-time. It makes use of AAD, which, as mentioned in Section 4 is a chain-rule based method to efficiently compute differentials, and is closely related to the backpropagation algorithm used to train neural networks.

Differential machine learning aims to improve existing methods for estimating risk sensitivities under small sample sizes and high dimensional problems. Machine learning models such as neural networks often require large amounts of data to accurately estimate an underlying function, and in the context of financial risk management, the time cost of simulating sufficiently large training sets can be infeasible. Training on payoff values alone, a neural network can converge to a good approximation of the pricing function, as will training only on differentials of payoffs with respect to asset prices. But, for a given sample size, a model trained on differentials will achieve faster convergence. In practice, Huge and Savine (2020) find that the best results can be achieved by training a model on both payoffs and differentials. As such, we follow this approach laid out in Section 6.2.

Differential machine learning also provides a bias-free form of regularisation. By training a model on both option prices and differentials, the potential to overfit to noisy prices can be mitigated, as incorrect differentials are penalised in the loss function, where the size of the penalty can be controlled by a parameter, similar to ridge regression or LASSO.

Although we focus on differential machine learning in a deep learning setting, it can be applied to other popular machine learning methods including standard regression and principal component analysis. It can also be applied in fields beyond finance, when differentials of training inputs with respect to labels are available.

6.1 Twin Network Architecture

We employ a twin neural network, adapted from Huge and Savine (2020), to estimate the Heston pricing function. Twin neural networks, also known as siamese neural networks, were first introduced in 1993 for the purpose of signature verification [3]. As the name suggests, a twin neural network consists of two sub-networks that have the same structure. Moreover, the parameters in the corresponding layers of each sub-network are shared.

We input a vector of asset prices into the twin network and obtain two output vectors: the corresponding option prices and deltas. The first half of the twin network applies the standard feedforward passes of a typical multi-layer perceptron in order to return the predicted values (option prices). The

second half takes the output of the first half as its input, and applies backpropagation-style computations in a feedforward manner in order to predict differentials of the predicted values with respect to the inputs (the deltas).

Feedforward equations

Let $\mathbf{x} = [x_1\ x_2\ \dots\ x_n]^T$ be a vector containing n samples of the input variable and $\mathbf{y} = [y_1\ y_2\ \dots\ y_n]^T$ be the corresponding target values. The feedforward equations applied in the first half of the network are as follows.

$$\begin{aligned}
&\mathbf{z}_0 = \mathbf{x} \\
&\mathbf{z}_l = g(\mathbf{z}_{l-1})\mathbf{w}_l + \mathbf{b}_l \quad \forall\, l = 1, ..., L \\
&\mathbf{y} = \mathbf{z}_L,
\end{aligned}$$

where $L-1$ is the number of hidden layers in each half of the network, $\mathbf{z}_l$ denotes the pre-activation values in layer l, g_l denotes the activation function (applied element wise), and $\mathbf{w}_l$, $\mathbf{b}_l$ denote the weights and biases of layer l. The dimensions of $\mathbf{z}_l$, $\mathbf{w}_l$ and $\mathbf{b}_l$ depend on the layer. If we consider a constant number h of hidden units (neurons) in each hidden layer, then $\mathbf{z}_l \in \mathbb{R}^{n\times h}$, $\mathbf{w}_l \in \mathbb{R}^{1\times h}$, $\mathbf{b}_l \in \mathbb{R}^h$ for $l = 1$, $\mathbf{z}_l \in \mathbb{R}^{n\times h}$, $\mathbf{w}_l \in \mathbb{R}^{h\times h}$, $\mathbf{b}_l \in \mathbb{R}^h$ for $1 < l < L-1$, and $\mathbf{z}_l \in \mathbb{R}^{n\times h}$, $\mathbf{w}_l \in \mathbb{R}^{h\times 1}$, $\mathbf{b}_l \in \mathbb{R}$ for $l = L-1$.

