King's College London

Department of Informatics

Faculty of Natural, Mathematical and Engineering Sciences

Using Long Short-Term Neural Networks for Deep Hedging Equity Derivatives in the Presence of Market Frictions

Raphaël ROLNIK (23131461)

MSc in Computational Finance

Individual Project Submission (7CCSMPRJ) - 2023/24

Supervised by Dr. Carmine Ventre

Word count : 10 090 without appendix (as calculated by Overleaf word counter)

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

Check the appropriate box below :

☑ **I agree** to the release of my project

☐ **I do not agree** to the release of my project

Signature : *Raphaël ROLNIK* **Date :** August 5, 2024

Originality Avowal

I certify that the thesis I have presented for examination for the MSc degree in Computational Finance of the King's College London is solely my own work other than where I have clearly indicated that it is the work of others. In which case the extent of any work carried out jointly by me and any other person is clearly identified in it.

In the context of this dissertation, a number of mathematical formulae and results are highlighted, in particular for purposes of explanation, contextualisation or illustration. These results are not the fruit of my personal work, and the names of their authors are duly mentioned. However, it is possible that the fairly substantial presence of this mathematical content within the essay could lead to suspicions of plagiarism by the software used to check this aspect. It is therefore important that markers take this issue into account when assessing this work.

I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to any trusted plagiarism detection service.

I confirm this report does not exceed 15,000 words (excluding appendices).

Acknowledgments

I would like to express my heartfelt gratitude to my family for their unwavering support, encouragement, and understanding throughout the course of my MSc studies and the completion of this dissertation.

Abstract

Risk management is one of the key foundations in the world of finance and financial markets, both from an academic and institutional perspective. With the rapid development of financial markets, increasingly complex products are emerging, requiring ever greater thought as to how to optimise the risk exposure of those who decide to invest in them. At the same time, advances in economic and financial research, particularly quantitative research, have led to the emergence of increasingly effective mathematical and computational hedging techniques. Artificial intelligence and machine learning are one of them, and in this dissertation, we will be analysing in as much detail as possible the possibilities offered by these innovative processes. One of our main concern will be to replicate the effectiveness of the Black-Scholes model - foundation of the traditional quantitative finance framework - with a neural network and data-driven based approach to solve hedging problems on derivatives and equity derivatives assets. Investigating the flexibility and robustness of these strategies will also be one of our main points of study, along with comparing it in depth with the the traditional quantitative finance approach. Taking account of market frictions, which represent one of the main limitations of the Black-Scholes model, will also be one of the central subjects of study in this dissertation

Contents

1 Introduction

One of the most revolutionary feature that the Black-Scholes model introduced is the idea pricing-hedging duality. The latter directly links pricing a derivative to the cost involved by the replication of its payoff. However, this inevitably created a hurdle in the environment of incomplete markets particularly as financial markets are complex and unpredictable. Indeed, most models assume that price dynamics are known and fixed, which creates a heavy dependence between price and hedging parameters.

To address these challenges, traders, academics and practitioners choose to use superhedges, that we can define as robust methods minimizing the cost of hedging without relying on specific models. These strategies aim to provide a future value that is higher than the risky position, even in the absence of perfect replication.

Hedging also highlighted the problem of handling market frictions such as transaction costs or taxes. A certain number of works, such as [1] tried to improve the concept of delta-hedging in the presence of low transaction costs and frequent rebalancing. This concept was extended to arbitrary option portfolios by researches such as [2], who laid theoretical foundations for the utility-based approach to hedging However, strong assumptions about the underlying dynamics are undermining these approaches.

We try to overcome these shortcomings in this dissertation by applying deep learning methods on historical stock prices data with the use of recurrent neural networks. [3] outlined the theoretical basis and presented strong evidence for the algorithm's performance in applying deep neural networks to the hedging problem. We view modern developments in machine learning and deep learning research applied to finance and particularly the financial markets as tools paving the way for the exploration of fully data-driven solutions based on historical data. One of the motivations of this dissertation is to start from the theory of certain relatively parameterised models based on purely synthetic data (of which the Black-Scholes model is one of the precursors), and to use large historical datasets to develop a hedging framework, supported by real-world financial data.

We first justify the attributes of our neural network model and elaborate on its structure. Next, we build a benchmark for our algorithm using the Black Scholes model configuration. In order to test the robustness of our model, we test it on synthetic data calibrated to our initial raw historical data. We also try to change the parameters and numbers of inputs of our algorithm in order to analyse their impacts on the results. We finally try to implement in our model a scenario with transaction costs, where we demonstrate its potential usefulness to improve financial decision-making and risk management.

2 Background Theories : From the Black Scholes Model to Deep Hedging

2.1 Black Scholes Model

2.1.1 Main results

Delta hedging is one of the foundations when developing the intuition behind the Black-Scholes option pricing model. This relies on the assumptions of a frictionless market and constant rebalancing.

In oder to theoretically price a European call option under the Black-Scholes model, we use the following formula :

$$V(t, S_t) = \mathcal{N}(d_+)S_t - \mathcal{N}(d_-)e^{-r(T-t)}K$$

where

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

$$d_- = d_+ - \sigma\sqrt{T-t}$$

and $\mathcal{N}(\cdot)$ represents the cumulative distribution function of a standard normal random variable.

This enables us to derive the following essential value :

$$\Delta_t = \frac{\partial V(t, S_t)}{\partial S_t} = \mathcal{N}(d_+).$$

where :

- $V(t, S_t)$ is the price of the European call option at time t when the underlying asset price is S_t
- S_t is the current price of the underlying asset
- K is the strike price of the option
- r is the risk-free interest rate
- σ is the volatility of the underlying asset
- T is the time to maturity of the option
- t is the current time
- Δ_t is the option's delta, measuring the sensitivity of the option's price to changes in the price of the underlying asset

2.1.2 Limits

In this subsection, we go deeper into the theoretical limitations encountered by the Black Scholes model.

Absence of continuous trading

Our agent has Δ_t units of the underlying asset at time t, which means there is a probability 1 of the payoff of the given asset being replicated in its entirety. As discussed by [4], even in the absence of frictions, continuous trading is not feasible in a real market environment, and failing to maintain the portfolio in a delta-neutral state for the investment period creates inevitably a hedging error.

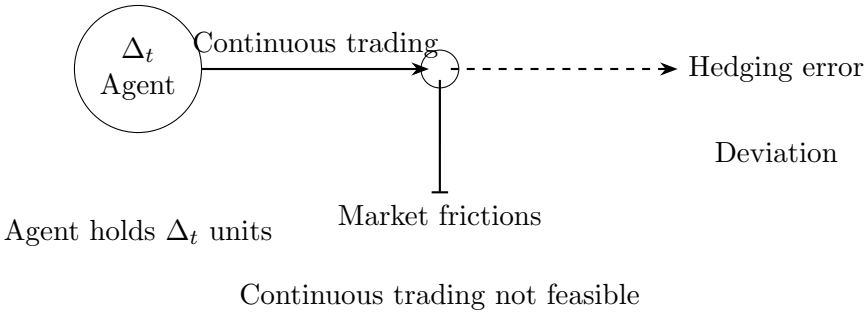


Figure 2.1: Illustration of hedging with Δ_t units, with hedging errors introduced by real markets.

Constant volatility

[5] demonstrated that market volatility is not constant and varies over time, which can lead to inaccuracies in the Black-Scholes model.

The price S_t of the underlying asset in the Black Scholes model is modeled as a geometric Brownian motion. The stochastic differential equation (SDE) describing the price dynamics is given by:

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t$$

where S_t is the price of the underlying asset at time t, μ is the constant drift rate, σ is the constant volatility, W_t is a standard Wiener process (known as Brownian motion).

The volatility σ is assumed to be constant over time. This means that σ does not change with respect to time t or the level of the underlying asset S_t. We can express this as:

$$\frac{\partial \sigma}{\partial t} = 0 \quad \text{and} \quad \frac{\partial \sigma}{\partial S_t} = 0$$

Frictionless markets

Let C represent the total transaction costs incurred, T represent the taxes on transactions and B represent the bid-ask spread. In a frictionless market, it is assumed that:

$$C = 0$$

$$T = 0$$

$$B = 0$$

In this environment,

- buying or selling the underlying asset incurs no cost
- there are no tax implications for the buying or selling of assets
- the buying price (ask) and the selling price (bid) of the underlying asset are the same

[6] highlights the impact of transaction costs, taxes, and bid-ask spreads on portfolio performance. Specifically, the study demonstrates a strong correlation (0.965) with conventional transaction-level estimations, thus validating this methodology. The Black-Scholes model ignores these factors, which can lead to an overestimation of returns and an underestimation of risks.

Single source of uncertainty

In the Black-Scholes model, the price dynamics of the underlying asset are described by the geometric Brownian motion:

$$dS_t = \mu S_t \, dt + \sigma S_t \, dW_t$$

where W_t is a standard Wiener process representing the single source of randomness in the model.

[7] introduce multifactor models that account for multiple sources of risk, arguing that real markets are influenced by various factors, and that a single-factor model like Black-Scholes oversimplifies financial market complexities.

Constant interest rates

Let r represent the risk-free interest rate. The model assumes:

$$\frac{\partial r}{\partial t} = 0$$

This means that r does not change with respect to time. [8] developed a model that accounts for changing interest rates, showing that interest rates fluctuate over time due to economic factors and monetary policies, impacting the pricing of financial derivatives, thus making this assumption questionable

Many studies and models demonstrated how monetary policies and economic factors cause interest rates to change over time, which affects how financial derivatives are priced. Among them, we can mention [8] or [9], which raises doubts about the validity of the constant interest rates assumption in real markets.

Liquidity

Let L represent the liquidity of the market. The Black Scholes model assumes that :

$$L = \infty$$

This implies that the underlying asset can be bought or sold in any quantity without affecting its price. The impact of liquidity on stock returns is demonstrated by [10], which explains that the Black-Scholes model's assumption of infinite liquidity is unrealistic because real markets have varying levels of liquidity, which affect asset prices and hedging strategies.

In the following table, we summarize the main limitations of the Black-Scholes model :

Table 2.1: Main limitations of the Black-Scholes model

Limitation	**Detail**	**Example of literature mentioning it**
Constant volatility	Assumes volatility remains constant over time	Black and Scholes (1973)
Lognormal distribution	Assumes asset returns follow a lognormal distribution	Merton (1976)
Continuous trading	Assumes trading can occur continuously	Cox and Ross (1976)
Frictionless markets	Ignores transaction costs, taxes, and bid-ask spreads	Baxter and Rennie (1996)
Single source of uncertainty	Only considers price of the underlying asset	Hull and White (1987)
Constant interest rates	Assumes constant interest rates over the life of the option	Heston (1993)
Liquidity	Does not account for market liquidity	Avellaneda and Stoikov (2008)
Risk-neutral valuation	Assumes all investors are risk-neutral	Rubinstein (1985)
No dividends	Assumes underlying assets do not pay dividends	Black and Scholes (1973)
Model calibration	Difficult to calibrate for complex instruments	Derman and Kani (1994)
Jumps in asset prices	Assumes continuous price changes, ignoring jumps	Merton (1976)

2.2 Developments to the Black-Scholes model

In this section, we aim to enhance a literature review of the work that has been made to overcome part of the major limitations of the Black-Scholes model mentioned in section 2.1, which also reflects the efforts to capture the complexities of financial markets more accurately.

2.2.1 Stochastic volatility models

Several stochastic volatility models have been proposed to address the Black-Scholes model's limiting assumption of constant volatility. [5] provided a closed-form solution for European options, allowing volatility to follow its stochastic process, by incorporating mean-reverting variance and the leverage effect, which aligns more closely with observed market behaviors. [11] aimed to capture volatility clustering by allowing current volatility to depend on past shocks and describing a conditional variance of the return series. The intuition behind this comes from the observation that high-volatility periods tend to be followed by high-volatility periods, and low-volatility periods tend to be followed by low-volatility periods.

2.2.2 Jump-diffusion models

Jump-diffusion models were developed to incorporate sudden market jumps to provide a more accurate depiction of asset price dynamics and go beyond the Black-Scholes model's assumption of continuous price paths. [12] allows for sudden price changes by extending the Black-Scholes model with the addition of a Poisson jump process.

2.2.3 Local volatility models

Capturing the local volatility smile is an element that the local volatility models seek to capture in option prices dynamics. [13] described volatility as a function of both the current asset price and time, providing a framework to fit the observed volatility surface.

2.2.4 Stochastic interest rate models

[8] addressed the constant interest rate assumption of the Black-Scholes model by incorporating a mean-reverting stochastic process to interest rates, which improved the pricing and hedging of interest rate-sensitive derivatives.

2.2.5 Multifactor models

[14] developed a framework modeling the whole evolution of the yield curve, incorporating multiple risk factors, thus improving the hedging of complex interest rate derivatives.

2.2.6 Machine learning and deep hedging

In contrast to the developments to the Black-Scholes model mentioned earlier, we emphasise the "model free" nature of the use of certain machine learning hedging methods, which was developed by [3] or [15].

We highlight the choice of a subset of machine learning that is neural networks as a possible alternative to traditional models due to their capacity to learn complicated patterns directly from data. We put the

emphasize on their adaptability and the fact that they use intricate interactions between inputs and outputs that more conventional models could overlook.

We also mention that neural networks have the important benefit of being data-driven, by directly analysing past market data, neural networks are able to identify empirical linkages and patterns that theoretical models could miss. This was showed by [16].

Moreover, nonlinearities and variable interaction effects—which are common in financial markets but challenging to capture with linear models or models with preset structures—are handled by neural networks naturally. As demonstraded by [17], neural networks have the potential to better predict price movements by capturing dependencies and interactions found in limit order books.

[18] emphasized neural networks' capacity to automatically extract pertinent features from raw data, making it possible to find features that might not be obvious and lessening the requirement for manual feature engineering, especially in the case of financial forecasting tasks and time-series data.

[19] identified neural networks as scalable solutions for deep hedging and pricing because of their ability to handle big datasets and high-dimensional problems with efficiency, which was made possible by advances in computer power and optimisation methods.

2.3 Neural Networks

2.3.1 Definition

We first cover the definition of the foundation of our model, which is a feedforward neural network.

Let L be a number of layers, and let $N_0, N_1, \ldots, N_L \in \mathbb{R}$ be the dimensions of the layers. We define activation functions $\sigma_i : \mathbb{R} \to \mathbb{R}$ for each $i = 1, 2, \ldots, L$, and let $A_i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ be affine transformations. A function $F : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ is described as:

$$F(x) = F_L \circ \cdots \circ F_1(x)$$

where

$$F_i = \sigma_i \circ A_i \quad \text{for} \quad i = 1, 2, \ldots, L$$

This function $F(x)$ is known as a neural network.

The activation function σ_i is applied to each component individually. Here, L denotes the number of layers in the network. The values $N_1, \ldots, N_{L-1}$ represent the sizes of the hidden layers, while N_0 and N_L are the dimensions of the input and output layers.

Additional informations relative to neural networks are given in appendix A.

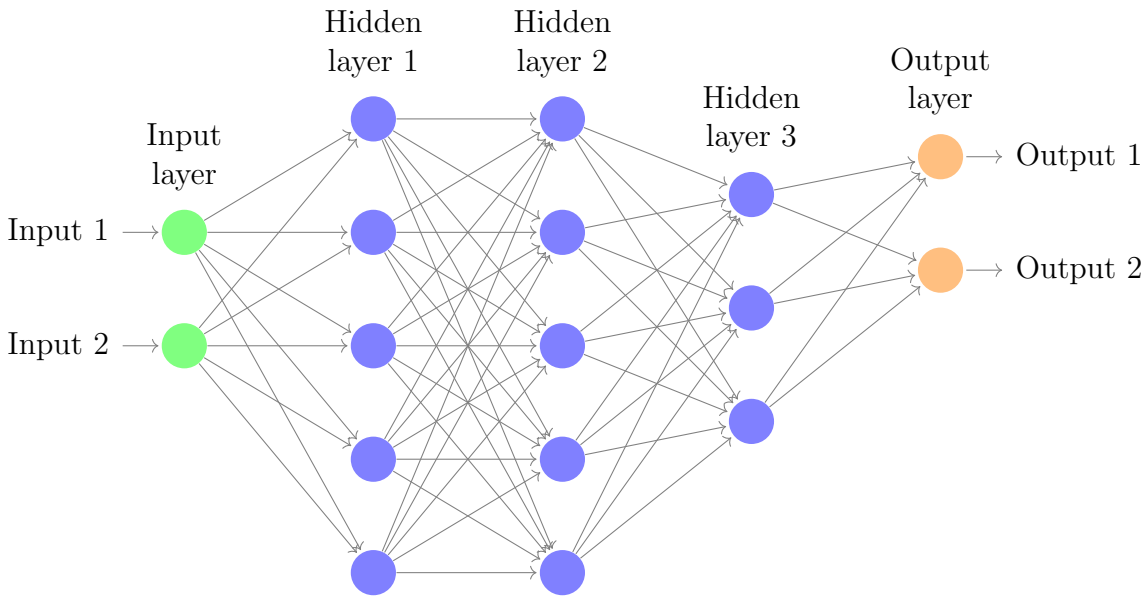


Figure 2.2: An example of a neural network for the following architecture: $L = 4, N_0 = 2, N_1 = N_2 = 5, N_3 = 3$, and $N_4 = 2$.

2.3.2 Recurrent Neural Networks

In the context of analyzing market processes, we are interesting in using a particular form of neural networks that are Recurrent Neural Networks (RNN), to take into account these dependencies into our modelling process.

RNN are series of normal feed-forward neural networks in which the hidden layers send values to the subsequent layers horizontally and to another layer for each next time step, with the application of a different activation function.

We can represent this as :

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

where h_t is the hidden state at time step t, W_{hh} is the weight matrix for the recurrent connections, W_{xh} is the weight matrix for the input to hidden state connections, x_t is the input at time step t, b_h is the bias term, and f is the activation function.

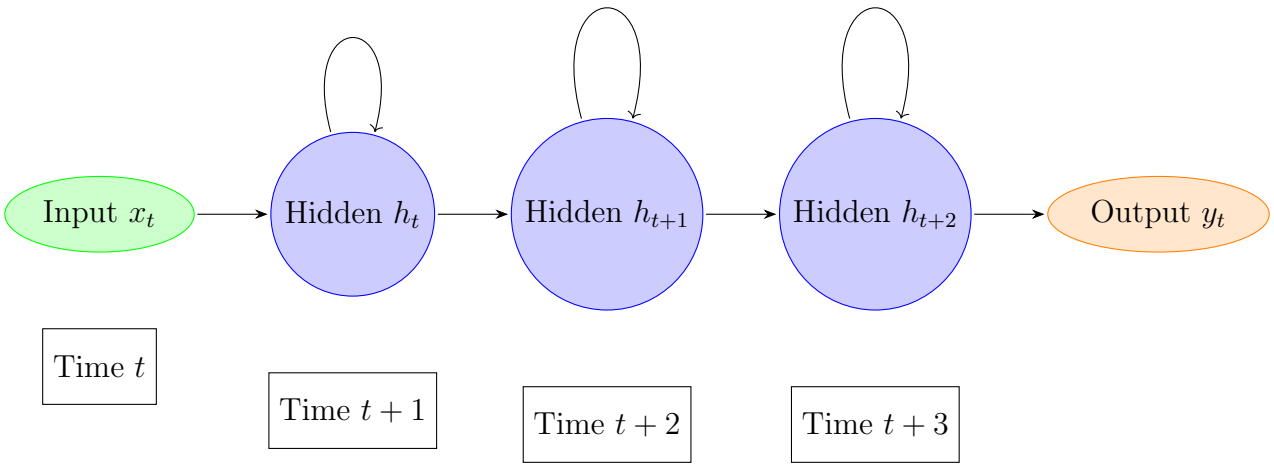


Figure 2.3: Illustration of a Recurrent Neural Network (RNN) : The nodes that loop back to the same layer represent the recurrent connections in a Recurrent Neural Network (RNN), which allow the network to maintain and update its state across different time steps by feeding the output of a node back into itself.

3 Objectives, Specification and Design of the Deep Hedging Algorithm

3.1 Settings

We will work on a discrete-time financial market with horizon T and trading dates $0 = t_0 < t_1 < \cdots < t_n = T$, and consider European call options, defined as contracts providing the holder with the right, but not the obligation, to purchase the underlying asset at a specified strike price K on the maturity date T.

The payoff of a European call option at maturity T is given by:

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

where S_T denotes the underlying stock price at maturity.

We use two historical datasets of underlying stock prices : Microsoft (MSFT) and Mercedes-Benz Group (MBG.DE). Given these historical datasets, we define the rolling At-the-Money (ATM) strike price as follows: we set the strike price K_{t_k} equal to the underlying asset price S_{t_k} at the beginning of each option period within a rolling window of W trading days. This can be mathematically represented as:

$$K_{t_k} = S_{t_k}, \quad \text{for} \quad t_k \in \{t_0, t_W, t_{2W}, \ldots, t_{(M-1)W}\}, \tag{3.1}$$

where $M = \lfloor \frac{N}{W} \rfloor$ is the number of option periods.

We will use this ATM strike price in our model for several reasons. Among them and as developed by [20], this ensures that the strike price remains relevant to current market conditions throughout the study period. For long-term data with possible notable price fluctuations like in our case, this is very crucial. We also refer to [21] in which it in emphasized that ATM strike prices have a high delta sensitivity, which makes them particularly useful in hedging strategies.

3.2 Objectives

We aim to develop a neural network model that uses historical stock price data to predict future prices and determine the best hedging strategies for managing financial risk. The goal is to create a data-driven approach that adapts to various market conditions and enhances the reliability of financial strategies.

Data-driven strategy learns patterns and makes predictions based on real historical data. This makes it possible for the model to adjust to various market circumstances and maybe offer more precise and adaptable hedging solutions. Our objective is to develop a more reliable and efficient approach to financial

risk management by depending on data instead of predetermined formulas.

The Black-Scholes model also assume ideal market circumstances without accounting for the expenses associated with buying and selling assets, hence the model has difficulty resolving market frictions, such as transaction costs. In real-world settings when these costs are substantial, this might result in erroneous pricing and unsuccessful hedging strategies. We aim to take this into account in our algorithm.

We present in the following table the input and output of our neural network :

Inputs	**Outputs**
Closing prices of Microsoft Corporation (MSFT) and Mercedes-Benz Group (MBG) stocks from 1986 to 2024.	Future closing price of the stocks.
	Optimal hedge ratio (delta), i.e. the proportion of the asset that should be hedged, i.e. the proportion of the underlying asset for which we should buy (or sell) a derivative

Table 3.1: Input and Output of the Deep Hedging Neural Network Algorithm

3.3 Black-Scholes model benchmark

For option pricing, a continuous-time framework is offered by the Black-Scholes model. We discretize the Black-Scholes model in order to compare it with our LSTM model, which runs in discrete time (day intervals).

Thus, we aim to "fit the Black Scholes model to our dataset". Precisely, we use a rolling window analysis approach to compute the historical volatility of our datasets that will be used as parameter for our Black Scholes model.

We set up a discrete time grid by assuming 252 trading days a year and a time horizon T corresponding to the number of years on which our historical datasets are running (we describe our datasets later in chapter 4).

For the purpose of our comparison, we use the last 30 days of our dataset to compute the rolling volatility σ_k with a window size of $W = 30$ trading days. The rolling volatility at time t_k is given by:

$$\sigma_k = \sqrt{\frac{252}{W} \sum_{i=k-W+1}^{k} (r_i - \bar{r})^2}, \tag{3.2}$$

where $\bar{r}$ is the mean return over the window:

$$\bar{r} = \frac{1}{W} \sum_{i=k-W+1}^{k} r_i. \tag{3.3}$$

We then calculate for the last 30 trading days t_k, the theoretical price $C(t_k)$ and delta $\Delta(t_k)$ of a European call option using the Black Scholes formula of 2.1. Our approach is then to implement a hedging strategy where the position in the underlying stock is adjusted daily based on the calculated delta.

At each trading day t_k, we adjust the position δ_k in the underlying stock to match the delta of the option, such as :

$$\Delta\delta_k = \delta_k - \delta_{k-1}$$

The comparison methodology between our neural network and Black Scholes hedge is then presented in the following table :

Simulation of hedging strategies:	
LSTM model simulation:	**Black-Scholes model simulation:**
• Use the trained LSTM model to predict the next day's closing price. • Calculate the hedging position based on the predicted price and historical data.	• For each trading day, calculate the theoretical delta using the Black-Scholes formula. • Adjust the hedging position based on the delta.

Table 3.2: Comparison of LSTM and Black-Scholes models

Of course, it is important to specify that this benchmark is by no means an absolute way of comparing our neural networks algorithm with the Black Scholes model. Indeed, the former directly uses historical data as is, whereas the latter is simply a model parameterised on the same historical data. In reality, it is more or less impossible to directly compare the two approaches perfectly. However, this benchmark will be important in our analysis in order to try to understand, compare and analyse the behaviour of the two approaches as well as possible.

3.4 Important remark on the approach to the data used in the algorithm

One piece of data that is particularly difficult to obtain (and even more over long periods, ranging from 5 years to several decades) is the historical price of a given option. In our research, we did not have access to this type of data, which is why it is important to stress that we are not, in particular, comparing the results that can be found by Black Scholes directly to predictions/hedges on option prices (the Black Scholes model, for its part, explicitly and directly prices options). We will use the underlying (in this case our two stocks MBG and MSFT, for which - historical data since the IPO of the companies - are extremely easily and publicly accessible), in order to ultimately compare the hedge ratios and the models' ability to provide the most appropriate portfolio rebalancing possible. In particular, we will seek for the potential of neural networks to learn directly from the statistical characteristics of the input data in order to outperform the Black Scholes model. Our final output will ultimately be the same, and the fact that one model prices the option and the other the underlying does not represent an obstacle to our main objective, which is to analyze in its broadest possible context the potentially innovative capacity of neural networks to hedge a given type of financial product. Obviously, proposing a fully data-driven approach presents limitations and positions as well as assumptions, which we will assume and develop in detail in this dissertation.

3.5 Adding other inputs to our neural network

In order to test the sensitivity of the results of our algorithm, we intend to add other inputs to our neural network that was initially composed of solely the historical underlying stock closing prices data to our theoretical option.

We choose three pertinent extra inputs that the model might get in order to improve its prediction power, each of them of a different nature :

- Relative Strength Index (RSI) as a technical indicator
- Interest rates as a macroeconomic indicator : the historical Federal Funds Effective Rate (FEDFUND) for the MSFT neural network and the historical Euro Main refinancing operations fixed rate for the MBG neural network
- Market index performance as a market indicator : the S&P500 historical data index for the MSFT neural network and the DAX historical data index for the MBG neural network

Our assumption is that these additional inputs will enable us to improve the performance of our neural network hedge. The potential results it will produce will enable us to confirm this hypothesis (or not). Indeed, the neural network may modify the hedging ratios more precisely during times of high volatility or trend changes by integrating RSI. Interest rates could enhance the model's capacity to forecast changes in prices, and market index performance could enable the model to more effectively hedge against market-wide occurrences by adjusting the hedging ratios in accordance with overall market changes.

4 Building a Neural Networks Deep Hedging Strategy Model against Black-Scholes : Methodology and Implementation

In this chapter, we define the data and parameters used in our algorithm, as well as the transformations made on them. Particularly, we describe the architecture of the long short-term neural network used. We try to computationally justify our choices, as well as support them with literature. We allow ourselves to refine parameters and numbers found and chosen in this chapter during the later running of our algorithm, depending on our results.

The Python ML library we choose to use in this dissertation is TensorFlow with Keras as our open-source deep learning API.

4.1 Data for the algorithm

4.1.1 Raw starting data

We perform our algorithm on two sets of historical data :

- the Microsoft stock (MSFT), with daily historical closing price data ranging from the 03/14/1986 to the 03/21/2024, with a size of approximately 9.5×10^3 data points.
- the Mercedes-Benz Group stock (MBG), with daily historical closing price data ranging from the 10/30/1986 to the 03/21/2024, with a size of approximately 7×10^3 data points.

Our choice to train two sets of historical data comes from our will to see the results of our algorithm on two different types of stocks. Indeed, the MSFT stock historically has a very high performance and upward trend, while the MBG stock is a much more stable asset in terms of price.

4.1.2 Data preprocessing

We attempt to make the scale and distribution of the input data consistent in order to improve the performance of our neural networks. Gradient descent is an optimisation process used in neural network training, and scaling helps to accelerate its convergence. Additionally, we aim to normalise our input data, which can enhance our neural network's stability and performance.

In this instance, the scaler that we are employing is MinMaxScaler. As discussed in [22], the fact that our data is a time series of stock prices is what led us to select this specific scaler rather than other types, like StandardScaler. MinMaxScaler is helpful when the data lacks a Gaussian (normal) distribution, as it does

(a) MBG historical closing prices from 1996 to 2024.

(b) MSFT historical closing prices from 1986 to 2024.

in our situation with stock prices, as it adjusts the data to a specified range, often [0, 1]. Neural networks, in our example, are among the algorithms that benefit greatly from this scaler since they do not presuppose

Property	Value
Count	7023.000000
Mean	46.158766
Std	14.736651
Min	14.505675
25%	33.860439
50%	44.881092
75%	57.318701
Max	85.052834

Table 4.1: Statistical Properties of MBG.DE Closing Prices

Property	Value
Count	9583.000000
Mean	54.984367
Std	83.172988
Min	0.090278
25%	5.609375
50%	27.260000
75%	46.020000
Max	429.369995

Table 4.2: Statistical Properties of MSFT Closing Prices

any distribution in the input data.

There are several arguments for this but we present a few of them here, based on [23] :

1. The MinMaxScaler transforms data x to x' in the range $[a, b]$ using the formula :

$$x' = a + \frac{(x - x_{\min}) \cdot (b - a)}{x_{\max} - x_{\min}}$$

   where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values in the dataset, respectively.

2. For non-stationary data with a changing scale over time, MinMaxScaler adapts to dynamic range :

$$x'_t = \frac{x_t - \min(x)}{\max(x) - \min(x)}$$

   where x_t represents the value at time t, and $\min(x)$ and $\max(x)$ are recalculated as the data evolves.

3. Outliers directly affect the scaling :

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

   If $x_{\max}$ or $x_{\min}$ are outliers, the scaled values will be disproportionately affected.

4.2 Neural networks architecture

We describe the Long Short-Term Memory (LSTM) neural network architecture used in our deep hedging algorithm in detail. The architecture is ideal for tasks involving the analysis and prediction of financial data because it is specifically made to handle time-series data. The Mercedes-Benz Group (MBG) and Microsoft (MSFT) historical stock prices are used to train the network to forecast future stock values, which are then

utilised to implement hedging strategies.

