Università degli Studi di Milano-Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di Laurea in Informatica

American Option Pricing via Hyperparameter Optimization of Deep Neural Networks

Relatore: Prof. Candelieri Antonio

Relazione della prova finale di:
Paolo D'Elia
Matricola 844630

Anno Accademico 2021-2022

A miei genitori, per il loro sostegno quotidiano.

Abstract

Nowadays, the increasing availability of data and access to computational resources is making the application of Deep Learning easier. Its application regards not only domains characterized by a high signal-to-noise ratio, but also other domains characterized by a higher uncertainty, such as the Financial sector.

One of the most challenging problems in finance is the pricing of options. Given the competitiveness of a market-making environment, it is essential to quickly quote option prices coherently with an ever-changing market environment. Fixed parametric models do not respond to structural changes in the data like newer machine learning models do. This thesis proposes a data-driven approach to tackle this problem, leveraging the power of Artificial Neural Networks and Bayesian Optimization.

The programming language used for this work is Python with the support of Data science libraries such as NumPy, Pandas, Jupyter Notebook and PyTorch.

Finding the best hyperparameters for a model often involves tons of trials by hand or preliminary knowledge of experts, which can be efficient in selecting influential parameters and narrowing down the search space. This can be avoided using a probabilistic hyperparameter optimization framework, such as Bayesian Optimization, which provides a robust configuration of the model's hyperparameters. The resulting Neural Networks aren't only trained on data generated from well-known pricing models for American Options, but also on real market data, to fully prove that they can be considered universal function approximators, given enough training data.

The numerical results apart from showing that the Neural Network pricer can approximate, with an excellent degree of accuracy, any given pricing model, show how the Deep learning model is sensitive to small variations of the input data.

Contents

Introduction

In a competitive sector, such as the Finance sector, there is a need to use ever more precise and faster approaches. Numerical methods are commonly used for the valuation of financial derivatives. Generally speaking, advanced financial asset models can capture nonlinear features that are observed in the financial markets. However, these asset price models are often multidimensional, and, as a consequence, do not give rise to closed-form solutions for option values.

Furthermore, since American options are path-dependent, conventional pricing models are computationally expensive. These approaches include the Binomial model, partial differential equation solvers, and Monte Carlo methods.

All these conditions made the right spot for the application of Deep Learning. As shown in [1] and [2] a data-driven approach is faster and can accurately approximate different pricing models, both with fixed volatility and stochastic volatility.

Differently, from parametric models, Neural Networks can adapt to new market conditions as new training data comes in. This property makes this approach more generalizable than other kinds of approaches.

The thesis is structured as follows: first options contracts are introduced in Chapter 1, then in Chapter 2 Neural Networks are introduced. The methods of collecting and creating the data used to train the neural network are described in Chapter 3. Chapter 4 briefly introduce Bayesian Optimization which is the optimization method used to find the best hyperparameters of the Neural Network models. The bayesian optimization results and the results of the final fine-tuned models in the various datasets are illustrated respectively in Section 5.1, Section 5.2.1, Section 5.2.2, and Section 5.2.3.

In the lastest two sections results of the volatility surfaces are presented.

Chapter 1

Option

An **option** is a contract that gives its owner, the right, but not the obligation to buy or sell an underlying asset or instrument at a specified strike price on or before a specified date, depending on the type of the option. The option is a financial derivative whose value depends on the underlying asset. Since options are derivative there must be also a form of valuation of the price of the option that may depend on a complex relationship between the underlying asset value, time until expiration, market volatility, and other factors.

An option that conveys to the holder the right to buy at a specified price is referred to as a **call option**, while one that conveys the right to sell at a specified price is known as the **put option**.

1.1 Contract Specifications

A financial option is a contract between two counterparties with the terms of the option specified in a term sheet. Option contracts may be quite complicated; however, at minimum, they usually contain the following specifications:

- whether the option holder has the right to buy (**call option**) or the right to sell (**put option**)
- the quantity and class of the **underlying asset(s)**
- the **strike price**, also known as the exercise price, is the price at which the underlying transaction will occur upon exercise
- the **expiration** date, or expiry, which is the last date the option can be exercised
- the **settlement's terms**, for instance, whether the writer must deliver the actual asset on exercise, or may simply tender the equivalent cash amount
- the terms by which the option is quoted in the market to convert the quoted price into the actual **premium** – the total amount paid by the holder to the writer

Option styles There are different styles of options, some of the most common are:

- **American Option** - may be exercised on any trading day on or before the expiration date
- **European Option** - may be exercised on the expiration date
- **Bermudian Option** - may be exercised only on predetermined dates before expiration; so this option is somewhat between European and American
- **Asian Option** - the payoff of these types of options is determined by the average of the price of the underlying asset during some period of time within the life of the option.

1.2 Payoff and profit of options

The payoff of an option depends on the possibility of exercising it or not. If the option is exercised its payoff is the difference between the exercise price and the price of the asset at maturity, but if it is not exercised the payoff is 0. We can define precisely the payoff functions for the call and put contracts of vanilla options. if K is the strike price and S_T is the price of the underlying asset at the exercise date T, then the payoff for a call option is

$$\max(S_T - K, 0)$$

since we would only exercise the option if $S_T > K$. The payoff function for a call option is shown in the Figure 1.1.

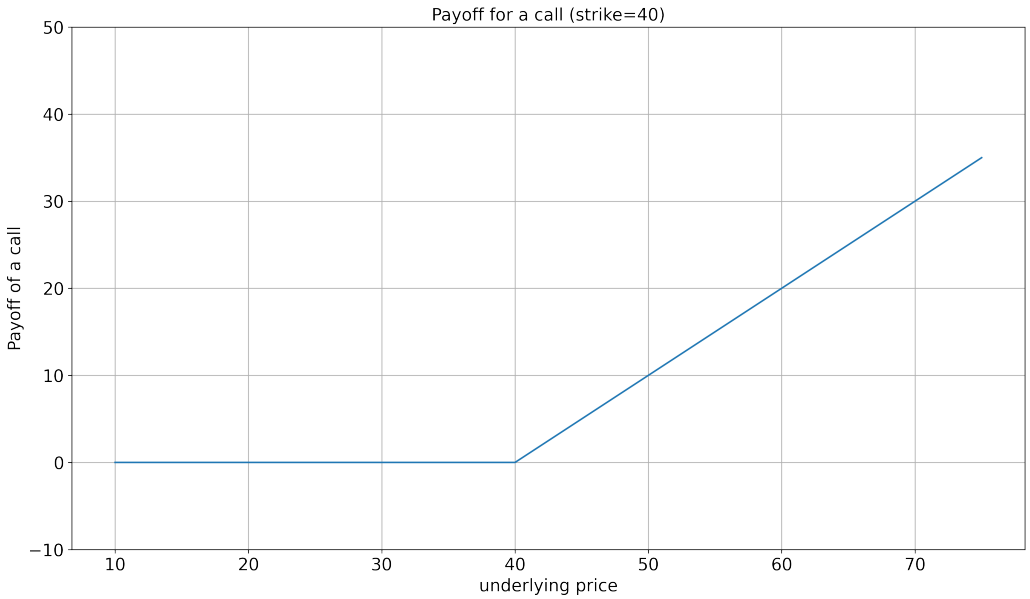


Figure 1.1
Payoff function for a call option

By analogous reasoning the payoff for a put option is

$$\max(K - S_T, 0)$$

The payoff function for a put option is shown in the Figure 1.2.

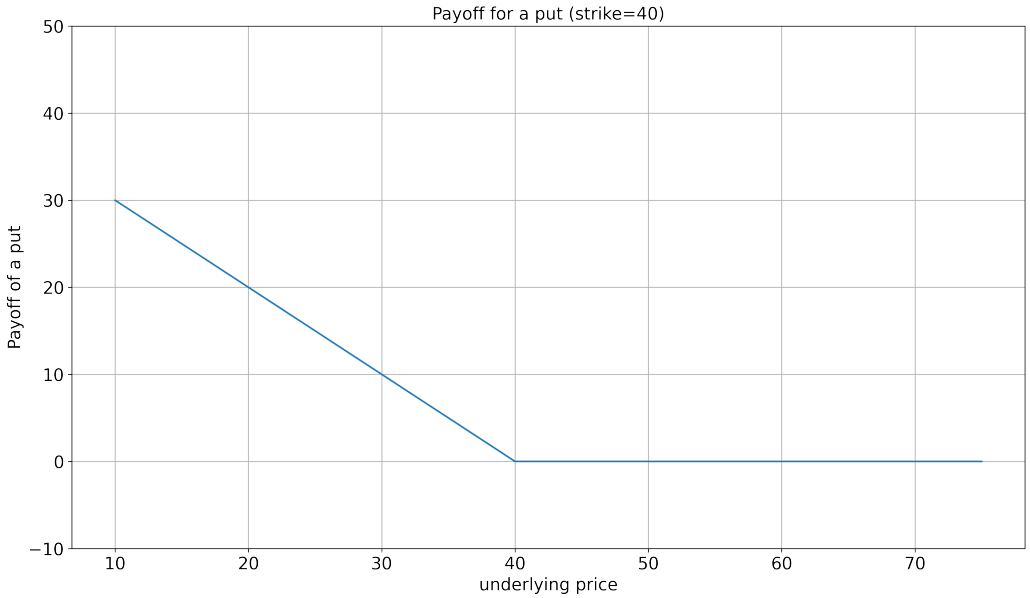


Figure 1.2
Payoff function for a put option

Since buying an option is not free, the profit function differs from the payoff function by the option **premium**. The premium of an option usually depends on several factors: the price S_0 of the underlying asset at that initial time t_0, the time T set for maturity, the volatility of the underlying asset σ, the strike price of the option K and a constant interest rate r for the period $\tau = T - t_0$. We denote $C(S_0, T)$ as the price of the option, which is always a positive amount. For calculating the profit of the option we should deduct the price paid for the option for the payoff. But we have to consider also that these amounts are given and received at different points in time, and that we could have done a safe investment instead, we must include the possible gains given by a risk-free asset with a constant interest rate r. Therefore, under all these considerations, the profit for buying a call option is:

$$\max(S_T - K, 0) - C(S_0, T)e^{r\tau}$$

If on the one hand, there is someone who buys an option, on the other hand there must be someone who sells it. A call option seller has an obligation to sell the underlying asset to the call buyer at a fixed price (the strike price). If the underlying price decreases, the seller of the call makes a profit in the amount of the premium. If the underlying price increases over the strike price by more than the amount of the premium, the seller loses money, with the

potential loss being unlimited.

If the seller does not own the stock when the option is exercised, they are obligated to purchase the stock in the market at the prevailing market price.

The profit for selling a call option is:

$$C(S_0, T) - \max(S_T - K, 0)$$

Figure 1.3 shows the profit function for buying/selling a call option.

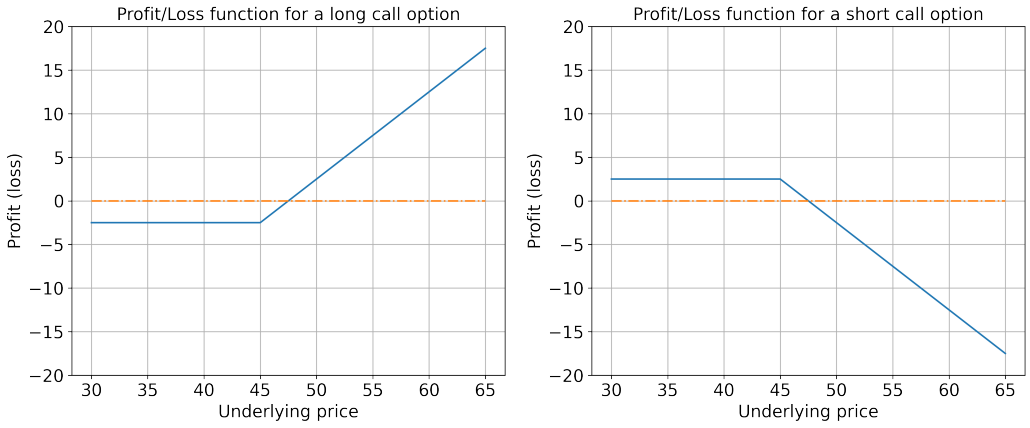


Figure 1.3
Profit function for buying/selling a call option

Similar reasoning can be done for a put option. Therefore, the profit for buying a put option is:

$$\max(K - S_T, 0) - P(S_0, T)e^{r\tau}$$

where $P(S_0, T)$ is the price of the put option. While the profit for selling a put option is:

$$P(S_0, T) - \max(K - S_0, 0)$$

Figure 1.4 shows the profit function for buying/selling a call option.

