Barcelona School of Economics

MSc Data Science

Master Project

Applying Artificial Neural Networking to Option Pricing

Authors:
Macarena Cruz
Ricardo Sánchez
Sergio-Yersi Villegas

Tutor:
Prof. Dr. Eulàlia Nualart

May, 2022

ABSTRACT

Option pricing models' best feature is the ability to complement the results with an intelligible interpretability, which is of great importance for the understanding of the market. However, limitations come from the non-closed formulae that many options have and the required computational time of these models, which can be unfeasible when dealing with this problem in a daily basis. Therefore, we propose a deep learning based approach, by means of an Artificial Neural Network (ANN), to compute the price of a European Call option under the Black-Scholes Model (BSM). Then, with the trained ANN, we compute the corresponding option Greeks, which have huge relevance for hedging purposes. The results show that the network excels at predicting the option prices, with an $MSE \sim 10^{-7}$ and $R^2 \sim 1$, while also significantly decreases the computational time of the predictions. The price to pay, however, when using this approach is precisely the best feature of the original models: interpretability. Finally, the prediction of the Greeks using the derivatives of the network was unfeasible: deriving target predictions is not equivalent to the true target's derivatives.

I. Introduction

In 1973 there was a breakthrough in the Quantitative Finance theory, especially in the pricing and modeling field, when Fischer Black and Myron Scholes presented their famous formula for pricing derivatives. The model starts by assuming a specific form of movement in the underlying asset, i.e., a geometric Brownian motion. Based on the conditional payment at maturity of the option, the model derives the partial differential equation that must be satisfied to meet specific economic constraints, i.e., arbitrage is unacceptable. Hence, in the absence of arbitrage, the combination of parameters must yield the risk-free interest rate, which is also assumed to be constant. In this framework, a closed-form solution of the option price is possible. Despite its dubious assumptions, the Black-Scholes Model (BSM) prices are pretty close to the prices observed in the market, so understandably, the BSM is one of the most cited and famous models in quantitative finance. However, a great deal of effort has been made for relaxing the assumptions of the original model; the celebrated BSM pricing formula has now been generalized, extended, and applied to several securities. In each case, either the original BSM or its extensions, the derivation of the pricing formula depends intimately on the parametric form of the underlying asset, where in some cases there is no close formula to obtain the price so it is used numerical methods. Miss-specification of the stochastic process of the underlying asset's dynamics will link to significant pricing and hedging errors. Hence, it is correct to say that the success or failure of the traditional approach, which we will refer to as the parametric method approach, of pricing derivatives, highly depends on the model's ability to capture the underlying asset's stochastic dynamics.

Nevertheless, market data is abundant, and it is possible to "learn" the function that generates the option prices in the market. Hutchinson et al. [1] were the first to demonstrate that a neural network is an efficient algorithm to "learn" the market's option pricing function. To illustrate the functionality of non-parametric methods, they estimated four popular models: Ordinary Least Squares (OLS), Radial Basis Function Networks (RBFN), Multilayer Perceptron Networks (MLP), and projection suit (PPR). The models were trained using two years' worth of option prices simulated using a Monte Carlo method and then tested on out-of-sample options, specifically S&P future options from 1987 to 1991. Hutchinson et al. [1] paper proved that learning networks can recover the Black-Scholes formula and that the resulting network could be used to obtain prices and successfully delta-hedge options out-of-sample. Their findings show that non-parametric models, especially the MLP and PPR, have very high R^2 values in the out-of-sample test set with a mean R^2 values of 95.53% and 96.56%, respectively. However, promising these first results were, the authors were warily optimistic about using these non-parametric methods. Firstly, and more pressing for our research, for the lack of proper statistical inference in the specification of the network architecture. Secondly, we notice that due to computational capacity and, as mentioned before, the lack of statistical inference methods, they only used one layer in the specification of the MLP, which abstained them from exploring deep learning algorithms.

Hutchinson et al. [1] left promising results and a wide range of questions to answer; since then, pricing European Options with an Artificial Neural Network (ANN) has been a recurring research topic. For instance, in 2000, Yao et al. [2] used back-propagation ANNs to forecast option prices of Nikkei 225 index futures. They pointed out that the BSM is still adequate for pricing at-the-money options and suggested partitioning the data according to moneyness when applying ANNs. Interestingly, they suggested that ANN option pricing models outperform the traditional BSM in volatile markets. Later in 2004, Bennell and Sutcliffe [3] compared the performance between a BSM and an ANN for pricing European options. There is a constant in these research papers; most of them differentiate themselves by investigating the effect that the structure of the inputs could have on the performance of ANNs, but few focus on

the architecture of these algorithms. The aforementioned can be explained by the current stage of knowledge that we have on these algorithms.

ANNs are one of the most remarkable programming paradigms ever invented. In the conventional basis of how a program works, we continuously guide our machine throughout the process so that it can fully understand what we need from it. In that way, we decompose each problem into many specific and more minor problems intelligible for our computer, so they are all easy to perform. By contrast, ANNs do not require human guidance in order to solve a problem; alternatively, a network uses its divide-and-conquer strategy to learn a function [4]. As we will see, each neuron in the network is a simple model that is tasked with learning a simple function. Then, the complete and more complex model implemented by the ANN is recursively built by combining these smaller ones, therefore accomplishing the original goal of this approach: the computer ends up figuring the solution to the problem on its own. However, there is still a significant knowledge gap in how these algorithms work and what makes them more efficient at specific tasks than others. This gap results in the lack of guarantee that the model will perform well in the specific problem. A significant amount of effort has been directed at investigating the modeling procedure of ANNs to introduce a systematic way that would lead to consistent performance of ANNs; the current standard practice calls for a repetitive trial-and-error process, which is time-consuming and delivers indecisive results. The research has focused on virtually every aspect of the ANN architecture development, such as training data, different types of activation functions, initialization of weights, training algorithms, and metrics [5]. Throughout this paper we will explore various combinations of these parameters described above in the construction of our ANN.

Although extensive research focuses on using ANNs to predict the prices of European and Exotic options, the methodology to build and utilize such nets is still missing a consensus. Hence, in this paper we revisit the methodology used by Liu et al. [6] to price European Options and extend their methodology by redefining the hyper-parameter optimization algorithm, as will be exposed in further sections. Hence, in this project we aim to build an ANN from scratch to value financial options with the four following purposes: firstly, obtaining the possible exactness of our predictions, so there is no difference with the use of the parametric option pricing model; secondly, accelerating these corresponding models, since the real-world problems of this field make the use of these models unfeasible, due to the requirement of massive computations in a daily basis; thirdly, estimating the Greeks associated to our financial option using a *Gradient Tape* technique with our trained ANN, in order to see if it is possible to compute them using our approach accurately, and not one where the ANN directly outputs these Greeks; and lastly, we aim to build a hyper-parameter optimization algorithm that could potentially be extended to any pricing model, with close and non-close formula. Additionally, we evaluate its computational cost and briefly comment on its real-life scalability. To do so, starting in section II we will introduce the option price model we are going to be working with, which corresponds to the BSM and the corresponding Greeks associated with it. Then, in section III we will provide a theoretical explanation of what neural networks are and how they work. Next, the primary block of this project is section IV, where we will detail the steps of our work: from the simulation of our financial data to the full implementation of our ANN. Then, our model will be evaluated based on the two chosen performance metrics, Mean Squared Error (MSE) and the R^2 coefficient, and will yield the predicted option prices and their corresponding Greeks, all of which will be presented in section V. Finally, after analyzing the previous results, we will provide our conclusions in section VI.

II. Option pricing

A. Black-Scholes-Merton Model

In 1973 Fischer Black and Myron Scholes proposed the world-famous "Black-Scholes formula", which estimates the theoretical price of European-style options [7]. The original formulation of the BSM was described as a second-order partial differential equation. Later, Robert Merton published a mathematical understanding of their model using stochastic calculus that helped formulate what became known as the Black-Scholes-Merton formula. The equation of the model is:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0, \quad t \in [0, T), \quad S \geq 0 \tag{1}$$

where V denotes the payoff function, S denotes the underlying price of the asset, r denotes the risk free interest rate, σ denotes the *constant* volatility of return, t denotes current time and T denotes the time of maturity.

When deriving the formula, Black and Scholes assumed the following ideal conditions in the market for the stock and the option:

1. The short-term interest rate is known and is constant through time.
2. The underlying assets follow a random walk; in other words, the underlying asset prices are log-normally distributed and follow the geometric Brownian motion. This point will be relevant in the simulation of the data and will be further developed in section IV A.
3. There are no transaction costs in buying or selling the stock nor the option.
4. It can exercise option rights at expiration date. Hence, it can only be used to price European Options, and the underlying assets do not generate dividends.

The Black-Scholes equation (see expression 1) knows nothing about what kind of option we are valuing, whether it is a call or a put, nor what is the strike and the maturity. We must specify the option value V as a function of the underlying asset at the expiry date T. Therefore, we describe V in terms of the strike and maturity, firstly in the case of a call option, as:

$$V(S,T) = \max(S - E, 0) \tag{2}$$

and secondly, in the case of a put:

$$V(S,T) = \max(E - S, 0) \tag{3}$$

A closed-form solution for a European call, using the call payoff formula in expression 2 as a final condition, is given by:

$$C(S_t,t) = S_t N(d_1) - Ke^{-r(T-t)}N(d_2) \tag{4}$$

where S_t is the stock price at time t, $C(S_0,t)$ is the price of the call option as a formulation of the stock price at time t, K is the strike price, $(T-t)$ is the time to maturity, σ represents the underlying volatility and r is the risk-free interest rate. Moreover, the terms $N(d_1)$ and $N(d_2)$ are cumulative distribution functions for a standard normal distribution with the following formulation:

$$d_1 = \frac{\ln\left(\frac{S_t}{K}\right) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}} \tag{5}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{6}$$

B. The Greeks

The Greeks refer to a set of calculations you can use to measure different factors that might affect the price of an options contract. With that information, you can make more informed decisions about which options to trade, and when to trade them.

The Greeks are defined as derivatives of the option value with respect to various variables and parameters. Popular Greeks are the following: Delta, Gamma, Theta, Vega and Rho[8].