4.2.1 Long Short-Term Memory (LSTM) neural network

We choose to use Long Short-Term Memory (LSTM) neural networks, a kind of Recurrent Neural Network (RNN) that is especially made to recognise and learn temporal connections in sequential input. LSTMs have memory cells and gating mechanisms, and they can keep and use information for longer than typical RNNs, which are inefficient for long sequences due to the vanishing gradient problem.

The input, forget, and output gates—which control the information flow into, out of, and through the memory cells—are the three gates that allow for the maintenance of long-term dependence. By enabling LSTMs to choose what data to retain, what to reject, and what to output at each stage, the gating mechanisms offer a potent way to model sequences where context is missing.

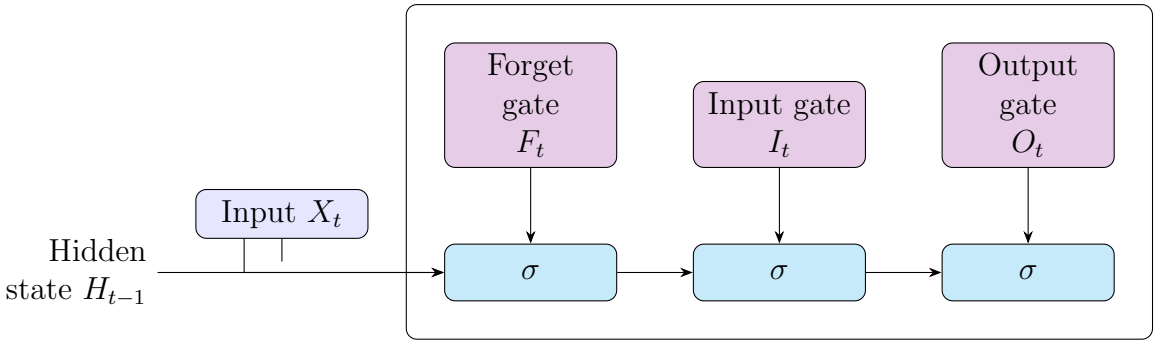


Figure 4.1: Illustration of the gating mechanism in a Long Short-Term Memory (LSTM) neural network, featuring three gates: the Forget gate (F_t), the Input gate (I_t), and the Output gate (O_t), with the hidden state from the previous time step (H_{t-1}) and the input at the current time step (X_t) being processed through these gates to manage the flow of information within the LSTM cell.

Literature support the use of Long Short-Term Memory (LSTM) neural networks for deep hedging algorithms, especially in predicting future stock prices and hedge ratios for equities derivatives like European call options. This reasoning is based on LSTMs' inherent skills to handle sequential data and their capacity to detect temporal correlations and long-term dependencies, both of which are critical for financial time series research.

Activation function

When choosing the activation of our neural network, we intend to handle a common challenge in training deep neural networks, known as the vanishing gradient problem. As discussed by [26], the network is essentially stopped from learning during backpropagation when the gradients of the loss function with respect to the weights get extremely small.

When the gradient $\delta^{[l]}$ becomes very small, it results in very small updates to the weights $W^{[l]}$, leading to the vanishing gradient problem.

We choose the Rectified Linear Unit (ReLU) activation function for our algorithm because it helps mitigate the vanishing gradient problem by keeping the gradient for positive input values non-zero.

The ReLU activation is defined as:

$$a^{[l]} = \text{ReLU}(z^{[l]}) = \max(0, z^{[l]}) \tag{4.1}$$

The derivative of the ReLU function is:

$$\frac{\partial \text{ReLU}(z)}{\partial z} = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{4.2}$$

This derivative ensures that for $z > 0$, the gradient does not vanish:

$$\delta^{[l]} = \frac{\partial L}{\partial z^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \cdot \frac{\partial \text{ReLU}(z^{[l]})}{\partial z^{[l]}} \tag{4.3}$$

The gradient with respect to the weights $W^{[l]}$ is then:

$$\frac{\partial L}{\partial W^{[l]}} = \delta^{[l]} \cdot (a^{[l-1]})^T \tag{4.4}$$

ReLU prevents the vanishing gradient issue and keeps gradients non-zero for positive input values, which facilitates effective network learning.

Choice of a dense layer

[27] discussed the Universal Approximation Theorem, which asserts that a feedforward neural network with at least one hidden layer and a finite number of neurons can approximate any continuous function to any desired degree of accuracy, given that it has enough neurons, lends support to the inclusion of a dense (fully connected) layer. This theorem supports dense layers' ability to learn intricate input-to-output mappings, which is crucial for hedging position computation and stock price prediction. This is why we choose to use a dense layer in our configuration.

Input shape

To capture enough historical background without being unnecessarily lengthy, a series length of 30 days (about one month) is selected, which aids in lowering the computational load. Furthermore, every input sequence has just one feature—the closing price. We use [28], suggesting that stock prices often exhibit monthly cycles, to justify an effective 30-day window to catch these monthly trends. We complete this with [29] to justify that the LSTM model learns dependencies more efficiently and prevents overfitting with shorter sequences.

Mathematically, let S be the time series of closing stock prices:

$$S = \{s_1, s_2, \ldots, s_T\}$$

We divide S into overlapping sequences of length 30. Each sequence X_i is defined as:

$$X_i = \{s_i, s_{i+1}, \ldots, s_{i+29}\}$$

where $i \in \{1, 2, \ldots, T - 29\}$.

The target for each sequence is the next day's closing price:

$$y_i = s_{i+30}$$

This approach results in a dataset of input-output pairs (X_i, y_i) for training the LSTM model:

$$\{(X_i, y_i) \mid i \in \{1, 2, \ldots, T - 30\}\}$$

We have data on MSFT stock spanning from 1986 to 2024, which we have divided into many 30-day sequences. To learn and forecast the price for the following day, the algorithm examines every 30-day sequence.

Formally, the input to the LSTM model is a tensor X of shape $(n\_samples, 30, 1)$, where $n\_samples$ is the number of 30-day sequences in the dataset.

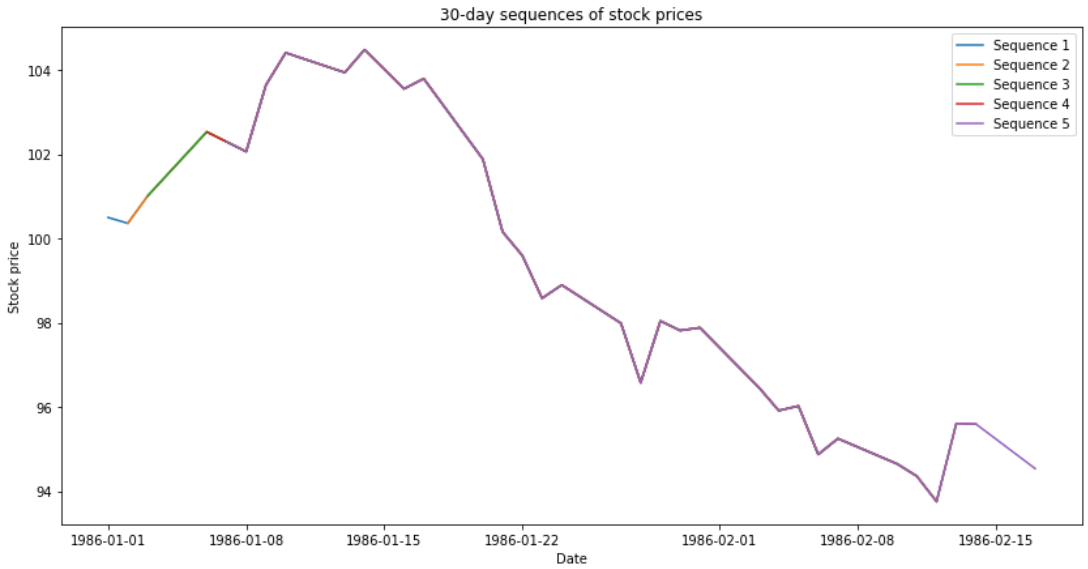


Figure 4.2: 30-day sequences of stock prices

Optimizer

As the optimizer, we choose Adam, that was presented in [30]. It was demonstrated how Adam performs better and reaches convergence more quickly than other stochastic optimisation techniques like RMSProp and AdaGrad because it can adjust the learning rate for each parameter.

Wen et al. showed that utilising Adam as an optimizer produced a lower mean squared error (MSE) on the test set compared to SGD and Adagrad in a study on deep learning for stock price prediction. With Adam, the MSE was lowered by about 15

[31] found that Adam offered more steady and consistent convergence in their work on financial time series forecasting, which decreased the possibility that the optimizer would become stuck in a local minimum or encounter strong oscillations while being trained.

Here we summarise some of the results of this work:

Optimizer	Test MSE
Adam	0.0021
SGD	0.0025
Adagrad	0.0024

Table 4.4: Test MSE for different optimizers in LSTM stock price prediction (Wen et al., 2020).

Optimizer	Convergence Epochs	Final Loss
Adam	30	0.015
SGD	50	0.020
RMSProp	40	0.018

Table 4.5: Convergence and final loss for different optimizers in financial time series forecasting (Liang et al., 2018).

Loss function and number of neurons

We perform a computational analysis on our two datasets (MBG and MSFT) in order to find the most suitable starting loss function for our model as well as the most optimal number of neurons for both datasets. Eventually, depending on the results of our algorithm, we give ourselves the freedom to test other loss functions, but we use this to define a choice of reference.

Based on [32] and [33], we choose to compare between Mean Squared Error (MSE) and Mean Absolute Error (MAE). We compute both of these loss functions between the actual and predicted closing prices for our two datasets.

Let $\hat{y}_i$ be the predicted value and y_i be the actual value for the i-th data point. The Mean Squared Error is defined as:

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

The Mean Absolute Error is defined as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

We choose the Mean Absolute Error (MAE) for MBG because we consider it is better suited for data with lower volatility and fewer severe price swings and that it is more resilient to outliers. We chose the Root Mean Squared Error (RMSE) for MSFT because it is more responsive to large deviations and outliers, which is important in controlling the high volatility and notable price swings seen in the MSFT dataset. By using this method, we can make sure that our model takes into account the unique volatility and risk characteristics of every stock.

Neurons	MSE (MBG)	MAE (MBG)	MSE (MSFT)	MAE (MSFT)
10	0.000296	0.012830	0.000023	0.002164
20	0.000236	0.011081	0.000028	0.002586
50	0.000202	0.010239	0.000024	0.002228
100	0.000216	0.010772	0.000040	0.004953
200	0.000326	0.014186	0.000033	0.002690

Table 4.6: Mean Squared Error (MSE) and Mean Absolute Error (MAE) for different neuron counts in LSTM model for MBG and MSFT datasets.

According on our findings, selecting 20 neurons for MBG and 200 neurons for MSFT seems sense. Given that the MSE is lowest at 10 neurons and the MAE is lowest at 20 neurons for MBG and that the MAE drops with increasing numbers of neurons while the MSE is consistently low across all neuron counts for MSFT, these choices provides a decent balance between the two metrics in both situations.

Batch size and epochs

In order to find the optimal batch size and epochs, we combine two methods :

- Based on [34], [35] and [36] for the batch size and [37], [33] and [38] for the epochs, we first conduct manual exploration using batch sizes of 16, 32, and 64, and epochs of 20, 30, and 50.
- Then, we adjust our parameter ranges based on the preliminary findings and employ a grid search, identifying the best combination of batch size and epochs based on the lowest mean squared error.

We show below the results of our computation :

Batch size	Epochs	MBG RMSE	MSFT RMSE	Training time (s)
16	20	0.013332	0.005198	229.684551
16	30	0.013298	0.004761	346.106342
16	50	0.015369	0.004431	552.772829
32	20	0.013666	0.004791	133.563448
32	30	0.015311	0.004340	214.703012
32	50	0.014437	0.004877	348.273082
64	20	0.017326	0.005360	91.086733
64	30	0.015115	0.006423	138.660685
64	50	0.015713	0.004421	228.936246

Table 4.7: Initial manual exploration results for batch size, epochs, RMSE, and training time

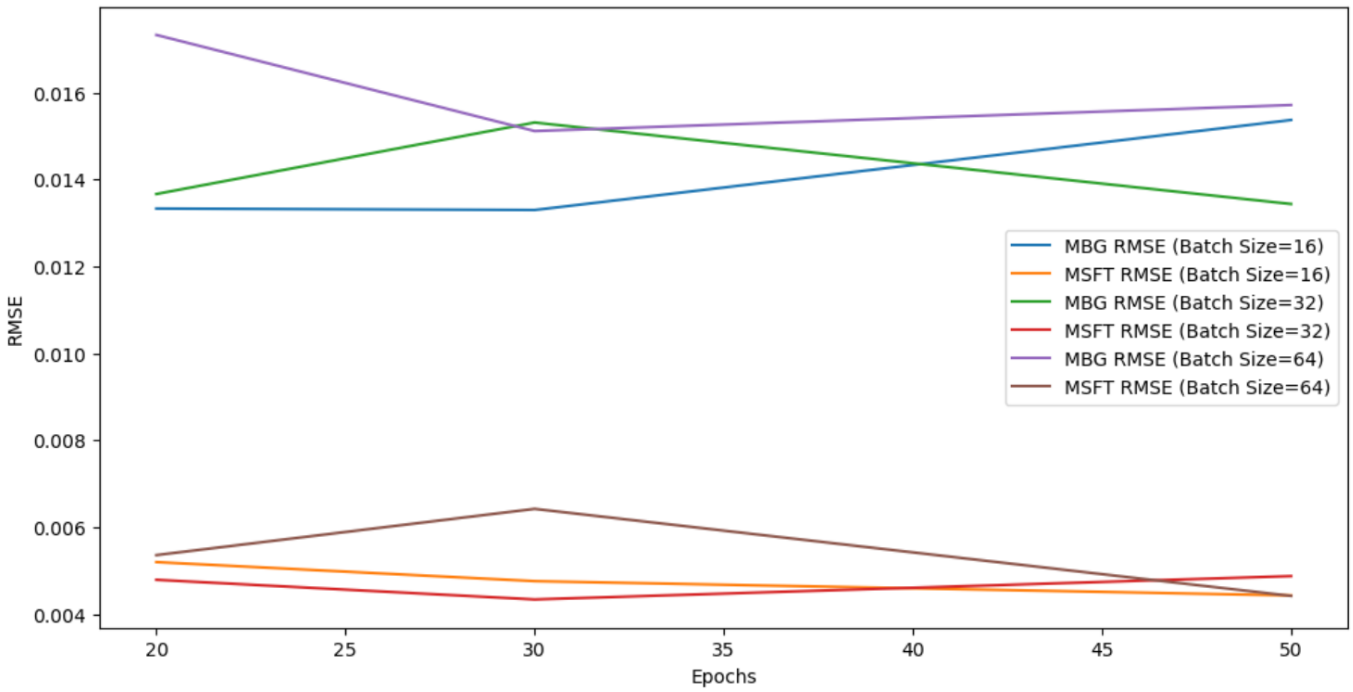


Figure 4.3: RMSE for different batch sizes and epochs

Training periods become longer (16) for smaller batch sizes and shortened (64) for bigger batch sizes. A trade-off exists between RMSE performance and training time. Though they perform better, smaller batch sizes necessitate much more training time.

For MBG, the optimal RMSE performance is obtained with a batch size of 16 and 30 epochs. Nevertheless, batch size 32 with 30 epochs also offers good performance with shorter training times when taking the training time trade-off into account. The best RMSE performance for MSFT is achieved with a batch size of 32 and 30 epochs, which balances error and training duration.

Number of LSTM layers

Given the nature of the task (predicting future stock prices and hedge ratios based on historical data) we choose to start with two LSTM layers. With this configuration, the model can represent increasingly intricate temporal connections without being overly complex. This is supported by [29]. More details are given on this in appendix A3.

- Primary temporal dependencies are captured in the first LSTM layer.
- Higher-level temporal patterns and interactions are captured by the second LSTM layer.
- To produce the final prediction, a dense layer is added after the LSTM layers.

Final configuration

Based on our previous analysis, we choose the following parameters for our algorithm :

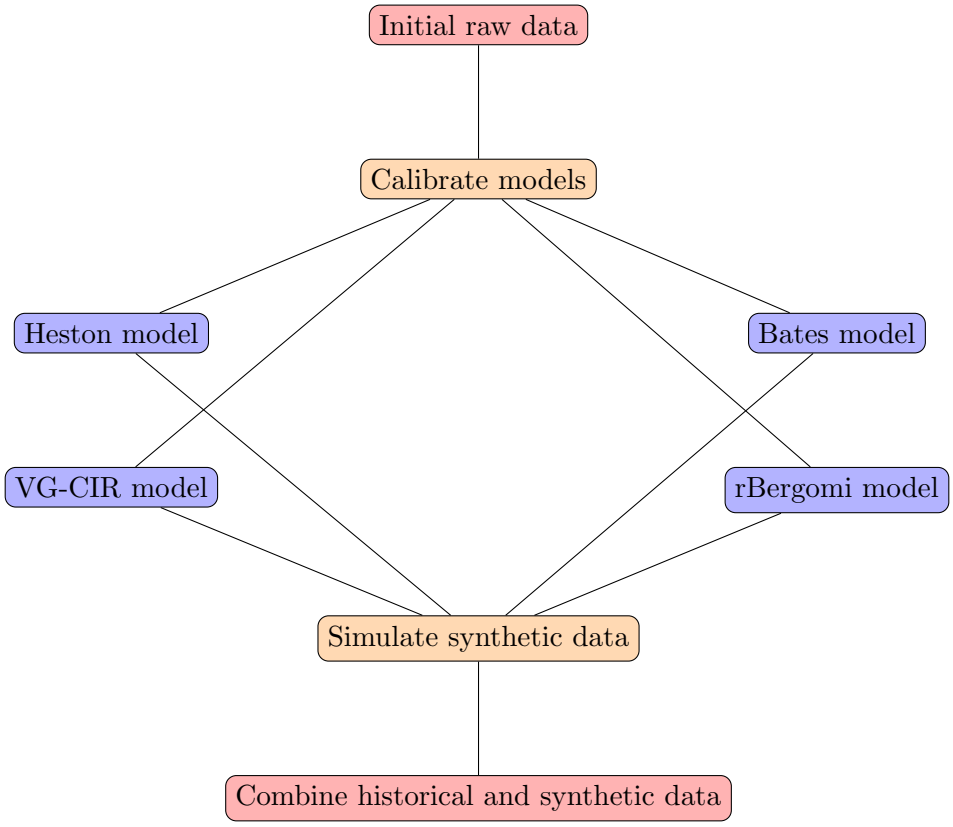
Characteristics	**Choice in our algorithm**
Type of NN	LSTM
Input data	Historical stock prices for Microsoft Corporation (MSFT) and Mercedes-Benz Group (MBG)
Output data	Future prices of underlying (stock price) and hedge ratio (delta)
Number of neurons	20 for MBG and 200 for MSFT
Number of LSTM layers	2
Activation function	ReLu
Choice of a dense layer	Yes
Input shape	Series length of 30 days
Optimizer	Adam
Loss function	Mean Absolute Error for MBG and Root Mean Squared Error for MSFT
Batch size	16 for MBG and 32 for MSFT
Epochs	30 for both MBG and MSFT

Table 4.8: Characteristics of the algorithm's neural network

4.3 Robustness of the neural network hedging algorithm

Only the actual market conditions are represented by historical data, which sometimes fails to capture extreme occurrences or uncommon market scenarios. In order to assess the robustness and adaptability of our algorithm, we will be fitting the model to data paths governed by a variety of stochastic processes. By doing so, we want to test the neural network's ability to generalize and perform reliably across different market scenarios, thereby providing a more rigorous evaluation of its practical applicability and resilience.

We evaluate the robustness of our algorithm by selecting four stochastic models : the Heston model, Bates model, VG-CIR model, and the rBergomi model. Each of these models captures different aspects of market dynamics and volatility, providing us with a solid testing framework. We further develop their characteristics in appendix A. and go into detail about our calibration in appendix B.



4.3.1 Heston Model

Heston incorporated stochastic volatility to the Black-Scholes model, and allows the volatility of the asset price to follow a mean-reverting square-root process. Using this model enables us to capture a certain observed market phenomenon where volatility is not constant but varies over time.

$$dS_t = \mu S_t dt + \sqrt{V_t} S_t dW_t^S \tag{4.5}$$

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^V \tag{4.6}$$

where W_t^S and W_t^V are two Wiener processes with correlation ρ.

4.3.2 Bates Model

Bates proposed an extension of the Heston model by including a jump diffusion process, which enables us to handle abrupt jumps in asset prices, common in real financial markets.

$$dS_t = \mu S_t dt + \sqrt{V_t} S_t dW_t^S + S_t dJ_t \tag{4.7}$$

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^V \tag{4.8}$$

where J_t is a Poisson jump process with intensity λ and jump size distribution.

4.3.3 VG-CIR Model

By combining the Variance Gamma (VG) model with the Cox-Ingersoll-Ross (CIR) process, we can build a framework to capture both the jumps and stochastic variance of asset returns.

$$dS_t = S_t(rdt + dZ_t) \tag{4.9}$$

$$dV_t = \kappa(\theta - V_t)dt + \sigma\sqrt{V_t} dW_t^V \tag{4.10}$$

where Z_t is a Variance Gamma process and V_t follows a CIR process.

4.3.4 rBergomi Model

Bayer, Friz, and Gatheral proposed a rough volatility framework enabling us to capture the empirical roughness observed in volatility time series.

$$dS_t = S_t\sqrt{V_t} dW_t \tag{4.11}$$

$$V_t = V_0 \exp\left(\eta \int_0^t (t-s)^{H-0.5} dW_s\right) \tag{4.12}$$

where W_t is a Brownian motion and H is the Hurst parameter.

4.4 Accounting for market frictions

4.4.1 Environment

The Black Scholes model requires the implementation of a continuous delta-hedging which becomes hugely expensive in the presence of market frictions. Incorporating transaction costs damages at least substantially the great majority of the theoretical outcomes.

As discussed by [39], for small to moderate trades sizes, transaction costs are roughly linear in many markets, particularly those with significant liquidity. For practical purposes, this makes linear costs an acceptable approximation. In order to address optimal execution and account for transaction costs, new models have been developed, such as the one developed by Almgren and Chriss, but they lack the closed-form solutions of the Black-Scholes formula and frequently requires numerical methods for solution.

We define linear transaction costs as :

$$c(n) = \sum_{i=1}^{d} c^i S_t^i |n^i|$$

with d the dimension of the underlying assets or the number of different assets, c^i the proportional transaction cost rate for the i-th asset, S_t^i,n^i the number of units of the i-th asset being transacted and $|n^i|$ the absolute value of the number of units of the i-th asset being transacted (to ensure the cost is non-negative).

We must change the loss function to include transaction costs in order to incorporate market frictions into our neural network model.

We define our new loss function as :

$$\mathcal{L}_{\text{new}} = \mathcal{L}_{\text{original}} + \lambda \sum_{i=1}^{d} c^i S_t^i |n^i|$$

where $\mathcal{L}_{\text{new}}$ is the new loss function that includes transaction costs, $\mathcal{L}_{\text{original}}$ is the original loss function (e.g., Mean Absolute Error for MBG or Root Mean Squared Error for MSFT), λ is a weighting factor to balance the original loss and the transaction costs.

We give an example illustrating further the impact of transaction costs in appendix D.

4.4.2 Refined Black-Scholes benchmark

In order to keep a relevant Black-Scholes benchmark for comparing the results of our neural network algorithm, and since the Black-Scholes model does not account for transaction costs, we adapt it by using a transaction cost adjusted delta hedging. After calculating the delta using the Black-Scholes formula at each time step, we choose to adjust the portfolio to match the new delta, incurring a transaction cost proportional to the change in delta and track the portfolio value and transaction costs over time.

If Δ_{t-1} is the delta from the previous time step, the change in delta is $\Delta_t - \Delta_{t-1}$.

The transaction cost incurred at time t is proportional to the change in delta and the current stock price:

$$\text{Transaction Cost}_t = k \cdot |\Delta_t - \Delta_{t-1}| \cdot S_t$$

The initial portfolio value V_0 is set to the Black-Scholes price P_{BS} of the option at $t = 0$.

$$V_0 = P_{\text{BS}}(S_0, K, T, r, \sigma)$$

$$\Delta_0 = P_{\text{BS}}(S_0, K, T, r, \sigma)$$

At each time step t, the portfolio value V_t is updated to reflect the new delta and any transaction costs incurred:

$$V_t = \Delta_t \cdot S_t + (V_{t-1} - \Delta_{t-1} \cdot S_{t-1}) \cdot e^{r\Delta t} - (C_t)$$

Here, $\Delta_t \cdot S_t$ is the value of the stock holdings, $(V_{t-1} - \Delta_{t-1} \cdot S_{t-1}) \cdot e^{r\Delta t}$ is the value of the cash holdings after accruing interest over the time interval Δt, and (C_t) is subtracted to account for the cost of adjusting the delta.

5 Results, Analysis and Evaluation

5.1 Pure data benchmark with Black-Scholes

Here, we aim to evaluate the performance of both the Black-Scholes model and the neural network model in hedging an options portfolio.

In order to provide an even more detailed and evocative analysis of the comparison between our algorithm and our Black-Scholes benchmark, we construct a Profit and Loss function. We start with an initial portfolio consisting of one long position in the European call option and a short position in the underlying asset to be delta-neutral. We adjust the hedge by buying or selling the underlying asset to maintain the delta-neutral position based on the model's predicted delta. We assume to rebalance daily and we do not consider transaction costs.

Below, we go in details into the results found :

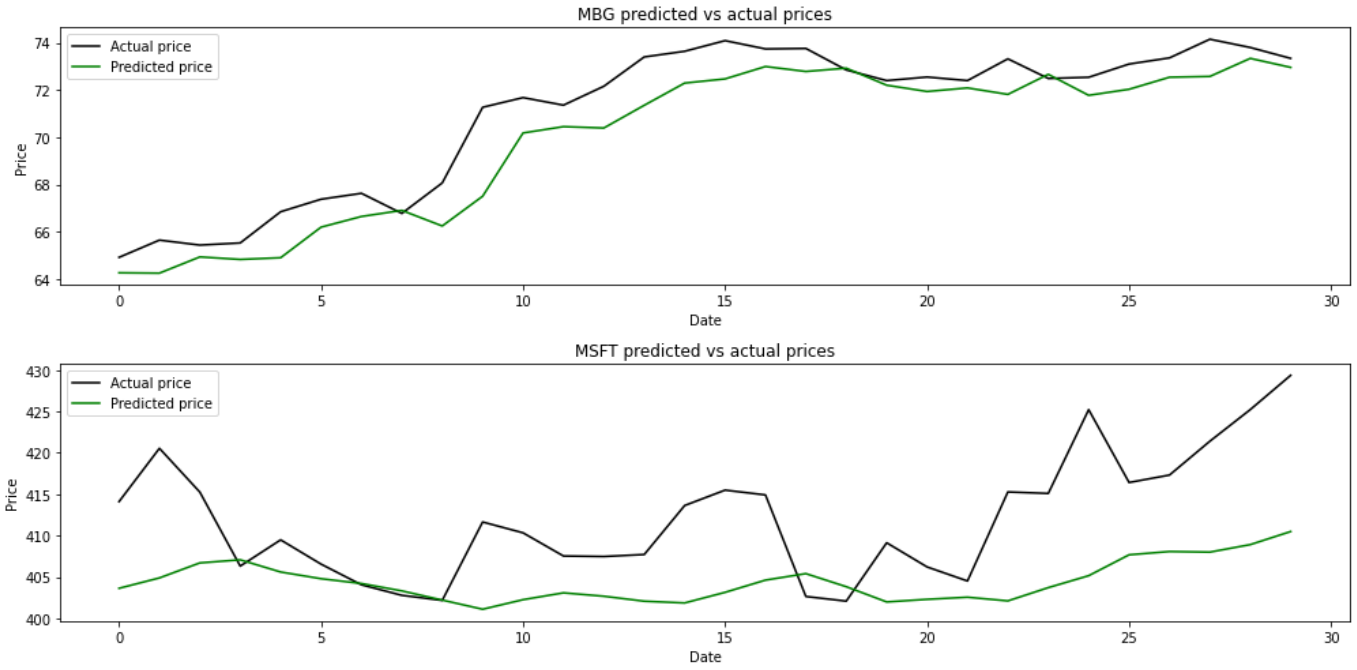


Figure 5.1: NN predicted prices vs actual historical prices over the datasets last 30 days (initial configuration of the neural networks)

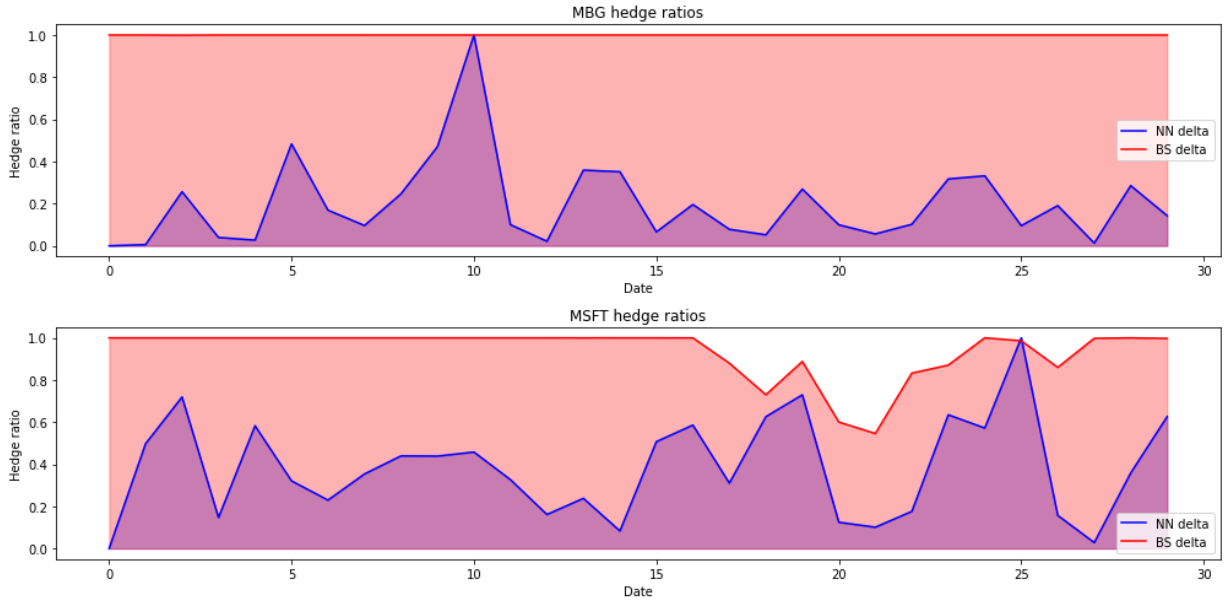


Figure 5.2: NN vs BS hedge ratios over the datasets last 30 days (initial configuration of the neural networks)

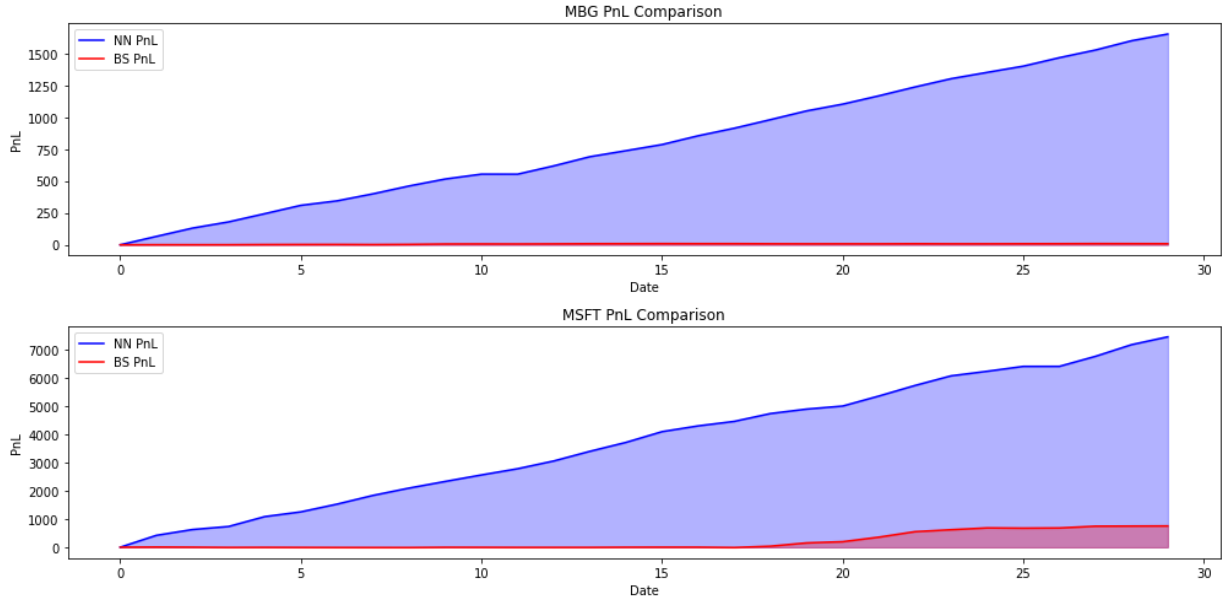


Figure 5.3: Comparison of the PnL of the NN and BS hedge over the last 30-days period (initial configuration of the neural networks)

In the case of MBG stock, a good match is shown by the predicted prices from the NN model being generally close to the actual prices. The NN model's forecasted prices for MSFT show more notable departures from the overall trend, particularly during moments of volatility.