1.3 Option Valuations

Because the values of option contracts depend on a number of different variables in addition to the value of the underlying asset, they are complex to value. There are many pricing models in use, although all essentially incorporate the concept of *rational pricing*, *moneyness*, *option time value*, and *put-call parity*.

The valuation itself combines a model of the behavior ("process") of the underlying price with a mathematical method that returns the premium as a function of the assumed behavior.

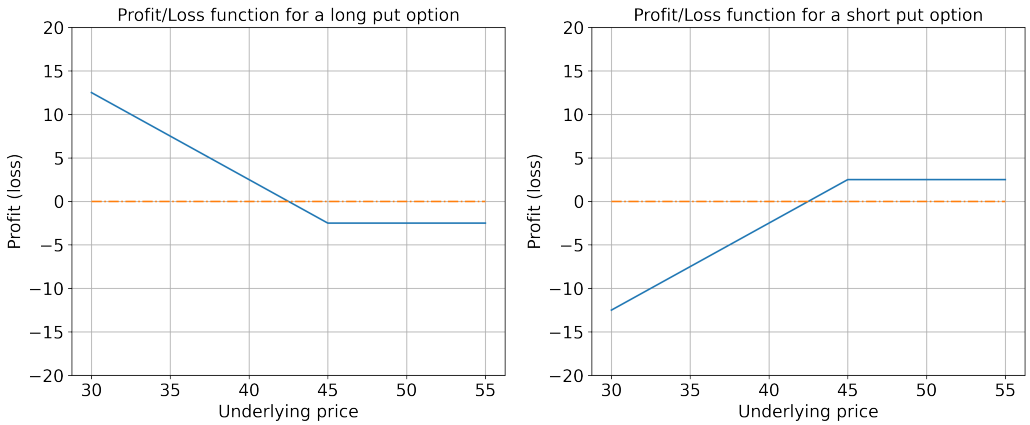


Figure 1.4
Profit function for buying/selling a put option

1.3.1 Basic decomposition

In its most basic terms, the price of an option is decomposed into two different parts:

- **Intrinsic value**: the difference between the underlying spot price and the strike price, to the extent that this is in favor of the option holder. For a call option, the option is in-the-money if the underlying spot price is higher than the strike price; then the intrinsic value is the underlying price minus the strike price. For a put option, the option is in-the-money if the strike price is higher than the underlying spot price; then the intrinsic value is the strike price minus the underlying spot price. Otherwise, the intrinsic value is zero.

- **Time value**: the option premium is always greater than the intrinsic value. This extra money is for the risk that the option writer is undertaking. This is called the time value. Time value is the amount the option trader is paying for a contract above its intrinsic value, with the belief that prior to expiration the contract value will increase because of a favorable change in the price of the underlying asset. The longer the length of time until the expiry of the contract, the greater the time value. So $TimeValue = OptionPremium - IntrinsicValue$

Other factors that are influencing the price of an option are:

- **Spot price**: any fluctuation in the price of the underlying asset has a direct effect on the price of the option contract. An increase in the underlying price causes an increase in the premium for a call option while decreasing the premium for a put option. The opposite is true when the underlying price decreases.

- **Strike price**: how far the spot price from the strike price also affects the premium. The more the spot price is far from the strike price the less the option premium, while the closer the spot price is to the strike price, the more expensive the option will be. That's because out of the money option must be cheaper than at the money options.

- **Volatility of the underlying**: the more volatile the underlying is the more expensive the option will be. This is because higher volatility increases the risks of the option seller.
- **Interest free-rate**: the risk-free rate also influences the premium; that's because the money invested by the buyer to buy the call could be invested in a risk-free asset, so the profit of a call option should be discounted considering that that money could be invested in a risk-free asset.
- **Dividends**: dividends do not directly impact the price of the derivative, but they influence the price of the stock. We know that if dividends are paid, the stock goes ex-dividend, therefore, the price of the stock will go down which will result in an increase in the put premium and a decrease in the call premium.

1.3.2 Pricing Models

The different approaches to price an option are:

- **Closed form** models: the most basic of these is the Black-Scholes formula and the Black model.
- **Lattice models**: Binomial options pricing model, Trinomial Tree
- **Monte Carlo methods** for option pricing
- **Finite difference methods** for option pricing
- **Volatility surface**-aware models in the local volatility and stochastic volatility families.

1.3.2.1 Black-Scholes-Merton Model

The Black-Scholes-Merton model is a mathematical model for pricing European options. From the partial differential equation in the model, known as the Black-Scholes equation, one can deduce the Black-Scholes formula, which gives a theoretical estimate of the price of European-style options and shows that the option has a unique price given the risk of the security and its expected return.

The key idea behind the model is to hedge the option by buying and selling the underlying asset in just the right way and, as a consequence, to eliminate risk. This type of hedging is called "continuously revised delta hedging".

Model Assumptions The model assumes that the market consist of at least one risky asset, usually a stock, and one riskless asset, usually called the money market, cash, or bond.

Regarding the assets, it is assumed that:

- the *rate of return* on the riskless asset is constant and thus called the risk-free interest rate.

- the stock price follows a *geometric Brownian motion*, where the drift and volatility are constant
- the stock does not pay dividends.

The assumption regarding the market are:

- The is no *arbitrage* opportunity
- there is the ability to borrow and lend any amount, even fractional, of cash at the riskless rate
- there is the ability to buy and sell any amount, even fractional, of the stock
- there are no fees or transaction costs

Black-Scholes equation The Black-Scholes equation is a partial differential equation governing the price evolution of a European call or European put under the Black-Scholes model. For a European call or put an underlying stock paying no dividends, the equation is:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where V is the price of the option as a function of stock price S and time t, r is the risk-free interest rate, and σ is the volatility of the stock.

Interpretation of the Black-Scholes PDE The equation can be rewritten in the form:

$$\underbrace{\frac{\partial V}{\partial t}}_{\text{time decay}} + \underbrace{\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}}_{\text{gamma}} = \underbrace{rV - rS\frac{\partial V}{\partial S}}_{\text{riskless position}}$$

where in the left-hand side $\frac{\partial V}{\partial t}$ is called also *theta* which is the change in derivative value with respect to time, $\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2}$ is the convexity of the derivative value respect to the underlying value. While the right-hand side consists in a riskless position from a long position in the derivative and short position in the underlying of $\frac{\partial V}{\partial S}$ shares.

The equation states that over any infinitesimal time interval the loss from theta and the gain from the gamma term must offset each other so that the result is a return at the riskless rate.

Black-Scholes formula In order to solve the Black-Scholes equation boundary conditions must also be provided. In the case of a call option those conditions are:

$$\begin{aligned} C(0,t) &= 0 \quad \text{for all } t \\ C(S,t) &\to S \quad \text{as } S \to \infty \\ C(S,T) &= \max\{S-K, 0\} \end{aligned}$$

The solution of the equation in the case of a call option is:

$$C(S,t) = S_t\Phi(d_1) - e^{-r(T-t)}K\Phi(d_2)$$

where:

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

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

and $\Phi(\cdot)$ is the cumulative distribution function of the standard normal distribution. The price of a corresponding put option based on *put-call parity* with discount factor $e^{-r(T-t)}$ is:

$$\begin{aligned} P(S_t,t) &= Ke^{-r(T-t)} - S_t + C(S_t,t) \\ &= \Phi(-d_2)Ke^{-r(T-t)} - \Phi(-d_1)S_t \end{aligned}$$

1.3.2.2 Binomial option pricing model

The **Binomial options pricing model** (**BOPM**) provides a generalizable numerical method for the valuation of options. Essentially, the model uses a "discrete-time" model of the varying price over time of the underlying financial instrument.

Since the BOPM is based on the description of an underlying instrument over a period of time rather than a single point. As a consequence, it is used to value American options that are exercisable at any point in a given interval.

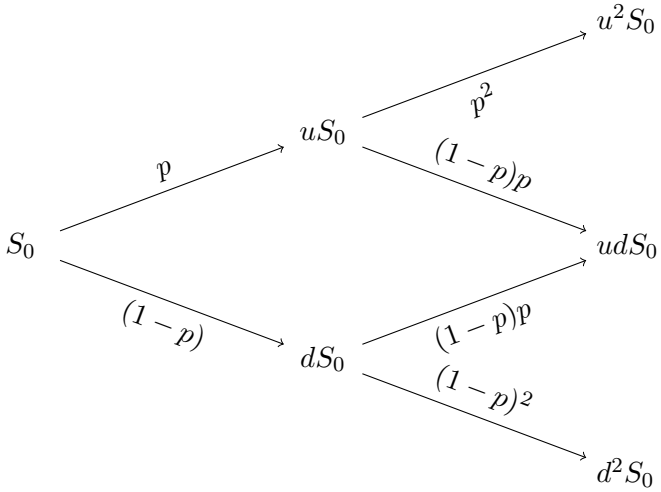
Model Assumptions The BOPM assumes that the daily continuous growth rates for the underlying stock are normally distributed around zero (the mean is $\alpha = 0$) with some variance σ^2. Although these assumptions are not quite true, they are close enough to true in certain circumstances to be useful.

The stock price is assumed to be a discrete-time process with some timestep Δt, and that at each timestep the stock price goes up by a factor of u or goes down by a factor of $d = \frac{1}{u}$. These two factors come from the assumption that the price is an Ito process with an α of zero. Therefore, we can compute the two factors from the volatility of the stock, and let:

$$u = e^{\sigma\sqrt{\Delta t}}$$
$$d = e^{-\sigma\sqrt{\Delta t}}$$

where σ is the volatility and $\sqrt{\Delta t}$ is a time-adjustment factor to scale the volatility by the timestep duration.

Create the binomial price tree Once we have u and d, we can starting at time $t = 0$, compute the possible stock prices at times $t = k\Delta t$ for all k starting from zero and going to the expiration date of the option. We can build a tree with one node for each possible stock price at each timestep, starting from $t = 0$ and $S = S_0$. The next timestep $t = \Delta t$ will then have two nodes, one for uS_0 and one for dS_0. The timestep afterwards for $t = 2\Delta t$ will have (technically) four nodes equal to u^2S_0, udS_0, duS_0 and d^2S_0. However, note that $ud = du = u\frac{1}{u} = 1$, which means that we can collapse the internal nodes into one. Therefore, at time $t = k\Delta t$, the will be a total of $k + 1$ nodes, because you will have prices equal to $S_k = u^i d^{k-i} S_0$ for every $i \in \{0, 1, \cdots, k\}$ representing the number of up ticks.



Find option value at each final node At each final node of the tree (i.e. at the expiration of the option) the option value is simply its instrinsic, or exercise, value:

$$\begin{aligned} &\max(S_k - K, 0) && \text{for a call option} \\ &\max(K - S_k, 0) && \text{for a put option} \end{aligned}$$

where K is the stike price and S_k is the spot price of the underlying asset at the k^{th} period.

Find values of earlier nodes In order to proceed further, we need a method of computing the option price at the internal nodes of the binomial model tree. For each internal node, the *binomial value* is found, using the risk neutrality assumption. If exercise is permitted at the node, then the model takes the greater binomial and exercise value at the node. Under the risk neutrality assumption, today's fair price of a derivative is equal to the expected value of its future payoff discounted by the risk-free rate. Therefore, the expected value is calculated using the option values from the later two nodes (Option up and Option down) weighted by their respective probabilities - "probability" p of an up move in the underlying, and "probability" $(1 - p)$ of a down move. The expected value is then discounted at r, the risk-free rate corresponding to the life of the option. The binomial value for each node is calculated as:

$$\text{Binomial Value} = [p \times \text{Option up} + (1-p) \times \text{Option down}]$$

or

$$C_{t-\Delta t,i} = e^{-r\Delta t}(pC_{t,i} + (1-p)C_{t,i+1})$$

where C_t, i is the option's value for the i^{th} node at time t, $p = \frac{e^{(r-q)\Delta t} - d}{u-d}$ is choosen such that the related binomial distribution $X \sim Binom(n,p)$ simulates the random geometric Brownian motion of a stock with parameters r and σ, q is the dividend yield of the underlying corresponding to the life of the option. Note that though it may be tempting to say that the binomial value is the options price, this may not be the case; in American style options, every node also has the option of exercising the option, so the options price is the maximum of the binomial value and the profit garnered by exercising the option at that point in time. The profit may be assessed in exactly the same manner as the computation for the leaves, with two cases, one for call and one for put options. However, there exist European style options, where an early exercise is not an option, so the binomial value is the options price; similarly, there exist Bermudan style options, where early exercise is only an option at some nodes, and only at those nodes do you choose the maximum of the potential profit and the binomial value.