1. The Delta

The Delta of an option, usually denoted by Δ, is the sensitivity of the option, or portfolio, to the underlying. It is the rate of change of the value with respect to the asset, such that:

$$\Delta = \frac{\partial V}{\partial S} \tag{7}$$

We can estimate the Delta of a European Call as follows:

$$\Delta = \frac{\partial C}{\partial S_t} = \frac{\partial}{\partial S_t}\left(S_t N(d_1) - Ke^{r(T-t)}N(d_2)\right) = N(d_1) + S_t\frac{\partial N(d_1)}{\partial d_1}\frac{\partial d_1}{\partial S_t} - Ke^{r(T-t)}\frac{\partial N(d_2)}{\partial d_2}\frac{\partial d_2}{\partial S_t} \tag{8}$$

where we know that, following equations (5) and (6):

$$\frac{\partial d_1}{\partial S_t} = \frac{\partial d_2}{\partial S_t} = \frac{1}{S_t\sigma\sqrt{T-t}} \tag{9}$$

and, deriving the expression of a Gaussian distribution:

$$N'(x) = \frac{\partial N(x)}{\partial x} = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}} \tag{10}$$

we can re-write (8) as:

$$\Delta = \frac{\partial C}{\partial S_t} = N(d_1) + \frac{1}{S_t\sigma\sqrt{T-t}}\left(S_t N'(d_1) - Ke^{-r(T-t)}N'(d_2)\right) \tag{11}$$

Therefore, in order to be able to group expressions accordingly with expression (10), we must perform the following algebraic steps, so that we get a dependency on d_1^2 and d_2^2. Firstly, reorganizing expression (5) as:

$$\ln\left(\frac{S_t}{K}\right) + r(T-t) = d_1\sigma\sqrt{T-t} - \frac{1}{2}\sigma^2(T-t)$$

we can get a perfect square in the right-hand-side of the equation, such that:

$$\ln\left(\frac{S_t}{K}\right) + r(T-t) = -\frac{1}{2}\left(\sigma\sqrt{T-t} - d_1\right)^2 + \frac{d_1^2}{2}$$

From here, we can easily identify that $\sigma\sqrt{T-t} - d_1 = -d_2$, which leads to:

$$\ln\left(\frac{S_t}{K}\right) = -\frac{d_2^2}{2} + \frac{d_1^2}{2} - r(T-t) \Rightarrow \frac{S_t}{K} = e^{-\frac{d_2^2}{2}+\frac{d_1^2}{2}-r(T-t)} \Rightarrow S_t e^{-\frac{d_1^2}{2}} = Ke^{-\frac{d_2^2}{2}}e^{-r(T-t)}$$

$$S_t N'(d_1) = KN'(d_2)e^{-r(T-t)} \tag{12}$$

Therefore, plugging the result into expression (11), we get that:

$$\Delta = N(d_1) + \frac{1}{S_t\sigma\sqrt{T-t}}\left(Ke^{-r(T-t)}N'(d_2) - Ke^{-r(T-t)}N'(d_2)\right) = N(d_1) \tag{13}$$

2. The Gamma

The Gamma (Γ) of an option, or a portfolio of options, is the second derivative with respect to the underlying. Equivalently as we did for the Δ, we can express the Gamma of a European Call as follows:

$$\Gamma = \frac{\partial^2 V}{\partial S_t^2} = \frac{\partial \Delta}{\partial S_t} = N'(d_1)\frac{\partial d_1}{\partial S_t} = \frac{1}{\sqrt{2\pi}}e^{-\frac{d_1^2}{2}}\frac{1}{S_t\sigma\sqrt{T-t}} \tag{14}$$

3. The Theta

The Theta (Θ) is used to measure the rate of change of the option price as the option nears expiration, keeping all other factors constant. The following lines show the derivation of the Θ for a European Call, where $\tau = T - t$. Given that the Θ at time T is equal to 0, we can write the derivative in the following manner:

$$\Theta = \frac{\partial V}{\partial t} = -\frac{\partial V}{\partial \tau} = -\left(S_t N'(d_1)\frac{\partial d_1}{\partial \tau} - Ke^{-r\tau}(-r)N(d_2) + Ke^{-r\tau}N'(d_2)\frac{\partial d_2}{\partial \tau} \right)$$

From expression (6) we can re-write the aforementioned expression as:

$$\begin{aligned}\Theta = -\frac{\partial V}{\partial \tau} &= -\left(S_t N'(d_1)\frac{\partial d_1}{\partial \tau} + Kre^{-r\tau}N(d_2) - \frac{\partial (d_1 - \sigma\sqrt{\tau})}{\partial \tau}Ke^{-r\tau}N'(d_2) \right) = \\ &= -\left(rKe^{-r\tau}N(d_2) + \frac{\partial d_1}{\partial \tau}\left[S_t N'(d_1) - Ke^{-r\tau}N'(d_2) \right] + Ke^{-r\tau}N'(d_2)\frac{\sigma}{2\sqrt{\tau}} \right)\end{aligned}$$

Here, we can use the result obtained in (12) for the two following purposes: cancelling the term that multiplies with $\frac{\partial d_1}{\partial \tau}$ and re-writing the last term. Therefore, the final expression for the Θ is:

$$\begin{aligned}\Theta = -\frac{\partial V}{\partial \tau} &= -\left(rKe^{-r\tau}N(d_2) + S_t N'(d_1)\frac{\sigma}{2\sqrt{\tau}} \right) = \\ &= -\frac{S_t \sigma}{2\sqrt{T-t}}\frac{1}{\sqrt{2\pi}}e^{-\frac{d_1^2}{2}} - rKe^{-r(T-t)}N(d_2)\end{aligned}$$

4. The Vega

The Vega (υ) is the sensitivity of the option value to the volatility. However, this Greek is completely different from the others, since it is the derivative with respect to a parameter and not a variable. Hence, its expression is:

$$\upsilon = \frac{\partial V}{\partial \sigma} = S_t N'(d_1)\frac{\partial d_1}{\partial \sigma} - Ke^{-r(T-t)}N'(d_2)\frac{\partial d_2}{\partial \sigma} \tag{15}$$

Now, equivalently as we have done before, we use (6) in order to re-write (15):

$$\upsilon = \frac{\partial V}{\partial \sigma} = S_t N'(d_1)\frac{\partial d_1}{\partial \sigma} - Ke^{-r(T-t)}N'(d_2)\frac{\partial}{\partial \sigma}\left(d_1 - \sigma\sqrt{T-t} \right) = \tag{16}$$

$$= \frac{\partial d_1}{\partial \sigma}\left(S_t N'(d_1) - Ke^{-r(T-t)}N'(d_2) \right) + Ke^{-r(T-t)}N'(d_2)\sqrt{T-t} \tag{17}$$

Therefore, using (12) equivalently as we did for the Θ, we can express υ as:

$$\upsilon = \frac{\partial V}{\partial \sigma} = Ke^{-r(T-t)}N'(d_2)\sqrt{T-t} = S_t N'(d_1)\sqrt{T-t} = S_t\sqrt{T-t}\frac{1}{\sqrt{2\pi}}e^{-\frac{d_1^2}{2}} \tag{18}$$

5. The Rho

The Rho (ρ) is the sensitivity of the option value to the interest used in the Black-Scholes formula.

$$\begin{aligned}\rho = \frac{\partial V}{\partial r} &= S_t N'(d_1)\frac{\partial d_1}{\partial r} - K\left(e^{-r(T-t)}\left(-(T-t)\right)N(d_2) + e^{-r(T-t)}N'(d_2)\frac{\partial d_2}{\partial r} \right) = \\ &= S_t N'(d_1)\frac{\partial d_1}{\partial r} - K\left(e^{-r(T-t)}\left[-(T-t)\right]N(d_2) + e^{-r(T-t)}N'(d_2)\frac{\partial}{\partial r}\left[d_1 - \sigma\sqrt{T-t} \right] \right) = \\ &= \frac{\partial d_1}{\partial r}\left(S_t N'(d_1) - Ke^{-r(T-t)}N'(d_2) \right) + (T-t)Ke^{-r(T-t)}N(d_2)\end{aligned}$$

where we have again used (6) to re-write the original expression, so we can also use (12) to have the following final expression for ρ:

$$\rho = (T-t)Ke^{-r(T-t)}N(d_2) \tag{19}$$

III. Artificial Neural Networks

A. Definition

Deep learning is a sub-division of machine learning that is inspired by the way humans gain knowledge and can be briefly simplified as a way to automate predictive analysis. In its most basic definition deep learning algorithms are based on constructing a network of connected computational elements. In traditional machine learning, the learning process is supervised and the programmer in charge has to be extremely specific when guiding the computer throughout the problem in question. This challenging process, known as feature extraction, and its success rate entirely depends upon the programmer's ability to accurately define the learning procedure, in a way that the computer understands what it has to do. Alternatively, the advantage of deep learning is the fact that the program independently builds the whole learning by itself, without the need of human supervision. Therefore, while the complexity of traditional machine learning algorithms remains constant during the whole model, deep learning algorithms are stacked in a hierarchy of increasing complexity and abstraction. This sub-division of machine learning can be summarized in a neural network model.

An ANN is a computational model that resembles the structure of the human brain, which is composed of a massive number of nerve cells, called neurons. These basic information units work in a very simple manner: if the incoming stimuli are strong enough, the neuron transmits an electrical pulse, called an action potential, to the other neurons that are connected to it. Therefore, a neuron acts as an all-or-none switch that takes in a set of inputs and, depending on the information it gets, returns an specific output [4]. Therefore, an ANN consists of a network of simple information processing units, called *neurons*, which are organized into different types of *layers*:

- The input layer, which receives the parameters in their original form and starts the model.
- The hidden layers, which allow the model to increase its complexity by breaking down the network into specific transformations of the data, determined by the number of neurons of each layer. These layers are black boxes: their outputs cannot be meaningfully interpreted, they can only be understood as some sort of derived feature that the network has found useful in generating its outputs.
- The output layer, which collects all the information from the last hidden layer and transforms it into the required shape, which depends on the problem: for regression problems, it outputs the $1-$dimensional numerical value, while for classification or multi-classification problems, it outputs the corresponding class of each sample.

The high amounts of connections between the neurons contained in these blocks of information is the source of predictive power this kind of models have. More specifically, an ANN that contains more than two hidden layers is known as a Deep Neural Network (DNN), which is represented in FIG. 1. However, these massive amount of connections that empowers ANNs is also the reason why these models have a disadvantage over conventional models: equivalently as their hidden layers, these models lack a meaningful interpretation of the results, since the power of ANNs to model complex relationships emerges from the "hidden" interactions between a large set of simple neurons, rather than from complex, but explicable, mathematical models.

As we can see in FIG. 1, all connections in a network between two neurons are directed, in the sense that information carried along each edge only flows in one direction. We call this a *Feed Forward Network* because the data is flowing forward, with early neurons feeding later neurons [10]. Moreover, each one of these connections has an associated *weight* with it: a connection weight is a single number that defines the way in which each neuron processes the information it receives along the connection. In other words, each weight represents the relevance of their corresponding input in the accurate computation of the output in question. The optimization of these weights is, in fact, the main goal of all ANNs. Therefore, we will first focus on how each neuron's weights are processed in a network.

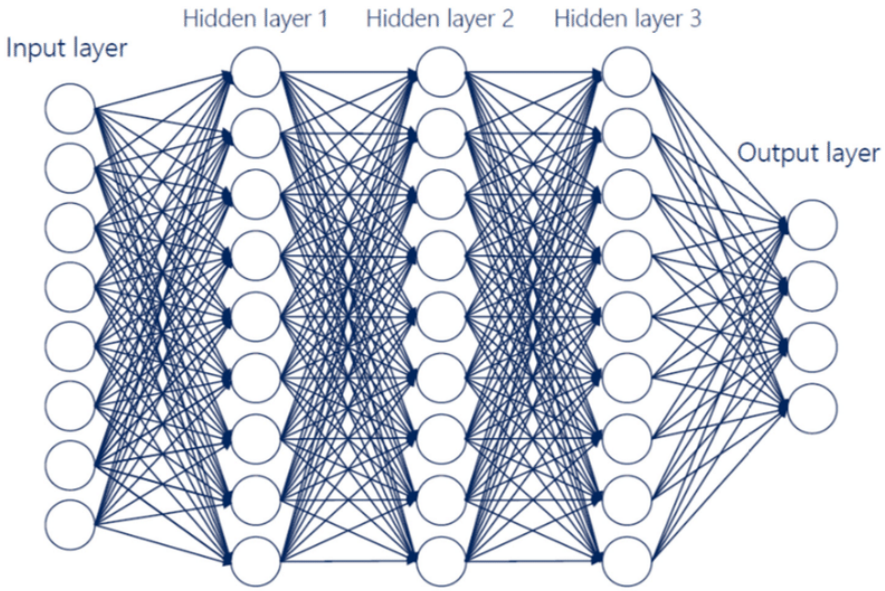


FIG. 1: DNN example [9], where circles represent the neurons of each layers, which are all connected by directed edges between contiguous layers.