The disparity in performance levels between MBG and MSFT suggests that asset-specific modification of the NN model may be necessary. This involves making adjustments for varying degrees of volatility and fluctuations in the market.

For both stocks, the NN hedge ratios exhibit more variability compared to the BS model, reflecting the dynamic nature of the NN approach. The NN model continuously learns and updates its parameters depending on new data, in contrast with BS model (as illustrated by our benchmark), which rely on static parameters and assumptions. As a result, it can react to the constantly shifting market conditions more successfully.

For MBG, the NN model significantly outperforms the BS model in terms of cumulative PnL and raw returns, indicating better capturing of profitable trading opportunities. The NN model also outperforms the BS model for MSFT, though the difference is less pronounced compared to MBG. We may attribute this to the NN model's ability to adapt to market conditions and capture short-term trends.

NN's greater overall profitability is also confirmed in terms of risk-adjusted measures, as shown in the table below :

Metric	**MBG**	**MSFT**
Sharpe ratio	0.6722	1.0019
Maximum drawdown	-0.000583	0.0331

Table 5.1: Summary of hedging performance metrics for initial configuration

The NN hedging strategy shows robust performance, particularly for MSFT with a Sharpe Ratio of 1.0019, indicating a good risk-adjusted returns. The small drawdowns for MBG and MSFT are influenced by the short 30-day hedging period, which limits significant adverse price movements.

We also test the sensitivity of the algorithm to changes in two crucial parameters : the number of neurons and the loss functions used in the training process. In addition to testing each stock on the other's initial loss function (MAE and RMSE), we add two other loss functions on which to train our datasets: Huber and log-cosh loss. We give more details about these loss functions in appendix D. The number of neurons we test is [10, 15, 20, 30, 40] for MBG and [150, 180, 200, 220, 250] for MSFT.

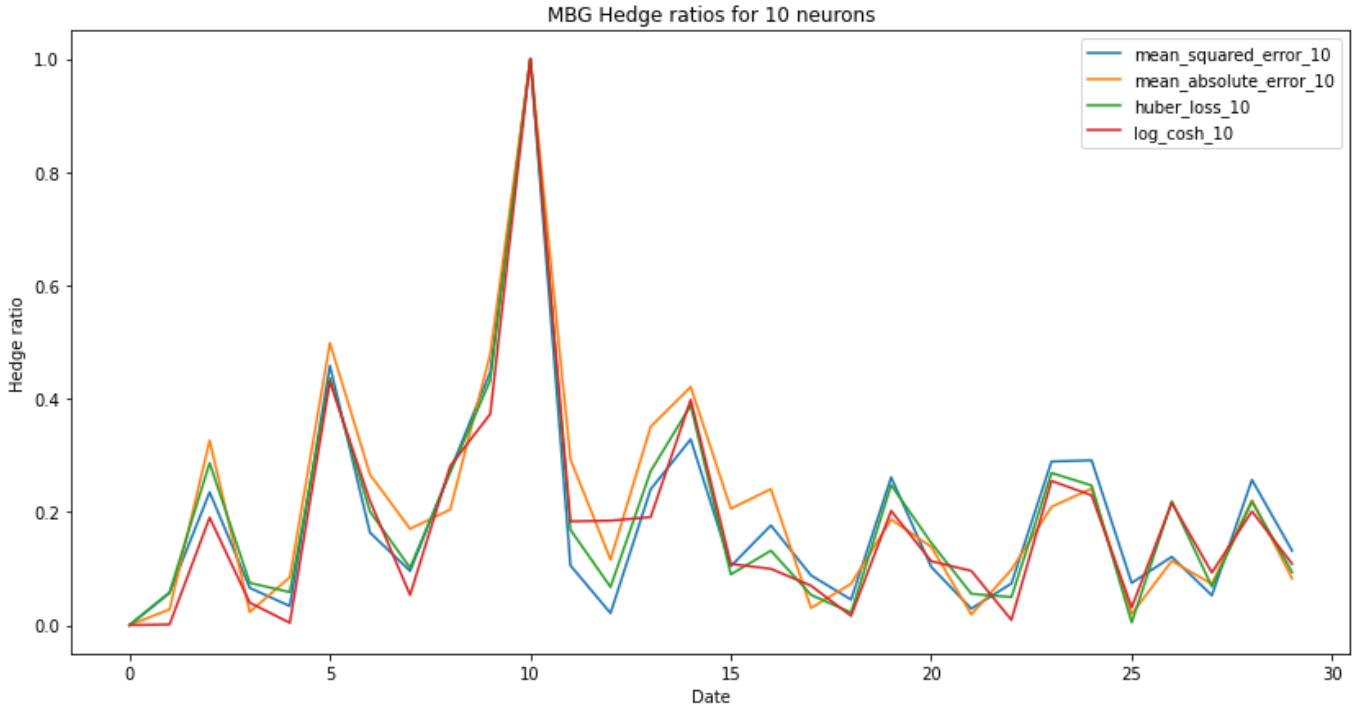


Figure 5.4: NN deltas for different loss functions and a number of neurons of 10 over the last 30 days of MBG dataset

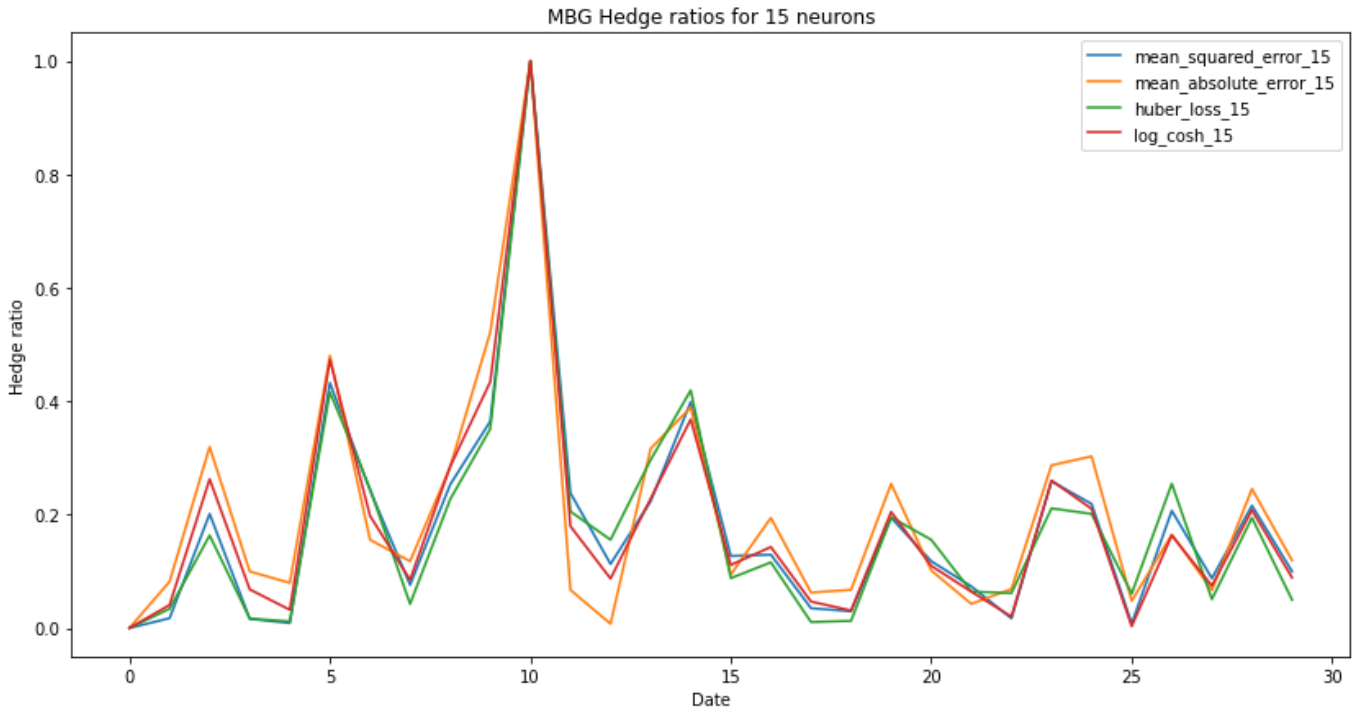


Figure 5.5: NN deltas for different loss functions and a number of neurons of 15 over the last 30 days of MBG dataset

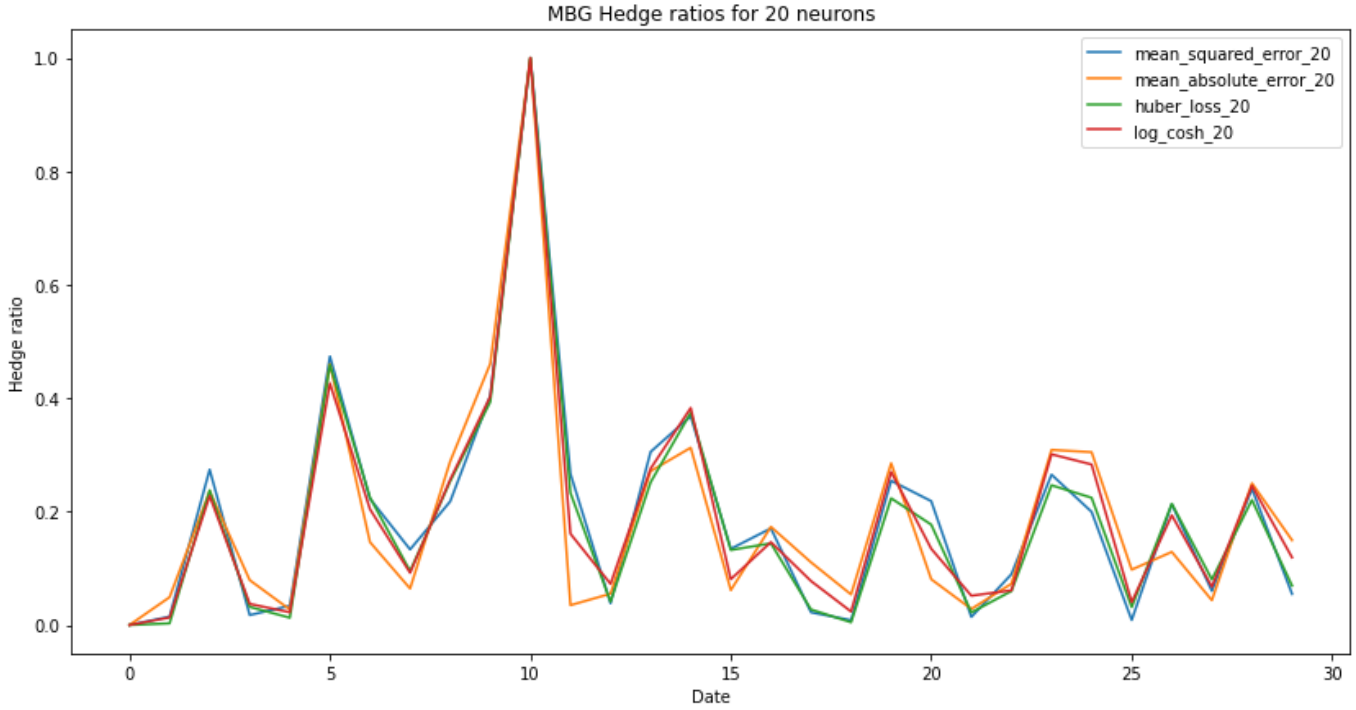


Figure 5.6: NN deltas for different loss functions and a number of neurons of 20 over the last 30 days of MBG dataset

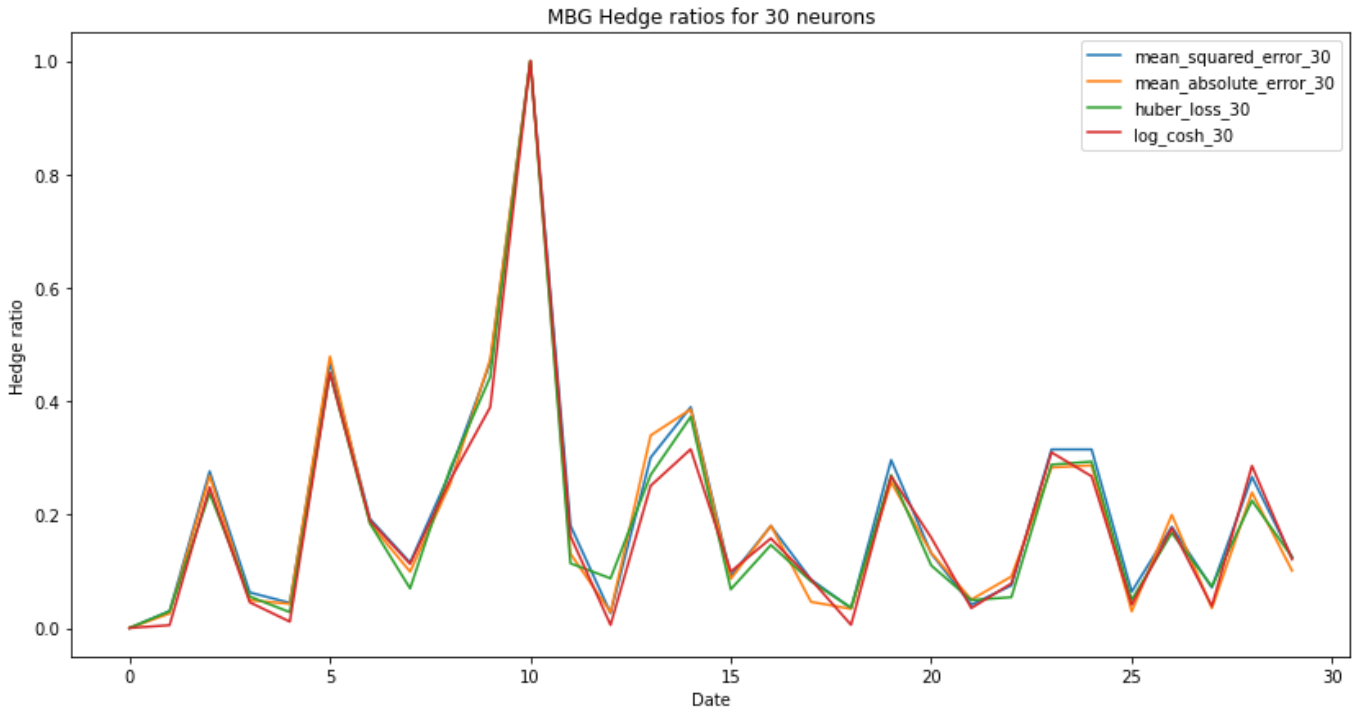


Figure 5.7: NN deltas for different loss functions and a number of neurons of 30 over the last 30 days of MBG dataset

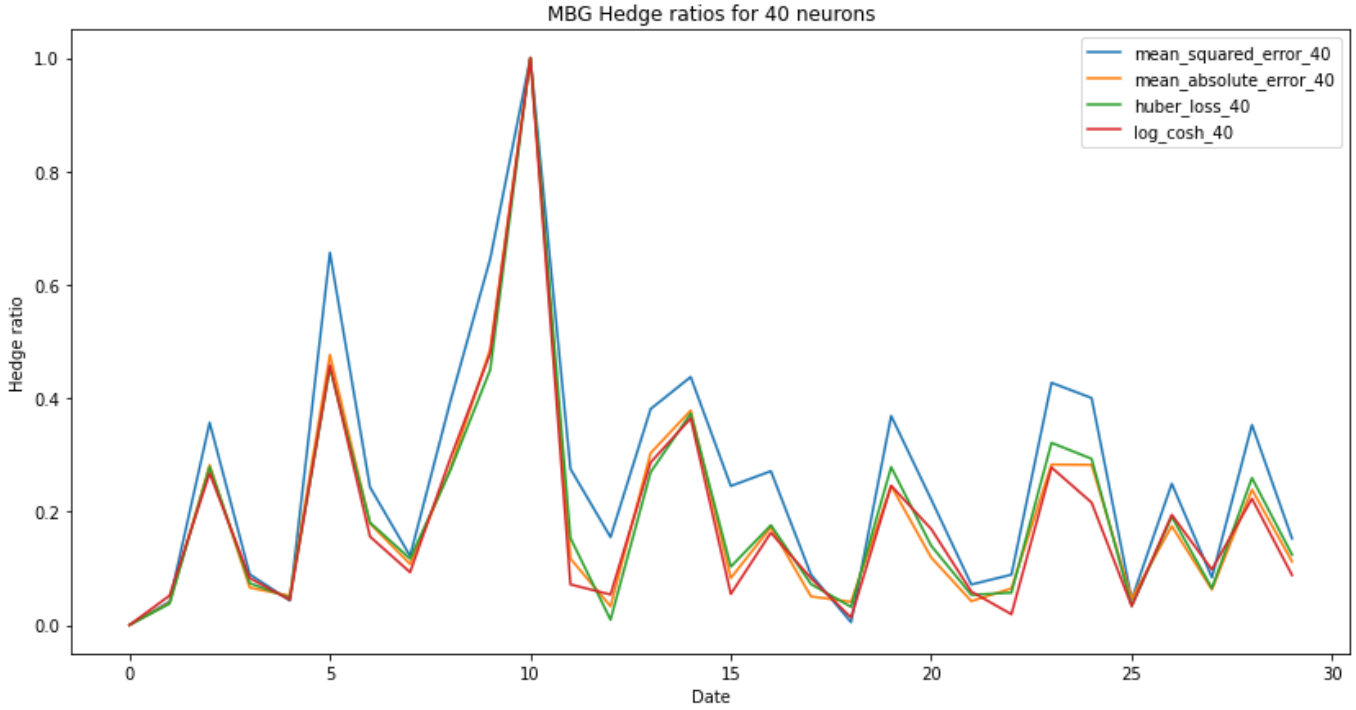


Figure 5.8: NN deltas for different loss functions and a number of neurons of 40 over the last 30 days of MBG dataset

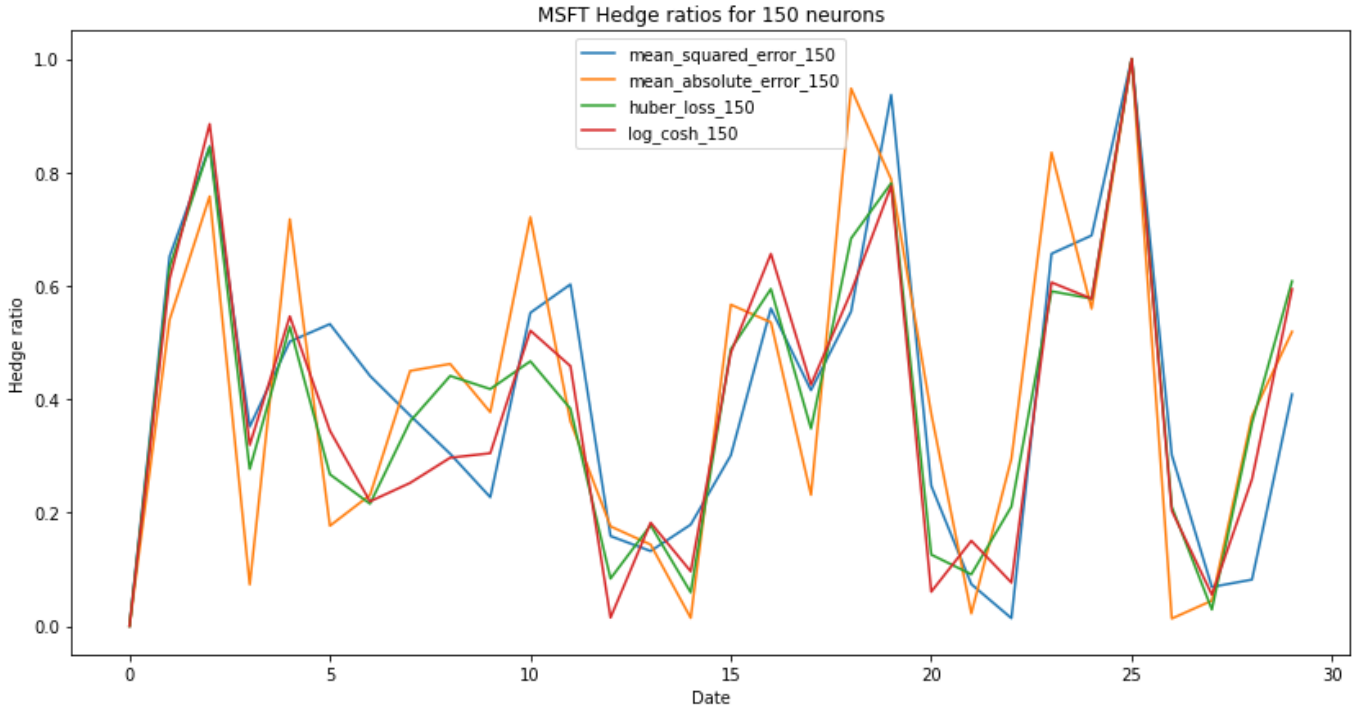


Figure 5.9: NN deltas for different loss functions and a number of neurons of 150 over the last 30 days of MSFT dataset

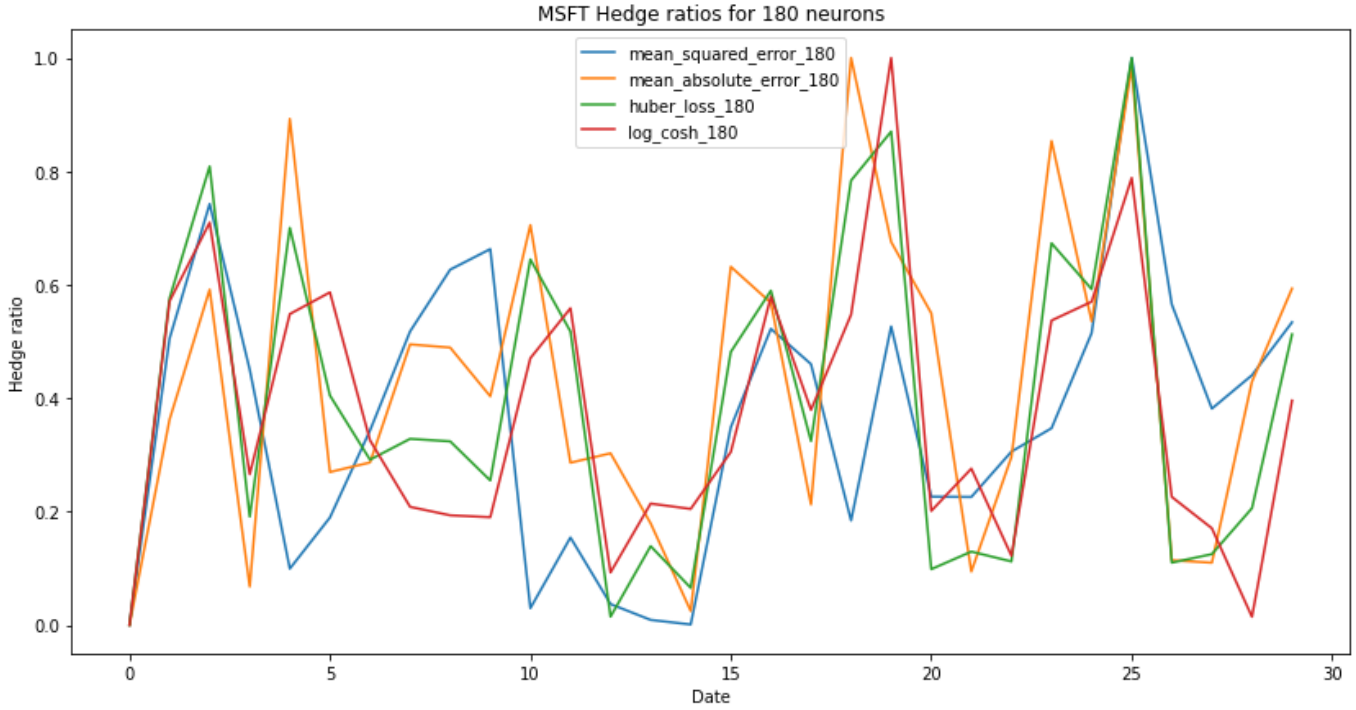


Figure 5.10: NN deltas for different loss functions and a number of neurons of 180 over the last 30 days of MSFT dataset

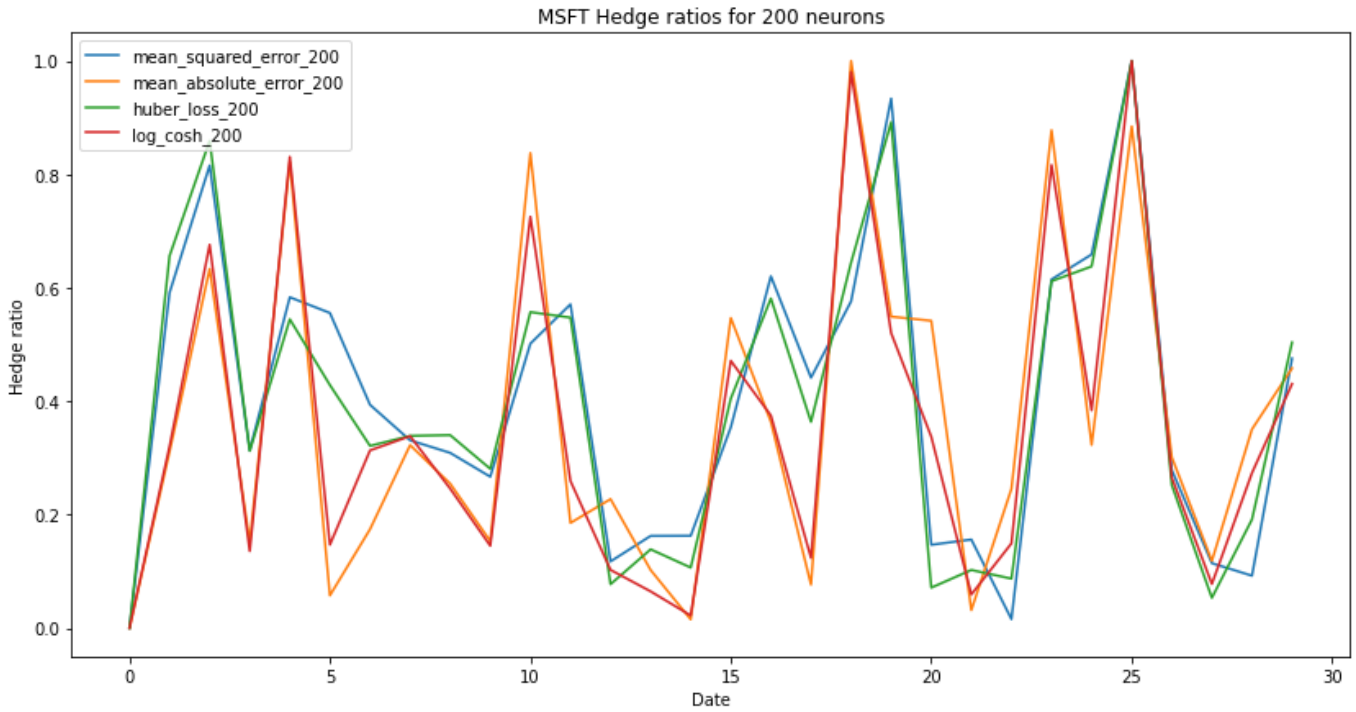


Figure 5.11: NN deltas for different loss functions and a number of neurons of 200 over the last 30 days of MSFT dataset