1.3.2.3 Trinomial option pricing model

An extension of the binomial option pricing model is the Trinomial option pricing model. At each timestep the price instead of having 2 possible paths, has 3 possible paths: it can go up, down, or stay at the same price. The values of the newer nodes are found by multiplying the value at the current node by the appropriate factor u, d or m where

$$\begin{aligned} u &= e^{\sigma\sqrt{2\Delta t}} \\ d &= e^{-\sigma\sqrt{2\Delta t}} = \frac{1}{u} \\ m &= 1 \end{aligned}$$

and the corresponding probabilities are:

$$\begin{aligned} p_u &= \left(\frac{e^{(r-q)\Delta t/2} - e^{-\sigma\sqrt{\Delta t/2}}}{e^{\sigma\sqrt{\Delta t/2}} - e^{-\sigma\sqrt{\Delta t/2}}}\right)^2 \\ p_d &= \left(\frac{e^{\sigma\sqrt{\Delta t/2}} - e^{(r-q)\Delta t/2}}{e^{\sigma\sqrt{\Delta t/2}} - e^{-\sigma\sqrt{\Delta t/2}}}\right)^2 \\ p_m &= 1 - (p_u + p_d). \end{aligned}$$

Once the tree prices have been calculated, the option price is found at each using the same process as for the binomial model, by working backward from the final nodes to the present node t_0. The difference is that the option value at each non-final node is determined based on the three - as opposed to two - later nodes and their corresponding probabilities.

1.3.2.4 Heston Model

One of the limitations of the Black-Scholes models is the assumption of constant volatility. A significant modeling step away from the assumption of constant volatility in asset pricing was made by modeling the volatility/variance as a diffusion process. The resulting models are the stochastic volatility (SV) models. The intuition to model volatility as a random variable comes from real financial data which, in most cases, indicates that the volatility of assets is also non-constant. Hence, by modeling the variance as a non-constant process, a model closer to reality can be obtained.

The most popular SV model is the Heston model [3], for which the system of stochastic equations under the risk-neutral measure reads,

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t^S, S_{t_0} = S_0, \tag{1.1}$$

$$dv_t = \kappa(\bar{v} - v_t)dt + \gamma\sqrt{v_t} dW_t^v, v_{t_0} = v_0, \tag{1.2}$$

$$dW_t^s dW_t^v = \rho dt, \tag{1.3}$$

with v_t the instantaneous variance, and W_t^s, W_t^v are two Wiener processes with correlation coefficient ρ. The equation 1.2 models a mean reversion process for variance, with parameters, r the risk-free interest rate, $\bar{v}$ the long term variance, κ the reversion speed, γ is the volatility of the variance, determining the volatility of v_t. There is an additional parameter v_0, the t_0-value of the variance.

Using the martingale approach, the following multidimensional Heston option pricing PDE is obtained,

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \kappa(\bar{v} - v)\frac{\partial V}{\partial v} + \frac{1}{2}vS^2\frac{\partial^2 V}{\partial S^2} + \rho\gamma Sv\frac{\partial^2 V}{\partial S \partial v} + \frac{1}{2}\gamma^2 v\frac{\partial^2 V}{\partial v^2} - rV = 0.$$

The Heston model does not have analytic solutions and is thus solved numerically. The numerical methods that can be used to find an approximated solution to the Heston model are the finite differences (FD), Monte Carlo simulations, and numerical integration methods.

1.3.2.5 Monte Carlo methods for option pricing

Monte Carlo Option Price is a method often used in Mathematical finance to calculate the value of an option with multiple sources of uncertainties and random features, such as changing interest rates, stock prices or exchange rates, etc.. This method is called Monte Carlo simulation, named after the city of Monte Carlo, which is noted for its casinos.

A discrete model for change in the price of a stock over a time interval $[0, T]$ is:

$$S_{n+1} = S_n + \mu S_n \Delta t + \sigma S_n \epsilon_{n+1} \sqrt{\Delta t}$$

where $S_n = S_{t_n}$ is the stock price at time $t_n = n\Delta t, n = 0, 1, ..., N-1$, $\Delta t = T/N, \mu$ is the annual growth rate of the stock, and σ is a measure of the stock annual price volatility or tendency to fluctuate. Each term in the sequence $\epsilon_1, \epsilon_2, ...$ takes on the value of 1 or -1 depending on the outcoming value of a coin tossing experiment, heads or tails respectively. In other words, for each $n = 1, 2, ...$

$$\epsilon_n = \begin{cases} 1 & \text{with probability } = 1/2 \\ -1 & \text{with probability } = 1/2 \end{cases}$$

Monte Carlo method The price of a European option (put or call) can be obtained by running a lot of simulations:

$$\{S_N^{(k)} = S^{(k)}(T), k = 1, ..., M\}$$

of M stock prices at expiration, generated by the equation:

$$S_{n+1}^{(k)} = S_n^{(k)} + r S_n^{(k)} \Delta t + \sigma S_n^{(k)} \epsilon_{n+1}^{(k)} \sqrt{\Delta t}$$

which is identical to the previous equation for each $k = 1, ..., M$, except that the growth rate μ has been replaced by the annual interest r. The option pricing theory requires that the average value of the payoffs $\{f(S_N^{(k)}) \;\; k = 1, ..., M\}$ be equal to the compounded total return obtained by investing the option premium, $C(s)$ (in this case a call), at rate r over the life of option,

$$\frac{1}{M} \sum_{k=1}^{M} f(S_N^{(k)}) = (1 + r\Delta t)^N C(s)$$

Solving the previous equation for $C(s)$ yield the Monte Carlo estimate

$$C(s) = (1 + r\Delta t)^{-N} \{\frac{1}{M} \sum_{k=1}^{M} f(S_N^{(k)})\}$$

for the call option price. So, the Monte Carlo estimate $C(s)$ is the present value of the average of the payoffs computed using rules of compound interest. Using the call-put parity we can find the price of the corresponding put option.

1.4 Greeks

The **Greeks** are the quantities representing the sensitivity of the price of options to a change in the underlying parameters. Greeks are usually computed using the Black-Scholes formula,

despite the fact that the Black-Scholes model is known to be a poor approximator to reality.

1.4.1 Delta

The **Delta**, Δ, is the first-order derivative of the value V of the option with respect to the underlying instrument's price S. It measures the rate of change of the theoretical option value with respect to changes in the underlying asset's price. We obtain that the Delta of a European call option is

$$\Delta = \frac{\partial C}{\partial S} = \Phi(d_1)$$

by using the call-put parity, we obtain that the Delta of a put option is:

$$\Delta = \frac{\partial P}{\partial S} = -\Phi(d_1) = \Phi(d_1) - 1$$

The absolute value of the Delta is often interpreted as the *implied probability* that the option will expire in-the-money (assuming that the market moves under Brownian motion in the risk-neutral measure). For example, if an out-of-the-money call option has a delta of 0.15, the trader might estimate that the option has approximately a 15% chance of expiring in-the-money. at-the-money calls and puts have a delta of approximately 0.5 and -0.5 respectively.

1.4.2 Gamma

The **Gamma**, Γ, is the second derivative of the value function with respect to the underlying price. It measures the rate of change of the Delta with respect to the change of the underlying price. The Gamma for a European options is

$$\Gamma = \frac{\partial \Delta}{\partial S} = \frac{\partial^2 V}{\partial S^2} = \frac{\phi(d_1)}{S\sigma\sqrt{T-t}}$$

1.4.3 Vega

The **Vega**, $\mathcal{V}$, is the first-order derivative of the option value with respect to the volatility of the underlying asset. The Vega measures sensitivity to volatility.

$$\mathcal{V} = \frac{\partial V}{\partial \sigma} = S\phi(d_1)\sqrt{T-t}$$

Vega is typically expressed as the amount of money per underlying share that the option's value will gain or lose as volatility rises or falls by 1 percentage point. All options (both calls and puts) will gain value with rising volatility.

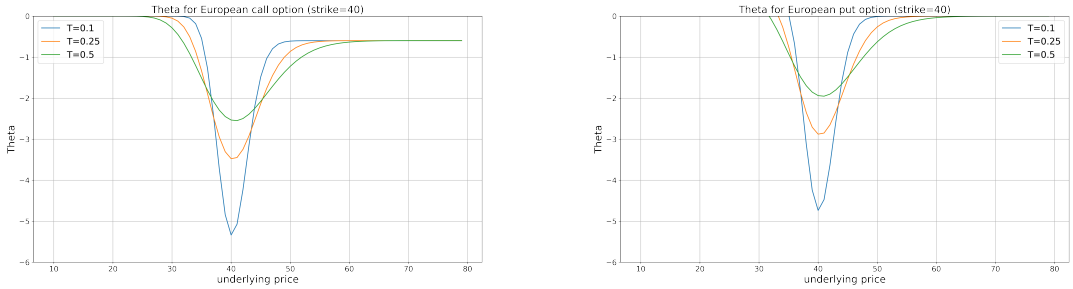
1.4.4 Theta

The **Theta**, θ, is the first-order derivative of the option value with respect to time to maturity. It measures the sensitivity of the option value to the passage of time, it is also called the "time decay". For a European call option the Theta is:

$$\theta = \frac{\partial C}{\partial t} = -\frac{S\phi(d_1)\sigma}{2\sqrt{T-t}} - rKe^{-r(T-t)}\Phi(d_2)$$

While for a European put option the Theta is:

$$\theta = \frac{\partial P}{\partial t} = -\frac{S\phi(d_1)\sigma}{2\sqrt{T-t}} + rKe^{-r(T-t)}\Phi(-d_2)$$



(a) Theta for a European Call Option as a Function of Stock Price

(b) Theta for a European Put Option as a Function of Stock Price

Figure 1.5
Theta for European Options as a function of the underlying

Figure 1.5 shows that theta is almost negative for long calls and puts, and positive for short calls and puts. As the price is approaching the strike price Theta gets bigger in absolute value and diminishes the further out from the price (in absolute value). Theta represents the *time value* of the option, which is the value of having the option of waiting longer before deciding to exercise.

1.4.5 Rho

The **Rho**, ρ, is the derivative of the option value with respect to the risk-free interest rate. It measures sensitivity to the interest rate. For a European call option the Rho is:

$$\rho = \frac{\partial C}{\partial r} = K(T-t)e^{-r(T-t)}\Phi(d_2)$$

While for a European put option the Rho is:

$$\rho = \frac{\partial P}{\partial r} = -K(T-t)e^{-r(T-t)}\Phi(-d_2)$$

Rho is typically expressed as the amount of money, per share of the underlying, that the value of the option will gain or lose as the risk-free interest rate rises or falls by 1.0% per annum (100 basis points).

1.5 Implied Volatility

The **implied volatility** of an option contract is the value of the volatility of the underlying instrument which, when input into an option pricing model, will return a theoretical value equal to the current market price of said option.

One of the most frequent inputs of the majority of option pricing models is an estimate of the future realized price volatility, σ, of the underlying asset, i.e.:

$$V = f(\sigma, \cdot)$$

where V is the theoretical value of an option, f is the pricing model that depends on σ, along with other inputs. Generally speaking, we would expect a higher option price given a higher value of the volatility as input of the pricing model. Hence, $f(\sigma, \cdot)$ is monotonically increasing in σ, so there is a unique value of σ for each option value V.

Assuming that there is some inverse function $g = f^{-1}$ for the given pricing model, such that:

$$\sigma_{\hat{V}} = g(\hat{V}, \cdot)$$

where $\hat{V}$ is the market price for an option. The value $\sigma_{\hat{V}}$ is the volatility **implied** by the market price $\hat{V}$, or the **implied volatility**.

Due to the structure of the Black-Scholes formula, the implied volatility cannot be found in closed form but only through numerical approximation methods. One of the most used numerical method used to find the implied volatility is the **Newton-Raphson**.

The Newton-Raphson method is a root-finding algorithm which produces successively better approximations to the roots of a real-valued function. Let f bet a single-variable function defined for a real variable x, let f' be the function's derivative, and let x_0 be an initial guess for a root of f. If the function satisfies sufficient assumptions and the initial guess is close, then

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

is a better approximation of the root than x_0. The process is repeated as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

In the case of implied volatility the function f is defined as:

$$f(\sigma) = C - g(\sigma)$$
$$g(\sigma) = S\Phi(d_1) - X\Phi(d_2)$$

where C is value of the call (the same reasoning can be applied to put options). As shown

in [4] if the starting point is

$$\sigma_f \coloneqq \sqrt{\frac{2|\log(S/K)|}{T}}$$

also called the inflection volatility point, allows to find the implied volatility of the market with great precision and in real time.

