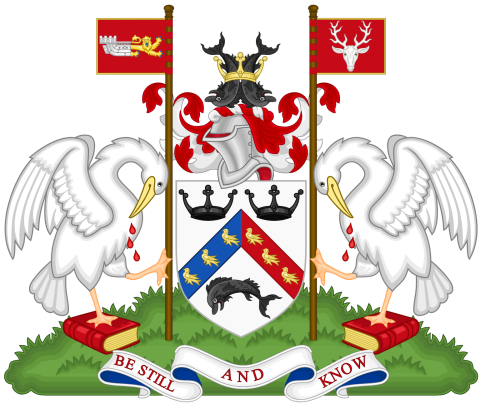


Option Pricing Using Multilayer Perceptron, Long Short Term Memory Networks, and Gated Recurrent Unit Neural Networks

Author
Kiana Asadzadeh
Cand. No: 277215

Supervisor
Antony Lewis

Msc Data Science
Master Thesis

University of Sussex
May 2024

University of Sussex

Abstract

This dissertation aims to enhance the accuracy and efficiency of stock options pricing by integrating traditional financial models with advanced machine learning techniques. Leveraging the Black-Scholes model and Geometric Brownian Motion (GBM) simulation, combined with Multi-Layer Perceptron (MLP), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU) neural networks, this research seeks to build and enhance neural network architectures, optimizing these models for better performance, and comparing the effectiveness of their outputs. The analysis focuses on two main targets: the call option price over the stock price and the logarithm of call option price over the stock price, with and without incorporating the GARCH feature. The study demonstrates that while MLP and GRU models with a mean fraction of -0.003 and 0.002 perform well, the LSTM model with a mean fraction of -0.0002 emerges as the nominated model for options pricing and an ideal choice for dynamic financial modeling. These enhanced models were particularly effective in capturing the nonlinear relationships in financial markets, leading to a more reliable estimation of call option prices. This research contributes to the development of more dynamic financial models, suggesting that integrating modern machine learning approaches can substantially improve financial decision-making and risk management strategies.

Contents

List of Tables

List of Figures

1 Introduction

The accurate prediction of stock options prices is a critical task in quantitative finance, influencing investment strategies, risk management, and financial decision-making. Since the introduction of the Black-Scholes (B-S) model in 1973 by economists Black, Scholes, and Merton, the field has seen significant advancements [4]. The B-S model, which focuses on the volatility of the underlying asset as a key determinant of an option's value, has become a cornerstone in financial modeling. The Black Scholes model (B-S) and other traditional option pricing methods make assumptions about asset and market dynamics, including constant volatility and log-normal returns. These assumptions may not be applicable in real-world markets, resulting in erroneous pricing and risk management [25].

Deep learning (DL) and machine learning (ML) are promising study topics for modeling and pricing financial derivatives, including options. These models, such as artificial neural networks (ANN) and recurrent neural networks (RNN), use neural networks to learn complicated correlations between inputs and outputs. RNNs and ANNs can identify nonlinear correlations between market variables that impact option prices. Training a neural network on historical market data allows them to generalize to new conditions, resulting in more accurate predictions. Therefore, with the advent of machine learning (ML) and Deep learning (DL) and their ability to handle complex, non-linear relationships in large datasets, there is potential to further refine these models. Neural networks can learn option pricing models from markets and train to resemble traders who specialize in a specific stock or index [30].

Advancements in mathematical analysis, computing hardware and software, and big data have made it possible to create commoditized robots capable of performing investment management, financial analysis, and trading tasks. I provide a brief overview of how artificial intelligence and deep learning can impact finance and summarize a paradigm for applying machine learning to option pricing and provide an overview of neural networks and hyperparameters for improving model accuracy.

Traditional option pricing models such as the B-S, assume that the underlying asset price follows a log-normal distribution and has constant volatility over time. However, in reality, the underlying asset price is influenced by a complex set of factors such as market sentiment, news events, and macroeconomic conditions, which can lead to its non-normal distribution, time-normal distribution, and time-varying volatility. RNNs and ANNs models can better represent the complicated nonlinear interactions of the underlying asset's risk and uncertainty with the option price and are capable of learning complex relationships between input and output variables, have shown promise in capturing these complex dynamics and enhancing the accuracy of option pricing. These models can use unstructured data, including news articles and social media sentiment, to better understand asset risk and uncertainty [38]. However, These models require enormous volumes of high-quality data, rigorous validation, and cautious interpretation. In addition, their performance may rely on the specific problem and data characteristics.

This dissertation seeks to bridge traditional financial models with modern RNNs and ANNs techniques to enhance the accuracy and efficiency of predicting stock options prices. The research objectives are to simulate stock prices using GBM and option prices using the Black-Scholes model. Build and develop the architecture of Multi-Layer Perceptron (MLP), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU) neural networks, optimize these models for better performance, and compare their outputs with each other. The analysis focused on two main targets: $\frac{\text{Call Option}}{\text{Stock Price}}$ and $\log\left(\frac{\text{Call Option}}{\text{Stock Price}}\right)$, with and without incorporating the GARCH feature. By doing so, this study aims to contribute to the development of more reliable and dynamic financial models that can adapt to the complex nature of financial markets.

This research is structured to include a detailed methodology section, reviewing related work in the field, outlining the machine learning techniques employed, and providing an extensive analysis of the results.

2 Abbreviations

Table 1 lists the abbreviations frequently used throughout this dissertation.

Abbreviation	**Description**
ANN	Artificial Neural Network
ATM	At-The-Money
B-S	Black Scholes
C	Call Option
DL	Deep Learning
GARCH	Generalized Autoregressive Conditional Heteroskedasticity
GBM	Geometric Brownian Motion
GRU	Gated Recurrent Unit
ITM	In-The-Money
LSTM	Long Short Term Memory
MAE	Mean Absolute Error
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
OTM	Out-of-The-Money
RMSE	Root mean square error
RNN	Recurrent neural networks
S	Stock Price

Table 1: Abbreviations.

3 Materials and Theoretical background

This section discusses several approaches relevant to financial modeling, including the use of Black-Scholes and Geometric Brownian Motion (GBM) simulations, as well as Multilayer Perceptron (MLP), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU) machine learning algorithms. It also investigates the use of related tools such as activation functions and optimizers. These techniques will be carried out using Python.