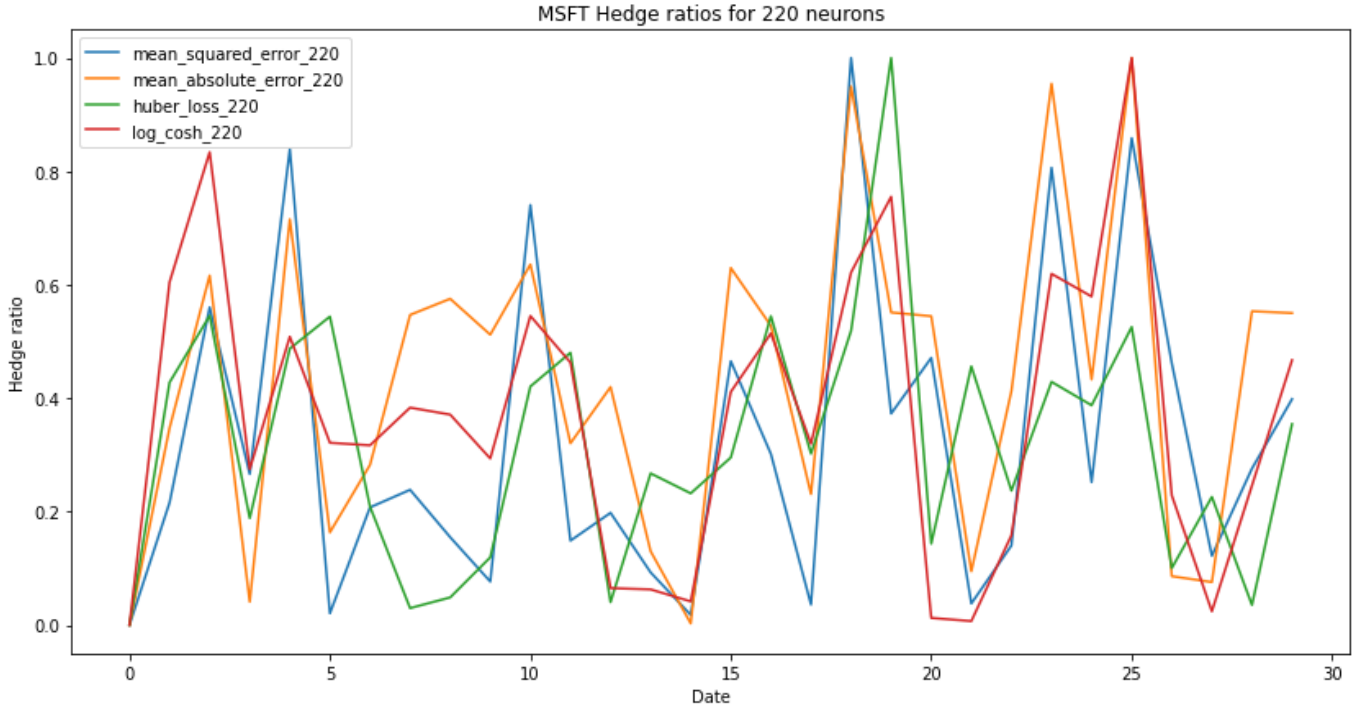


Figure 5.12: NN deltas for different loss functions and a number of neurons of 220 over the last 30 days of MSFT dataset

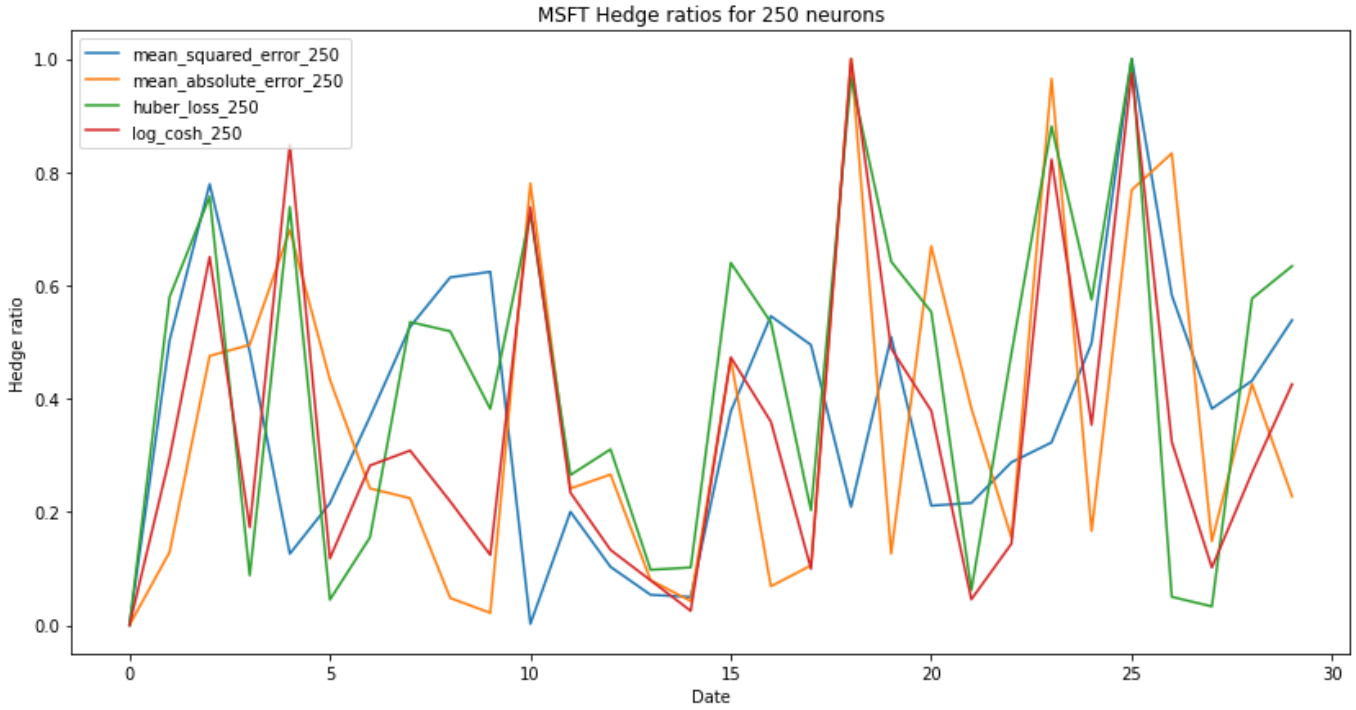


Figure 5.13: NN deltas for different loss functions and a number of neurons of 250 over the last 30 days of MSFT dataset

Within each neuron configuration, the hedge ratios for the various loss functions (mean squared error, mean absolute error, Huber loss, and log cosh) for both MBG and MSFT shows comparable patterns. This consistency implies that the design of the model (the total number of neurons) is somehow more important than the selection of the loss function. The hedge ratios are not significantly affected by the loss function selection, which might suggest that other factors, such as the computing efficiency or the training stability, can be taken into account when making this decision.

A tendency towards smoother hedge ratios may be seen in both stocks when the number of neurons increases. Nevertheless, in order to reach a degree of stability similar to MBG, which has fewer neurons (30 or 40), MSFT, which has more price volatility, needs more neurons (220 or 250). More neurons lead to less sensitivity to sudden shifts and improved trend capture. This is true for both MBG and MSFT, yet the precise amount of neurons needed for best results varies because of the stock volatility.

The ideal neuron configuration for MBG seems to be between 30 and 40 neurons, balancing the computational efficiency with the model performance. This is consistent with a less unstable historical price history. MSFT benefits from a higher neuron configuration (220-250 neurons) to produce smooth and steady hedge ratios because of its higher volatility and growth trajectory.

While these first results are illustrating a model that is generally robust, the NN model's performance differences across the two stocks depicts the fact that its consistency could vary.

5.2 Robustness analysis

Using [40] (more on this on appendix B1.), we start with reasonable initial guesses for calibrating the parameters of the models we use to test the robustness of our algorithm. In order to calibrate our models, we estimate the parameters for the Heston, Bates, VG-CIR, and rBergomi models using historical log returns. We use a simple optimization technique (L-BFGS-B) for parameter estimation (more on this in appendix B.). Next, we generate synthetic stock price paths using the estimated parameters for each model.

Finally, we split the generated synthetic data into training and validation sets to optimize the model on the training set and evaluate its performance on the validation set, ensuring that the model generalizes well. More details about these steps can be found in appendix B3.

Below we show the results of our algorithm refined with synthetic data :

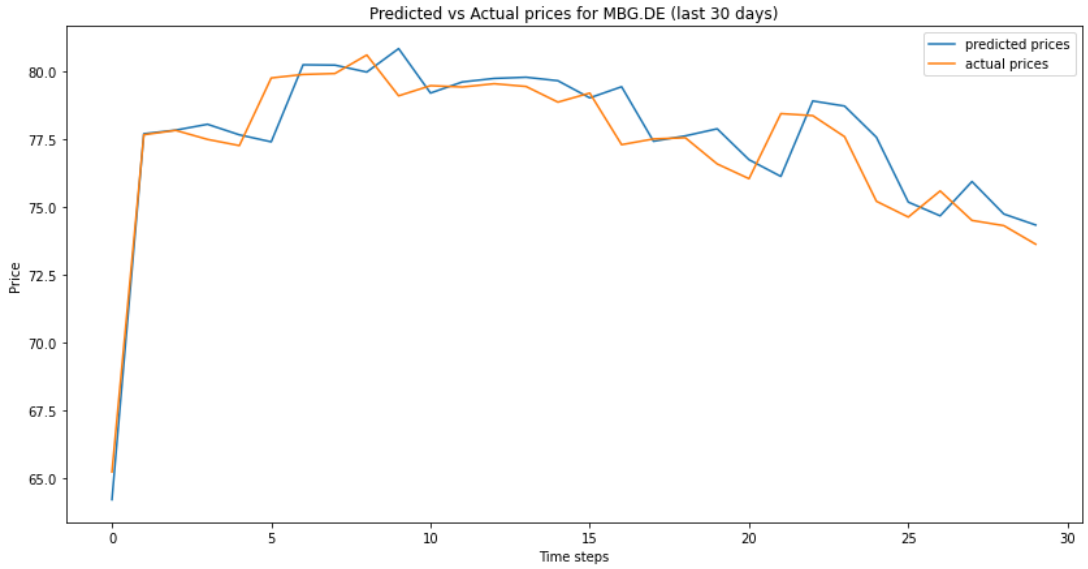


Figure 5.14: NN predicted prices for MBG synthetic data over the last 30 days

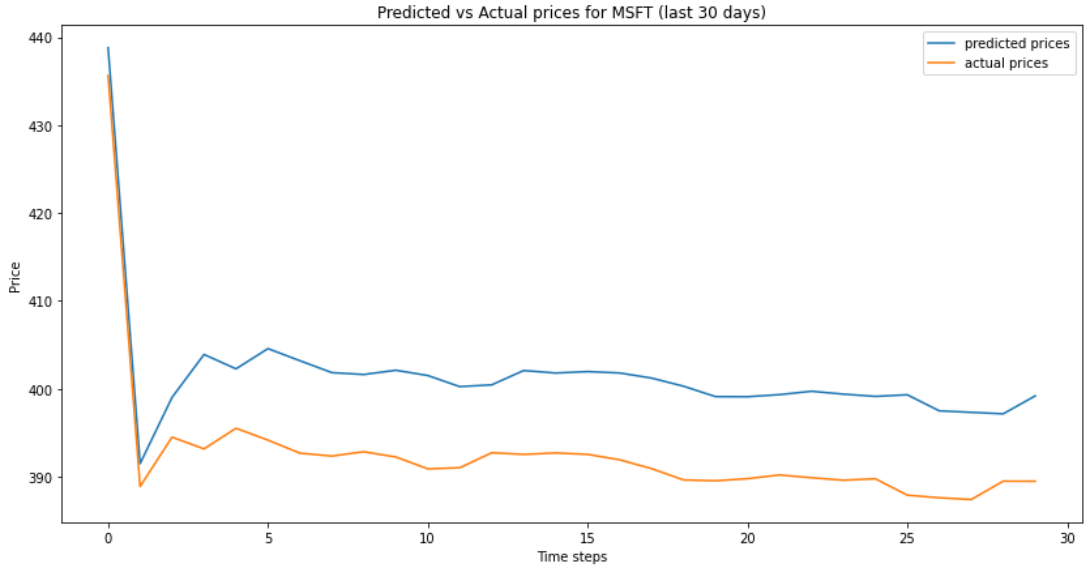


Figure 5.15: NN predicted prices for MSFT synthetic data over the last 30 days

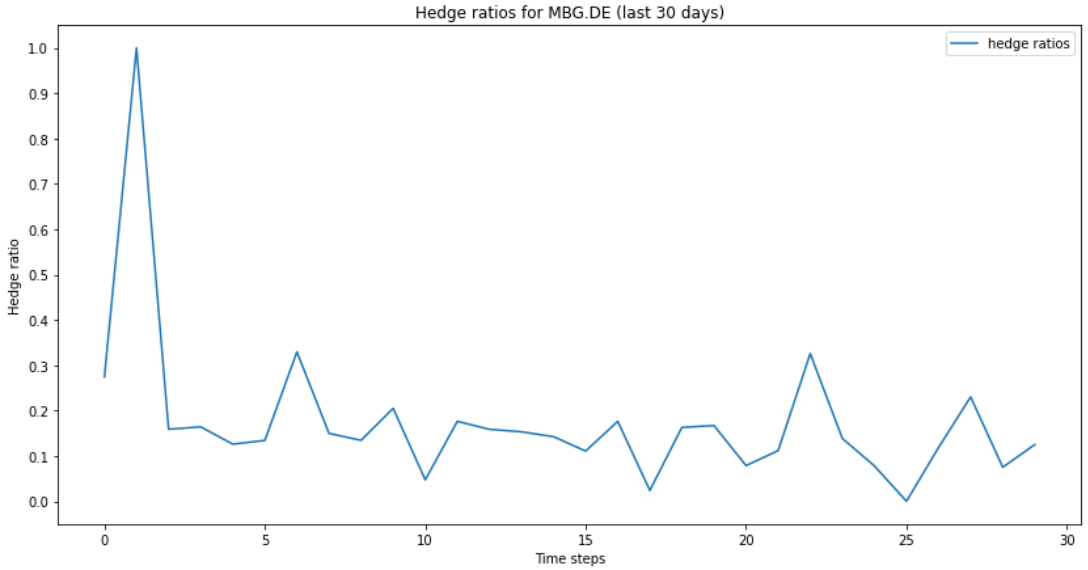


Figure 5.16: NN deltas for MBG synthetic data over the last 30 days

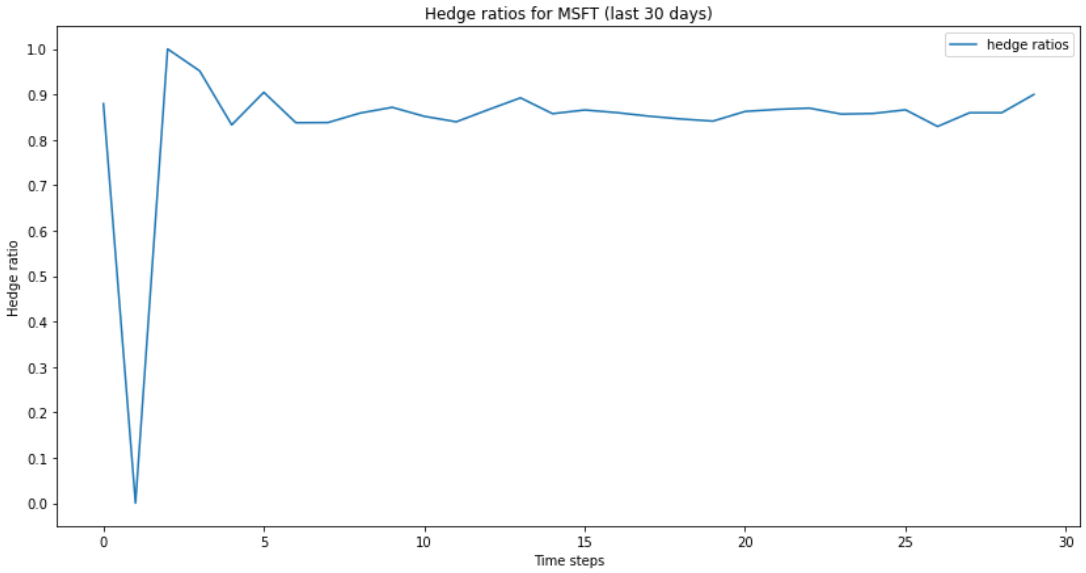


Figure 5.17: NN deltas for MSFT synthetic data over the last 30 days

The predicted prices for both MBG.DE and MSFT align closely with the actual synthetic prices over the last 30 days, which we may analyse as indicating the NN's ability to generalize from the training data and predict future prices accurately.

The hedge ratios for MBG.DE show more variability compared to those for MSFT. This variability reflects the differences in market dynamics and volatility captured by the synthetic paths. The NN's dynamic adjustment of hedge ratios suggests a certain responsiveness to changing market conditions. However, the higher variability in MBG's hedge ratios might indicate overfitting or an inherent market characteristic.

There are a few possible explanations that we can think of for the significant shifts in hedging ratios that occur when moving from raw to synthetic data. Although our synthetic data is designed to resemble real market data, it may possess distributional features that are not identical to those of the real market data, thus impacting our model's learnt patterns. Additionally, our model may overfit to the particular features of the synthetic data, resulting in some hedging ratios that are less applicable to actual market situations and more customised to the synthetic scenarios. Our model's sensitivity to input data changes could also lead to variations in hedge ratios, and as it adjusts to the synthetic data's characteristics, the resulting hedge ratios may differ significantly from those derived from our raw data.

5.3 Results with additional inputs

Since we have historical data for interest rates and market index performance, the only indicator we have to compute is RSI. In order to do so, we :

1. Calculate the price changes each closing day for both datasets (Δ) :

$$\Delta_i = \text{Close}_i - \text{Close}_{i-1}$$

2. Separate gains and losses :

$$\text{Gain}_i = \begin{cases} \Delta_i & \text{if } \Delta_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{Loss}_i = \begin{cases} -\Delta_i & \text{if } \Delta_i < 0 \\ 0 & \text{otherwise} \end{cases}$$

3. Calculate the Average Gain (AG) and Loss (AL) over the period of the two datasets :

$$\text{AG}_i = \frac{1}{n} \sum_{k=0}^{n-1} \text{Gain}_{i-k}$$

$$\text{AL}_i = \frac{1}{n} \sum_{k=0}^{n-1} \text{Loss}_{i-k}$$

4. Calculate the Relative Strength (RS):

$$RS_i = \frac{\text{AG}_i}{\text{AL}_i}$$

5. Compute the index of the RS (our desired RSI):

$$RSI_i = 100 - \left(\frac{100}{1 + RS_i} \right)$$

Using our raw data with historical stock prices as a performance baseline (and the same initial neural networks parameters), we choose to test various configurations with our additional neural networks inputs :

- adding only Relative Strength Index (RSI)
- adding only interest rates
- adding only market index performance
- adding RSI and interest rates
- adding RSI and market index performance
- adding market index performance and interest rates

- adding all additional inputs

We show below the results for these configurations :

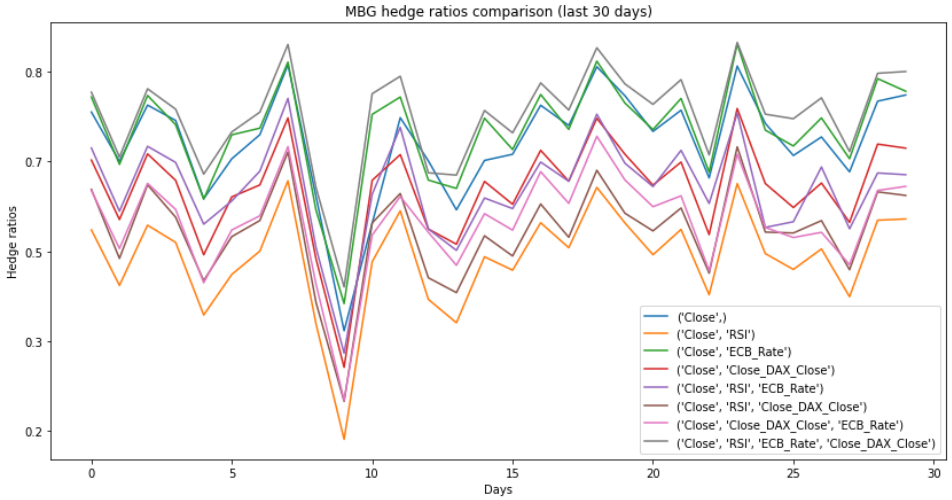


Figure 5.18: Comparison of NN hedge ratios for MBG with different configurations of inputs over the datasets last 30 days : "Close" is the initial historical price of the stock

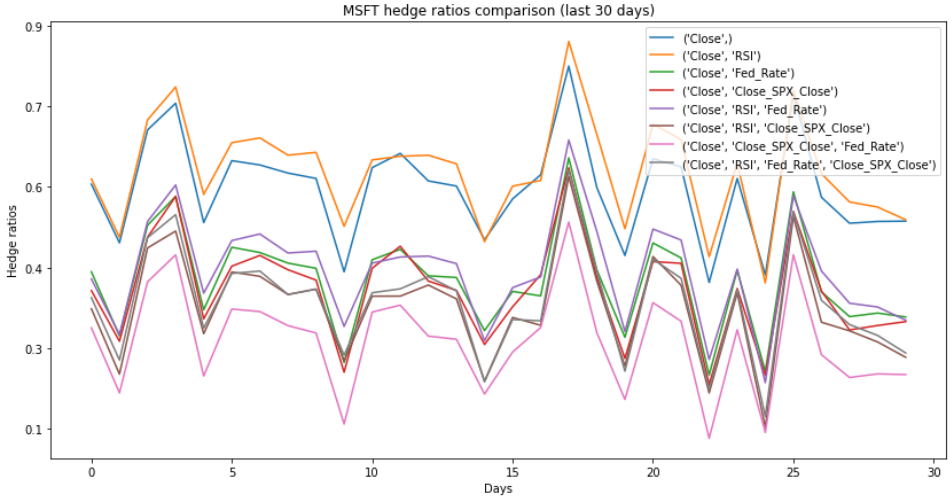


Figure 5.19: Comparison of NN hedge ratios for MSFT with different configurations of inputs over the datasets last 30 days : "Close" is the initial historical price of the stock

The inclusion of RSI, interest rates, and market index performance most of the time led us to more stable and consistent hedge ratios. This is suggesting that these indicators add a valuable context beyond the mere price movements captured by our initial historical prices, thus enabling our model to make better hedging decisions in the given environment we are building. We observe a reduced variability in hedge ratios with additional inputs, which might indicate a certain robustness from the model, showing us that it is capable of maintaining the hedging strategy despite market fluctuations.

The analysis appears to be fairly similar for both datasets. Hedge ratios are varying significantly depending on the input features used in the NN model. Using only the closing prices as input results in higher fluctuating hedge ratios with more noticeable peaks and troughs. This baseline configuration appears to capture pretty accurately basic price movements, but might lack the depth provided by additional indicators. Indeed, when the RSI is included, the hedge ratios stabilize slightly. The RSI, that we could describe as a momentum oscillator, helps the model better understand overbought or oversold conditions, leading to potentially better hedge adjustments. The hedge ratios with RSI show a reduced variability, indicating what might be a more consistent hedging strategy. Adding interest rates results in more stable hedge ratios, and our model react positively to them. In fact, our model is using them to for understand better the broader economic environment, because they are influencing investor sentiment and market volatility. Moreover, the hedge ratios with interest rates input show fewer extreme fluctuations, providing a hedging strategy that seems more reliable. Including the index performance as an input further smoothens the hedge ratios. In our case, we can indeed consider this latter as a good a benchmark for the overall market, because it looks like it helps the model align its hedging strategy with more general market trends.

The most stable and consistent hedge ratios are observed when combining multiple inputs (historical underliying, RSI, index performance, and interest rate). This leverages the strengths of each indicator, resulting in a robust hedging strategy. Indeed, combining these inputs mitigate the weaknesses of individual indicators and provide a well-rounded view of market dynamics. The enhanced predictability in hedge adjustments reflects the model's improved ability to anticipate market trends and adjust positions accordingly. One of the reasons of the robustness of our NN model is that it is significantly enhanced by incorporating a diverse set of inputs, which mitigates the risk of overfitting to specific market conditions and improves the model's generalizability across different market scenarios.

However, the higher variability in hedge ratios (particularly when compared with the BS benchmark) implies more frequent trading, leading to higher transaction costs, which can erode the profitability advantages. Frequent rebalancing might also be impacted by market liquidity, as adjusting positions frequently can be substantial in less liquid markets.

In our situation, our results show that we have made the right choices in terms of additional inputs, since these appear to substantially improve the performance of our model. One of the questions that arises concerns the impact of a poor choice of additional inputs, i.e. an input that does not correlate well enough with our model and the predictions we want to make. To what extent would this input erode the performance of our model? Could we ignore it and still make relatively accurate predictions?

5.4 Market frictions

In the raw data run of our algorithm, we experimented the NN model's higher variability in hedge ratios than the BS benchmark, which will imply more frequent trading, inevitably leading to higher transaction costs. This is a critical element to take into account in our analysis as it can erode the profitability advantages observed in the PnL analysis. Obviously, frequent rebalancing might also be impacted by market liquidity. In less liquid markets, the cost of adjusting positions frequently can be substantial.

The transaction cost rate for large-cap equities usually ranges from 5 to 20 basis points, according to studies such as [41] and industry data. Thus, we will use for our algorithm the average value within this range, i.e. 10 basis points.

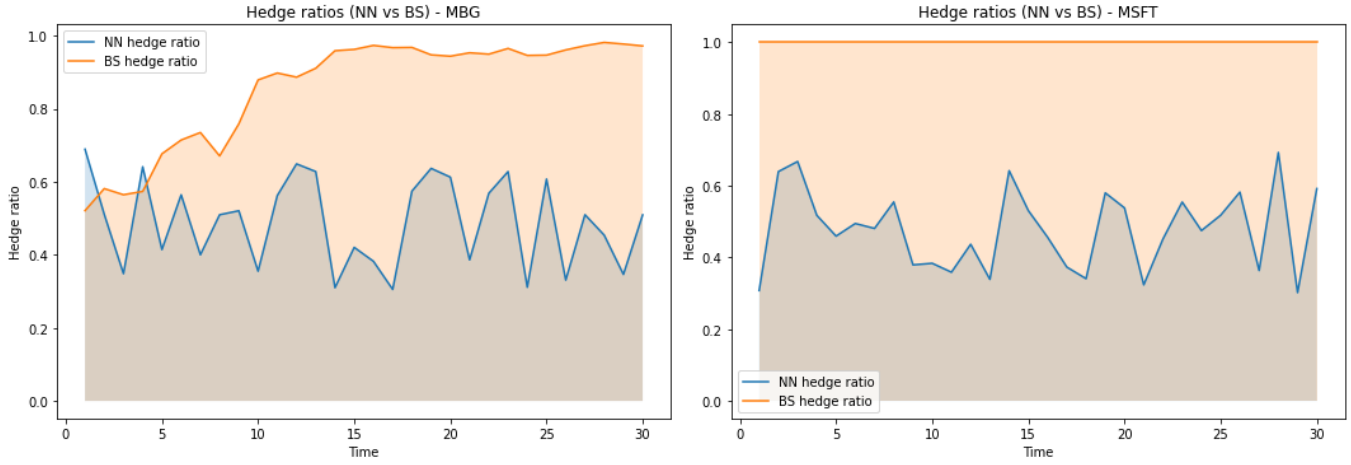


Figure 5.20: NN vs BS hedge ratios over the datasets last 30 days (initial configuration of the neural networks with transaction costs)

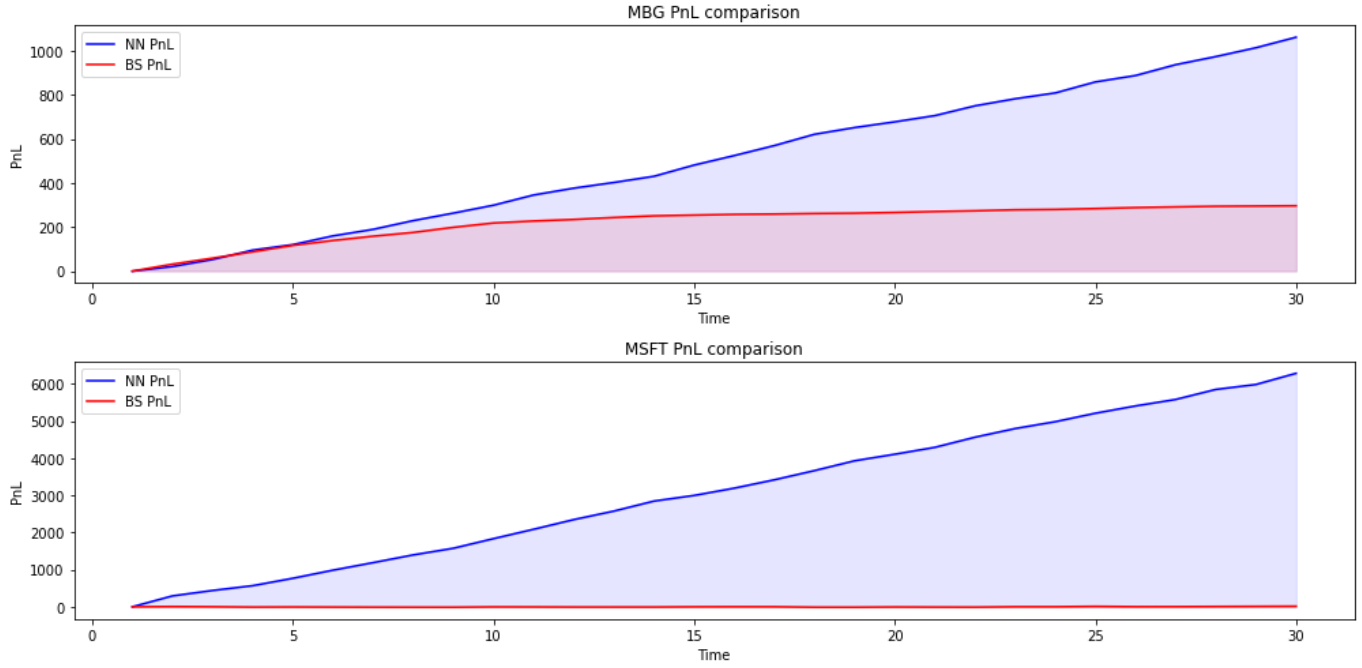


Figure 5.21: Comparison of the PnL of the NN and BS hedge over the last 30-days period (initial configuration of the neural networks with transaction costs)

Figure 5.22: Comparison of the NN hedge ratios over the last 30-days period with and without transaction costs (initial configuration of the neural networks)

Metric	Without TC	With TC
MBG hedge ratio stats		
Mean	0.194205	0.511600
Median	0.112136	0.479571
Std Dev	0.198195	0.115025
Min	0.000000	0.316766
Max	1.000000	0.699005
MSFT hedge ratio stats		
Mean	0.370439	0.453790
Median	0.358556	0.414903
Std Dev	0.255306	0.118775
Min	0.000000	0.309478
Max	1.000000	0.693172

Table 5.2: Comparison of hedge ratio statistics for MBG and MSFT with and without transaction costs

The hedge ratios with transaction costs (TC) show a noticeable increase in mean and median values compared to those without TC. This is something expected, as transaction costs add a penalty to frequent trading, which is prompting our model to adjust hedges in a less aggressive way. The increased standard deviation in hedge ratios with TC highlights the existence of a trade-off between hedge effectiveness and cost management. The NN model, despite higher variability in hedge ratios, results in a higher cumulative PnL. However, the introduction of transaction costs erodes this advantage significantly. The BS model, with its more stable but less dynamic hedging strategy, incurs lower transaction costs, leading to a narrower PnL gap when costs are considered.

