UNIVERSITÀ DEGLI STUDI DEL PIEMONTE ORIENTALE

DIPARTIMENTO DI STUDI PER L'ECONOMIA E L'IMPRESA

Corso di Laurea Magistrale in Management and Finance

Neural Network Approaches to Option Pricing and Hedging

Approcci basati su reti neurali per il pricing e l'hedging di opzioni

Relatore:
Prof. Gianluca Fusai

Correlatore:
Prof. Kwo Lik CHAN

Laureando: Gabriele Cartella
Matricola: 20027938

Anno Accademico 2025/2026

data ufficiale di laurea: Luglio 2026

"Intelligence is the ability to adapt to change."
— Stephen Hawking

Acknowledgements

This thesis is the result of many hours of work and research. It was a demanding process, but a rewarding one.
My first thanks go to my supervisors, Professor Gianluca Fusai and Professor Kwo Lik Chan. They sparked my interest in statistics and quantitative finance, and they dedicated their time to guiding me through the writing of this work.
I also wish to thank my family, who supported me unconditionally at every moment.
Finally, I want to thank my friends. My time at university has been a wonderful experience, and it has shaped who I am today.

Gabriele Cartella

Contents

Introduction

Pricing an option is one of the classic problems of quantitative finance. The payoff of the contract at maturity is fixed by its terms, but its value at any earlier date is not: it depends on the uncertain future of the underlying asset and on several market variables that enter in a nonlinear way. Finding a fair value for this contract before it expires has occupied the financial literature for more than a century (Hull, 2012).

The reference answer is the Black–Scholes–Merton model, introduced in 1973 (Black & Scholes, 1973; Merton, 1973). From a small set of assumptions on the underlying dynamics, it delivers a closed-form formula for the price of a European option, and it has remained the benchmark in both academic work and market practice (Hull, 2012). The model assumes that the underlying follows a geometric Brownian motion with constant volatility, that the market is frictionless and free of arbitrage, and that trading takes place continuously. These same assumptions, which give the model its closed-form solution, also constrain it. Returns are not normally distributed, volatility is not constant, and the volatility implied by observed prices changes with strike and maturity. When the assumed dynamics depart from the data, the model produces pricing and hedging errors on the contracts written on that asset. A different route leaves the functional form unspecified and recovers the pricing function directly from observed data, making no assumption of log-normality or constant volatility. Artificial neural networks are the most flexible estimators of this type, and Hutchinson et al. (1994) were among the first to apply them to option pricing, obtaining a data-driven pricing function that could also be differentiated to produce a hedge ratio.

This thesis replicates the experiment of Hutchinson et al. (1994) and extends it with a modern architecture. The full Python implementation is available in a public GitHub repository.[1]

Three learning networks are compared against a dividend-adjusted Black–Scholes benchmark: a shallow multilayer perceptron and a radial basis function network, both implemented from scratch following the original study, and a deep multilayer perceptron added as an extension. Each network takes the moneyness and the time to maturity as inputs and returns the normalised option price; the option delta is obtained by differentiating this price with respect to the underlying price. The models are first tested on data simulated under geometric Brownian motion, where the true pricing function is known, and then on real S&P500 index options observed between 2016 and 2023. Performances are measured both by the out-of-sample pricing fit and by the tracking error of a delta-hedging strategy that uses each model's own delta.

[1] https://github.com/gcartella/master-thesis-nn-options

Two questions guide the analysis. The first is whether the networks recover the Black–Scholes formula when it is the true model, as on the simulated data. The second is whether they improve on Black–Scholes when its assumptions fail, as on the real data. The results address both. On the simulated data all three networks reproduce the pricing surface almost exactly, yet none beats the benchmark, since there is nothing to improve on. On the real data the networks beat Black–Scholes on out-of-the-money options during turbulent periods, where the constant-volatility assumption is most strained, but not in the calm periods nor across the sample as a whole. In both datasets, the deep network is the most accurate and the most stable of the three. These findings reflect the cautious conclusion of the original study: learning networks rival Black–Scholes, and improve on it where it is misspecified, without overturning it. More than three decades after the work of Hutchinson et al. (1994), the question it raised has only grown in relevance, as data-driven methods take on a larger role in financial markets.

The thesis is organised in four chapters. Chapter 1 introduces derivative contracts and the basic features of options. Chapter 2 sets out the theoretical foundations of option pricing, from the stochastic processes used to model asset prices to the Black–Scholes–Merton formula and its parametric extensions. Chapter 3 presents the non-parametric approach and the neural network architectures used in the empirical work. Chapter 4 carries out the replication and the extension, describing the datasets, the implementation of the models, the hedging methodology, and the results on both simulated and real data. The conclusions then summarise the findings and outline directions for future work.

Chapter 1

Derivative contracts and options: a theoretical overview

1.1 An introduction to derivative contracts

This chapter reviews the fundamental notions on derivative contracts and options that the rest of the thesis builds on. These are standard concepts, not original to this work; they are drawn mainly from Hull (2012) and recalled here to keep the thesis self-contained.

Derivative contracts can be defined as financial instruments whose value depends on, or derives from, an **underlying** entity, which can be an individual asset, a group of assets, or a benchmark. Usually, the variable that drives the price of a derivative is the price of the underlying. For example, a futures contract on a stock is a derivative whose value and payoff depend on the price of the stock.

The first derivative contracts date back to ancient times. According to Aristotle, *Thales of Miletus* (580 BC) accumulated wealth by buying in winter, when demand was low, an option on the use of a number of olive presses for the following autumn, when demand was expected to be high. A later example dates to 1164, in Genoa, where the municipality sold its future tax revenues to a financial institution, known as the *Monte*, in exchange for an immediate advance; an arrangement regarded as the first derivative contract entered into by a local authority. The first organised derivatives markets date back to the sixteenth and seventeenth centuries: in 1600 forward contracts were admitted to trading at the *Royal Exchange* in London (CONSOB, 2026).

Since the second half of the twentieth century, derivatives have experienced massive growth as a result of a series of events, such as the collapse of the Bretton Woods system in 1971, which introduced exchange rate risk in international positions, and the oil shocks of 1973 and 1979 (CONSOB, 2026). The internationalisation of markets and the spread of computer technology also contributed to the growth of these instruments. In 1973, Fischer Black and Myron Scholes derived a closed-form formula for the price of a European option (Black & Scholes, 1973). In the same year, Robert C. Merton independently extended and generalised the underlying theory (Merton, 1973); for this reason the framework is commonly referred to as the Black–Scholes–Merton model. This work laid the foundations of modern derivative pricing and is examined extensively in this thesis. Together with

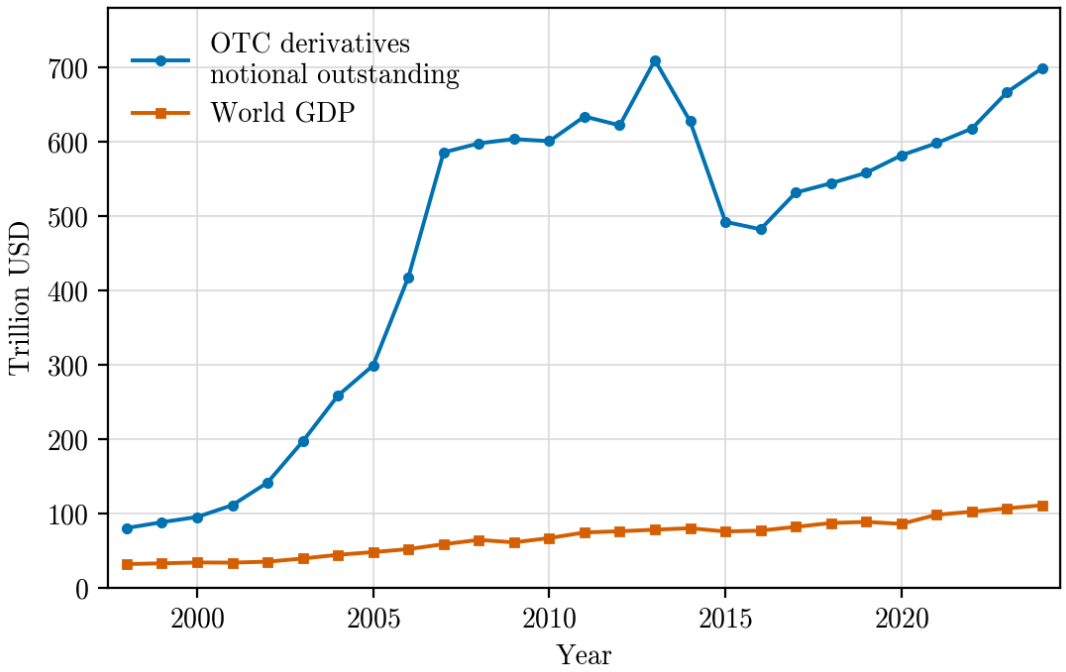


Figure 1.1: Comparison of world GDP and OTC derivatives notional amounts outstanding, 1998–2024. Sources: BIS OTC derivatives statistics (notional amounts outstanding, all contracts); World Bank (`NY.GDP.MKTP.CD`).

the rapid growth in computational power and the increasing globalisation of financial markets, it contributed to the strong expansion of derivative trading during the 1980s and 1990s. Over time, derivatives became essential instruments for hedging, speculation, and portfolio management.

However, the global financial crisis highlighted the risks associated with the excessive and opaque use of complex derivative products, particularly in **over-the-counter (OTC)** markets. Instruments such as credit derivatives and insufficiently regulated structured products amplified systemic vulnerabilities and contributed to financial instability. As a consequence, the post-crisis period was characterised by a broad regulatory response, including stronger reporting requirements, central clearing obligations, higher capital standards, and enhanced market transparency.

Nevertheless, by December 2010 the notional amount of outstanding OTC derivatives had reached approximately USD 600 trillion, out of a total of about USD 670 trillion including exchange-traded contracts (CONSOB, 2026). Considering that global GDP in the same period was estimated at around USD 70 trillion, these figures illustrate both the scale of the derivatives market and its systemic relevance within the global financial system (Figure 1.1).

1.2 Main categories of derivative contracts

Derivative contracts can generally be divided into three main categories: **term contracts** (forward and futures), **swaps** and **options**. Although these instruments differ in their structures and uses, they all share one common feature: their value depends on the performance of an underlying asset or financial variable, such as stocks, bonds, commodities, currencies, interest rates, or market indices.

Among the most basic forms of derivatives are **forward** and **futures** contracts. In both cases, two parties agree to buy or sell a certain quantity of an underlying asset at a predetermined price on a future date, known as the *maturity date*. Their value therefore changes according to movements in the price of the underlying asset.

For the buyer, the main risk is that the market price at maturity falls below the agreed price, meaning the asset must be purchased at a less favourable value. For the seller, the opposite risk applies: if the market price rises above the contractual price, the asset must be delivered below its current market value.

Despite these similarities, forward and futures contracts differ in the way they are traded. Forward contracts are private agreements negotiated directly between the two parties in the over-the-counter (OTC) market, and their terms can be customised to specific needs. Futures contracts, by contrast, are standardised agreements traded on organised exchanges, where contract size, maturity dates, and settlement procedures are predetermined. This standardisation generally improves liquidity and, together with the presence of clearing houses, reduces **counterparty risk**.

Swaps are instruments used by two parties to exchange cash flows at certain times. These can be based upon interest rates, currencies and assets. **Interest rate swaps (IRS)** are contracts where the parties exchange payments arising from interest calculated on the notional principal amount, for a period of time defined by the contract. The most common IRS is the *plain vanilla swap*, where one cash flow is fixed, while the other one is indexed to a floating rate. The expectation of the floating interest moving against the fixed one represents the risk (and the reward) of such type of contract. In most IRS contracts the floating rate is SOFR (Secured Overnight Financing Rate), a broad measure of the cost of borrowing cash overnight collateralised by U.S. Treasury securities in the repurchase (repo) market. Published daily by the Federal Reserve Bank of New York, it replaced LIBOR in 2023[1].

Currency swaps are contracts in which two counterparties exchange principal amounts and interest payments in one currency for principal amounts and interest payments in another currency. These may involve fixed vs fixed, fixed vs floating, or floating vs floating interest payment structures. For example, one counterparty may exchange fixed rate interest payments denominated in euros for floating-rate payments denominated in U.S. dollars linked to USD LIBOR (or, more recently, SOFR). These derivatives are fundamental to manage currency risk and interest rate exposure.

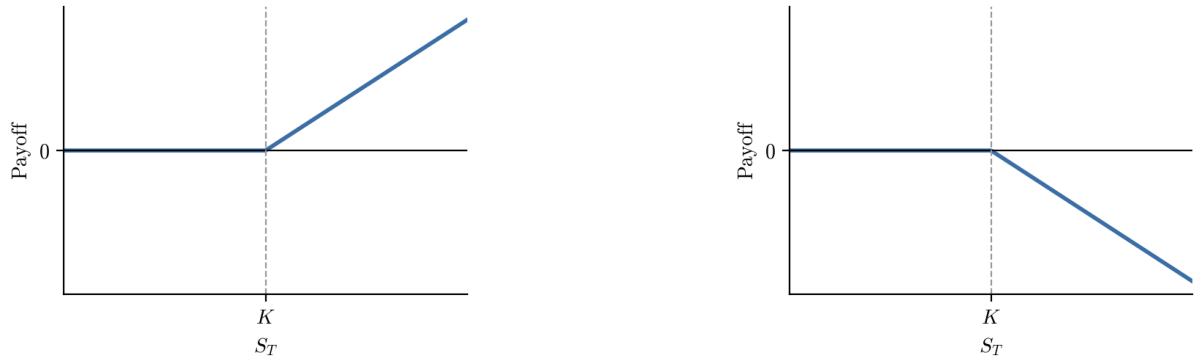
Asset swaps are derivatives in which one counterparty holds a bond or another security

[1]The Federal Reserve Bank of New York describes SOFR as "a much more resilient rate than LIBOR", being an overnight secured rate grounded in deep and liquid markets (Federal Reserve Bank of New York, 2021).

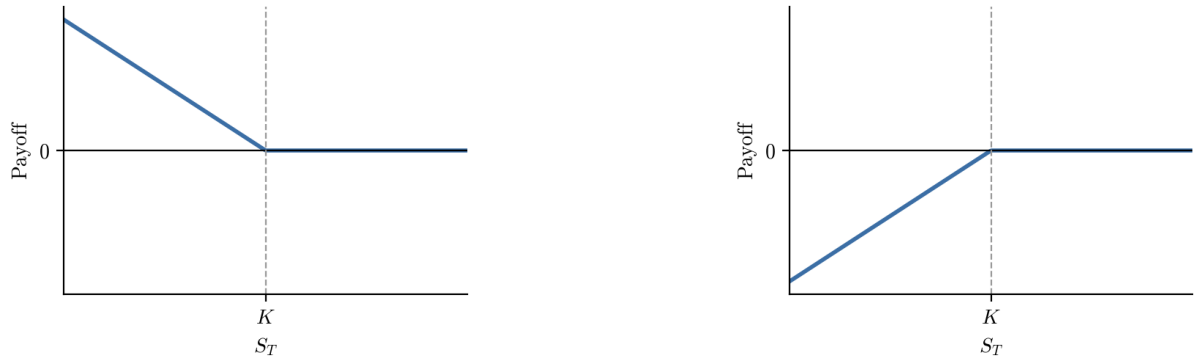
and agrees to transfer the cash flows generated by that asset to another party in exchange for a different stream of interest payments. Typically, fixed-rate payments from the bond are exchanged for floating-rate payments, or vice versa. These instruments are used to transform the interest rate characteristics of an asset in order to manage interest rate exposure.
A particular category of swaps is represented by **Credit Default Swaps (CDS)**. In these contracts, one party, known as the protection buyer, makes periodic payments to the other party, known as the protection seller, in exchange for protection against specified credit events affecting a reference entity or asset.
The underlying reference may consist of a bond issue, a corporate or sovereign issuer, or an entire portfolio of financial instruments. Consequently, the primary purpose of a CDS is to hedge the credit risk associated with a particular exposure, although such instruments may also be used for speculative or investment purposes.
Having briefly outlined the principal classes of derivative instruments, attention can now be directed to option contracts, which represent the central focus of this thesis and will therefore be examined in greater detail in the following sections.

1.3 Option contracts

Options represent one of the most important and studied categories of derivative instruments, and play a central role in financial markets. An option is a financial contract that gives its holder the *right*, but not the *obligation*, to buy or sell an underlying asset at a predetermined price, known as the *strike price*, within a specified period of time or at the expiration date.
The two basic forms of option contracts are **call options** and **put options**. A call gives the holder the right to purchase the underlying asset, while a put option gives the holder the right to sell it under the agreed contractual conditions.
These instruments are traded both on organised exchanges and in over-the-counter (OTC) markets. Among organised exchanges, the **Chicago Board Options Exchange (CBOE)** has historically been one of the leading and most influential marketplaces for option trading worldwide.
Options can also be classified according to the timing of their exercise rights. **American** options may be exercised at any time up to the expiration date, whereas **European** options may only be exercised on the expiration date itself.
The underlying assets may include individual stocks (such as Apple or NVIDIA), foreign currencies, market indices (such as the S&P500 or the Dow Jones Industrial Average) and futures contracts.
Although option contracts may be linked to many underlying assets, in what follows the thesis focuses specifically on *European options*. This choice is consistent with both the empirical dataset employed and the pricing models analysed in the following chapters.
Regardless of the underlying type, options are characterised by an asymmetric payoff structure. Unlike forward or futures contracts, an option gives the holder the right, but not the obligation, to exercise it, so the downside is limited to the premium paid. The upside, by contrast, is potentially unlimited for a call, since the underlying price can



(a) Long call **(b)** Short call



(c) Long put **(d)** Short put

Figure 1.2: Payoff at maturity of the four basic option positions, as a function of the underlying price S_T, for a strike price K. The holder of a long position pays a premium and faces limited downside; the writer of a short position receives the premium and bears the corresponding obligation if the option is exercised.

rise without bound, whereas for a put it is bounded, reaching its maximum when the underlying price falls to zero.
At expiration, the payoff of a call option is given by:

$$\max(S_T - K, 0) \tag{1.1}$$

where S_T is the price of the underlying stock at maturity T, and K represents the strike price specified in the contract. The operator $\max(\cdot)$ indicates that the payoff is equal to the greater value between $S_T - K$ and zero.
This means that the holder of a call option will exercise the contract only when it is economically advantageous to do so, that is, when the market price of the stock is higher than the strike price. In that case, the investor can purchase the asset at the lower contractual price and obtain an immediate gain equal to $S_T - K$.
Conversely, if the stock price at maturity is below the strike price, exercising the option would not be rational, since the asset could be purchased on the market at a lower price.

In such a case, the option expires and the payoff is equal to zero.
The payoff of a put option, instead, is given by:

$$\max(K - S_T, 0) \tag{1.2}$$

Unlike a call, the holder of a put option will exercise the contract only when the market price of the stock is below the strike price. In that case, the investor can sell the asset at the higher contractual price and obtain an immediate gain equal to $K - S_T$.
For the holder of a put, then, both the loss and the gain are bounded: the loss is limited to the premium paid, while the gain reaches its maximum, equal to K, when the underlying price falls to zero. This contrasts with the call, whose gain is unlimited.
An investor may also take different positions in option contracts, giving rise to four basic payoff structures, as illustrated in Figure 1.2. A *long position* refers to the purchase of the option contract, whereas a *short position* indicates that the investor has sold, or written, the option. In the latter case, the premium is received upfront, but the losses can be potentially unlimited in the case of a written call, since the obligation lies within the writer if the option is exercised.
To illustrate clearly how an option contract works, consider Example 1 below:

Example 1. Consider a European call option on a stock currently trading at \$50, with strike price $K = \$55$ and a maturity of three months.
If, at expiration, the price of the stock rises to $S_T = \$60$, the holder will exercise the option, buying the stock at the contractual strike price and obtaining a payoff of \$5.
However, if the price of the stock at maturity is $S_T = \$45$, exercising the option is not optimal, since the stock can be bought directly in the market at a price lower than the strike ($45 < 55$). In this case, the option expires worthless and the payoff is zero.

1.4 Hedging positions and option strategies

Due to the flexibility of option positions and contractual combinations, these instruments can be used both for hedging purposes and for the implementation of speculative strategies based upon specific market expectations, such as periods of high volatility. Their asymmetric payoff structure allows investors to put a floor on losses while preserving the upside potential, net of the premium paid.
To illustrate the hedging function of option contracts, consider Example 2 below.

Example 2. Consider an investor holding a stock currently worth \$100. Since future market movements are uncertain, the investor fears a possible decline in the stock price and decides to purchase a European put option with strike price $K = \$95$. Suppose that, at expiration, the stock price falls to $S_T = \$80$. In this scenario, the stock position suffers a loss of \$20, while the put option generates a payoff of \$15.
As a consequence, the gain generated by the put option offsets a substantial portion of the loss suffered on the underlying stock position. Therefore, the investor effectively establishes a minimum portfolio value close to the strike price, net of the premium paid for the option.

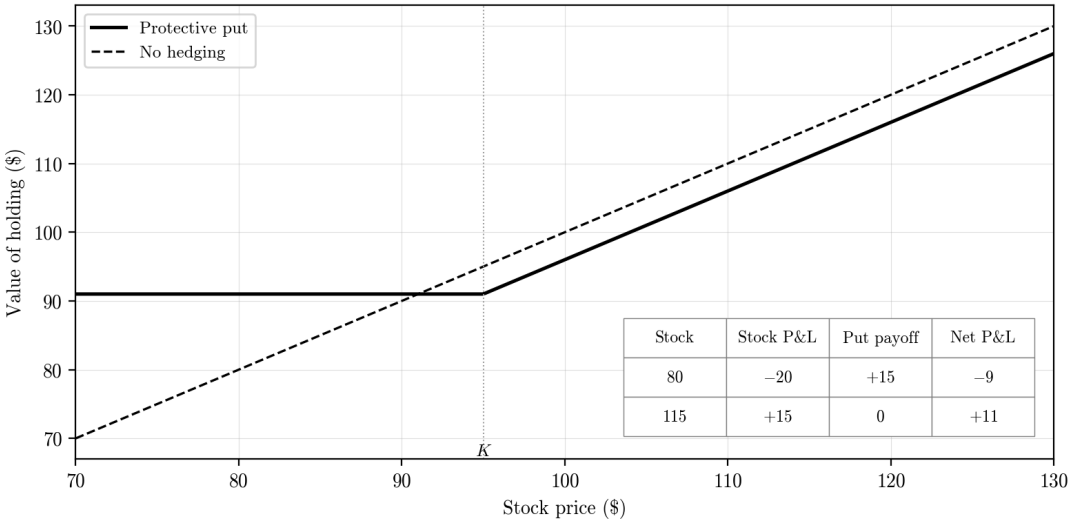


Figure 1.3: Portfolio value at maturity under a protective put strategy (stock plus a long put with strike $K = \$95$ and premium $p = \$4$), compared with an uncovered stock position.

Conversely, if the stock price rises above the strike price, for example to \$115, the put option expires worthless, but the investor still benefits from the appreciation of the stock. This example illustrates how option contracts can be used to hedge downside risk while maintaining upside potential.

As illustrated in Figure 1.3, the protective put establishes a lower bound for the loss, while maintaining the upside gains.

This flexibility is also demonstrated in option strategies, which result from the combination of two or more contracts and are used for protection or speculation. These strategies can generally be divided into two broad categories: **spreads** and **combinations** (Hull, 2012).

Spreads are constructed by taking positions in two or more options of the same class (either calls or puts), typically with different strike prices and/or maturities.

Common examples include bull spreads, bear spreads, calendar spreads, and diagonal spreads. Such strategies are often used to express directional market views while limiting risk exposure.

Combinations, by contrast, involve the simultaneous use of both call and put options written on the same underlying asset. Examples include the straddle, strip, strap, and strangle. These strategies are usually employed when investors want to take positions on expected volatility rather than on a specific market direction.

For example, a long straddle involves taking a long position in both a call and a put written on the same underlying asset, with the same strike price and expiration date. This strategy is typically used when an investor expects significant volatility, but is uncertain about the direction of the movement.

To illustrate a speculative option strategy based on volatility expectations, consider Example 3 below.

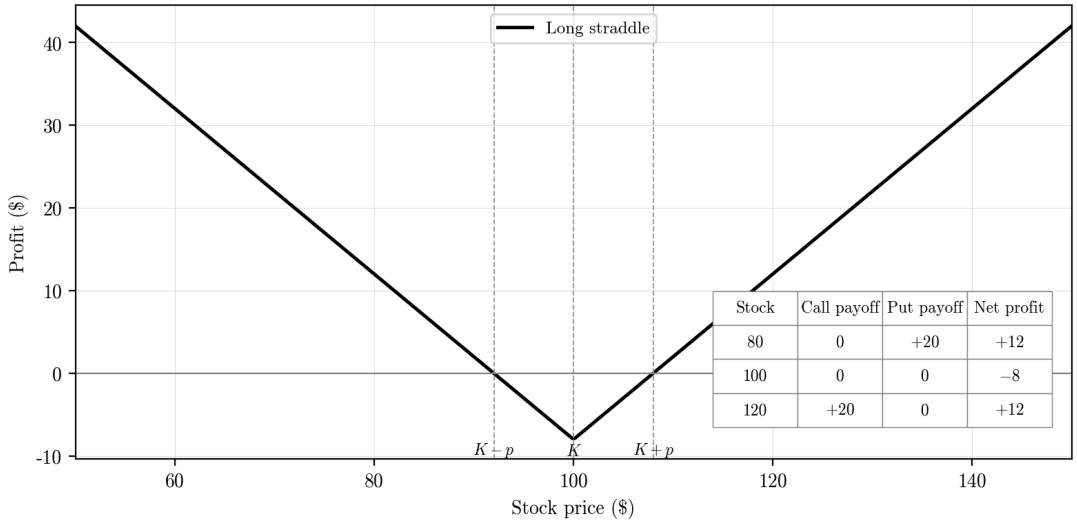


Figure 1.4: Profit at maturity of a long straddle (long call plus long put) with strike $K = \$100$ and total premium $p = \$8$. The dashed lines mark the break-even points $K \pm p$.

Example 3. Consider a stock currently trading at $S_0 = \$100$. Suppose that an investor simultaneously purchases a European call option and a European put option on the same underlying asset, with identical strike price $K = \$100$ and the same expiration date.
Assume that the premium paid for the call option is \$4 and the premium paid for the put option is also \$4, resulting in a total strategy cost of \$8. If, at expiration, the stock price rises to $S_T = \$120$, the call option generates a payoff of \$20 while the put option expires worthless, so the net profit of the strategy is \$12. Conversely, if the stock price falls to $S_T = \$80$, the put option generates a payoff of \$20 while the call option expires worthless, and again the net profit is \$12. Finally, if the stock price at expiration remains equal to the strike price ($S_T = \$100$), both options expire worthless and the investor incurs a loss equal to the total premium paid, namely \$8. The break-even points of the strategy are therefore $100 - 8 = 92$ and $100 + 8 = 108$.
This strategy is commonly referred to as a *long straddle*, and is typically employed when the investor expects high volatility but is uncertain about the direction of future price movements.

As shown in Figure 1.4, large upward movements generate gains through the call option, while large downward movements generate gains through the put option. The maximum loss is limited to the total premium paid for the two options and occurs when the stock price remains close to the strike price at expiration.
As shown in this chapter, while the payoff of an option at maturity follows from its contractual terms, assigning a value before expiration is far harder, since that value depends on the future dynamics of the underlying. This is the problem addressed in the next chapter, which introduces the main theoretical approaches to option pricing, and the Black–Scholes–Merton model in particular.

Chapter 2

Theoretical foundations of option pricing

2.1 Introduction to option pricing

As discussed in the previous chapter, while the payoff of an option at maturity can be determined straight forward, assigning a *fair value* to an option prior to its expiration is a significantly more complex problem. This comes from uncertainty about the future evolution of the underlying asset and the multiple factors that influence option prices.

This issue has been extensively studied in the financial literature, starting from Bachelier (1900) with his work *Théorie de la spéculation* and leading to the development of a wide range of pricing models designed to capture the key determinants of option prices while preserving computational efficiency. Among these contributions, the Black–Scholes–Merton model, introduced by Black and Scholes (1973) in *The Pricing of Options and Corporate Liabilities* and, independently, by Merton (1973) in *Theory of Rational Option Pricing*, represents a fundamental milestone and still remains both an academic benchmark and a professional standard.

In the following years, numerous extensions and alternative models have been proposed, relaxing some of the assumptions of Black and Scholes and adding features such as stochastic volatility (Heston, 1993), jumps (Merton, 1976), and more flexible, data-driven approaches (Hutchinson et al., 1994); these extensions are reviewed, with their original references, in Section 2.6.