3.1 Geometric Brownian Motion Simulating (GBM)

GBM is a stochastic process and mathematical model in financial mathematics used for simulating the dynamics of different financial assets, such as stock prices. GBM depicts a stochastic process operating in continuous time with a Brownian motion represented by the drift in the financial instrument's logarithm. [41]. The model is defined by the stochastic differential equation (SDE):

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

Where S_t is the price of the asset at time t, W_t is a Wiener process or Brownian motion, and mu (the percentage drift) and $sigma$ (the percentage volatility) are constants. GBM is widely employed in financial modeling due to its simplicity and ability to simulate the log-normal behavior seen in real-world stock prices. GBM assumes that the logarithm of stock prices is normally distributed, making it an appropriate model for assets that cannot assume negative values, such as stock prices. This feature is critical when modeling financial derivatives, such as options, because the method for pricing them (such as the Black-Scholes formula) requires an assumption about the underlying asset's price distribution [13].

3.2 Options and Call Option

Options are a valuable financial instrument in today's society. Financial options allow holders to purchase or sell an underlying asset at a specified strike price on a specific date or series of dates, as explained in the introduction. There are two main categories of possibilities. The first type of option is called a call that allows buyers to purchase an underlying asset at a specific price (strike price) and time(s). The second form of the option is the put which allows buyers to sell a specific asset (underlying) at a defined price (strike price) and time(s). Puts and calls are simple but can be paired with other assets to form sophisticated financial instruments for various situations. Both individuals and businesses utilize options for hedging, insurance, and other purposes. There are several subcategories of puts and calls. There are considerable differences between European and American styles. European options can only be exercised at their expiration date. European option holders can only pick on the expiration date. On the other hand, American options provide a bit more flexibility. They can be exercised at any moment up to and including the expiration date [1].

A call option is a financial contract in which the holder has the right, but not the obligation, to purchase a certain quantity of an underlying asset at a predefined price, known as the strike price, within a specified time period. This financial instrument is commonly utilized in the context of stock markets, but it can also be applied to other assets such as commodities, currencies, and indexes. When an investor buys a call option, they are essentially buying the right to profit from the future growth in the value of the underlying asset. If the asset's price climbs above the strike price before the option expires, the investor may exercise the option by purchasing the asset at the lower strike price and potentially selling it at the current market price for a profit. The maximum profit is theoretically unlimited because it depends on how far the asset's price can rise above the strike price. However, if the price of the underlying asset does not rise over the strike price within the given time frame, the call option will expire worthless. In this situation, the investor's loss is limited to the premium paid to acquire the option. The premium is the price of the call option, which is decided by a number of criteria such as the current price of the underlying asset, the strike price, the amount of time to expiration, volatility, and interest rates. Call options are commonly utilized as hedging or speculative techniques. For example, investors may utilize call options to hedge against the risk of price increases in assets they plan to purchase later. Alternatively, speculators may acquire call options to gain exposure to possible price increases without investing the entire amount of capital required to buy the underlying asset outright [8].

3.3 Black-Scholes model (B-S)

The Black-Scholes model evaluates the theoretical value of European-style options using an equation developed from the notions of stochastic calculus. The conventional parameters of the model will be modified to take into account the distinct features of the stock market, namely extreme volatility and lack of payouts.

The Black-Scholes model implies that stock values follow a lognormal distribution since asset prices cannot be negative. In other words, they are zero-bounded. The model assumes that equities do not pay dividends or returns and that options can only be exercised on the expiration or maturity date, hence it cannot appropriately price American options. Rather, the strategy is widely used in the European options market. In B-S the stock market is extremely unpredictable, and a state of random walk is assumed because market direction can never be accurately forecast. The model makes no assumptions about transaction costs such as commissions and brokerage fees, interest rates are assumed to be constant, hence the underlying asset is deemed risk-free and stock returns are normally distributed, implying that the volatility of the market is constant over time, and without arbitrage, the opportunity of making a riskless profit is avoided [7].

As introduced, the Black-Scholes equation, a fundamental tool in financial options pricing, is expressed as:

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

where V represents the option price, S the current stock price, σ the volatility of the stock, r the risk-free interest rate, and t the time.

In the Black-Scholes framework, specific assumptions lead to detailed solutions for call options. Here we explore these conditions:

- The value of a call option at time t with zero remaining life is zero, i.e., $C(0,t) = 0$ for all t.
- As the stock price S approaches infinity, the call option behaves asymptotically as $C(S,t) = S - K$.
- At expiration, the value of a call option is $C(S,T) = \max\{S - K, 0\}$, where K is the strike price.

These conditions relate to the behavior of the option under different scenarios but do not describe 'variables' by themselves.

The solution to the Black-Scholes equation for a European call option, assuming no dividends are paid, is given by:

$$C(S,t) = SN(d_1) - Ke^{-r(T-t)}N(d_2) \tag{2}$$

where $N(x)$ is the cumulative distribution function of the standard normal distribution. The parameters d_1 and d_2 are defined as:

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left(\ln\frac{S}{K} + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right), \tag{3}$$

$$d_2 = d_1 - \sigma\sqrt{T-t}. \tag{4}$$

Where :

- C = call option price
- N = CDF of the normal distribution
- S = the current price of the underlying stock
- K = Strike Price, the exercise price of the option
- $T - t$ = Time to Expiration (in years), the time remaining until the option expires
- σ = Volatility, the standard deviation of the stock's returns
- r = the interest rate of a risk-free investment

This model derivation assumes that the stock follows a geometric Brownian motion with constant drift and volatility, and that the market is free of arbitrage opportunities.

However Black-Scholes model has several significant limitations such as [26] :

Constant Volatility Assumption: The Black-Scholes model assumes that the volatility of the underlying asset is constant over the life of the option. However, in reality, volatility is often stochastic and can vary significantly over time. This discrepancy can lead to inaccuracies in option pricing.

Assumption of Log-Normal Distribution: The model presumes that the returns of the underlying asset follow a log-normal distribution and that the price process is continuous. In practice, financial data often exhibit jumps and fat tails, indicating that returns may not be normally distributed.