Chapter 2

Neural Networks

Inspired by the biological neural networks, an **Artificial Neural Network** (**ANN**) is a mathematical model which constitutes the foundation of Deep Learning. The goal of an ANN is to approximate some function f^*.

The most simple Artificial Neural Networks are **Deep feedforward networks**, also often called **feedforward neural networks**, or **multilayer perceptrons** (MLPs). A feedforward network defines a mapping $y = f(x;\theta)$ and learns the value of the parameters theta that result in the best function approximation. These models are called **feedforward** because information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model can be also seen as a direct acyclic graph describing how the function are composed together.

For instance, we might have three functions $f^{(1)}, f^{(2)}$ and $f^{(3)}$ connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. In this case $f^{(1)}$ is called the **first layer** of the network, $f^{(2)}$ is called the second layer, and so on. The overall length of the chain gives the **depth** of the model. It is from this terminology that the name "deep learning" arises. The final layer of a feedforward network is called the **output layer**. Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the **width** of the model. Rather than thinking of a layer as a vector-to-vector function, we can decompose it into many units that act in parallel, each representing a vector-to-scalar function. Mathematically speaking each unit computes a weighted sum of the inputs:

$$h_j^{(l)} = \phi^{(l)}(\sum_k w_{jk}^{(l)} h_k^{(l-1)} + b_j^{(l)})$$

where $h_j^{(l)}$ is the activation of the j^{th} neuron in the l^{th} layer, $w_{jk}^{(l)}$ is the weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer and $b_j^{(l)}$ is the bias of the j^{th} neuron in the l^{th} layer. $\phi^{(l)}$ is the activation function used in the l^{th} layer, used to capture nonlinearities in the data. All weights can be grouped in a matrix and all biases in a vector, thus obtaining a more compact notation for the activation of each layer as in equation (2.1).

$$\boldsymbol{h}^{(l)} = \phi^{(l)}\left(\boldsymbol{W}^{(l)\top}\boldsymbol{h}^{(l-1)} + \boldsymbol{b}^{(l)}\right) \tag{2.1}$$

$$\boldsymbol{h}^{(l)} = \phi^{(l)}(\boldsymbol{a}^{(l)}) \tag{2.2}$$

where $\boldsymbol{W}^{(l)}$ is the weight matrix of the l^{th} layer, $\boldsymbol{b}^{(l)}$ is the bias vector of the l^{th} layer and $\phi^{(l)}$ is the activation function of the l^{th} layer. The activation of first layer is given by:

$$\boldsymbol{h}^{(1)} = \phi^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

where $\boldsymbol{x}$ is the input of the network.

2.1 Architecture

The term architecture usually refers to the overall structure of the network: this includes the number of layers, the width of those layers, the activation function used in each layer, and so on.

2.1.1 Activation function

One of the most important choices when dealing with the architecture of a neural network is the activation function. Each layer can have the same activation functions or they can have different activation function depending on the goal that the researcher wants to achieve with the neural network.

One of the most common choices is the **ReLU** (Rectified linear units) [5] activation function defined as follows:

$$ReLU(x) = \max\{0, x\} \tag{2.3}$$

Rectified linear units are easy to optimize because they are similar to linear units, the only difference is that when the input is negative the ReLU function deactivates those neurons.

One of the drawbacks of rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero. A variety of rectified linear unit generalizations have been proposed to guarantee that they receive gradients everywhere.

Those generalizations are based on using a non-zero slope α_i when $x_i < 0 : h_i = \phi(\boldsymbol{x}, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, x_i)$. For example, the **LeakyReLU** [6] fixes α_i to a small value like 0.01 while a **parametric ReLU** [7] threats α_i as a learnable parameter. Other types of generalization tend to smooth the ReLU, making it a differentiable function. One of those examples is the **ELU** activation function defined as

$$ELU(\boldsymbol{x}) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

2.1.2 Universal Approximation Properties

Linear models can be trained way more easily than neural networks, but they can only represent linear functions. Unfortunately, we often want to represent non-linear functions.

At first sight, we might presume to make an ad hoc model that the kind of non-linearity that we want to approximate. Fortunately, feedforward neural networks with hidden layers provide a universal approximation framework. This is because the **universal approximation theorem** states that a feedforward neural network with linear output and at least one hidden layer with any non-linear activation function can approximate any kind of continuous function on a closed and bounded subset of $\mathbb{R}^n$ (Borel measurable function).

Despite the original theorems being formulated in terms of hidden layers with a sigmoid activation function, which has the downside of saturating for very negative and very positive values, the universal approximation theorem has also been proved for a wider class of activation functions, which include also the rectified linear unit [8].

The universal approximation theorem has also been proved in the case where the network has bounded width and arbitrary depth. It has been shown by [9] that a network of width $n + 4$ with ReLU activation function can approximate any Lebesgue-integrable function on n-dimensional input space.

Width and depth are both important and should be carefully tuned together for the best performance and the expressiveness of the neural networks, since depth may determine the abstraction level but the width may influence the loss of information in the forward pass. [9] showed empirically that depth may be more effective than width for the expressiveness of ReLU networks.

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large feedforward neural network will be able to represent this function. However, we are not guaranteed that the training algorithm will be able to learn that function.

2.2 Training

Training a neural network is an optimization problem with respect to its set of weights, which can in principle be addressed by using any kind of optimization method. The de facto optimization algorithm used in deep learning is gradient descent.

2.2.1 Gradient descent

Gradient descent is a first-order optimization algorithm for finding a local minimum of a differentiable function. To find a local minimum of a function using gradient descent, we take steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

Given an unconstrained optimization problem in a n-dimensional space

$$\min f(x), \quad x \in \mathbb{R}^n$$

the gradient descent technique is based on the observation that, if $f(x)$ is defined and differentiable in a neighborhood of a point x then $f(x)$ decreases fastest if one goes from x in the direction of the negative gradient of f in x, $\nabla f(x)$. It follow that the next point z, will be defined as

$$z = x - \eta \nabla f(x)$$

for a η in $\mathbb{R}_+$ small enough (called learning rate), then $F(x) \geq F(z)$. The term $\eta \nabla f(x)$ is subtracted from x because we want to move against the gradient, toward the local minimum. With that in mind, considering an initial guess x_0 for a local minimum of f, using gradient descent a sequence $x_0, x_1, x_2, \cdots$ is obtained, such that

$$x_{t+1} = x_t - \eta \nabla f(x), \quad t > 0$$

and

$$f(x_0) \leq f(x_1) \leq f(x_2) \leq \cdots$$

is a monotonic sequence.

Given a dataset $(\boldsymbol{x}, \boldsymbol{y})$ of n samples the neural network training can be expressed as the following optimization problem:

$$\min_{\theta} \; J(\theta) = \min_{\theta} \; L(f(\boldsymbol{x}; \theta), \boldsymbol{y}) \tag{2.4}$$

where $J(\theta) = L(f(\boldsymbol{x}; \theta), \boldsymbol{y})$ is some cost function that has to be minimize and θ are the neural network parameters. Using gradient descent to solve 2.4, at each iteration the neural network's weights are updated in the following way:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \, L(f(x_i; \theta_t), y_i)$$

In practice, what is used to optimize neural network are variants of stochastic gradient descent.

Stochastic gradient descent The stochastic gradient descent (SGD) algorithm is a drastic simplification of gradient descent. Instead of computing the gradient of $f(x)$ exactly, each iteration estimates this gradient on the basis of a single randomly picked example (x_i, y_i):

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \, L(f(x_i; \theta_t), y_i)$$

Generally, each parameter update in SGD is computed with respect to a few training examples or a minibatch as opposed to a single example. The reason for this is twofold: first,

this reduces the variance in the parameter update and can lead to more stable convergence, second this allows the computation to take advantage of highly optimized matrix operations that should be used in a well-vectorized computation of the cost and gradient.

The SDG pseudocode is shown in Algorithm 1.

A crucial parameter for the SGD algorithm is the learning rate. Previously, the SGD has been described using a fixed learning rate η. In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration k as η_k.

As a matter of fact, newer Deep learning optimization algorithms such as Adam [10], AdaGrad [11], and RMSProp [12] are all algorithms with an adaptive learning rate.

Even though SGD can converge to a good solution in most cases, the algorithm has no such convergence guarantee and is sensitive to the values of the initial parameters.

Algorithm 1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate η_k
Require: Network parameters θ
while stopping criterion not met **do**
    Sample a minibatch of m examples from the trainig set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
    Compute gradient estimate $\hat{g} \leftarrow +\frac{1}{m}\nabla_\theta \sum_i L(f(\boldsymbol{x}^{(i)}; \theta), \boldsymbol{y}^{(i)})$
    Apply update: $\theta \leftarrow \theta - \eta\hat{g}$

It is clear that to optimize the neural network the gradient needs to be computed. The only way to get it is through the **back-propagation**.

2.2.2 Backpropagation

The back-propagation algorithm [13] (also called **backprop**), allows the information from the cost to flow backward through the network to compute the gradient.

Back-propagation is often misunderstood as being specific to a multi-layer neural network, but in principle, it can compute derivatives of any function represented by a **computational graph**.

A computation graph is a direct acyclic graph where each node in the graph indicates a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type. To formalize our graphs, we also need to introduce the idea of an operation. An operation is a simple function of one or more variables. Our graph language is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together. If a variable y is computed by applying an operation to a variable x, then we draw a direct edge from x to y.

Figure 2.1 show the computational graph which represents the computation $y = f(x_1, x_2) = x_1^2 + x_1 x_2 - \cos(x_2)$.

As shown in [14] back-propagation is just a particular case of the reverse accumulation mode of Automatic Differentiation (AD). When training a neural network we would like to

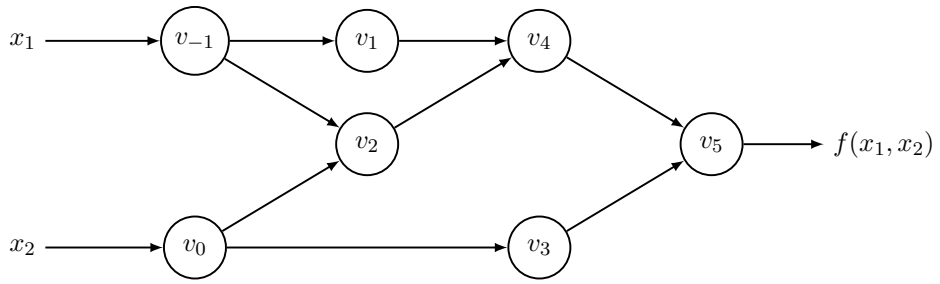


Figure 2.1
Computational graph of the example $f(x_1, x_2) = x_1^2 + x_1 x_2 - \cos(x_2)$. v_0 and v_{-1} are the input variables, v_1, v_2, v_3, v_4 are the intermediate variables and v_5 is the output variable.

calculate for each weight of each layer the derivative such that the newly updated weight minimized the cost function, hence

$$\bar{w}_{jk}^{(l)} = \frac{\partial J(\theta)}{\partial w_{jk}^{(l)}}$$

each of those partial derivatives represent the sensitivity of cost function with respect to changes in $w_{jk}^{(l)}$.

In the reverse mode of AD, derivatives are computed in the second phase of a two-phase process. In the first phase, the original function code is run *forward*, populating intermediate variables and recording the dependencies in the computational graph through a book-keeping procedure. In the second phase, derivatives are calculated by propagating the partial derivatives in *reverse*, from the outputs to the inputs.

Returning to the example $y = f(x_1, x_2) = x_1^2 + x_1 x_2 - \cos(x_2)$, Table 2.1 shows how the reverse mode works. For example, if we want to obtain the partial derivative of v_0 with respect to y for the chain rule we got that:

$$\frac{\partial y}{\partial v_0} = \frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_0} + \frac{\partial y}{\partial v_3}\frac{\partial v_3}{\partial v_0}$$

A neural network is a particular type of computational graph that consists of the combination of function composition and matrix multiplication:

$$f(\boldsymbol{x}) = \phi^{(L)}(\boldsymbol{W}^{(L)}\phi^{(L-1)}(\boldsymbol{W}^{(L-1)} \cdots \phi^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \cdots))$$

Algorithm 2 shows the reverse mode of AD works into a neural network.