To better discuss the existing literature, it is important to divide the models in two broad categories: **parametric** and **non-parametric**. Parametric approaches, like the Black–Scholes–Merton model, rely on specific assumptions about the underlying price dynamics, while non-parametric methods define the pricing function directly from observed data, providing the relaxation of assumptions, and a better flexibility.

This chapter provides an overview of the main option pricing models, with particular emphasis on the Black–Scholes framework and its role as a benchmark for following developments.

2.2 Parametric and non-parametric models for option pricing

To structure the discussion of option pricing models, it is essential to draw a distinction between parametric and non-parametric approaches, two foundational concepts of statistical learning. Before doing so, it is worth answering the question of what a model is, since this notion lies at the core of the thesis. In short, a statistical model is a set of probability distributions on a sample space (McCullagh, 2002).
In the general statistical learning framework of James et al. (2013), the relationship between a response variable Y and a set of predictors X can be written as

$$Y = f(X) + \varepsilon, \tag{2.1}$$

where $f(\cdot)$ is an unknown function and ε is a random error term. The aim is to estimate $f(\cdot)$ from observed data, either for prediction or for inference (James et al., 2013). Different approaches arise depending on how this function is specified and estimated.
The parametric approach specifies a functional form for $f(\cdot)$ *a priori*, giving an approximation $\hat{f}(\cdot)$. The number of parameters is then fixed regardless of the sample size: new data refine the parameter estimates but do not alter the structure of the model. A simple example is

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2^2 + \cdots + \beta_p x_p^p + \varepsilon, \tag{2.2}$$

where the coefficients $\beta_0, \beta_1, \ldots, \beta_p$ form a finite set of parameters to be estimated.
It is worth noting that, even when such a model is nonlinear in the regressors, it remains linear in its parameters. This property allows the coefficients to be estimated in closed form via **ordinary least squares (OLS)** or **maximum likelihood estimation (MLE)**. However, not all parametric models share this property. A GARCH model (Bollerslev, 1986), for instance, is nonlinear in its parameters and is estimated numerically by maximum likelihood; the Black–Scholes model is likewise nonlinear, although it is a no-arbitrage pricing formula that is typically calibrated or inverted for implied volatility rather than fitted to data (Hull, 2012). In all these cases, the problem reduces from estimating an unknown, infinite-dimensional function to estimating a finite set of $p + 1$ parameters.
To illustrate how a simple parametric model works, consider the following example:

Example 4. Consider an investor who wants to price options from historical data. Under a parametric model the investor assumes a specific relationship between the inputs (such as the underlying price and the time to maturity) and the option price, and then estimates the parameters. Suppose the chosen function is

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2^2 + \varepsilon, \tag{2.3}$$

where Y is the option price, x_1 the price of the underlying asset, and x_2 the time to maturity. Using the historical observations in Table 2.1,
and substituting these values into the model gives the system in Equation (2.4):

Option price Y	Underlying price x_1	Time to maturity x_2
5	100	1
25	110	2
55	120	3

$$\begin{cases} 5 = \beta_0 + 100\beta_1 + 1\beta_2 \\ 25 = \beta_0 + 110\beta_1 + 4\beta_2 \\ 55 = \beta_0 + 120\beta_1 + 9\beta_2 \end{cases} \tag{2.4}$$

Table 2.1: Historical observations used for parameter estimation.

Suppose now that a new observation becomes available, with $x_1 = 115$ and $x_2 = 2$. The investor obtains a price prediction by substituting these values into the estimated equation:

$$\hat{Y} = -50 + 0.5 \times 115 + 5 \times 4 = 27.5. \tag{2.5}$$

The arrival of this observation changes neither the functional form nor the estimates $\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2$. The form was fixed a priori, and new data serve only as inputs to the prediction, not as information that reshapes the model itself (Figure 2.1).

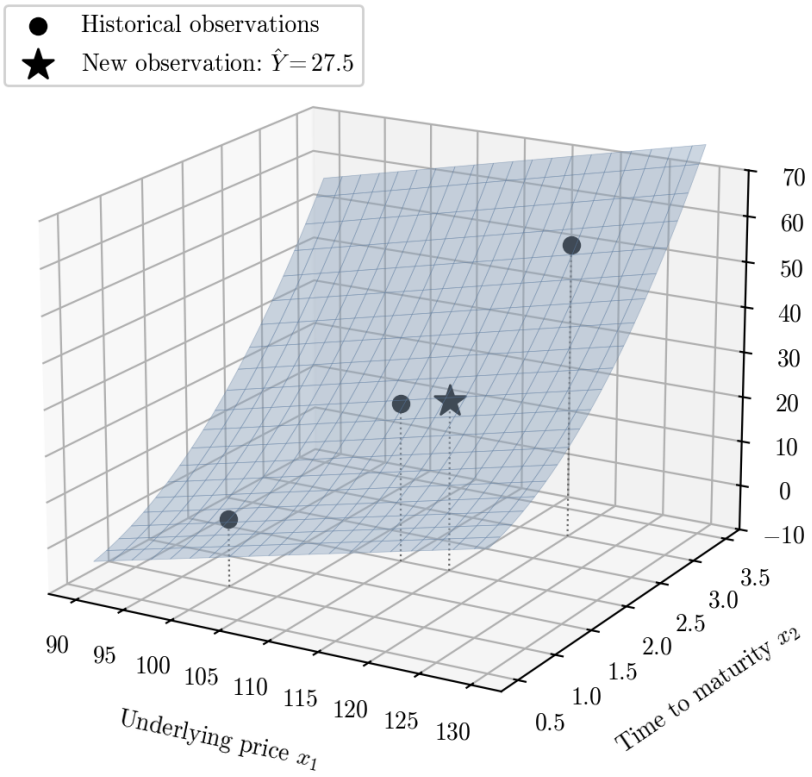


Figure 2.1: The parametric model $\hat{Y} = -50 + 0.5\,x_1 + 5\,x_2^2$, fitted to the three historical observations of Table 2.1 (circles). With three points and three parameters the surface interpolates the data exactly, so the fitted points lie on it. The star marks the price predicted for a new observation, $(x_1, x_2) = (115, 2)$, which the model places at $\hat{Y} = 27.5$ without any change to its functional form or to the estimated coefficients.

Although parametric models are simpler to implement, their main limitation is that an incorrect choice of functional form can make the whole model misspecified. A particular case is *underfitting*, which occurs when the model is too simple to capture the data-generating process. The natural remedy is a more flexible function, but a balance must be found, since greater complexity can lead to *overfitting*: the model then adapts excessively to the data, fitting the noise as well as the signal, and predicting poorly out of sample.
Non-parametric models, by contrast, do not fix the form of $f(\cdot)$ a priori but recover it directly from the observed data. This makes them more flexible, but at a higher computational cost and with a greater need for data: accurate estimates require large samples. The risk of overfitting is also higher than in the parametric case, although it can be managed through techniques such as regularisation and early stopping; if the model is not properly controlled, it adapts too closely to the noise and predicts poorly on the test data (Goodfellow et al., 2016; James et al., 2013).

2.3 Towards the Black–Scholes model: stochastic processes

Before discussing the Black–Scholes–Merton model, it is necessary to introduce *stochastic processes*, since they are a modern approach to describing stock price movements and a fundamental concept to understand option pricing. From a theoretical point of view, a stochastic process is a collection of random variables in a probability space indexed by time (Øksendal, 2003; Shreve, 2004). This means that variables that change over time with uncertainty are *stochastic variables*.
Stochastic processes can be *discrete* or *continuous* in time. A discrete time process evolves at fixed time steps $(0, 1, 2, 3 \dots)$, while a continuous time process is defined for every $t \geq 0$. In finance, continuous time models are preferred because they are mathematically manageable, pricing formulas are easier to derive, and they better reflect the behaviour of asset prices over short intervals.
Among continuous time processes, **Brownian motion** is certainly the most important. It is the foundation of most modern financial models, including the Black–Scholes–Merton framework, and serves as the theoretical basis of the **geometric Brownian motion**. To build it rigorously, it is natural to start from the discrete case and pass to the limit, which is precisely the role of the *scaled symmetric random walk* introduced below. The construction that follows builds Brownian motion as the limit of a simple discrete process, following Shreve (2004, Section 3.2). Consider a *fair* coin toss, which means that the probability of having one side or the other is the same (50-50). The experiment results in the following sequence of random variables, *independent and identically distributed (i.i.d.)*: $\{X_j\}_{j\geq 1}$.
Each random variable X_j is defined as:

$$X_j = \begin{cases} +1, & \text{if the } j\text{-th toss results in Heads,} \\ -1, & \text{if the } j\text{-th toss results in Tails.} \end{cases}$$

Since the coin is fair, the probability of Heads is the same as that of Tails, namely $\frac{1}{2}$.

It follows that the expected value of X_j and its variance are 0 and 1, respectively.
The symmetric random walk is defined as:

$$M_k = \sum_{j=1}^{k} X_j, \quad k = 0, 1, 2, \ldots \tag{2.6}$$

with the starting point being $M_0 = 0$.
At each step, the process moves upward or downward by one, depending on the outcome of the coin toss. Therefore, the trajectory of $\{M_k\}$ represents the cumulative effect of successive random shocks.
The expected value and the variance of M_k are the following:

$$\mathbb{E}[M_k] = \sum_{j=1}^{k} \mathbb{E}[X_j] = 0 \qquad \mathrm{Var}(M_k) = \sum_{j=1}^{k} \mathrm{Var}(X_j) = k \tag{2.7}$$

An important property of the random walk is that it has *independent increments*. For any integers $0 \leq l < m$, the increment:

$$M_m - M_l = \sum_{j=l+1}^{m} X_j \tag{2.8}$$

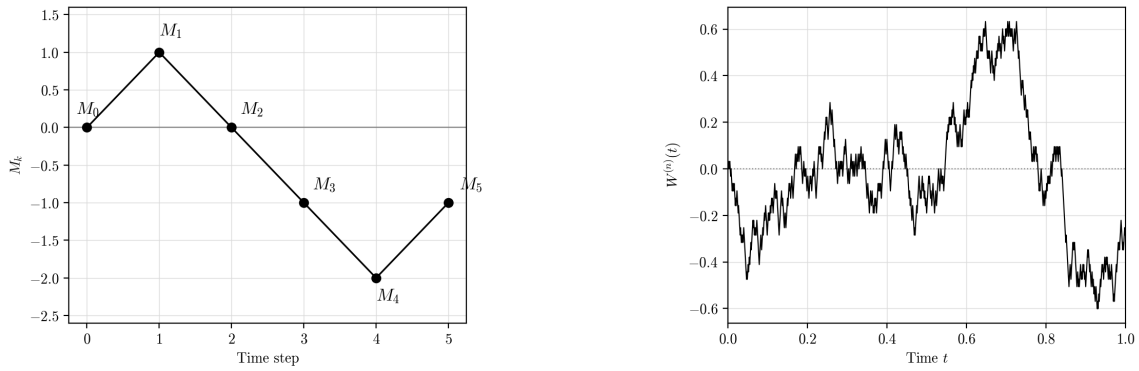
depends only on the outcomes of the coin tosses between times $l + 1$ and m, and is independent of the past.
Moreover, since the expected value of each increment is zero, the process satisfies:

$$\mathbb{E}[M_{k+1} \mid M_1, \ldots, M_k] = M_k, \tag{2.9}$$

which implies that the symmetric random walk is a **martingale**.
A martingale is a stochastic process in which the expected value of the next observation, conditional on the information available up to the present, equals the most recent value, as shown in the previous equation (Shreve, 2004, Sec. 2.4).
The process described above is discrete in time: time steps are integers and shocks take values $+1$ or -1. To approximate the Brownian motion, time steps must be made finer and shocks smaller. This is achieved through the *scaled symmetric random walk*:

$$W^{(n)}(t) = \frac{1}{\sqrt{n}} M_{nt}, \tag{2.10}$$

where n is a positive integer and M_{nt} is the value of the random walk after nt steps. When nt is not an integer, the process is defined by linear interpolation.
Two adjustments are made simultaneously. First, time is accelerated by a factor of n, so the process takes n steps per unit of time. Second, each jump is scaled down by $1/\sqrt{n}$. As n grows, the process moves more frequently but in smaller increments, as illustrated in Figure 2.2.
Despite the jumps shrinking, the accumulated squared movement does not vanish. To see why, consider a simple example: the movements $[-1, +1, +1, -1]$ sum to zero, yet



(a) Symmetric random walk.

(b) Scaled symmetric random walk.

Figure 2.2: Comparison between a discrete symmetric random walk and a scaled symmetric random walk.

their squares sum to $(-1)^2 + (+1)^2 + (+1)^2 + (-1)^2 = 4$. Squaring removes the sign, so cancellation no longer occurs and the trajectory accumulates a nonzero **quadratic variation**.

For the scaled random walk, each jump has size $\pm 1/\sqrt{n}$, so each squared jump contributes $1/n$. Over the interval $[0, t]$ the process makes approximately nt jumps, giving the following quadratic variation:

$$\left[W^{(n)}, W^{(n)}\right]_t = nt \cdot \frac{1}{n} = t. \tag{2.11}$$

The jumps become smaller, but more frequent, so the quadratic variation remains exactly t.

Equally, the variance is t:

$$\mathrm{Var}\big(W^{(n)}(t)\big) = \frac{1}{n}\,\mathrm{Var}(M_{nt}) = \frac{1}{n} \cdot nt = t. \tag{2.12}$$

For example, consider time $t = 0.50$ and the scaling factor $n = 100$. The scaled random walk is then generated by $nt = 100 \times 0.50 = 50$ coin tosses, resulting in the random variable M_{50}. Each coin toss produces $+1$ or -1, so M_{50} ranges from -50 to 50; since 50 is even, the possible values are: $M_{50} \in \{-50, -48, -46, \ldots, 46, 48, 50\}$ This occurs because if h denotes the number of heads, then the number of tails is $50-h$, and therefore: $M_{50} = h - (50 - h) = 2h - 50$.

As h increases by one, M_{50} increases by two, so the range of possible values has steps of two.

According to the scaled equation discussed before, the scaled process is then:

$$W^{(100)}(0.50) = \frac{1}{\sqrt{100}}\,M_{50} = \frac{1}{10}\,M_{50},$$

so, the possible values of the scaled random walk are:

$$W^{(100)}(0.50) \in \{-5.0, -4.8, -4.6, \ldots, 4.6, 4.8, 5.0\}.$$

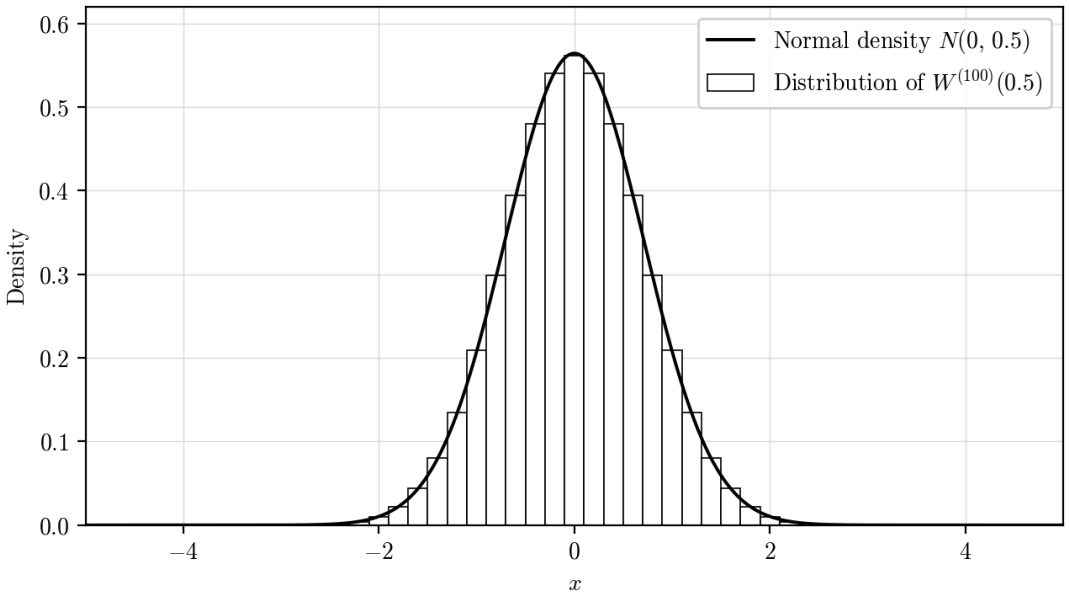


Figure 2.3: The normal density provides a close fit to the histogram of $W^{(100)}(0.50)$.

As shown by Shreve (2004) (Section 3.2.6), the scaled symmetric random walk provides a natural construction of Brownian motion. As $n \to \infty$, the process converges in distribution to a normal random variable. This follows from the **Central limit theorem**, since $W^{(n)}(t)$ is the normalised sum of independent and identically distributed random variables.

More precisely, **Theorem 3.2.1** in Shreve states that: *for any fixed time $t \geq 0$, as $n \to \infty$, the distribution of the scaled random walk $W^{(n)}(t)$ converges to a normal distribution with mean zero and variance t.*

$$W^{(n)}(t) = \frac{1}{\sqrt{n}} M_{nt} \xrightarrow{d} N(0,\, t), \qquad n \to \infty, \tag{2.13}$$

where $N(0, t)$ denotes a normal distribution with mean zero and variance t, having density:

$$f(x) = \frac{1}{\sqrt{2\pi t}}\, e^{-\frac{x^2}{2t}}.$$

As illustrated in Figure 2.3, the normal density provides a close approximation to the histogram of the scaled random walk considered in the example above.

The scaled symmetric random walk thus provides a concrete construction that anticipates the properties of Brownian motion: zero-mean increments, variance equal to t, quadratic variation equal to t, and, in the limit, normally distributed marginals. The next section formalises these properties into a rigorous definition of Brownian motion.

2.4 Introduction to geometric Brownian motion

As discussed in the previous section, Brownian motion can be introduced as the continuous time limit of a scaled symmetric random walk. This construction provides the intuition behind its main properties: independent and zero mean increments, variance proportional to time, and nonzero quadratic variation.
Building on this discrete time intuition, this section provides a more formal introduction to Brownian motion and explains why it plays a central role in financial modelling, especially through its use in geometric Brownian motion.
The Brownian motion describes the random movement of particles suspended in a medium (liquid or gas). It is named after Robert Brown (1827), a Scottish botanist who first observed this phenomenon while analysing the movement of pollen under a microscope. Even though Brown could not explain the physics underlying this motion, his research was fundamental in laying the basis for the mathematical formulation of stochastic processes.
A major step forward was made by Louis Bachelier (1900), in his doctoral thesis "*Théorie de la spéculation*", in which he modelled stock price fluctuations as random processes, independently arriving at the same mathematical structure that would later be formalised as Brownian motion.
In particular, he anticipated the fundamental concepts discussed in the previous section, not by directly mentioning the Brownian motion, but by modelling the random behaviour of the stock prices underlying options.
In 1905, Albert Einstein made one of the greatest contributions, providing a physical explanation for the phenomenon by modelling the random movement of small particles suspended in a fluid as the result of collisions with the surrounding molecules and deriving the probability distribution of their displacements.
Norbert Wiener (1923) later provided a mathematical foundation for these concepts, proving the existence of the Brownian motion as a stochastic process and constructing the probability measure over the space of continuous trajectories. For this reason, the Brownian motion is also known as the **Wiener process** (Figure 2.4).
The Wiener process $\{W_t\}_{t\geq 0}$ is a stochastic process starting at 0, which inherits the following properties from the scaled symmetric random walk:

- $W_0 = 0$ which is the starting point,
- W_t has independent increments,
- $W_t - W_s \sim \mathcal{N}(0, t-s)$ for $t > s$, which means that increments are also normally distributed,
- The sample paths of W_t are continuous in time, but nowhere differentiable, a consequence of the nonzero quadratic variation accumulated along every path[1].
- W_t is a martingale with respect to its natural filtration $\{\mathcal{F}_t\}_{t\geq 0}$, meaning that $\mathbb{E}[W_t|\mathcal{F}_s] = W_s$ for $t > s$.

[1] See (Øksendal, 2003, p. 24).

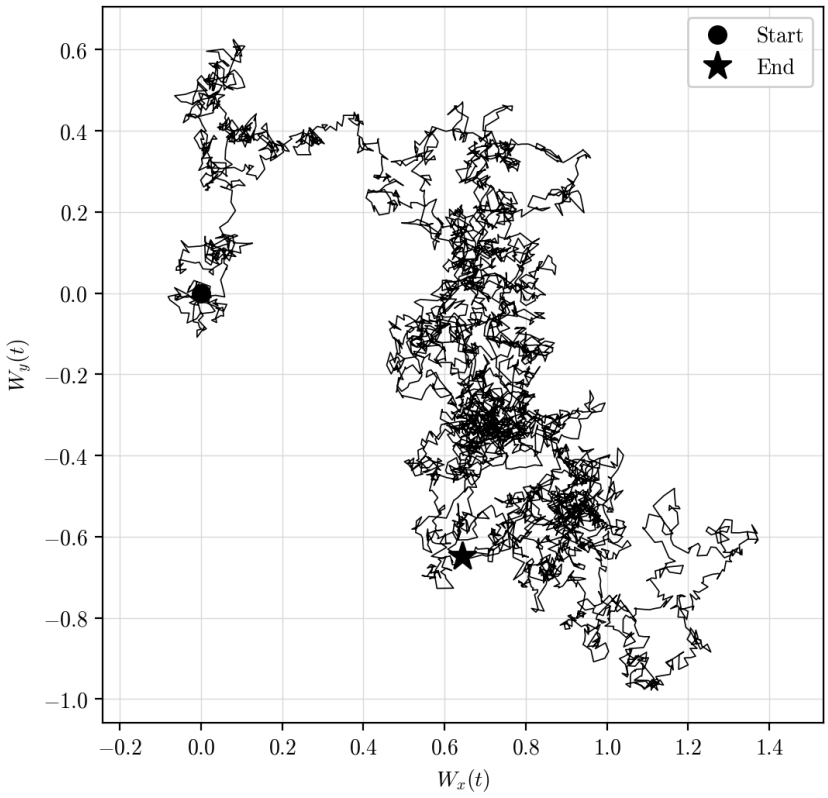


Figure 2.4: Simulation of two-dimensional Brownian motion of a single particle.

Regarding the filtration $\{\mathcal{F}_t\}_{t\geq 0}$, it represents the information accumulated by observing the process up to time t. More precisely, $\mathcal{F}_t$ contains all the information available at time t, and grows over time as new observations are made. The independence of future increments from $\mathcal{F}_t$ implies that, given the current state of the process, its future evolution cannot be predicted from past observations, a property that, in financial models, is related to the weak form of the efficient market hypothesis (Shreve, 2004).
Moreover, the Brownian motion satisfies the Markov property. This means that, conditional on the current value of the process, its future evolution is independent of its past history. In other words, all relevant information for predicting the future behaviour of the process is contained in its current value.
The standard Brownian motion $W(t)$ can be generalised by introducing a *drift* term μ and a *volatility* term σ, yielding the **Brownian motion with drift**:

$$X(t) = \mu t + \sigma W(t), \tag{2.14}$$

where $\mu \in \mathbb{R}$ controls the average direction of the process and $\sigma > 0$ scales the random fluctuations. The process $X(t)$ inherits the properties of $W(t)$: it has independent and normally distributed increments, with $X(t) - X(s) \sim \mathcal{N}(\mu(t-s), \sigma^2(t-s))$ for $t > s$.
While the Brownian motion provides the probabilistic basis, to model stock prices (here assumed to be non-dividend-paying), it is necessary to introduce the geometric Brownian motion.

In ordinary calculus, the chain rule allows one to differentiate a composition of functions. For stochastic processes, the analogous tool is the **Itô-Doeblin formula** (Shreve, 2004), which accounts for the nonzero quadratic variation of Brownian motion.
Let $X(t)$ be an Itô process of the form:

$$dX(t) = \mu(t)\,dt + \sigma(t)\,dW(t), \tag{2.15}$$

and let $f(t,x)$ be a function with continuous partial derivatives f_t, f_x, and f_{xx}. Then:

$$df(t,X(t)) = f_t(t,X(t))\,dt + f_x(t,X(t))\,dX(t) + \frac{1}{2}f_{xx}(t,X(t))\,\sigma^2(t)\,dt. \tag{2.16}$$

The extra term $\frac{1}{2}f_{xx}\,\sigma^2\,dt$, absent in ordinary calculus, arises precisely because $dW(t)^2 = dt$.
The Brownian motion with drift introduced above cannot directly model asset prices, since it can also take negative values. To solve this issue one must consider the exponential of an Itô process, which is strictly positive. Let $X(t)$ be an Itô process and $S(0) > 0$ be the initial asset price: applying the Itô-Doeblin formula to $f(x) = S(0)e^x$ yields the **geometric Brownian motion** (Shreve, 2004, p. 148):

$$S(t) = S(0)\exp\left\{\sigma W(t) + \left(\alpha - \frac{1}{2}\sigma^2\right)t\right\}, \tag{2.17}$$

which in differential form is:

$$dS(t) = \alpha S(t)\,dt + \sigma S(t)\,dW(t), \tag{2.18}$$

where $\alpha = \mu + \frac{1}{2}\sigma^2$ is the instantaneous mean rate of return and σ is the volatility.

2.5 The Black, Scholes, Merton model

The previous sections introduced the statistical foundations and the stochastic properties used to describe the dynamics of asset prices underlying financial derivatives. These concepts formed the basis of the work developed by Black and Scholes (1973) and, independently, by Merton (Merton, 1973) in the early 1970s, which twenty years later led Scholes and Merton to receive the Nobel Prize in economics.
Before introducing the formulas, the discussion will focus on the assumptions at the base of the model, in order to understand its strengths and limitations.
First, the dynamics of stock prices $S(t)$ is modelled after the geometric Brownian motion, discussed in the previous section.
Starting from its equation, divide by $S(0)$ and apply the natural logarithm on both sides:

$$\ln\frac{S(t)}{S(0)} = \sigma W(t) + \left(\alpha - \frac{1}{2}\sigma^2\right)t. \tag{2.19}$$

The left hand side of the equation represents the *logarithmic return*[2] of the stock over the time horizon. The right hand side of the equation is the sum of a deterministic term, $\left(\alpha - \frac{1}{2}\sigma^2\right) t$, and a random term, $\sigma W(t)$.
Since σ is constant, the random term can be rewritten as the *Itô integral*:

$$\sigma W(t) = \int_0^t \sigma \, dW(s).$$

Shreve (2004, Theorem 4.4.9, p. 149) (the Itô integral of a deterministic integrand) shows that the Itô integral of a deterministic integrand $\Delta(s)$ is normally distributed with mean zero and variance $\int_0^t \Delta^2(s)\, ds$.
Setting $\Delta(s) \equiv \sigma$, the following relationship is obtained:

$$\sigma W(t) \sim \mathcal{N}\left(0,\, \sigma^2 t\right). \tag{2.20}$$

Since the sum of a normal random variable and a constant is still a normal random variable (with just shifted mean and variance):

$$\ln \frac{S(t)}{S(0)} \sim \mathcal{N}\left(\left(\alpha - \frac{1}{2}\sigma^2\right) t,\; \sigma^2 t\right). \tag{2.21}$$

Therefore, the log-return over the time horizon t follows a normal distribution. In figure 2.7b the typical distribution of logarithmic returns is illustrated against the normal approximation. As one can notice, the normal approximation provides a good fit, even though the financial literature is actually trying to depart from this assumption (as discussed in the next section).
Second, it is necessary to introduce a method to estimate **volatility** from sample data. Volatility is a measure of uncertainty about the returns of a stock and it usually falls between 15% and 60% annually (from real data observations) (Hull, 2012, p. 303). Note that, to obtain a daily volatility measure, one cannot simply divide the annual volatility by the number of trading days in a year (252 or 253, depending on the assumptions). The standard deviation of log-returns over a time interval of length τ, expressed in years, is: $\sigma_\tau = \sigma\sqrt{\tau}$, therefore the daily measure is obtained by dividing by the **square root**[3] of time.

[2] In quantitative finance, logarithmic returns $\ln(S(t)/S(0))$ are often preferred to simple returns $(S(t) - S(0))/S(0)$. First, they are additive over time: the log-return over $[0, T]$ is the sum of the log-returns over any partition of the interval,which does not hold for simple returns. Second, under geometric Brownian motion log-returns are exactly normally distributed, whereas simple returns are not. Third, for small returns the two measures are numerically close, since $\ln(1+x) \approx x$ for x near zero by a first-order Taylor expansion.