B. Neuron structure

Each neuron in the ANN performs a two-phase procedure in order to transform a set of inputs into each neuron's output. The first stage of the process implements a *weighted sum*, which can be expressed as:

$$y = \sum_{i=1}^{n} x_i \cdot \omega_i + b \tag{20}$$

where x_i represents the i-th input, ω_i represents its associated weight, b represents the bias term and y is the neuron's output. Regarding the meaning of the bias, while weights enable an ANN to adjust the strength of connections between neurons, biases can be used to make adjustments within neurons themselves. This term is usually also thought as the weight of the $0-$input, which is defined as $x_0 = 1$. Therefore, (20) could be re-defined as:

$$y = \sum_{i=0}^{n} x_i \cdot \omega_i \tag{21}$$

Then, the second stage of the process introduces *activation functions*: the weighted value previously computed is passed through an specified function, always with a non-linear nature, that ends the mapping process from inputs to the final output. ANNs need non-linear activation functions in order to avoid the low accuracy caused by underfitting the model: only using linear-type computations such as in (21) in order to predict real-life models, where the vast majority of them are complex and non-linear, would result in a too simple, not accurate model. In section IV, we will introduce the most relevant activation functions used in deep learning and justify the particular use of them due to the purpose of our project. Moreover, in FIG. 2 we can graphically observe the internal functioning of each neuron in a network, displaying the two-phase process we just described.

Now, recalling what we previously stated, since the objective of an ANN is to optimize the set of weights, we will conclude this section by describing the best-known optimization algorithm to be used in deep learning: the *Adam* optimizer.

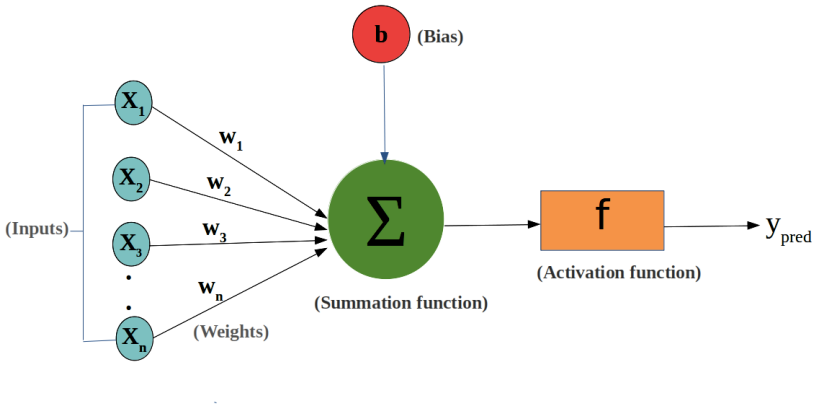


FIG. 2: Structure of an artificial neuron [11].

C. Weights' optimization using the Adam optimizer

ANN models aim to generalize the information they receive from the *training* data so that they can perform the possible accurate prediction on the unknown *test* data. During the training phase, the algorithm searches for the optimal set of weights by minimizing the *loss* function chosen by the user. Therefore we evaluate the derivatives of the *loss* with respect to the weights; then, these derivatives are used to find the weight values that minimize the error function by using optimization methods. Hence, at each realization of the algorithm through the complete dataset, known as *epoch*, of the network's training phase, weights and biases are updated towards the direction where the minimum of the loss function is aimed to be located, which is determined by the direction of each function's gradient. The algorithm for evaluating the derivatives of the loss function is known as back-propagation, as it corresponds to a propagation of the errors backward through the networks. Back-propagation is the core algorithm behind deep learning and is important to note that it is defined in two stages. In the first stage, we evaluate the derivatives of the error function with respect to the weights, and in the second, the derivatives are then used to adjust the weights. It is in the second stage where the optimization technique plays a tremendous role in the efficiency of the training phase [12]. In this updating, the *learning rate*, which is a parameter of the optimization function, plays a crucial role: when the optimal direction to follow is known, this parameter determines how "fast" we move along this road. There is a trade-off to be considered when setting this parameter: while small values can make the algorithm to get stuck at a local minimum (since it would take small steps in the right direction and may never arrive to the optimal destination), high values may cause to jump over the desired minimum and also end up at a local one. Therefore, correctly setting this parameter will be of great relevance, as we will later see in this section. Moreover, as we will later see in section IV B 4, the computational constraints caused by the high amount of possible parameter combinations led us to make some decisions in order to shorten this value. Therefore, in this project, we will focus on the *Adam* optimizer, which has been proven to be the best choice for ANN models, as we already saw in section I: it is straightforward to implement, has both faster running times and lower memory requirements and, finally, requires more minor tuning than any other optimization algorithm when trying to reach the optimal results.

Adam, whose name comes from adaptive moment estimation, is one of the deepest extensions of Stochastic Gradient Descent (SGD) and was designed to integrate together the advantages of two other SGD extensions: it combines the great performance of Adaptive Gradient Descent (AdaGrad) with sparse gradients and the capability of Root Mean Squared Propagation (RMSProp) of working with online data, allowing it to work in online learning. Adam's most relevant feature is its way of adapting the learning rate, which was inspired by AdaGrad: instead of maintaining a constant learning rate throughout the entire network as in SGD, Adam individually updates the learning rate for each weight, ω_i. Furthermore, since it uses both first and second moments of the gradients in the individual adaptive learning rates, it outperforms the updating process in RMSProp, which only considers the first moment.

In order to see the mathematical notation of these steps [13], let $f(\theta)$ be the stochastic objective function we are computing. Once the algorithm has computed its gradient at time t with respect to its θ parameters, $g_t = \nabla_\theta f_t(\theta)$,

Adam performs the mentioned first (m_t) and second (v_t) moment updates as follows:

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t = (1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} \cdot g_i \tag{22}$$

$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (g_t \odot g_t) = (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot (g_i \odot g_i) \tag{23}$$

where the $\odot$ symbol represents the element-wise product and β_1, β_2 monitor the exponential decay of these estimates, which can soften the learning curve and allow the algorithm to get out of local minima. Then, taking into account the bias of these updates towards 0, due to the particular initialization used, the algorithm can easily correct this deviation by matching the expectation of both m_t and v_t to their true first and second moment, respectively. This correction is a further improvement on RMSProp, which does not consider it. Before obtaining these results, let us first first express the sum of a finite geometric series, which will be used in the following computations:

$$S_n = \sum_{i=1}^{t} r^{t-i} = \left(1 + r + r^2 + ... + r^{t-2} + r^{t-1}\right) \cdot \frac{1-r}{1-r} =$$

$$= \frac{1}{1-r} \cdot \left(1 + r + r^2 + ... + r^{t-2} + r^{t-1} - r - r^2 - ... - r^{t-1} - r^t\right) = \frac{1-r^t}{1-r} \tag{24}$$

Then, the matches between update expectation and true moment values can be expressed as follows:

$$\mathbb{E}\left[m_t\right] = \mathbb{E}\left[(1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} \cdot g_i\right] = (1-\beta_1) \sum_{i=1}^{t} \mathbb{E}\left[\beta_1^{t-i} \cdot g_i\right] = \mathbb{E}\left[g_t\right] \cdot (1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} + \zeta = \mathbb{E}\left[g_t\right] \cdot (1-\beta_1^t) + \zeta \tag{25}$$

$$\mathbb{E}\left[v_t\right] = \mathbb{E}\left[(1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2\right] = (1-\beta_2) \sum_{i=1}^{t} \mathbb{E}\left[\beta_2^{t-i} \cdot g_i^2\right] = \mathbb{E}\left[g_t^2\right] \cdot (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} + \zeta = \mathbb{E}\left[g_t^2\right] \cdot (1-\beta_2^t) + \zeta \tag{26}$$

where we have defined $g_i^2 = g_i \odot g_i$ and ζ as the error coming from approximating any $\mathbb{E}[g_i] \approx \mathbb{E}[g_t]$. Moreover, note the use of (24) in the last step of both expressions above. As we can see, $(1-\beta_j^t)$ for $j = \{1, 2\}$ is the extra term between expected and true values. Therefore, Adam will correct this bias by canceling these last terms:

$$\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t} \tag{27}$$

$$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t} \tag{28}$$

Finally, the parameters are updated such that:

$$\theta_t \leftarrow \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{29}$$

where η corresponds to the learning rate and ϵ is a small constant for numerical stability, in order to avoid divergences caused by null denominators. If we want to express it in terms of the adaptive learning rate, η_t, it would look as follows:

$$\eta_t = \eta \cdot \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} = \eta \cdot \frac{\sqrt{\frac{v_t}{\hat{v}_t}}}{\frac{m_t}{\hat{m}_t}} \tag{30}$$

$$\hat{\epsilon} = \epsilon \cdot \sqrt{1 - \beta_2^t} \tag{31}$$

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \cdot \frac{\frac{m_t}{\hat{m}_t}}{\sqrt{\frac{v_t}{\hat{v}_t}}} \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} = \theta_{t-1} - \eta_t \cdot \frac{m_t}{\sqrt{v_t} + \hat{\epsilon}} \tag{32}$$

Concluding this section, now that we have deeply gone through the nature of the algorithm, we can better understand some of its other advantages. Firstly, re-scaling the gradient does not affect Adam's updates, which outperforms SGD and allows the algorithm to avoid getting stuck in regions where the gradient is small, such as saddle points. Then, Adam's adaptive learning rate has lower and upper bounds [13], which provide some mathematical intuition that other algorithms lacked for the value of this parameter. Finally, as we stated before, since it excels in both problems where AdaGrad and RMSProp do, Adam has a wider range to be implemented.

IV. Methodology

In this section we present the methodology used to estimate the price of a European call option, together with its respective Greeks, using ANNs. In order to achieve this, first the financial data was simulated. Then, we introduce an architecture of our ANN and its subsequent hyper-parameterization.

A. Data Simulation

First of all, in order to generate our simulated financial data, we make use of the BSM we previously introduced in section II for the case of a European call option. This model requires the following inputs in order for it to be able to compute the call price: stock price (S_t), time to maturity (τ), interest rate (r), volatility (σ) and strike price (K). We must note that the S_t is calculated using S_0 as input. Hence, from this point and using the BSM, the entire trajectory of the action is obtained until its maturity is reached. Then, applying the closed-form expression (4) of the BSM, we obtain the option price (V_t). We follow this process for all the possible combinations of these variables, which have their corresponding range of values shown in TABLE I.

Input	**Range**	**Step**
Interest rate (r)	[0.05,0.95]	0.1
Volatility (σ)	[0.05,0.95]	0.1
Time to maturity (τ)	[0.1,1]	0.1
Initial stock price (S_0)	[10,19]	1
Strike price (K)	[10,19]	1

TABLE I: Parameter range of the inputs.

Output	**Wide range**	**Narrow range**
Call price (V_t)	[0.0,799.62]	[0,50]
Scaled call price (V_t/K)	[0.0,49.97]	[0,1]

TABLE II: Parameter range of the outputs.