No Arbitrage and Perfect Hedging: The derivation of the Black-Scholes model relies on the assumption of no arbitrage and the ability to perfectly hedge the option by continuously rebalancing a portfolio of the underlying asset and a risk-free bond. However, in the real world, transaction costs and other market frictions make perfect hedging impossible.

Interest Rate Assumption: The model assumes a constant risk-free interest rate. In reality, interest rates can fluctuate, impacting the present value of future cash flows and thus affecting the option price.

Dividend Payments: The original Black-Scholes model does not account for dividends paid on the underlying asset. This limitation has led to modifications like the Merton model, which includes continuous dividend yields, but this still does not fully address all types of dividend structures encountered in practice.

These limitations highlight the challenges of applying the Black-Scholes model in real-world scenarios, where market conditions often deviate from the model's assumptions. As a result, various extensions and alternative models have been developed to address these issues and provide more accurate pricing under different market conditions.

3.4 Artificial Neural Networks

Machine learning has been increasingly popular in recent years. Computational capabilities and speeds have grown exponentially throughout time. Computational advancements have led to new opportunities. Machine learning is one such door. Machine learning uses several ways to learn and predict from data. Machine learning outperforms classical econometric methods in terms of forecasting capacity but lacks interpretability. A vast range of machine learning techniques exist some are simply more intricate variants of regression, whilst others are far more sophisticated [36]. Neural networks are a type of machine learning that aims to replicate brain operations. Simulated neurons transmit information in various ways to better grasp underlying relationships Figure 1 [42]. These neurons can be arranged in a number of patterns and layers. Training data is provided to the network, which learns and can predict future data. Neural networks have typically been well-suited for prediction.

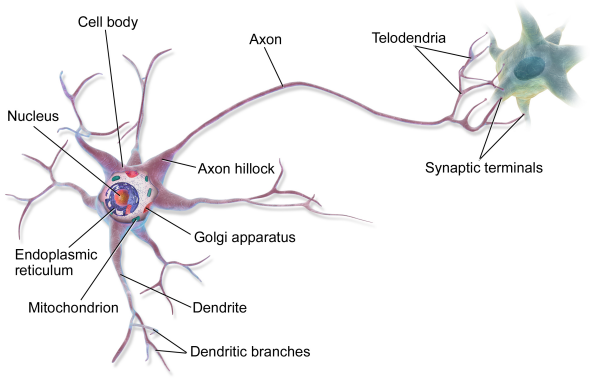


Figure 1: Biological neuron.

3.4.1 Multilayer Perceptron

McCulloch and Pitts (1943) [33] built the first neural network topologies by modeling how neurons in mammals' brains interact. Rosenblatt (1958) [36] introduced the Perceptron, a basic neural network consisting of a single layer and one neuron, inspired by biological neurons.

The Figure 2 depicts a neural network. Inputs are directed to the red nodes on the left. These inputs are subsequently transferred to the hidden layer. The inputs are weighted and processed by activation

functions, a bias may also be added. The findings are submitted to the output layer, where a similar process takes place before the final result is calculated [1].

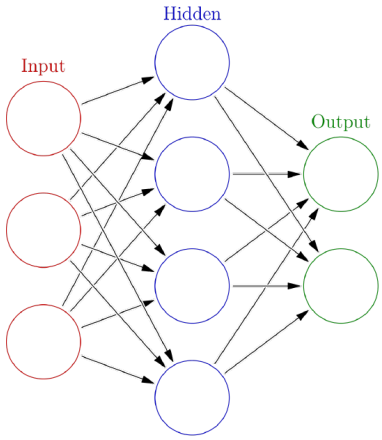


Figure 2: A Simple Neural Network.

The following Figure 3 shows the process from inputs to output in more detail.

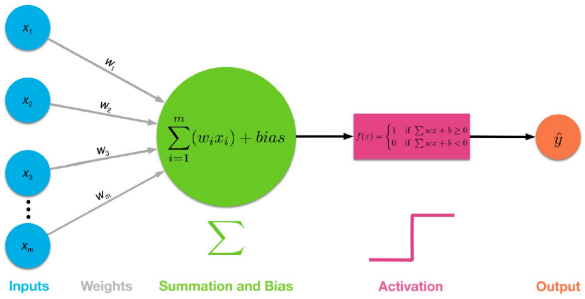


Figure 3: Process in Neural Network.

Where, the inputs (x_1, x_2, x_3, x_n) represent independent variables from a single observation. The synaptic weights $w_1, w_2, w_3, and w_n$ indicate the intensity of each input. The weights are multiplied by the corresponding input. Input functions, such as summation, collect information from inputs. The final option, b, allows for an adjustable offset. The activation function uses the input function's results to generate the output y.

Multilayer Perceptron (MLP) a more complex model was first developed by Fukushima (1980)[16] Based on the perceptron. As seen in Figure 2 The first layer is also known as the input layer. This contains the input vector. The input layer's elements connect to each neuron in the first hidden layer. Additionally, each neuron in the first hidden layer receives a weighted total of all inputs, including the bias factor. The last layer is the output layer. All neurons in the output layer are connected to their outputs from the last hidden layer. The layers in between are the secret layers. The Multilayer Perceptron allows for any number of hidden layers and neurons between the input and output layers. As David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams said [37], an MLP is a type of neural network that consists of multiple layers of neurons, including an input layer, one or more hidden layers, and an output layer. Each neuron in one layer is connected to every neuron in the next layer. In a Multilayer Perceptron (MLP), the core objective is to adjust the weights of the connections in a network to minimize the error between the actual output and the desired output. This adjustment process is primarily achieved through a learning procedure known as back-propagation. The MLP consists of an input layer, multiple hidden layers, and an output layer. Each unit, or neuron, in these layers is connected to units in the subsequent layer through weighted connections. The total input x_i to a unit j is calculated as a linear combination of the outputs y_i from the previous layer and the corresponding weights w_{ij} on these connections. The weighted sum or total input to a neuron can be expressed mathematically as:

$$x_j = \sum_i w_{ij} y_i + b_j \tag{5}$$

where:

- x_j is the total input to neuron j,
- w_{ij} is the weight from neuron i to neuron j,
- y_i is the output of neuron i from the previous layer,
- b_j is the bias of neuron j.