(Feedforward) Backpropagation equations

The set of feedforward equations applied in the second half of the network resemble backpropagation. To obtain the pre-activation values in hidden layer l in the second half of the network, we apply the chain rule to compute the derivatives of the predicted values $\mathbf{y}$ with respect to the pre-activation values in hidden layer $L-l$ in the first half of the network. As such, the output of the second half of the network is a vector of differentials of the predicted values $\mathbf{y}$ with respect to the inputs $\mathbf{x}$. This process can be summarised by the following equations.

$$\begin{aligned}
&\frac{\partial \mathbf{y}}{\partial \mathbf{z}_L} = \frac{\partial \mathbf{y}}{\partial \mathbf{y}} = \mathbf{1} \\
&\frac{\partial \mathbf{y}}{\partial \mathbf{z}_{l-1}} = (\frac{\partial \mathbf{y}}{\partial \mathbf{z}_l}\mathbf{w}_l^T) \odot g_l'(\mathbf{z}_{l-1}) \quad \forall\, l = 1, ..., L \\
&\frac{\partial \mathbf{y}}{\partial \mathbf{z}_0} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}},
\end{aligned}$$

where $\odot$ is the Hadamard (element-wise) product and $g'()$ is the first derivative of the activation function applied in the first half of the network.

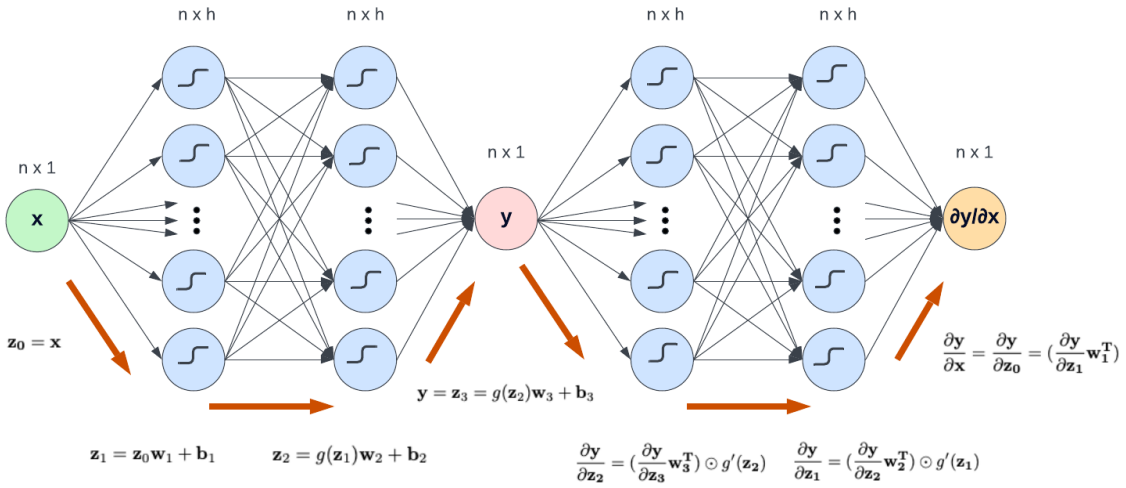

Figure 2: Simplifed diagram of the twin neural network

Figure 2 shows a simplified diagram of our twin neural network, shown with two hidden layers in each sub-network, and with equations defining each layer displayed below.

Our twin neural network contains four hidden layers in each sub-network, each with 20 hidden units. The choice of activation functions for the network is limited, as neurons in the second half of the network are activated using the first derivative of the initial activation functions. As a result, we require activation functions that are doubly differentiable, as training via backpropagation will involve computing second derivatives. This rules out the use of popular functions such as ReLU. In our implementation we use the softplus activation function, which is a smooth approximation of ReLU, with first derivative given by the logistic function. The softplus function is defined as:

$$softplus(x) = log(1 + exp(x))$$

The neural network is trained by minimising a composite cost function given by:

$$C = \alpha \cdot C_{\text{values}} + \beta \cdot C_{\text{differentials}},$$

where C_{values} is the MSE of the predicted of values and $C_{\text{differentials}}$ is the MSE of the predicted differentials. The parameters α and β control the relative weights of the loss function components, where $\alpha + \beta = 1$. We found that weighting both components equally was sufficient to produce good results. Note that the twin network, by design, can still predict deltas if differential labels are not included in the cost function (that is, if $\beta = 0$). However, as discussed in Huge and Savine (2020), and as we find in our results, training on differential labels can substantially improve model performance in realistic number of training samples. We talk about minimizing this cost function in Section 6.2

We use the *Adam* optimization algorithm to train the model and use an exponentially decaying learning rate. To improve training performance, we scale the raw asset and option prices by subtracting their respective means and dividing by their respective standard deviations. The raw differential labels are treated slightly differently; they are scaled by the ratio of the standard deviations of option prices and asset prices. Due to the scaling of asset and option prices, the predicted differentials (that is, the output of the first half of the network) will be scaled by the ratio of standard deviations and so we must adjust the differential labels accordingly.