For MBG, the mean hedge ratio increases from 0.19 to 0.51 with TC, and for MSFT, it increases from 0.37 to 0.45. This suggests that transaction costs can substantially alter hedging strategies, making them more conservative to minimize trading costs. We may observe an overreaction from our NN model, and while it may be more adaptive, it could lead in certain situations to less stable hedging strategies. The NN's dynamic adjustment, which we intended to use to optimize trading under TC, might cause fluctuations that could erode profitability, especially in less liquid markets where the cost of adjusting positions frequently can be substantial. The higher mean and median hedge ratios indicate that the NN model tends to hold larger positions to mitigate frequent trading costs. However, The strategy chosen by our algorithm in which it tends to hold larger positions to mitigate frequent trading costs may not always align with optimal hedging practices, potentially exposing our portfolio to higher risks.

The higher variability in the NN model's hedge ratios with TC may also indicate a potential overfitting issue. Indeed, while our model is trained to optimize performance under transaction costs, this tailored approach might limit its generalizability to different market conditions or asset classes. The fact that our model is relying on historical data and synthetic scenarios, while useful for training, may not fully capture the nuances of real market environments, leading to suboptimal performance when applied to new data.

Our model's adaptability to transaction costs may also come at the expense of robustness. Our results may show that the fine-tuning required to balance hedge effectiveness and cost management is resulting in a model that is overly sensitive to specific market conditions, weakening its use across a broader range of financial instruments and market scenarios.

We also need to mention that the simplification that we've made regarding the average value of 10 basis points for transaction costs might not capture the full complexity of real-world trading costs, which can vary significantly based on market conditions and trading volumes.

6 Legal, Social, Ethical and Professional Issues

Throughout the development and implementation of this project, adherence to the Code of Conduct & Code of Good Practice as outlined by the British Computer Society (BCS) was strictly maintained. These standards we considered crucial to ensure that professional, legal, and ethical norms were met, and to protect the integrity and credibility of this work, as well as protecting intellectual property rights and ensuring the ethical use of data and resources.

The project involved the use of various open-source libraries and datasets. Proper citations and acknowledgments were given to all third-party sources, to ensure compliance with licensing agreements and avoiding plagiarism. The sources for the datasets used were mentioned in appendix G. This practice aimed to respect the intellectual property rights of others but also to enhance the transparency and reproducibility of the research.

One of the key ethical concerns in this research was ensuring the confidentiality and security of the data used. The datasets employed were historical stock prices, interest rates and performance indexes, which are publicly available and do not contain personally identifiable information. However, measures were taken to ensure that all data handling processes adhered to ethical standards, avoiding any potential misuse or misrepresentation. Moreover, the ethical use of artificial intelligence and machine learning techniques was considered in the context of using neural networks.

The social implications of this research are significant, as financial models and trading algorithms can have a real impact on markets and society. By developing robust and transparent models, the project aims to contribute in a positive way to the field of quantitative finance, promoting potentially more stable and efficient financial markets. However, the potential misuse of advanced financial models also poses a risk, that was carefully identified and took into account when thinking about the impact of this dissertation. Indeed, if misapplied, these models could lead to an increase in market volatility or be exploited for manipulative trading practices. Thus, our research emphasizes the importance of ethical usage and the need for regulatory control to prevent such outcomes.

Ensuring professional competence was also a major part of the considerations of this research : extensive literature reviews were conducted to ensure the deepest possible understanding of the subject matter. The project also involved a rigorous testing and validation of the models to ensure their accuracy and reliability. Transparency and academic integrity were maintained throughout the project, and all methodologies and analytical techniques were documented in detail, allowing for examination and replication by students or

researchers.

7 Conclusion

In this dissertation, we aimed to create an adaptable and strong neural network-based model for financial product hedging, particularly with regard to equity derivatives and stocks. Our main goal was to evaluate our neural network model's performance against the conventional Black-Scholes model in a range of market scenarios, accounting for market frictions. Our findings emphasized a number of important aspects of neural networks' potential use in financial risk management.

To train and test the model, we made use of long-term historical data that covered a long time span, which allowed for a grasp of price changes and market dynamics over a sizable amount of time. The neural network model demonstrated a good ability to predict future prices for two different profiles of stocks, aligning with actual prices over the tested period. This accuracy suggests that our neural network effectively learned and generalized from historical data, allowing it to predict future trends with a reasonable degree of accuracy. The performance of our model in predicting prices indicates its potential for application in real-world trading scenarios where accurate price forecasts are essential.

The hedge ratios derived from the neural network model exhibited more variability compared to those from the BS model. This made us think of this responsiveness as a double-edged sword; while it shows the model's adaptability, it also suggests potential overfitting or the presence of some inherent market characteristics that may not be entirely captured by the data that we used for training.

We also placed a particular focus on showing a certain sensitivity analysis in the architecture of our neural network. Particularly, the number of neurons had a significant impact on the hedge ratios. Different configurations of neurons and loss functions were tested, and it was found that while the choice of loss function influenced the results, the number of neurons played a more critical role. Thus, we highlighted the importance of carefully tuning the neural network architecture to achieve optimal performance. Additionally, incorporating various technical, macroeconomic, and market indicators into the neural network model seemed to have enhanced its hedging performance. Inputs such as Relative Strength Index (RSI), interest rates, and market index performance provided additional context, enabling the model to adapt better to changing market conditions. This demonstrates our model's capacity to process and integrate diverse data sources, enhancing its predictive accuracy and robustness. But it also showed the extent to which results could vary if the neural network was not correctly calibrated, highlighting the tailor-made nature of this approach.

One of the distinctive features of our approach is that it has enabled us to craft the incorporation of transaction costs into our hedging model with our neural network, which provided a more realistic assessment of its practical applicability. The adjusted loss function, which included transaction costs, demonstrated the model's ability to manage these costs effectively while maintaining robust hedging performance. Our work has also enabled us to gain practical experience of the significant impact of market frictions on the profitability and optimality of trading strategies, portfolio management and risk management.

Despite the promising results, our work suffers from several limitations that requires further investigation.

The variability in hedge ratios, particularly for less volatile stock, suggests that the neural network's adaptability might come at the cost of stability. This variability could result from the model's sensitivity to the specific features of the training data, implying that the neural network might overreact to minor changes in market conditions. A more robust validation approach, possibly incorporating cross-validation techniques or out-of-sample testing, could mitigate this issue. Additionally, while our model incorporates several market indicators, it remains relatively simplistic in its feature selection. Financial markets are influenced by a huge number of factors, including macroeconomic events, geopolitical risks, and investor sentiment. Future work should explore incorporating a broader range of features, potentially leveraging alternative data sources such as social media sentiment or news analytics, to capture an even more realistic and hopefully powerful picture of market dynamics. Another critical aspect we need to mention is the "black-box" nature of neural network models, that introduces challenges for understanding the underlying decision-making process. Indeed, a huge work might need to be done to interpret and explain neural network predictions, in order to enhance their acceptability and trustworthiness among financial practitioners. Additionally, the model's framework does not account for liquidity-related costs or the impact of large trades on market prices, which can significantly affect hedging strategies. Indeed, in less liquid markets, the cost of adjusting positions frequently can be substantial, and the NN model's tendency to react dynamically to transaction costs might struggle with these issues. Incorporating liquidity constraints into the model would highlight potential risks associated with trading in different market environments.

Moreover, in this dissertation, we looked at vanilla financial products (equities and equity derivatives), and questions are being asked as to whether similar results can be found for more complex and composite financial products (multiple underlyings, barrier options, swaps, etc.). The nature and functioning of neural networks suggest that they can potentially take into account and scale a large number of variables and indicators, but this remains hypothetical and needs to be explored in detail. Moreover, our algorithm involved long-term data (time series of several decades), and did not allow us to develop the issue of high-frequency finance environments, in which the granularity of price movements and sales volumes, as well as trading strategies, are sometimes totally different. However, high frequency finance is also one of the favourite subjects for study in current research into machine learning and quantitative finance, and exploring this area can certainly provide new avenues and conclusions for our work. Moreover, advancements in neural network architectures, such as Transformers and Generative Adversarial Networks (GANs), offer promising directions for improving the model's predictive power and efficiency. These advanced models can capture more intricate patterns and dependencies in the data, potentially leading to better hedging strategies.

Bibliography

[1] Hayne E. Leland. "Option Pricing and Replication with Transactions Costs". In: *The Journal of Finance* 40.5 (1985), pp. 1283–1301.

[2] Stewart D. Hodges and Anthony Neuberger. "Optimal Replication of Contingent Claims Under Transaction Costs". In: *The Review of Futures Markets* 8.2 (1992), pp. 222–239. URL: `https://EconPapers.repec.org/RePEc:wsi:rfmxx:vod:08:y:1992:i:02:p:222-239`.

[3] Hans Buehler et al. "Deep Hedging". In: *Quantitative Finance* 19.1 (2019), pp. 10–24.

[4] Robert C. Merton. *Continuous-Time Finance*. Blackwell, 1990.

[5] Steven L. Heston. "A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options". In: *Review of Financial Studies* 6.2 (1993), pp. 327–343.

[6] Joel Hasbrouck. "Transaction Costs and Portfolio Performance: Past versus Future". In: *Journal of Financial and Quantitative Analysis* (2009).

[7] Darrell Duffie and Kenneth J. Singleton. "Multifactor Models of the Term Structure". In: *Journal of Financial Economics* 23.1 (1997), pp. 5–42.

[8] John C. Cox, Jonathan E. Ingersoll, and Stephen A. Ross. "A Theory of the Term Structure of Interest Rates". In: *Econometrica* 53.2 (1985), pp. 385–407.

[9] Allan Jonathan da Silva and Jack Baczynski. "Discretely Distributed Scheduled Jumps and Interest Rate Derivatives: Pricing in the Context of Central Bank Actions". In: *Journal of Derivatives* (2005).

[10] Lubos Pastor and Robert F. Stambaugh. "Liquidity Risk and Expected Stock Returns". In: *Journal of Political Economy* 111.3 (2003), pp. 642–685.

[11] Tim Bollerslev. "Generalized Autoregressive Conditional Heteroskedasticity". In: *Journal of Econometrics* 31 (1986), pp. 307–327.

[12] Robert C. Merton. "Option Pricing When Underlying Stock Returns Are Discontinuous". In: *Journal of Financial Economics* 3.1-2 (1976), pp. 125–144.

[13] Bruno Dupire. "Pricing with a Smile". In: *Risk* (1994).

[14] David Heath, Robert Jarrow, and Andrew Morton. "Bond Pricing and the Term Structure of Interest Rates: A New Methodology". In: *Econometrica* 60.1 (1992), pp. 77–105.

[15] James M. Hutchinson, Andrew W. Lo, and Tomaso Poggio. "A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks". In: *Journal of Finance* 49.3 (1994), pp. 851–889.

[16] James B. Heaton, Nicholas G. Polson, and Jan Hendrik Witte. "Deep Learning for Finance: Deep Portfolios". In: *Applied Stochastic Models in Business and Industry* 33.1 (2017), pp. 3–12.

[17] Justin Sirignano and Rama Cont. "Universal Features of Price Formation in Financial Markets: Perspectives from Deep Learning". In: *Quantitative Finance* 19.9 (2019), pp. 1391–1399.

[18] Shaojie Zhang, Stefan Zohren, and Stephen Roberts. "Deep Learning for Portfolio Optimization". In: *Journal of Financial Data Science* 1.1 (2019), pp. 1–18.

[19] Marco Avellaneda, Sasha Stoikov, and Jie Zhang. "High-Frequency Trading in a Limit Order Book". In: *Quantitative Finance* 11.1 (2011), pp. 1–22.

[20] John C. Hull. *Options, Futures, and Other Derivatives*. Pearson, 2018.

[21] Riccardo Rebonato. *Volatility and Correlation: The Perfect Hedger and the Fox*. Wiley, 2004.

[22] Kady Sako, Berthine Nyunga Mpinda, and Paulo Canas Rodrigues. "Neural Networks for Financial Time Series Forecasting". In: (2020).

[23] Jie Liu, Xiaowei Li, and Wenqing Zhong. "Unsupervised outlier detection in multidimensional data". In: *Journal of Big Data* 5.1 (2018), pp. 1–21. URL: `https://journalofbigdata.springeropen.com/articles/10.1186/s40537-018-0142-5`.

[24] Klaus Greff et al. "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2017), pp. 2222–2232.

[25] Qingsong Wen, Zhicheng Zhang, and Weiwei Peng. "A Multi-Horizon Quantile Recurrent Forecaster". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[26] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 315–323.

[27] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks are Universal Approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366.

[28] Robert A. Haugen and Philippe Jorion. "Seasonality in Stock Returns". In: *Journal of Financial Economics* 21.1 (1989), pp. 117–144.

[29] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[30] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimisation". In: *arXiv preprint arXiv:1412.6980* (2014).

[31] Wen Liang et al. "Exploring the Applications of Deep Learning Models in Financial Time Series Forecasting". In: *Journal of Financial Forecasting* (2020).

[32] Thomas Fischer and Christopher Krauss. "Deep Learning with Long Short-Term Memory Networks for Financial Market Predictions". In: *European Journal of Operational Research* 270.2 (2018), pp. 654–669.

[33] Weijie Bao, Jun Yue, and Yulei Rao. "A Deep Learning Framework for Financial Time Series Using Stacked Autoencoders and Long Short-Term Memory". In: *Neurocomputing* 287 (2017), pp. 178–190.

[34] Daniel Masters and Lorenzo Luschi. "Revisiting Small Batch Training for Deep Neural Networks". In: *arXiv preprint arXiv:1804.07612* (2018).

[35] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *arXiv preprint arXiv:1609.04836* (2017).

[36] Leslie N. Smith and Nicholay Topin. "Don't Decay the Learning Rate, Increase It". In: *arXiv preprint arXiv:1708.07120* (2017).

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[38] Jean-Remy Gamboa. "Deep Learning for Time-Series Analysis". In: *arXiv preprint arXiv:1701.01887* (2017).

[39] Robert Almgren and Neil Chriss. "Optimal Execution of Portfolio Transactions". In: *Journal of Risk* 3.2 (2000), pp. 5–39.

[40] Jarosław Gruszka and Janusz Szwabiński. "Parameter Estimation of the Heston Volatility Model with Jumps in the Asset Prices". In: *Central European Journal of Economic Modelling and Econometrics* 5.4 (2013), pp. 239–265.

[41] Steven A. Berkowitz, Dennis E. Logue, and Eugene A. Noser. "The Total Costs of Transactions on the NYSE". In: *Journal of Finance* 43.1 (1988), pp. 97–112.

[42] Torben G. Andersen and Luca Benzoni. "Stochastic Volatility". In: *Handbook of Financial Econometrics: Tools and Techniques*. Ed. by Yacine Aït-Sahalia and Lars Peter Hansen. Vol. 1. Elsevier, 2010, pp. 67–137.

[43] Gurdip Bakshi, Charles Cao, and Zhiwu Chen. "Empirical Performance of Alternative Option Pricing Models". In: *Journal of Finance* 52.5 (1997), pp. 2003–2049.

[44] Mark Broadie, Mikhail Chernov, and Michael Johannes. "Model Specification and Risk Premia: Evidence from Futures Options". In: *Journal of Finance* 62.3 (2007), pp. 1453–1490.

[45] Bjorn Eraker, Michael Johannes, and Nicholas Polson. "The Impact of Jumps in Volatility and Returns". In: *Journal of Finance* 58.3 (2003), pp. 1269–1300.

[46] David S. Bates. "Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options". In: *Review of Financial Studies* 9.1 (1996), pp. 69–107.

[47] David S. Bates. "The Crash of '87: Was It Expected? The Evidence from Options Markets". In: *Journal of Finance* 46.3 (1991), pp. 1009–1044.

[48] Bjorn Eraker. "Do Stock Prices and Volatility Jump? Reconciling Evidence from Spot and Option Prices". In: *Journal of Finance* 59.3 (2004), pp. 1367–1403.

[49] Dilip B. Madan, Peter Carr, and Eric C. Chang. "The Variance Gamma Process and Option Pricing". In: *European Finance Review* 2.1 (1998), pp. 79–105.

[50] Dilip B. Madan and Eugene Seneta. "The Variance Gamma (V.G.) Model for Share Market Returns". In: *Journal of Business* 63.4 (1990), pp. 511–524.

[51] Rama Cont and Peter Tankov. *Financial Modelling with Jump Processes*. Chapman and Hall/CRC, 2004.

[52] Christian Bayer, Peter Friz, and Jim Gatheral. "Pricing Under Rough Volatility". In: *Quantitative Finance* 16.6 (2016), pp. 887–904.

[53] Jim Gatheral, Thibault Jaisson, and Mathieu Rosenbaum. "Volatility is Rough". In: *Quantitative Finance* 18.6 (2018), pp. 933–949.

Appendix

A. More precision on neural networks

In order to give further explanations and insights on neural networks, that are the main theoretical basis of this dissertation, we give more precisions on their features.

These precisions, particularly the mathematical formalizations, were extracted from the following sources :

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
- Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735-1780.
- LeCun, Y., Bengio, Y., Hinton, G. (2015). Deep Learning. Nature, 521(7553), 436-444.

A1. Loss function

We define loss functions, which are an essential part of neural nets. They are employed to quantify the differences between the predicted output by the neural network and the actual target value. The neural network's training objective is to minimise this loss function in order to increase the model's prediction accuracy.

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ be the training dataset, where $x_i \in \mathbb{R}^d$ denotes the input features and $y_i \in \mathbb{R}$ (or $\mathbb{R}^C$ for classification tasks) represents the target value for the i-th data point. The neural network model can be represented as a function f_θ parameterized by θ, where $\hat{y}_i = f_\theta(x_i)$ is the predicted output.

The loss function L quantifies the error between the predicted outputs $\hat{y}_i$ and the actual target values y_i. Mathematically, it can be defined as:

$$L(y_i, \hat{y}_i) = L(y_i, f_\theta(x_i))$$

During the training process, we aim to find the optimal parameters θ that minimize the average loss over the entire training dataset. The empirical risk $R(\theta)$ is given by:

$$R(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f_\theta(x_i))$$

The optimization problem can then be formulated as:

$$\theta^* = \arg\min_\theta R(\theta) = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(y_i, f_\theta(x_i))$$

To solve the optimization problem, we iteratively update the parameters θ in the direction that reduces the loss. We define the gradient descent's update rule as :

$$\theta \leftarrow \theta - \eta \nabla_\theta R(\theta)$$

where η is the learning rate, and $\nabla_\theta R(\theta)$ is the gradient of the empirical risk with respect to the parameters θ.

The gradient $\nabla_\theta R(\theta)$ can be computed as:

$$\nabla_\theta R(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L(y_i, f_\theta(x_i))$$

A2. Batch size and epochs

Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ be the training dataset with N samples. The dataset is divided into B batches, where each batch $\mathcal{B}_k$ contains m samples: $\mathcal{B}_k = \{(x_{i_k}, y_{i_k})\}_{i_k=1}^{m}$.

An epoch is a full pass through the entire dataset $\mathcal{D}$. For each epoch e (where $e \in \{1, 2, \ldots, E\}$ and E is the total number of epochs), the following steps are performed:

For each batch $\mathcal{B}_k$ in the dataset, the model parameters θ are updated using the gradient descent method or its variants:

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\mathcal{B}_k)$$

Here, η is the learning rate, and $L(\mathcal{B}_k)$ is the loss computed on the batch $\mathcal{B}_k$.

Once all B batches have been processed, one epoch is completed. The process is repeated for E epochs.

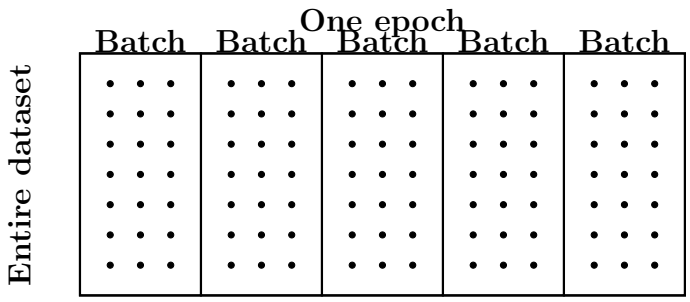


Figure 1: Illustration of epochs and batch size in neural network training.

A3. Number of LSTM layers

Let $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T]$ be the input sequence, where $\mathbf{x}_t \in \mathbb{R}^n$ represents the input vector at time step t. The hidden state $\mathbf{h}_t^{(1)}$ and cell state $\mathbf{c}_t^{(1)}$ at time step t in the first LSTM layer are computed as:

$$
\begin{aligned}
\mathbf{f}_t^{(1)} &= \sigma(\mathbf{W}_f^{(1)}\mathbf{x}_t + \mathbf{U}_f^{(1)}\mathbf{h}_{t-1}^{(1)} + \mathbf{b}_f^{(1)}) \\
\mathbf{i}_t^{(1)} &= \sigma(\mathbf{W}_i^{(1)}\mathbf{x}_t + \mathbf{U}_i^{(1)}\mathbf{h}_{t-1}^{(1)} + \mathbf{b}_i^{(1)}) \\
\mathbf{c}_t^{(1)} &= \mathbf{f}_t^{(1)} \odot \mathbf{c}_{t-1}^{(1)} + \mathbf{i}_t^{(1)} \odot \tanh(\mathbf{W}_c^{(1)}\mathbf{x}_t + \mathbf{U}_c^{(1)}\mathbf{h}_{t-1}^{(1)} + \mathbf{b}_c^{(1)}) \\
\mathbf{o}_t^{(1)} &= \sigma(\mathbf{W}_o^{(1)}\mathbf{x}_t + \mathbf{U}_o^{(1)}\mathbf{h}_{t-1}^{(1)} + \mathbf{b}_o^{(1)}) \\
\mathbf{h}_t^{(1)} &= \mathbf{o}_t^{(1)} \odot \tanh(\mathbf{c}_t^{(1)})
\end{aligned}
$$

where σ denotes the sigmoid function, $\odot$ denotes element-wise multiplication, and $\mathbf{W}$, $\mathbf{U}$, and $\mathbf{b}$ are weight matrices and bias vectors, respectively.

The hidden state $\mathbf{h}_t^{(2)}$ and cell state $\mathbf{c}_t^{(2)}$ at time step t in the second LSTM layer are computed based on the output of the first LSTM layer:

$$
\begin{aligned}
\mathbf{f}_t^{(2)} &= \sigma(\mathbf{W}_f^{(2)}\mathbf{h}_t^{(1)} + \mathbf{U}_f^{(2)}\mathbf{h}_{t-1}^{(2)} + \mathbf{b}_f^{(2)}) \\
\mathbf{i}_t^{(2)} &= \sigma(\mathbf{W}_i^{(2)}\mathbf{h}_t^{(1)} + \mathbf{U}_i^{(2)}\mathbf{h}_{t-1}^{(2)} + \mathbf{b}_i^{(2)}) \\
\mathbf{c}_t^{(2)} &= \mathbf{f}_t^{(2)} \odot \mathbf{c}_{t-1}^{(2)} + \mathbf{i}_t^{(2)} \odot \tanh(\mathbf{W}_c^{(2)}\mathbf{h}_t^{(1)} + \mathbf{U}_c^{(2)}\mathbf{h}_{t-1}^{(2)} + \mathbf{b}_c^{(2)}) \\
\mathbf{o}_t^{(2)} &= \sigma(\mathbf{W}_o^{(2)}\mathbf{h}_t^{(1)} + \mathbf{U}_o^{(2)}\mathbf{h}_{t-1}^{(2)} + \mathbf{b}_o^{(2)}) \\
\mathbf{h}_t^{(2)} &= \mathbf{o}_t^{(2)} \odot \tanh(\mathbf{c}_t^{(2)})
\end{aligned}
$$

Let $\mathbf{y}_t$ be the final output at time step t, which is computed as:

$$
\mathbf{y}_t = \mathbf{W}_d\mathbf{h}_t^{(2)} + \mathbf{b}_d
$$

where $\mathbf{W}_d$ and $\mathbf{b}_d$ are the weight matrix and bias vector of the dense layer, respectively.

A4. Activation function

Considering a neural network with L layers. The activation of the l-th layer is denoted by $a^{[l]}$, and $W^{[l]}$ and $b^{[l]}$ are the weights and biases for layer l. The forward propagation for a given layer is:

$$
z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \tag{1}
$$

The activation $a^{[l]}$ is passed through a non-linear activation function σ:

$$
a^{[l]} = \sigma(z^{[l]}) \tag{2}
$$

During backpropagation, we compute the gradients of the loss L with respect to the weights. The gradient at layer l is:

$$
\delta^{[l]} = \frac{\partial L}{\partial z^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \cdot \frac{\partial a^{[l]}}{\partial z^{[l]}} \tag{3}
$$

For common activation functions like the sigmoid or hyperbolic tangent (tanh), the derivatives can become very small:

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z)) \quad \text{(sigmoid)} \tag{4}$$

$$\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z) \quad \text{(tanh)} \tag{5}$$

When the gradient $\delta^{[l]}$ becomes very small, it results in very small updates to the weights $W^{[l]}$, leading to the vanishing gradient problem.

B. More precision on the models used for robustness testing

In this appendix, we try to give more detailed explanations on the models we used in order to test the robustness of our algorithm and generate synthetic data based on our initial raw historical dataset.

These precisions, particularly the mathematical formalizations are extracted from the following sources :

- Heston, S. L. (1993). A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. The Review of Financial Studies, 6(2), 327-343.
- Bates, D. S. (1996). Jumps and Stochastic Volatility: Exchange Rate Processes Implicit in Deutsche Mark Options. The Review of Financial Studies, 9(1), 69-107.
- Madan, D. B., & Seneta, E. (1990). The Variance Gamma (VG) Model for Share Market Returns. Journal of Business, 63(4), 511-524.
- Cox, J. C., Ingersoll, J. E., & Ross, S. A. (1985). A Theory of the Term Structure of Interest Rates. Econometrica, 53(2), 385-407.
- Bayer, C., Friz, P. K., Gatheral, J. (2016). Pricing under rough volatility. Quantitative Finance, 16(6), 887-904.

B1. Heston Model

The Heston model accounts for the fact that volatility, a measure of the asset's price fluctuations, is not constant but varies over time.

The Heston model is defined by the following system of stochastic differential equations (SDEs):

$$dS_t = \mu S_t \, dt + \sqrt{V_t} S_t \, dW_t^S, \tag{6}$$

$$dV_t = \kappa(\theta - V_t) \, dt + \sigma \sqrt{V_t} \, dW_t^V, \tag{7}$$

$$dW_t^S \cdot dW_t^V = \rho \, dt, \tag{8}$$

where:

S_t is the asset price at time t,
V_t is the variance (squared volatility) at time t,
μ is the drift rate of the asset price (the expected return),
κ is the rate at which V_t reverts to θ (speed of mean reversion),
θ is the long-run average price variance (long-term mean),
σ is the volatility of the variance process (volatility of volatility),
ρ is the correlation between the two Wiener processes W_t^S and W_t^V,
W_t^S is a standard Wiener process driving the asset price,
W_t^V is a standard Wiener process driving the variance.

Equation (1) describes the evolution of the asset price S_t. The term $\mu S_t\,dt$ represents the deterministic part of the asset price change, while the term $\sqrt{V_t}S_t\,dW_t^S$ represents the stochastic part, with $\sqrt{V_t}$ being the stochastic volatility.

Equation (2) describes the evolution of the variance V_t. The term $\kappa(\theta - V_t)\,dt$ represents mean reversion, indicating that V_t tends to revert to the long-term mean θ at a rate κ. The term $\sigma\sqrt{V_t}\,dW_t^V$ represents the stochastic part of the variance.

Equation (3) specifies the instantaneous correlation ρ between the two Wiener processes W_t^S and W_t^V. The correlation affects the joint behavior of the asset price and its variance.

B2. Bates Model

The Bates model is an extension of the Heston stochastic volatility model that includes jumps in the asset price. It is used to capture both the continuous and discontinuous movements of financial assets.

The Bates model is defined by the following system of stochastic differential equations (SDEs):

$$dS_t = S_t\left(\mu\,dt + \sqrt{V_t}\,dW_t^S + (e^J - 1)\,dN_t\right), \tag{1}$$

$$dV_t = \kappa(\theta - V_t)\,dt + \sigma\sqrt{V_t}\,dW_t^V, \tag{2}$$

$$dW_t^S \cdot dW_t^V = \rho\,dt, \tag{3}$$

where:

S_t is the asset price at time t,
V_t is the variance (squared volatility) at time t,
μ is the drift rate of the asset price (the expected return),
κ is the rate at which V_t reverts to θ (speed of mean reversion),
θ is the long-run average price variance (long-term mean),
σ is the volatility of the variance process (volatility of volatility),
ρ is the correlation between the two Wiener processes W_t^S and W_t^V,
W_t^S is a standard Wiener process driving the asset price,
W_t^V is a standard Wiener process driving the variance,
J is the random jump size,
N_t is a Poisson process with intensity λ, representing the number of jumps.

Equation (1) describes the evolution of the asset price S_t. The term $\mu S_t\,dt$ represents the deterministic part of the asset price change, $\sqrt{V_t}S_t\,dW_t^S$ represents the stochastic part with stochastic volatility, and $(e^J - 1)S_t\,dN_t$ represents the jump component where J is the jump size and N_t is the Poisson process.

Equation (2) describes the evolution of the variance V_t. The term $\kappa(\theta - V_t)\,dt$ represents mean reversion, indicating that V_t tends to revert to the long-term mean θ at a rate κ. The term $\sigma\sqrt{V_t}\,dW_t^V$ represents the stochastic part of the variance.

Equation (3) specifies the instantaneous correlation ρ between the two Wiener processes W_t^S and W_t^V. The correlation affects the joint behavior of the asset price and its variance.

B3. VG-CIR Model

The VG-CIR model combines the Variance Gamma process for asset price jumps with the Cox-Ingersoll-Ross (CIR) process for stochastic volatility. This model captures both the discontinuous jumps in asset prices and the stochastic behavior of volatility.

The VG-CIR model is defined by the following system of stochastic differential equations (SDEs):

The Variance Gamma (VG) process is defined by:

$$S_t = S_0 \exp\left(\mu t + \sigma W_{G_t} + \nu G_t\right), \tag{1}$$

where:

$G_t \sim \text{Gamma}(\lambda t, \lambda)$,
W_{G_t} is a Brownian motion evaluated at G_t,
μ is the drift term,
σ is the volatility parameter of the Brownian motion,
ν is the drift of the Gamma process,
λ is the rate of the Gamma process.