Note that, implicitly, using the strike and stock prices ranges, we can say that the range for the initial moneyness (at time 0), which is defined as $\frac{S_0}{K}$, goes from 0.53 to 1.9. As per the table presented above we note that the range of the scaled call and the call has a wide range of values. This range is not representative of our sample as the most dense area of the interval falls within $[0, 1]$ and $[0, 50]$. Finally, once we have the database with the European call prices and their corresponding parameters, we proceed to calculate the Greeks using the derivatives obtained from the BSM close formula. It is important to mention that the data for the final simulation has a total of 1 million observations, which results from the combination of all variable values multiplied by 10, which is the extension of each call path (the aforementioned S_t).

B. ANN Implementation

According to what we previously stated, the original goal of our project was to build an ANN that predicts European call prices as accurate as possible, taking into account that the real price is that of the BSM. In order to do that, the target variable (the y) of our original model was set to be the scaled option price (V_t/K), which we will later justify. Then, once we had our trained ANN, our next goal would be performing *Gradient tape* on it, in order to see if Greek predictions can also be accurately obtained with our approach. It is important to mention that this procedure is completely different from predicting Greeks using ANNs that directly output them; i.e., ANNs that have their targets defined as the Greeks. Alternatively, since our ANN will be outputting option prices and the Greeks are defined as the derivatives with respect of their various variables and parameters, we will see if deriving these predictions also yields accurate predictions. However, note that our original model uses V_t/K as output and, as we will later see, the moneyness as one of its inputs. Hence, we will propose two models, each one with a different ANN structure that allows us to compute: firstly, the scaled price of the European call, corresponding and defined as our **Original model**; secondly, the regular European call price with the right inputs, so we are able to compute the corresponding Greeks, which will be defined as our **Greeks model**. In the upcoming sub-sections we will deeply describe both models, their inputs, outputs and the different data treatment of each one.

1. Architecture

a Original model

For this first model we use the following variables as inputs: interest rate (r), volatility (σ), time to maturity (τ) and moneyness (S_t/K). Then, the chosen output set to target the training phase will be the the option price calculated by the BSM (V_t) divided by the strike price (K). The reason we define the model's output in this way, as well as using the moneyness instead of the strike and asset price separately, is to scale down the output and increase the model's predictive power. Additionally, we standardize the variables by subtracting the mean and dividing them by the standard deviation, before passing them through the ANN. It is worth mentioning that centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set; hence, the standard score of a sample x is computed as:

$$z = \frac{x - \bar{x}}{\delta_x} \tag{33}$$

where $\bar{x}$ is the mean of the training samples and δ_x is the standard deviation of the training samples. Scaling both the inputs and target variable is well-known to be a good practise in regression modelling. The scaling of the data eases the learning process for a machine learning model, making it to better understand the problem. Particularly in deep learning, an independent variable with a wide spread of values may result in a large loss in training and testing, which can turn into having an unstable learning process. Finally, recall that due to not having any null values in our dataset, since we simulated the data from scratch, we were released from having to perform any imputation techniques in order to take care of empty values.

However, we must note that this model does not allow calculating the Greeks: due to the scaled call price used (V_t/K), the gradients (corresponding to the derivatives) that we would obtain from the target variable with respect to the inputs would not match the definition of any Greek. In order to obtain them, we would be making use of *Gradient Tape*, an API for automatic differentiation included within TensorFlow that allows to compute the gradients of the outputs with respect to the inputs. The structure that we follow for this model consists of 4 nodes in the input layer, which correspond to each of the variables, followed by 3 hidden layers and a final output layer where we use the target variable (V_t/k). Note that we have chosen the number of hidden layers as the most basic DNN structure, since we wanted to surpass the complexity of a simple ANN without building a too complex one. Therefore, since there is no consensus on the right number of layers to use and we are also limited by our computational power, as we will later discuss in section VI, we have chosen the minimum number of layers for our model to be defined as an ANN. This first model's architecture can be seen in the FIG. 3.

b Greeks model

Now, for this second model, while the inputs used are the interest rate (r), volatility (σ), time to maturity (τ), stock price (S_t) and strike price (K), the output is just the price of the European call (V_t). Here, there will be no data

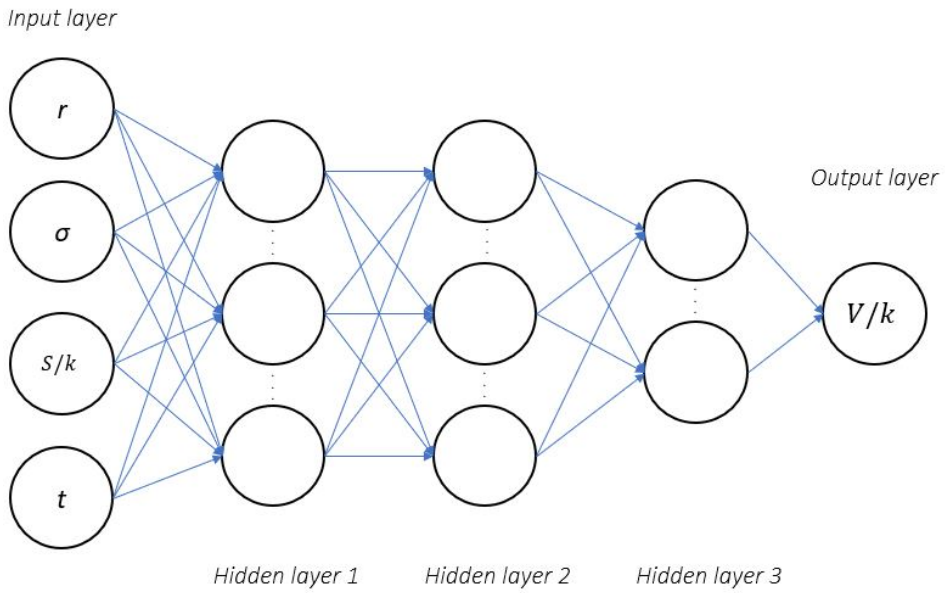


FIG. 3: Basic DNN architecture for the Original model.

scaling of any type: in order to be able to compare the true Greek values and the ones we will extract from the derivatives of the predicted call price with respect to each parameter, we cannot perform any scaling of the data, since we would be comparing non-analogous values. Hence, the price to pay of this model, for it to be able to compute the Greeks, will be the learning stability provided by the data scaling. As we can see, with this new input/output structure we can already obtain the Greeks, since the gradients of the output with respect of the input parameters will directly match four out of the five Greeks we introduced in section II B: ρ, υ, Θ and Δ. Finally, the structure is similar to that of the original model, with the same basic DNN structure, as shown in FIG. 4.

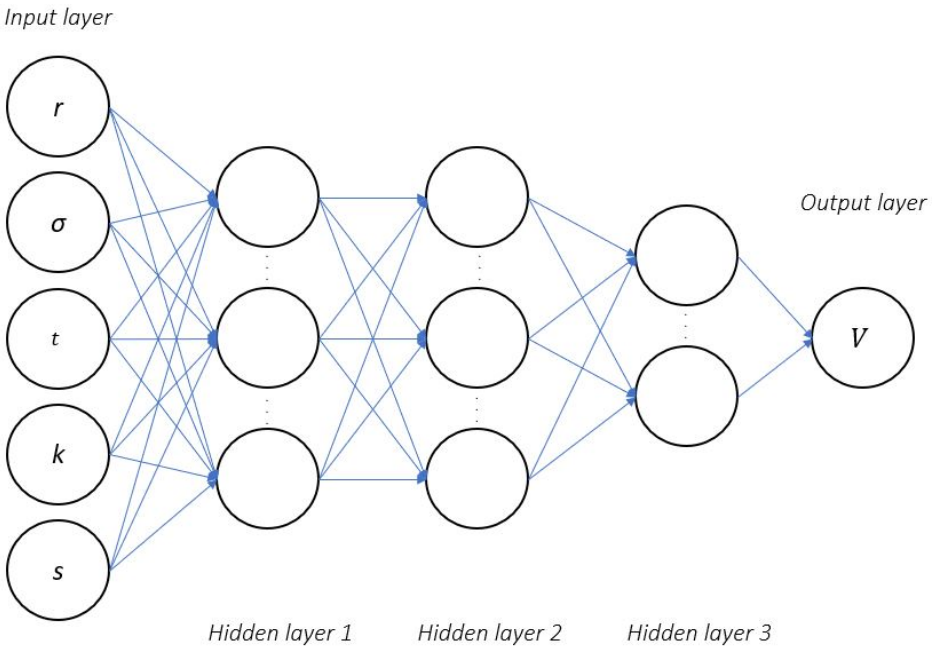


FIG. 4: Basic DNN architecture for the Greeks model.

2. Activation functions

As mentioned in section III, activation functions define how the weighted sum of the inputs is transformed into an output from a node, or nodes, in a layer of the network. ANNs need non-linear activation functions in order to avoid the low accuracy caused by underfitting the model. Therefore, we have to select the activation function that best suits our model. Below we explain and go through the most used functions in deep learning, in order to justify which one best fits our model:

a Rectified Linear Activation (ReLU)

ReLU is a piece-wise non-linear function that will output the input directly if it is positive and, otherwise, it will output zero. It can be easily expressed as:

$$g(x) = \max(0, x) \tag{34}$$

It has become the default activation function for many types of ANNs, since a model that uses it is easier to train and often achieves better performance results. More specifically, it is usually used in any regression problem, specially for those problems where negative solutions are not a possibility. Among the advantages of this activation function [14], we must highlight the following: its computational simplicity, since it only requires the maximum function; its capability of generating a true zero value, which is known as a sparse representation and is a desirable property in representational learning, as it can speed up the learning process and simplify the model; and finally, its well-known performance in training ANNs with many hidden layers.

b Sigmoid

The sigmoid function is a special form of the logistic function and is usually denoted by $\sigma(x)$. It is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{35}$$

When passing the sigmoid function through a neuron, it guarantees that the output of this unit will always be between 0 and 1. Furthermore, as the sigmoid is a non-linear function, the output of this unit would be a non-linear function of the weighted sum of inputs. Therefore, it is especially used for models where the output has the form of a probability score, which is equivalent to a classification problem, since it has the right range of values.

c Hyperbolic Tangent (Tanh)

Tanh is also like sigmoid but the range of the tanh function is from -1 to 1. tanh is also sigmoidal:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{36}$$

The tanh function is mainly used in classification problems between two classes. Together with the sigmoid activation function, they are used in feed-forward classification networks. The advantage of it is that the negative inputs will be mapped strongly negative, while the zero inputs will be mapped near its null value.

Taking into consideration the three described activation functions, we can conclude that the one that best suits our case is the ReLU, since we are facing a regression problem and our target variable, the European call price, can only take values greater or equal than 0, without having any type of bound between 0 and 1 (such as the one provided by the Sigmoid) or between -1 and 1 (for the Tanh).

3. Performance evaluation and loss function

Before entering into details of our training phase, we must introduce the metrics that will be used to assess the performance of the models. As we stated in section III C, the chosen optimization algorithm (Adam, in our case) updates weights and biases towards the direction where the minimum of the loss function is aimed to be located. Therefore, one must choose this loss function for the algorithm to know what to minimize. In our case, the regression

nature of our problem leads us to choose the MSE as the loss function that is set to be optimized. It tells you how close a regression line is to a set of points and its mathematical expression is:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2 \tag{37}$$

where y is the true value, $\hat{y}$ is the estimated value from the regression and N is the training sample size. Therefore, the lower the MSE, the better the model's prediction is.