6.2 Training with differential labels

The twin network architecture presented in Section 6.1 estimates the correct pricing function $f(x)$ by an approximate function $\hat{f}(x;\{w_l,b_l\})$, learning optimal weights w_l's and biases b_l's from our augmented training set with differential labels computed using AAD $(x^{(i)}, y^{(i)}, \partial y^{(i)}/\partial x^{(i)})$. Our training procedure then finds weights and biases minimizing some cost function. We will see what this cost function should be for our architecture.

Classic training

Let us denote $X = [x^{(i)}]$, i.e. a vector of $x^{(i)}$ where i runs from 1 to the size of our training set. Similarly, let $Y = [y^{(i)}]$ and $\bar{X} = [\partial y^{(i)}/\partial x^{(i)}]$. Our neural network in the first half computes $Z_l = [z_l^{(i)}]$ with an extra index for the layer and in the second half computes $\bar{Z}_l = [\bar{z}_l^{(i)}]$.

As seen in Section 4.2, approximation obtained by global minimization of MSE converges to the correct pricing function, hence:

$$C\Big(\{w_l,b_l\}\Big) = MSE = \frac{1}{m}(Z_L - Y)^T(Z_L - Y)$$

The second half of the twin network does not influence this cost function, so training is done by back-propagation through the standard feedforward network alone.

Differential training

Training is instead done with pathwise differentials $\partial y^{(i)}/\partial x^{(i)}$ instead of payoffs $x^{(i)}$ by minimizing the MSE (denoted here by $\overline{MSE}$) between predicted differentials and pathwise differentials.

$$C\Big(\{w_l,b_l\}\Big) = \overline{MSE} = \frac{1}{m}\mathrm{tr}\Big[(\bar{Z}_0 - \bar{X})^T(\bar{Z}_0 - \bar{X})\Big]$$

To evaluate $\bar{Z}_0$ the twin network must be evaluated in full, doubling our computation times as evident when we run the code. Backpropagation through the twin network minimizes $\overline{MSE}$ by calculating second-order differentials in its second half. This imposes severe constraint on what activation functions can be used in our twin network. One choice for the activation function is the softplus, as mentioned in the previous section.

Pathwise differentials only work for continuous payoffs, so calculating these for payoffs of, say, digital options, is not well defined. Given that the differentials are well-defined in our case, risk sensitivities are expectations of pathwise differentials,

$$\frac{\partial f(x)}{\partial x} = \frac{\partial E(Y|X=x)}{\partial x} = E\left[\frac{\partial Y}{\partial X} \middle| X = x \right]$$

It follows that pathwise differentials are unbiased estimates of risk sensitivities. Approximations trained by minimizing $\overline{MSE}$ converge to correct differentials (only constrained by the capacity of the approximator), and therefore capture the right pricing function plus an additive constant.

Classic + Differential training

We can combine classic and differential training by combining the values errors and derivatives errors in a composite cost function:

$$C = MSE + \lambda \overline{MSE}$$

Notice the similarity with classic regularisations like Ridge and LASSO, where overfitting is avoided by imposing a penalty for large weights. This comes at the cost of introduced bias and careful tuning of hyperparameter.

Differential training also stops fitting noisy labels by introducing a penalty for wrong differentials. However, since classic training and training on differentials both converge to the correct approximation of pricing function, this regularization comes without the bias-variance tradeoff.

7 Results

In this section we start with evaluating the out-of-sample performance of the twin network in predicting option prices and deltas under the Heston model. Here, the input to our twin network comes directly from the Monte Carlo simulations and we do not use the training dataset obtained via LSM. To highlight the benefits of differential machine learning, we compare the performance of the twin network trained on both differential and option price labels (henceforth referred to as differential training) against the performance of an equivalent twin network trained with only price labels (henceforth referred to as standard training). We assess performance based on the root mean squared error (RMSE) and the mean absolute error (MAE) of the predictions.