An important advantage of back-propagation is that to compute the full gradient ∇f only one application of the reverse mode is sufficient. On the downside, this advantage come with the cost of increased storage requirements growing in proportion of the number of operations in the evaluated function.

Algorithm 2 Backward computation for deep neural networks which use in addition to the input x and a target y. This computation yields the gradients on the activations $a^{(k)}$ for each layer k, starting from the output layer and going backward to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

1: After the forward computation, compute the gradient on the output layer
2: $\boldsymbol{g} \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
3: **for** $k = l, l-1, ..., 1$ **do**
4: Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation:
5: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}_{(k)}} J = \boldsymbol{g} \odot \phi^{'}(\boldsymbol{a}^{(k)})$
6: Compute the gradients on weights and biases:
7: $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$
8: $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top}$
9: Propagate the gradients with respect to the next lower hidden layer's activations:
10: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

Table 2.1
Reverse mode AD example, with $y = f(x_1, x_2) = x_1^2 + x_1 x_2 - \cos(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse (cf. Figure 2.1). Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$.

Forward Primal Trace			Reverse Adjoint (Derivative) Trace		
$v_{-1} = x_1$		$= 2$	$\bar{\boldsymbol{x}}_{\mathbf{1}} = \bar{\boldsymbol{v}}_{\mathbf{-1}}$		$\mathbf{= 9}$
$v_0 = x_2$		$= 5$	$\bar{\boldsymbol{x}}_{\mathbf{2}} = \bar{\boldsymbol{v}}_{\mathbf{0}}$		$\mathbf{= 1.041}$
$v_1 = v_{-1}^2$		$= 2^2$	$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 \times 2v_{-1}$	$= 9$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$	$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.041$
			$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$v_3 = \cos v_0$		$= \cos 5$	$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times (-\sin v_0)$	$= -0.959$
$v_4 = v_1 + v_2$		$= 4 + 10$	$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
			$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$v_5 = v_4 - v_3$		$= 14 - 0.284$	$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
			$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$y = v_5$		$= 13.716$	$\bar{v}_5 = \bar{y}$	$= 1$	

Chapter 3

Datasets

To prove the approximation power of neural networks different datasets, based on different pricing models, were generated. The pricing models chosen are the Binomial option pricing model, the trinomial pricing model, and the Heston pricing model. The first two models follow the same mechanics and represent a discrete version of the Black-Scholes models. While the Heston model does have more parameters to deal with, it has the advantage to assume that also the volatility follows a stochastic process.

3.1 Binomial-Trinomial dataset generation

Both the binomial dataset and trinomial dataset are being generated using the same underlying process since they have the same features: the underlying price S, the strike price K, the volatility σ, the interest free-rate r, the time to expiration τ, and the type of option (call or put).

The main idea for generating the two datasets is the concept of an option chain. An option chain is a list of all available option contracts for given security. So for a given value of underlying price, volatility, interest-free rate, and time to expiration all the call (put) options within a level of moneyness of 0.66 and 1.5 are priced using the binomial pricing model or the trinomial pricing model.

Table 3.1 lists all the range of the input parameters.

Table 3.1
Binomial-Trinomial dataset parameter ranges

	Parameters	**Range**	**Unit**
Input	Moneyness	[0.66, 1.5]	-
	Time to Maturity, τ	[0.1, 1.0]	year
	Risk free rate, r	[0.01, 0.1]	-
	Volatility, σ	[0.05, 1.0]	-

With regards to parameter sampling, for each input parameter, the list of parameters is generated equally spacing from the smallest value of the range to the bigger one using the given step size.

3.2 Heston dataset generation

The generation of the Heston dataset uses the same concept of the option chains but has a different sampling procedure. Here the number of parameters is bigger than the binomial and trinomial case, so using a grid approach results to be time-consuming.

The sampling technique used is the **Latin hypercube sampling (LHS)**, which is able to generate random samples of the parameter value from a multinomial distribution. Table 3.2 shows the parameter ranges in the case of the Heston dataset.

The approach used to find the option prices, in this case, was the Least Square Monte Carlo approach [15].

Table 3.2
The Heston parameter ranges for traning the ANN

	Parameters	**Range**	**Unit**
	Moneyness, $m = S_0/K$	[0.5, 2]	-
	Time to maturity, τ	[0.1, 1.1]	year
	Risk free rate, r	[0.01, 0.10]	-
	Correlation, ρ	[-0.9, 0.0]	-
Input	Reversion speed, κ	[0.0, 2.0]	-
	Long average variance, $\bar{\nu}$	[0.0, 0.5]	-
	Volatility of volatility, γ	[0.0, 0.5]	-
	Initial variance, ν_0	[0.05, 0.5]	-

3.3 Real data dataset

In addition to the generation of synthetic data based on well-known pricing models, real market daily data about American stock options was gathered and stored in a dataset. The stocks subject of this work were the tech stocks whose options have the most open interest and volume: TSLA, FB, AMZN, AMD, NVDA, AAPL, NFLX, and MSFT. No type of limit has been imposed regarding the expiration of these options, all the option chains of the available expiration date were downloaded.

The dataset is composed of two months of historical options data of the selected stocks reaching a size of more than 600,000 samples. In the case of the interest rate, the yield of the 3-month American T-bill was used.

Chapter 4

Hyperparameter Optimization

Training neural networks involve numerous choices for parameters whose value is used to control the training process. These parameters are often referred to as *hyperparameters*. They can be involved in building the structure of the model, such as the number of hidden layers, the number of layers, and the activation function, or in determining the efficiency and accuracy of model training, such as the learning rate, the optimizer, and the batch size.

Since the training process usually is a computationally demanding process, it is not practical for a human to try out all the combinations of the hyper-parameters to find the best model. To remove humans from the loop Hyperparameter Optimization (HPO) techniques have been introduced to automatically optimize the hyper-parameters of a machine learning model. As a trade of human efforts, HPO demands a large number of computational resources, especially when several hyperparameters are optimized together.

There are different techniques to implement the automatic search. One of the basic algorithms is *Grid Search* which consists of performing an exhaustive search on the hyper-parameter set specified by the user. Despite its simplicity, grid search is only feasible when the hyperparameter set is fairly small, as the number hyper-parameters gets bigger and bigger the algorithms suffer from the curse of dimensionality because the number of possible combinations grows exponentially. In practice, grid search is preferable to use when users have enough experience of these hyper-parameters to enable the definition of a narrow search space and no more than three hyper-parameters need to be tuned simultaneously.

An improvement of grid search is *Random Search* [16]. This kind of algorithm makes a randomized search over certain distributions over possible parameter values. The searching process continues till the predetermined budget is exhausted, or until the desired accuracy is reached. Even though in most cases random search is more effective than grid search, it is still a computationally intensive method.

Both grid search and random search don't use the results obtained in previous trials to "guess" what could be the next set of hyper-parameters, which means that these methods waste tons of computational resources on trials that are going to result in "bad choices".

Recently newer methods that are leveraging the knowledge of previous trials, have been

developed to efficiently reduce the computational cost by navigating through the hyperparameters space. A class of those algorithms is based on *Bayesian optimization* [17].

4.1 Bayesian Optimization

Bayesian Optimization is a class of machine-learning-based optimization method focused on solving the problem

$$\max_{x \in A} f(x) \tag{4.1}$$

where the feasible set and objective function typically have the following properties:

- the input x is in $\mathbb{R}^d$ for a value of d that is not too large. Typically $d \leq 20$ is the most successful application of BayesOpt.
- The feasible set A is a simple set, in which it is easy to assess membership.
- The objective function f is continuous.
- f is "expensive to evaluate" in the sense that each evaluation takes a substantial amount of time, or monetary cost (or any kind of cost)
- f is a "black box" which means that f lacks a known special structure like concavity or linearity that would make it easy to optimize.

BayesOpt consists of two main components: a **Bayesian statistical model** for modeling the objective function, and the **acquisition function** for deciding where to sample next. After evaluating the objective according to an initial space-filling experimental design, they are used iteratively to allocate the remainder of the budget for N function evaluation as shown in Algorithm 3.

Algorithm 3 Pseudo-code for Bayesian Optimization

1: Place a Gaussian process prior on f
2: Observe f at n_0 points according to an initial space-filling experimental design
3: $n = n_0$
4: **while** $n \leq N$ **do**
5: Update the posterior probability distribution on f using all available data
6: Let x_n be a maximizer of the acquisition function over x, where the acquisition function is computed using the current posterior distribution
7: Observe $y_n = f(x_n)$
8: $n++$
9: **return** either the point evaluated with the largest $f(x)$, or the point with the largest posterior mean.

The statistical model, which is invariably a **Gaussian Process**, provides a Bayesian posterior probability distribution that describes potential values for $f(x)$ at a candidate point x.

The role of the acquisition function is to guide the search for the optimum. Typically, acquisition functions are defined such that high acquisition corresponds to potentially high values of the objective function, whether because the prediction is high, the uncertainty is great, or both. Maximizing the acquisition function is used to select the next point at which to evaluate the function.

4.1.1 Gaussian Process

A Gaussian process can be seen as a generalization of multivariate Gaussian distribution to infinitely many variables. It is formally defined as:

Definition 1 (Gaussian Process) *A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution.*

A Gaussian process is specified by its mean function and covariate function. The mean function $m(\mathbf{x})$ and the covariance function $k(\mathbf{x}, \mathbf{x}^{'})$ of a real process $f(x)$ are defined as

$$
\begin{aligned}
m(\mathbf{x}) &= \mathbb{E}[f(\mathbf{x})], \\
k(\mathbf{x}, \mathbf{x}^{'}) &= \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}^{'}) - m(\mathbf{x}^{'}))]
\end{aligned}
$$

In other words, a Gaussian process is a random process where any point $\mathbf{x} \in \mathbb{R}^d$ is assigned a random variable $f(\mathbf{x})$ and where the joint distribution of a finte number of these variables $p(f(\mathbf{x_1}), ..., f(\mathbf{x_N}))$ is itself Gaussian:

$$p(\mathbf{f}|X) = \mathcal{N}(\mathbf{f}|\mu, K)$$

where $\mathbf{f} = (f(\mathbf{x_1}), ..., f(\mathbf{x_N})), \mu = (m(\mathbf{x_1}), ..., m(\mathbf{x_N}))$ and $K_{ij} = k(x_i, x_j)$. Usually is common to use $m(\mathbf{x}) = 0$ as GPs are flexible enough to model the mean arbitrary well. The kernel is chosen so that points x_i, x_j that are closer in the input space have a large positive correlation, encoding the belief that they should have more similar function values than points that are far apart.

Given a training dataset with noise-free function values $\mathbf{f}$ at inputs X, a GP prior can be converted into a GP posterior $p(\mathbf{f}_*|X_*, X, \mathbf{f})$ which can then be used to make predictions $\mathbf{f}_*$ at new inputs X_*. By definition of a GP, the joint distribution of observed values $\mathbf{f}$ and predictions $\mathbf{f}_*$ is again Gaussian:

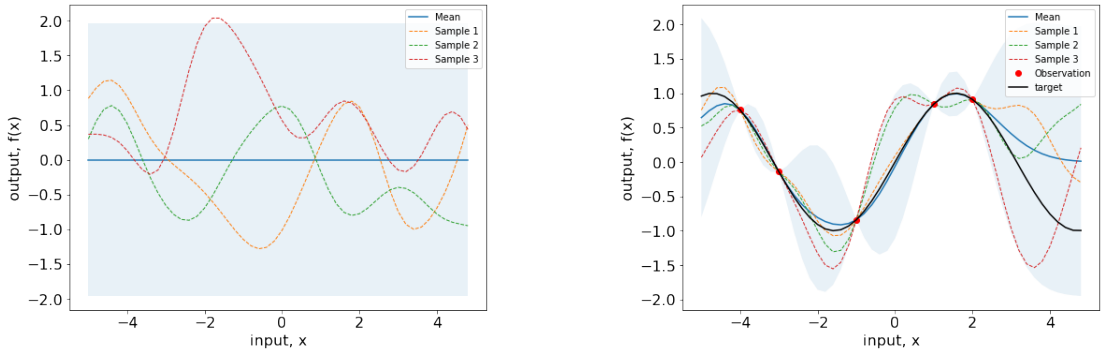
$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left[\mathbf{0}, \begin{pmatrix} K & K_* \\ K_*^\top & K_{**} \end{pmatrix}\right]$$

where $K_* = k(X, X_*)$ and $K_{**} = k(X_*, X_*)$. Using stanrdard rules for conditioning Gaussians, the predictive distribution is given by:

$$p(\mathbf{f}_*|X_*, X, \mathbf{f}) = \mathcal{N}(\mathbf{f}_*|\mu_*, \Sigma_*)$$
$$\mu_* = K_*^\top K^{-1}\mathbf{f}$$
$$\Sigma_* = K_{**} - K_*^\top K^{-1} K_*$$

where μ_* consist in a weighted average between the prior and an estimate based on the data, while Σ_* is equal to the prior covariance less a term that corresponds to the variance remove by the observed data.