The CIR process for stochastic volatility is defined by:

$$dV_t = \kappa(\theta - V_t)\,dt + \sigma_V \sqrt{V_t}\,dW_t^V, \tag{2}$$

where:

V_t is the variance at time t,
κ is the rate at which V_t reverts to θ,
θ is the long-run average variance,
σ_V is the volatility of the variance process,
W_t^V is a Wiener process driving the variance.

The combined model incorporates both the VG process for asset prices and the CIR process for stochastic volatility:

$$dS_t = S_t\left(\mu\,dt + \sqrt{V_t}\,dW_{G_t}\right), \tag{3}$$

where the variance V_t follows the CIR process:

$$dV_t = \kappa(\theta - V_t)\,dt + \sigma_V \sqrt{V_t}\,dW_t^V. \tag{4}$$

Equation (1) describes the evolution of the asset price S_t. The term $\mu S_t\,dt$ represents the deterministic part of the asset price change, while $\sqrt{V_t}S_t\,dW_{G_t}$ represents the stochastic part with volatility $\sqrt{V_t}$, where W_{G_t} is a Brownian motion evaluated at a Gamma process G_t.

Equation (2) describes the evolution of the variance V_t. The term $\kappa(\theta - V_t)\,dt$ represents mean reversion, indicating that V_t tends to revert to the long-term mean θ at a rate κ. The term $\sigma_V \sqrt{V_t}\,dW_t^V$ represents the stochastic part of the variance.

B4. rBergomi model

The rBergomi model is a rough volatility model used to describe the dynamics of financial assets. It extends the classical Bergomi model by incorporating a fractional Brownian motion to account for the roughness in volatility.

The rBergomi model is defined by the following system of stochastic differential equations (SDEs):

The rBergomi model is characterized by the following dynamics for the asset price S_t:

$$S_t = S_0 \exp\left(\int_0^t \left(-\frac{1}{2}V_s\right) ds + \int_0^t \sqrt{V_s} dW_s\right), \tag{1}$$

where the variance process V_t is given by:

$$V_t = \xi_t^2 \exp\left(\eta \int_0^t K(t,s) dW_s^H\right), \tag{2}$$

and

$$\xi_t = \xi_0 \exp\left(-\lambda t\right) + \alpha \int_0^t \exp\left(-\lambda(t-s)\right) dW_s. \tag{3}$$

Equation (1) describes the evolution of the asset price S_t. The term $-\frac{1}{2}V_s$ inside the exponential represents the adjustment for the variance, ensuring the correct martingale property of S_t. The term $\int_0^t \sqrt{V_s} dW_s$ captures the stochastic part of the asset price with volatility $\sqrt{V_s}$.

Equation (2) describes the variance process V_t, who is driven by a combination of a deterministic function ξ_t and a rough fractional Brownian motion term $\int_0^t K(t,s) dW_s^H$ with Hurst parameter H.

The function ξ_t evolves over time and includes an exponential decay term $\xi_0 \exp\left(-\lambda t\right)$ and a stochastic integral term $\alpha \int_0^t \exp\left(-\lambda(t-s)\right) dW_s$.

The kernel function $K(t,s)$ characterizes the roughness of the volatility process, with H being the Hurst parameter that determines the roughness.

C. More precision on the calibration of the robustness testing models in the algorithm

C1. Initial parameter guesses

In this subsection, we give more details and literature-based evidences and founding on the initial parameter guesses for the calibration of our four models : Heston, Bates, VG-CIR and rBergomi.

1) Heston

- **Long-term average variance (θ)**: estimated historical volatility ([5] ; [20]).
- **Initial Variance (V_0)**: approximately equal to θ, so in our case, also the historical volatility ([42]).
- **Rate of mean reversion (κ)**: a commonly assumed value is around 1.0, corresponding to a moderate rate of reversion ([43]).
- **Volatility of volatility (ξ)**: according to empirical studies such as [44], a typical value is around 0.3 to 0.5.
- **Correlation (ρ)**: often assumed to be around -0.5 to -0.7 ([45]).

$$\boxed{[\theta, V_0, \kappa, \xi, \rho] = [\textbf{historical volatility}, \textbf{historical volatility}, 1.0, 0.4, -0.6]}$$

2) Bates

- **Parameters of the Heston model**: we use the same initial guesses as the Heston model.
- **Jump intensity (λ_J)**: Set based on observed jump frequency or literature, typically around 0.1 to 0.3 ([46]).
- **Average jump size (μ_J)**: some typical values are around -0.1 to -0.05 ([47]).
- **Jump volatility (σ_J)**: often around 0.1 to 0.2 ([48]).

$$[\theta, V_0, \kappa, \xi, \rho, \lambda_J, \mu_J, \sigma_J] =$$

$$[\textbf{historical volatility}, \textbf{historical volatility}, 1.0, 0.4, -0.6, 0.2, -0.075, 0.15]$$

3) VG-CIR

- **Long-term average variance (θ)**: same as Heston.
- **Initial variance (V_0)**: same as Heston.
- **Rate of mean reversion (κ)**: same as Heston.
- **Volatility of volatility (ξ)**: same as Heston.
- **VG parameters**:
  - **VG variance (σ_{VG})**: typically around 0.2 to 0.3 ([49]).

– **VG theta (θ_{VG})**: typically around -0.1 to -0.2 ([50]).

– **VG kappa (κ_{VG})**: often set to 0.1 to 0.3 ([51]).

$$[\boldsymbol{\theta}, \boldsymbol{V_0}, \boldsymbol{\kappa}, \boldsymbol{\xi}, \boldsymbol{\sigma_{VG}}, \boldsymbol{\theta_{VG}}, \boldsymbol{\kappa_{VG}}] =$$

$$[\textbf{historical volatility}, \textbf{historical volatility}, \mathbf{1.0}, \mathbf{0.4}, \mathbf{0.25}, \mathbf{-0.15}, \mathbf{0.2}]$$

4) rBergomi

rBergomi Model

- **Volatility of volatility (ξ)**: an initial guess is around 1.0 ([52]).
- **Hurst parameter (H)**: a typical value is around 0.1 to 0.3 ([53]).
- **Initial forward variance curve (F)**: a typical value is around 0.04 ([53]).

$$\boxed{[\boldsymbol{\xi}, \boldsymbol{H}, \boldsymbol{F}] = [\mathbf{1.0}, \mathbf{0.2}, \mathbf{0.04}]}$$

C2. Objective function

In the context of calibrating financial models, the objective function $f(\theta)$ is defined as the sum of squared differences between market prices and model prices of financial instruments, such as options. The parameter vector θ represents the model parameters to be calibrated.

$$f(\theta) = \sum_{i=1}^{n} \left(P_{\text{market},i} - P_{\text{model},i}(\theta)\right)^2, \tag{4}$$

where:

$P_{\text{market},i}$ is the market price of the i-th financial instrument,
$P_{\text{model},i}(\theta)$ is the model price of the i-th financial instrument with parameters θ,
n is the number of financial instruments.

C3. More precision on the calibration method used

In this section, we go in more details into the Limited-memory Broyden–Fletcher–Goldfarb–Shanno with Box constraints (L-BFGS-B) algorithm we use to calibrate the four models we use to test the robustness of our neural networks hedge. Part of the content used to explain this comes from Byrd, R. H., Lu, P., Nocedal, J., & Zhu, C. (1995), A Limited Memory Algorithm for Bound Constrained Optimization, SIAM Journal on Scientific Computing, 16(5), 1190-1208.

The algorithm aims to minimize $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$, and its main steps are the following :

1. Each variable x_i is subject to box constraints :

$$l_i \leq x_i \leq u_i \quad \forall i = 1, 2, \ldots, n$$

2. Compute the gradient $\nabla f(\mathbf{x})$:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]^\top$$

3. Approximate the inverse Hessian matrix :

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \mathbf{Q}_k$$

where $\mathbf{Q}_k$ is a correction term.

4. The search direction $\mathbf{d}_k$ at iteration k is :

$$\mathbf{d}_k = -\mathbf{B}_k \nabla f(\mathbf{x}_k)$$

5. Find the optimal step size α_k that satisfies the Wolfe conditions :

$$f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \nabla f(\mathbf{x}_k)^\top \mathbf{d}_k$$

$$\nabla f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)^\top \mathbf{d}_k \geq c_2 \nabla f(\mathbf{x}_k)^\top \mathbf{d}_k$$

where $0 < c_1 < c_2 < 1$.

6. Update the variables :

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$$

7. Project the variables back to the feasible region :

$$x_{k+1,i} = \min(\max(x_{k+1,i}, l_i), u_i) \quad \forall i = 1, 2, \ldots, n$$

C4. Synthetic data paths

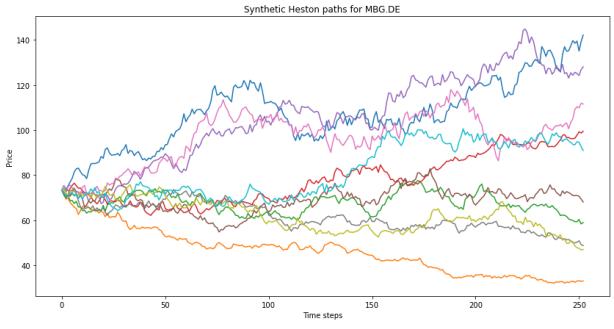


Figure 2: Synthetic paths generation with Heston model from MBG historical stock price data

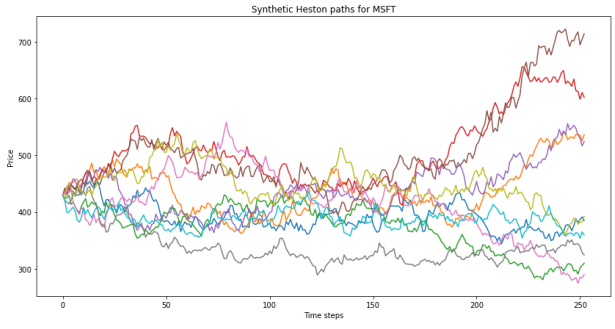


Figure 3: Synthetic paths generation with Heston model from MSFT historical stock price data

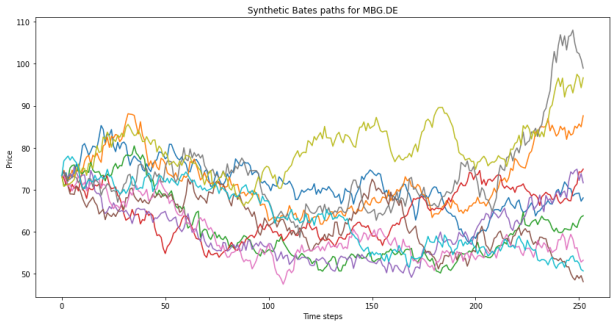


Figure 4: Synthetic paths generation with Bates model from MBG historical stock price data

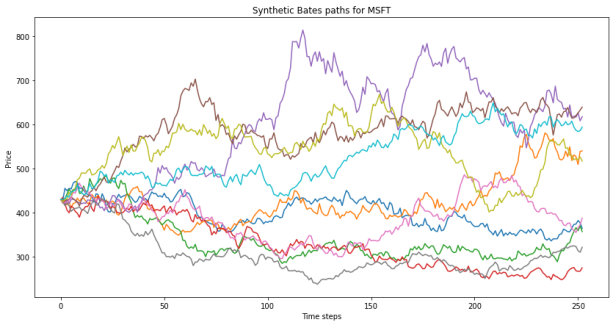


Figure 5: Synthetic paths generation with Bates model from MSFT historical stock price data

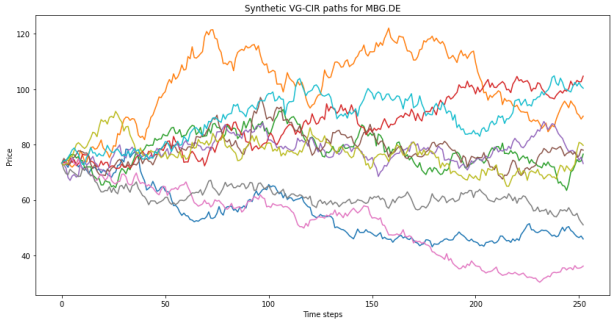


Figure 6: Synthetic paths generation with VG-CIR model from MBG historical stock price data

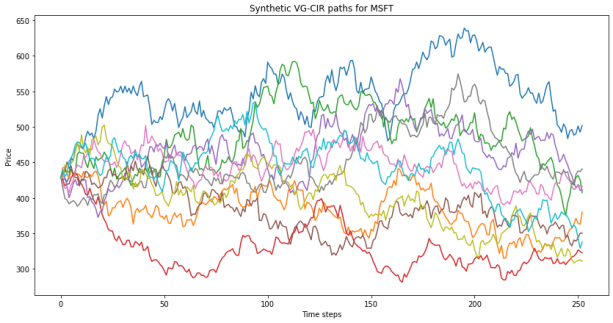


Figure 7: Synthetic paths generation with VG-CIR model from MSFT historical stock price data

D. Illustration of the impact of transaction costs on the effectiveness of the Black Scholes model

To illustrate more in details the impact of transaction costs on the Black-Scholes model mentioned in 4.5, we give below a concrete example.

The Black-Scholes model assumes that a portfolio can be continuously rebalanced to maintain a risk-free position through a delta-hedging strategy.

- S_t is the price of the underlying asset at time t.
- Δ_t is the delta of the option i.e. the sensitivity of the option's price to the underlying asset's price.
- $V(t, S_t)$ is the price of the option at time t.

The portfolio Π_t at time t consists of:

- Δ_t shares of the underlying asset S_t
- Cash position that includes proceeds from or costs of transactions.

Without transaction costs, the continuous delta-hedging strategy involves continuously adjusting the number of shares Δ_t to perfectly hedge the option's price:

$$\Pi_t = \Delta_t S_t - V(t, S_t)$$

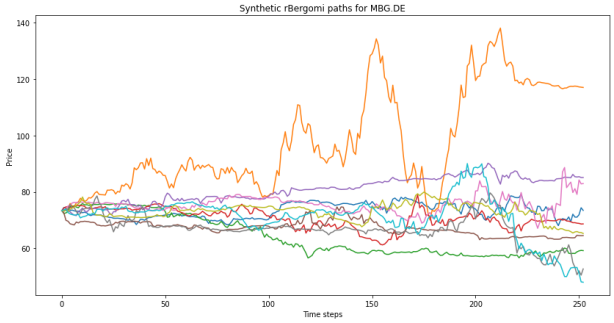


Figure 8: Synthetic paths generation with rBergomi model from MBG historical stock price data

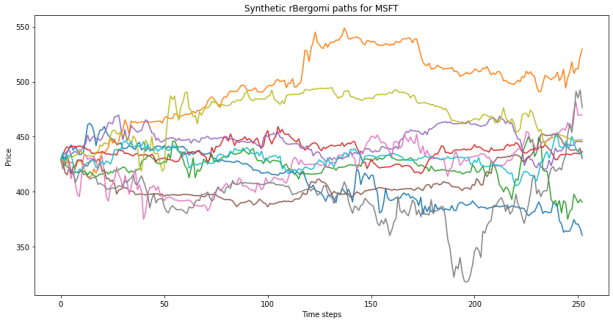


Figure 9: Synthetic paths generation with rBergomi model from MSFT historical stock price data

Now, let c be the proportional transaction cost rate. Every time we adjust Δ_t, we incur a cost proportional to the change in the position size. The cost incurred when adjusting from $\Delta_{t-\Delta t}$ to Δ_t is:

$$\text{Cost} = c \times S_t \times |\Delta_t - \Delta_{t-\Delta t}|$$

Over a hedging period T, the total transaction costs can be approximated as the sum of the costs incurred at each adjustment step, which, when continuously adjusting the position, can be represented as :

$$\text{Total Cost} \approx c \int_0^T S_t \left| \frac{d\Delta_t}{dt} \right| dt$$

When adding transaction costs, we need to account for these costs when valuating the implied portfolio :

$$\Pi_t = \Delta_t S_t - V(t, S_t) - \text{Total Cost}$$

Example

Suppose $c = 0.01$ (1% transaction cost), and assume we rebalance every day for a month (30 days). Let S_t change by \$1 each day, and Δ_t changes by 0.05 each day.

Below are the total transaction cost over the month :

$$\text{Total Cost} = \sum_{i=1}^{30} c \times S_i \times |\Delta_i - \Delta_{i-1}|$$

$$\text{Total Cost} = 0.01 \times \sum_{i=1}^{30} S_i \times 0.05$$

$$\text{Assuming } S_i \approx 100 \text{ on average,}$$

$$\text{Total Cost} \approx 0.01 \times 30 \times 100 \times 0.05 = 1.5$$

Thus, transaction costs of \$1.5 significantly impact the profitability and efficiency of the hedging strategy.

E. Notes on additional loss functions used for comparison

This section of the appendix complements part 5.1, where we introduce two additional loss functions to compare with the initial loss functions (RMSE and MAE) selected for our algorithm.

These precisions, particularly the mathematical formalizations are extracted from the following sources :

- Huber, P. J. (1964). Robust Estimation of a Location Parameter. The Annals of Mathematical Statistics, 35(1), 73-101.
- Zhang, Z. (2020). Machine Learning Algorithms: Popular Algorithms for Data Science and Machine Learning.

E1. Huber loss function

The Huber loss function is defined as follows:

$$\mathcal{L}_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \delta \\ \delta\left(|y - \hat{y}| - \frac{1}{2}\delta\right) & \text{for } |y - \hat{y}| > \delta \end{cases}$$

where:

y is the true value,

$\hat{y}$ is the predicted value,

δ is a threshold parameter.

The Huber loss function combines the mean squared error (MSE) for small residuals and the mean absolute error (MAE) for large residuals, providing robustness to outliers.

E2. Logarithm of the hyperbolic cosine (log-cosh) loss function

The log-cosh loss function is defined as follows:

$$\mathcal{L}(y, \hat{y}) = \log\left(\cosh\left(\hat{y} - y\right)\right)$$

where:

y is the true value,

$\hat{y}$ is the predicted value.

The hyperbolic cosine function, $\cosh(x)$, is given by:

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

Thus, the log-cosh loss can also be expressed as:

$$\mathcal{L}(y, \hat{y}) = \log\left(\frac{e^{\hat{y}-y} + e^{-(\hat{y}-y)}}{2}\right)$$

F. Source code

F1. Initial configuration

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Input, ReLU
from tensorflow.keras.optimizers import Adam

# Getting the directory of the current script and defining the relative paths to the data files
dir = os.path.dirname(__file__)
mbg_path = os.path.join(dir, 'Data', 'D1. Raw starting data', 'MBG.DE')
msft_path = os.path.join(dir, 'Data', 'D1. Raw starting data', 'MSFT')

df_mbg = pd.read_csv(mbg_path, delimiter=",")
df_msft = pd.read_csv(msft_path, delimiter=",")

# Computing daily returns and rolling volatility (standard deviation of returns) for our two datasets
df_mbg['Return'] = df_mbg['Close'].pct_change()
df_msft['Return'] = df_msft['Close'].pct_change()

# We use a window size of 30 days for calculating the volatility
window_size_vol = 30

# Annualizing the volatilities
df_mbg['Volatility'] = df_mbg['Return'].rolling(window=window_size_vol).std() * np.sqrt(252)
df_msft['Volatility'] = df_msft['Return'].rolling(window=window_size_vol).std() * np.sqrt(252)

# Dropping rows with NaN and extracting the last 30 rows
df_mbg = df_mbg.dropna().reset_index(drop=True)
df_msft = df_msft.dropna().reset_index(drop=True)
df_mbg_last_30 = df_mbg.tail(30).reset_index(drop=True)
df_msft_last_30 = df_msft.tail(30).reset_index(drop=True)

print("Initial MBG data:")
print(df_mbg.head())
print("Initial MSFT data:")
```

```
print(df_msft.head())

# Scaling the data with MinMaxScaler()
scaler_mbg = MinMaxScaler()
scaler_msft = MinMaxScaler()
df_mbg['Close'] = scaler_mbg.fit_transform(df_mbg[['Close']])
df_msft['Close'] = scaler_msft.fit_transform(df_msft[['Close']])

print("Preprocessed MBG data:")
print(df_mbg.head())
print("Preprocessed MSFT data:")
print(df_msft.head())

# Preparation of the training data
# The function prepare_training_data transforms our preprocessed data (stock prices) into input-output pairs for model training
# For each position in the series, it extracts a subsequence of length n_steps as the input (X) and the following value as the output (y)
# We generate multiple input-output pairs, which are then converted into numpy arrays and returned to train the stock prices
def prepare_training_data(series, n_steps):
    X, y = [], []
    for i in range(len(series) - n_steps):
        seq_X = series[i:i+n_steps]
        seq_y = series[i+n_steps]
        X.append(seq_X)
        y.append(seq_y)
    return np.array(X), np.array(y)

n_steps = 30
X_mbg, y_mbg = prepare_training_data(df_mbg['Close'].values, n_steps)
X_msft, y_msft = prepare_training_data(df_msft['Close'].values, n_steps)

# Reshaping data for our LSTM model and splitting the data into training and testing
# We train the data on the whole dataset minus the last 30 days of the dataset and test it on the last 30 days of the dataset
X_mbg = X_mbg.reshape((X_mbg.shape[0], X_mbg.shape[1], 1))
X_msft = X_msft.reshape((X_msft.shape[0], X_msft.shape[1], 1))

split_idx_mbg = len(X_mbg) - 30
split_idx_msft = len(X_msft) - 30

X_train_mbg, X_test_mbg = X_mbg[:split_idx_mbg], X_mbg[split_idx_mbg:]
y_train_mbg, y_test_mbg = y_mbg[:split_idx_mbg], y_mbg[split_idx_mbg:]
```

```
77
78 X_train_msft, X_test_msft = X_msft[:split_idx_msft], X_msft[split_idx_msft:]
79 y_train_msft, y_test_msft = y_msft[:split_idx_msft], y_msft[split_idx_msft:]
80
81
82
83 # Definition of the LSTM model
84 # The function create_lstm_model constructs and compiles our LSTM-based neural network
   model
85 # We initializes a Sequential model and adds an input layer with a specified shape,
   followed by two LSTM layers with a specified number of neurons and ReLU activation
86 # The compiled model is returned using the Adam optimizer and a specified loss function
   (which is specified right after in our code, depending on the dataset)
87 def create_lstm_model(input_shape, num_neurons, loss_function):
88     model = Sequential([
89         Input(shape=input_shape),
90         LSTM(num_neurons, activation='relu', return_sequences=True),
91         LSTM(num_neurons, activation='relu'),
92         Dense(1, activation='sigmoid')
93     ])
94     model.compile(optimizer='adam', loss=loss_function)
95     return model
96
97
98 # Training of the model on both dataset, with MSE as a loss function for MSFT and MAE
   for MBG
99 model_msft = create_lstm_model((n_steps, 1), 200, 'mse')
100 history_msft = model_msft.fit(X_train_msft, y_train_msft, epochs=30, batch_size=32,
   verbose=0)
101
102 model_mbg = create_lstm_model((n_steps, 1), 20, 'mae')
103 history_mbg = model_mbg.fit(X_train_mbg, y_train_mbg, epochs=30, batch_size=16,
   verbose=0)
104
105 # Plotting the training loss for checking
106 plt.figure(figsize=(12, 6))
107
108 plt.subplot(1, 2, 1)
109 plt.plot(history_mbg.history['loss'])
110 plt.title('MBG Model Training Loss')
111
112 plt.subplot(1, 2, 2)
113 plt.plot(history_msft.history['loss'])
114 plt.title('MSFT Model Training Loss')
```

```
115
116 plt.tight_layout()
117 plt.show()
118
119 # Predicting prices using our pre-trained models and printing them
120 predictions_msft = model_msft.predict(X_test_msft)
121 predictions_mbg = model_mbg.predict(X_test_mbg)
122
123 print("Raw MSFT Predictions:")
124 print(predictions_msft)
125
126 print("Raw MBG Predictions:")
127 print(predictions_mbg)
128
129 # Computing deltas with the function compute_deltas
130 # The function calculates the differences between our consecutive prices predictions
    values : it initializes an array deltas with the same shape as predictions, sets the
    first element to zero, and computes the difference between each pair of consecutive
    prices predictions
131 # The results are stored in the subsequent elements of deltas, and the function returns
    the resulting array of deltas
132 def compute_deltas(predictions):
133     deltas = np.zeros_like(predictions)
134     deltas[1:] = predictions[1:] - predictions[:-1]
135     return deltas
136
137 deltas_mbg = compute_deltas(predictions_mbg)
138 deltas_msft = compute_deltas(predictions_msft)
139
140 # Normalization and adjustment of deltas
141 deltas_mbg = np.abs(deltas_mbg)
142 deltas_msft = np.abs(deltas_msft)
143
144 deltas_mbg = MinMaxScaler().fit_transform(deltas_mbg)
145 deltas_msft = MinMaxScaler().fit_transform(deltas_msft)
146
147 # Initialization of NN_Delta with NaNs and filling of NN_Delta with the computed deltas
148 df_mbg_last_30['NN_Delta'] = np.nan
149 df_msft_last_30['NN_Delta'] = np.nan
150
151 df_mbg_last_30.loc[:, 'NN_Delta'] = deltas_mbg.flatten()
152 df_msft_last_30.loc[:, 'NN_Delta'] = deltas_msft.flatten()
153
154 # Printing the results
```

```
print(df_mbg_last_30)
print(df_msft_last_30)

# Defining the Black-Scholes function with the given formulas from the BS model
def black_scholes(S, K, T, r, sigma, option_type='call'):
    from scipy.stats import norm
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    if option_type == 'call':
        price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
        delta = norm.cdf(d1)
    elif option_type == 'put':
        price = K * np.exp(-r * T) * norm.cdf(-d2) - S * norm.cdf(-d1)
        delta = -norm.cdf(-d1)
    return price, delta

# The calculate_bs function computes Black-Scholes prices and deltas using a rolling ATM strike (cf report part 3.3 for more informations on it)
# We initialize the lists bs_prices and bs_deltas with NaNs for the specified window_size (in our case it will be 30 days)
# For each data point beyond the initial window_size, we calculate the current price (S), the strike price from window_size days ago (K), one day to maturity (T), a risk-free rate (r), and current volatility (sigma)
# If sigma is not NaN, we use the black_scholes function to compute the price and delta, and append them to the lists; otherwise, we append NaNs to the list
# The function finally returns the lists bs_prices and bs_deltas.
def calculate_bs(df, window_size):
    bs_prices = [np.nan] * window_size
    bs_deltas = [np.nan] * window_size

    for i in range(window_size, len(df)):
        S = df['Close'].iloc[i]  # Current price
        K = df['Close'].iloc[i - window_size]
        T = 1 / 252
        r = 0.0
        sigma = df['Volatility'].iloc[i]

        if not np.isnan(sigma):
            price, delta = black_scholes(S, K, T, r, sigma)
            bs_prices.append(price)
            bs_deltas.append(delta)
        else:
            bs_prices.append(np.nan)
            bs_deltas.append(np.nan)
```

```
194
195     return bs_prices, bs_deltas
196
197 # Calculation of Black-Scholes prices and deltas with rolling ATM strike prices (we
    choose a window size of 30 days, more on this on part 3.3 of our report) for each
    dataset
198 window_size = 30
199 bs_prices_mbg, bs_deltas_mbg = calculate_bs(df_mbg, window_size)
200 bs_prices_msft, bs_deltas_msft = calculate_bs(df_msft, window_size)
201
202 # Adding the calculated BS prices and deltas to the dataframes and printing the last few
    rows to verify
203 # Since our model aims to predict deltas for the last 30 days of our dataset with the
    training on the whole dataset minus these last 30 days
204 df_mbg_last_30['BS_Price'] = bs_prices_mbg[-30:]
205 df_mbg_last_30['BS_Delta'] = bs_deltas_mbg[-30:]
206
207 df_msft_last_30['BS_Price'] = bs_prices_msft[-30:]
208 df_msft_last_30['BS_Delta'] = bs_deltas_msft[-30:]
209
210 print("Final MBG data with BS calculations:")
211 print(df_mbg_last_30[['Date', 'Close', 'BS_Price', 'BS_Delta', 'NN_Delta']])
212 print("Final MSFT data with BS calculations:")
213 print(df_msft_last_30[['Date', 'Close', 'BS_Price', 'BS_Delta', 'NN_Delta']])
214
215 # Visualization of the hedge ratios for the last 30 days
216 plt.figure(figsize=(14, 7))
217
218 plt.subplot(2, 1, 1)
219 plt.plot(df_mbg_last_30['NN_Delta'], label='NN delta', color='blue')
220 plt.plot(df_mbg_last_30['BS_Delta'], label='BS delta', color='red')
221 plt.fill_between(range(len(df_mbg_last_30['NN_Delta'])), df_mbg_last_30['NN_Delta'],
    alpha=0.3, color='blue')
222 plt.fill_between(range(len(df_mbg_last_30['BS_Delta'])), df_mbg_last_30['BS_Delta'],
    alpha=0.3, color='red')
223 plt.title('MBG hedge ratios')
224 plt.xlabel('Date')
225 plt.ylabel('Hedge ratio')
226 plt.legend()
227
228 plt.subplot(2, 1, 2)
229 plt.plot(df_msft_last_30['NN_Delta'], label='NN delta', color='blue')
230 plt.plot(df_msft_last_30['BS_Delta'], label='BS delta', color='red')
231 plt.fill_between(range(len(df_msft_last_30['NN_Delta'])), df_msft_last_30['NN_Delta'],
    alpha=0.3, color='blue')
```

```
232 plt.fill_between(range(len(df_msft_last_30['BS_Delta'])), df_msft_last_30['BS_Delta'],
    alpha=0.3, color='red')
233 plt.title('MSFT hedge ratios')
234 plt.xlabel('Date')
235 plt.ylabel('Hedge ratio')
236 plt.legend()
237
238 plt.tight_layout()
239 plt.show()
240
241 # Denormalization of predicted prices for visualization
242 denorm_predictions_mbg = scaler_mbg.inverse_transform(predictions_mbg)
243 denorm_predictions_msft = scaler_msft.inverse_transform(predictions_msft)
244
245 # Visualize predicted prices vs actual prices for last 30 days
246 plt.figure(figsize=(14, 7))
247
248 plt.subplot(2, 1, 1)
249 plt.plot(df_mbg_last_30['Close'], label='Actual price', color='black')
250 plt.plot(denorm_predictions_mbg, label='Predicted price', color='green')
251 plt.title('MBG predicted vs actual prices')
252 plt.xlabel('Date')
253 plt.ylabel('Price')
254 plt.legend()
255
256 plt.subplot(2, 1, 2)
257 plt.plot(df_msft_last_30['Close'], label='Actual price', color='black')
258 plt.plot(denorm_predictions_msft, label='Predicted price', color='green')
259 plt.title('MSFT predicted vs actual prices')
260 plt.xlabel('Date')
261 plt.ylabel('Price')
262 plt.legend()
263
264 plt.tight_layout()
265 plt.show()
266
267 # Definition of a PnL test with the function calculate_pnl
268 # We base the computation on a specified hedge column, and initialize a list pnl with
    zero for the first row
269 # Then, for each subsequent row, we calculates the PnL as the difference in closing
    prices adjusted by the previous hedge value, and append the result to PnL
270 # The function returns the cumulative sum of the PnL values
271 def calculate_pnl(df, hedge_column):
272     pnl = [0]
```