[3] The square root of time scaling is a direct consequence of the assumption that log-returns are serially uncorrelated. Under this null hypothesis, the variance of cumulative n-period log-returns equals n times the variance of one-period returns, so that the *variance ratio* $\widehat{\mathrm{VR}}(n) = \hat{\sigma}^2_{cum}/n\,\hat{\sigma}^2_{\Delta}$ should be close to one, where $\hat{\sigma}^2_{\Delta}$ is the variance estimated from one-period returns and $\hat{\sigma}^2_{cum}$ the variance estimated from non-overlapping n-period log-returns. Values above one indicate positive autocorrelation, values below one indicate negative autocorrelation. Lo and MacKinlay (1988) show that, under the random-walk null, the variance ratio is asymptotically normally distributed, $\widehat{VR}(n) \sim \mathcal{N}\left(1,\, \frac{2(n-1)}{Tn}\right)$, where Tn denotes the total number of one-period returns, so that the standardised statistic $z(n) = \left(\widehat{VR}(n) - 1\right)/\sqrt{2(n-1)/(Tn)}$

Following Hull (2012, p. 304), volatility can be estimated from historical data by the **sample standard deviation estimator**:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(u_i - \bar{u})^2}, \tag{2.22}$$

where $u_i = \ln S_i/S_{i-1}$, with $i = (1, 2, \ldots, n)$, and $\bar{u}$ is the sample mean of u_i. Since u_i have standard deviation $\sigma\sqrt{\tau}$, the variable s is itself an estimate of $\sigma\sqrt{\tau}$, and the annualized volatility is therefore estimated as $\hat{\sigma} = s/\sqrt{\tau}$. Also, it can be demonstrated that the standard error of this estimator is approximately $\hat{\sigma}/\sqrt{2n}$. The choice of the sample size n involves a trade-off: longer samples reduce the standard error of the estimate, but include older observations that may no longer be representative of current market conditions. As a practical rule of thumb, Hull suggests using daily closing prices over a time frame comparable to the maturity of the valued option.
Third, market assumptions are introduced to derive the Black–Scholes–Merton differential equations.
According to Hull (2012, p. 309), the following assumptions are considered:

1. the stock price follows the stochastic process described in the previous section, with drift μ and volatility σ constant;
2. short selling of securities is permitted, with full use of the proceeds;
3. the stocks underlying the options pay no dividends during the life of the contract;
4. there are no arbitrage opportunities, i.e., it is not possible to obtain a riskless profit;
5. securities are perfectly divisible and there are no transaction costs;
6. trading takes place continuously in time;
7. the risk-free rate r is constant and the same for all maturities.

These assumptions make it possible to construct a *riskless portfolio*, which provides the main intuition behind the model. The portfolio consists of a *short position* in the option and a *long position* in the underlying stock. The two positions are held in a way that their price movements compensate each other in any short time interval: since the option price is a function of the stock price, the two are perfectly correlated over infinitesimal time intervals, and an appropriate combination of the two positions yields a portfolio whose value is locally deterministic (since the stochastic component is cancelled out). The resulting portfolio is riskless, so by the no-arbitrage assumption it must earn the risk-free rate r. This condition leads directly to the Black–Scholes–Merton differential equation.

follows a standard normal distribution. The null is rejected at the 5% level whenever $|z(n)| > 1.96$. Empirical studies on equity returns have documented departures from this null, indicating that the constant volatility assumption underlying geometric Brownian motion does not hold exactly in financial data.

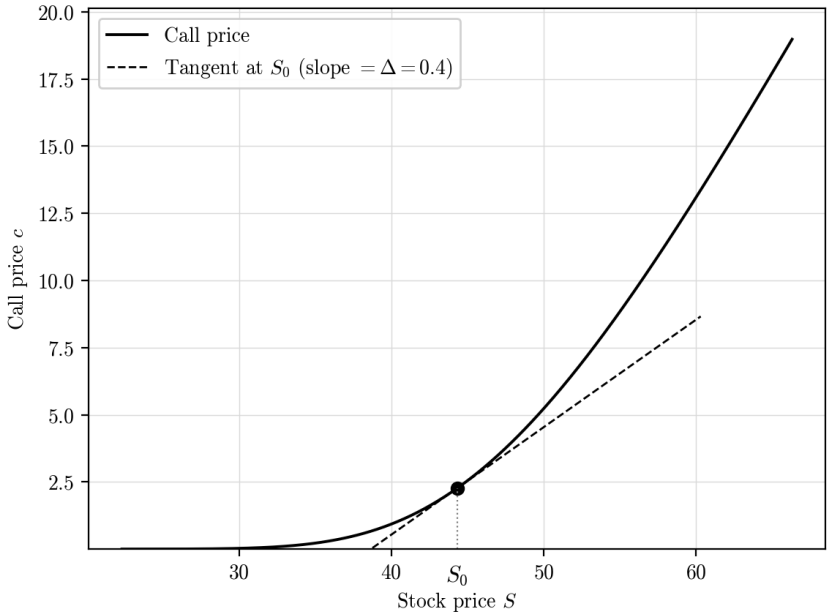


Figure 2.5: Relationship between the call price and the stock price. The slope of the tangent at S_0 corresponds to the delta of the option, here equal to 0.4. Source: own computation following Hull (2012, p. 308).

To illustrate the idea numerically, consider a European call whose price c changes by $\Delta c = 0.4\,\Delta S$ in response to a small change ΔS in the underlying stock price (Hull, 2012, p. 304). As shown in Figure 2.5, the coefficient 0.4 corresponds to the slope of the tangent to the call price curve at the current stock price S_0, and is the *delta* of the option. A portfolio composed of a long position in 0.4 shares and a short position in one call is then insensitive to small movements in S: any gain on the stock is exactly offset by the loss on the short call position, and vice versa. The value of such a portfolio over a short interval is therefore known with certainty. This riskless property holds only locally, since the delta depends on S and on time, and changes as the option moves towards maturity. Maintaining a riskless portfolio therefore requires *continuous rebalancing* of the proportions of stock and option held.
Following Hull (2012, p. 309), the Black–Scholes–Merton differential equation is derived starting from the stochastic process introduced for the underlying asset:

$$dS = \mu S\,dt + \sigma S\,dW. \tag{2.23}$$

Consider the price of the derivative as a function $f(S,t)$ of the stock price and time; applying the Itô–Doeblin formula to f yields:

$$df = \left(\frac{\partial f}{\partial S}\mu S + \frac{\partial f}{\partial t} + \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right)dt + \frac{\partial f}{\partial S}\sigma S\,dW. \tag{2.24}$$

The random terms in equations (2.23) and (2.24) are driven by the same Brownian motion W, meaning that they share the same randomness. This means that the Wiener process can be eliminated by building the riskless portfolio.

Now, consider the portfolio formed by a short position in one option and a long position in $\partial f/\partial S$ shares of the underlying stock (-1 derivative + $\partial f/\partial S$ shares). Its value is:

$$\Pi = -f + \frac{\partial f}{\partial S} S, \tag{2.25}$$

and its variation over a small interval dt is:

$$d\Pi = -df + \frac{\partial f}{\partial S}\, dS.$$

Substituting equations (2.23) and (2.24) into this expression, the stochastic terms in dW and the μ terms cancel out, leaving the equation below:

$$d\Pi = \left(-\frac{\partial f}{\partial t} - \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2 \right) dt. \tag{2.26}$$

The portfolio is therefore locally deterministic: its evolution over the next instant is known with certainty.
A locally riskless portfolio must, by the no-arbitrage assumption, earn the risk-free rate of return r, so that

$$d\Pi = r\,\Pi\, dt.$$

Substituting equations (2.25) and (2.26) into this condition and rearranging the terms gives the **Black–Scholes–Merton partial differential equation**,

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} = rf. \tag{2.27}$$

Equation (2.27) admits many solutions, each corresponding to a different derivative whose payoff depends on S. The specific contract being priced is selected by imposing appropriate boundary conditions. For a European call option with strike K and maturity T, the relevant boundary condition is $f(S,T) = \max(S-K, 0)$; for a European put, $f(S,T) = \max(K-S, 0)$.
It is important to emphasise that the drift μ of the underlying does not appear in equation (2.27). Since μ reflects the risk preferences of investors, this implies that the price of the derivative is independent of such preferences. In general, higher risk aversion corresponds to a higher expected return on the stock, so a pricing model that relied on μ would also require explicit assumptions on investor preferences. The fact that μ drops out of equation (2.27) avoids this issue entirely: since risk preferences do not enter the equation, they cannot affect its solution, and any set of preferences can be assumed when computing the price. The most convenient choice is to assume that all investors are risk-neutral. In such a hypothetical world, the expected return on every asset equals the risk-free rate r, and any cash flow can be valued by discounting its expected value at r. This approach, known as *risk-neutral valuation*, simplifies the analysis of derivatives and yields the same price that would be obtained under risk aversion, since the two effects (the higher expected growth of the stock and the higher discount rate required by investors) offset each other exactly.

With the European call and put options boundaries mentioned above, it is possible to achieve their pricing formulas:

$$c = S_0\, N(d_1) - Ke^{-rT}\, N(d_2), \tag{2.28}$$

$$p = Ke^{-rT}\, N(-d_2) - S_0\, N(-d_1), \tag{2.29}$$

where $N(\cdot)$ is the cumulative distribution function of the standard normal distribution, and the terms d_1 and d_2 are defined as the following:

$$d_1 = \frac{\ln(S_0/K) + \left(r + \frac{1}{2}\sigma^2\right) T}{\sigma\sqrt{T}}, \qquad d_2 = d_1 - \sigma\sqrt{T}. \tag{2.30}$$

The variables c and p denote the prices of the European call and put options, respectively, S_0 is the stock price at time zero, K is the strike price, r is the continuously compounded risk-free rate, σ is the volatility of the underlying, and T is the time to maturity. The cumulative distribution function $N(x)$ represents the probability that a standard normal random variable takes a value less than or equal to x. In the Black–Scholes formula, $N(d_2)$ corresponds to the risk-neutral probability that the call option will be exercised at maturity, that is, the probability that $S_T > K$ under the measure in which the underlying drifts at the risk-free rate; equally, $N(-d_2)$ is the risk-neutral probability of exercise for the put. The term $N(d_1)$ coincides instead with the delta of the option $\partial c/\partial S_0$: it measures the sensitivity of the call price to small changes in the underlying and, as shown in Figure 2.5, corresponds to the number of shares required to construct a riskless hedging portfolio.

To verify that the derivation of the Black–Scholes–Merton formulas is correct, Hull shows their behaviour when extreme parameter values are considered. When the call is deep in the money, that is, when the stock price S_0 becomes very large, exercise is certain, and the call price tends to the value of a forward contract: $S_0 - Ke^{-rT}$. This is consistent with equation (2.28): as $S_0 \to \infty$, both d_1 and d_2 tend to $+\infty$, $N(d_1)$ and $N(d_2)$ tend to 1, and $c \to S_0 - Ke^{-rT}$. The European put price instead goes to zero, as expected, since the right to sell a stock at K has no value when the stock is worth significantly more. When the volatility approaches 0 the underlying becomes deterministic, growing at the risk-free rate r and reaching $S_0 e^{rT}$ at maturity. The call payoff is then $\max(S_0 e^{rT} - K, 0)$, which, discounted at the risk-free rate, gives the following equation:

$$e^{-rT} \max(S_0 e^{rT} - K, 0) = \max(S_0 - Ke^{-rT}, 0).$$

This is consistent with the limit of equation (2.28) as $\sigma \to 0$, which confirms that the closed-form solution behaves correctly at the boundary of the domain.

To clarify these concepts, consider the example in Hull (2012, p. 315): the current stock price is $S_0 = 42$, the strike price is $K = 40$, the continuously compounded risk-free rate is $r = 10\%$ yearly, the yearly volatility is $\sigma = 20\%$, and the option expires in six months, so that $T = 0.5$. Substituting these values into equation (2.30) gives:

$$d_1 = \frac{\ln(42/40) + \left(0.10 + \frac{1}{2} \cdot 0.20^2\right) \cdot 0.5}{0.20\sqrt{0.5}} = 0.7693, \qquad d_2 = d_1 - 0.20\sqrt{0.5} = 0.6278,$$

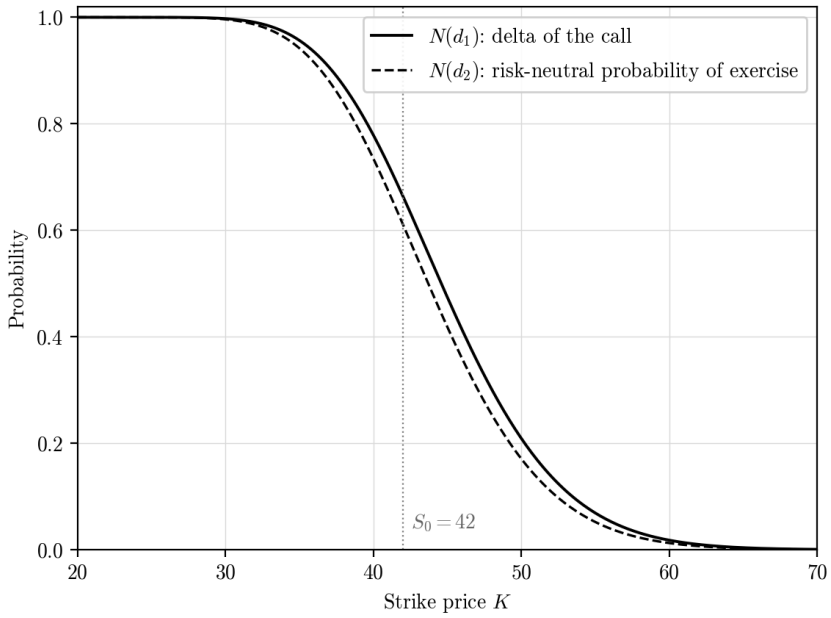
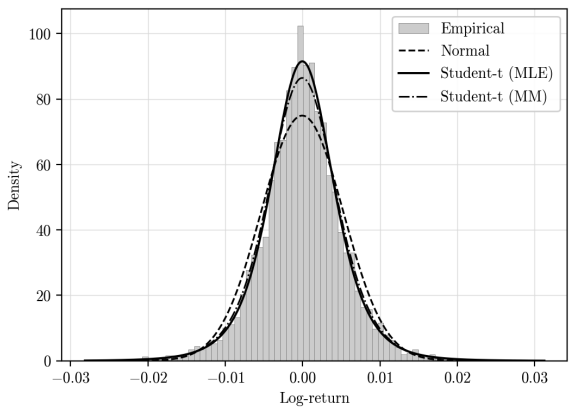


Figure 2.6: Behaviour of $N(d_1)$ and $N(d_2)$ as functions of the strike price K, computed with the parameters of the example in Hull (2012, p. 315): $S_0 = 42$, $r = 10\%$, $\sigma = 20\%$, $T = 0.5$. The two curves are both decreasing in K and bounded between 0 and 1. The inequality $N(d_1) \geq N(d_2)$ holds for every strike, since $d_1 = d_2 + \sigma\sqrt{T} > d_2$. The vertical dotted line marks the current stock price S_0. Source: own computations.

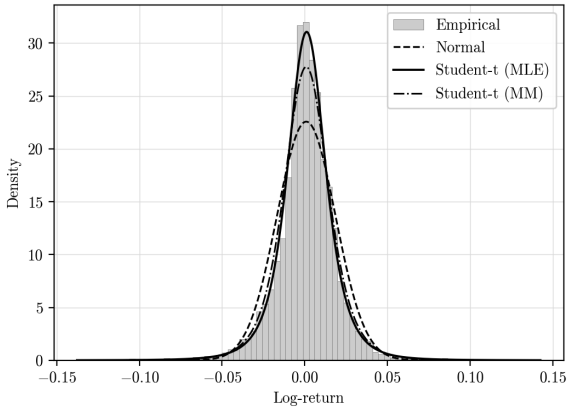
from which $N(d_1) = 0.7791$ and $N(d_2) = 0.7349$. The discounted strike is $Ke^{-rT} = 40 \cdot e^{-0.05} = 38.049$. Substituting into equations (2.28) and (2.29) yields a European call price of $c = 4.76$ and a European put price of $p = 0.81$.
In conclusion, the Black–Scholes–Merton model discussed provides a closed-form pricing formula, that has represented the standard in both academic and professional contexts for more than fifty years. However, it is empirically shown that several of its assumptions do not hold in the real world. In particular, the constant volatility hypothesis and the normality of log-returns are often inconsistent with real data observations. For this reason, traders usually invert the pricing formula and recover the value of σ implicit in observed option prices, a quantity known as *implied volatility* and discussed in the following section.

2.6 Parametric extensions of the BSM model

Since real world data often contradict the assumptions of the Black–Scholes–Merton model, the financial literature has produced a substantial body of work aimed at extending the model in different directions.
The first issue concerns the assumption of normality of log-returns. As shown in Figure 2.7, the empirical distribution of daily log-returns on equity indices and foreign exchange rates is different from the normal curve. In particular, the empirical distributions exhibit heavier tails than the normal, meaning that large movements, both positive and



(a) EUR/USD daily log-returns.



(b) FTSE MIB daily log-returns.

Figure 2.7: Empirical distribution of daily log-returns compared to the normal (Gaussian) and Student's t densities. The Student's t is fitted using both the method of moments (MM) and maximum likelihood estimator (MLE). Source: own computations.

negative, occur much more frequently than the model predicts. This phenomenon is captured by an *excess kurtosis* above the value of 3 implied by the normal distribution, and is often modelled by alternative distributions with fatter tails, such as the *Student's t* distribution.

Additionally, equity markets empirical distribution presents *negative skewness*: large negative returns are more frequent than large positive ones, a feature that the symmetric normal distribution cannot reproduce. On foreign exchange markets, instead, the distribution remains more symmetric, with heavy tails on both sides.

Corrado and Su (1996) propose an extension of the Black–Scholes–Merton formula that accounts for these market features. They apply a *Gram-Charlier* series expansion to the normal density of standardised log-returns, obtaining a closed-form option pricing formula that adds two correction terms reflecting non-normal skewness and kurtosis. Their empirical results on the S&P500 index options show that the skewness and kurtosis adjusted formula reduces the deviation between theoretical and observed prices, in particular for deep in-the-money and deep out-of-the-money contracts where Black–Scholes–Merton exhibits the largest errors.

Another extension of the Black–Scholes–Merton framework is provided by Heston (1993), who proposes a continuous-time model in which the volatility of the underlying is itself a stochastic process. The stock price S_t and its instantaneous variance v_t are both modelled under the following differential equations:

$$dS_t = \mu S_t\, dt + \sqrt{v_t}\, S_t\, dW_t^{(1)}, \tag{2.31}$$

$$dv_t = \kappa(\theta - v_t)\, dt + \sigma\sqrt{v_t}\, dW_t^{(2)}, \tag{2.32}$$

where κ is the speed of mean reversion of the variance towards its long-run level θ, σ is the volatility of volatility (when equal to 0, volatility is deterministic), and the two Brownian

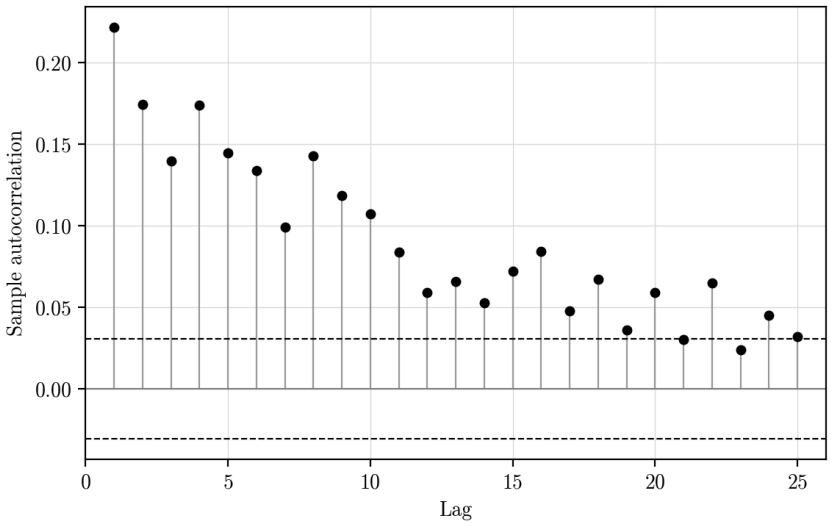
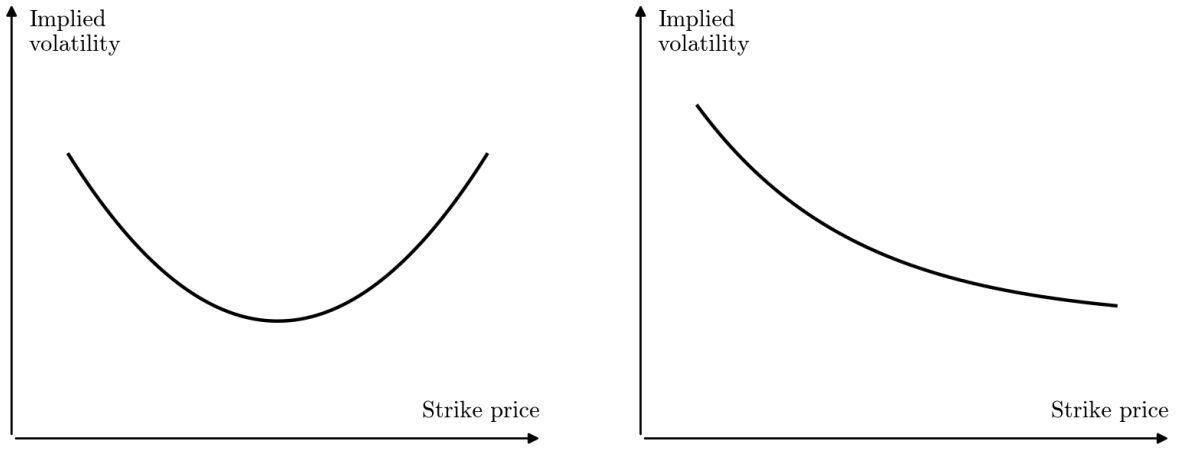


Figure 2.8: Sample autocorrelation function of squared log-returns. Several lags exceed the asymptotic confidence bands (dashed lines), indicating significant serial correlation in volatility. This is a feature commonly referred to as volatility clustering. Source: own computations.

motions $W^{(1)}$ and $W^{(2)}$ have constant correlation ρ. The instantaneous variance follows a square-root process of the type introduced by Cox et al. (1985), which guarantees that v_t remains non-negative. The key feature of the model is the correlation parameter ρ: it allows volatility to react to movements in the underlying, the model generates skewness and excess kurtosis in the distribution of S_t, and reproduces patterns in option prices that the Black–Scholes–Merton model cannot. Heston (1993) derives a closed-form solution for European call options through characteristic functions, which can be evaluated numerically with little computational effort. The main cost of this flexibility is the increased number of parameters that must be estimated, typically through calibration on observed option prices.

Another solution to the constant volatility assumption issue comes from the *GARCH* models. Financial returns exhibit *volatility clustering*, which is the tendency for periods of high volatility to be followed by other periods of high volatility. While the log-returns are considered close to be serially uncorrelated, their squares are not: Figure 2.8 shows the sample autocorrelation function of squared log-returns on an example financial series, where several lags exceed the confidence bands corresponding to the null of no autocorrelation. The same conclusion is confirmed by the *Ljung-Box* test applied to squared returns, which rejects the null hypothesis of zero autocorrelation. This pattern is inconsistent with the constant volatility assumption of Black–Scholes–Merton.

To account for this, Engle (1982) introduces the *Autoregressive Conditional Heteroskedasticity (ARCH)* model, in which the variance of the return at time t is a function of past squared shocks. Bollerslev (1986) generalises this construction by allowing past conditional variances to enter the variance equation as well, obtaining the *Generalised ARCH (GARCH)* model. In the GARCH(1,1) model, the conditional variance h_t of the log-return



(a) Volatility smile. **(b)** Volatility skew (or smirk).

Figure 2.9: Typical shapes of the implied volatility as a function of the strike price, for a given maturity. Panel (a) shows the symmetric smile observed in foreign exchange option markets, where implied volatility is lower at the money and higher for both in-the-money and out-of-the-money strikes. Panel (b) shows the downward-sloping skew observed in equity option markets since the 1987 crash, where implied volatility decreases monotonically with the strike. Adapted from Hull (2012).

ε_t evolves according to:

$$\varepsilon_t \mid \mathcal{F}_{t-1} \sim \mathcal{N}(0, h_t), \qquad h_t = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 + \beta_1 h_{t-1}, \tag{2.33}$$

with $\alpha_0 > 0$, $\alpha_1 \geq 0$, $\beta_1 \geq 0$, and $\alpha_1 + \beta_1 < 1$ to ensure stationarity. The current variance depends on both the most recent squared innovation, which captures the impact of new information, and the previous conditional variance, which captures the persistence of volatility through time. The GARCH(1,1) specification reproduces both volatility clustering and the leptokurtosis of the unconditional return distribution. Although the model is set in discrete time, it can be embedded in an option pricing framework following Duan (1995), who derives a risk-neutral version of the GARCH process and proposes Monte Carlo simulations to compute option prices. Unlike Heston, GARCH-based option pricing does not admit a closed-form solution and relies on simulation.

The empirical features just discussed motivate a common practical use of the Black–Scholes–Merton model: rather than using it to price options, market participants often invert the formula to extract the value of σ that, when plugged into equation (2.28), reproduces the observed market price of an option. This quantity is known as the **implied volatility**. If the model held exactly, the implied volatility extracted from options on the same underlying would be a single constant, independent of strike and maturity. Empirical evidence instead shows that implied volatility, plotted against the strike price for a given maturity, exhibits **volatility smiles**.

As shown in Figure 2.9, the shape of the smile differs across markets. For options on foreign exchange rates, the implied volatility curve is approximately symmetric, with low

values at the money and higher values for both low and high strikes. For the equity market, the pattern is more asymmetric: the implied volatility decreases as the strike increases, assuming a *volatility skew*. These two patterns are direct consequences of the characteristics of the distributions of log-returns. FOREX log-returns are more symmetric but display heavy tails on both sides, which translates into higher implied volatilities for both deep in-the-money and deep out-of-the-money options. Equity log-returns, by contrast, have also negative skewness: large negative movements are much more frequent than large positive ones, so out-of-the-money put options, which provide protection against such movements, are priced by the market at a premium. Since these options correspond to low strikes, their higher prices translate into higher implied volatilities, generating the downward shape of the equity skew.

All these extensions share the same logic: they add structure to the parametric specification, at the cost of greater complexity. The next chapter takes a different route, in which the pricing function is learned directly from the data, without imposing log-normality, constant volatility, or any other distributional assumption.

Chapter 3

Neural Network models

3.1 Non-parametric approaches

As discussed in Section 2.2, pricing models can be divided into parametric and non-parametric. The Black–Scholes–Merton model, together with the extensions discussed in Section 2.6, belongs to the parametric class: the functional form is specified a priori, based on assumptions about the dynamics of the underlying, and only a small set of parameters has to be estimated. Non-parametric approaches reverse this logic, recovering the functional form directly from the observed data. Once estimated, the model becomes the pricing function itself and can be used to price options on new data, in the same way as a closed-form formula.
This transition from parametric to non-parametric models is motivated by the mismatch between empirical data and the Black–Scholes assumptions discussed in Section 2.6. Since non-parametric models do not assume the normality of log-returns or constant volatility, they avoid the specification errors that arise when the process of the underlying is misspecified. Because the pricing function is not tied to any particular model of the underlying, it can also accommodate a wider range of dynamics than a formula derived under a single set of assumptions.
This flexibility comes at a price. Non-parametric models require large amounts of historical data to perform reliably, which makes them less suitable for thinly traded derivatives. Moreover, when the dynamics of the underlying are well understood, a correctly specified parametric model remains the better choice: if the Black–Scholes assumptions held exactly, the Black–Scholes formula would outperform any non-parametric estimator, which could at best approximate it (Hutchinson et al., 1994).