Figure 4.1 shows how the inference of a Gaussian process works. Figure 4.1a depicts the GP prior distribution, while Figure 4.1b depicts the GP posterior distribution when 5 noise-free observations are made.



(a) prior (b) posterior

Figure 4.1
Panel (a) shows three functions drawn at random from a GP prior; the blue line represent the prior mean of the GP; the three other functions have been drawn as lines by joining a large number of evaluated points. Panel (b) shows three random functions drawn from the posterior, i.e. the prior conditioned on the five noise free observations indicated. In both plots the shaded area represents the pointwise mean plus and minus two times the standard deviation for each input value (corresponding to the 95% confidence region), for the prior and posterior respectively

If the training dataset contains noisy evaluation $\mathbf{y} = \mathbf{f} + \epsilon$ where noise $\epsilon \sim \mathcal{N}(0, \sigma_y^2\mathbf{I})$ is independently added to each observation then the predictive distribution is given by

$$p(\mathbf{f}_*|X_*, X, \mathbf{y}) = \mathcal{N}(\mathbf{y}_*|\mu_*, \Sigma_* + \sigma_y^2\mathbf{I})$$
$$\mu_* = K_*^\top K_y^{-1}\mathbf{y}$$
$$\Sigma_* = K_{**} - K_*^\top K_y^{-1} K_*$$

where $K_y = K + \sigma_y^2\mathbf{I}$.

4.1.2 Acquisition Functions

The acquisition function measures the value that would be generated by evaluation of the objective function at a new point x, based on the current posterior distribution over f. One of the most used acquisition functions is the **expected improvement** (EI). Assuming that we observe f without noise, the optimal choice is the previously evaluated point with the largest observed value. Let $f_n^* = \max_{m \leq n} f(x_m)$ be the value of this point, where n is the number of times we have evaluated f thus far. If we evaluate at x, we will observe $f(x)$. After this new evaluation, the value of the best observed will either be $f(x)$ or f_n^*. The improvement in the value of the best observed point is then $f(x) - f_n^*$ if the quantity is positive, and 0 otherwise.

While we would like to choose x so that this improvement is large, $f(x)$ is unknown until after the evaluation. What we can do, however, is to take the expected value of this improvement and choose x to maximize it. We define the expected improvement as,

$$EI(x) = \mathbb{E}[\max(f(x) - f_n^*, 0)]$$

If f is a gaussian process the expected improvement can be evaluated in closed form using integration by parts, as described in [18], resulting in

$$EI(x) = [\Delta(x)]^+ + \sigma(x)\phi(\frac{\Delta(x)}{\sigma(x)}) - |\Delta(x)|\Phi(\frac{\Delta(x)}{\sigma(x)})$$

where $\mu(x)$ and $\sigma(x)$ are the mean and the standard deviation of the Gaussian process posterior predictive at x, respectively. $\Delta(x) = \mu(x) - f_n^*$ is the expected difference in quality between the proposed point x and the previous best. $\Phi(\cdot)$ and $\phi(\cdot)$ are the standard normal density and distribution function.

Figure 4.2 shows the contours of $EI(x)$ in terms of $\Delta(x)$ and the posterior standard deviation $\sigma(x)$. $EI(x)$ is increasing in both $\Delta(x)$ and $\sigma(x)$. Curves of $\Delta(x)$ versus $\sigma(x)$ with equal EI define an implicit tradeoff between evaluating at points with high expected quality (high $\Delta(x)$) versus high uncertainty (high $\sigma(x)$).

In an optimization problem, evaluating points with high $\Delta(x)$ is valuable because good approximate global optima are likely to reside at such points. However, evaluating points with high uncertainty is also valuable because it helps the objective to explore locations where we have little knowledge.

Choosing where to evaluate based on the tradeoff between high expected performance and high uncertainty appears also in other domains such as multi-armed bandits and reinforcement learning [19], and is often called the "exploration vs. exploitation tradeoff" [20].

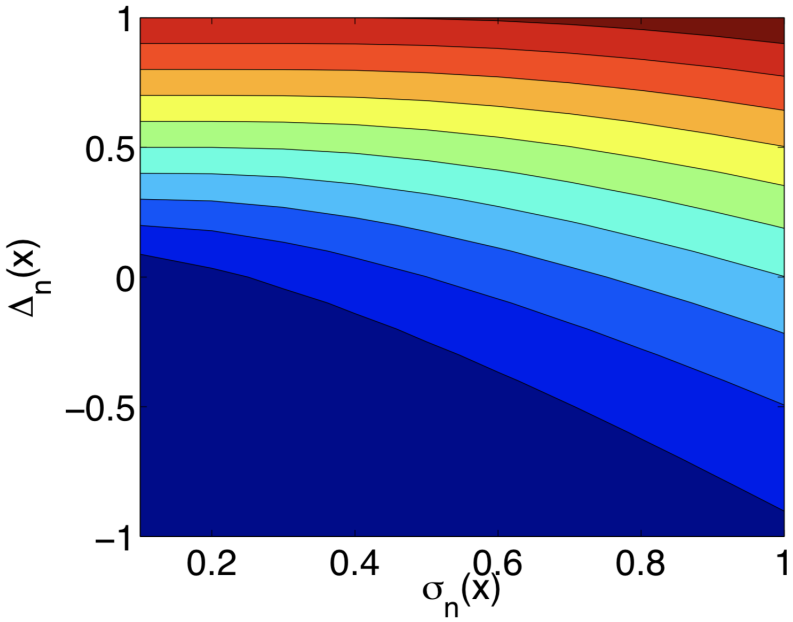


Figure 4.2
Contour plot of EI(x), in terms of $\Delta(x)$ and the posterior standard deviation $\sigma(x)$. Blue indicated smaller values and red higher ones.

Chapter 5

Numerical Results

The model architecture decided to use is a fully connected neural network with a residual connection every two layers. Residual connections have been added because they help to preserve the magnitude of the gradient across layers [21].

Adam [10] was chosen as the neural network optimizer as it turns out to be an optimization algorithm that has proven to be robust with a large range of architectures and problems.

5.1 Bayesian Optimization results

The search space of Bayesian Optimization is described in Table 5.1. The type parameters of the search space are both architecture-related (such as the number of layers of the network) and training-related (such as the learning rate).

The total number of trials of the optimization is 50. The strategy used is the following: in the first 10 iterations, the models generated from a Sobol sequence [22] of 10 samples are fitted to the training dataset, after this initialization phase the Bayesian optimization starts. The objective to minimize is the **Mean Squared Error**, defined as

$$MSE = \frac{1}{n}\sum(y_i - \hat{y}_i)^2$$

and the acquisition function used is the Expected Improvement.

Since in the optimization phase neural networks may not necessarily converge to a global minimum (that would require a lot of time), each model is trained for 50 epochs on a fraction of the total data.

5.1.1 Result on the Binomial-Trinomial dataset

The similarity of the mechanics and the type of the input parameters of the binomial and trinomial models, makes them the candidates to be approximated by a single model. This model is referred to as **LATTICE-ANN**.

The parameters of the best **LATTICE-ANN** model obtained from the BO are shown in Table 5.2

Table 5.1
Parameters Ranges

Parameters	**Options or Range**
Hidden size	4, 6, 8
Activation	ReLU, LeakyReLU, ELU
Neurons	400, 600, 800
Batch size	[512, 2048]
Learning rate	[1e-5, 0.1]

Table 5.2
Best parameters LATTICE-ANN

Parameters	**Options**
Hidden size	6
Activation	LeakyReLU
Neurons	600
Batch size	1393
Learning Rate	0.000092

The best model correspond to the 28-th iteration of the bayesian optimization, as shown in Figure 5.1. After this iteration the BO can't find any better configuration.

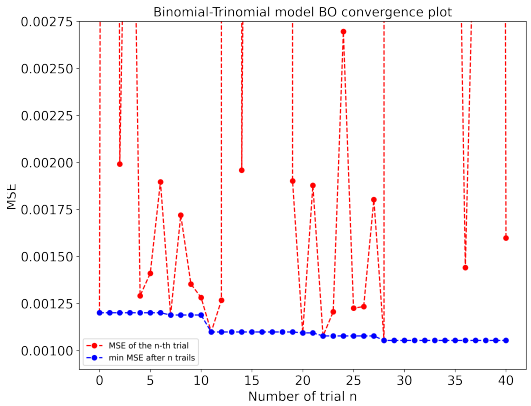
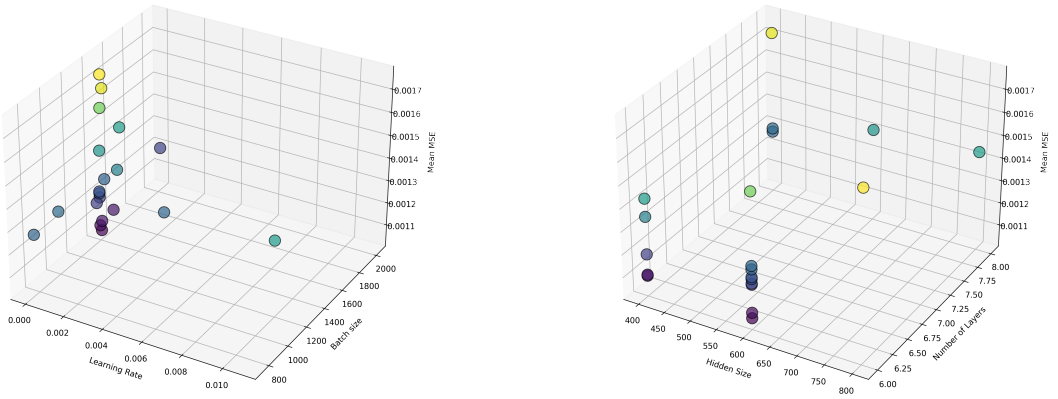


Figure 5.1
Convergence plot of the Binomial-Trinomial model Bayesian Optimization

This result is reasonable because Figure 5.2a shows that, in the top 20 configuration, the most sampled area is the area where the learning rate is closer to 0.0001 and batch size is between 1400 and 1600.

Regarding the hyperparameters which are related to the structure of the neural network from Figure 5.2b, the configuration with the lowest mean squared error is contended by the architecture with 6 hidden layers of 600 neurons each and the architecture with 4 hidden layers of 400 neurons each. Because the architectures with hidden size of 600 have been sampled the



(a) Mean MSE as the learning rate and batch size vary

(b) Mean MSE as the hidden dimension and number of layers vary

Figure 5.2
Top 20 configuration

most by the BO, that means the results obtained with this architecture are more robust than the results obtained with the alternative architecture.

Being the most common activation function in the top 20 configuration (as shown in Figure 5.3), we can be even more certain that the optimal activation function for the **LATTICE-ANN** is the *LeakyReLU* activation function.

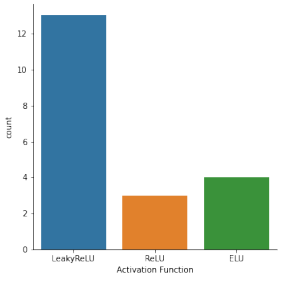


Figure 5.3
Activation function in the top 20 configurations

5.1.2 Result on the Heston dataset

Intuitively the Heston data may seem more difficult to price than the data generated from a model where the volatility is fixed, in fact as shown in Table 5.3 the number of parameters of this model is bigger than the **LATTICE-ANN**. This model is referred to as **HESTON-ANN**.

Table 5.3
Best parameters HESTON-ANN

Parameters	**Options**
Hidden size	4
Activation	LeakyReLU
Neurons	800
Batch size	1031
Learning Rate	0.000059

In this case the BO finds the best configuration at the 11-th iteration as shows in Figure 5.4, even though the BO get close to the best configuration in the subsequent iteration without improvining it.