Additionally, biases can be introduced by adding an extra input to each unit with a constant value, typically set to 1, and its corresponding weight acts as the bias.

Each unit's output y_j is a non-linear function of its total input x_j. A commonly used non-linear function is the sigmoid function, defined as:

$$y_j = f(x_j) \tag{6}$$

where f could be a sigmoid or ReLU function.

The goal of the learning process is to minimize the total error E, which is the sum of the errors across all the output units for all training cases. The error E for a single output neuron is calculated using the difference between the actual output y_j and the desired output d_j. The error for a single case can be calculated as:

$$E = \frac{1}{2} \sum_j (d_j - y_j)^2$$

where d_j is the desired output and y_j is the actual output of the unit j.

The factor of $\frac{1}{2}$ is used to simplify the derivative calculation during backpropagation.

The backpropagation algorithm updates the weights to minimize the error. This involves computing the gradient of the error with respect to each weight and adjusting the weights in the direction that reduces the error.

To minimize the error E using gradient descent, we need to compute the partial derivatives of E with respect to each weight in the network. This is done in two passes: the forward pass and the backward pass. In the forward pass, the inputs are propagated through the network to compute the outputs. In the backward pass, the error derivatives are propagated backward from the output layer to the input layer.

For Output Layer Error :

$$\delta_j = (d_j - y_j) f'(x_j) \tag{7}$$

where δ_j is the error term for the output neuron j, and $f'(x_j)$ is the derivative of the activation function with respect to the input x_j.

For a hidden neuron h, the error term δ_h is computed by backpropagating the errors from the neurons in the next layer:

$$\delta_h = f'(x_h) \sum_j \delta_j w_{hj} \tag{8}$$

where w_{hj} is the weight from hidden neuron h to output neuron j.

The weights are updated using the gradient descent rule:

$$\Delta w_{ij} = \eta \delta_j y_i \tag{9}$$

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij} \tag{10}$$

where η is the learning rate, and Δw_{ij} is the change in the weight from neuron i to neuron j.

The training process of an MLP begins with initializing the weights and biases with small random values. For each training sample, a forward pass is performed to compute the output of each neuron from the input layer to the output layer. The error at the output layer is then computed. During the backward pass, the error term δ is computed for each neuron, starting from the output layer back to the input layer. The weights and biases are then updated using the computed error terms and learning rate. This process is repeated for multiple epochs until the error is minimized to an acceptable level. This process allows the MLP to learn from the training data by adjusting the weights to minimize the error, effectively learning the mapping from input to output.

3.4.2 Long Short Term Memory networks (LSTM)

Recurrent neural network (RNNs) differ from typical neural networks by using a transformation weight to transfer information across time. The transition weight indicates that the model has memory, as the next state is dependent on the preceding. RNNs use hidden layers to store information from previous stages. The term "recurrent" refers to a model that predicts future values by repeating the same task for each element in the sequence using previously gathered information. 4 represents An RNN used for processing input time series data. In this architecture, the network contains a sequence of states $s_1, s_2, \ldots, s_p$ that represent the hidden layers in the RNN. The initial state s_0 receives the first input of the time series x_{t-p}. As the network processes each subsequent time series input $(x_{t-p+1}, x_{t-p+2}, \ldots, x_{t-1})$, it updates its state from s_1 to s_p.

The last state s_p is particularly crucial as it captures the relevant information from the entire sequence of inputs and uses this information to produce the output $y = \hat{x}_t$, which is the prediction of the next value in the time series. This output is generated based on the accumulated knowledge of the prior states and inputs, reflecting the network's ability to remember and synthesize information over time to predict future values in the sequence. This characteristic is fundamental in applications such as time series forecasting, where past information is used to predict future events [17].

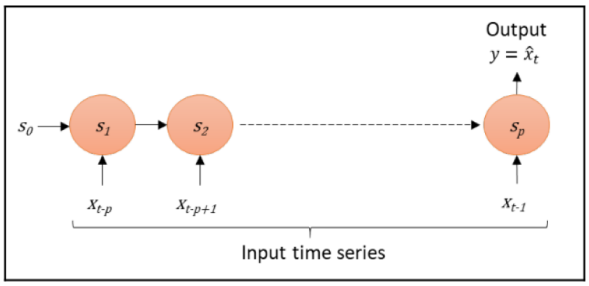


Figure 4: Recurrent neural network with p time steps.

Long Short Term Memory networks, or "LSTMs," are a type of RNN that can learn long-term dependencies. They were presented by Hochreiter and Schmidhuber (1997) [20] and improved and popularized by numerous others in subsequent work. They function exceptionally well for a wide range of conditions and are now frequently used.

LSTMs are specifically designed to avoid the long-term reliance problem. Remembering information for lengthy periods of time is almost automatic for them. All recurrent neural networks are composed of a chain of neural network modules that repeat. In typical RNNs, this repeating module will be quite simple, with only one tanh layer. the diagram 5 shows a complete diagram of LSTM. The LSTM has four components: input gates, forget gate, cell state and output gate [35].

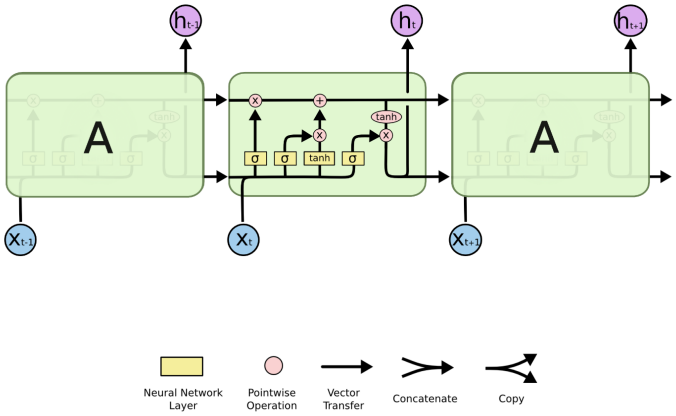


Figure 5: The repeating module in an LSTM contains four interacting layers.