3.2 Neural Networks: comprehensive overview

The non-parametric approach discussed in the previous section is, in practice, carried out by machine learning algorithms. Among these, **Artificial Neural Networks (ANNs)** are one of the most interesting and discussed families, and the one on which the empirical work of this thesis focuses. As Haykin (1999) puts it, a neural network is a machine designed to model the way the brain performs a particular task or function of interest; in

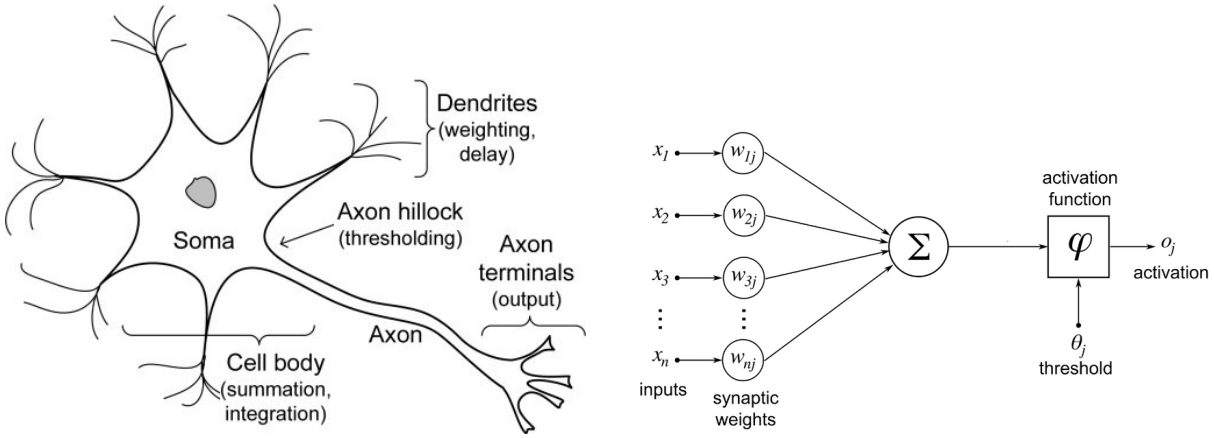
order to implement it, electronic components or digital software are used to simulate how the brain learns and works. Haykin defines it more precisely as an *adaptive machine*, that is, "*a massively parallel distributed processor made up of simple processing units*" having a natural propensity for storing experiential knowledge and making it available for use. Two fundamental properties characterise such system:

1. knowledge is acquired from the environment through a learning process;

2. the acquired knowledge is stored in the strength of the interneuron connections, that is, in the weights.

The learning process is carried out by a learning algorithm, discussed in Section 3.5. These algorithms are continuously developed by the scientific community in order to improve training performance and speed of execution.
Since these models draw their inspiration from the brain, a parallel between the two is drawn throughout most of the literature (Goodfellow et al., 2016; Haykin, 1999). The human brain is made up of billions of *neurons*, small cells that form the basic units of the nervous system and whose function is to transmit electrical signals in response to stimuli and to store information. A biological neuron receives signals from other neurons through its *dendrites*, accumulates them in the cell body, and when the combined stimulation exceeds a threshold, fires an electrical impulse along its *axon* towards the neurons to which it is connected. These connections occur at the *synapses*, whose strength is not fixed but changes with experience, and it is in this adjustment of synaptic strength that learning is usually understood to take place (Haykin, 1999). The artificial neuron reproduces this scheme in a simplified form: the incoming signals become the **input** values, the synaptic strengths become the **weights**, the integration in the cell body becomes a **weighted sum**, and the firing threshold becomes the **activation function** that determines the output of the unit. The first mathematical version of this mechanism was given by McCulloch and Pitts (1943), who described the neuron as a unit that compares a weighted sum of its inputs against a threshold, producing an output only when that threshold is exceeded. This abstraction remains the building block from which the architectures[1] discussed below are assembled.
However, the analogy should not be pushed too far. The resemblance between an artificial unit and a real neuron is loose, and modern networks are not meant to model how the brain actually works. As Goodfellow et al. (2016) point out, neural network research today is guided mainly by mathematical and engineering considerations. Neuroscience offers inspiration, but it does not drive the design of these models, since the brain is still poorly understood.
Still, the biological metaphor is useful to motivate the terminology and the basic structure of the network. The reason for using neural networks in option pricing lies in their mathematical properties, in particular the universal approximation result discussed below (Cybenko, 1989; Hornik et al., 1989), not in any claim of biological realism.

[1] Goodfellow, Bengio, and Courville define an architecture as the overall structure of the network; how many units and how these connect to each other



(a) Biological neuron **(b)** Artificial neuron

Figure 3.1: The structure of a biological neuron (a) alongside the mathematical model of an artificial neuron (b). Dendrites collect weighted inputs, the cell body integrates them, and the axon hillock fires once a threshold is exceeded; the artificial unit mirrors this by summing the weighted inputs $w_{ij}x_i$ and passing the result through an activation function φ.

Neural networks are not a single model but a broad family of models, which differ in how their units are arranged and connected. The most basic distinction is between **feedforward** and **recurrent** networks. In a feedforward network the information flows in one direction, from the input to the output, without cycles; in a recurrent network some connections feed back on themselves, which makes the output depend also on previous states and suits the processing of sequential data such as time series or text. The networks used in this thesis belong to the feedforward class. Within this class, the present work follows Hutchinson et al. (1994) and concentrates on two architectures: the **multilayer perceptron** and the **radial basis function** network, examined respectively in Sections 3.3 and 3.6. These were two of the three learning networks compared in the original paper, the third being **projection pursuit regression** (Friedman & Stuetzle, 1981), which is not implemented here. Section 3.7 then outlines how the same feedforward principle extends to deeper architectures, which form the basis of the modern field of **deep learning** (LeCun et al., 2015).

3.3 Multilayer Perceptron (MLP)

The **Multilayer Perceptron (MLP)** is one of the simplest and most studied artificial neural network architectures. It belongs to the family of feedforward networks, in which information flows from the input to the output through one or more hidden layers, without forming cycles; this distinguishes the MLP from recurrent networks, in which connections may feed back on themselves to process sequential data.
The MLP is built from elementary units known as **perceptrons**, first proposed by Rosenblatt (1958) as mathematical abstractions of biological neurons. In his seminal paper *The*

Input Pattern		Number of "ON" bits in input		Output Pattern
00	→	0	→	0
01	→	1	→	1
10	→	1	→	1
11	→	2	→	0

Table 3.1: Truth table of the XOR (exclusive or) operation, $y = x_1 \oplus x_2$. The output equals one when exactly one of the two inputs is one.

Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Rosenblatt provided a probabilistic model for how the brain stores information and forms new neural connections to recall and generalise learned patterns. The single perceptron, however, is severely limited in its expressive power: as shown by Minsky and Papert (1969), it cannot represent functions that are not linearly separable.
A classical example of this limitation is the **XOR problem** (exclusive or), discussed in Chapter 6 of Goodfellow et al. (2016). The problem takes two binary inputs $x_1, x_2 \in \{0, 1\}$ and defines the output y as equal to 1 if and only if exactly one of the two inputs is 1, and 0: no linear function of the inputs can correctly classify all four input combinations, since the two classes cannot be separated by a line or a hyperplane in $\mathbb{R}^2$. The Multilayer Perceptron overcomes this limitation by introducing a hidden layer that transforms the input space into a new representation in which the problem becomes linearly separable. This extension yields a class of models that can approximate any continuous function, a property formalised by the universal approximation theorem of Cybenko (1989) and Hornik, Stinchcombe and White (1989)[2].
The following subsections develop the mathematics of the MLP using the XOR problem as an illustrative example, following Goodfellow et al. (2016).

3.3.1 Definition of Input

In the Multilayer Perceptron, the input layer is composed of nodes that represent the data fed into the network. The number of nodes in the input layer corresponds to the number of features that describe each observation. Each observation can be expressed as a vector $\mathbf{x} \in \mathbb{R}^d$, where d is the number of features. The entire dataset of N observations is collected into an input matrix $X \in \mathbb{R}^{N \times d}$, and it is summarised in Table 3.1. In addition, in the XOR problem, the dataset takes the following matrix form:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{4 \times 2}, \tag{3.1}$$

[2]Multilayer Feedforward Networks are Universal approximators: provided a sufficient number of hidden units, a Multilayer Perceptron can approximate any continuous function in the space; thus, MLPs are universal approximators.

where each row corresponds to one observation and each column to one feature, resulting in $N = 4$ observations with $d = 2$ features each. No transformation is applied to the inputs at this stage.

3.3.2 Definition of Hidden Layers

The hidden layers are the core of the MLP, where the input data are processed through a sequence of transformations that allow the network to capture non-linear relationships between inputs and output. In the XOR problem, a single hidden layer with $m = 2$ neurons is sufficient to produce a correct solution.
The computation performed by the hidden layer consists of two steps. First, a linear transformation is applied to the input:

$$Z = XW^{(1)} + \mathbf{1}b^{(1)}, \tag{3.2}$$

where $W^{(1)} \in \mathbb{R}^{d\times m}$ is the weight matrix, $b^{(1)} \in \mathbb{R}^{1\times m}$ is the bias vector, and $\mathbf{1} \in \mathbb{R}^{N\times 1}$ is a column vector of ones which allows to spread the bias to all observations. The resulting matrix $Z \in \mathbb{R}^{N\times m}$ is then passed through a non-linear activation function $\sigma(\cdot)$, applied element-wise:

$$H = \sigma(Z), \tag{3.3}$$

where $H \in \mathbb{R}^{N\times m}$ is the matrix of hidden activations, which serves as input to the output layer.
In the XOR problem, Goodfellow et al. (2016) show that the following weights and biases allow the hidden layer to produce a useful representation:

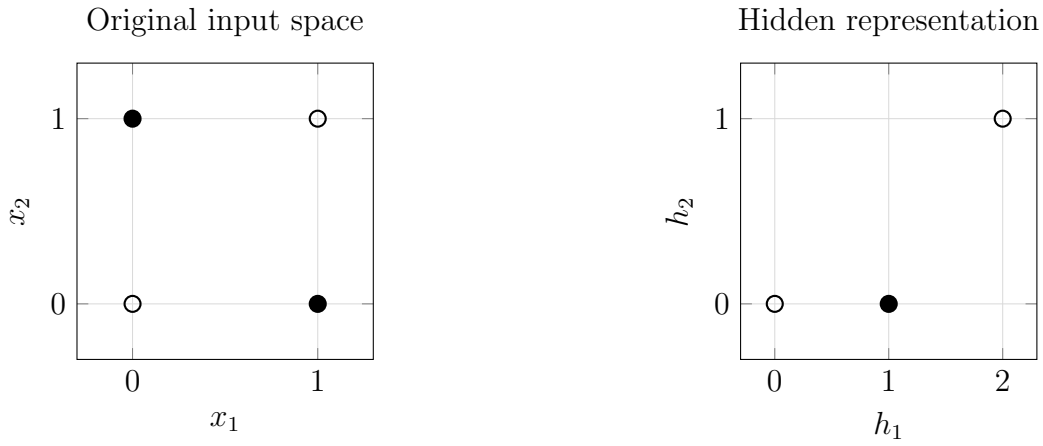
$$W^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \qquad b^{(1)} = \begin{bmatrix} 0 & -1 \end{bmatrix}. \tag{3.4}$$

Substituting the input matrix X from equation (3.1) into equation (3.2) gives the pre-activation matrix:

$$Z = XW^{(1)} + \mathbf{1}b^{(1)} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 & -1 \\ 0 & -1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{3.5}$$

Then, the activation function is applied to Z. Activation functions are mathematical functions that tell when a neuron must activate, by introducing non-linear properties to the network. Since activation functions are many, with different properties, those will be presented in a dedicated section. The activation function used in this example is the rectified linear unit (ReLU), defined as $\mathrm{ReLU}(x) = \max(0, x)$, which sets all negative values to zero and leaves positive values unchanged. It is used here because it yields an exact closed-form solution with integer weights, making the calculations easy to verify. Applying ReLU element-wise to Z gives the hidden activation matrix:

$$H = \mathrm{ReLU}(Z) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{3.6}$$



(a) Original space: the two classes (circles $y = 0$, dots $y = 1$) are not linearly separable.

(b) Hidden space: the two classes are now linearly separable.

Figure 3.2: Effect of the hidden layer transformation on the XOR problem. In the original input space the two classes cannot be separated by a straight line; after the linear transformation and ReLU activation, the representation in the hidden space is linearly separable.

Comparing H with the original input matrix X, a key observation emerges: the two observations $(0, 1)$ and $(1, 0)$, which belonged to different output classes but could not be separated by any straight line in the original input space, are now both mapped to the same hidden representation $(1, 0)$. Meanwhile, $(0, 0)$ maps to $(0, 0)$ and $(1, 1)$ maps to $(2, 1)$, which belongs to the same class as $(0, 0)$ but is now clearly distant from $(1, 0)$. The hidden layer has therefore transformed the input space into a new representation in which the XOR problem is linearly separable, as illustrated in Figure 3.2.
In this new space the two classes are linearly separable, so a linear output layer is sufficient to solve the problem.

3.3.3 Definition of Output

The output layer receives the values of the hidden activations H and applies a final linear transformation to produce the prediction:

$$\hat{Y} = HW^{(2)} + \mathbf{1}b^{(2)}, \tag{3.7}$$

where $W^{(2)} \in \mathbb{R}^{m\times 1}$ is the weight vector connecting the hidden layer to the output neuron, $b^{(2)} \in \mathbb{R}$ is the output bias, and $\hat{Y} \in \mathbb{R}^{N\times 1}$ is the vector of predictions, one for each observation.
In the XOR problem, the same authors show that the following output weights and bias produce the correct solution:

$$W^{(2)} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \qquad b^{(2)} = 0. \tag{3.8}$$

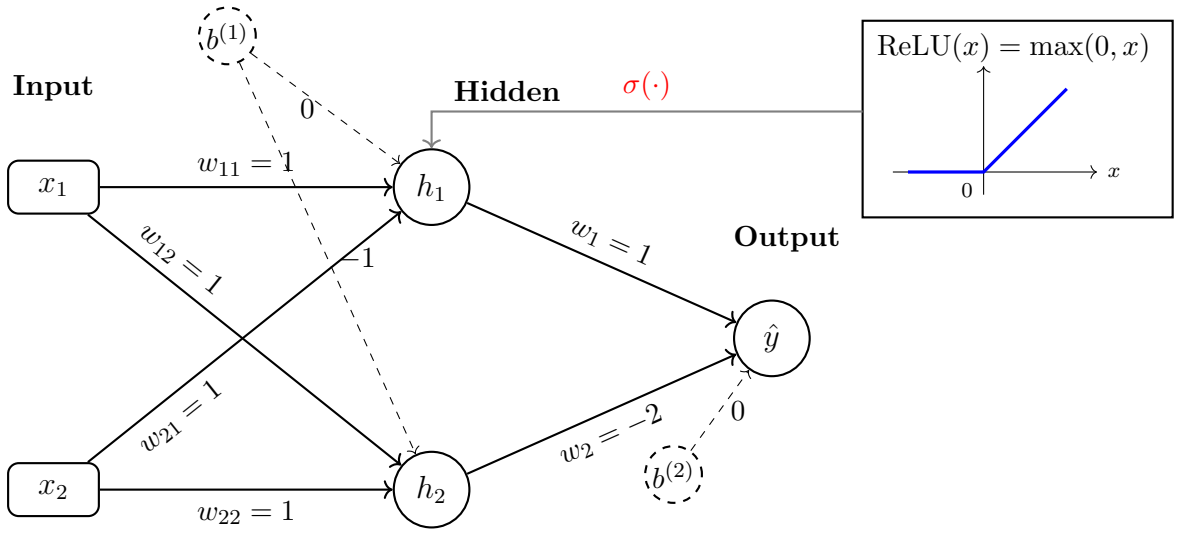


Figure 3.3: Architecture of the MLP used to solve the XOR problem: two input nodes, one hidden layer with two neurons and ReLU activation, and one linear output neuron. The weights correspond to the solution in equations (3.4) and (3.8). The red arrow indicates the ReLU activation function $\sigma(\cdot)$ applied element-wise to the pre-activation matrix Z.

Substituting the hidden activation matrix H from equation (3.6) into equation (3.7) gives:

$$\hat{Y} = HW^{(2)} + 0 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \cdot 1 + 0 \cdot (-2) \\ 1 \cdot 1 + 0 \cdot (-2) \\ 1 \cdot 1 + 0 \cdot (-2) \\ 2 \cdot 1 + 1 \cdot (-2) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{3.9}$$

The predicted output $\hat{Y}$ matches the target y exactly for all four observations, confirming that the MLP with two hidden neurons and ReLU activation solves the XOR problem correctly.
Additionally, the full neural network can be written compactly as the function:

$$f(\mathbf{x}) = W^{(2)\top} \max\big(0,\, W^{(1)\top}\mathbf{x} + b^{(1)}\big) + b^{(2)}, \tag{3.10}$$

which makes explicit that the MLP is a composition of a linear transformation, a non-linear activation, and a second linear transformation.
Note that the output layer in this example is linear and no activation function is applied. This is appropriate for regression problems where the output is a real-valued quantity without a natural bound. For classification problems, an activation function such as the sigmoid is typically applied at the output layer as well, to map the prediction into the interval $(0, 1)$ and interpret it as a probability.
In this textbook example, the solution was already given, with weights that produced zero error. In a real application, the training data is very large, and a solution must be found through the training process, discussed in Section 3.5

3.4 The activation functions

As discussed in Subsection 3.3.2, the *activation function* $\sigma(\cdot)$ is a fundamental piece of the neural network. Without it, stacking multiple layers would reduce the entire network to a single linear transformation, regardless of depth, making it incapable of capturing complex patterns in the data.
Moreover, the choice of activation function has a direct impact on the learning performance of the network, since different functions differ in their range, differentiability, and monotonicity, all of which affect the whole learning process.
A wide range of activation functions exists, each with different properties and suited to different tasks; an overview and comparison of these can be found in Morozov et al. (2024). This section focuses only on the functions most relevant to the present work, shown in Figure 3.4.
First, the **sigmoidal** activation function (or logistic) is one of the most widely used activation functions. It is a non-linear function that maps any real input into the interval (0, 1).
It is represented by the following equation and its derivative:

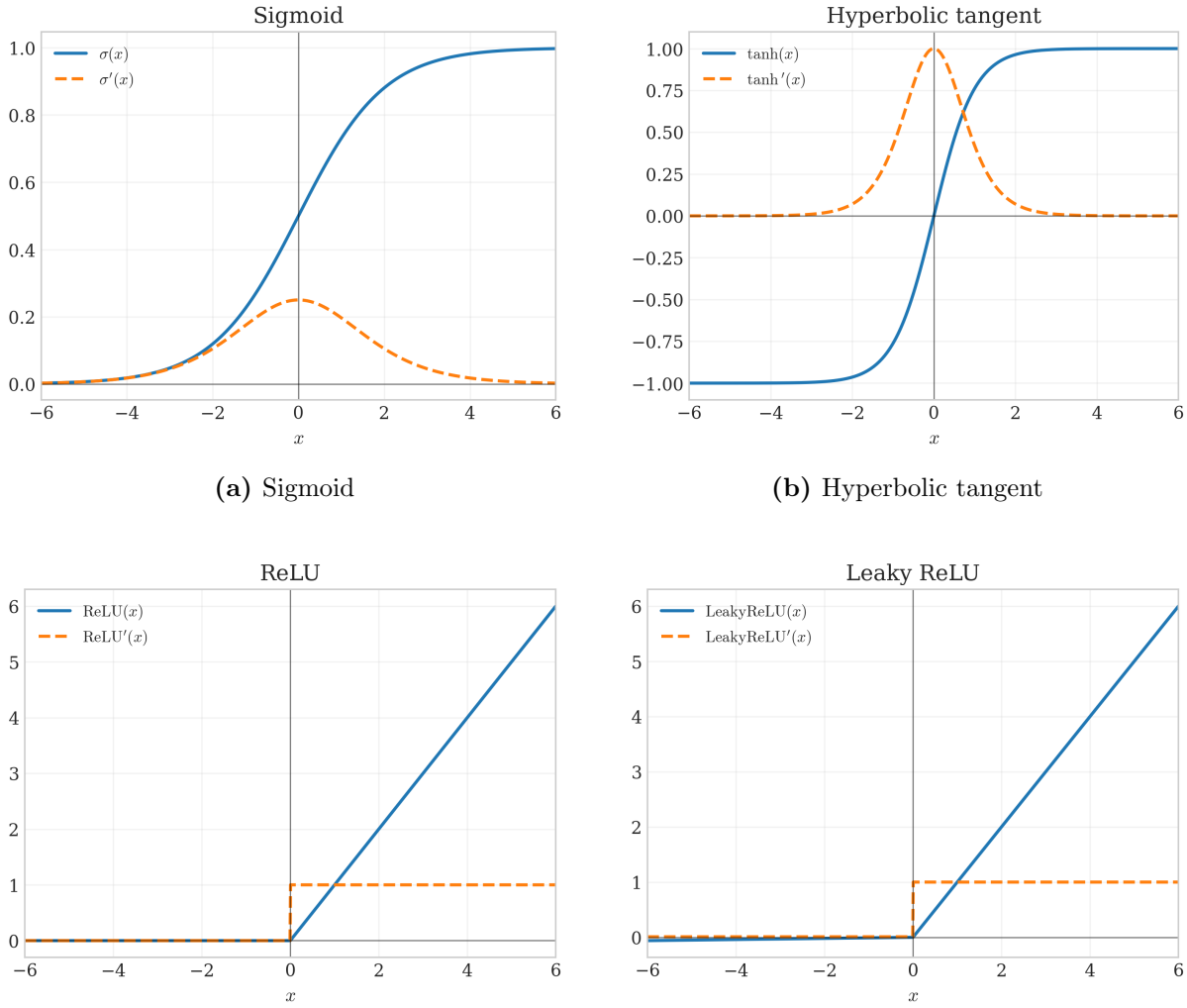
$$\sigma(x) = \frac{1}{1 + e^{-x}}, \quad \sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big) \tag{3.11}$$

Additionally, it is a continuous smooth function, strictly increasing and differentiable at every point. The derivative takes values in a range between 0 and 0.25 and decays to zero as $|x| \to \infty$.
Given its range, it maps input data into numbers that can be interpreted as probability values. For this reason, it is widely used in many applications (such as classification, predictions, natural language processing), including the option pricing context of Chapter 4, where the normalised option price lies between 0 and 1.
However, a critical issue is *saturation*: if input values are very large or very small, the derivative goes to 0, and the learning process could be stalled.
Also, the *vanishing gradient problem*[3] can be detrimental to the learning process of a neural network based on the sigmoidal activation function.
A related activation function is the **hyperbolic tangent (tanh)**, which shares the smooth and differentiable shape. Unlike the sigmoid, it can take any value in the interval (-1, 1), including negative ones, and is defined as:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \quad \tanh'(x) = 1 - \tanh(x)^2 \tag{3.12}$$

Being zero-centered, its output has zero mean, a property that tends to accelerate convergence during training compared to the sigmoid. The derivative assumes values ranging from 0 to 1, and for values of $|x| \to \infty$ it goes to 0. While this function is often used to

[3]The vanishing gradient problem occurs in deep networks when the gradient of the loss function, propagated backwards through the layers, becomes progressively smaller until it is close to zero in the layers nearest to the input, stalling the update of their parameters (Roodschild et al., 2020). The phenomenon and its connection to the sigmoid derivative are addressed in greater detail in Section 3.7.



(a) Sigmoid

(b) Hyperbolic tangent

(c) ReLU

(d) Leaky ReLU

Figure 3.4: Four activation functions used in feedforward networks, each shown together with its derivative (dashed line). The sigmoid and hyperbolic tangent saturate at both ends, so their derivatives vanish for large $|x|$; ReLU and its leaky variant keep a constant gradient on the positive side, which mitigates the vanishing gradient problem during training.

speed up the convergence, and to scale or center data, it still suffers the same saturation issue as the sigmoid function.
The **Rectified linear unit function (ReLU)** is one of the most widely used activation functions, and is the one already introduced to solve the XOR problem of the previous section. It works by activating the values greater than 0, and deactivating those that are negative.
The ReLU function is simple to implement and does not saturate for positive inputs, which largely mitigates the vanishing gradient problem in deep architectures. It is not differentiable at $x = 0$, but still performs well for gradient based optimization.
However, the main disadvantage is that if the neuron receives only negative numbers, it will not contribute to training, since it will be turned off.
This issue, known as "dying ReLU", is tackled by the **Leaky ReLU**[4] **function**, which introduces a small positive slope $\alpha > 0$ for negative input values, preventing the death of the neurons: $\text{LeakyReLU}(x) = \max(\alpha x, x)$, where α is typically set to a small constant such as 0.01. By keeping the gradient nonzero across the entire real line, Leaky ReLU prevents neurons from dying: a unit operating in the negative regime still receives a small gradient, proportional to α, and can therefore recover during training.

3.5 Neural network learning

Once the structure of a neural network has been defined, the final step is the training process to learn the parameters from the data. Training consists of minimising a chosen loss function, which quantifies the discrepancy between the outputs generated by the network and the corresponding target values.
In a feedforward neural network, training involves two main phases: the forward pass (**forward propagation**) and the backward pass (**backpropagation**). During the forward pass, the inputs are propagated through the network by applying weighted sums followed by activation functions until the output is produced. In the initial iteration, the weights are typically initialised at small random values. The biases, instead, are initialised either at zero or at small positive values.
The backward pass then adjusts these weights using the backpropagation algorithm, which propagates the error backward through the network layers. This process determines how each weight should be modified to minimise the loss function, based on the gradients computed via the chain rule. The backpropagation algorithm only computes these gradients; the parameters are then updated by an optimisation algorithm, typically *stochastic gradient descent (SGD)*, discussed later in this section.
Additionally, backpropagation is not specific to the multilayer perceptron, but is a general method for computing the gradient of the loss function in any feedforward network of differentiable units. Indeed, it can be extended to other architectures and functions as well.

[4] "Dead" ReLU always outputs zero regardless of its input, typically as a result of having learned a large negative bias. Since the gradient of the function at zero is also zero, gradient descent cannot update the weights, and the neuron is unlikely to recover. Leaky ReLU mitigates this by assigning a small positive gradient to negative inputs, giving the neuron a chance to reactivate.

In its modern form, the algorithm was popularised and applied to multilayer networks by Rumelhart et al. (1986b), who showed that it could learn the internal representations needed to solve problems that a single-layer perceptron cannot, such as the XOR problem discussed by Minsky and Papert (1969).
The procedure relies on the chain rule of differential calculus, which can be applied because of the following chain of functional dependencies within the network:

1. the loss is a function of the network output;
2. the output is a function of the activations of the hidden layer;
3. the activations of the hidden layer are a function of the weighted inputs and the activation function;
4. the weighted inputs depend on the input data and the network weights.

Therefore, by applying the chain rule of differentiation, the error can be propagated backward through the network to obtain the gradient of the loss function with respect to each weight.
The derivation that follows adapts the standard backpropagation procedure for feedforward networks (Goodfellow et al., 2016, Section 6.5) to the single-hidden-layer architecture considered here, with a mean squared error loss and sigmoid activations. The general formulation given in the reference relies on the formalism of computational graphs and automatic differentiation, which is not required for the architecture considered here and is therefore omitted.
For the output-layer weights, this chain of dependencies can be written as

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{(2)}} \cdot \frac{\partial Z^{(2)}}{\partial W^{(2)}},$$

where $Z^{(2)}$ denotes the output pre-activation. Since the output layer is linear, $\hat{y} = Z^{(2)}$ and the middle factor reduces to the identity; the same reasoning, with one additional non-linear step, applies to the hidden layer.
The loss function $\mathcal{L}$ is defined as the mean squared error between the network predictions and the target values:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 ,$$

where N is the number of training observations, y_i is the target value, and $\hat{y}_i$ is the value predicted by the network. The **mean squared error**[5] is the standard loss function for regression problems, in which the target is a real-valued quantity.

[5]Goodfellow et al. (2016, p. 216) note that the mean squared error, once widely used, is often a poor choice for output units that saturate, such as the sigmoid, since the resulting gradient becomes very small and slows down learning. In such cases a loss based on maximum likelihood, typically the cross-entropy, is preferred. This concern applies mainly to classification; for regression problems with a real-valued target, as considered here, the mean squared error remains the standard choice.