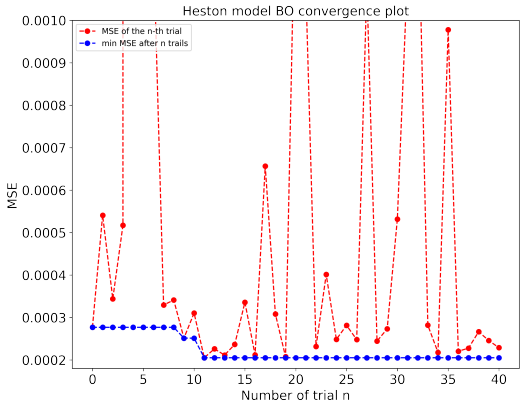
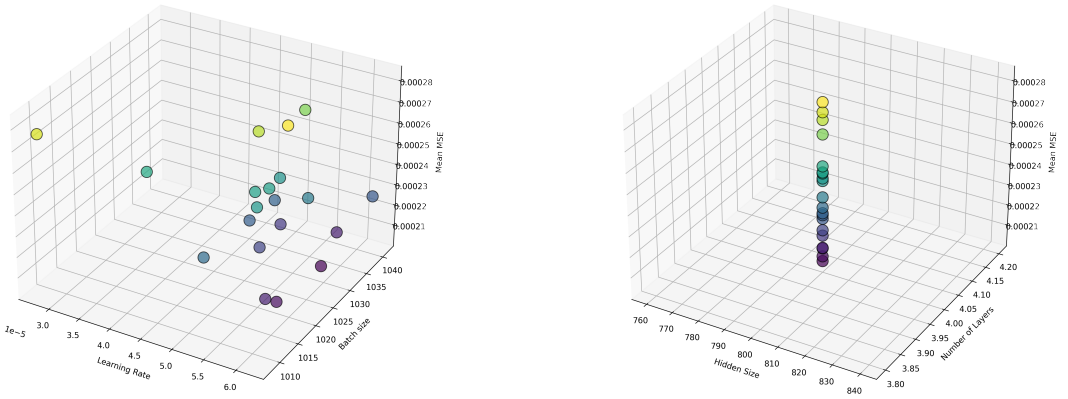


Figure 5.4
Convergence plot of the Heston model Bayesian Optimization

By looking at Figure 5.5a is it clear that learning rates closer to 0.00006 and batch sizes between 1010 and 1040 results in a lower MSE.

Figure 5.5a shows that all the top 20 configurations have the same architecture with 4 hidden layers with 800 neurons each. That means that the optimal model has been found only varying the learning rate and the batch size.



(a) Mean MSE as the learning rate and batch size vary

(b) Mean MSE as the hidden dimension and number of layers vary

Figure 5.5
Top 20 configuration of the BO of the Heston-ANN

5.1.3 Results on real dataset

The results obtained on real data, shown in Table 5.4, are almost the same as those obtained with the Heston dataset. The reason may be that the Heston model approximates well real financial time series since the variance is non-constant. This model is referred to as **REAL-ANN**.

Table 5.4
Best parameters on real dataset

Parameters	**Options**
Hidden size	4
Activation	LeakyReLU
Neurons	800
Batch size	774
Learning Rate	0.00006

The convergence diagram shown in Figure 5.6, in this case, goes down quite smoothly: from the 8th iteration to the 16th iteration we often get models that improve the MSE, from 16 to 25 the error metric does not improve, as probably the BO algorithm will have explored other possible configurations to then reach the 26th iteration where the best hyperparameters are obtained.

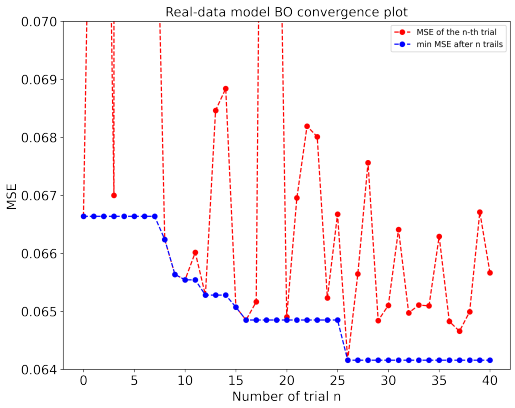


Figure 5.6
Convergence plot of the REAL-ANN Bayesian Optimization

5.2 Training results

All three models have been trained using 80% of the total data (after suitable shuffling), while for the remaining 20% it was divided equally between the validation set and the test set. Both the training error and the validation error where recorder during the training phase.

To avoid over-fitting an early stopping technique was used. This early stopping technique consist in stopping the training phases if the validation error does not improve for 20 consecutive epochs. Both the training error and the validation error were recorder during the training phase.

To measure the model precision, apart from using the mean squared error, other three metrics were used:

$$RSME = \sqrt{MSE}$$
$$MAE = \frac{1}{n}\sum |y_i - \hat{y}_i|$$
$$MAPE = \frac{1}{n}\sum \frac{|y_i - \hat{y}_i|}{y_i}$$

the MSE is used as the trainig metric to update the weights, and all above metrics are employed to evaluate the selected ANN.

5.2.1 LATTICE-ANN Results

The final **LATTICE-ANN** model has been trained on 80% of the whole binomial-trinomial dataset, which contains more than 3,200,000 samples. From figure 5.7 show that the model training has last for 275 epochs, ending when the validation loss no longer improves.

Table 5.5 shows the error metrics of the model in different datasets: the errors obtained in the different datasets for all error metrics are very close to each other, this means that the model generalizes well the data it has never seen. The root averaged mean-squared error

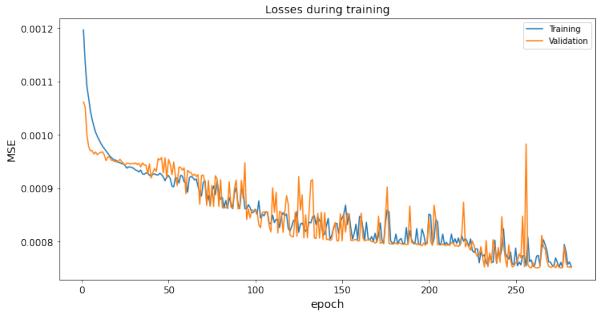


Figure 5.7
Training and validation losses of the final LATTICE-ANN during training

(RMSE) is around 2.7×10^{-2}, which is an indication that the average pricing error is 2.7% of the strike price.

Table 5.5
LATTICE-ANN performance on the whole data set

LATTICE-ANN	MSE	RMSE	MAE	MAPE
Training set	$7.47 \cdot 10^{-4}$	$2.72 \cdot 10^{-2}$	$8.94 \cdot 10^{-3}$	$5.75 \cdot 10^{-2}$
Validation set	$7.51 \cdot 10^{-4}$	$2.73 \cdot 10^{-2}$	$8.98 \cdot 10^{-3}$	$5.49 \cdot 10^{-2}$
Testing set	$7.46 \cdot 10^{-4}$	$2.72 \cdot 10^{-2}$	$8.62 \cdot 10^{-3}$	$5.65 \cdot 10^{-2}$

5.2.2 HESTON-ANN Results

The final **HESTON-ANN** model has been trained on 80% of the whole heston dataset, which contains more than 1,000,000 samples. Despite the heston model is a more complex model than the binomial (or trinomial) model, it is interesting to note, from figure 5.8, that the training phases has only last for 86 epochs compared to 275 epochs obtained in the **LATTICE-ANN** training phase. The final model MSE is also lower, ending with a loss under 0.0001, than the final MSE obtained after the LATTICE-ANN training.

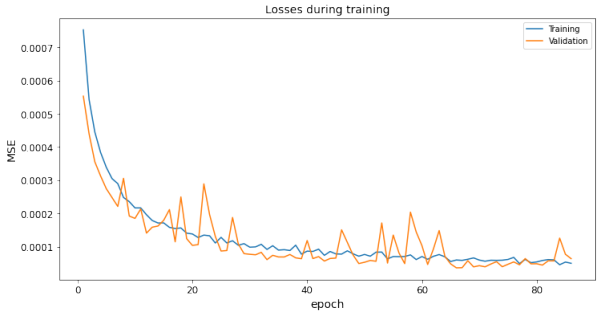


Figure 5.8
Training and validation losses of the final HESTON-ANN during training

The error metrics are shown in Table 5.6 show, also in this case, that the final model is able to generalize well on data it has never seen. The fact the error metrics obtained with the Heston dataset are better than the error metrics obtained in the Binomial-Trinomial dataset

is probable due to the fact the neural nets are capable of capturing the mechanics of more complex underlying models, such as the Heston model.

The root averaged mean-squared error (RMSE), in this case, is around 8.0×10^{-3}, which is an indication that the average pricing error is 0.8% of the strike price.

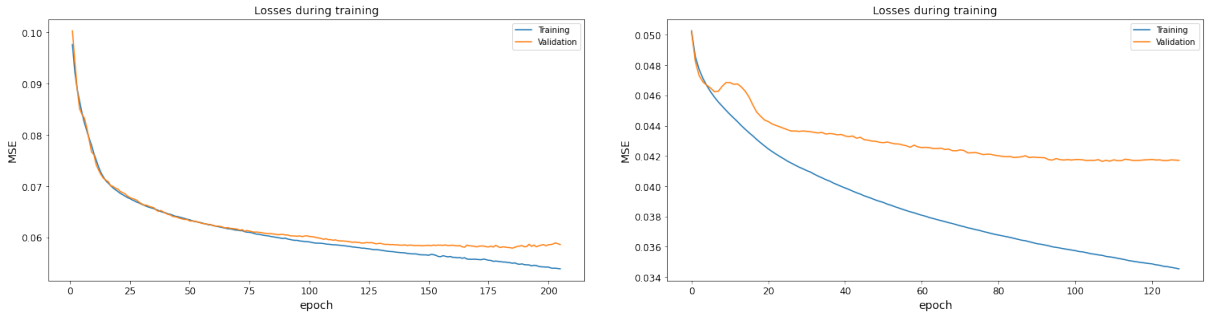
Table 5.6
HESTON-ANN performance on the whole data set

HESTON-ANN	**MSE**	**RMSE**	**MAE**	**MAPE**
Training set	$5.91 \cdot 10^{-5}$	$7.68 \cdot 10^{-3}$	$6.12 \cdot 10^{-3}$	$3.52 \cdot 10^{-2}$
Validation set	$6.38 \cdot 10^{-5}$	$7.98 \cdot 10^{-3}$	$6.30 \cdot 10^{-3}$	$3.73 \cdot 10^{-2}$
Testing set	$6.38 \cdot 10^{-5}$	$7.98 \cdot 10^{-3}$	$6.28 \cdot 10^{-3}$	$4.06 \cdot 10^{-2}$

5.2.3 REAL-ANN Results

Regarding the **REAL-ANN** model training, two different datasets have been distinguished, i.e, the "total" dataset (which is the whole dataset) and a "cleaned" dataset. The cleaned dataset has been obtained by removing all the illiquid options (options with a high bid-ask spread) and all the options with an extreme value of the moneyness.

Two different models, with the same hyperparameters, have been trained on these two datasets. Figure 5.9a shows the training and validation losses during training of the **REAL-ANN** on the total dataset, while Figure 5.9b shows the losses during training of the **REAL-ANN** on the cleaned dataset. The training of the **REAL-ANN** on the cleaned dataset results in lower epochs and better final MSE.



(a) Total dataset (b) Cleaned dataset

Figure 5.9
Training and validation losses of the final REAL-ANN during training

Indeed, comparing the results obtained in Table 5.8 and the results obtained in Table 5.7, an improvement in all the error metrics is visible. Similarly to before, not noticing a big difference between the training error and the validation and testing errors, it can be said that the model has generalized well the training data.

Despite the **REAL-ANN** having the same architecture as the **HESTON-ANN** the mean MSE obtained with the different sets of the real dataset are higher by two orders of magnitude. One of the possible reasons could be the higher complexity of the option pricing models used by the market makers.

Table 5.7
Model performance on the whole data set

REAL-ANN	MSE	RMSE	MAE	MAPE
Training set	$5.61 \cdot 10^{-2}$	$2.35 \cdot 10^{-1}$	$8.57 \cdot 10^{-2}$	$5.37 \cdot 10^{-1}$
Validation set	$5.86 \cdot 10^{-2}$	$2.40 \cdot 10^{-1}$	$8.77 \cdot 10^{-2}$	$4.44 \cdot 10^{-1}$
Testing set	$6.28 \cdot 10^{-2}$	$2.49 \cdot 10^{-1}$	$9.01 \cdot 10^{-2}$	$6.98 \cdot 10^{-1}$

Table 5.8
Model performance on the cleaned data set

REAL-ANN	MSE	RMSE	MAE	MAPE
Training set	$3.53 \cdot 10^{-2}$	$1.84 \cdot 10^{-1}$	$5.92 \cdot 10^{-2}$	$4.85 \cdot 10^{-1}$
Validation set	$4.17 \cdot 10^{-2}$	$1.99 \cdot 10^{-1}$	$6.25 \cdot 10^{-2}$	$4.27 \cdot 10^{-1}$
Testing set	$3.51 \cdot 10^{-2}$	$1.84 \cdot 10^{-1}$	$5.99 \cdot 10^{-2}$	$4.61 \cdot 10^{-1}$

Overall, it seems a good practice to train the ANN on the cleaned dataset, that's because a better quality of the data improves the performance of the model.