In Figure 5, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned

neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied and the copies going to different locations. The LSTM model is defined below. Let x_t, h_t, and C_t represent the input, control state, and cell state at timestep t. Given a set of inputs $(x_1, x_2, ..., x_m)$, the LSTM computes the h-sequence $(h_1, h_2, ..., h_m)$ and the C-sequence $(C_1, C_2, ..., C_m)$ as follows:

Input Gate: the purpose is to receive new information x_t via two functions: r_t and d_t. The rt concatenates the prior hidden vector h_{t-1} with the new information x_t, resulting in $[h_{t-1}, x_t]$. It then multiplies it with the weight matrix W_r and a noise vector b_r. The d_t performs a similar job. The cell state c_t is then calculated by multiplying r_t and d_t element-wise.

$$r_t = \sigma\left(W_f \cdot [h_{t-1}, x_t]\right) + b_f \tag{11}$$

$$d_t = \tanh\left(W_d \cdot [h_{t-1}, x_t]\right) + b_d \tag{12}$$

Forget Gate: Looking quite similar to r_t in the input gate, the forget gate f_t determines the limit up to which a value is retailed in memory.

$$f_t = \sigma\left(W_i \cdot [h_{t-1}, x_t]\right) + b_i \tag{13}$$

Cell State: An element-wise multiplication is performed between the previous cell state $(C_t - 1)$ and the forget gate f_t. The cell state then adds the results from the input gate (r_t times d_t):

$$C_t = f_t \cdot C_{t-1} + r_t \cdot d_t \tag{14}$$

Output gate: Here, ot represents the output gate at time step t, where W_o and b_o are the output gate's weights and bias. The hidden layer ht either advances to the next time step or outputs as y_t, which is achieved by applying another tanh on h_t.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t]) + b_o \tag{15}$$

$$h_t = o_t \tanh C_t \tag{16}$$

Note that the output gate o_t is not the output y_t, but rather the gate that controls the output.

3.4.3 Gated recurrent unit (GRU)

A Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) architecture that excels in understanding and predicting sequences of data and uses gates to control the flow of information and remember important data over long periods. GRU uses gating techniques to regulate the flow of input, which addresses the vanishing gradient problem found in standard RNNs, these gates effectively allow the model to keep or forget information, which is critical for preserving long-term dependencies on data such as stock prices, which are influenced by long-term economic cycles and short-term market swings. GRU is particularly successful in capturing long-range dependencies in sequential data due to their selective preservation and forgetting of information from the past, making them useful for applications requiring long-sequence memory. GRU combines the forget and input gates into a single update gate. Figure 6 shows the GRU that consists of two main gates: the reset gate R_t and the update gate Z_t, both of which employ sigmoid activation functions (denoted by σ) to control the flow of information.

The reset gate determines how much of the past information (previous hidden state H_{t-1}) needs to be forgotten, while the update gate controls how much of the new information will be used to update the hidden state. The combination of these gates allows the GRU to effectively capture dependencies from varying time intervals without the vanishing gradient problem typical in standard recurrent networks.

The candidate hidden state $\tilde{H}_t$ is computed using a tanh activation function, which considers both the current input X_t and the past hidden state modified by the reset gate. This candidate state suggests a possible new value for the hidden state.

The actual hidden state H_t is then updated as a blend of the previous hidden state H_{t-1} and the candidate hidden state $\tilde{H}_t$, where the proportions are determined by the update gate. This allows the GRU to maintain information over longer periods or overwrite it as needed based on the new input and the internal state of the gates.

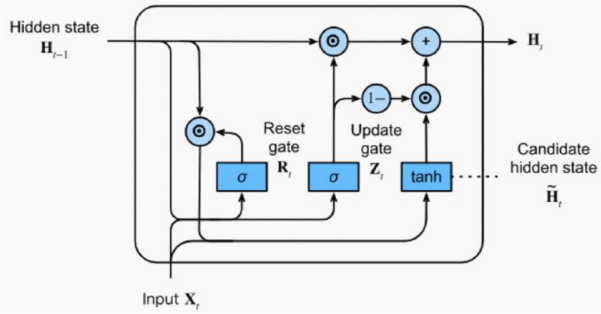


Figure 6: GRU structure.

Consider $x_t \in \mathbb{R}^{n\times d}$ (number of examples: n, number of inputs: d). After the previous hidden step $x_{t-1} \in \mathbb{R}^{n\times h}$ (number of hidden units: h), reset gate $R_t \in \mathbb{R}^{n\times h}$ and update gate $z_t \in \mathbb{R}^{n\times h}$. Integrate the reset gate R_t, leading to the candidate's hidden state [11].

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r) \tag{17}$$

$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z) \tag{18}$$

$$\hat{H}_t = \tanh(X_t W_{xh} + (R_t \circ H_{t-1}) W_{hh} + b_h) \tag{19}$$

Where $W_{xh} \in \mathbb{R}^{n\times h}$ and $W_{hh} \in \mathbb{R}^{h\times h}$ are weight parameters, $b_h \in \mathbb{R}^{1\times h}$ is the bias, and the symbol $\circ$ is the Hadamard product.

Reset Gate: The gate R_t determines if the previous hidden state must be ignored. The gate Z_t is generated for the update gate with hat (H_t). W_z and W_r are the weight parameters to be trained while b_z and b_r are the noise vectors.

Update Gate: (Part I): This part multiplies R_t and $H_t - 1$. The multiplication means how much of $H_t - 1$ is retained or ignored. Thus, a temporal hat (H_t) is created to be used for the update of H_t. W_h and b_h are weight parameters and the noise vectors, respectively.

Update Grade: (Part II): This part computes the weighted average between $H_t - 1$ and hat $(H_t\ )$, according to the weight Z_t. If Z_t is close to zero, then the past information contributes little and new information contributes more.

3.5 Neural Networks Challenges

To build a robust model, it is essential to understand the challenges we may encounter during the process. In this section, we discuss the challenges associated with Neural Networks, as illustrated in Figure 7. Additionally, we provide some suggestions to address each challenge.

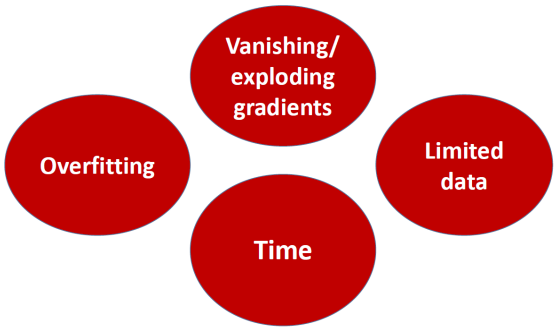


Figure 7: Neural Networks challenges.