Training amounts to solving the minimisation problem

$$\min_{W,\,b}\ \mathcal{L}(W,b),$$

which is carried out through repeated iterations of the backpropagation algorithm. The first step is to differentiate the loss with respect to the network output. For a single observation i, the loss term is

$$\ell_i = (\hat{y}_i - y_i)^2,$$

and its derivative with respect to $\hat{y}_i$, obtained through the power rule, is

$$\frac{\partial \ell_i}{\partial \hat{y}_i} = 2(\hat{y}_i - y_i).$$

Since the total loss is the mean of the individual terms, $\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \ell_i$, its derivative with respect to the prediction vector is

$$\frac{\partial \mathcal{L}}{\partial \hat{Y}} = \frac{2}{N}(\hat{Y} - Y).$$

The output layer is linear, so that

$$\hat{Y} = HW^{(2)} + \mathbf{1}\, b^{(2)}.$$

By the chain rule, the derivatives with respect to the parameters of the output layer are

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = H^\top \frac{\partial \mathcal{L}}{\partial \hat{Y}}, \qquad \frac{\partial \mathcal{L}}{\partial b^{(2)}} = \mathbf{1}^\top \frac{\partial \mathcal{L}}{\partial \hat{Y}},$$

while the error propagated back to the hidden layer is

$$\frac{\partial \mathcal{L}}{\partial H} = \frac{\partial \mathcal{L}}{\partial \hat{Y}}\, {W^{(2)}}^\top.$$

The hidden activations depend on the pre-activation values $Z^{(1)} = XW^{(1)} + \mathbf{1}\, b^{(1)}$ through the sigmoid function. A further application of the chain rule yields

$$\frac{\partial \mathcal{L}}{\partial Z^{(1)}} = \frac{\partial \mathcal{L}}{\partial H} \odot \sigma'(Z^{(1)}) = \frac{\partial \mathcal{L}}{\partial H} \odot H \odot (1 - H),$$

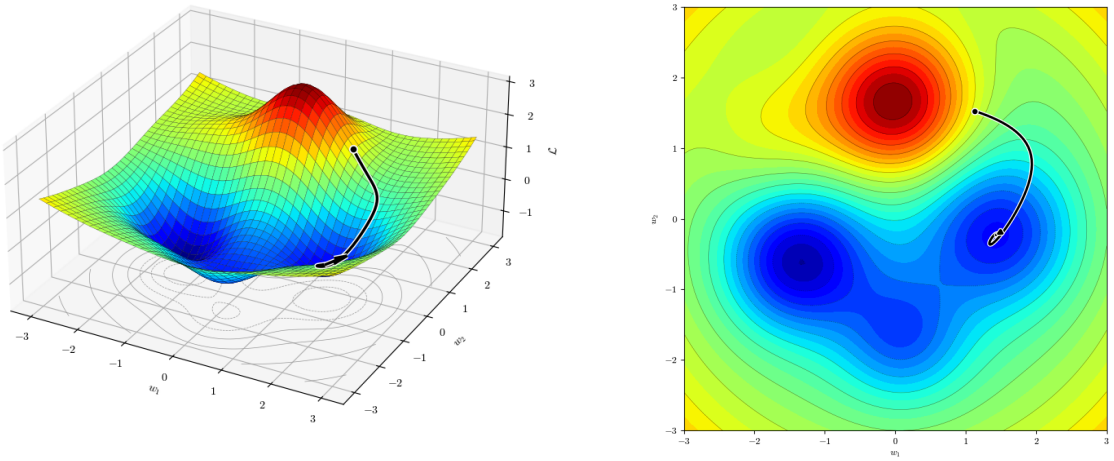
where $\odot$ denotes the elementwise product and the identity $\sigma'(Z^{(1)}) = H \odot (1 - H)$ follows from the derivative of the sigmoid given in Section 3.4. The gradients with respect to the hidden layer parameters are then

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = X^\top \frac{\partial \mathcal{L}}{\partial Z^{(1)}}, \qquad \frac{\partial \mathcal{L}}{\partial b^{(1)}} = \mathbf{1}^\top \frac{\partial \mathcal{L}}{\partial Z^{(1)}}.$$

Finally, all parameters are updated by gradient descent:

$$W^{(2)} \leftarrow W^{(2)} - lr\, \frac{\partial \mathcal{L}}{\partial W^{(2)}}, \qquad b^{(2)} \leftarrow b^{(2)} - lr\, \frac{\partial \mathcal{L}}{\partial b^{(2)}},$$

$$W^{(1)} \leftarrow W^{(1)} - lr\, \frac{\partial \mathcal{L}}{\partial W^{(1)}}, \qquad b^{(1)} \leftarrow b^{(1)} - lr\, \frac{\partial \mathcal{L}}{\partial b^{(1)}},$$



(a) The loss surface

(b) Level sets seen from above

Figure 3.5: A non-convex loss surface with several minima, shown as a three-dimensional surface (a) and as a contour map from above (b), whose concentric lines are level sets of the loss. From a region of high loss the parameters move in the direction of steepest descent and the trajectory rolls down into a valley; the momentum term adds a notion of velocity that bends the path, much like a ball rolling into a basin. The descent settles in a local minimum rather than the deepest one, which illustrates the non-convexity of the problem.

where lr is the learning rate. Through repeated iterations of this forward and backward procedure, the network gradually adjusts its parameters to minimise the loss, thereby learning the non-linear relationship between the inputs and the target values.
The update rule presented above corresponds to *batch* gradient descent, in which the gradient is computed over the entire training set before each parameter update. An alternative is *on-line* (or *stochastic*) gradient descent, in which the parameters are updated after each individual observation, using the gradient of the loss evaluated at that single point. Mini-batch schemes, which update the parameters using small subsets of the data, lie between these two extremes.
A further refinement, introduced together with the backpropagation algorithm by Rumelhart et al. (1986a), is the addition of a *momentum* term. The update for a generic parameter θ at iteration n becomes

$$\Delta\theta^{(n)} = -lr\,\frac{\partial \mathcal{L}}{\partial \theta} + \gamma\,\Delta\theta^{(n-1)},$$

where $\gamma \in [0, 1)$ is the momentum coefficient. By adding a fraction of the previous update to the current one, momentum dampens oscillations in directions of high curvature and accelerates convergence along directions of consistent gradient, which is particularly useful on the non-convex error surfaces typical of neural networks. Intuitively, the momentum term can be understood through a physical analogy: while plain gradient descent behaves

like a point that always moves in the direction of steepest descent, momentum introduces a notion of velocity, so that the parameter update accumulates speed along directions of consistent descent, much like a ball rolling down a valley (Figure 3.5). This accelerates progress towards the minimum, although it may cause the trajectory to overshoot, which is why the coefficient γ, controlling how much of the previous update is retained, must be chosen with care.

3.6 Radial Basis Function networks (RBF)

The second architecture considered in this work is the **radial basis function (RBF) network**. Whereas the multilayer perceptron is modelled after biological neurons, the RBF network has its roots in the problem of *interpolation*: finding a surface that passes through a set of known data points. Broomhead and Lowe (1988) were the first to reshape this interpolation technique as a layered network, and the approach was later extended from exact interpolation to an approximation, in which the surface is no longer required to pass through every point, but only to capture the underlying trend in the data (Bishop, 1991).

The contrast with the MLP lies in how a single hidden unit responds to its input. A hidden unit of the MLP applies a sigmoid to a weighted sum of the inputs, so what matters is on which side of a hyperplane the input falls; its response is global, in the sense that it changes over the whole input space. A hidden unit of an RBF network instead measures the distance between the input and a fixed point, called a centre, and its response depends only on that distance. The unit is therefore active only for inputs lying close to its centre, and its influence is local (Broomhead & Lowe, 1988). The same XOR problem of Section 3.3, which a single perceptron cannot represent, can also be solved by an RBF network whose centres are placed at the corners of the input square (Broomhead & Lowe, 1988).

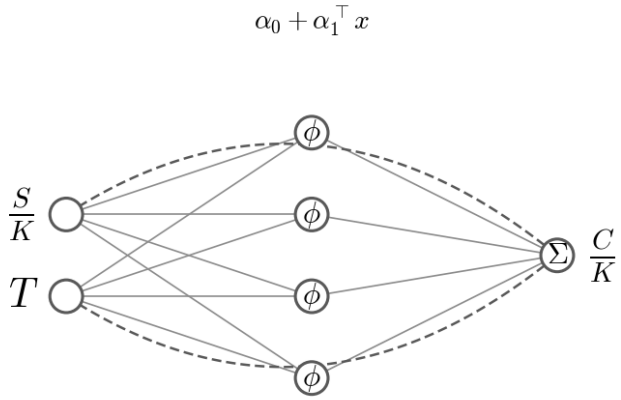
Writing the input as $x \in \mathbb{R}^d$ and using k centres, the network computes a weighted sum of k radial functions, to which a linear term is added:

$$\hat{f}(x) = \sum_{i=1}^{k} c_i \, \phi(\|x - z_i\|) + \alpha_0 + \alpha_1^\top x, \tag{3.13}$$

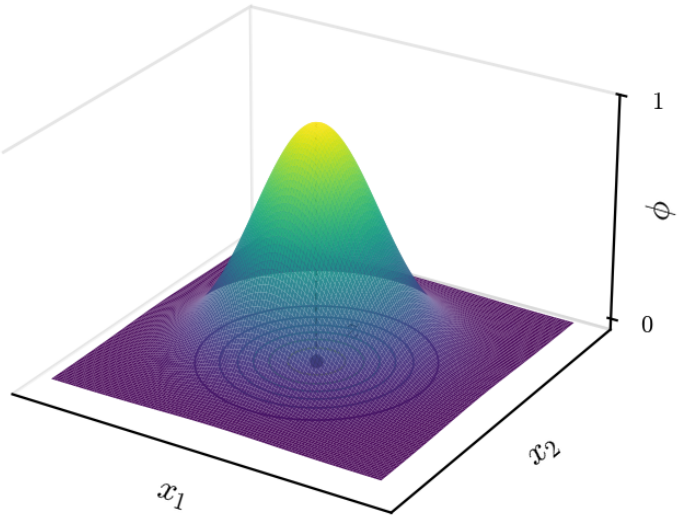
where $z_i \in \mathbb{R}^d$ are the centres, c_i are the scalar coefficients, $\|\cdot\|$ is the Euclidean norm, and ϕ is the radial basis function. The term $\alpha_0 + \alpha_1^\top x$ is a polynomial of degree one, a constant, and a linear part, which allows the network to represent any global trend in the data directly, rather than through the basis functions (Broomhead & Lowe, 1988). Equation (3.13) follows the general form adopted by Hutchinson et al. (1994).

The function ϕ depends on its argument only through the distance $r = \|x - z_i\|$, which is what makes the response radial. The two most common choices are the Gaussian and the multiquadric,

$$\phi(r) = \exp\left(-\frac{r^2}{\sigma^2}\right) \qquad \text{and} \qquad \phi(r) = \sqrt{r^2 + \sigma^2}, \tag{3.14}$$



(a)



(b)

Figure 3.6: Radial basis function network and its building block. **(a)** The network maps the two inputs, the moneyness S/K and the time to maturity T, through a hidden layer of four radial basis functions ϕ to the linear output C/K; the dashed connections carry the polynomial term $\alpha_0 + \alpha_1^\top x$ directly from the inputs to the output. **(b)** A single Gaussian radial basis function $\phi(\|x - z\|)$: a localised bump centred at z that decays with distance. Each hidden unit in (a) is one such function, and the network output is their weighted sum together with the linear term. The architecture follows Broomhead and Lowe (1988) and Poggio and Girosi (1990); the general form follows Hutchinson et al. (1994).

where the width σ sets the scale over which a unit responds (Bishop, 1991; Broomhead & Lowe, 1988). The Gaussian is the choice used in this work: it decays smoothly to zero away from the centre, so each unit contributes only in its own neighbourhood.

Like the MLP, the RBF network can be read as a three-layer feedforward structure: an input layer, a single hidden layer whose units evaluate the radial functions $\phi(\|x - z_i\|)$, and a linear output layer that forms the weighted sum in (3.13) (Broomhead & Lowe, 1988). The difference from the MLP is confined to the hidden layer, where the sigmoid of a dot-product is replaced by a distance-based response.

A defining feature of the RBF network is the way it is trained. Once the centres z_i and the width σ are fixed, the output in equation (3.13) is linear in the remaining parameters, the coefficients c_i and the polynomial terms α_0 and α_1. Fitting the network to the data is therefore a linear least-squares problem, solved in one step by

$$\theta = \left(\Phi^\top \Phi\right)^{-1} \Phi^\top y = \Phi^+ y, \tag{3.15}$$

where θ collects the coefficients, Φ is the matrix whose rows hold the k radial responses and the polynomial terms evaluated at each input, y is the vector of targets, and Φ^+ is the Moore–Penrose pseudo-inverse, computed in practice by singular value decomposition (Bishop, 1991; Broomhead & Lowe, 1988). The solution is unique and globally optimal, and it is obtained without iteration. This stands in contrast with the multilayer perceptron of Section 3.3, whose loss is not convex in its weights and is minimised by the iterative gradient-based procedure of backpropagation, which is not guaranteed to reach a global minimum.

The centres and the width must be chosen before this solve. In the strict interpolation problem a centre is placed at every data point, so that the surface passes exactly through each observation. However, with noisy data, this reproduces the noise and generalises poorly. That is why far fewer centres than observations are used in practice, placed on a representative subset of the inputs or at the centres of clusters found in the data (Bishop, 1991; Broomhead & Lowe, 1988). The width σ acts as a smoothing parameter: too small a value makes each unit respond only in a tight neighbourhood of its centre, too large a value makes the basis functions nearly indistinguishable and the linear system ill-conditioned (Bishop, 1991).

The centres and the width need not stay fixed. They may be estimated together with the coefficients, at the cost of the linearity: the fitting problem then becomes nonlinear and is solved by an iterative optimisation rather than in one step (Poggio & Girosi, 1990). This is the route taken by Hutchinson et al. (1994), who estimate the centres, the shape of the basis functions and the coefficients jointly by the Levenberg–Marquardt algorithm. Both strategies are used in this work, the closed-form solution as a fast baseline and the Levenberg–Marquardt refinement following the estimation strategy of Hutchinson et al. (1994); either way the functional form of equation (3.13) is unchanged. Their implementation and comparison on the option data are given in Chapter 4.

3.7 A modern extension: deep feedforward neural network

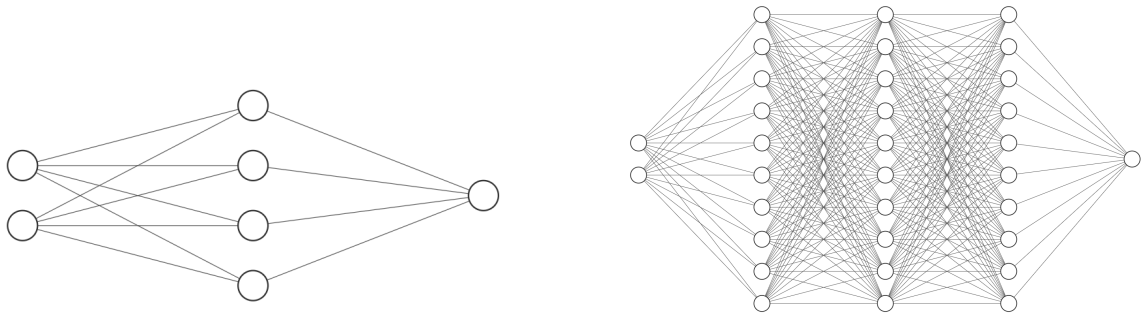
The multilayer perceptron implemented in the empirical part of this thesis, following Hutchinson et al. (1994), uses a single hidden layer. At the time, a single hidden layer was the standard choice, and not a limitation of the architecture: deeper[6] networks were not yet a practical option, partly because training them was too demanding for the hardware of that period, and partly because the techniques that make deep training feasible had not yet been developed. Since then, hardware has improved substantially, and modern CPUs and GPUs are many times faster than the machines available in the early 1990s. In parallel, new optimisation algorithms have been developed, allowing faster and more efficient training. The interest in deeper networks is not only practical. As discussed in Section 3.3, the universal approximation theorem guarantees that a single hidden layer is in principle sufficient to approximate the pricing function (Cybenko, 1989; Hornik et al., 1989). More recent results show, however, that depth can be advantageous: there exist functions that a deep network can represent with a moderate number of units, but that a shallow network (one hidden layer) can match only by becoming exponentially wide (Eldan & Shamir, 2016; Telgarsky, 2016). Depth is therefore the more parsimonious option, achieving the same expressive power with far fewer units in total, which is the sense in which these results favour depth over width.

Building on these premises, several studies have revisited the option pricing problem of Hutchinson et al. (1994) with modern deep networks. Culkin and Das (2017) train a fully connected feedforward network with four hidden layers that reproduces the Black–Scholes formula to a high degree of accuracy, while Karatas et al. (2019) extend the approach to a range of processes and contract types, studying how performance varies with the number of layers, the activation functions, and the optimisation routine. Both works rely on innovations that were not available to Hutchinson, Lo and Poggio: the use of more than one hidden layer, activation functions better suited to deep architectures, and more sophisticated optimisation algorithms.

This section presents an extension of the original network along these lines. The single-hidden-layer architecture is replaced by a deeper multilayer perceptron, the sigmoid hidden activations are replaced by the hyperbolic tangent, and the optimisation is carried out with the **Adam (Adaptive Moment Estimation)** algorithm in place of plain stochastic gradient descent. The aim is not to claim that a deeper network is necessarily superior, but to assess empirically whether, on the specific pricing problem considered here, these modern techniques improve on the architecture of Hutchinson, Lo and Poggio. The results of this comparison are reported in Section 4.9 of Chapter 4.

In a network with several hidden layers, the choice of activation function has a direct effect on whether the network can be trained at all. The sigmoid activation used in the single-layer architecture has a derivative that never exceeds 0.25 and tends to zero as its input grows in magnitude, as shown in Section 3.4. During backpropagation the gradient is propagated backward as a product of such derivatives, one for each layer, so that in a deep network it becomes progressively smaller and may vanish before reaching the layers

[6]The number of layers that contribute to a model is referred to as its depth (Chollet, 2021)



(a) Shallow network: one hidden layer

(b) Deep network: three hidden layers

Figure 3.7: A shallow feedforward network with a single hidden layer (a) and a deep network with several hidden layers (b). Both map the same input to the same output; the deep network reaches the output through a longer sequence of transformations, which is what allows it to represent certain functions with fewer units overall.

closest to the input. This is the vanishing gradient problem already noted in Section 3.4, and it makes deep networks with sigmoid activations difficult to train.

The rectified linear unit, $\mathrm{ReLU}(x) = \max(0, x)$, introduced in the same section, avoids this difficulty. For positive inputs its derivative is exactly one, so the gradient passes through the unit without being attenuated, and the function does not saturate as the input grows. Stacking many ReLU layers therefore preserves the gradient far better than stacking sigmoids, which is the main reason the rectified linear unit has become the standard activation for deep architectures (Goodfellow et al., 2016).

The extension of this thesis departs from this standard on one point: the quantity used for hedging is the derivative of the network with respect to moneyness, and the piecewise-linear unit would make that derivative piecewise constant. The hidden activations are therefore kept smooth, with the hyperbolic tangent in place of the original sigmoid; at the moderate depth used here the vanishing gradient is not a practical obstacle. The reasons are set out in full in Section 4.5.

The third modern technique concerns the optimisation algorithm. The original network is trained with stochastic gradient descent, with the momentum term described in Section 3.5. Plain stochastic gradient descent uses a single learning rate for all parameters and keeps it fixed throughout training, which makes its performance sensitive to the choice of that rate and slow to converge on the non-convex error surfaces typical of deep networks. The Adam (Adaptive Moment Estimation) algorithm (Kingma & Ba, 2015) addresses this by adapting the step size separately for each parameter. It keeps running averages of both the gradient and its square, the first and second moments, and uses them to scale each update: parameters with large or noisy gradients take smaller steps, while parameters with small, consistent gradients take larger ones. This makes training faster and less dependent on a careful tuning of the learning rate, and Adam has indeed been found to converge faster than stochastic gradient descent in option pricing networks of this kind (Karatas et al., 2019).

Adam can be seen as a refinement of the momentum scheme already introduced in Section 3.5: the running average of the gradient plays a role analogous to momentum, accumulating a direction of consistent descent, while the additional average of the squared gradient is what makes the step size adaptive. For this reason the extension replaces plain stochastic gradient descent with Adam, while retaining the same mean squared error loss used throughout.

These are the two learning networks, together with the deep extension, that are implemented and compared in the next chapter, which replicates the experiment of Hutchinson, Lo and Poggio on both simulated and real option data, assessing the models on pricing accuracy and on their ability to hedge.

Chapter 4

Empirical replication of Hutchinson et al. (1994)

4.1 Introduction and paper overview of Hutchinson, Lo and Poggio (1994)

The purpose of this chapter is to review, replicate and extend the results of Hutchinson et al. (1994), who introduced a non-parametric approach to pricing and hedging derivative securities through learning networks. Their study is one of the earliest applications of artificial neural networks to the pricing of financial derivatives and is regarded as a fundamental contribution to the field. The authors show that data-driven models can approximate option pricing functions without relying on restrictive assumptions, particularly in cases where the pricing equation implied by the no-arbitrage condition cannot be solved analytically.

In traditional option pricing models, such as the Black–Scholes framework, valuation depends on the assumed stochastic dynamics of the underlying asset. As a consequence, any misspecification of the process followed by $S(t)$ may generate pricing and hedging errors for derivatives written on that asset. Hutchinson et al. (1994) propose an alternative framework in which the key characteristics of the option contract and of the underlying asset are used as inputs to a neural network. Through a series of nonlinear transformations, the network produces the estimated option price as its output. The network itself becomes the pricing function and can be used in the same way as a traditional analytical formula, including for hedging. Since the smooth activation functions make the output differentiable with respect to the inputs, the option delta can be obtained by differentiating the estimated price with respect to the underlying.

The main advantage of non-parametric models over parametric ones is that they are driven directly by observed market data and therefore do not require assumptions such as log-normality of returns. This makes them potentially more robust to misspecification of the underlying dynamics, although at the cost of requiring more data. They also adapt to the specific dataset under analysis and can capture nonlinear relationships that standard parametric models may fail to detect. In addition, advances in computational methods have made them more efficient and relatively simple to implement.

However, non-parametric models also have limitations. Like most machine learning techniques, they typically require large amounts of historical data to achieve reliable performance. Their application may therefore be less effective for illiquid derivatives, thinly traded contracts, or newly introduced instruments, for which only limited data are available.
More than three decades later, this contribution remains highly relevant, particularly given the growing role of artificial intelligence in financial markets. Replicating their experiment therefore offers both a valuable historical perspective and practical insight into the development of data-driven approaches to option pricing.

4.2 Computational environment

The empirical analysis was implemented in *Python 3*, which has become one of the most widely used languages in quantitative finance, statistics and machine learning, largely because of its broad ecosystem of open-source scientific libraries. Three libraries are employed in this work.
NumPy (Harris et al., 2020) provides the multidimensional array together with routines for linear algebra and random number generation. It is the basis of the numerical work in this thesis: the simulation of the underlying price paths and the from-scratch implementation of the multilayer perceptron and the radial basis function network, including the matrix-form backpropagation derived in the previous chapter.
Pandas (McKinney, 2010), through its `Series` and `DataFrame` structures, is used for data manipulation and preprocessing: loading the option datasets, handling missing values, filtering contracts and reshaping the data before estimation.
Keras (Chollet et al., 2015), running on top of TensorFlow, is used for the deep multilayer perceptron of the extension. Implementing the shallow network by hand in NumPy is instructive and keeps full control over the training procedure, but the deeper architecture is more conveniently expressed in a dedicated framework, which provides the Adam optimiser and the automatic differentiation used in the extension.
The experiments were run locally in Visual Studio Code, on a desktop machine equipped with 32 GB of RAM, an AMD Radeon RX 5700 XT graphics card and an AMD Ryzen 5 5600x CPU. Since the networks are built in NumPy and Keras, the computation runs on the CPU; the GPU is not used, as the deep learning frameworks with GPU support rely on CUDA, which is specific to NVIDIA hardware.
For this reason, training the MLP network on the simulated data takes long (the entire pipeline runs in 321 minutes, with the MLP training taking almost the 50% of the time), which places a practical limit on the number of configurations that can be explored and motivates the use of a dedicated framework for the deep extension.
By contrast, Hutchinson et al. (1994) worked with the computing resources available in the early 1990s, far more limited than those used today: model estimation was correspondingly slower and imposed even tighter constraints on dataset size and on the range of experiments that could be carried out.
Having described the computational environment, the next section turns to the construction of the datasets used in the empirical analysis, both the simulated option data and

the real market observations.

4.3 Dataset design and data sources

This section describes the datasets used in the empirical replication. Following Hutchinson et al. (1994), the analysis uses both simulated option data and real market observations. The simulated dataset is useful because the true pricing function is known, so that model performance can be assessed against a known benchmark. Real market data provide a more demanding setting, in which changing volatility and market frictions affect prices. In each dataset the observations are split into a training set, used to estimate the models, and a test set, used to measure performance on unseen data.

4.3.1 Simulated dataset

The simulated dataset is generated through a Monte Carlo simulation of the underlying price. Under the standard Black–Scholes assumptions, the stock price follows the geometric Brownian motion introduced in Section 2.4:

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t. \tag{4.1}$$

Time is discretised into steps of length $\Delta t = 1/253$, each corresponding to one trading day. At each step the price evolves according to the closed-form solution of the geometric Brownian motion (Hull, 2012, p. 447):

$$S_{t+\Delta t} = S_t \exp\left[\left(\mu - \tfrac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t}\, Z\right], \tag{4.2}$$

where $Z \sim \mathcal{N}(0,1)$ is drawn independently at each step. Starting from the initial price S_0 and repeatedly sampling Z produce a complete simulated price path. The function below implements the simulation:

```
def stock_sim(S0=50, mu=0.1, sigma=0.2, n_steps=506):
    # one step is one trading day, with 253 trading days in a year
    dt = 1 / 253
    # one standard-normal shock per day: the random part of each return
    Z = np.random.normal(0, 1, n_steps)
    # daily log return: a fixed drift term plus the random shock
    log_returns = (mu - 0.5 * sigma**2) * dt + sigma * np.sqrt(dt) * Z
    # output holds the start price plus one price per simulated day
    S = np.empty(n_steps + 1)
    # the path begins exactly at S0 on day 0
    S[0] = S0
    # compound the returns from S0 to get the price on each following day
    S[1:] = S0 * np.exp(np.cumsum(log_returns))
    return S
```

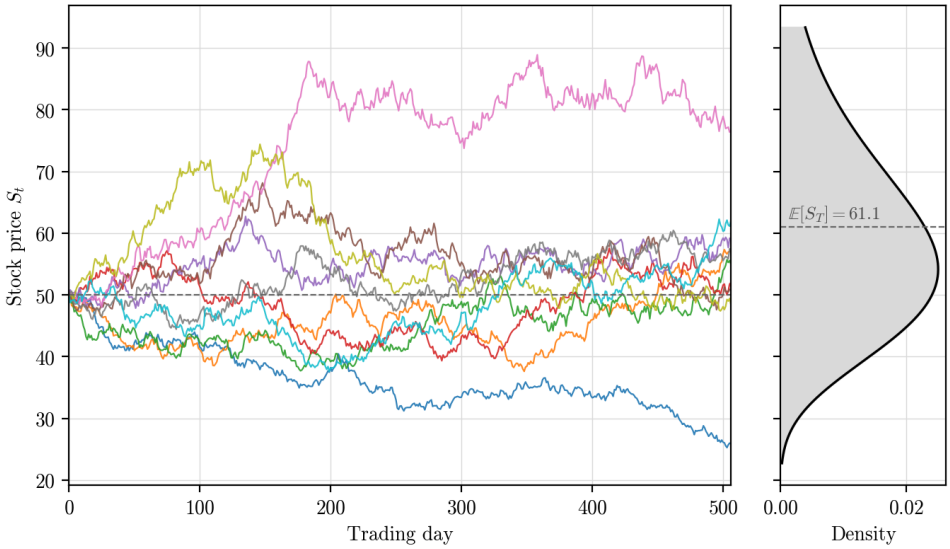


Figure 4.1: The ten simulated training paths over two trading years (506 days), generated under geometric Brownian motion with drift μ and volatility σ, all starting from $S_0 = 50$.

Here `S0` is the initial price, `mu` and `sigma` are the drift and volatility, and `n_steps` corresponds to two trading years of 253 days each. The path starts at S_0 on day 0 and then evolves over the time horizon. The training set is obtained by simulating ten independent paths. Note that, while the underlying evolves under the real-world drift μ, the options written on it are priced under the risk-neutral measure, using the risk-free rate r in the Black–Scholes formula. Figure 4.1 shows the ten simulated training paths.