5.3 Implied Volatility

As introduced in Section 1.5 the implied volatility of an option contract is the value of volatility which, when given as an input to an option pricing model, will return the value of the said option. Usually, this value is derived by the ask (or the bid) of a specific option contract. In this way is possible to get to know how the market is pricing the implied volatility of the interest option contract.

The importance of implied volatility for market participants stems from the fact that it is one of the only data that is **forward-looking**. This is because market participants always trade contracts with an expiration date later in time. Consequently, obtaining the value of the implied volatility of various option contracts is crucial for traders.

In section 1.5 it was also explained how to obtain the implied volatility value using the Newton-Raphson method. Since previous trained ANNs models, can be considered approximators of the target option pricing model, the values of the implied volatility can be also obtained from those ANNs models.

Figure 5.10 shows the implied volatility surface of an various option chain of puts generated using the Heston model with parameters $\kappa = 1.5, \rho = -0.05, \theta = 0.45, x_i = 0.3, v_0 = 0.25, r = 0.02$, with a value of moneyness between 0.66 and 1.5 and with time to expiration between 0.1 and 1.1. The volatility smile and the term structure of volatility of both surfaces are similar, while the iv surface obtained from the HESTON-ANN prediction is smoother than the iv surface obtained from the put target prices. It is interesting to note that the iv values of the ANN iv surface are higher than the iv values of the "real" iv surface.

The iv surface can be also obtained using the real dataset. In this case, the value of the

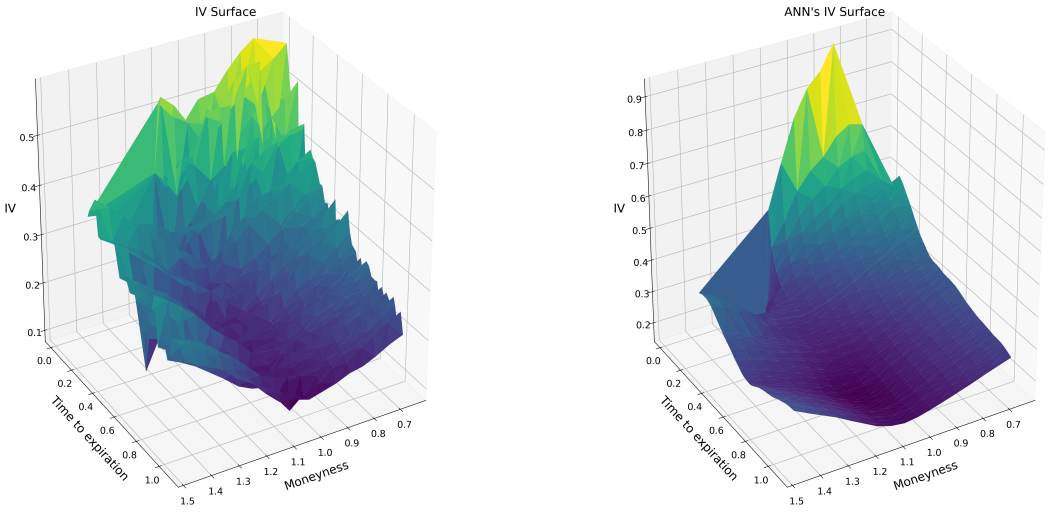


Figure 5.10
Implied volatility surface derived from the HESTON-ANN prices and the real target prices

"real" implied volatilities are not obtained using the Newton-Raphson method, but they are directly given by the data provider. So values of the REAL-ANN implied volatility can be directly compared with real market implied volatility values.

Figure 5.11 shows the implied volatility surfaces of all the put options of the NVDA stock on the date 2022/05/25. The structure of the two surfaces is quite similar. Interestingly, as in the previous case, we have that the surface obtained from the predictions of the neural network turns out to be smoother than the real volatility surface. It should also be noted that the implicit volatility values obtained for short expiration options in the ANN surface are consistently higher than in the real iv surface.

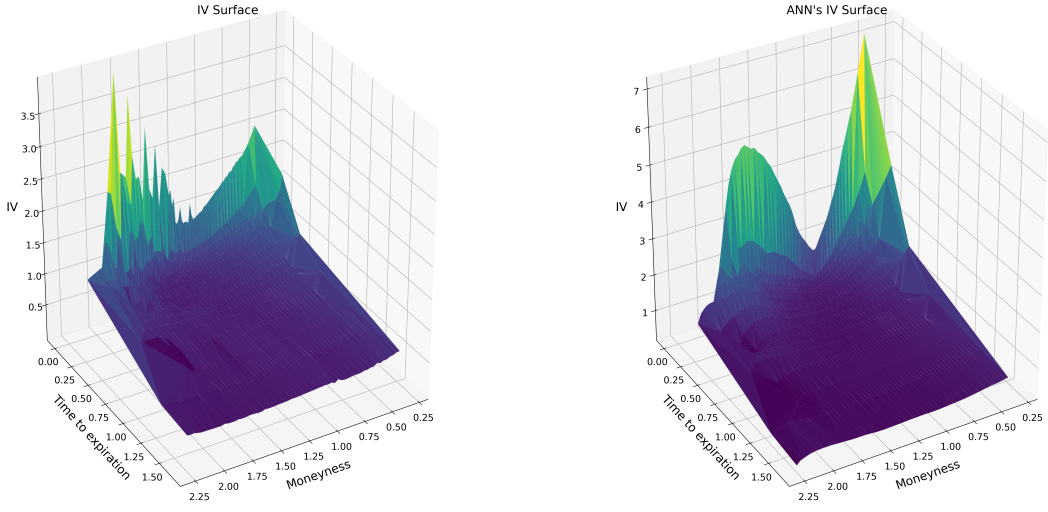


Figure 5.11
Implied volatility surface derived from the REAL-ANN predictions and the real target prices

Chapter 6

Conclusions

In this work, it has been shown how neural networks, as universal function approximators, can learn different option pricing models. This is true for pricing models that assume that volatility is constant, and more complex pricing models that instead assume that volatility itself is a stochastic process. Neural Networks are not only able to learn those known mathematical models but it has been also shown that they can learn real market option pricing mechanics with a high degree of accuracy making them good candidates for real usage. In this context, the quality of the data used in the training phases is crucial to obtaining an accurate and consistent ANN pricer. Often, in the Deep Learning community, the priority is given to the quantity of the data sacrificing the quality of the data.

Deep Learning models depend on different not learnable parameters, called hyperparameters, that directly control the training process. Navigating the hyperparameter space often requires expert knowledge and can be time-consuming. Leveraging a probabilistic optimization technique, such as the Bayesian optimization, allowed us to find not only the best combination of the hyperparameters but also the most robust.

Although this data-driven approach can be time-consuming in the training phases, the online prediction will be faster compared to partial differential equations solvers and Monte Carlo methods. That's because at the core level Neural Nets consist of matrix multiplication and function application, which is way cheaper than other methods. Moreover, parallel computing allows Neural Networks to process option contracts "in batch mode", boosting, even more, the computation.

Implied volatility values can be obtained not only from the known pricing model but also from the ANN pricers too. As pointed out in section 5.5, these values are crucial for market participants because they are one of the only values that are forward-looking. Obtaining those values from the Deep learning model can also help to understand how the model is pricing the risk of a specific option contract.

Further research could focus on trying different Neural Network architectures, exploiting newer tricks to improve the accuracy. It could also be interesting to verify the performance

of the neural network on a bigger real dataset, which contains data on various assets and with several years of history, and how it can adapt to market conditions. Furthermore, the options Greeks can be calculated from the trained ANN using the gradient information or using Automatic differentiation to calculate the derivatives w.r.t the inputs.

Bibliography

[1] David Anderson e Urban Ulrych. "Accelerated American Option Pricing with Deep Neural Networks". In: *SSRN* (2021). DOI: 10.2139/ssrn.4000756. URL: http://dx.doi.org/10.2139/ssrn.4000756 (cit. a p. v).

[2] Shuaiqiang Liu, Cornelis Oosterlee e Sander Bohte. "Pricing Options and Computing Implied Volatilities using Neural Networks". In: *Risks* 7.1 (2019), p. 16. DOI: 10.3390/risks7010016. URL: https://doi.org/10.3390%2Frisks7010016 (cit. a p. v).

[3] Steven L. Heston. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". In: *The Review of Financial Studies* 6.2 (1993), pp. 327–343. ISSN: 08939454, 14657368. URL: http://www.jstor.org/stable/2962057 (visitato il 21/06/2022) (cit. a p. 11).

[4] Giuseppe Orlando e Giovanni Taglialatela. "A review on implied volatility calculation". In: *Journal of Computational and Applied Mathematics* 320 (2017), pp. 202–220. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2017.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0377042717300602 (cit. a p. 16).

[5] Xavier Glorot, Antoine Bordes e Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323 (cit. a p. 18).

[6] Andrew L. Maas. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". In: 2013 (cit. a p. 18).

[7] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. DOI: 10.48550/ARXIV.1502.01852. URL: https://arxiv.org/abs/1502.01852 (cit. a p. 18).

[8] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861–867. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80131-5. URL: https://www.sciencedirect.com/science/article/pii/S0893608005801315 (cit. a p. 19).

[9] Zhou Lu et al. *The Expressive Power of Neural Networks: A View from the Width*. 2017. DOI: 10.48550/ARXIV.1709.02540. URL: https://arxiv.org/abs/1709.02540 (cit. a p. 19).

[10] Diederik P. Kingma e Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980 (cit. alle pp. 21, 33).

[11] John Duchi, Elad Hazan e Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: http://jmlr.org/papers/v12/duchi11a.html (cit. a p. 21).

[12] Geoffrey Hinton, Nitish Srivastava e Kevin Swersky. "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent". In: *Cited on* 14.8 (2012), p. 2 (cit. a p. 21).

[13] David E. Rumelhart, Geoffrey E. Hinton e Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0 (cit. a p. 21).

[14] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: (2015). DOI: 10.48550/ARXIV.1502.05767. URL: https://arxiv.org/abs/1502.05767 (cit. a p. 21).

[15] Francis Longstaff e Eduardo Schwartz. "Valuing American Options by Simulation: A Simple Least-Squares Approach". In: *Review of Financial Studies* 14 (feb. 2001), pp. 113–47. DOI: 10.1093/rfs/14.1.113 (cit. a p. 26).

[16] James Bergstra e Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html (cit. a p. 27).

[17] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. 2018. DOI: 10.48550/ARXIV.1807.02811. URL: https://arxiv.org/abs/1807.02811 (cit. a p. 28).

[18] Donald Jones, Matthias Schonlau e William Welch. "Efficient Global Optimization of Expensive Black-Box Functions". In: *Journal of Global Optimization* 13 (dic. 1998), pp. 455–492. DOI: 10.1023/A:1008306431147 (cit. a p. 31).

[19] R.S. Sutton e A.G. Barto. "Reinforcement Learning: An Introduction". In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 1054–1054. DOI: 10.1109/TNN.1998.712192 (cit. a p. 31).

[20] L. P. Kaelbling, M. L. Littman e A. W. Moore. "Reinforcement Learning: A Survey". In: (1996). DOI: 10.48550/ARXIV.CS/9605103. URL: https://arxiv.org/abs/cs/9605103 (cit. a p. 31).

[21] Alireza Zaeemzadeh, Nazanin Rahnavard e Mubarak Shah. *Norm-Preservation: Why Residual Networks Can Become Extremely Deep?* 2018. DOI: 10.48550/ARXIV.1805.07477. URL: https://arxiv.org/abs/1805.07477 (cit. a p. 33).

[22] I.M Sobol'. "On the distribution of points in a cube and the approximate evaluation of integrals". In: *USSR Computational Mathematics and Mathematical Physics* 7.4 (1967), pp. 86–112. ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(67)90144-9. URL: https://www.sciencedirect.com/science/article/pii/0041555367901449 (cit. a p. 33).