3.5.1 Vanishing/exploding gradients

- Initialization

In deep neural networks, the vanishing gradient problem occurs when gradients become very small, causing the network to learn very slowly or not at all. The exploding gradient problem occurs when

gradients become very large, causing the network weights to become unstable. Both issues can be detrimental to the training of deep networks. For solving this we can use an initialization method that ensures that the variances of the input signals are preserved across layers, preventing the magnitudes of the signals from either exponentially decreasing (vanishing) or increasing (exploding). This method is particularly designed for networks using rectified linear units ReLU and its variants. the variance of the output at each layer is maintained if the weights are initialized from a zero-mean Gaussian distribution with variance $2/n$, where n is the number of input units. This scaling factor ensures that the output variance is approximately equal to the input variance divided by 2 after each layer due to the ReLU non-linearity, which only allows half of the activations (the positive part) to pass [19].

- Batch Normalization

Batch Normalization is a crucial technique in deep learning that addresses the issue of internal covariate shift. Internal covariate shift refers to the change in the distribution of network activations due to the updating of network parameters during training. This shift can slow down the training process and make it difficult to tune the learning rate. Batch Normalization normalizes the input to each layer so that the mean output is close to zero and the standard deviation is close to one. This normalization helps in stabilizing and accelerating the training of deep neural networks[23]. By defining is input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$ and output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$ then Steps in Batch Normalization are :

- The mean of the mini-batch is calculated as:

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

  Here, $\mu_{\mathcal{B}}$ represents the average of the mini-batch values. This step ensures that the average activation of the batch is close to zero.

- The variance of the mini-batch is computed as:

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

  $\sigma_{\mathcal{B}}^2$ measures the spread of the mini-batch values around the mean. This helps in understanding the variability within the batch.

- Each input x_i in the mini-batch is normalized using the mini-batch mean and variance:

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

  Here, ϵ is a small constant added for numerical stability to prevent division by zero. This step scales the inputs to have a mean of zero and a standard deviation of one.

- After normalization, the normalized input $\hat{x}_i$ is scaled by a parameter γ and shifted by a parameter β to produce the final output:

$$y_i \leftarrow \gamma\hat{x}_i + \beta = \mathrm{BN}_{\gamma,\beta}(x_i)$$

  These parameters γ and β are learned during the training process and allow the model to restore the representation power lost during normalization.

Batch Normalization is a powerful technique that enhances the training of deep neural networks by normalizing the inputs to each layer, stabilizing the learning process, and allowing for higher learning rates and less dependency on careful initialization. It contributes to more efficient and effective deep network training, making it an essential component in modern neural network architectures.

3.5.2 Limited data

In machine learning and deep learning, one of the most important aspects influencing model performance is the amount of data given to the algorithm. However, in any machine learning or deep learning challenge, there is insufficient data to accurately train the model. In this circumstance, dealing with the problem with minimal data is critical without sacrificing precision. One solution is to transfer learning and use other machines and customize it to our needs and the other could be to simulate the data.

3.5.3 Time Limits

One challenge we often face in neural networks is how to reduce the training time from hours to minutes.

- Optimizer and activation functions

One solution is to use activation functions like ELU and an optimizer such as Adam [27] that will be explained more in 5.

- Learning Rate

Another suggestion is optimizing the learning rate through a disciplined approach involving the learning rate range test, cyclical learning rates, and super-convergence. These methods enable faster convergence, reduce the number of training epochs, and improve overall performance, providing a practical and efficient solution to the time challenges in deep learning training. Cyclic Learning Rates (CLR) involves specifying minimum and maximum learning rate boundaries and step size, creating a cycle where the learning rate linearly increases to the maximum value and then decreases back to the minimum. This approach allows for faster convergence as it adapts the learning rate dynamically during training. The method helps to escape local minima and saddle points more effectively, leading to faster and more robust training. The concept of learning rate scheduling is pivotal in the optimization process of machine learning models. Different scheduling strategies are employed to adjust the learning rate over time to improve training efficiency and performance. Power scheduling reduces the learning rate according to the formula :

$$\eta(t) = \eta_0/(1 + t/s)^c$$

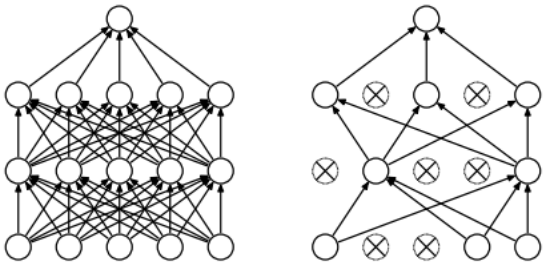
where η_0 is the initial learning rate, t is the current time step, s is a scaling parameter, and c is the decay rate. Exponential scheduling, defined by

$$\eta(t) = \eta_0 \cdot 0.1^{t/s}$$

decreases the learning rate exponentially with time, facilitating faster convergence in some scenarios. Other methods include piecewise constant scheduling, performance scheduling, and 1-cycle scheduling, each offering unique benefits to address specific challenges in the training process. These techniques are crucial for achieving optimal performance and understanding and applying these strategies can significantly impact the effectiveness of model training[39].

3.5.4 Over Fitting

Overfitting occurs when a model learns the training data too well, including its noise and outliers, leading to poor generalization on unseen data. A method for enhancing neural networks by reducing overfitting is called dropout. Using traditional backpropagation learning, brittle co-adaptations are developed that are effective on training data but not on unknown data. These co-adaptations are broken up by random dropouts making the presence of any specific hidden unit unpredictable. In numerous application domains, such as object classification, digit identification, speech recognition, document classification, and computational biology data analysis, it was discovered that this technique enhanced the performance of neural nets. This implies that dropout is a general strategy that isn't domain-specific. The fact that dropout lengthens training duration is one of its disadvantages. It usually takes a dropout network two to three times as long to train as a regular neural network with the same topology. This rise can be attributed in large part to the extremely noisy parameter adjustments. Effectively, every training example attempts to learn a distinct random architecture [40].