Furthermore, as an additional metric to be computed by the DNN at each epoch, we have chosen the R^2 coefficient, which provides a measure of how well observed outcomes are replicated by the model, based on the proportion of total variation of outcomes explained by the model. Its mathematical expression is:

$$R^2 = 1 - \frac{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}{\frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2} = 1 - \frac{\text{MSE}}{\text{Var(y)}} \tag{38}$$

where y is the true value, $\hat{y}$ is the estimated value from the regression, $\bar{y}$ is the mean estimated value from the regression and N is the training sample size. We can clearly see the dependence of this coefficient with the MSE, since they are opposite metrics.

4. Training and Hyper-Parameter Optimization

Following, in this section we explain the training process and the optimization of the hyper-parameters for two proposed models. We must understand training and hyper-parameter optimization as the process in which the model is tested with different parameter values (such as learning rate, number of neurons, etc.) to finally select the one that results into the lowest loss function. Therefore, we begin by dividing the data into the training and test sets, using a proportion of 80% and 20%, respectively. Additionally, 20% of the training data will be used as the validation data during the training process in order to avoid over-fitting: oppositely to under-fitting, this occurs when a model is only capable of performing good on the data it is fed, resulting in poor performances when tested with unseen data. This concept will be further detailed in section V. Our approach will propose to optimize the hyper-parameters in two stages, where we will perform a first *Random Search* and then, a *Grid Search*, both of them using the *scikit-learn* package in Python.

a Random Search

In the first stage of our optimization process, we use a Random Search to tune the hyper-parameters: it consists in independent draws from a uniform density from the same configuration space that will be spanned by a regular grid, as an strategy for producing a trial set. In contrast with a Grid Search, this technique reduces the tried parameter combinations to a fixed and chosen value. Random search best feature is its efficiency in hyper-parameter optimization for the case of several learning algorithms, due to not all hyper-parameters being equally important to tune: it trades off computational running time and quality of the solution when choosing the sets of parameters to try out. Furthermore, random experiments are also easy to carry out for practical reasons, which are related to the statistical independence of every trial and lower computational time required [15]. By means of this tool, we optimize the following hyper-parameters:

1. Initialization of parameters

   The first parameter that must be considered when building an ANN is the initialization of parameters (or weights): if done correctly, the optimization algorithm will converge faster to the optimal set of weights. Otherwise, converging to a minimum could take unfeasible running times. In this case we are going to try out the two most common initializers:

   - Random uniform: generates weights and biases with a uniform distribution, which means that the initial weights are randomly set between 0 and 1.

- Glorot uniform (*Xavier* uniform initializer): draws weights from a uniform distribution within $[-\text{lim}, \text{lim}]$, which are given by:

$$\text{lim} = (\frac{6}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}})^{0.5} \tag{39}$$

  where fan_{in} is the number of input units in the weight tensor and fan_{out} is the number of output units.

2. Learning rate

   Since we use the Adam optimizer, we must understand this parameter as the proportion in which the weights are updated. Larger values (e.g. 3×10^{-1}) result in faster initial learning before the rate is updated, while smaller values (e.g. 3×10^{-5}) slow down the learning process during training. The values to be tried out are the following: $[1 \times 10^{-3}, 1 \times 10^{-4}, 1 \times 10^{-5}]$.

3. Number of neurons in hidden layers

   The hidden layers, and the corresponding number of neurons used in each one, are responsible for the learning capabilities of the network, as discussed in section III. It is important to optimize this hyper-parameter: while a small number could produce under-fitting, as the network may not learn enough, a high one could produce over-fitting, since the network would learn too much from training data without generalizing. Hence, there must be an intermediate number of neurons that ensures a good and balanced training. In our case, we use the following values for the amount of neurons: $[64, 128, 256, 512]$.

   Additionally, we will be testing whether reducing, or not, the number of neurons in each hidden layer improves the results. The way in which we test this is by halving, or not, the number of neurons from the previous hidden layer. E.g., in the case of having 64 neurons in the first hidden layer, the second layer would have 32 and the third 16. This technique is commonly used in image prediction training as it can be seen in [16], where it is applied to a Real-Time Object Detection model. Even though ANNs perform great with huge amounts of neurons, which divide the information into many small pieces, reducing the number of neurons while advancing through the net is also considered to be a good practise in regression problems: since it will eventually have to provide a single, final output value for each sample, squeezing the information in each layer can help the model accomplish that goal.

4. Batch size

   Batch size defines the number of samples we use in one epoch to train an ANN. For instance, let's consider the training size of 1000 samples and the batch size of 100. The DNN will take the first 100 samples in the first epoch and do forward and back-propagation. After that, it will take the next 100 samples in the second epoch and repeat the process until the set number of epochs is met or the desired condition is reached. The batch size affects several indicators in the training phase: while high values take less running time but can result into a too quick and over-fitted learning process, low values can significantly increase the running time and provide an under-fitted results. Therefore, taking into account the size of our training data, the values to be tried out in the random search are: $[512, 1024, 2048, 4096]$.

5. Batch normalization

   Batch normalization is a technique when training ANNs: it standardizes the inputs in a layer for each batch through a normalization step that fixes the means and variances of each layer's inputs. This has the effect of stabilizing the learning process, preventing overfitting and dramatically reducing the number of training epochs required to train ANNs. We test whether or not using this technique increases the performance of the model; i.e., we define this parameter as a boolean or binary variable.

6. Dropout rate

   ANNs are likely to quickly over-fit a training dataset. In order to avoid this, we can randomly select a percentage of nodes that will be dropped. For our case of study, we try whether or not dropping a fixed 10% of neurons from the last hidden layer improves the results. Again, this is treated as a boolean or binary variable.

As a summary, TABLE III shows all the hyper-parameters with which we train the DNN and their possible values:

Input	Range/Options
Initialization	Uniform, Glorot_uniform
Learning rate	$[1 \times 10^{-3}, 1 \times 10^{-4}, 1 \times 10^{-5}]$
Number of neurons	[64,128,256,512]
Decreasing neurons	Yes, No
Batch size	[512,1024,2048,4096]
Batch normalization	Yes, No
Dropout rate	[0,0.1]

TABLE III: Hyper-parameter values to be optimized in the first random search phase.

Additionally, we use 50 epochs to train the model with the random search, a cross-validation of 3 and a patience of 2. An epoch is the moment when the model has processed the whole training data set and we implement the cross-validation to reduce the probability of over-fitting. On the other hand, the patience parameter defines the early stopping criterion, which corresponds to the number of epochs in which training is stopped once the model's performance stops improving: i.e., if two epochs go by and the loss function has not decreased, the algorithm stops and moves on to the next set of parameters or cross-validation set to be tried out.

b Grid Search

Now, in the second stage of our optimization process we use Grid Search, which is the simplest algorithm for hyper-parameter tuning. Basically, we divide the domain of the hyper-parameters into a discrete grid. Then, we try every combination of values of this grid while calculating some performance metrics using cross-validation. The point of the grid that maximizes the average value in cross-validation corresponds to the optimal combination of values for the tested hyper-parameters. Grid search is an exhaustive algorithm that spans all the combinations, so it can actually find the best point in the domain. Nevertheless, its great drawback is the high running time required when dealing with a large parameter grid, which is why we do not use it as a first stage.

The reason we perform this second optimization phase is to re-train the model with all the optimal parameters returned by Random Search but re-optimizing batch normalization and dropout rate, both as the same binary variables as before. We have chosen these two parameters since they are the ones that can be easily interpreted and are useful in the prevention of over-fitting: the fact that random search limits the number of combinations to be tried out can result into missing the optimal set of parameters. Therefore, once we get the best parameter values chosen by the random search, we try again these two variables to make sure that they are completely optimized. On the contrary, we do not try the rest of hyper-parameters, since their values are tried based on heuristics and we cannot outperform a machine in choosing the best set of them. This time, since our parameter grid has a length of 4 (two possible values of the two parameters to be tried out) we use a higher cross-validation (10) and we keep the previous 50 epochs. In this way, we ensure that the model does not present over-fitting and we can obtain more reliable training metrics, which have been previously introduced.

Finally, once the model has been further optimized in this second stage, we train the final parameters again, now using 200 epochs and a patience of 10. Thus, we expand the capacity for the model to improve and we can obtain better and more clear results.

V. Results & Discussion

A. Final architecture of the models

To begin with, the final structure of the Original model is shown in FIG. 5 and its corresponding optimal parameters in TABLE IV. In the same way, FIG. 6 and TABLE V show the same information for the Greeks model.

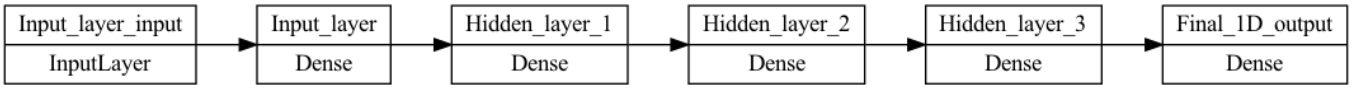


FIG. 5: Final DNN architecture for the Original model.

Parameter	Final value
Epochs	200
Activation function	ReLU
Number of hidden layers	3
Initialization	Uniform
Learning rate	1×10^{-5}
Number of neurons	256
Decreasing neurons	Yes
Batch size	512
Batch normalization	No
Dropout rate	No

TABLE IV: Optimal hyper-parameters for the Original model.

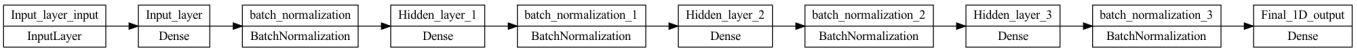


FIG. 6: Final DNN architecture for the Greeks model.

Parameter	Final value
Epochs	200
Activation function	ReLU
Number of hidden layers	3
Initialization	Uniform
Learning rate	1×10^{-3}
Number of neurons	512
Decreasing neurons	Yes
Batch size	2048
Batch normalization	Yes
Dropout rate	No

TABLE V: Optimal hyper-parameters for the Greeks model.

One of the differences between the Original and the Greeks model is the learning rate. In the case of the first model, we obtain an optimal learning rate of 1×10^{-5}, while for the case of the Greeks model, this number increases to 1×10^{-3}. Other differences are found in the number of neurons and the batch sizes: the Greeks model uses larger values than the Original one, with 512 vs. 256 for the number of neurons and 2048 vs. 512 for the batch size. However, both models agree on decreasing the number of neurons at each layer, which means that compressing the information at each layer helps the model in predicting the call prices, as we already suggested in section IV B 4. Another interesting feature that we can observe is in regard of the two boolean variables we used to prevent overfitting. While neither of the models use a dropout layer, they differ in the use of batch normalization: the Greeks model uses it and not the Original one.

All of these differences can be explained by the fact that the Greeks model has more input variables and a larger scale on both input and output variables. This could then be passed to a more complex DNN to approximate the form of the function that generates the call price.

B. Training performance results

Now, in this section we show the obtained results in the training phase in terms of the two performance metrics we introduced in section IV B 3: MSE and R^2. These results correspond to the last stage of the training phase, once the parameters have already been optimized in the random and grid searches, where we perform the last step with 200 epochs and a patience of 10, as we previously specified.