Figures 3 and 4 show the performance of the models in predicting option prices using training sets of size 800 and 9000 observations, respectively. We plot predicted and target values for options prices as a function of the underlying asset price. Both models perform well, with the standard training model achieving a RMSE of 0.8 under the larger sample size and the differential training model achieving a smaller RMSE of 0.1 (Table 1). Our primary concern, however, is how well the models can predict the

deltas. The differential model clearly outperforms when predicting deltas, as shown in Figures 5 and 6. After increasing the training sample size from 800 to 9000 observations, the differential model can predict the deltas with almost perfect accuracy. On the other hand, we observe minimal improvement in the performance of the standard training model.

Figure 3: Predicted versus target option prices for models with a training set size of 800 and a testing set size of 200.

Figure 4: Predicted versus target option prices for models with a training set size of 9000 and testing set size of 1000.

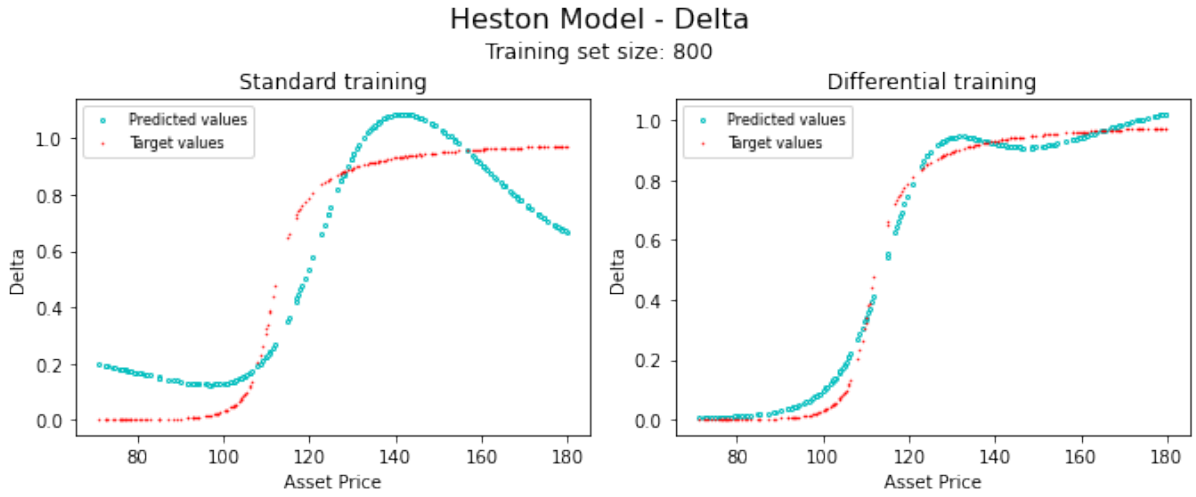

Figure 5: Predicted versus target deltas for models with a training set size of 800 and a testing set size of 200.