```
    for i in range(1, len(df)):
        pnl.append(df['Close'].iloc[i] - df['Close'].iloc[i-1] *
        df[hedge_column].iloc[i-1])
    return np.cumsum(pnl)

# Calculation of the PnL for each dataset and adding them to the results dataframe and
printing of the last few rows to verify
df_mbg_last_30['NN_PnL'] = calculate_pnl(df_mbg_last_30, 'NN_Delta')
df_mbg_last_30['BS_PnL'] = calculate_pnl(df_mbg_last_30, 'BS_Delta')

df_msft_last_30['NN_PnL'] = calculate_pnl(df_msft_last_30, 'NN_Delta')
df_msft_last_30['BS_PnL'] = calculate_pnl(df_msft_last_30, 'BS_Delta')

print("Final MBG Data with BS calculations and PnL:")
print(df_mbg_last_30[['Date', 'Close', 'BS_Price', 'BS_Delta', 'NN_Delta', 'NN_PnL',
'BS_PnL']])
print("Final MSFT Data with BS calculations and PnL:")
print(df_msft_last_30[['Date', 'Close', 'BS_Price', 'BS_Delta', 'NN_Delta', 'NN_PnL',
'BS_PnL']])


# Visualization of the PnL for the last 30 days
plt.figure(figsize=(14, 7))

plt.subplot(2, 1, 1)
plt.plot(df_mbg_last_30['NN_PnL'], label='NN PnL', color='blue')
plt.plot(df_mbg_last_30['BS_PnL'], label='BS PnL', color='red')
plt.title('MBG PnL comparison')
plt.xlabel('Date')
plt.ylabel('PnL')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(df_msft_last_30['NN_PnL'], label='NN PnL', color='blue')
plt.plot(df_msft_last_30['BS_PnL'], label='BS PnL', color='red')
plt.title('MSFT PnL comparison')
plt.xlabel('Date')
plt.ylabel('PnL')
plt.legend()

plt.tight_layout()
plt.show()

# Calculate daily returns for the NN PnL and replacing infinities and NaNs resulting
from pct_change calculation
```

```
df_mbg_last_30['NN_Return'] = df_mbg_last_30['NN_PnL'].pct_change()
df_msft_last_30['NN_Return'] = df_msft_last_30['NN_PnL'].pct_change()

df_mbg_last_30['NN_Return'].replace([np.inf, -np.inf], np.nan, inplace=True)
df_mbg_last_30['NN_Return'].fillna(0, inplace=True)

df_msft_last_30['NN_Return'].replace([np.inf, -np.inf], np.nan, inplace=True)
df_msft_last_30['NN_Return'].fillna(0, inplace=True)

# Calculate Sharpe ratio with a risk-free rate of 1%
# We compute the mean and standard deviation of the 'NN_Return' column for the last 30 days of both df_mbg_last_30 and df_msft_last_30
# Then, the Sharpe ratio is calculated by subtracting the daily risk-free rate (annual rate divided by 252 trading days) from the mean return and dividing by the standard deviation of returns for both datasets
# The results are stored in sharpe_ratio_mbg and sharpe_ratio_msft
risk_free_rate = 0.01

mean_return_mbg = df_mbg_last_30['NN_Return'].mean()
std_return_mbg = df_mbg_last_30['NN_Return'].std()
sharpe_ratio_mbg = (mean_return_mbg - risk_free_rate/252) / std_return_mbg

mean_return_msft = df_msft_last_30['NN_Return'].mean()
std_return_msft = df_msft_last_30['NN_Return'].std()
sharpe_ratio_msft = (mean_return_msft - risk_free_rate/252) / std_return_msft

print(f'Sharpe ratio for MBG: {sharpe_ratio_mbg}')
print(f'Sharpe ratio for MSFT: {sharpe_ratio_msft}')

# The function calculate_max_drawdown computes the maximum drawdown of our PnL series then finds the cumulative maximum of these returns (peak)
# We compute the drawdown as the difference between cumulative returns and the peak, divided by the peak
# The function returns this maximum drawdown value
def calculate_max_drawdown(pnl_series):
    cumulative_returns = (1 + pnl_series).cumprod()
    peak = cumulative_returns.cummax()
    drawdown = (cumulative_returns - peak) / peak
    max_drawdown = drawdown.min()
    return max_drawdown

# Appliying the maximum drawdown for both datasets and printing them
max_drawdown_mbg = calculate_max_drawdown(df_mbg_last_30['NN_Return'])
max_drawdown_msft = calculate_max_drawdown(df_msft_last_30['NN_Return'])
```

```
352
353 print(f'Maximum drawdown for MBG: {max_drawdown_mbg}')
354 print(f'Maximum drawdown for MSFT: {max_drawdown_msft}')
355 \end{lstlisting}
356
```

F2. Additional loss functions and number of neurons

```
1  from scipy.optimize import minimize
2  from tensorflow.keras.losses import Huber, LogCosh
3
4  df_mbg_v2 = pd.read_csv(mbg_path, delimiter=",")
5  df_msft_v2 = pd.read_csv(msft_path, delimiter=",")
6
7  # Computing daily returns and rolling volatility
8  df_mbg_v2['Return'] = df_mbg_v2['Close'].pct_change()
9  df_msft_v2['Return'] = df_msft_v2['Close'].pct_change()
10
11 window_size_vol = 30
12 df_mbg_v2['Volatility'] = df_mbg_v2['Return'].rolling(window=window_size_vol).std() *
   np.sqrt(252)
13 df_msft_v2['Volatility'] = df_msft_v2['Return'].rolling(window=window_size_vol).std() *
   np.sqrt(252)
14
15 # Dropping rows with NaN values and extracting the last 30 rows
16 df_mbg_v2 = df_mbg_v2.dropna().reset_index(drop=True)
17 df_msft_v2 = df_msft_v2.dropna().reset_index(drop=True)
18
19 df_mbg_last_30_v2 = df_mbg_v2.tail(30).reset_index(drop=True)
20 df_msft_last_30_v2 = df_msft_v2.tail(30).reset_index(drop=True)
21
22 print("Initial MBG data:")
23 print(df_mbg_v2.head())
24 print("Initial MSFT data:")
25 print(df_msft_v2.head())
26
27 # Scaling the data using the scaler functions (scaler_mbg and scaler_msft) defined
   previously
28 df_mbg_v2['Close'] = scaler_mbg.fit_transform(df_mbg[['Close']])
29 df_msft_v2['Close'] = scaler_msft.fit_transform(df_msft[['Close']])
30
31 print("Preprocessed MBG data:")
```

```
print(df_mbg_v2.head())
print("Preprocessed MSFT data:")
print(df_msft_v2.head())

# Preparing the training of the data using the previously defined function prepare_training_data
X_mbg_v2, y_mbg_v2 = prepare_training_data(df_mbg_v2['Close'].values, n_steps)
X_msft_v2, y_msft_v2 = prepare_training_data(df_msft_v2['Close'].values, n_steps)


# Changing the shape of X_mbg and X_msft to have three dimensions: the number of samples, the number of time steps, and one feature per time step
# Necessary for compatibility with the LSTM layer, which expects input in the form (samples, time steps, features)
X_mbg_v2 = X_mbg_v2.reshape((X_mbg_v2.shape[0], X_mbg_v2.shape[1], 1))
X_msft_v2 = X_msft_v2.reshape((X_msft_v2.shape[0], X_msft_v2.shape[1], 1))

# Splitting data into training and testing as in the initial run
split_idx_mbg_v2 = len(X_mbg_v2) - 30
split_idx_msft_v2 = len(X_msft_v2) - 30

X_train_mbg_v2, X_test_mbg_v2 = X_mbg_v2[:split_idx_mbg_v2], X_mbg_v2[split_idx_mbg_v2:]
y_train_mbg_v2, y_test_mbg_v2 = y_mbg_v2[:split_idx_mbg_v2], y_mbg_v2[split_idx_mbg_v2:]

X_train_msft_v2, X_test_msft_v2 = X_msft_v2[:split_idx_msft_v2], X_msft_v2[split_idx_msft_v2:]
y_train_msft_v2, y_test_msft_v2 = y_msft_v2[:split_idx_msft_v2], y_msft_v2[split_idx_msft_v2:]


# Training of the model on both datasets with different loss functions and neuron counts (we choose number of neurons that are around the initial parameters upward and downward)
loss_functions_msft = [MeanSquaredError(), MeanAbsoluteError(), Huber(), LogCosh()]
neuron_counts_msft = [200, 180, 150, 220, 250]
results_msft = {}

for neurons in neuron_counts_msft:
    for loss_function in loss_functions_msft:
        model_msft = create_lstm_model((n_steps, 1), neurons, loss_function)
        history_msft = model_msft.fit(X_train_msft_v2, y_train_msft_v2, epochs=30, batch_size=32, verbose=0)
        predictions_msft = model_msft.predict(X_test_msft_v2)
        results_msft[f'{loss_function.name}_{neurons}'] = {
```

```
            'history': history_msft.history['loss'],
            'predictions': scaler_msft.inverse_transform(predictions_msft)
        }

loss_functions_mbg = [MeanSquaredError(), MeanAbsoluteError(), Huber(), LogCosh()]
neuron_counts_mbg = [20, 10, 15, 30, 40]
results_mbg = {}

for neurons in neuron_counts_mbg:
    for loss_function in loss_functions_mbg:
        model_mbg = create_lstm_model((n_steps, 1), neurons, loss_function)
        history_mbg = model_mbg.fit(X_train_mbg_v2, y_train_mbg_v2, epochs=30,
        batch_size=16, verbose=0)
        predictions_mbg = model_mbg.predict(X_test_mbg_v2)
        results_mbg[f'{loss_function.name}_{neurons}'] = {
            'history': history_mbg.history['loss'],
            'predictions': scaler_mbg.inverse_transform(predictions_mbg)
        }

# Plotting training loss for each loss function and neuron count
plt.figure(figsize=(16, 16))

plt.subplot(2, 2, 1)
for key, result in results_msft.items():
    plt.plot(result['history'], label=key)
plt.title('MSFT model training loss')
plt.legend()

plt.subplot(2, 2, 2)
for key, result in results_mbg.items():
    plt.plot(result['history'], label=key)
plt.title('MBG model training loss')
plt.legend()

plt.tight_layout()
plt.show()


# Initializing dataFrame for last 30 days of data with NaNs
df_mbg_last_30_v2.drop(columns=['NN_Delta'], inplace=True, errors='ignore')
df_msft_last_30_v2.drop(columns=['NN_Delta'], inplace=True, errors='ignore')

# Computing and filling NN_Delta for each loss function and neuron count (we use the
compute_deltas function that was already initialized in the initial configuration of our
NN)
```

```
for key, result in results_mbg.items():
    deltas_mbg_v2 = compute_deltas(result['predictions'])
    deltas_mbg_v2 = np.abs(deltas_mbg_v2)
    deltas_mbg_v2 = MinMaxScaler().fit_transform(deltas_mbg_v2)
    df_mbg_last_30_v2[f'NN_Delta_{key}'] = deltas_mbg_v2.flatten()

for key, result in results_msft.items():
    deltas_msft_v2 = compute_deltas(result['predictions'])
    deltas_msft_v2 = np.abs(deltas_msft_v2)
    deltas_msft_v2 = MinMaxScaler().fit_transform(deltas_msft_v2)
    df_msft_last_30_v2[f'NN_Delta_{key}'] = deltas_msft_v2.flatten()

# Printing results
print(df_mbg_last_30_v2)
print(df_msft_last_30_v2)
```

F3. Robustness testing

```
df_mbg_v3 = pd.read_csv(mbg_path, delimiter=",")
df_msft_v3 = pd.read_csv(msft_path, delimiter=",")

# Displaying the first few rows of the datasets
print("MBG.DE data head:")
print(df_mbg_v3.head())
print("MSFT data head:")
print(df_msft_v3.head())

# Calculating log returns and historical volatility (annualized)
df_mbg_v3['Log_Returns'] = np.log(df_mbg_v3['Close'] / df_mbg_v3['Close'].shift(1))
df_msft_v3['Log_Returns'] = np.log(df_msft_v3['Close'] / df_msft_v3['Close'].shift(1))
df_mbg_v3.dropna(inplace=True)
df_msft_v3.dropna(inplace=True)

historical_volatility_mbg = df_mbg_v3['Log_Returns'].std() * np.sqrt(252)
historical_volatility_msft = df_msft_v3['Log_Returns'].std() * np.sqrt(252)

print(f"Historical volatility for MBG.DE: {historical_volatility_mbg}")
print(f"Historical volatility for MSFT: {historical_volatility_msft}")

# Normalizing the data
scaler_mbg = MinMaxScaler(feature_range=(0, 1))
scaler_msft = MinMaxScaler(feature_range=(0, 1))
```

```
scaled_mbg_v3 = scaler_mbg.fit_transform(df_mbg_v3[['Close', 'Log_Returns']])
scaled_msft_v3 = scaler_msft.fit_transform(df_msft_v3[['Close', 'Log_Returns']])

# Defining the heston_model_params function that estimates the parameters of the Heston model
# It takes two arguments: log_returns and historical_volatility, and an inner function named objective calculates the sum of squared differences between the log_returns and their mean for given parameters (kappa, theta, sigma, rho, and v0). This objective function serves as the target for the optimization process.
# The minimize function from the SciPy library is used to perform the optimization, employing the L-BFGS-B method, and function finds the set of parameters that minimize the objective function
# The estimated Heston model parameters are returned
def heston_model_params(log_returns, historical_volatility):
    def objective(params):
        kappa, theta, sigma, rho, v0 = params
        return np.sum(np.square(log_returns - np.mean(log_returns)))

    initial_guess = [historical_volatility, historical_volatility, 1.0, -0.6, historical_volatility]
    bounds = [(0.001, None), (0.001, None), (0.001, None), (-1, 1), (0.001, None)]
    result = minimize(objective, initial_guess, bounds=bounds, method='L-BFGS-B')
    return result.x

mbg_params = heston_model_params(df_mbg_v3['Log_Returns'].values, historical_volatility_mbg)
print(f"Estimated Heston parameters for MBG.DE: {mbg_params}")

msft_params = heston_model_params(df_msft_v3['Log_Returns'].values, historical_volatility_msft)
print(f"Estimated Heston parameters for MSFT: {msft_params}")

# Defining initial guesses for Bates, VG-CIR, and rBergomi models in the same way as we did with Heston model
# Bates model parameters estimation function
def bates_model_params(log_returns, historical_volatility):
    def objective(params):
        kappa, theta, sigma, rho, v0, lambda_, mu_j, sigma_j = params
        return np.sum(np.square(log_returns - np.mean(log_returns)))

    initial_guess = [historical_volatility, historical_volatility, 1.0, -0.6, historical_volatility, 0.2, -0.075, 0.15]
    bounds = [(0.001, None), (0.001, None), (0.001, None), (-1, 1), (0.001, None), (0.001, None), (-0.2, 0.2), (0.001, None)]
```

```
    result = minimize(objective, initial_guess, bounds=bounds, method='L-BFGS-B')
    return result.x

mbg_bates_params = bates_model_params(df_mbg_v3['Log_Returns'].values, historical_volatility_mbg)
print(f"Estimated Bates parameters for MBG.DE: {mbg_bates_params}")

msft_bates_params = bates_model_params(df_msft_v3['Log_Returns'].values, historical_volatility_msft)
print(f"Estimated Bates parameters for MSFT: {msft_bates_params}")

# VG-CIR model parameters estimation function
def vg_cir_model_params(log_returns, historical_volatility):
    def objective(params):
        kappa, theta, sigma, v0, sigma_vg, theta_vg, kappa_vg = params
        return np.sum(np.square(log_returns - np.mean(log_returns)))

    initial_guess = [historical_volatility, historical_volatility, 1.0, historical_volatility, 0.25, -0.15, 0.2]
    bounds = [(0.001, None), (0.001, None), (0.001, None), (0.001, None), (0.1, 0.3), (-0.2, -0.1), (0.1, 0.3)]
    result = minimize(objective, initial_guess, bounds=bounds, method='L-BFGS-B')
    return result.x

mbg_vg_cir_params = vg_cir_model_params(df_mbg_v3['Log_Returns'].values, historical_volatility_mbg)
print(f"Estimated VG-CIR parameters for MBG.DE: {mbg_vg_cir_params}")

msft_vg_cir_params = vg_cir_model_params(df_msft_v3['Log_Returns'].values, historical_volatility_msft)
print(f"Estimated VG-CIR parameters for MSFT: {msft_vg_cir_params}")

# rBergomi model parameters estimation function
def rbergomi_model_params(log_returns):
    def objective(params):
        eta, H, xi = params
        return np.sum(np.square(log_returns - np.mean(log_returns)))

    initial_guess = [1.0, 0.2, 0.04]
    bounds = [(0.001, None), (0.1, 0.3), (0.001, None)]
    result = minimize(objective, initial_guess, bounds=bounds, method='L-BFGS-B')
    return result.x

mbg_rbergomi_params = rbergomi_model_params(df_mbg_v3['Log_Returns'].values)
```

```
95 print(f"Estimated rBergomi parameters for MBG.DE: {mbg_rbergomi_params}")
96
97 msft_rbergomi_params = rbergomi_model_params(df_msft_v3['Log_Returns'].values)
98 print(f"Estimated rBergomi parameters for MSFT: {msft_rbergomi_params}")
99
100 # simulate_heston_paths function generates synthetic price paths for a given asset using
    the Heston model
101 # It takes as inputs the initial price (S0), model parameters (kappa, theta, sigma, rho,
    v0), the total time period (T), the number of time steps (N), and the number of
    simulated paths (M)
102 # It initializes arrays to store the simulated prices and volatilities by setting the
    initial values to S0 and v0
103 # Then it iterates over each time step, generating random variables z1 and z2 that are
    correlated via rho, and the volatility at each time step is updated using these random
    variables and the Heston model's stochastic differential equation
104 # The price is updated using an exponential function that incorporates the new
    volatility value, and the loop continues for all time steps, simulating M paths
105 # The function returns the array of simulated prices
106 def simulate_heston_paths(S0, params, T=1, N=252, M=10):
107     kappa, theta, sigma, rho, v0 = params
108     dt = T / N
109     prices = np.zeros((N + 1, M))
110     prices[0] = S0
111     v = np.zeros((N + 1, M))
112     v[0] = v0
113     for t in range(1, N + 1):
114         z1 = np.random.normal(size=M)
115         z2 = rho * z1 + np.sqrt(1 - rho**2) * np.random.normal(size=M)
116         v[t] = np.abs(v[t-1] + kappa * (theta - v[t-1]) * dt + sigma * np.sqrt(v[t-1] *
            dt) * z1)
117         prices[t] = prices[t-1] * np.exp((0 - 0.5 * v[t]) * dt + np.sqrt(v[t] * dt) *
            z2)
118     return prices
119
120 # Function to simulate Bates model paths
121 def simulate_bates_paths(S0, params, T=1, N=252, M=10):
122     kappa, theta, sigma, rho, v0, lambda_, mu_j, sigma_j = params
123     dt = T / N
124     prices = np.zeros((N + 1, M))
125     prices[0] = S0
126     v = np.zeros((N + 1, M))
127     v[0] = v0
128     for t in range(1, N + 1):
129         z1 = np.random.normal(size=M)
```

```
130         z2 = rho * z1 + np.sqrt(1 - rho**2) * np.random.normal(size=M)
131         jump = (np.random.poisson(lambda_ * dt, M) * (np.exp(mu_j + sigma_j *
            np.random.normal(size=M)) - 1))
132         v[t] = np.abs(v[t-1] + kappa * (theta - v[t-1]) * dt + sigma * np.sqrt(v[t-1] *
            dt) * z1)
133         prices[t] = prices[t-1] * np.exp((0 - 0.5 * v[t]) * dt + np.sqrt(v[t] * dt) *
            z2) * (1 + jump)
134     return prices
135
136 # Function to simulate VG-CIR model paths
137 def simulate_vg_cir_paths(S0, params, T=1, N=252, M=10):
138     kappa, theta, sigma, v0, sigma_vg, theta_vg, kappa_vg = params
139     dt = T / N
140     prices = np.zeros((N + 1, M))
141     prices[0] = S0
142     v = np.zeros((N + 1, M))
143     v[0] = v0
144     for t in range(1, N + 1):
145         z1 = np.random.normal(size=M)
146         z2 = np.random.normal(size=M)
147         v[t] = np.abs(v[t-1] + kappa * (theta - v[t-1]) * dt + sigma * np.sqrt(v[t-1] *
            dt) * z1)
148         prices[t] = prices[t-1] * np.exp((0 - 0.5 * v[t]) * dt + np.sqrt(v[t] * dt) *
            z2)
149     return prices
150
151 # Function to simulate rBergomi model paths
152 def simulate_rbergomi_paths(S0, params, T=1, N=252, M=10):
153     eta, H, xi = params
154     dt = T / N
155     prices = np.zeros((N + 1, M))
156     prices[0] = S0
157     v = np.zeros((N + 1, M))
158     v[0] = xi
159     for t in range(1, N + 1):
160         z1 = np.random.normal(size=M)
161         z2 = np.random.normal(size=M)
162         fBM = np.random.normal(size=M) * dt ** H
163         v[t] = np.abs(v[t-1] + eta * np.sqrt(v[t-1] * dt) * fBM)
164         prices[t] = prices[t-1] * np.exp((0 - 0.5 * v[t]) * dt + np.sqrt(v[t] * dt) *
            z2)
165     return prices
166
167 # Simulating paths using estimated parameters and combining all synthetic paths for each
    stock
```

```
S0_mbg = df_mbg_v3['Close'].iloc[-1]
S0_msft = df_msft_v3['Close'].iloc[-1]

synthetic_paths_mbg_heston = simulate_heston_paths(S0_mbg, mbg_params)
synthetic_paths_msft_heston = simulate_heston_paths(S0_msft, msft_params)

synthetic_paths_mbg_bates = simulate_bates_paths(S0_mbg, mbg_bates_params)
synthetic_paths_msft_bates = simulate_bates_paths(S0_msft, msft_bates_params)

synthetic_paths_mbg_vg_cir = simulate_vg_cir_paths(S0_mbg, mbg_vg_cir_params)
synthetic_paths_msft_vg_cir = simulate_vg_cir_paths(S0_msft, msft_vg_cir_params)

synthetic_paths_mbg_rbergomi = simulate_rbergomi_paths(S0_mbg, mbg_rbergomi_params)
synthetic_paths_msft_rbergomi = simulate_rbergomi_paths(S0_msft, msft_rbergomi_params)

synthetic_paths_combined_mbg = np.hstack((synthetic_paths_mbg_heston,
synthetic_paths_mbg_bates, synthetic_paths_mbg_vg_cir, synthetic_paths_mbg_rbergomi))
synthetic_paths_combined_msft = np.hstack((synthetic_paths_msft_heston,
synthetic_paths_msft_bates, synthetic_paths_msft_vg_cir, synthetic_paths_msft_rbergomi))

# Create training and testing sets
def create_train_test(paths, look_back=30):
    X_train, y_train, X_test, y_test = [], [], [], []
    for path in paths.T:
        for i in range(look_back, len(path) - 30):
            X_train.append(path[i-look_back:i])
            y_train.append(path[i])
        for i in range(len(path) - 30, len(path) - 1):
            X_test.append(path[i-look_back:i])
            y_test.append(path[i])
    return np.array(X_train), np.array(y_train), np.array(X_test), np.array(y_test)

look_back = 30
X_train_mbg_v3, y_train_mbg_v3, X_test_mbg_v3, y_test_mbg_v3 =
create_train_test(synthetic_paths_combined_mbg, look_back)
X_train_msft_v3, y_train_msft_v3, X_test_msft_v3, y_test_msft_v3 =
create_train_test(synthetic_paths_combined_msft, look_back)

# Reshaping input data to 3D for LSTM [samples, time steps, features]
X_train_mbg_v3 = X_train_mbg_v3.reshape(X_train_mbg_v3.shape[0], look_back, 1)
X_test_mbg_v3 = X_test_mbg_v3.reshape(X_test_mbg_v3.shape[0], look_back, 1)
X_train_msft_v3 = X_train_msft_v3.reshape(X_train_msft_v3.shape[0], look_back, 1)
X_test_msft_v3 = X_test_msft_v3.reshape(X_test_msft_v3.shape[0], look_back, 1)

```

```
# Define LSTM model
def create_lstm_model(neurons, input_shape, loss):
    model = Sequential()
    model.add(LSTM(neurons, return_sequences=True, input_shape=input_shape))
    model.add(LSTM(neurons))
    model.add(Dense(1))
    model.compile(optimizer=Adam(), loss=loss)
    return model

# Model parameters
epochs = 30
batch_size_mbg = 16
batch_size_msft = 32

# Training of LSTM models for both datasets
model_mbg_v3 = create_lstm_model(20, (look_back, 1), 'mean_absolute_error')
model_mbg_v3.fit(X_train_mbg_v3, y_train_mbg_v3, epochs=epochs,
batch_size=batch_size_mbg, validation_data=(X_test_mbg_v3, y_test_mbg_v3))

model_msft_v3 = create_lstm_model(200, (look_back, 1), 'mean_squared_error')
model_msft_v3.fit(X_train_msft_v3, y_train_msft_v3, epochs=epochs,
batch_size=batch_size_msft, validation_data=(X_test_msft_v3, y_test_msft_v3))

# evaluate_model function computes the test loss and generates predictions
# It takes a model and test data (X_test and y_test) as inputs and computes the test
loss using model.evaluate(X_test, y_test), which returns the loss value
# It then prints the test loss, and generates predictions using model.predict(X_test),
and returns these predictions.
def evaluate_model(model, X_test, y_test):
    test_loss = model.evaluate(X_test, y_test)
    print(f'test loss: {test_loss}')
    predictions = model.predict(X_test)
    return predictions

predictions_mbg_v3 = evaluate_model(model_mbg_v3, X_test_mbg_v3, y_test_mbg_v3)
predictions_msft_v3 = evaluate_model(model_msft_v3, X_test_msft_v3, y_test_msft_v3)

# Approximation of hedge ratios as the difference between consecutive predictions
def compute_hedge_ratios(predictions):
    hedge_ratios = np.diff(predictions, axis=0)
    return hedge_ratios

hedge_ratios_mbg_v3 = compute_hedge_ratios(predictions_mbg_v3)
hedge_ratios_msft_v3 = compute_hedge_ratios(predictions_msft_v3)
```

```
248
249 # Results for the last 30 days
250 print("predicted prices for MBG.DE (last 30 days):")
251 print(predictions_mbg_v3[-30:])
252 print("hedge ratios for MBG.DE (last 30 days):")
253 print(hedge_ratios_mbg_v3[-30:])
254
255 print("predicted prices for MSFT (last 30 days):")
256 print(predictions_msft_v3[-30:])
257 print("hedge ratios for MSFT (last 30 days):")
258 print(hedge_ratios_msft_v3[-30:])
259
260
261 # Plotting of predicted prices and actual prices for the last 30 days for MBG.DE
262 plt.figure(figsize=(14, 7))
263 plt.plot(predictions_mbg_v3[-30:], label='predicted prices')
264 plt.plot(y_test_mbg_v3[-30:], label='actual prices')
265 plt.title('Predicted vs Actual prices for MBG.DE (last 30 days)')
266 plt.xlabel('Time steps')
267 plt.ylabel('Price')
268 plt.legend()
269 plt.show()
270
271 # Plotting hedge ratios for the last 30 days for MBG.DE
272 plt.figure(figsize=(14, 7))
273 plt.plot(hedge_ratios_mbg_v3[-30:], label='hedge ratios')
274 plt.title('Hedge ratios for MBG.DE (last 30 days)')
275 plt.xlabel('Time steps')
276 plt.ylabel('Hedge ratio')
277 plt.legend()
278 plt.show()
279
280 # Plotting predicted prices and actual prices for the last 30 days for MSFT
281 plt.figure(figsize=(14, 7))
282 plt.plot(predictions_msft_v3[-30:], label='predicted prices')
283 plt.plot(y_test_msft_v3[-30:], label='actual prices')
284 plt.title('Predicted vs Actual prices for MSFT (last 30 days)')
285 plt.xlabel('Time steps')
286 plt.ylabel('Price')
287 plt.legend()
288 plt.show()
289
290 # Plotting hedge ratios for the last 30 days for MSFT
291 plt.figure(figsize=(14, 7))
```

```
plt.plot(hedge_ratios_msft_v3[-30:], label='hedge ratios')
plt.title('Hedge ratios for MSFT (last 30 days)')
plt.xlabel('Time steps')
plt.ylabel('Hedge ratio')
plt.legend()
plt.show()
```

F4. Additional inputs

```
# Additional file paths (historical data for interest rates and performance index)
fedfunds_path = os.path.join(dir, 'Data', 'D2. Additional inputs', 'Macroeconomic indicators (interest rates)', 'Fed rates')
ecb_rates_path = os.path.join(dir, 'Data', 'D2. Additional inputs', 'Macroeconomic indicators (interest rates)', 'ECB rates')
spx_path = os.path.join(dir, 'Data', 'D2. Additional inputs', 'Market indicators (market indexes)', '^SPX')
daxi_path = os.path.join(dir, 'Data', 'D2. Additional inputs', 'Market indicators (market indexes)', '^GDAXI')

# Loading data
df_mbg_v4 = pd.read_csv(mbg_path)
df_msft_v4 = pd.read_csv(msft_path)
df_fedfunds = pd.read_csv(fedfunds_path)
df_ecb_rates = pd.read_csv(ecb_rates_path)
df_spx = pd.read_csv(spx_path)
df_daxi = pd.read_csv(daxi_path)

# Verifying columns in each DataFrame and renaming columns for consistency
print("Columns in df_fedfunds:", df_fedfunds.columns)
print("Columns in df_ecb_rates:", df_ecb_rates.columns)

if 'DATE' in df_fedfunds.columns:
    df_fedfunds.rename(columns={'DATE': 'Date'}, inplace=True)
if 'FEDFUNDS' in df_fedfunds.columns:
    df_fedfunds.rename(columns={'FEDFUNDS': 'Fed_Rate'}, inplace=True)

if 'DATE' in df_ecb_rates.columns:
    df_ecb_rates.rename(columns={'DATE': 'Date'}, inplace=True)
if 'Main refinancing operations - Minimum bid rate/fixed rate (date of changes) - Level (FM.D.U2.EUR.4F.KR.MRR_RT.LEV)' in df_ecb_rates.columns:
    df_ecb_rates.rename(columns={'Main refinancing operations - Minimum bid rate/fixed rate (date of changes) - Level (FM.D.U2.EUR.4F.KR.MRR_RT.LEV)': 'ECB_Rate'}, inplace=True)
```