FIG. 7 and FIG. 8 show how the improvement of the MSE at each epoch for both models: Original and Greeks. In the last epoch, the Original model achieved an MSE of 4.62×10^{-7} in training and validation set. On the other hand, the Greeks model had a worse result, with an MSE of 6×10^{-2} for the training set and 5×10^{-2} for

validation set. To begin with, the first feature worth mentioning is the different evolution behaviors both models had: in the case of the Original model, the MSE curve throughout the epochs is much smoother than the evolution for the Greeks model. This means that the learning process of the Original model happens in a nice, continuous and constant way, since the model does not learn neither too fast or too slow, decreasing the loss function at each epoch, with just a few minor points with instabilities. On the contrary, the validation learning process for the Greeks model has a completely opposite behavior: as it can be observed, the curve is constantly veering from low to high values, resulting in an unstable learning process for the validation set. The reason for this opposite behavior is the nature of the data used in each model. As we saw in section IV B 1, while the inputs for the Original model went through several scaling steps, in order to improve the model's performance, the ones for the Greeks model did not went through these steps, in order for it to be eligible to compute the corresponding Greeks. Hence, stability in the learning process was the price to pay for the Greeks, as we stated, and can be perfectly noticed in the results.

Then, it must also be pointed out the behavior of both models regarding over or under-fitting signals: using this validation set during the training phase makes the model to test its performance on unseen data at each epoch. Therefore, in an ideal learning process, we would have aligned learning curves for both the train and validation sets, meaning that the model would be learning for both known and unknown data, without losing generalization (over-fitting) nor losing the right complexity (under-fitting). Hence, starting with the Original model, since the training and validation curves are almost perfectly aligned, which represents the learning stability we previously commented, it presents the perfect balance in over or under-fitting issues along the whole learning process. Now, regarding the Greeks model, the constant veering in the validation's loss function value causes the model to have several over-fitting periods, in which the network struggles to learn on unseen data and only improves in the training data. However, even though this behavior does not match with the ideal learning process, the final values obtained in the last epoch are equivalent for both train and validation sets, therefore concluding in a final and balanced result.

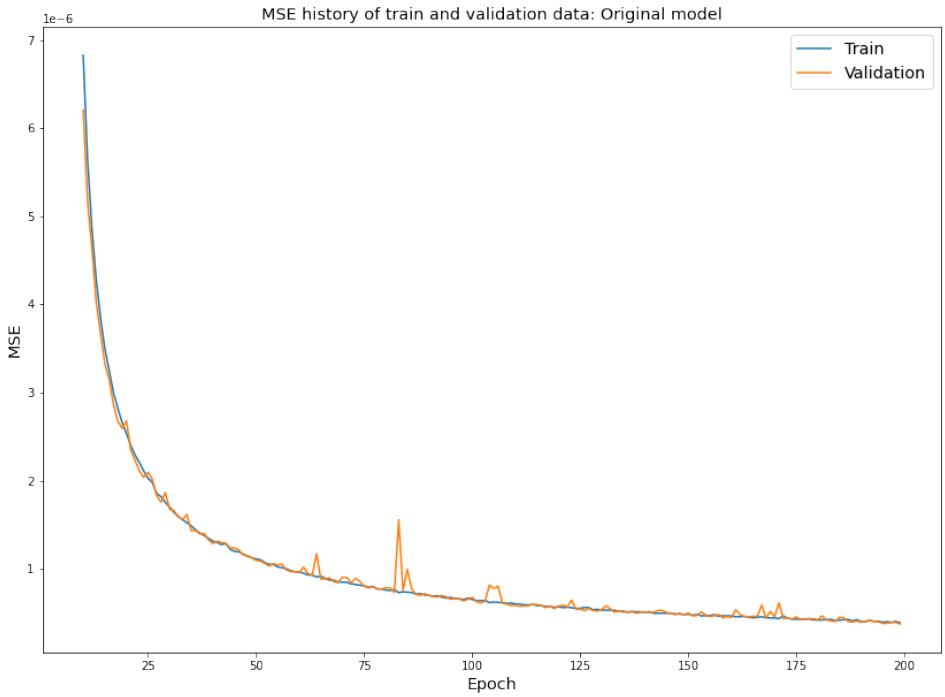


FIG. 7: Loss function (MSE) comparison between the train and validation data of the original DNN model.

Finally, regarding FIG. 9 and FIG. 10, where we show the improvement of the R^2 coefficient at each epoch for both models, its behavior and most noticeable characteristics can be totally explained with the comments regarding the MSE, since R^2 is totally related to the MSE value, as we saw in (38). Here, we can see that both models end up with superlative results: while the Original model has a final R^2 of 1, indicating the perfect prediction results in both training and validation data, the Greeks model has R^2 values of 0.9986 and 0.9989 for the training and validation data, respectively, which are extremely close to 1. Hence, both models show perfect prediction performances in the training phases. Moreover, it is also worth noticing that, even though the Greeks model still shows clear evidence of instability, the range of values shown in FIG. 10 for the R^2 coefficient is extremely narrow: in other words, even

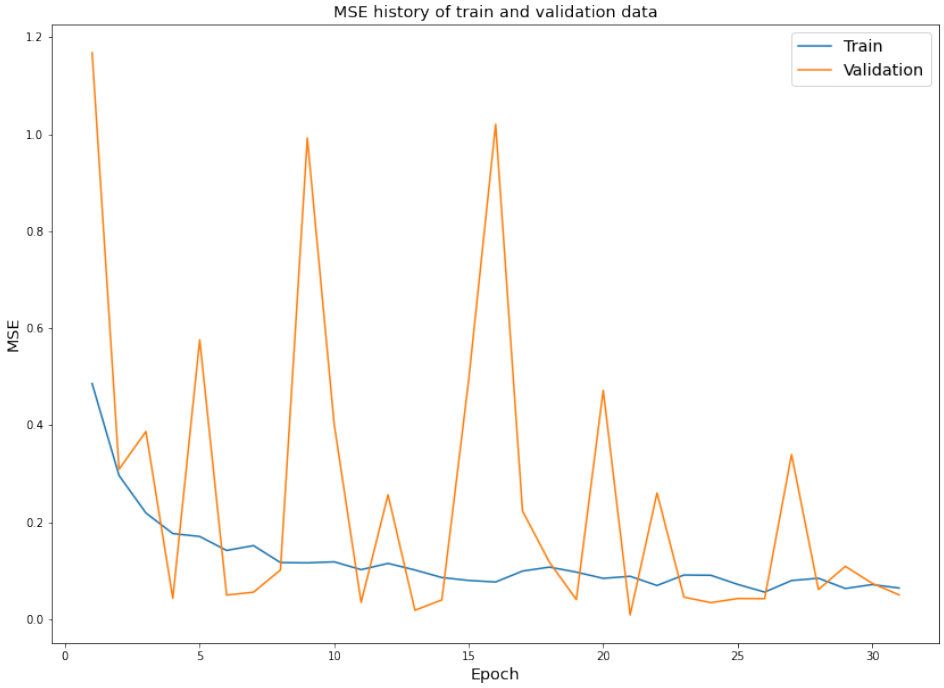


FIG. 8: Loss function (MSE) comparison between the train and validation data of the Greeks DNN model.

though it seems to decrease to much lower R^2 values in some epochs, the reality is that the changes are of the order of 0.02 in R^2 values of, at least, 0.975. Hence, the actual final results are unstable yet accurate, still.

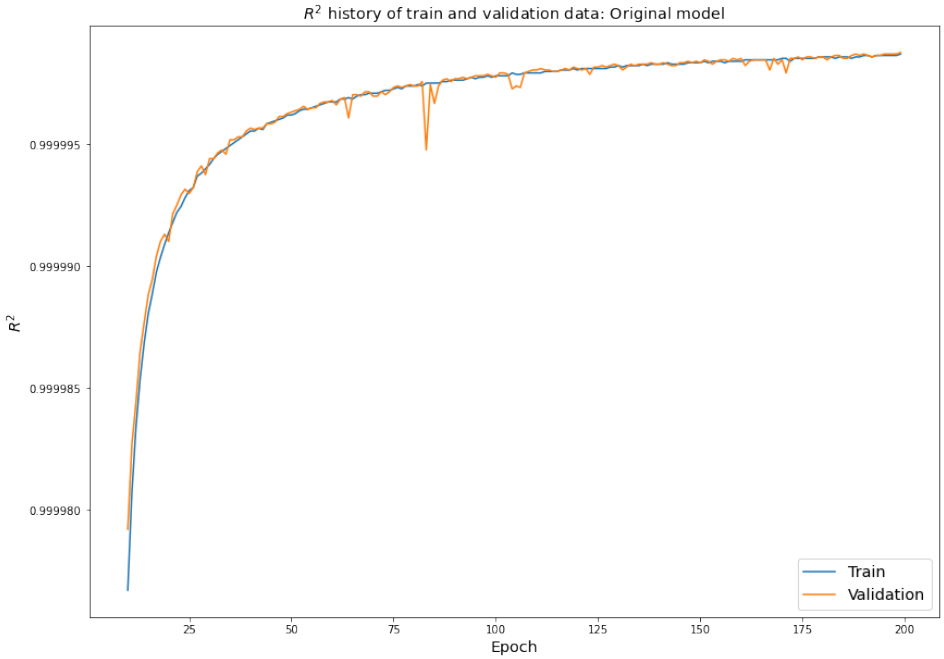


FIG. 9: Accuracy metric (R^2) comparison between the train and validation data of the original DNN model.

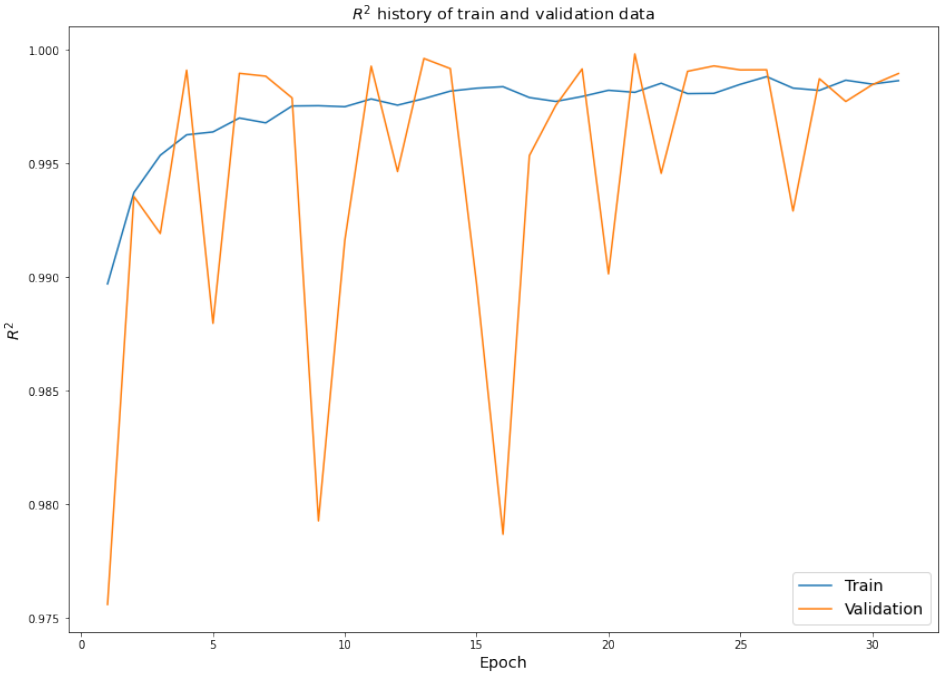


FIG. 10: Accuracy metric (R^2) comparison between the train and validation data of the Greeks DNN model.