Once the stock paths are simulated, option contracts are generated following the listing conventions of the Chicago Board Options Exchange (CBOE), as described in Hull (2012). Two rules govern the generation. The first concerns the strike prices: they lie on a grid spaced five units apart, and the two strikes bracketing the current price are selected. When the price lies within one unit of a strike, a third strike is added so that the price remains well bracketed. Only strikes inside the \$25–\$200 range given in the paper are retained. Figure 4.2 illustrates this rule, which is implemented as follows:

```
def strikes_generator(price, step=5, near=1):
    # nearest strike below the price, on the grid
    low = int(np.floor(price / step) * step)
    # nearest strike above: one grid step higher
    up = low + step
    # start with the two strikes that bracket the current price
    strikes = [low, up]

    # if the price almost touches a bracket strike, add one more strike
    # on that side so the price stays well inside the listed range
    if abs(price - low) <= near:
        strikes.append(low - step)    # price near the lower strike
```

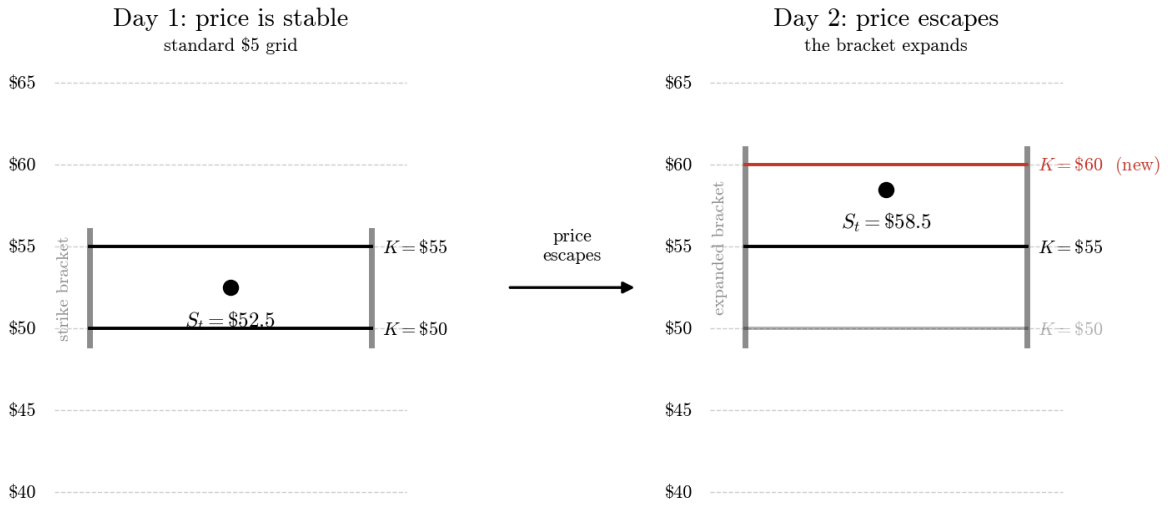


Figure 4.2: Intuitive representation of the CBOE rule: the strike "bracket" expands dynamically to keep the underlying price contained. On day 1 the price sits between the two listed strikes, $K = \$50$ and $K = \$55$; when it moves up out of the bracket, a new higher strike ($K = \$60$) is added, while the strikes already listed remain active.

```
    elif abs(price - up) <= near:
        strikes.append(up + step)     # price near the upper strike

    # the CBOE only lists strikes within the 25 to 200 range used here
    strikes = [k for k in strikes if 25 <= k <= 200]

    # drop any repeats and return the strikes in increasing order
    return sorted(set(strikes))
```

The second rule concerns the expiration dates. For each trading day up to four are active: the current and the next monthly expiration, and the two following quarterly expirations, where the quarterly months are March, June, September and December. Each calendar month is approximated by 21 trading days, and each monthly expiration is placed on the 15th trading day of the month, which falls close to the third Friday. The function that generates the active expirations for a given day is the following:

```
def expirations(day, time_frame):
    # which month we are in: each month is about 21 trading days
    month = day // 21
    exps = []

    # current monthly expiration, on the 15th trading day
    # if it has already passed, move on to next month's expiration
    exp0 = month * 21 + 15
```

```
    if exp0 <= day:
        month += 1
        exp0 = month * 21 + 15
    exps.append(exp0)

    # the next monthly expiration, one month later
    exps.append((month + 1) * 21 + 15)

    # then the next two quarterly expirations
    quarters = [2, 5, 8, 11]   # Mar, Jun, Sep, Dec
    quarter_exps = []
    for shift in range(0, 24):
        check = month + shift
        # only the months that fall on the quarterly cycle
        if check % 12 in quarters:
            expiration = check * 21 + 15
            # keep it only if it still lies in the future
            if expiration > day:
                quarter_exps.append(expiration)
            # two quarterly dates are enough
            if len(quarter_exps) >= 2:
                break
    exps.extend(quarter_exps[:2])

    # keep only expirations that are in the future and within the sample
    exps = sorted(i for i in exps if i > day and i < time_frame)
    return exps[:4]   # at most four contracts are listed at once
```

The function returns, for each day, the set of expiration dates currently listed, expressed as day indices within the simulated path. Together with the strike rule above, this determines the full grid of option contracts available on any given trading day. Following note 7 of Hutchinson et al. (1994), contracts with fewer than five trading days to expiration are excluded from the dataset.

Combining the simulated prices with the generated strikes and maturities yields the final dataset, in which each observation corresponds to one option contract on one trading day. Figure 4.3 shows the contracts generated along a typical path. The main variables are reported in Table 4.1a, and an extract is shown in Table 4.1b.

Each contract is then priced with the Black–Scholes formula for a European call on a non-dividend-paying stock (Black & Scholes, 1973):

$$c = S_0\, N(d_1) - K e^{-rT} N(d_2). \tag{4.3}$$

The pricing is implemented in the following function:

```
def bs_call(S, K, T, r=0.05, sigma=0.2):
    # at or past expiry there is no time value left:
```

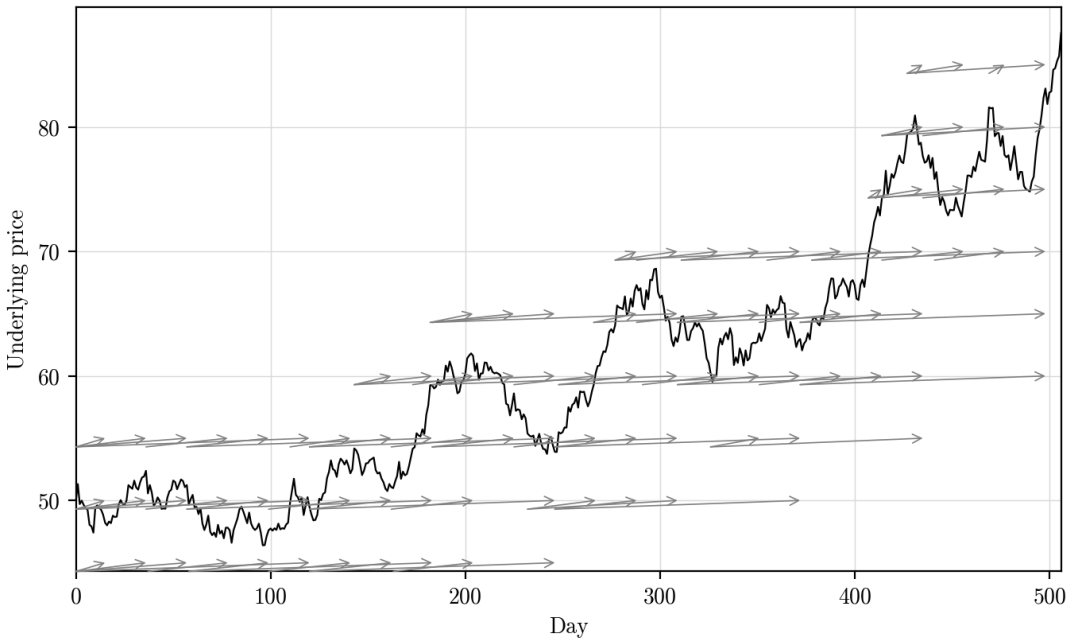


Figure 4.3: Typical simulated training path. The solid line is the underlying price, while the arrows represent the option contracts generated along it under the CBOE rule. The y-coordinate of the tip of each arrow indicates the strike price, and the arrows are slightly slanted so that contracts sharing a strike but with different listing and expiration dates remain distinguishable.

```
if T <= 0:
    return max(S - K, 0.0)
# d1 and d2: the two standard Black-Scholes terms
d1 = (np.log(S / K) + (r + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
d2 = d1 - sigma * np.sqrt(T)
# call price: the stock term minus the discounted strike term
return S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
```

The computed prices serve as the supervised learning targets. Following Hutchinson et al. (1994), the variables are normalised by the strike price. This rests on a property of the Black–Scholes price: for fixed r and σ, the call is homogeneous of degree one in S and K, that is $c(\lambda S, \lambda K, T) = \lambda\, c(S, K, T)$ for any $\lambda > 0$. Scaling the underlying and the strike by the same factor scales the price by that factor, so that the normalised price depends only on the ratio S/K. The problem can therefore be reduced from three inputs to two, and the network learns the mapping

$$\left(\frac{S}{K},\, T\right) \longrightarrow \frac{c}{K}, \tag{4.4}$$

where S/K measures moneyness, c/K is the normalised call price, and T is the time to maturity.[1] This reduction lowers the number of inputs and, with it, the amount of data needed for training (Hutchinson et al., 1994).

Variable	Description
day	Trading day index
S	Stock price
expiration	Expiration day
K	Strike price
T_days	Days to maturity
T	Maturity (years)
option_id	Contract identifier

(a) Variable description.

day	S	exp	K	T_days	T
0	50.00	15	45	15	0.059
0	50.00	15	50	15	0.059
0	50.00	15	55	15	0.059
0	50.00	36	45	36	0.142
0	50.00	36	50	36	0.142
1	50.33	15	45	14	0.055

(b) Dataset extract.

Table 4.1: Structure of the simulated option dataset: variable description (left) and an extract of the generated contracts (right).

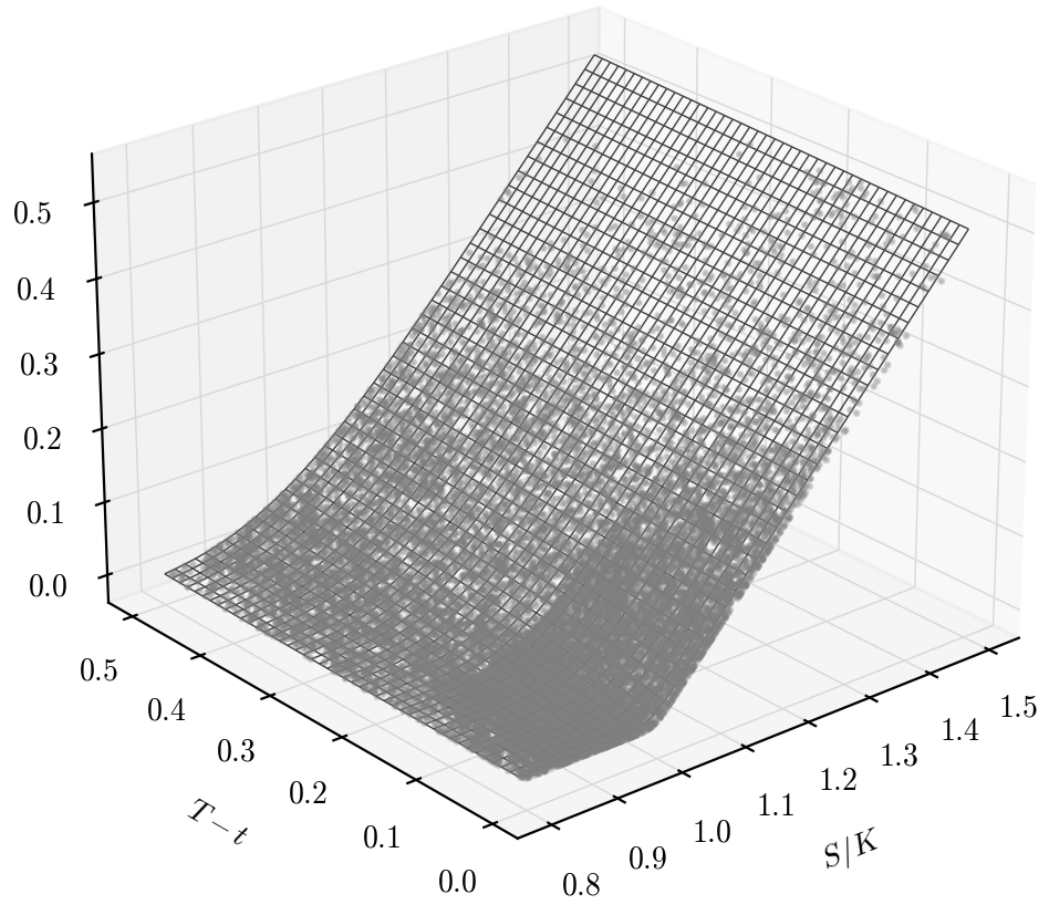


Figure 4.4: Simulated observations and theoretical Black–Scholes normalised pricing surface.

Figure 4.4, inspired by Figure 4 in Hutchinson et al. (1994), compares the simulated observations (points) with the theoretical Black–Scholes pricing surface (wireframe). If the model is correctly specified and sufficiently trained, the predicted prices should approximate this surface closely. Each simulated training path yields a cross-section of option contracts: in this setup the number of distinct contracts per path ranges from about 80 to 90, for a total of roughly 6,000 data points per path, in line with the figures reported by Hutchinson et al. (1994).[2] The dataset is then split into a training and a test set, following the experimental design of the original study: the networks are trained on ten independent simulated paths, and evaluated on a separate set of 500 independent test paths. Each trained network is assessed on the same 500 test paths, so that the performance of the different models is directly comparable.

4.3.2 Real dataset

To assess the practical relevance of the model, Hutchinson et al. (1994) complement the simulated experiment with a real dataset, consisting of daily closing prices of call options on S&P500 futures from 1987 to 1991. This thesis follows the same approach, on the same underlying: daily end-of-day call options on the S&P500 index (ticker SPX), from 2016 to 2023.

The choice of SPX options is motivated by four considerations. First, SPX options are European: they can be exercised only at maturity, which is exactly the setting used throughout this thesis and the Black–Scholes formula. No early-exercise adjustment is needed, unlike American options on individual stocks. Second, the underlying is the S&P500 itself, so the experiment remains as close as possible to the original study.[3] Third, SPX options are among the most liquid in the world, with dense strike grids and many simultaneous expirations, so that each subperiod contains enough observations to train a network, which requires a large amount of data. Fourth, the index does not undergo stock splits, so its price series is continuous, and the data report the dividend yield and the forward price, which allow the option to be priced correctly.

Each yearly file reports, for every quoted contract, the quote date, the expiration date, the days to expiration (DTE), the underlying index level, the forward price, the risk-free rate, the dividend yield, the strike, the implied volatility, the option type, the traded volume, and the bid and ask quotes. The preprocessing concatenates the yearly files, keeps only the call options, takes the option price to be the bid–ask midpoint, and builds the normalised variables used throughout: moneyness S/K, normalised price C/K, and maturity $T = \text{DTE}/365$. Contracts with fewer than seven or more than 180 days to

[1] In the implementation the time to maturity is divided by a constant before being fed to the network, so that both inputs lie on comparable numerical scales. This rescaling is only a matter of numerical conditioning and does not change the mapping being learned.

[2] Hutchinson et al. (1994) report between 5,227 and 6,847 data points per path, with an average of about 6,000. The Python implementation developed for this thesis yields a comparable dataset size.

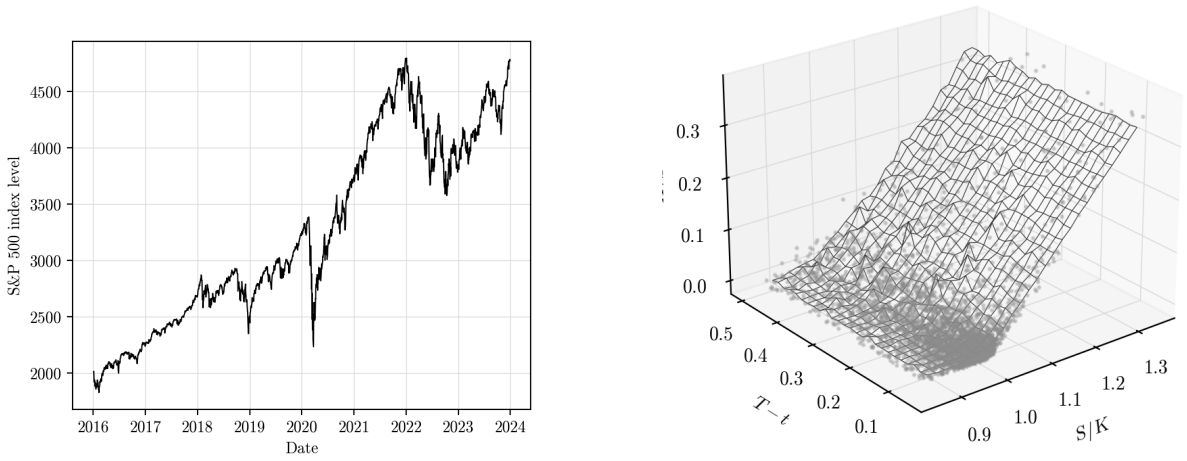
[3] Hutchinson et al. (1994) use options on S&P500 *futures*, which are American-style contracts, whereas the contracts used here are European options on the index itself. The futures price embeds the cost of carry, while the index option is priced directly on the index, using the dividend yield reported in the data.

expiration are discarded, together with quotes that violate the European no-arbitrage lower bound $c \geq e^{-rT}(F-K)^+$, where F is the forward price, and quotes with a non-positive price.

```
def load_and_preprocess(data_dir="dataset", max_dte=180, min_dte=7):
    # read every yearly SPX file and stack them into one table
    files = sorted(glob.glob(os.path.join(data_dir, "spx_*.csv")))
    df = pd.concat((pd.read_csv(f) for f in files), ignore_index=True)
    # keep calls only (type == 1); puts are not used in this study
    df = df[df["type"] == 1].copy()

    # use the mid quote as the option price, halfway between bid and ask
    df["mid"] = (df["BID"] + df["ASK"]) / 2
    # parse the date columns so they can be compared and sorted
    df["QUOTE_DATE"]  = pd.to_datetime(df["QUOTE_DATE"])
    df["EXPIRE_DATE"] = pd.to_datetime(df["EXPIRE_DATE"])

    # inputs: moneyness S/K, normalised price C/K, maturity in years
    df["S/K"] = df["UNDERLYING_LAST"] / df["STRIKE"]
    df["C/K"] = df["mid"] / df["STRIKE"]
    df["T"]   = df["DTE"] / 365.0

    # keep only the chosen maturity window (7 to 180 calendar days)
    df = df[(df["DTE"] >= min_dte) & (df["DTE"] <= max_dte)]
    # European no-arbitrage floor
    disc  = np.exp(-df["INTEREST_RATE"] * df["T"])
    lower = (disc * (df["FORWARD"] - df["STRIKE"])).clip(lower=0)
    # drop quotes that sit below this floor or have a non-positive price
    df = df[(df["mid"] >= lower) & (df["mid"] > 0)]
    return df.reset_index(drop=True)
```

Unlike the simulated case, the parameters of the Black–Scholes benchmark are taken from the data rather than fixed. The risk-free rate and the dividend yield are read directly from each quote, and the benchmark prices every option with the dividend-adjusted (Merton) form of the Black–Scholes formula, so that the dividends paid by the index constituents are properly accounted for. The volatility is still estimated from a rolling window. Following Hutchinson et al. (1994), s is the standard deviation of the 60 most recent daily log returns, computed here on the index. Equation (17) of the original study defines the estimator as $\hat{\sigma} = s/\sqrt{60}$, which is not on the annual scale required by the inputs of the Black–Scholes formula; in this thesis the daily standard deviation is therefore annualised as $\hat{\sigma} = s\sqrt{253}$, with a trading year of 253 days as in the simulated data. The time to maturity is instead measured in calendar days; this mixed convention, trading days for the volatility and calendar days for maturity, is standard market practice.

After preprocessing, the sample contains 319,137 call quotes on the S&P500, spanning January 2016 to December 2023, over 745 distinct strikes. The index level rises from



(a) S&P500 index level, 2016–2023.

(b) Empirical normalised price surface.

Figure 4.5: The real S&P500 (SPX) option dataset. Panel (a) shows the daily index level over the sample. Panel (b) shows the observed C/K against moneyness S/K and maturity $T-t$ near the money, with a grid interpolated through the daily observations; unlike the simulated case (Figure 4.4), the interpolated surface is bumpy, reflecting time-varying volatility and market frictions. Source: own computations.

quote date	S	K	DTE	mid	S/K	C/K
2016-01-04	2012.98	2005	10	31.25	1.004	0.016
2016-01-04	2012.98	2015	10	25.10	0.999	0.012
2016-01-04	2012.98	2045	10	11.31	0.984	0.006

Table 4.2: Extract of the preprocessed SPX call option dataset (319,137 observations, 2016–2023).

about 2,000 to roughly 4,800 over the period, with the sharp fall of March 2020 and the drawdown of 2022 visible (Figure 4.5a). Being an index, it is not affected by stock splits (unlike single stocks), so the series is continuous and the normalisation by the strike serves only the homogeneity argument used for the simulated data. The observed moneyness covers a far wider range than in the simulations, from deep out-of-the-money to deep in-the-money contracts, so the pricing and hedging comparisons that follow focus on the near-the-money region, where the options are most liquid and the differences between models are most informative. The empirical relationship between C/K, moneyness and maturity is shown in Figure 4.5b, as a grid interpolated through the daily observations. Unlike the simulated data, where the observations lie exactly on the Black–Scholes surface, the real surface is visibly bumpy, since the same pair $(S/K, T)$ maps to slightly different prices on different days as the volatility varies over time. An extract of the preprocessed dataset is reported in Table 4.2.

Following the real-data design of Hutchinson et al. (1994), the sample is divided into non-

overlapping six-month windows. The boundaries are set here at the June and December monthly expirations (the third Friday of the month), so that the windows match the way the options are listed. A separate network is later trained on each window, which limits the effect of non-stationarity. The window boundaries are built from the third-Friday rule:

```
def third_friday(year, month):
    """Third Friday of a month: the monthly SPX expiration day."""
    # start from the first day of the month
    d = pd.Timestamp(year=year, month=month, day=1)
    # days from the 1st to the first Friday (weekday 4 is Friday)
    offset = (4 - d.weekday()) % 7
    # add two more weeks to reach the third Friday
    return d + pd.Timedelta(days=offset + 14)
```

4.4 Implementation of the models

This section describes the models estimated in the empirical study. Two learning networks replicate the original study: a multilayer perceptron (MLP) and a radial basis function (RBF) network. A third, deeper network is added as an extension and is presented in Section 4.5. The architectures of these networks were introduced in Chapter 3; the focus here is on the concrete configuration used for option pricing and on the way each network is fitted to the data. All models share the same setup: they take the normalised inputs $(S/K, T)$ defined in Section 4.3, are trained to reproduce the normalised price C/K, and are evaluated on the same test data, both on pricing accuracy and on the delta-hedging performance introduced in Section 4.6. Throughout, their predictions are compared with the Black–Scholes benchmark.

4.4.1 Multilayer perceptron

The first nonlinear model is the multilayer perceptron introduced in Chapter 3. In the configuration adopted here it has two inputs, a single hidden layer of four units, and one output, with the sigmoid activation applied at both the hidden and the output layer. The number of hidden units follows Hutchinson et al. (1994), who adopt four nonlinear terms for every network they consider, having found diminishing returns beyond that point in preliminary out-of-sample tests. The inputs are the normalised moneyness S/K and the time to maturity T, the output is the normalised call price C/K.
Writing the input as $x = (S/K,\, T)$, the network computes

$$\hat{f}(x) = \sigma\left(b^{(2)} + \sum_{j=1}^{4} W_j^{(2)}\, \sigma\left(b_j^{(1)} + \sum_{i=1}^{2} W_{ij}^{(1)}\, x_i\right)\right), \qquad \sigma(z) = \frac{1}{1+e^{-z}}, \tag{4.5}$$

where $W^{(1)}$ and $W^{(2)}$ are the weights of the hidden and output layers and $b^{(1)}, b^{(2)}$ the corresponding biases. The outer sigmoid is the output transfer function used by Hutchinson

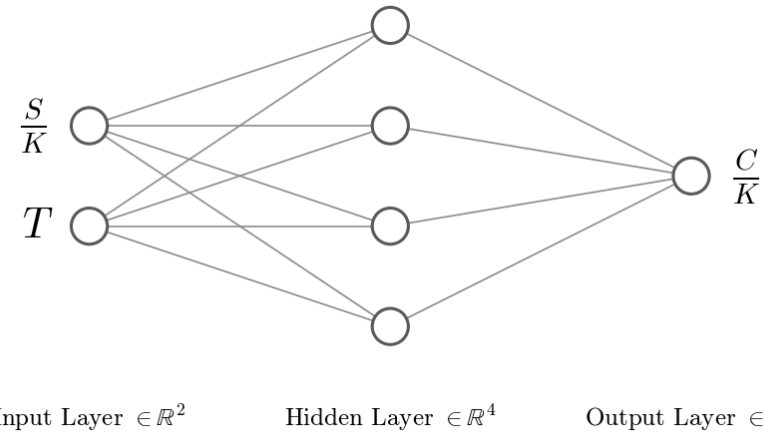


Figure 4.6: Multilayer perceptron for option pricing. The network maps two inputs, the moneyness S/K and the time to maturity T, through a single hidden layer of four units to the normalised call price C/K. The architecture follows Hutchinson et al. (1994).

et al. (1994): it maps the unbounded weighted sum into the interval $(0, 1)$, reflecting prior knowledge of the range of the target. Near the money the normalised call price C/K falls within this interval, so the learning experiment is restricted to contracts with $C/K < 1$. This is consistent with the simulated setting, where the CBOE bracketing rule tends to keep every option near the money. The comparisons are focused on the near-the-money region: the deep in-the-money tail, where C/K exceeds one, is part of the full sample described in Section 4.3.2 but is left outside the learning experiment.[4] The parameters and the forward pass are implemented as follows:

```
def __init__(self, input_size=2, hidden_size=4,
             output_size=1, lr=0.01, momentum=0.9):
    # initialize weights and biases
    self.W1 = np.random.uniform(-0.05, 0.05, (input_size, hidden_size))
    self.b1 = np.zeros((1, hidden_size))
    self.W2 = np.random.uniform(-0.05, 0.05, (hidden_size, output_size))
    self.b2 = np.zeros((1, output_size))

    self.lr       = lr
    self.momentum = momentum

    # velocity term for momentum
    self.vW1 = np.zeros_like(self.W1)
    self.vb1 = np.zeros_like(self.b1)
```

[4]Since the sigmoid approaches one only asymptotically, contracts whose normalised price lies very close to one would be hard to fit exactly. Near the money, however, the observed C/K sits well below one, so this is not a practical limitation.

```
        self.vW2 = np.zeros_like(self.W2)
        self.vb2 = np.zeros_like(self.b2)

    def forward(self, X):
        # hidden layer: weighted sum of the inputs, then the sigmoid
        self.Z1 = X @ self.W1 + self.b1
        self.A1 = self.sigmoid(self.Z1)
        # output layer: weighted sum, then the output sigmoid
        self.Z2 = self.A1 @ self.W2 + self.b2
        self.A2 = self.sigmoid(self.Z2)
        return self.A2
```

The network is trained by minimising the mean squared error between its output and the target price, using online gradient descent with momentum, following Hutchinson et al. (1994); the learning algorithm itself was described in Chapter 3. Training is online: the weights are updated after each single observation rather than on the full batch, and the data are reshuffled at the start of every epoch so that the order of the updates does not introduce a systematic bias. The momentum step follows the classic form of Rumelhart et al. (1986a), in which each weight change combines the current gradient with a fraction of the previous change,

$$v \leftarrow \gamma\, v - lr\, \nabla_W L, \qquad W \leftarrow W + v, \tag{4.6}$$

where lr is the learning rate and γ the momentum coefficient. The backward pass and the weight update are implemented as follows:

```
    def backward(self, X, y, Y_hat):
        # gradient of the MSE loss with respect to the output
        n     = X.shape[0]
        dL_dY = 2 * (Y_hat - y) / n

        # the output is a sigmoid, so its gradient
        #carries the sigmoid derivative
        dZ2 = dL_dY * self.sigmoid_deriv(Y_hat)
        dW2 = self.A1.T @ dZ2
        db2 = np.sum(dZ2, axis=0, keepdims=True)

        # propagate the error back through the sigmoid hidden layer
        dA1 = dZ2 @ self.W2.T
        dZ1 = dA1 * self.sigmoid_deriv(self.A1)
        dW1 = X.T @ dZ1
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        return dW1, db1, dW2, db2

    def update(self, grads):
```

```
        # momentum: each velocity blends the new gradient
        # with the previous step,
        # then the weights move by the velocity
        dW1, db1, dW2, db2 = grads

        self.vW1 = self.momentum * self.vW1 - self.lr * dW1
        self.vb1 = self.momentum * self.vb1 - self.lr * db1
        self.vW2 = self.momentum * self.vW2 - self.lr * dW2
        self.vb2 = self.momentum * self.vb2 - self.lr * db2

        self.W1 += self.vW1
        self.b1 += self.vb1
        self.W2 += self.vW2
        self.b2 += self.vb2
```