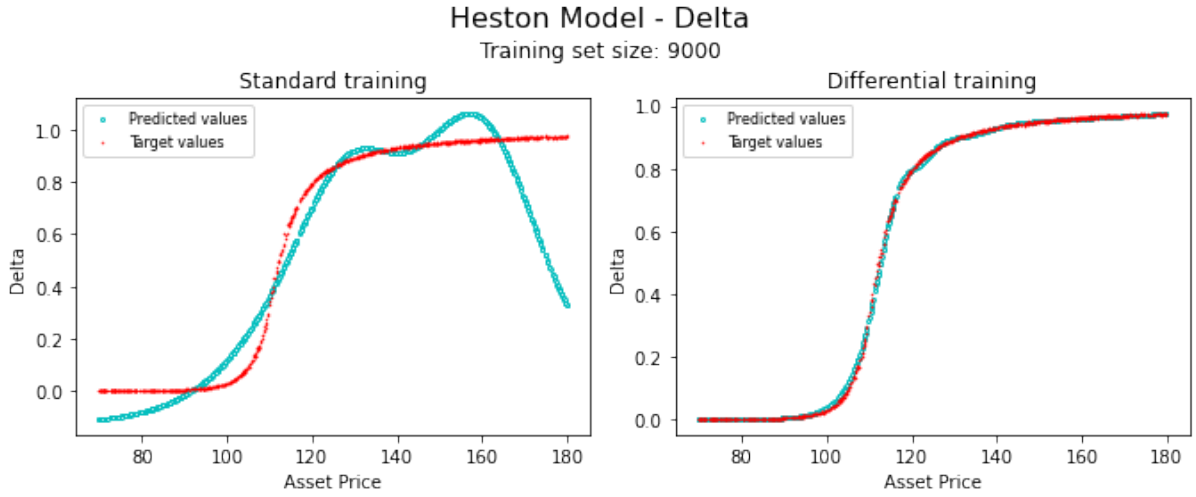

Figure 6: Predicted versus target deltas for models with a training set size of 9000 and testing set size of 1000.

Table 1: Results - Heston Model with Monte Carlo training set

Target Variable	**Sample size**	**RMSE**		**MAE**	
		Standard	**Differential**	**Standard**	**Differential**
Option Prices	800	1.319	0.375	1.148	0.309
Option Prices	9000	0.766	0.099	0.497	0.078
Deltas	800	0.154	0.043	0.136	0.035
Deltas	9000	0.149	0.008	0.094	0.005

Underlying mechanism

To understand this difference between standard training and differential training, take a closer look at the predicted versus target deltas with training set size of just 800 data points, as shown in Fig. 5. Since we explicitly provide this as the training data to our neural network architecture, we observe that with differential training the deltas converge at a faster rate to the target values than standard training.

Training with LSM dataset [1]

Next, we will try to learn from the training dataset generated using LSM. This requires discretizing the pair of SDEs equations in Equations 1 and 2, which has proven to be quite tricky. This is because traditional discretization techniques like Euler-Maruyama and Runge-Kutta are vulnerable to negative volatilities, even after setting a hard bound on these values. One possible reason for this is that too many training points are being pushed to particular values to satisfy these conditions. A potential resolution is to use numerical methods to get around this. Although we tried absorbing the negative variance, we find that this approach works for the model better with a small number of meaningful points.

Despite this, we still observe that differential machine learning architecture outperforms standard training as seen in Figures 7 and 8. This means that although we could not achieve our ultimate goal of learning from a readily computed noisy dataset, we can observe that training on differential labels and our twin network architecture are powerful tools that can make determining pricing functions computationally tractable.

Figure 7: Predicted vs Target Option Prices upon LSM training

[1] Interestingly, our parameter initialization did not satisfy Feller's condition: $2\kappa\theta > \epsilon^2$. This is worth taking a closer look at!