(a) Standard Neural Net (b) After applying dropout.

Figure 8: Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right:An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

3.6 GARCH (Generalized Autoregressive Conditional Heteroskedasticity) model

The Generalized Autoregressive Conditional Heteroskedasticity (GARCH) model is a popular tool for modeling time-varying volatility in financial time series data. Integrating the GARCH model into options pricing helps address the limitations of constant volatility assumptions found in traditional models like Black-Scholes. The GARCH model captures time-varying volatility and volatility clustering, phenomena often observed in financial markets where periods of high volatility are followed by more high volatility. By incorporating GARCH, the model can adjust to the stochastic nature of volatility, providing a more realistic representation of market conditions. This integration allows neural networks to learn and predict under these varying conditions, enhancing their ability to model the nonlinear relationships between volatility and option prices. As a result, GARCH-enhanced models offer improved stability and accuracy, making them more robust in capturing market dynamics that influence option pricing.

Given that the log-normal diffusion hypothesis of continuous volatility is not exactly followed by prices. Thus, time-varying volatility, a noisy estimate of the mean, serial correlation in returns, and the non-normal distribution of returns all contribute to issues [14]. It extends the Autoregressive Conditional Heteroskedasticity (ARCH) model by incorporating lagged conditional variances into the volatility equation, providing a more flexible and accurate representation of volatility clustering.

Mathematically, a GARCH(1,1) model is defined by two equations: the mean equation and the variance equation. The mean equation models the returns as a function of past returns and a stochastic error term. The variance equation models the conditional variance (volatility) as a function of past squared residuals and past conditional variances.

The mean equation can be written as:

$$r_t = \mu + \epsilon_t \tag{20}$$

where r_t represents the return at time t, μ is the mean return, and ϵ_t is the error term, which is assumed to be normally distributed with a mean of zero and time-varying variance σ_t^2. given that the log-normal diffusion hypothesis of continuous volatility is not exactly followed by prices. Thus, time-varying volatility, a noisy estimate of the mean, serial correlation in returns, and the non-normal distribution of returns all contribute to issues [12].

The variance equation for a GARCH(1,1) model is given by:

$$\sigma_t^2 = \omega + \alpha\epsilon_{t-1}^2 + \beta\sigma_{t-1}^2 \tag{21}$$

where σ_t^2 is the conditional variance at time t, ω is a constant, α is the coefficient for the lagged squared residual (ϵ_{t-1}^2), and β is the coefficient for the lagged conditional variance (σ_{t-1}^2).

The parameters ω, α, and β are estimated using maximum likelihood estimation (MLE). The model captures the persistence in volatility through the lagged terms $\alpha\epsilon_{t-1}^2$ and $\beta\sigma_{t-1}^2$, allowing for periods of high and low volatility to cluster together over time.

In summary, the GARCH(1,1) model provides a robust framework for modeling financial time series with time-varying volatility, capturing the phenomenon of volatility clustering observed in financial markets. By including both past squared residuals and past variances, the GARCH model effectively describes the evolution of volatility, making it a valuable tool for risk management and financial forecasting.

3.7 Evaluation metrics

Evaluating the performance of a machine learning model is crucial to understanding its accuracy and reliability. One such method involves using the ratio of predicted values to actual values minus one, expressed as $\frac{y_{\text{pred}}}{y_{\text{test}}} - 1$. This metric, referred to as the Fractional Error, offers a unique perspective on the model's performance by providing a direct measure of the relative difference between predicted and actual values.

Let y_{pred} denote the predicted values generated by the machine learning model, and let y_{test} denote the actual values from the test dataset. The Fractional Error is defined as:

$$\text{Fractional Error} = \frac{y_{\text{pred}}}{y_{\text{test}}} - 1 \tag{22}$$

Using $\frac{y_{\text{pred}}}{y_{\text{test}}} - 1$ as an evaluation metric provides several benefits:

Firstly, this method provides insights into the proportional accuracy of predictions, making it particularly useful in applications where relative errors are more significant than absolute errors. By expressing the error as a ratio, it highlights both overestimations and underestimations directly.

Positive values of this metric indicate that the model has overestimated the actual value, while negative values reveal underestimations. This dual representation allows for a clear understanding of the nature and direction of the model's errors, which is critical for fine-tuning and improving model performance.

Moreover, this evaluation method facilitates comparison across different scales and units. By normalizing the prediction errors in relation to the actual values, it becomes easier to compare performance across various datasets or models, regardless of the magnitude of the values involved. This is particularly advantageous in fields such as finance or engineering, where the scale of values can vary significantly.

While this metric offers significant insights, it is important to be aware of potential drawbacks. The ratio can be sensitive to very small actual values (y_{test}), leading to disproportionately large error values. This sensitivity could skew the overall assessment of model performance, especially if the dataset contains a mix of large and small values.

To mitigate this, it is advisable to use additional evaluation metrics alongside $\frac{y_{\text{pred}}}{y_{\text{test}}} - 1$. Complementing this metric with others in Table 2, can provide a more comprehensive picture of the model's performance.

Using $\frac{y_{\text{pred}}}{y_{\text{test}}} - 1$ as an evaluation metric, referred to as the Fractional Error, provides valuable insights into the proportional accuracy of a machine learning model's predictions. It highlights the nature and direction of prediction errors, allowing for targeted improvements and facilitating comparison across different datasets and contexts. While it offers significant benefits, it should be complemented with other evaluation metrics to ensure a robust and balanced assessment of model performance [31].

Four different measures of forecast errors for evaluating the model performance and the accuracy of the methods have been used which are MAE, MSE, RMSE, AND R-squared where $haty$ t are the forecasted values, y_t the observed values, n is the number of forecasts and MUE is the average of measurements [17].