The full training loop applies these two steps over the data for a fixed number of epochs:

```
def fit(self, X, y, epochs=1000, shuffle=True):
    X = np.array(X, dtype=float)
    y = np.array(y, dtype=float).reshape(-1, 1)
    n_samples = len(X)

    for epoch in range(epochs):
        # reshuffle each epoch so the update order does not bias
        if shuffle:
            idx = np.random.permutation(n_samples)
            X, y = X[idx], y[idx]

        # online update: one forward, backward and
        #weight step per observation
        for i in range(n_samples):
            xi    = X[i:i+1]
            yi    = y[i:i+1]
            y_hat = self.forward(xi)
            grads = self.backward(xi, yi, y_hat)
            self.update(grads)
```

Because both activations are smooth, the price surface learned by the network is differentiable, and the delta needed for hedging is available in closed form rather than by finite differences. The delta of the call is $\partial C/\partial S$. By the homogeneity used to normalise the data, $C = K\,\hat{f}(S/K, T)$, so that

$$\frac{\partial C}{\partial S} = \frac{\partial (C/K)}{\partial (S/K)} = \hat{f}\,(1-\hat{f}) \sum_{j=1}^{4} W_j^{(2)}\,\sigma'\!\left(z_j^{(1)}\right) W_{1j}^{(1)}, \tag{4.7}$$

where the factor $\hat{f}\,(1-\hat{f})$ is the derivative of the output sigmoid. The implementation differentiates the network output with respect to its first input and clips the result to $[0,1]$, the range of a call delta:

```
def delta(self, X):
    # forward pass through both layers
    X  = np.array(X, dtype=float)
    Z1 = X @ self.W1 + self.b1
    A1 = self.sigmoid(Z1)
    Z2 = A1 @ self.W2 + self.b2
    A2 = self.sigmoid(Z2) # network output C/K, in (0, 1)

    # chain rule w.r.t. the first input (S/K); W1[0, :]
    dA1   = A1 * (1 - A1) # sigmoid derivative of the hidden layer
    inner = np.sum(dA1 * self.W1[0, :] * self.W2[:, 0],
                   axis=1, keepdims=True)
    delta = A2 * (1 - A2) * inner # times g'(f) from the output sigmoid

    # a long call delta lies between 0 and 1
    return np.clip(delta, 0.0, 1.0)
```

4.4.2 Radial basis function network

The second nonlinear model is the radial basis function network introduced in Section 3.6. It is placed alongside the multilayer perceptron of Section 4.4.1 and trained on the same inputs, the normalised moneyness S/K and the time to maturity T, with the same target, the normalised call price C/K. Following Hutchinson et al. (1994), the network uses four nonlinear terms, matching the four hidden units of the perceptron.
The model implemented here is the form of equation (3.13): a sum of four Gaussian basis functions and a linear term,

$$\hat{f}(x) = \sum_{i=1}^{4} c_i \exp\left(-\frac{\|x - z_i\|^2}{\sigma^2}\right) + \alpha_0 + \alpha_1^\top x. \tag{4.8}$$

The centres z_i are placed by k-means on the training inputs, the width σ is fixed from the spread of the centres, and, once centres and width are set, the coefficients c_i, α_0, α_1 are obtained in closed form by least squares, as described in Section 3.6.
One difference from the perceptron is worth stating, because it shapes the experiment. The perceptron has a sigmoid output, bounded in $(0,1)$, so it can represent only contracts with $C/K < 1$. The radial basis function network implemented here has a linear output and carries no such restriction: it can represent $C/K > 1$ as well. The specification differs on this point from Hutchinson et al. (1994), whose empirical networks use multiquadric basis functions augmented with an output sigmoid, so that their output is bounded in $(0,1)$ exactly as in the perceptron; Gaussians and multiquadrics are both listed in the original study among the most common choices of basis function. The Gaussian basis

with a linear output is retained here as the form of Section 3.6, and the bounded output becomes unnecessary once the experiment is restricted to the near-the-money domain. For the comparison to be made equal, both models are trained and tested on the same near-the-money domain, $S/K \in [0.8, 1.2]$ with $C/K < 1$.

```
def fit(self, X, y, epochs=None, shuffle=None, verbose=True):
    X = np.array(X, dtype=float)
    y = np.array(y, dtype=float).reshape(-1)

    # 1. place the centres with a small k-means
    self.centers = self._kmeans(X)

    # 2. one global width from the spread of the centres
    c    = self.centers
    dmax = np.max(np.sqrt(np.sum((c[:, None, :]
    - c[None, :, :]) ** 2, axis=2)))
    self.sigma = dmax / np.sqrt(2 * self.k) if dmax > 0 else 1.0

    # 3. solve the linear weights in closed form
    Phi = self._design(X)
    self.weights, *_ = np.linalg.lstsq(Phi, y, rcond=None)
```

The design matrix Φ collects, for each training input, the four Gaussian responses together with the constant and the two linear terms, so that the network output over the sample is $\Phi\theta$ with $\theta = (c_1, \ldots, c_4, \alpha_0, \alpha_1^\top)^\top$. The least-squares solution is computed by `numpy.linalg.lstsq`, which uses a singular value decomposition and so handles a rank-deficient design.

The delta needed for hedging is again available in closed form, and is simpler than for the perceptron because the linear output has no sigmoid factor. Differentiating equation (4.8) with respect to the first input gives

$$\frac{\partial (C/K)}{\partial (S/K)} = \sum_{i=1}^{4} c_i \exp\left(-\frac{\|x - z_i\|^2}{\sigma^2}\right)\left(-\frac{2\,(S/K - z_{i,1})}{\sigma^2}\right) + \alpha_{1,1}, \tag{4.9}$$

where $z_{i,1}$ is the first coordinate of centre i and $\alpha_{1,1}$ is the coefficient of the linear term in S/K; the time term and the constant do not depend on S/K and drop out. As with the perceptron, the result is clipped to $[0, 1]$, the range of a call delta.

```
def delta(self, X):
    X   = np.array(X, dtype=float)
    c   = self.weights[:self.k]  # Gaussian coefficients
    a1  = self.weights[self.k + 1:] # linear coefficients
    phi = self._phi(X) # Gaussian responses

    x1   = X[:, [0]]  # the S/K column
    z1   = self.centers[:, 0] # first coordinate of each centre
```

```
    dphi = phi * (-2.0 * (x1 - z1) / self.sigma ** 2)

    delta = dphi @ c + a1[0] # a1[0] multiplies S/K
    return np.clip(delta, 0.0, 1.0).reshape(-1, 1)
```

The estimation described so far places the centres by k-means, fixes a single global width, and solves the linear coefficients in closed form. This is the fast one-step solution, and it serves here as the starting point for the estimation strategy of Hutchinson et al. (1994), who do not keep the centres fixed: they estimate the centres, the shape of the basis functions and the coefficients jointly by the Levenberg–Marquardt algorithm.[5] The same is implemented here.

Levenberg–Marquardt is an iterative method for nonlinear least squares. It minimises the sum of squared pricing residuals over all the network parameters at once, interpolating between Gauss–Newton and gradient descent through a damping term that is raised when a step fails to lower the error and lowered when it succeeds. Starting from the closed-form solution, which provides a good initial guess, the algorithm refines the centres, the global width and the coefficients together; only steps that reduce the training error are accepted, so the refined network is never worse on the training data than the fixed-centre solution it starts from. The model itself is unchanged, a sum of Gaussian terms and a linear part as in equation (4.8), so the analytical delta of equation (4.9) continues to hold; the refinement only moves the centres, the width and the coefficients to better values.

```
def fit_lm(self, X, y, epochs=None, shuffle=None, max_iter=60,
           tol=1e-8, verbose=True):
    # start from the closed-form solution
    self.fit(X, y, verbose=False)
    theta = np.concatenate([self.centers.ravel(),
                            [np.log(self.sigma)],
                            self.weights])
    p = len(theta)

    r    = resid(theta)
    cost = 0.5 * np.dot(r, r)
    mu   = 1e-3

    for it in range(max_iter):
        # finite-difference Jacobian of the residuals wrt the parameters
        J = np.empty((len(y), p))
        for j in range(p):
            step = 1e-6 * max(1.0, abs(theta[j]))
            tj = theta.copy(); tj[j] += step
            J[:, j] = (resid(tj) - r) / step
```

[5] In the original study the norm inside the basis functions is weighted by a matrix estimated together with the other parameters; here that weighting reduces to the single global width σ.

```
A = J.T @ J
g = J.T @ r
diagA = np.maximum(np.diag(A), 1e-12)   # Marquardt scaling

# damped Gauss-Newton step: mu grows until the error drops
delta = np.linalg.solve(A + mu * np.diag(diagA), -g)
...
```

The refinement converges in a few tens of iterations and in a fraction of the perceptron's training time. This matches the experience of Hutchinson et al. (1994), who report roughly seven minutes per radial basis function network with Levenberg–Marquardt, in 10 to 80 iterations, against roughly 300 minutes per perceptron trained by gradient descent, and conclude that second-order methods seem preferable for this application. The effect of the refinement on the out-of-sample results is examined in Section 4.7, alongside the perceptron and the Black–Scholes benchmark.

4.5 An extension of the Hutchinson, Lo and Poggio approach

The deep network outlined in Section 3.7 is implemented here as a third learning network and put through the same experiment as the shallow multilayer perceptron and the radial basis function network. It takes the same inputs: the moneyness S/K and the normalised time to maturity, is trained on the same near-the-money target C/K, and is judged on the same out-of-sample pricing R^2 and prediction error of Section 4.6. The only thing that changes is the model: a deeper architecture, trained with a modern optimiser. Unlike the multilayer perceptron and the radial basis function network, which are written from scratch in NumPy, the deep network is built with Keras and trained with Adam (Kingma & Ba, 2015). Coding a deep network and its gradients by hand would add little: the techniques that make depth work, set out in Section 3.7, are exactly those a mature library already implements and tests. Using one keeps the extension close to how these models are built in practice, as in Culkin and Das (2017) and Karatas et al. (2019), and lets the comparison turn on whether depth helps rather than on the mechanics of training it. The network has three hidden layers of sixty-four units, with a smooth activation in the hidden layers and a sigmoid output, which keeps the predicted C/K in $(0, 1)$ as in the shallow network[6]:

```
# three hidden layers, a smooth activation, and a sigmoid output
# to keep C/K in (0,1)
model = keras.Sequential(
    [keras.layers.Input(shape=(2,))]
```

[6]These values are standard defaults rather than the outcome of a hyperparameter search. The aim is not to find the best possible network, but to test whether depth and a modern optimiser improve on the original design, so an untuned configuration is used throughout, with overfitting controlled by early stopping rather than by tuning the architecture.

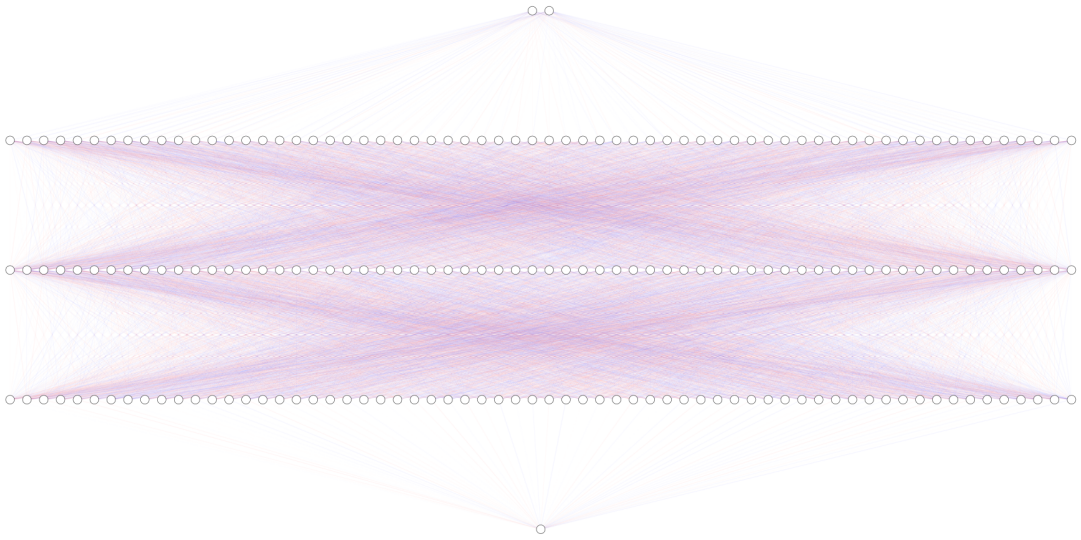


Figure 4.7: Layout illustration of the deep MLP. While maintaining the same input and output dimensions as the baseline model, this architecture features an increased number of hidden layers and a significantly higher density of neurons per layer to capture higher-order non-linearities.

```
    + [keras.layers.Dense(64, activation="tanh") for _ in range(3)]
    + [keras.layers.Dense(1, activation="sigmoid")]
)
model.compile(optimizer=keras.optimizers.Adam(1e-3), loss="mse")
```

Section 3.7 motivates the rectified linear unit as the standard remedy for the vanishing gradient of deep sigmoid networks. Here, though, the hedge ratio used for delta-hedging is the derivative of the pricing network with respect to moneyness, so the smoothness of that derivative matters as much as the accuracy of the price. A rectified linear unit is piecewise linear and its derivative is piecewise constant, which would give a stepped, discontinuous delta. A smooth, bounded activation, the hyperbolic tangent, is used instead: at three layers the network still trains without difficulty, and the resulting delta is smooth (Figure 4.10), as it is for the shallow network with its sigmoid units. The delta is obtained by automatic differentiation of the network output rather than from a hand-derived formula, and is clipped to $[0, 1]$ like the others:

```
def delta(self, X):
    # d(C/K)/d(S/K) by automatic differentiation, clipped to [0, 1]
    x = tf.convert_to_tensor(np.asarray(X, dtype="float32"))
    with tf.GradientTape() as tape:
        tape.watch(x)
        f = self.model(x, training=False)
    g = tape.gradient(f, x)
    return np.clip(g[:, 0:1].numpy(), 0.0, 1.0)
```

Training minimises the same mean squared error as the other networks, now with Adam in place of stochastic gradient descent with momentum. A deep network has far more parameters than the four-unit perceptron and can overfit a single six-month subperiod, so training stops once the validation loss stops improving and the best weights are restored. The deep network thus enters the comparison under the same conditions as the other two, and its pricing and hedging results, set against the shallow networks and the Black–Scholes benchmark, are discussed in Section 4.7.

4.6 Methodology and performance measuring

The three networks are trained to reproduce the normalised call price C/K, so the first measure of their accuracy is the out-of-sample coefficient of determination R^2 of the predicted prices against the observed ones, defined in the usual way as $R^2 = 1 - \mathrm{SSE}/\mathrm{SST}$ and computed only on the test data. Here SSE is the sum of the squared differences between predicted and observed prices, while SST is the sum of the squared differences between the observed prices and their mean, so that R^2 is the share of price variation the network reproduces, with one being a perfect fit and zero the fit of simply predicting the average price. The R^2 tells how closely a network reproduces the price surface but, as Hutchinson et al. (1994) point out, it says little about the practical value of a pricing formula. A more informative measure is the tracking error of a hedging strategy that uses the formula's own delta as the hedge ratio: a formula that prices well should also hedge well. The rest of this section sets out this second measure, which is the one carried through to Section 4.7.

The hedging strategy is the discrete counterpart of the riskless portfolio of equation (2.25), and is not re-derived here. At date 0 one call is sold at its Black–Scholes price and, with the proceeds together with riskless borrowing, $\Delta(0)$ shares are bought, where Δ is the delta implied by the formula being tested: the analytical network delta for the MLP and the RBF, equations (4.7) and (4.9), and the dividend-adjusted Black–Scholes delta for the benchmark. Selling the call at the Black–Scholes price in every case means the formula under test enters the strategy only through the hedge ratio, so the comparison is one of hedging quality with the initial sale price held fixed. The share purchase is financed entirely by the sale of the call and by the bond, so the portfolio is worth nothing at inception, $V(0) = 0$. At every following trading day $t = k\tau$, with τ equal to one day, the stock and bond holdings are rebalanced so that

$$V_S(t) = S(t)\,\Delta(t), \tag{4.10}$$

$$V_B(t) = e^{r\tau}\,V_B(t-\tau) - S(t)\big(\Delta(t) - \Delta(t-\tau)\big), \tag{4.11}$$

following Hutchinson et al. (1994). Equation (4.11) makes the strategy self-financing: every adjustment of the share position is paid for out of the bond account, with nothing added or withdrawn after date 0. At maturity the short call pays its intrinsic value, and whatever the portfolio is then worth is the **tracking error** $V(T)$. Under continuous and costless hedging with the correct model this residual would be zero; in practice it is not, because hedging is discrete and, for the networks, because the pricing formula is only an approximation. The strategy is implemented as a single pass over one price path:

```
def hedge_path(delta_fn, S_path, K, r, sigma):
    T_days = len(S_path) - 1
    dt     = 1 / 253

    # day 0: short one call, hold delta shares, put the rest in bonds,
    # so that the initial portfolio value is exactly zero
    delta_t = delta_fn(S_path[0], K, T_days / 253)
    C0      = bs_call(S_path[0], K, T_days / 253, r, sigma)
    V_S     = S_path[0] * delta_t
    V_C     = -C0
    V_B     = -(V_S + V_C) # chosen so that V(0) = 0

    # daily rebalancing (eq. 13-14)
    for t in range(1, T_days + 1):
        St    = S_path[t]
        T_rem = (T_days - t) / 253
        V_B   = np.exp(r * dt) * V_B  # bonds accrue at the risk-free rate
        if T_rem > 0:
            new_delta = delta_fn(St, K, T_rem)
            # self-financing adjustment
            V_B      -= St * (new_delta - delta_t)
            delta_t   = new_delta
        V_S = St * delta_t

    # at expiration the short call pays its intrinsic value
    C_T = max(S_path[-1] - K, 0.0)
    return V_S + V_B - C_T
```

Two summary statistics are built on the tracking error, following Hutchinson et al. (1994). Because the index level rises from about 2,000 to 4,800 over the sample, a given dollar error means very different things at the two ends of the period, so each tracking error is first expressed as a fraction of the spot at inception and discounted to date 0. Writing $v = e^{-rT} V(T)/S(0)$ for this normalised quantity, the **expected absolute tracking error** is

$$\xi = E\big[\,|v|\,\big], \tag{4.12}$$

the average size of the hedging miss, and the **prediction error** is

$$\eta = \sqrt{E^2[v] + \mathrm{Var}[v]}\,. \tag{4.13}$$

Since $E^2[v]+\mathrm{Var}[v] = E[v^2]$, the prediction error is the root mean square of the normalised tracking error: it measures the distance of v from zero, not from its own mean. For this reason η, together with R^2, is the measure reported in Section 4.7, with ξ kept only as an intermediate quantity. In Hutchinson et al. (1994), who work in dollars, with the initial stock price fixed at \$50, the normalisation by the spot is unnecessary and the discount factor is written explicitly; both moments are then taken over the dollar tracking error, as below:

```
def compute_xi_eta(VT, T0, r):
    # xi  = e^{-rT} * E[|V(T)|]                          (eq. 15)
    # eta = e^{-rT} * sqrt( E[V(T)]^2 + Var[V(T)] )      (eq. 16)
    xi  = np.exp(-r * T0) * np.mean(np.abs(VT))
    eta = np.exp(-r * T0) * np.sqrt(np.mean(VT)
                          ** 2 + np.var(VT, ddof=1))
    return xi, eta
```

On the real data the same η is computed on the spot-normalised error v, with the discount already folded in.

Alongside η, two model-free comparisons against Black–Scholes are reported. The first is the fraction of test options whose absolute tracking error is smaller than that of Black–Scholes on the same option: a value above one half means the network hedged the majority of options better than the benchmark. The second is a paired, one-sided t-test on the per-option difference in absolute tracking error, with the alternative that the network error is the smaller of the two. As Hutchinson et al. (1994) caution, the strong dependence between option paths and the large number of contracts make this an informal test, so its sign matters more than its exact value.

The two experiments estimate η over different populations, and the distinction matters for how the figures are read. In the simulated experiment the design of Hutchinson et al. (1994) is followed exactly: each of the ten training paths yields one pricing formula (Section 4.3), and for a fixed option, with given strike and time to maturity, that formula hedges the 500 independent test paths. The prediction error is estimated from the sample mean and variance of v over those 500 paths, and its mean, standard error, minimum and maximum across the ten formulas are reported, as in the original study. Here η describes the dispersion of the hedging error over repeated hedges of the *same* option. In the real-data experiment the formula trained on one six-month window is tested on the following one, and the test set is a heterogeneous cross-section of contracts with many strikes and maturities. The prediction error is therefore estimated separately within each maturity-by-moneyness cell, using breakpoints of two and five months in the time to maturity and of 0.97 and 1.03 in moneyness, following Hutchinson et al. (1994). Here η describes the dispersion across a cross-section of *different* options, so the two figures, although obtained in the same way, are not interchangeable.

The cells are reported separately for calm and turbulent periods, the empirical counterpart of the crash and non-crash split of the original study. Rather than fixing the split by calendar date, it is left to the data: the difficulty of each subperiod is measured by the median absolute tracking error of Black–Scholes itself over that period, and a subperiod is labelled turbulent when this difficulty exceeds the median difficulty across all fifteen subperiods, and calm otherwise. The turbulent periods are thus those in which the benchmark struggles most, which is where the networks have the most room to improve on it.

```
# difficulty of each subperiod: median absolute present-valued,
# spot-normalised Black-Scholes tracking error
```

```
difficulty = h.groupby("test_period")["pv_BS"].apply(
                              lambda s: s.abs().median())

# split at the median difficulty across the sixteen subperiods:
# turbulent above it, calm at or below it
threshold = difficulty.median()
regime    = np.where(difficulty > threshold, "turbulent", "calm")
```

4.7 Discussion of the results

The results of the two experiments are discussed in this section: first the simulated data, where the data-generating process is known and was described in Section 4.3.1, then the real SPX options of Section 4.3.2, where it is not.

4.7.1 Simulated data

On the simulated data, Black–Scholes is the true model: the paths are geometric Brownian motion and the prices come from the Black–Scholes formula, with the same volatility and rate used in the benchmark. The networks can therefore only approximate Black–Scholes, not beat it. The question here is whether they recover it from the data.

They do. Out of sample, all three reproduce the price surface almost exactly: the mean R^2 is 99.55% for the shallow MLP, 99.72% for the RBF and 99.88% for the deep MLP (Table 4.3). The deep network is the most accurate, but the three are within half a point of each other.

The hedging results say the same thing. Black–Scholes is the true model, so its prediction error is the lowest in almost every cell (Table 4.4), and no network beats it on more than half of the test paths. Black–Scholes is not perfect only because rebalancing is daily, not continuous; this discretisation error is the floor, and the networks sit just above it.

Among the networks the ranking is clear and matches the pricing one: the deep MLP has the lowest error, the RBF next, the shallow MLP last. At the money and three months the errors are 0.306, 0.444 and 0.562, against 0.225 for Black–Scholes (Tables 4.5–4.7). The networks do best near the money, where the price surface is most curved and the data densest: there the deep network's win rate is highest, around 0.30–0.38 (Tables 4.8–4.10). At the extremes Black–Scholes is almost exact, so the networks have little room, and in several of the outermost cells the win rate falls to a few per cent. Away from the money the error of every model grows with maturity, while at the money it stays roughly flat.

Figure 4.8 shows the prediction errors by cell, and Figure 4.9 the deltas. The network deltas are smooth and close to the Black–Scholes one, the deep MLP closest; the shallow networks fall short in the deep in-the-money tail. On data made by the model itself the networks recover it well, the deep MLP best. Whether they can improve on Black–Scholes when its assumptions fail is left to the real data. The full results are in Tables 4.3–4.10 at the end of the chapter.

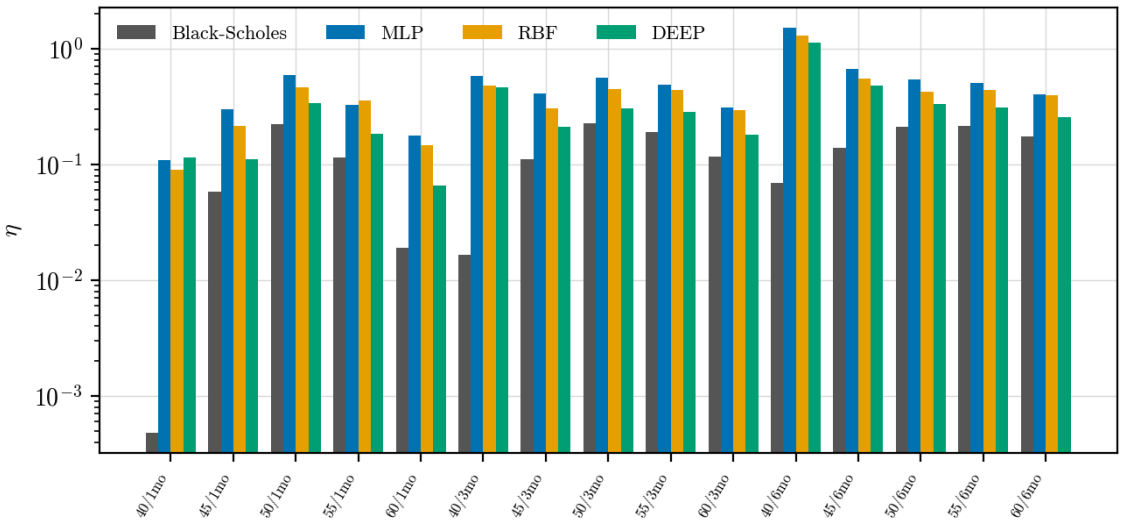


Figure 4.8: Delta-hedging prediction error η for each strike and maturity on the simulated data, for the three networks and the Black–Scholes benchmark, averaged over the ten training paths (log scale). Each colour is one model, with Black–Scholes in grey. Black–Scholes, the true model here, is lowest almost everywhere; among the networks the deep one is the best, and the error of every model grows with maturity.

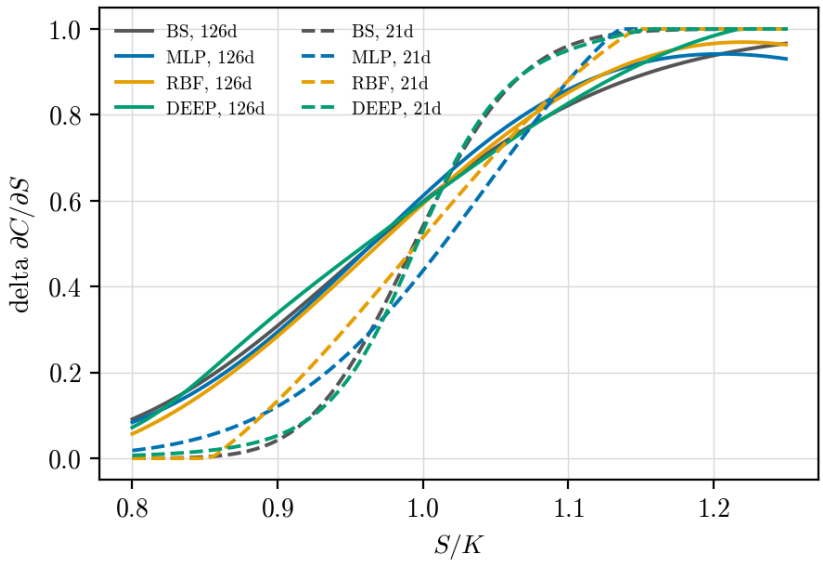


Figure 4.9: Delta $\partial C/\partial S$ against moneyness S/K for a network of each type and for Black–Scholes, at two maturities (solid 126 days, dashed 21 days). Each colour is one model, with Black–Scholes in grey. The networks recover a smooth, monotone hedge ratio close to Black–Scholes, the deep network most closely; the shallow networks fall short in the deep in-the-money tail.

4.7.2 Real data

On the real SPX options, Black–Scholes is no longer the model that generates the data: the true dynamics are unknown, volatility moves, and the constant-volatility assumption holds only as an approximation. Here, then, the networks can in principle beat the benchmark, and the question is no longer whether they recover a known formula but when they improve on a misspecified one.

At pricing, they do well across the dataset. Over the fifteen out-of-sample periods, the deep network has the highest mean R^2, 96.02%, ahead of the radial basis function network at 94.17% and well ahead of the shallow perceptron at 79.08% (Table 4.13). The gap is widest in the worst periods: the deep network's R^2 never falls below 86.16%, whereas the shallow perceptron drops to 18.68% on the hardest period.