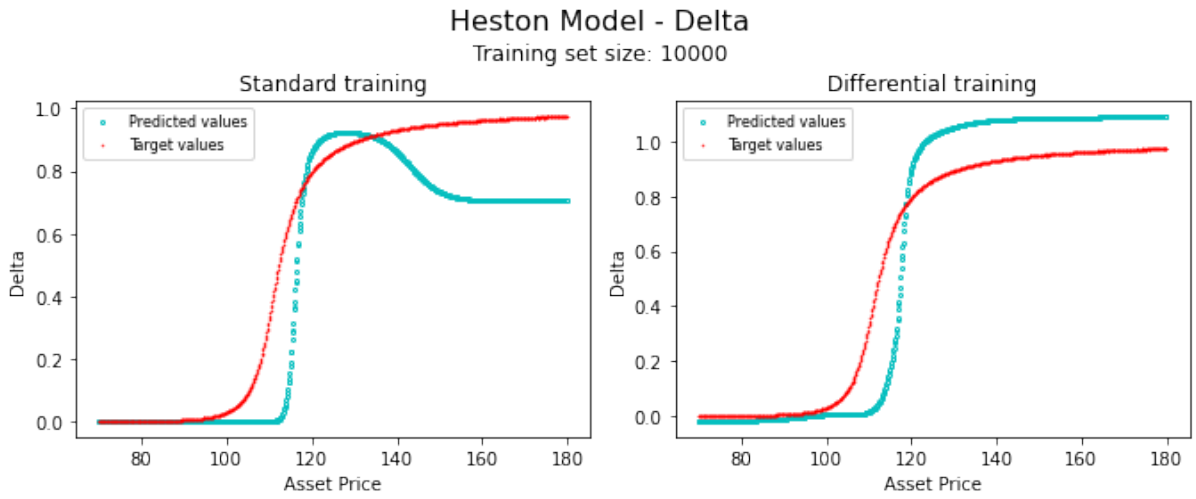

Figure 8: Predicted vs Target Deltas upon LSM training

Table 2: Results - Heston Model trained with LSM dataset

Target Variable	**RMSE**		**MAE**	
	Standard	**Differential**	**Standard**	**Differential**
Option Prices	5.412	2.068	3.918	1.507
Deltas	0.186	0.159	0.131	0.121

8 Discussion

Overall, we can see that the differential twin network noticeably outperforms standard training in all instances for both the Monte Carlo training and test split, and the LSM training set. We can therefore confirm that our model outperforms conventional deep learning methods for fundamentally different data generation methods, even for noisier data such as LSM. It manages to do this while also computing both option prices and deltas in a single pass.

Nevertheless, it is important to consider certain factors in light of our findings, which could potentially contribute to extensions of our research. First of all, it is crucial to recognize that the Monte Carlo simulations should not be regarded as "ground truth" benchmark for testing purposes. However, they are very good approximations if generated with a large number of simulation paths and small time steps. We opted to use these simulations because they provided the only available metric for comparing our results, and are the industry standard. Ultimately, though, the accuracy of the option prices generated by these simulations is dependent on their initial setup and thorough training. Despite this we can observe with our results that the Monte Carlo simulations perform really well.

The biggest downside with our simulations, however, is the high computation time. This is due to the fact that our simulations generate 100,000 Monte Carlo paths for an interval of 1,000 and 10,000 spot price initializations. To remedy this issue, we explore the method of generating noisy sample data with LSM and using it as a training set to compare against our Monte Carlo benchmark. Our results

suggest that implementing a noisy data generation method such as the LSM using a simple discretization method with hard bounds on non-negativity for volatility values makes our discretization susceptible to errors. Despite this, our differential twin network still outperforms standard training considerably. A better discretization scheme with appropriate handling of negative volaitlity values should help improve the results.

Beyond these discussion points, this paper naturally leads to other possibilities for further exploration and research. First of all, differential machine learning can be applied to other models such as regression and principal components analysis as a dimensionality reduction technique for higher-dimension models. Huge and Savine (2020) [9] briefly discuss the two methods as an effective method of pricing and risk function approximation, claiming it can improve over classic regression and well-known forms of regularization such as ridge regression. Differential regression can be applied to the estimation of simpler, lower-dimension models such as the Black-Scholes model, or possibly even the Heston model, and may produce results comparable to our twin network.

The findings of our paper suggest potential for differential machine learning to be applied to more advanced and complex models in higher dimensions. Alternative stochastic volatility models such as the SABR, or higher-dimension ones that aim to capture the irregularities of volatility such as rough volatility models, are good starting points for further research. These models provide different perspectives on modelling volatility and don't have simple closed-form solutions. Finding a discretization scheme that will make it possible to learn from these models in real time, along with using "noisy" datasets with theoretical convergence guarentees can be a crucial step in fully realizing the potential of using neural networks to get option prices and deltas.

At the same time, another avenue to explore would be the computation of the other Greeks: *Vega*, *Rho*, and *Theta*. Second order Greeks such as the *Gamma* are also possible to compute, but it may necessitate more sophisticated network architecture and a special activation function. The ideal scenario would be to have our differential neural network train and predict the prices and all the Greeks simultaneously, and to have them altogether as an output. This would, however, necessitate incorporating variations in volatility, the risk-free rate of return, and time to maturity and getting those respective derivatives. The architecture would need to adapt these three additional variations as inputs, and pathwise differentiation would need to be trained for the rest of the Greeks. This sets restrictive conditions on these activation functions, which even for *delta* needs to be in class C^2.