C. Predictions

Concluding this section, with both final and trained DNN models, we will test their performance in predicting the relevant financial models for this project: European call prices under the Black-Scholes formula and their corresponding Greeks. Besides from the two metrics already used in the training phase, we will base our results in three extra regression metrics that measure the accuracy for continuous variables: Root Mean Squared Error (RMSE), Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE).

To begin with, RMSE, which measures the standard deviation of residuals, just corresponds to taking the square root of the MSE, such that:

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2} \tag{40}$$

Then, MAE measures the average magnitude of the errors in a set of forecasts, without considering their direction.

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}| \tag{41}$$

It is worth noting that all of the aforementioned metrics weight the individual value differences equally. Thus, we introduce our final metric. When dealing with financial or economical data, such as financial options, we may also want to care about relative quantities, instead of just focusing on absolute quantities such as the ones we have already introduced. Thus, since European call options values can range from 0.001 to > 1, we can also test the performance of our model with the relative error $\frac{\hat{y}-y}{y}$, in order to have a more individualized metric for all option prices. With MAPE, which we already introduced before, we can precisely measure how close in percentage the model estimation is relative to the actual data. This metric is computed as:

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\frac{|\hat{y}_i - y_i|}{|y_i|} \tag{42}$$

where $\hat{y}$ is the estimated value from the regression and y is the true value. Thus, equivalently as the previous metrics, the lower the MAPE, the better the estimation model.

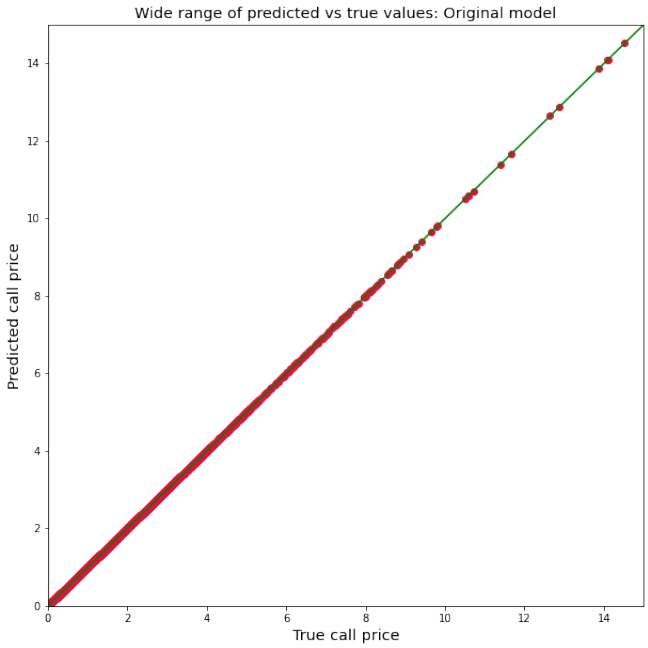
Firstly, in TABLE VI below we display the obtained results in predicting European call prices, using both models and each metric with the unseen data stored in the test set. As one could expect from the obtained validation results, we see that predictions for both models are very close to the true values, since the R^2 values go to 1. Then, by looking at the rest of the error metrics, it seems that the results for the Original model are better than for the Greeks model. However, it must be noticed that while the Greeks model uses the value of the call price as output, without any scaling or standardization steps applied, the Original one uses scaled inputs and output. Hence, the range of values that the DNN has to be trained with is much wider in the Greeks model than in the Original, which will obviously result in higher error values: if your target variable values are all between 0 and 1, the error metrics will have much lower scores than the ones coming from a variable that is spread in a much wider range, such as the $[0, 200]$ interval. Therefore, in order to do a fair comparison regarding the loss of both models, we will be focusing in the MAPE results: while for the Original model this metric is interpreted as the average prediction being 8.7% far from the actual price, for the Greek model this metric increases to a 16.6% of error. Therefore, we can say that the Original model outperforms the Greeks model, leading us to acknowledge that the preprocessing steps done have helped the model understand the problem.

Model	**MSE**	**RMSE**	**MAE**	**MAPE**	$\mathbf{R^2}$
Original Model	4.62×10^{-7}	6.80×10^{-4}	4.55×10^{-4}	8.70×10^{-2}	1.00
Greeks Model	1.06×10^{-2}	1.03×10^{-1}	6.60×10^{-2}	1.66×10^{-1}	1.00

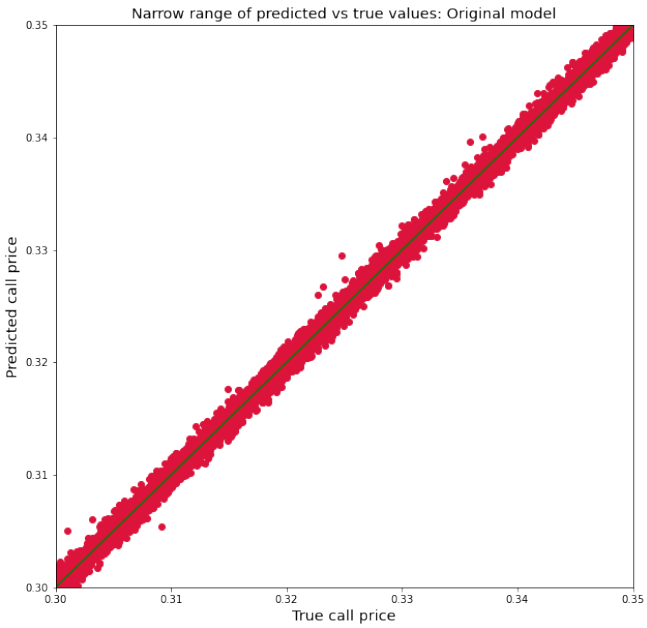
TABLE VI: Original and Greeks DNN model performance on the test set.

Then, FIG. 11 and FIG. 12 show the comparison between the true and predicted call prices (true values in x axis and predicted values in y axis). Thus, the way to understand this graph is the following: if the ordered pair of values, distributed each one along the x and y axes, results in points located on the drawn 45-degree line, which corresponds to the regression line of the model, the prediction is equal to the true value one. Hence, as we can see in FIG. 11a and 12a and we could expect from the R^2 values shown in TABLE VI, both models present almost all points located in the regression line, corresponding to highly accurate predictions. Moreover, since this result could be misinterpreted due to the wide scale of the axes used, FIG. 11b and 12b show the same information but with a narrower interval of call prices. As in the previous plots, the predictions are very close to the real ones: even though this time we can see points that are not perfectly on the line, the high proximity of them towards it and the small range of values used shows the high accuracy of the predictions.

Moreover, in FIG. 13 below we can see the corresponding predictions vs true plots for the Greeks: ρ, υ Θ and Δ. Recall that these predictions were obtained using the Greeks model and applying the *Gradient tape* tool described in section IV. It can be easily noticed that there is no Greek where the predictions are close to the real, true values: the case of ρ is the best-predicted Greek but still, the predictions are very far from the real values calculated with the closed-form expressions from the BSM. In the next, we will provide the relevant conclusions of this issue.

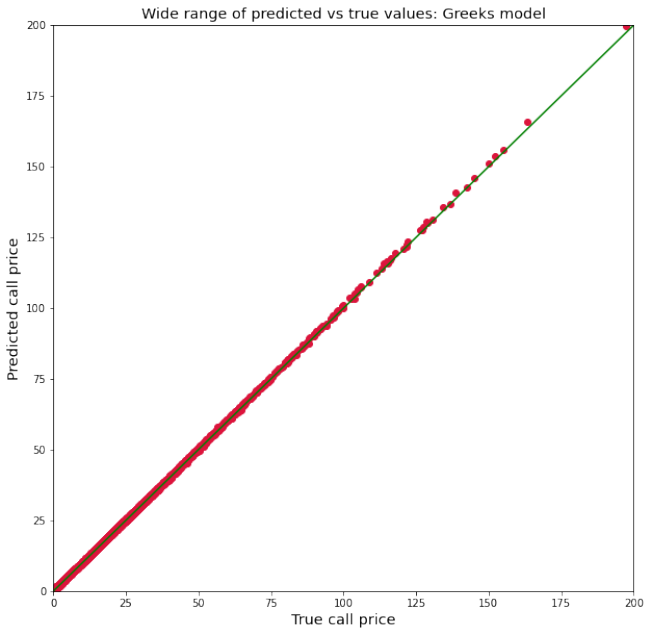


(a) Wide interval of call prices: $[0, 14]$.

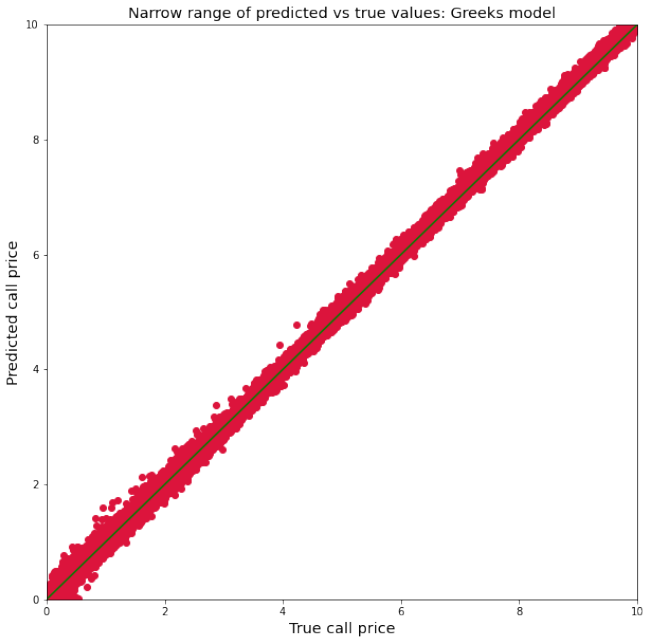


(b) Narrow interval of call prices: $[0.3, 0.35]$.

FIG. 11: Predicted vs true values of the call price for the original DNN model.