Hedging is where the misspecification of Black–Scholes starts to matter, and the results are split by regime. In the calm periods Black–Scholes remains hard to beat: the assumptions hold and its error is the lowest in nearly every cell (Table 4.11). The networks are following it there: the radial basis function and the deep network sit close behind the benchmark, while the shallow perceptron is clearly last. In the turbulent periods the picture changes (Table 4.12). Where the constant-volatility assumption is most strained, all three networks beat Black–Scholes on the out-of-the-money options at every maturity, the cells in which a mispriced volatility hurts the benchmark most. They do not beat it everywhere: in and near the money, where Black–Scholes is still close to right, it keeps the edge almost everywhere. The improvement is concentrated exactly where theory says it should be.

The same regime split runs through the win rates (Table 4.14). The fraction of options each network hedges better than Black–Scholes is low in the calm periods and rises in the turbulent ones. In the first half of 2019 the deep network passes one half (0.503), hedging the majority of options better than the benchmark, and in the first half of 2020, around the COVID-19 crash, the shallow perceptron and the radial basis function do the same (0.534 and 0.520). Across the whole sample the deep network has the highest win rate in both regimes. Figure 4.11 shows the period-by-period prediction error behind these averages: the bars of all four models rise together in the stressed periods, most visibly the first half of 2020, and the networks close on, and in places overtake, the benchmark precisely there.

The paired test of Table 4.15 sets the result in its proper place. Pooled across every option, all three statistics are negative, so on average none of the networks outperforms Black–Scholes: the calm, near-the-money options, where the benchmark is strong and which dominate the sample, outweigh the turbulent gains. The deep network's statistic is the closest to zero of the three, consistent with its higher win rate and lower pricing error. This matches the cautious conclusion of Hutchinson et al. (1994), who found that the networks rival but do not systematically beat Black–Scholes; the test is informal, as they note, because of the strong dependence between option paths.

Figure 4.10, explains the hedging behaviour. The networks learn a smooth delta close to the Black–Scholes one in the central region, the deep network most closely, so their hedges track the benchmark in calm conditions and depart from it only where the data pull them away, in the stressed, away-from-the-money cells. Taken together, the real-data

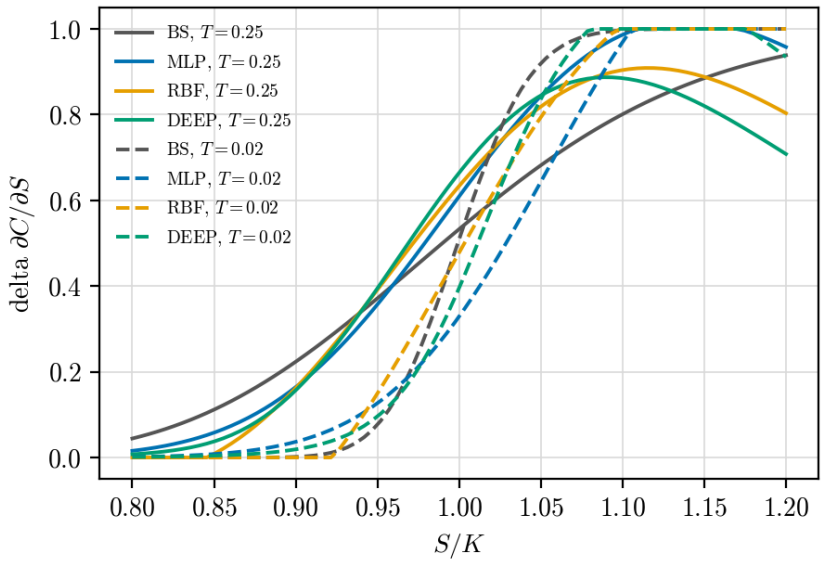


Figure 4.10: Delta $\partial C/\partial S$ against moneyness S/K for the three networks and the Black–Scholes benchmark on the real SPX data, for a network trained on a mid-sample period and evaluated at two maturities (solid $T = 0.25$ years, dashed $T = 0.02$ years). Each colour is one model, with Black–Scholes in grey. The hyperbolic tangent activation gives the deep network a smooth, monotone hedge ratio, in line with the shallow network and with Black–Scholes, and free of the steps a rectified linear unit would produce.

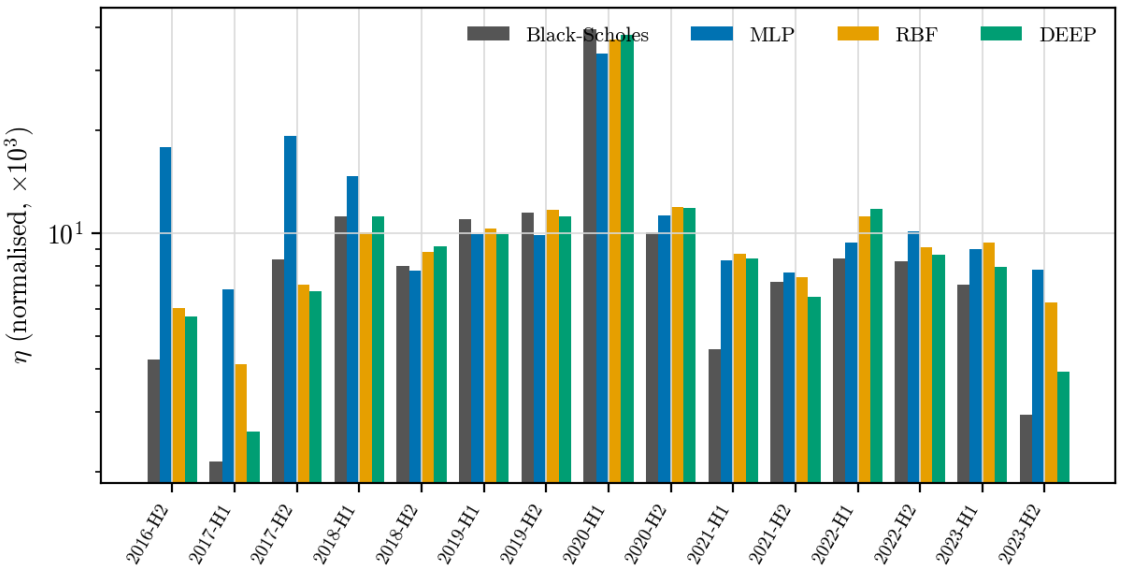


Figure 4.11: Delta-hedging prediction error η (normalised by the spot, $\times 10^3$) for each test subperiod on the real SPX data, for the three networks and the Black–Scholes benchmark (log scale). Each colour is one model, with Black–Scholes in grey. The error of every model rises in the stressed periods, most visibly the first half of 2020, where the networks close on the benchmark.

results temper the simulated ones: a network cannot beat a correct model, but against a misspecified one it can, and does, in the regimes where the misspecification bites. The deep network is the best of the three throughout, and the most reliable, without ever overturning Black–Scholes outright.

4.8 Chapter conclusions

This chapter replicated Hutchinson et al. (1994) and extended it with a deep network. The shallow MLP and the RBF were rebuilt from scratch and tested, first on simulated data and then on real SPX options, and a deep MLP was added as a third model.

On the simulated data the three networks recover Black–Scholes almost exactly, with out-of-sample R^2 above 99%, but none beats it: there Black–Scholes is the true model, so it cannot be improved on. On the real data Black–Scholes is only an approximation, and the networks beat it where that approximation fails, on out-of-the-money options in turbulent periods. In calm periods, where historical volatility is already a good input, Black–Scholes still wins.

It is important to notice that there is an asymmetry of information: the benchmark receives a estimated volatility every day, while the networks see only moneyness and time to maturity, so the two do not compete on equal terms. Later work removes this asymmetry by feeding the volatility to the network as an input (Culkin & Das, 2017; Karatas et al., 2019); in that setting the network faces the parametric model on the same information. Within the replication of this thesis, the networks' gains in the turbulent regimes are obtained with less information than the benchmark uses.

The extension confirms that depth helps. The deep MLP is the best of the three models throughout: the highest pricing R^2, the highest fraction of options hedged better than Black–Scholes, and the hedging error closest to the benchmark. It is also the most reliable, with none of the unstable periods that the shallow MLP produced on the real data. More depth and a modern optimiser do not overturn Black–Scholes, but they make the network a steadier and more accurate approximation of option prices and hedges.

4.9 Resulting Tables

This section collects the full result tables, for the simulated data (Tables 4.3–4.10) and for the real SPX options (Tables 4.11–4.15), referenced from the discussion in Section 4.7.

Table 4.3: Out-of-sample pricing R^2 (in per cent) for the three learning networks and the Black–Scholes benchmark, summarised across the ten training and out-of-sample test paths. Black–Scholes is the model that generates the data, so its fit is exact.

	RBF	MLP	Deep MLP	B–S
Min	99.42	99.42	99.61	100.00
Mean	99.72	99.55	99.88	100.00
Max	99.86	99.63	99.98	100.00

Table 4.4: Prediction error η of the Black–Scholes delta-hedging strategy on the simulated data, by strike K and time to maturity $T-t$ (in months), estimated across the 500 independent test paths. The Black–Scholes parameters are known rather than estimated, so these errors do not vary across training paths.

	$K=40$	$K=45$	$K=50$	$K=55$	$K=60$
$T-t=1$	0.000	0.058	0.221	0.114	0.019
$T-t=3$	0.017	0.110	0.225	0.192	0.117
$T-t=6$	0.069	0.140	0.211	0.216	0.174

Table 4.5: Estimated prediction error η of a delta-hedging strategy using the RBF network, by strike K and time to maturity $T-t$ (in months), estimated across the 500 independent test paths. Within each panel the first entry is the mean across the ten training paths, the second its standard error, the last two the minimum and maximum.

		$K=40$	$K=45$	$K=50$	$K=55$	$K=60$
$T-t=1$	Mean	0.089	0.217	0.464	0.357	0.146
	(SE)	(0.021)	(0.015)	(0.011)	(0.016)	(0.008)
	Min	0.021	0.166	0.425	0.299	0.116
	Max	0.166	0.293	0.520	0.445	0.196
$T-t=3$	Mean	0.479	0.304	0.444	0.442	0.294
	(SE)	(0.070)	(0.008)	(0.017)	(0.017)	(0.013)
	Min	0.189	0.279	0.385	0.382	0.250
	Max	0.741	0.359	0.531	0.539	0.361
$T-t=6$	Mean	1.302	0.555	0.424	0.442	0.399
	(SE)	(0.124)	(0.048)	(0.007)	(0.019)	(0.032)
	Min	0.688	0.387	0.402	0.380	0.309
	Max	1.780	0.752	0.469	0.547	0.613

Table 4.6: Estimated prediction error η of a delta-hedging strategy using the shallow MLP, by strike K and time to maturity $T-t$ (in months), estimated across the 500 independent test paths. See Table 4.5 for the panel layout.

		$K=40$	$K=45$	$K=50$	$K=55$	$K=60$
$T-t=1$	Mean	0.109	0.300	0.592	0.326	0.176
	(SE)	(0.023)	(0.006)	(0.005)	(0.003)	(0.003)
	Min	0.026	0.276	0.572	0.314	0.166
	Max	0.205	0.333	0.615	0.345	0.193
$T-t=3$	Mean	0.582	0.407	0.562	0.487	0.310
	(SE)	(0.080)	(0.003)	(0.010)	(0.006)	(0.004)
	Min	0.203	0.384	0.524	0.464	0.295
	Max	0.880	0.416	0.605	0.512	0.329
$T-t=6$	Mean	1.515	0.665	0.539	0.505	0.404
	(SE)	(0.157)	(0.055)	(0.005)	(0.005)	(0.006)
	Min	0.694	0.430	0.510	0.483	0.383
	Max	2.063	0.869	0.559	0.529	0.431

Table 4.7: Estimated prediction error η of a delta-hedging strategy using the deep MLP, by strike K and time to maturity $T-t$ (in months), estimated across the 500 independent test paths. See Table 4.5 for the panel layout.

		$K=40$	$K=45$	$K=50$	$K=55$	$K=60$
$T-t=1$	Mean	0.114	0.110	0.341	0.183	0.065
	(SE)	(0.036)	(0.011)	(0.021)	(0.008)	(0.006)
	Min	0.003	0.072	0.251	0.145	0.031
	Max	0.310	0.203	0.499	0.241	0.106
$T-t=3$	Mean	0.462	0.213	0.306	0.284	0.181
	(SE)	(0.108)	(0.015)	(0.015)	(0.014)	(0.008)
	Min	0.087	0.146	0.248	0.223	0.132
	Max	1.031	0.298	0.422	0.391	0.231
$T-t=6$	Mean	1.137	0.476	0.335	0.310	0.255
	(SE)	(0.203)	(0.081)	(0.013)	(0.014)	(0.012)
	Min	0.411	0.192	0.273	0.244	0.214
	Max	2.170	0.935	0.402	0.400	0.338

Table 4.8: Fraction of the 500 test paths on which the absolute delta-hedging error of the RBF network was lower than that of Black–Scholes, by strike K and time to maturity $T-t$ (in months). Within each panel the first entry is the mean across the ten training paths, the second its standard error, the last two the minimum and maximum.

		$K=40$	$K=45$	$K=50$	$K=55$	$K=60$
$T-t=1$	Mean	0.233	0.073	0.199	0.066	0.086
	(SE)	(0.008)	(0.013)	(0.005)	(0.003)	(0.013)
	Min	0.198	0.026	0.174	0.046	0.020
	Max	0.264	0.124	0.212	0.082	0.130
$T-t=3$	Mean	0.030	0.167	0.206	0.193	0.145
	(SE)	(0.004)	(0.006)	(0.012)	(0.015)	(0.012)
	Min	0.012	0.136	0.162	0.110	0.082
	Max	0.044	0.196	0.258	0.242	0.212
$T-t=6$	Mean	0.045	0.173	0.268	0.245	0.197
	(SE)	(0.007)	(0.001)	(0.010)	(0.014)	(0.021)
	Min	0.018	0.164	0.226	0.174	0.090
	Max	0.086	0.178	0.320	0.290	0.260

Table 4.9: Fraction of the 500 test paths on which the absolute delta-hedging error of the shallow MLP was lower than that of Black–Scholes, by strike K and time to maturity $T-t$ (in months). See Table 4.8 for the panel layout.

		$K=40$	$K=45$	$K=50$	$K=55$	$K=60$
$T-t=1$	Mean	0.237	0.040	0.190	0.082	0.001
	(SE)	(0.011)	(0.005)	(0.002)	(0.001)	(0.000)
	Min	0.174	0.026	0.180	0.074	0.000
	Max	0.270	0.068	0.196	0.086	0.004
$T-t=3$	Mean	0.030	0.105	0.158	0.169	0.113
	(SE)	(0.004)	(0.003)	(0.005)	(0.003)	(0.004)
	Min	0.010	0.094	0.138	0.152	0.098
	Max	0.042	0.120	0.178	0.182	0.132
$T-t=6$	Mean	0.034	0.130	0.225	0.282	0.302
	(SE)	(0.002)	(0.005)	(0.002)	(0.002)	(0.007)
	Min	0.024	0.112	0.218	0.274	0.266
	Max	0.042	0.158	0.232	0.290	0.326

Table 4.10: Fraction of the 500 test paths on which the absolute delta-hedging error of the deep MLP was lower than that of Black–Scholes, by strike K and time to maturity $T-t$ (in months). See Table 4.8 for the panel layout.

		$K=40$	$K=45$	$K=50$	$K=55$	$K=60$
$T-t=1$	Mean	0.160	0.235	0.297	0.253	0.025
	(SE)	(0.044)	(0.021)	(0.023)	(0.026)	(0.006)
	Min	0.000	0.080	0.198	0.150	0.004
	Max	0.300	0.288	0.422	0.438	0.060
$T-t=3$	Mean	0.062	0.297	0.354	0.326	0.307
	(SE)	(0.013)	(0.015)	(0.017)	(0.017)	(0.012)
	Min	0.004	0.206	0.268	0.258	0.250
	Max	0.142	0.356	0.470	0.440	0.362
$T-t=6$	Mean	0.087	0.271	0.352	0.377	0.344
	(SE)	(0.019)	(0.019)	(0.009)	(0.013)	(0.015)
	Min	0.006	0.198	0.308	0.294	0.260
	Max	0.208	0.404	0.394	0.434	0.412

Table 4.11: Delta-hedging prediction error η (normalised by the spot, $\times 10^3$) by maturity and moneyness on the real SPX data, for the calm test periods, following Hutchinson et al. (1994).

Maturity	Moneyness	RBF	MLP	Deep MLP	B–S
Short	In the money	6.56	15.33	7.80	6.64
	Near the money	6.11	9.18	5.52	4.59
	Out of the money	5.01	4.61	3.59	2.61
Medium	In the money	16.30	31.84	18.54	13.36
	Near the money	13.22	18.93	13.44	11.32
	Out of the money	8.91	9.55	8.44	7.27
Long	In the money	26.33	32.98	27.64	23.07
	Near the money	19.43	28.87	21.07	20.82
	Out of the money	14.58	14.38	14.22	10.78

Table 4.12: Delta-hedging prediction error η (normalised by the spot, $\times 10^3$) by maturity and moneyness on the real SPX data, for the turbulent test periods. All three networks beat Black–Scholes on out-of-the-money options across every maturity, where the constant-volatility assumption is most strained.

Maturity	Moneyness	RBF	MLP	Deep MLP	B–S
Short	In the money	23.92	22.62	28.03	21.82
	Near the money	13.14	12.84	13.73	13.36
	Out of the money	13.26	13.24	13.04	14.84
Medium	In the money	40.28	39.32	39.33	28.19
	Near the money	27.35	24.66	29.29	22.79
	Out of the money	21.87	20.16	21.18	28.34
Long	In the money	57.36	58.14	57.29	47.83
	Near the money	41.11	36.53	40.92	31.81
	Out of the money	34.71	30.89	33.55	45.29

Table 4.13: Out-of-sample pricing R^2 (in per cent) for the three networks and the dividend-adjusted Black–Scholes benchmark on the real SPX data, summarised across the fifteen out-of-sample test periods.

	RBF	MLP	Deep MLP	B–S
Min	80.24	18.68	86.16	54.20
Mean	94.17	79.08	96.02	93.16
Max	98.72	97.56	99.05	98.37

Table 4.14: Fraction of out-of-sample options whose absolute delta-hedging error was lower than that of Black–Scholes, for each test period, following Hutchinson et al. (1994). Values above 0.5 mean the network beat Black–Scholes on a majority of options.

Test period	RBF	MLP	Deep MLP
2016-H2	0.278	0.203	0.312
2017-H1	0.162	0.136	0.277
2017-H2	0.203	0.163	0.335
2018-H1	0.457	0.377	0.421
2018-H2	0.278	0.327	0.300
2019-H1	0.419	0.455	0.503
2019-H2	0.329	0.419	0.473
2020-H1	0.520	0.534	0.487
2020-H2	0.291	0.356	0.441
2021-H1	0.221	0.239	0.262
2021-H2	0.266	0.283	0.343
2022-H1	0.280	0.395	0.361
2022-H2	0.345	0.317	0.449
2023-H1	0.247	0.233	0.320
2023-H2	0.159	0.170	0.249

Table 4.15: Paired t-test comparing the absolute delta-hedging error of each network with that of Black–Scholes on the same option, over all real test sets, following Hutchinson et al. (1994). The one-sided alternative is that the network error is smaller. All statistics are negative, so pooled across every option the networks do not outperform Black–Scholes on average; the deep network is the closest. As Hutchinson et al. (1994) caution, the strong dependence between option paths and the large sample make this an informal test.

Pair	t-statistic	p-value
RBF vs B–S	-44.94	1.0000
MLP vs B–S	-38.09	1.0000
Deep MLP vs B–S	-16.73	1.0000

Conclusions and future work

This thesis has asked whether artificial neural networks can price and hedge options, and whether they improve on the Black–Scholes–Merton model. The question was addressed by replicating and extending Hutchinson et al. (1994), moving from the theory of option pricing to the empirical comparison that puts it to the test.

The first two chapters set out the problem and the theory. Chapter 1 introduced options, their payoffs and their main uses, and reached the problem the thesis addresses: the payoff of an option at maturity follows from its contractual terms, but assigning a fair value before expiration is far harder, because that value depends on the future behaviour of the underlying. Chapter 2 supplied the theory behind the reference model. Brownian motion was built as the limit of a scaled random walk, the geometric Brownian motion was obtained through the Itô–Doeblin formula, and the Black–Scholes–Merton equation was derived from a continuously rebalanced riskless portfolio, with a solution that does not depend on the drift of the underlying. The chapter also showed where the model breaks down: returns are not normal, volatility is not constant, and implied volatility varies with the strike. The parametric extensions that address these features, from the skewness and kurtosis correction of Corrado and Su (1996) to the stochastic volatility of Heston (1993) and the GARCH family (Bollerslev, 1986), add structure to the model at the cost of greater complexity, which motivated a different route.

That route was the non-parametric approach. Chapter 3 set out the two architectures used in the replication, the multilayer perceptron and the radial basis function network, together with their training, and outlined a deep extension. The reason to use neural networks here is not any analogy with the brain, which the chapter set aside, but the universal approximation property, which guarantees that a network of sufficient size can approximate the pricing function arbitrarily well. Chapter 4 put this into practice. The shallow perceptron and the radial basis function network were rebuilt from scratch, a deeper network was added, and the three were compared with a Black–Scholes benchmark on two datasets: option prices simulated under geometric Brownian motion, and real S&P500 index options from 2016 to 2023.

The results divide along the two datasets. On the simulated data Black–Scholes is the true model, so the networks could only try to recover it, and they did: out-of-sample pricing R^2 was above 99% for all three, highest for the deep network, and none beat the benchmark. On the real data Black–Scholes is only an approximation. There the deep network reached the highest pricing accuracy, a mean R^2 of 96.02% against 93.16% for the benchmark, and in the turbulent periods all three networks hedged out-of-the-money options better than Black–Scholes at every maturity, the contracts on which a misspecified,

constant volatility hurts the benchmark most. Pooled across the whole sample, however, the networks did not beat Black–Scholes, because the calm, near-the-money options that dominate the data outweigh the gains made under stress. This matches the cautious conclusion of Hutchinson et al. (1994), who found that learning networks rival but do not systematically beat the parametric formula.

Two findings stand out. First, the value of a non-parametric model is concentrated where the parametric one fails: the networks follow Black–Scholes closely when its assumptions hold and depart from it only in the stressed, away-from-the-money region, where data-driven flexibility is most useful. Second, depth helps. On pricing accuracy, on the share of options hedged better than the benchmark, and on stability over time, the deep network was the best of the three, without the unstable periods produced by the shallow perceptron.

The practical reading is straightforward. A learning network complements an arbitrage-based formula rather than replacing it. It is most useful when the dynamics of the underlying are unknown or misspecified, and least useful when a correct model is already available. The cost of that flexibility is data: networks need large samples to train reliably, which limits their use for illiquid or newly listed contracts and makes liquid index options a suitable setting. The experiment also confirms a methodological point: pricing accuracy alone is not enough to judge a pricing formula, and the tracking error of the delta-hedging strategy it implies is the more demanding test, since a formula that prices well should also hedge well.

One limitation should be stated. The comparison is not made on equal terms, because the Black–Scholes benchmark receives an estimated volatility every day, while the networks see only moneyness and time to maturity. The gains made by the networks under stress are therefore obtained with less information than the benchmark uses, which makes them more notable and points to the next step. The most immediate extension is to remove this asymmetry by giving the volatility, and other state variables, to the network as inputs, so that the two approaches compete on the same information, as in later work such as Culkin and Das (2017) and Karatas et al. (2019). The input set could be enriched further, for instance with the implied volatility surface or the term structure of interest rates. The architecture, too, could be developed: recurrent networks could handle path-dependent payoffs, and a no-arbitrage or smoothness constraint could be imposed on the network directly. Finally, the experiment itself could be made more realistic, by including transaction costs in the hedging strategy and by replacing the untuned defaults of the deep network with a proper hyperparameter search. Each of these would sharpen and not overturn the central finding of this thesis: neural networks do not displace Black–Scholes, but they approximate option prices and hedges accurately, and they improve on the benchmark where it is known to fail.

Bibliography

Bachelier, L. (1900). Théorie de la spéculation. *Annales Scientifiques de l'École Normale Supérieure*, *17*, 21–86.

Bishop, C. (1991). Improving the generalization properties of radial basis function neural networks. *Neural Computation*, *3*(4), 579–588. https://doi.org/10.1162/neco.1991.3.4.579

Black, F., & Scholes, M. (1973). The pricing of options and corporate liabilities. *Journal of Political Economy*, *81*(3), 637–654.

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, *31*(3), 307–327.

Broomhead, D. S., & Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems*, *2*, 321–355.

Chollet, F., et al. (2015, March). Keras. https://keras.io

Chollet, F. (2021). *Deep learning with python* (2nd). Manning Publications.

CONSOB. (2026). *I derivati* [Accessed: 2026-05-27]. https://www.consob.it/web/investor-education/i-derivati

Corrado, C. J., & Su, T. (1996). Skewness and kurtosis in S&P 500 index returns implied by option prices. *Journal of Financial Research*, *19*(2), 175–192.

Cox, J. C., Ingersoll, J. E., & Ross, S. A. (1985). A theory of the term structure of interest rates. *Econometrica*, *53*(2), 385–407.

Culkin, R., & Das, S. R. (2017). Machine learning in finance: The case of deep learning for option pricing. *Journal of Investment Management*, *15*(4), 92–100.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314.

Duan, J.-C. (1995). The GARCH option pricing model. *Mathematical Finance*, *5*(1), 13–32.

Eldan, R., & Shamir, O. (2016). The power of depth for feedforward neural networks. *Proceedings of the 29th Conference on Learning Theory (COLT)*, *49*, 907–940.

Engle, R. F. (1982). Autoregressive conditional heteroskedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, *50*(4), 987–1008.

Federal Reserve Bank of New York. (2021). *Transition from LIBOR*. Alternative Reference Rates Committee (ARRC). Retrieved June 3, 2026, from https://www.newyorkfed.org/arrc/sofr-transition

Friedman, J. H., & Stuetzle, W. (1981). Projection pursuit regression. *Journal of the American Statistical Association*, *76*(376), 817–823.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Haykin, S. (1999). *Neural networks: A comprehensive foundation* (2nd). Prentice Hall.

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

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366.

Hull, J. C. (2012). *Options, futures, and other derivatives* (8th ed.). Pearson Prentice Hall.

Hutchinson, J. M., Lo, A. W., & Poggio, T. (1994). A nonparametric approach to pricing and hedging derivative securities via learning networks. *Journal of Finance*, *49*(3), 851–889.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning: With applications in r*. Springer.

Karatas, T., Oskoui, A., & Hirsa, A. (2019). Supervised deep neural networks (dnns) for pricing/calibration of vanilla/exotic options under various different processes. *arXiv preprint arXiv:1902.05810*.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444.

Lo, A. W., & MacKinlay, A. C. (1988). Stock market prices do not follow random walks: Evidence from a simple specification test. *The Review of Financial Studies*, *1*(1), 41–66. Retrieved June 4, 2026, from http://www.jstor.org/stable/2962126

McCullagh, P. (2002). What is a statistical model? *The Annals of Statistics*, *30*(5), 1225–1310.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133.

McKinney, W. (2010). Data structures for statistical computing in Python. In S. van der Walt & J. Millman (Eds.), *Proceedings of the 9th python in science conference* (pp. 56–61). https://doi.org/10.25080/Majora-92bf1922-00a

Merton, R. C. (1973). Theory of rational option pricing. *Bell Journal of Economics and Management Science*, *4*(1), 141–183.

Merton, R. C. (1976). Option pricing when underlying stock returns are discontinuous. *Journal of Financial Economics*, *3*(1), 125–144. https://doi.org/https://doi.org/10.1016/0304-405X(76)90022-2

Minsky, M., & Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. MIT Press.

Morozov, A. V., Levkivskyi, V. L., & Plechystyy, D. D. (2024). Activation functions in neural networks: Overview and comparison. *Vcheni Zapysky TNU imeni V.I. Vernadskoho. Seriia: Tekhnichni Nauky*, *35(74)*(3), 144–151.

Øksendal, B. (2003). *Stochastic differential equations: An introduction with applications* (6th ed.). Springer.

Poggio, T., & Girosi, F. (1990). Networks for approximation and learning. *Proceedings of the IEEE*, *78*, 1481–1497. https://doi.org/10.1109/5.58326

Roodschild, M., Gotay Sardiñas, J., & Will, A. (2020). A new approach for the vanishing gradient problem on sigmoid activation. *Progress in Artificial Intelligence*, *9*(4), 351–360.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986a). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition, volume 1: Foundations* (pp. 318–362). MIT Press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986b). Learning representations by back-propagating errors. *Nature*, *323*, 533–536.

Shreve, S. E. (2004). *Stochastic calculus for finance ii: Continuous-time models*. Springer.

Telgarsky, M. (2016). Benefits of depth in neural networks. *Proceedings of the 29th Conference on Learning Theory (COLT)*, *49*, 1517–1539.