Next, there still remains scope to improve predictive performance through further hyper-parameter tuning. A twin neural network applied to differential machine learning is a novel approach. As a result, user-friendly pipelines provided by libraries such as Tensorflow to optimise hyper-parameters can not easily be integrated with our model. Incorporating methods to control potential overfitting such as *dropout* and *early stopping* may also improve performance, although as we mentioned, training on

differentials can provide a form of regularisation on its own.

9 Conclusion

In conclusion, differential machine learning applied to a twin neural network architecture has successfully computed good approximations of option prices and their respective *deltas*. We compare our findings with a standard twin network that trains only on option price labels, and we evaluate out-of-sample performance using RMSE and MAE. In doing so, we find that our differential twin network outperforms standard training when we use Monte Carlo simulations as a training and test set, along with our noisier training set generated via LSM. Our results suggest that our network performs even better with more input data. At the same time, it allows us to predict both option prices and their deltas faster than traditional methods and in a single pass.

These findings are worthy of note as the prediction of *delta* is not straightforward. By approximating the shape of the function instead of their individual points, differential training proves to be effective in calculating both values and their sensitivities. This unique technique is particularly pertinent for higher-dimension models without closed-form solutions such as the Heston model, where sometimes our only available benchmark are computationally-intensive Monte Carlo simulations. Our results promisingly demonstrate comparable results when Monte Carlo simulations are used for comparison. However, they may not even be necessary for a training set. By generating noisy approximations of the payoff with the computation cost of one Monte Carlo path (i.e. LSM), we are still able to get better results with differential training (compared to the standard one). LSM can be further improved with a better discretization scheme that can handle negative volatility values effectively.

Finally, it is worth noting that our paper presents ample opportunities for further extension and exploration. Naturally, differential machine learning can also be applied to more complex financial models, and other first and second-order Greeks beyond *delta* can also be estimated. Furthermore, predictive performance of our network can still be improved with further hyper-parameter tuning.

References

[1] Elisa Alos and David Garcia Lorite. *Malliavin Calculus in Finance: Theory and Practice*. 1st ed. Chapman and Hall/CRC financial mathematics series. Chapman amp; Hall, 2021.

[2] Leif Andersen. "Simple and efficient simulation of the Heston Stochastic Volatility Model". In: *The Journal of Computational Finance* 11.3 (2008), pp. 1–42. DOI: 10.21314/jcf.2008.189.

[3] Jane Bromley et al. "Signature verification using a" siamese" time delay neural network". In: *Advances in neural information processing systems* 6 (1993).

[4] John C. Butcher. "Runge–Kutta Methods for Ordinary Differential Equations". In: 2015.

[5] Macarena Cruz, Ricardo Sanchez, and Sergio-Yersi Villegas. "Applying Artificial Neural Networking to Option Pricing". MA thesis. Barcelona School of Economics, 2022.

[6] Christa Cuchiero, Wahid Khosrawi, and Josef Teichmann. "A generative adversarial network approach to calibration of local stochastic volatility models". In: *Risks* 8.4 (2020), p. 101. DOI: 10.3390/risks8040101.

[7] Eric Fournié et al. "Applications of malliavin calculus to Monte Carlo Methods in Finance". In: *Finance and Stochastics* 3.4 (1999), pp. 391–412. DOI: 10.1007/s007800050068.

[8] Blanka Horvath, Aitor Muguruza, and Mehdi Tomas. "Deep learning volatility: A deep neural network perspective on pricing and calibration in (rough) volatility models". In: *Quantitative Finance* 21.1 (2020), pp. 11–27. DOI: 10.1080/14697688.2020.1817974.

[9] Brian Huge and Antoine Savine. "Differential machine learning". In: *arXiv preprint arXiv:2005.02347* (2020). URL: https://arxiv.org/abs/2005.02347.

[10] Shuaiqiang Liu, Cornelis Oosterlee, and Sander Bohte. "Pricing options and computing implied volatilities using Neural Networks". In: *Risks* 7.1 (2019), p. 16. DOI: 10.3390/risks7010016.

[11] Shuaiqiang Liu et al. "A neural network-based framework for financial model calibration". In: *Journal of Mathematics in Industry* 9.1 (2019). DOI: 10.1186/s13362-019-0066-7.

[12] Mary Malliaris and Linda Salchenberger. "A neural network model for estimating option prices". In: *Applied Intelligence* 3.3 (1993), pp. 193–206. DOI: 10.1007/bf00871937.