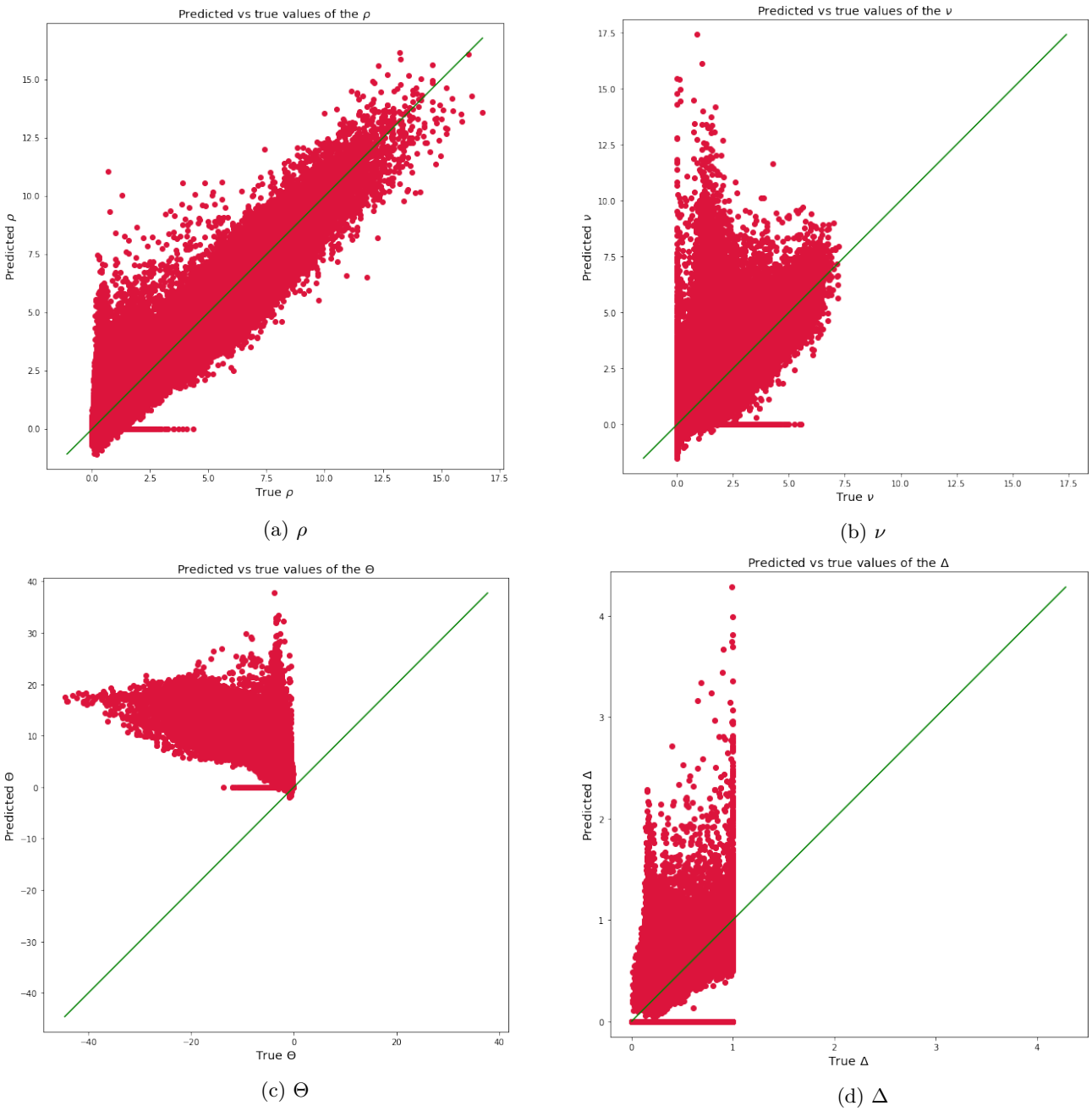


(a) Wide interval of call prices: $[0, 200]$.



(b) Narrow interval of call prices: $[0, 10]$.

FIG. 12: Predicted vs true values of the call price for the Greeks DNN model.



(a) ρ (b) ν

(c) Θ (d) Δ

FIG. 13: Predicted vs true values of the four computed Greeks in our modified DNN model: ρ, ν, Θ and Δ.

Finally, TABLE. VII shows the time (in seconds) it took to run the two models on the training and prediction stages (call price and Greeks), compared to the time it takes to calculate these values with the BSM. In the case of the predictions, the time shown corresponds to the time it takes each model to obtain the results for $200,000$ samples. Clearly, our models present a better performance than the BSM in terms of prediction running time. For the call prices, the BSM took 162.03 seconds, while our models took 1.03 (Original model) and 0.54 (Greeks model), i.e., they are approximately 157 or 300 times faster. However, it should be noted that DNNs have a disadvantage: the training time (random, grid and final columns) is a very long stage. For the case of the Original model, the training took 35.6 hours and in the Greek model, 18.5 hours. On the other hand, using only the BSM this training time is not necessary. Let's move to the next and last section of this project, where we will also provide further conclusions on this matter.

Model	Random	Grid	Final	Predict	Greeks
Original	58699.77	65438.51	4000	1.03	-
Greeks	61079.67	5319.62	352	0.54	0.15
Black-Scholes	-	-	-	162.03	152.74

TABLE VII: Duration in seconds of the different stages during the train and test phases of the models: Original, Greeks and Black-Scholes.

VI. Conclusions

In this paper we built two ANNs, both of which predicted the call prices and the difference between them relied on their main objective: while the first DNN (*Original model*) focuses on the predictive performance of the model regarding the option prices, the second DNN (*Greeks model*) was developed with the purpose of estimating the Greeks of the model.

The main objective of the paper is to examine whether ANNs are a reasonable proxy of stochastic models. To do so we simulated European call option prices based on the BSM, a stochastic closed formula to test this assumption. The extent of our simulation includes both "In The Money" and "Out of the Money" options, which allows us to train the DNN with a wide range of moneyness. Then we used the outputs of the simulation, S_t/K moneyness, time to maturity $T-t$, interest r, and volatility σ as the inputs of the *Original model* to obtain V/K as the corresponding output. Equivalently for the *Greeks model*, we used the price of the underlying S_t, strike K, time to maturity $T-t$, volatility σ, and r interest rate, to estimate the corresponding output: V, the call price. Two stages were used to optimize the hyper-parameters of the DNN: Random Search and GridSearch. The five measurements defined in the paper, R^2, MSE, RMSE, MAE, and MAPE, indicate that these DNNs reproduce the outcomes generated by the stochastic process. Regardless, we observe a slightly better performance in the *Original model* than in the *Greeks model*. The improved performance of the *Original model* compared to the *Greeks model* is due to how the inputs are introduced. By introducing inputs scaled by the strike, the model learns more clearly and smoother as the range of variables it takes in and the range of its output is smaller compared to a model that does not scale. On the other hand, it is essential to mention the differences in data preprocessing in both models. The critical difference in the preprocessing of the models is the scaling of the data; while the *Original model* is scaled in the preprocessing, the second model is not. The objective of the *Greeks model* did not allow for this preprocessing as it sought to compare the derivatives of the DNN with the derivatives obtained by the closed formula.

The results of both models are rather optimistic in terms of predictability; however, we must not lose sight of the limitations of these models in terms of their applicability in the real world. Firstly, it is well known that a machine learning model is only as good as the data it is fed. Therefore, it is essential to highlight our training data's simulated - not real - nature. The data simulation is the result of a stochastic close formula. The BSM is based on assumptions that limit the model's capability to represent the natural world. Hence the prices that its estimates approximate the prices in the real world. When we think of real-world financial data, we think of data that includes periods of extreme volatility. The variability of the economic variables that govern the estimation of the call prices can negatively affect the predictive power of our model if this is not properly trained and calibrated. To improve the performance of the DNN in periods of high volatility is vital to train the DNN with data that replicates similar inputs. There are two options to have said data: through simulation or real-world data with periods of high volatility. Both options are costly. While obtaining real-world data represents monetary cost, generating simulated data that includes said periods is computationally expensive.

Building on the computation cost of the original BSM compared to the ANN alternative. It is evidently more computationally expensive to price European option using the traditional BSM, as it is approximately 160 times slower than its not parametric alternative. However we must not ignore the fact that all Machine Learning algorithms, in order to perform at its optimal they need to be both trained and continuously calibrated. The training and the calibration of the model can potentially be time-consuming and computationally expensive. Note that in our particular example, in order to train this models, the training phase lasted 35 hours for the original model and 18 hours for the Greeks model.

Now, regarding the predictions of the Greeks, the results in FIG. II B clearly show that our approach is not valid for this purpose. Let's try to understand why this happens. The final output of any ANN is computed by an

approximated expression of the target variable; i.e., in our case, if our target variable followed the BSM closed-form expression, the final expression used to compute the output should aim to best resemble this closed-form result. It is important to highlight this last line: it tries to best resemble the *result*, not the dependencies with the parameters. In fact, as we already stated, one of the disadvantages of deep learning is the lack of interpretability of their results: it is not possible to extract information regarding how the output depends on the inputs. Hence, ANNs do not try to imitate the target dependencies, but its numerical value instead, since they are data-driven models. Therefore, it is completely reasonable to think that, even though we may be getting to almost equivalent numerical results, the mathematical expressions to get to these results can totally differ. Recalling, our approach (using the Greeks model) was to obtain the derivative of the output (option value, V) with respect to each input and see if it was equivalent to computing the Greeks, since they are the derivatives of V with respect to each parameter: following what we have said, even though the accuracy of option prices predictions is superlative (reaching $R^2 \sim 1$), the dependencies do not have any indicator to be the same. Therefore, obtaining completely different results when deriving both expressions (the closed-form BSM and the approximated DNN expression) is totally reasonable.

Finally, it is important to point out that the perfect optimization technique when building ANNs is still one big mystery. In our case, we have implemented two hyper-parameter searches in order to find the best set of them, where we encountered computational limitations: we found ourselves forced to perform an initial random search over the whole parameter grid, since trying the totality of combinations would have ended up in an unfeasible computational running time. Hence, since this first random search limited the combinations to a 100, as we already stated in section IV B 4, we may have missed the optimal set of parameters due to not being able to go over all the possible combinations. Then, we performed a following grid-search, where we only optimized the Boolean parameters (whether to use, or not, batch normalization and/or drop out) and we kept constant the already found ones in the random search: its purpose was to have these Boolean parameters completely optimized, since they were the only ones that could be interpreted in terms of preventing over or underfitting, as we already covered. On top of these limitations, this optimization process is originally limited by the chosen range of parameters to be examined: there is no exact science in the number of layers, neurons or learning rate values, among others. All things considered, the reason these limitations are faced is caused by the no existence of a clear methodology to construct a guaranteed optimal architecture. Nevertheless, by having access to more powerful machinery, we could extend both the parameters' range and their amount of combinations, in order to improve our model prediction power.

[1] J. M. Hutchinson, A. W. Lo, and T. Poggio, "A nonparametric approach to pricing and hedging derivative securities via learning networks," *The journal of Finance*, vol. 49, no. 3, pp. 851–889, 1994.
[2] J. Yao, Y. Li, and C. L. Tan, "Option price forecasting using neural networks," *Omega*, vol. 28, no. 4, pp. 455–466, 2000.
[3] J. Bennell and C. Sutcliffe, "Black–scholes versus artificial neural networks in pricing ftse 100 options," *Intelligent Systems in Accounting, Finance & Management: International Journal*, vol. 12, no. 4, pp. 243–260, 2004.
[4] J. D. Kelleher, *Deep learning*. MIT press, 2019.
[5] P. Benardos and G.-C. Vosniakos, "Optimizing feedforward artificial neural network architecture," *Engineering applications of artificial intelligence*, vol. 20, no. 3, pp. 365–382, 2007.
[6] S. Liu, C. W. Oosterlee, and S. M. Bohte, "Pricing options and computing implied volatilities using neural networks," *Risks*, vol. 7, no. 1, p. 16, 2019.
[7] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, p. 637–654, 1973.
[8] P. Wilmott, *Paul Wilmott on quantitative finance*. John Wiley & Sons, 2013.
[9] M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for ai-enabled iot devices: A review," *Sensors*, vol. 20, no. 9, p. 2533, 2020.
[10] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*, vol. 4. Springer, 2006.
[11] S. Ganesh, "What's the role of weights and bias in a neural network?," *Towards Data Science*, 2020. `https://miro.medium.com/max/1400/1*upfpVueoUuKPkyX3PR3KBg.png`.
[12] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995.
[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[14] B. A. a. B. Y. Glorot X., "Deep sparse rectifier neural networks," *Proceedings of Machine Learning Research*, vol. 15, pp. 315–323, 2011.
[15] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.
[16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2015.