```
28
29 # Converting date columns to datetime format
30 df_mbg_v4['Date'] = pd.to_datetime(df_mbg_v4['Date'])
31 df_msft_v4['Date'] = pd.to_datetime(df_msft_v4['Date'])
32 df_fedfunds['Date'] = pd.to_datetime(df_fedfunds['Date'])
33 df_ecb_rates['Date'] = pd.to_datetime(df_ecb_rates['Date'])
34 df_spx['Date'] = pd.to_datetime(df_spx['Date'])
35 df_daxi['Date'] = pd.to_datetime(df_daxi['Date'])
36
37 # Handling missing values in GDAXI
38 df_daxi.fillna(method='ffill', inplace=True)
39
40 # Normalizing the relevant columns in each DataFrame and merging DataFrames on 'Date'
41 scaler = MinMaxScaler()
42 df_mbg_v4[['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']] =
   scaler.fit_transform(df_mbg_v4[['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']])
43 df_msft_v4[['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']] =
   scaler.fit_transform(df_msft_v4[['Open', 'High', 'Low', 'Close', 'Adj Close',
   'Volume']])
44 df_fedfunds[['Fed_Rate']] = scaler.fit_transform(df_fedfunds[['Fed_Rate']])
45 df_ecb_rates[['ECB_Rate']] = scaler.fit_transform(df_ecb_rates[['ECB_Rate']])
46 df_spx[['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']] =
   scaler.fit_transform(df_spx[['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']])
47 df_daxi[['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']] =
   scaler.fit_transform(df_daxi[['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']])
48
49 df_mbg_merged = pd.merge(df_mbg_v4, df_daxi[['Date', 'Close']], on='Date', how='left',
   suffixes=('', '_DAX_Close'))
50 df_mbg_merged = pd.merge(df_mbg_merged, df_ecb_rates[['Date', 'ECB_Rate']], on='Date',
   how='left')
51 df_msft_merged = pd.merge(df_msft_v4, df_spx[['Date', 'Close']], on='Date', how='left',
   suffixes=('', '_SPX_Close'))
52 df_msft_merged = pd.merge(df_msft_merged, df_fedfunds[['Date', 'Fed_Rate']], on='Date',
   how='left')
53
54 # calculate_rsi function computes the Relative Strength Index (RSI) for a given
   dataframe
55 # It takes a dataframe (df), a column name (column), and a period (period) as inputs,
   and then calculates the difference (delta) between consecutive values in the specified
   column
56 # It then identifies gains (gain) as the mean of positive differences over the specified
   period, and losses (loss) as the mean of negative differences (converted to positive)
   over the same period
57 # The relative strength (rs) is calculated as the ratio of average gain to average loss
```

```
# The RSI is then computed using the formula 100 - (100 / (1 + rs))
def calculate_rsi(df, column='Close', period=14):
    delta = df[column].diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=period).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=period).mean()
    rs = gain / loss
    rsi = 100 - (100 / (1 + rs))
    return rsi

df_mbg_merged['RSI'] = calculate_rsi(df_mbg_merged, column='Close')
df_msft_merged['RSI'] = calculate_rsi(df_msft_merged, column='Close')

# Defining columns to normalize and normalizing DataFrames
columns_to_normalize_mbg = ['Close', 'RSI', 'ECB_Rate', 'Close_DAX_Close']
columns_to_normalize_msft = ['Close', 'RSI', 'Fed_Rate', 'Close_SPX_Close']

scaler = MinMaxScaler()
df_mbg_merged[columns_to_normalize_mbg] =
scaler.fit_transform(df_mbg_merged[columns_to_normalize_mbg])
df_msft_merged[columns_to_normalize_msft] =
scaler.fit_transform(df_msft_merged[columns_to_normalize_msft])

# Filling NaN values with column means
df_mbg_merged[columns_to_normalize_mbg] =
df_mbg_merged[columns_to_normalize_mbg].fillna(df_mbg_merged[columns_to_normalize_mbg].mean())
df_msft_merged[columns_to_normalize_msft] =
df_msft_merged[columns_to_normalize_msft].fillna(df_msft_merged[columns_to_normalize_msft].mean(

# create_sequences function generates sequences and corresponding targets from time
series data
# It takes as inputs the data (data) and the number of time steps for each sequence
# The function iterates over the data to extract sequences of the specified length and
their corresponding targets
# For each position i, it creates a sequence from data[i:i + time_steps] and a target as
the value at data[i + time_steps].
# These sequences and targets are then collected into lists and converted to numpy
arrays before being returned
def create_sequences(data, time_steps=30):
    sequences = []
    targets = []
    for i in range(len(data) - time_steps):
        sequence = data[i:i + time_steps]
        target = data[i + time_steps]
        sequences.append(sequence)
```

```
        targets.append(target)
    return np.array(sequences), np.array(targets)

# prepare_data function formats the data for an LSTM model
# It takes as inputs a dataframe (df), the names of the feature columns (feature_columns), the name of the target column (target_column), and the number of time steps for each sequence as inputs
# The function extracts the feature data and target data from the dataframe as numpy arrays, and then calls the create_sequences function to generate sequences (X) and their corresponding targets (y) from the feature data
# It returns these sequences and targets
def prepare_data(df, feature_columns, target_column, time_steps=30):
    data = df[feature_columns].values
    target = df[target_column].values
    X, y = create_sequences(data, time_steps)
    return X, y

# List of configurations for inputs (cf report part 5.3)
configurations_mbg = [
    ['Close'],
    ['Close', 'RSI'],
    ['Close', 'ECB_Rate'],
    ['Close', 'Close_DAX_Close'],
    ['Close', 'RSI', 'ECB_Rate'],
    ['Close', 'RSI', 'Close_DAX_Close'],
    ['Close', 'Close_DAX_Close', 'ECB_Rate'],
    ['Close', 'RSI', 'ECB_Rate', 'Close_DAX_Close']
]

configurations_msft = [
    ['Close'],
    ['Close', 'RSI'],
    ['Close', 'Fed_Rate'],
    ['Close', 'Close_SPX_Close'],
    ['Close', 'RSI', 'Fed_Rate'],
    ['Close', 'RSI', 'Close_SPX_Close'],
    ['Close', 'Close_SPX_Close', 'Fed_Rate'],
    ['Close', 'RSI', 'Fed_Rate', 'Close_SPX_Close']
]

# build_model function constructs an LSTM neural network model by taking as inputs the shape of the input data (input_shape) and the number of neurons in each LSTM layer (neurons)
# It initializes a Sequential model and adds two LSTM layers, each with the specified number of neurons and ReLU activation
```

```
# The first LSTM layer is configured to return sequences, while the second one is not, and a dense layer with a single neuron is added as the output layer
# The model is then compiled using the Adam optimizer and mean squared error loss function, and the compiled model is returned
def build_model(input_shape, neurons):
    model = Sequential()
    model.add(LSTM(neurons, activation='relu', return_sequences=True, input_shape=input_shape))
    model.add(LSTM(neurons, activation='relu'))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# train_and_evaluate function trains and evaluates an LSTM model using different configurations of feature columns from the input dataframe
# It takes as inputs the dataframe (df), a list of feature configurations (configurations), the name of the target column (target_column), the number of neurons in the LSTM layers (neurons), the number of training epochs (epochs), and the batch size (batch_size)
# For each configuration in configurations, it prints the configuration being used, and then prepares the data by calling prepare_data, which formats the feature and target data into sequences and their corresponding targets
# It also prints the shapes of the input features (X) and targets (y)
# Next, the function builds an LSTM model with the specified input shape and number of neurons by calling build_model, and trains the model using model.fit with the prepared input features and the first column of the target data, for the specified number of epochs and batch size
# After training, it generates predictions using model.predict and prints the shape of the predictions, and then calculates the deltas by comparing the predictions to the first column of the target data
# The deltas for the last 30 days are stored in the results dictionary, with the configuration tuple as the key
# The function returns the results dictionary containing the deltas for the last 30 days for each configuration
def train_and_evaluate(df, configurations, target_column, neurons, epochs, batch_size):
    results = {}
    time_steps = 30

    for config in configurations:
        print(f"\nTraining with configuration: {config}")
        X, y = prepare_data(df, config, target_column, time_steps)
        print(f"Shape of X: {X.shape}")
        print(f"Shape of y: {y.shape}")

```

```
        model = build_model((X.shape[1], X.shape[2]), neurons)
        model.fit(X, y[:, 0], epochs=epochs, batch_size=batch_size, verbose=1)  # Fit the model with the first column of y

        predictions = model.predict(X)
        print(f"Shape of predictions: {predictions.shape}")

        deltas = predictions - y[:, 0].reshape(-1, 1)  # Compare predictions to the first column of y
        results[tuple(config)] = deltas[-30:]  # Last 30 days of deltas
    return results

mbg_deltas_v4 = train_and_evaluate(df_mbg_merged, configurations_mbg, 'Close', 20, 30, 16)
msft_deltas_v4 = train_and_evaluate(df_msft_merged, configurations_msft, 'Close', 200, 30, 32)

# Plotting results
def plot_results(deltas, title):
    plt.figure(figsize=(14, 7))
    for config, delta in deltas.items():
        plt.plot(delta, label=str(config))
    plt.title(title)
    plt.xlabel("Days")
    plt.ylabel("Hedge ratios")
    plt.legend()
    plt.show()

plot_results(mbg_deltas_v4, "MBG hedge ratios comparison (last 30 days)")
plot_results(msft_deltas_v4, "MSFT hedge ratios comparison (last 30 days)")
```

F5. Market frictions

```
df_mbg_v5 = pd.read_csv(mbg_path, delimiter=",")
df_msft_v5 = pd.read_csv(msft_path, delimiter=",")

# Preprocessing data
def preprocess_data(df):
    df['Return'] = df['Adj Close'].pct_change()
    df['Volatility'] = df['Return'].rolling(window=30).std() * np.sqrt(252)
    df.dropna(inplace=True)
    return df
```

```
10
11 df_mbg_tc = preprocess_data(df_mbg_v5)
12 df_msft_tc = preprocess_data(df_msft_v5)
13
14 # Preparing data for use in the LSTM model by creating sequences of a specified length
   (look_back) and their corresponding target values
15 # Initializing empty lists to store the sequences (dataX) and targets (dataY), and
   looping through the dataframe to extract sequences and targets, appended to the lists
16 # Converting the lists to numpy arrays and returning them
17 # The look_back period is set to 30, and the datasets for MBG and MSFT are created using
   the 'Adj Close' column from the respective dataframes
18 def create_dataset(df, look_back=30):
19     dataX, dataY = [], []
20     for i in range(len(df) - look_back):
21         dataX.append(df.iloc[i:(i + look_back), :].values)
22         dataY.append(df['Adj Close'].iloc[i + look_back])
23     return np.array(dataX), np.array(dataY)
24
25 look_back = 30
26 X_mbg_v5, y_mbg_v5 = create_dataset(df_mbg_tc[['Adj Close']], look_back)
27 X_msft_v5, y_msft_v5 = create_dataset(df_msft_tc[['Adj Close']], look_back)
28
29 # Defining LSTM model
30 def create_model():
31     model = Sequential()
32     model.add(LSTM(50, input_shape=(look_back, 1)))
33     model.add(Dense(1))
34     model.compile(loss='mae', optimizer='adam')
35     return model
36
37 # Training and prediction for the last 30 days
38 model_mbg_v5 = create_model()
39 model_mbg_v5.fit(X_mbg_v5, y_mbg_v5, epochs=5, batch_size=32, verbose=1)
40
41 model_msft_v5 = create_model()
42 model_msft_v5.fit(X_msft_v5, y_msft_v5, epochs=5, batch_size=32, verbose=1)
43
44 pred_mbg_v5 = model_mbg_v5.predict(X_mbg[-30:])
45 pred_msft_v5 = model_msft_v5.predict(X_msft[-30:])
46
47 # Normalizing NN predictions
48 scaler_mbg = MinMaxScaler(feature_range=(0, 1))
49 scaler_msft = MinMaxScaler(feature_range=(0, 1))
50 scaled_pred_mbg = scaler_mbg.fit_transform(pred_mbg_v5)
```

```
scaled_pred_msft = scaler_msft.fit_transform(pred_msft_v5)

# Calculating BS deltas
def black_scholes_delta(S, K, T, r, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    return norm.cdf(d1)

K = df_mbg_v5['Adj Close'].iloc[-30:].values[0]
T = 30 / 252
r = 0.01

df_mbg_v5.loc[df_mbg_tc.index[-30:], 'BS_Delta'] = black_scholes_delta(df_mbg_tc['Adj Close'].iloc[-30:].values, K, T, r, df_mbg_tc['Volatility'].iloc[-30:].values)
df_msft_v5.loc[df_msft_tc.index[-30:], 'BS_Delta'] = black_scholes_delta(df_msft_tc['Adj Close'].iloc[-30:].values, K, T, r, df_msft_tc['Volatility'].iloc[-30:].values)

# Defining transaction costs rate (cf report part 5.4)
k = 0.001

# Calculating the transaction costs for both MBG and MSFT based on the absolute changes in NN_Delta and the adjusted close price (Adj Close)
# Initializing the portfolio value (Portfolio_Value) for the last 30 days using the adjusted close price 30 days before the end
# For each of the last 30 days, the portfolio value is updated by calculating the new portfolio value based on the previous value, the current and previous NN_Delta, the adjusted close price, and the transaction costs
# The exponential term np.exp(r * (1 / 252)) accounts for the risk-free rate over a single trading day, with the loop iterating to update the portfolio value for each day, considering transaction costs
df_mbg_tc['NN_Transaction_Cost'] = k * np.abs(df_mbg_tc['NN_Delta'].diff()) * df_mbg_v5['Adj Close']
df_msft_tc['NN_Transaction_Cost'] = k * np.abs(df_msft_tc['NN_Delta'].diff()) * df_msft_v5['Adj Close']

V0_mbg = df_mbg_tc['Adj Close'].iloc[-30]
V0_msft = df_msft_tc['Adj Close'].iloc[-30]

df_mbg_tc['Portfolio_Value'] = V0_mbg
df_msft_tc['Portfolio_Value'] = V0_msft

for t in range(1, 30):
    df_mbg.loc[df_mbg.index[-30 + t], 'Portfolio_Value'] =
    df_mbg_tc['NN_Delta'].iloc[-30 + t] * df_mbg_tc['Adj Close'].iloc[-30 + t] + \
                                          (df_mbg_tc['Portfolio_Value'].iloc[-30 +
                                          t - 1] - df_mbg_tc['NN_Delta'].iloc[-30 +
                                          t - 1] * df_mbg_tc['Adj Close'].iloc[-30
                                          + t - 1]) * np.exp(r * (1 / 252)) - \
```

```
                                                df_mbg_tc['NN_Transaction_Cost'].iloc[-30
                                                + t]

    df_msft.loc[df_msft.index[-30 + t], 'Portfolio_Value'] =
    df_msft_tc['NN_Delta'].iloc[-30 + t] * df_msft_tc['Adj Close'].iloc[-30 + t] + \
                                                (df_msft_tc['Portfolio_Value'].iloc[-30 +
                                                t - 1] - df_msft_tc['NN_Delta'].iloc[-30
                                                + t - 1] * df_msft_tc['Adj
                                                Close'].iloc[-30 + t - 1]) * np.exp(r *
                                                (1 / 252)) - \

                                                df_msft_tc['NN_Transaction_Cost'].iloc[-30
                                                + t]


import matplotlib.ticker as mtick

# Plotting results
fig, axs = plt.subplots(2, 2, figsize=(15, 10))

# Plotting hedge ratios with shaded areas
axs[0, 0].plot(range(1, 31), df_mbg_tc['NN_Delta'].iloc[-30:], label='NN hedge ratio')
axs[0, 0].plot(range(1, 31), df_mbg_tc['BS_Delta'].iloc[-30:], label='BS hedge ratio')
axs[0, 0].fill_between(range(1, 31), df_mbg_tc['NN_Delta'].iloc[-30:], alpha=0.2)
axs[0, 0].fill_between(range(1, 31), df_mbg_tc['BS_Delta'].iloc[-30:], alpha=0.2)
axs[0, 0].set_title('Hedge ratios (NN vs BS) - MBG')
axs[0, 0].set_xlabel('Time')
axs[0, 0].set_ylabel('Hedge ratio')
axs[0, 0].legend()

axs[0, 1].plot(range(1, 31), df_msft_tc['NN_Delta'].iloc[-30:], label='NN hedge ratio')
axs[0, 1].plot(range(1, 31), df_msft_tc['BS_Delta'].iloc[-30:], label='BS hedge ratio')
axs[0, 1].fill_between(range(1, 31), df_msft_tc['NN_Delta'].iloc[-30:], alpha=0.2)
axs[0, 1].fill_between(range(1, 31), df_msft_tc['BS_Delta'].iloc[-30:], alpha=0.2)
axs[0, 1].set_title('Hedge ratios (NN vs BS) - MSFT')
axs[0, 1].set_xlabel('Time')
axs[0, 1].set_ylabel('Hedge ratio')
axs[0, 1].legend()

plt.tight_layout()
plt.show()


# calculate_pnl function computes the profit and loss (PnL) for a given dataframe based
on the specified hedge column.
```

```
# It initializes a list pnl with zero for the first row, and for each subsequent row, it calculates the PnL as the difference between the current and previous close prices, adjusted by the previous value of the specified hedge column
# It returns the cumulative sum of the PnL values
def calculate_pnl(df, hedge_column):
    pnl = [0]  # Initialize with zero for the first row
    for i in range(1, len(df)):
        pnl.append(df['Close'].iloc[i] - df['Close'].iloc[i-1] *
        df[hedge_column].iloc[i-1])
    return np.cumsum(pnl)

# Selecting the last 30 days data for MBG and MSFT, calculating PnL and add it to the dataframe
df_mbg_last_30_tc = df_mbg_tc.iloc[-30:].copy()
df_msft_last_30_tc = df_msft_tc.iloc[-30:].copy()

df_mbg_last_30_tc['NN_PnL'] = calculate_pnl(df_mbg_last_30_tc, 'NN_Delta')
df_mbg_last_30_tc['BS_PnL'] = calculate_pnl(df_mbg_last_30_tc, 'BS_Delta')

df_msft_last_30_tc['NN_PnL'] = calculate_pnl(df_msft_last_30_tc, 'NN_Delta')
df_msft_last_30_tc['BS_PnL'] = calculate_pnl(df_msft_last_30_tc, 'BS_Delta')

# Print the last few rows to check
print("Final MBG data with BS calculations and PnL:")
print(df_mbg_last_30_tc[['Date', 'Close', 'BS_Delta', 'NN_Delta', 'NN_PnL', 'BS_PnL']])
print("Final MSFT data with BS calculations and PnL:")
print(df_msft_last_30_tc[['Date', 'Close', 'BS_Delta', 'NN_Delta', 'NN_PnL', 'BS_PnL']])

# Visualizing PnL
plt.figure(figsize=(14, 7))

plt.subplot(2, 1, 1)
plt.plot(range(1, 31), df_mbg_last_30_tc['NN_PnL'], label='NN PnL', color='blue')
plt.fill_between(range(1, 31), df_mbg_last_30_tc['NN_PnL'], color='blue', alpha=0.1)
plt.plot(range(1, 31), df_mbg_last_30_tc['BS_PnL'], label='BS PnL', color='red')
plt.fill_between(range(1, 31), df_mbg_last_30_tc['BS_PnL'], color='red', alpha=0.1)
plt.title('MBG PnL comparison')
plt.xlabel('Time')
plt.ylabel('PnL')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(range(1, 31), df_msft_last_30_tc['NN_PnL'], label='NN PnL', color='blue')
plt.fill_between(range(1, 31), df_msft_last_30_tc['NN_PnL'], color='blue', alpha=0.1)
```

```
plt.plot(range(1, 31), df_msft_last_30_tc['BS_PnL'], label='BS PnL', color='red')
plt.fill_between(range(1, 31), df_msft_last_30_tc['BS_PnL'], color='red', alpha=0.1)
plt.title('MSFT PnL comparison')
plt.xlabel('Time')
plt.ylabel('PnL')
plt.legend()

plt.tight_layout()
plt.show()

# Calculating the stats
def calculate_stats(df, column_name):
    return {
        'Mean': np.mean(df[column_name]),
        'Median': np.median(df[column_name]),
        'Std Dev': np.std(df[column_name]),
        'Min': np.min(df[column_name]),
        'Max': np.max(df[column_name])
    }

# Collecting stats in tables
mbg_stats = pd.DataFrame({
    'Metric': ['Mean', 'Median', 'Std Dev', 'Min', 'Max'],
    'Without TC': [calculate_stats(df_mbg_last_30, 'NN_Delta')[key] for key in ['Mean',
    'Median', 'Std Dev', 'Min', 'Max']],
    'With TC': [calculate_stats(df_mbg_last_30_tc, 'NN_Delta')[key] for key in ['Mean',
    'Median', 'Std Dev', 'Min', 'Max']]
})

msft_stats = pd.DataFrame({
    'Metric': ['Mean', 'Median', 'Std Dev', 'Min', 'Max'],
    'Without TC': [calculate_stats(df_msft_last_30, 'NN_Delta')[key] for key in ['Mean',
    'Median', 'Std Dev', 'Min', 'Max']],
    'With TC': [calculate_stats(df_msft_last_30_tc, 'NN_Delta')[key] for key in ['Mean',
    'Median', 'Std Dev', 'Min', 'Max']]
})

# Plotting the comparison of hedge ratios for the last 30 days
plt.figure(figsize=(14, 7))

plt.subplot(2, 1, 1)
plt.plot(range(1, 31), df_mbg_last_30['NN_Delta'], label='Without TC', color='blue')
plt.plot(range(1, 31), df_mbg_last_30_tc['NN_Delta'], label='With TC', color='red')
plt.fill_between(range(1, 31), df_mbg_last_30['NN_Delta'], color='blue', alpha=0.1)
```

```
plt.fill_between(range(1, 31), df_mbg_last_30_tc['NN_Delta'], color='red', alpha=0.1)
plt.title('MBG hedge ratio comparison (last 30 days)')
plt.xlabel('Time')
plt.ylabel('Hedge ratio')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(range(1, 31), df_msft_last_30['NN_Delta'], label='Without TC', color='blue')
plt.plot(range(1, 31), df_msft_last_30_tc['NN_Delta'], label='With TC', color='red')
plt.fill_between(range(1, 31), df_msft_last_30['NN_Delta'], color='blue', alpha=0.1)
plt.fill_between(range(1, 31), df_msft_last_30_tc['NN_Delta'], color='red', alpha=0.1)
plt.title('MSFT hedge ratio comparison (last 30 days)')
plt.xlabel('Time')
plt.ylabel('Hedge ratio')
plt.legend()

plt.tight_layout()
plt.show()

# Printing the stats for verification
print("MBG hedge ratio stats:")
print(mbg_stats)
print("\nMSFT hedge ratio stats:")
print(msft_stats)
```

F6. Computational justifications for the choices in the neural network architecture and features

Number of neurons in the LSTM NN

```
# We define a range of neurons to test for the LSTM models and initialize a dictionary to store the results
# For each number of neurons in the range, we create and train an LSTM model for both MBG and MSFT using the specified number of neurons
# The training is done for 20 epochs with a batch size of 32, and the verbose output is turned off
# After training, we generate predictions for the training data and evaluates the model using mean squared error (MSE) and mean absolute error (MAE) metrics.
# The results, including the number of neurons, MSE, and MAE for both MBG and MSFT, are stored in the dictionary
neuron_range = [10, 20, 50, 100, 200]
results_v6 = {'neurons': [], 'mse_mbg': [], 'mae_mbg': [], 'mse_msft': [], 'mae_msft': []}
```

```
8
9  for num_neurons in neuron_range:
10     # Creating and training the model
11     model_mbg_v6 = create_lstm_model((n_steps, 1), num_neurons)
12     model_mbg_v6.fit(X_mbg_v6, y_mbg_v6, epochs=20, batch_size=32, verbose=0)
13
14     model_msft_v6 = create_lstm_model((n_steps, 1), num_neurons)
15     model_msft_v6.fit(X_msft_v6, y_msft_v6, epochs=20, batch_size=32, verbose=0)
16
17     # Predicting and evaluating the model
18     predictions_mbg_v6 = model_mbg_v6.predict(X_mbg)
19     mse_mbg = mean_squared_error(y_mbg_v6, predictions_mbg_v6)
20     mae_mbg = mean_absolute_error(y_mbg_v6, predictions_mbg_v6)
21
22     predictions_msft_v6 = model_msft_v6.predict(X_msft_v6)
23     mse_msft = mean_squared_error(y_msft_v6, predictions_msft_v6)
24     mae_msft = mean_absolute_error(y_msft_v6, predictions_msft_v6)
25
26     # Store the results
27     results_v6['neurons'].append(num_neurons)
28     results_v6['mse_mbg'].append(mse_mbg)
29     results_v6['mae_mbg'].append(mae_mbg)
30     results_v6['mse_msft'].append(mse_msft)
31     results_v6['mae_msft'].append(mae_msft)
32
33 # Converting results to DataFrame
34 results_df_v6 = pd.DataFrame(results)
35
36 # Plot the results
37 plt.figure(figsize=(14, 7))
38
39 plt.subplot(2, 1, 1)
40 plt.plot(results_df_v6['neurons'], results_df_v6['mse_mbg'], marker='o', label='MSE
   MBG')
41 plt.plot(results_df_v6['neurons'], results_df_v6['mae_mbg'], marker='o', label='MAE
   MBG')
42 plt.xlabel('Number of neurons')
43 plt.ylabel('Error')
44 plt.title('Error vs number of neurons (MBG)')
45 plt.legend()
46
47 plt.subplot(2, 1, 2)
48 plt.plot(results_df_v6['neurons'], results_df_v6['mse_msft'], marker='o', label='MSE
   MSFT')
```

```
plt.plot(results_df_v6['neurons'], results_df_v6['mae_msft'], marker='o', label='MAE MSFT')
plt.xlabel('Number of neurons')
plt.ylabel('Error')
plt.title('Error vs number of neurons (MSFT)')
plt.legend()

plt.tight_layout()
plt.show()

# Displaying the results
print(results_df_v6)
```

Loss function

```
# We train and evaluate the LSTM models for MBG and MSFT using two different loss functions: Mean Squared Error (MSE) and Mean Absolute Error (MAE)
# We initialize a dictionary to store the results and for each loss function, we create and compile LSTM models for both MBG and MSFT, as well as train the models using the specified loss function, and generate predictions for the training data
# The evaluation metrics (Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE)) are calculated and stored in the results dictionary
results = {}
for loss_function in ['mse', 'mae']:
    model_mbg_v7 = create_lstm_model((n_steps, 1), loss_function)
    model_msft_v7 = create_lstm_model((n_steps, 1), loss_function)

    model_mbg_v7.fit(X_mbg_v7, y_mbg_v7, epochs=20, batch_size=32, verbose=0)
    model_msft_v7.fit(X_msft_v7, y_msft_v7, epochs=20, batch_size=32, verbose=0)

    y_mbg_pred_v7 = model_mbg_v7.predict(X_mbg_v7)
    y_msft_pred_v7 = model_msft_v7.predict(X_msft_v7)

    results_v7[loss_function] = {
        'MBG RMSE': np.sqrt(mean_squared_error(y_mbg_v7, y_mbg_pred_v7)),
        'MBG MAE': mean_absolute_error(y_mbg_v7, y_mbg_pred_v7),
        'MSFT RMSE': np.sqrt(mean_squared_error(y_msft_v7, y_msft_pred_v7)),
        'MSFT MAE': mean_absolute_error(y_msft_v7, y_msft_pred_v7)
    }

# Displaying the results
results_df_v7 = pd.DataFrame(results_v7)
print(results_df_v7)
```

Batch size and number of epochs

```
# We define lists of batch sizes and epochs to test and we initialize an empty list to store the results
# We iterate over each combination of batch size and number of epochs, creating and compiling LSTM models for MBG and MSFT
# The start time is recorded before training the models with the current batch size and number of epochs
# The training time is calculated after training, and predictions for the training data are generated for both MBG and MSFT models
# The results for the current combination, including the batch size, number of epochs, RMSE for MBG, RMSE for MSFT, and training time, are stored in the results list
# The results list is then converted to a DataFrame and printed for an easier analysis

batch_sizes = [16, 32, 64]
epochs_list = [20, 30, 50]
initial_results = []

for batch_size in batch_sizes:
    for epochs in epochs_list:
        model_mbg_v8 = create_lstm_model((n_steps, 1))
        model_msft_v8 = create_lstm_model((n_steps, 1))

        start_time = time.time()
        model_mbg_v8.fit(X_mbg_v8, y_mbg_v8, epochs=epochs, batch_size=batch_size, verbose=0)
        model_msft_v8.fit(X_msft_v8, y_msft_v8, epochs=epochs, batch_size=batch_size, verbose=0)
        training_time = time.time() - start_time

        y_mbg_pred_v8 = model_mbg_v8.predict(X_mbg_v8)
        y_msft_pred_v8 = model_msft_v8.predict(X_msft_v8)

        initial_results_v8.append({
            'Batch Size': batch_size,
            'Epochs': epochs,
            'MBG RMSE': np.sqrt(mean_squared_error(y_mbg_v8, y_mbg_pred_v8)),
            'MSFT RMSE': np.sqrt(mean_squared_error(y_msft_v8, y_msft_pred_v8)),
            'Training Time (s)': training_time
        })

initial_results_df_v8 = pd.DataFrame(initial_results_v8)
print("Initial manual exploration results:")
print(initial_results_df_v8)

```

```
import seaborn as sns

# Plotting RMSE for different batch sizes and epochs
plt.figure(figsize=(12, 6))

for batch_size in batch_sizes:
    subset = initial_results_df_v8[initial_results_df_v8['Batch size'] == batch_size]
    plt.plot(subset['Epochs'], subset['MBG RMSE'], label=f'MBG RMSE (Batch
    size={batch_size})')
    plt.plot(subset['Epochs'], subset['MSFT RMSE'], label=f'MSFT RMSE (Batch
    size={batch_size})')

plt.xlabel('Epochs')
plt.ylabel('RMSE')
plt.title('RMSE for different batch sizes and epochs')
plt.legend()
plt.show()
```

G. Datasets sources

In this section, we list the various sources from which the datasets used in our algorithm are derived.

- MBG.DE and MSFT historical stock prices : Yahoo! Finance ; Yahoo! Finance
- Historical Federal Funds Effective Rate (FEDFUND) : Macrotrends
- Euro Main refinancing operations fixed rate : ECB Data Portal
- S&P500 historical data index : MarketWatch
- DAX historical data index : Yahoo! Finance