[13] Mathieu Rosenbaum and Jianfei Zhang. "Deep calibration of the quadratic rough Heston model". In: *arXiv:2107.01611* (July 2021).

[14] Johannes Ruf and Weiguan Wang. "Neural networks for option pricing and Hedging: A Literature Review". In: *The Journal of Computational Finance* (2020). DOI: 10.21314/jcf.2020.390.

[15] Jingtao Yao, Yili Li, and Chew Lim Tan. "Option price forecasting using Neural Networks". In: *Omega* 28.4 (2000), pp. 455–466. DOI: 10.1016/s0305-0483(99)00066-3.

A Appendix

A.1 Quadratic-Exponential Algorithm (Andersen, 2008)

The QE simulation step for computing $\hat{V}(t)$ to $\hat{V}(t+\Delta)$ comes from (Andersen 2008) [2].

Select an arbitrary threshold $\Psi_c \in [1, 2]$.

1. Given $\hat{V}(t)$, compute:

$$m = \theta + (\hat{V}(t) - \theta)e^{-\kappa\Delta}$$
$$s^2 = \frac{\hat{V}(t)}{\epsilon}^2 e^{-\kappa\Delta}\kappa\Big(1 - e^{-\kappa\Delta}\Big) + \frac{\theta\epsilon^2}{2\kappa}\Big(1 - e^{-\kappa\Delta}\Big)^2$$

2. Compute $\Psi = \frac{s^2}{m^2}$.

3. If $\Psi \leq \Psi_c$,

   (a) Compute:

$$a = \frac{m}{1 + b^2}$$
$$b^2 = 2\Psi^{-1} - 1 + \sqrt{2\Psi^{-1}}\sqrt{2\Psi^{-1} - 1} \geq 0$$

   (b) Compute $Z_V = \Phi^{-1}(U_V)$ using algorithm.

   (c) Set $\hat{V}(t+\Delta) = a(b + Z_V)^2$.

4. Else:

   (a) Compute:

$$\beta = \frac{1-p}{m} = \frac{2}{m(\Psi+1)} > 0$$
$$p = \frac{\Psi - 1}{\Psi + 1} \in [0, 1)$$

   (b) Set $\hat{V}(t+\Delta) = \Psi^{-1}(U_V; p; \beta)$, where:

$$\Psi^{-1}(u) = \Psi^{-1}(u; p, \beta) = \begin{cases} 0 & 0 \leq u \leq p \\ \beta^{-1} ln(\frac{1-p}{1-y}) & p < u \leq 1 \end{cases} \tag{8}$$

A.2 Runge-Kutta Discretization Scheme

For our implementation of Least Squares Monte-Carlo, we used Runge Kutta discretization to discretize the stochastic differntial equation of the price process and Euler discretization for the volatility process. This method works by making an estimate of the change in the dependent variable over a small time step based on the values of the derivative at the start, middle, and end of the time step. These estimates are then combined to give an overall best estimate of the change over the time step.

We then make necessary adjustments to use this with a stochastic process. Since this method can't be used with non-linear SDEs like the SDE for the volatility process, we use the conventional Euler scheme for this discretization.

In our case, this calculations can be outlined as follows:

$$
\begin{aligned}
k_1 &= \Delta T \times \Big(\sigma\sqrt{V}Z + \kappa(\theta - V)\Big) \\
k_2 &= \Delta T \times \Big(\sigma\sqrt{V + 0.5k_1}Z + \kappa(\theta - V - 0.5k_1)\Big) \\
k_3 &= \Delta T \times \Big(\sigma\sqrt{V + 0.5k_2}Z + \kappa(\theta - V - 0.5k_2)\Big) \\
k_4 &= \Delta T \times \Big(\sigma\sqrt{V + k_3}Z + \kappa(\theta - V - k_3)\Big) \\
\text{Update: } S(T + \Delta T) &= S(T)\exp\left(\frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\right)
\end{aligned}
$$

and discretization of volatility process follows simple Euler discretization,

$$V_1 = V + \kappa(\theta - V)\Delta T + \epsilon\sqrt{V}\sqrt{\Delta T}W$$

where Z and W are correlated with $\text{corr}(Z, W) = \rho$.