Evaluation Metrics	**Equation**
Mean square error (MSE)	$\frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{y}_t)^2$
Root mean square error (RMSE)	$\sqrt{\frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{y}_t)^2}$
Mean absolute error (MAE)	$\frac{1}{n}\sum_{t=1}^{n}\lvert y_t - \hat{y}_t\rvert$
R-squared	$1 - \frac{\sum_{t=1}^{n}(y_t - \hat{y}_t)^2}{\sum_{t=1}^{n}(y_t - \mu)^2}$
Fractional Error	$\frac{y_{\text{pred}}}{y_{\text{test}}} - 1$

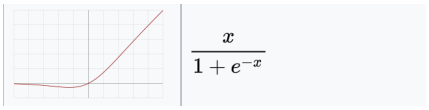
Table 2: Evaluation Metrics

Choosing the right evaluation metrics is crucial for assessing the performance of machine learning models in financial contexts. Metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared offer quantitative insights into a model's accuracy and reliability. MAE provides a straightforward average of errors, highlighting absolute discrepancies between predictions and actual values. MSE emphasizes larger errors due to its squaring of differences, making it sensitive to outliers. RMSE, as the square root of MSE, offers an interpretable scale of error magnitude in the same units as the target variable. R-squared measures the proportion of variance explained by

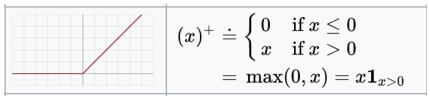
the model, indicating its goodness-of-fit. Additionally, the Fractional Error metric, calculated as the ratio of predicted to actual values minus one, offers insight into proportional accuracy. It highlights overestimations and underestimations, providing a nuanced view of model performance. Employing a combination of these metrics allows for a comprehensive assessment, guiding model selection and refinement to ensure robust predictions.

3.8 Activation functions

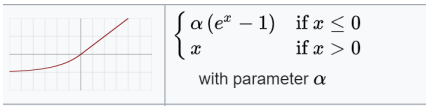
According to many studies, such as [24], ELU, ReLU, and Leaky ReLU could be among the best activation functions for option pricing. Therefore, we use these activation functions in our models. Neural networks use activation functions to model non-linear relationships between input and output variables. The activation function of a node in an artificial neural network is a function that computes the node's output depending on its inputs and weights. Neural networks depend on activation functions to determine whether a neuron should be activated or not. They give the model non-linear characteristics, which allow it to pick up intricate patterns [6].



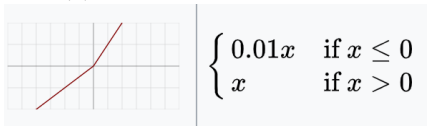
(a) Sigmoid Activation Function



(b) ReLU Activation Function.



(c) ELU Activation Function.



(d) Leaky ReLU Activation Function.

Figure 9: Activation Function

Activation functions for neural networks vary widely. The sigmoid function is a typical activation function, It converts inputs to values between 0 and 1 Figure 9a. ReLu was first proposed by Hahnloser et al. (2000) [18] proposed a function that is linear for positive values and always zero for negative values. When training a neural network with backpropagation, the ReLu activation function resembles a linear activation function but differs due to the non-constant derivative Figure 9b. Unlike ReLU, which outputs zero for all negative inputs, ELU allows for negative values, which helps in learning representations for negative inputs more effectively. This can be particularly useful in reducing the dead neuron problem, where neurons become inactive and stop learning Figure 9c. Leaky ReLU allows for a small, non-zero gradient when the input is negative, typically a small slope such as 0.01. This ensures that neurons continue to update even with negative input values, which can help in maintaining the flow of gradients throughout the network and potentially improve overall learning performance. By retaining a slight slope for negative inputs, Leaky ReLU provides a more robust alternative to standard ReLU, helping to mitigate issues related to gradient-based optimization in deep learning models Figure 9d.

Traditional activation functions like sigmoid tend to saturate, meaning that they squash input values into a small range, resulting in gradients that are either too small (vanishing) or too large (exploding) during backpropagation, use Non-saturating activation functions like ReLU mitigate because they do not squash input values in the same way [43].

3.9 Optimizer

According to many studies like [24] Adam could be one of the best optimizers for option pricing so we use this optimizer for our models. Adam, short for Adaptive Moment Estimation, is an algorithm designed for first-order gradient-based optimization of stochastic objective functions. This method is particularly useful in machine learning applications involving large datasets and high-dimensional parameter spaces. Adam is built upon the principles of two other optimization techniques: AdaGrad and RMSProp, integrating their advantages while addressing some of their limitations. Adam computes adaptive learning rates for each parameter. It achieves this by maintaining running averages of both the gradients and their second moments (the uncentered variances). The key hyperparameters involved are α (alpha): The stepsize or learning rate, β_1: The exponential decay rate for the first moment estimates, β_2: The exponential decay rate for the second moment estimates, and ϵ (epsilon): A small constant to prevent

division by zero. The updates of the parameters in Adam first Initialize first moment vector m_0 and second moment vector v_0 to zero. At each timestep t :

- Compute the gradient g_t.
- Update biased first moment estimate: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$.
- Update biased second raw moment estimate: $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$.
- Correct bias in first moment estimate: $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$.
- Correct bias in second raw moment estimate: $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$.
- Update parameters: $\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$.

Adam incorporates bias correction terms to counteract the initialization bias in the moment estimates, ensuring that the estimates are unbiased towards zero during the initial steps of training. Adam's update rule is invariant to diagonal rescaling of the gradients, making it robust to varying gradient scales. The effective stepsize is bounded, ensuring that the updates remain within a trust region around the current parameter values. This makes it easier to select the learning rate *alpha* in advance. Adam has demonstrated superior performance in various machine learning tasks, including training deep neural networks. It converges faster and more reliably than other stochastic optimization methods, such as SGD with momentum, Adagrad, and RMSProp, especially in the presence of noisy and sparse gradients [28].

Here is an explanation of how Adams works [17]. Step size ϵ (suggested default: 0.001), Exponential decay rates for moment estimates, ρ_1 and ρ_2 in [0, 1) (suggested defaults: 0.9 and 0.999), Small constant δ (1e-8), and Initial parameters θ are required. first initialize first and second moment variables $s = 0$ and $r = 0$ and time step $t = 0$. while not stopping criteria:

- Sample a minibatch of size m from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$
- Compute gradient estimate: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
- Increment time step: $t \leftarrow t + 1$
- Update biased first moment estimate: $s \leftarrow \rho_1 s + (1 - \rho_1) g$
- Update biased second moment estimate: $r \leftarrow \rho_2 r + (1 - \rho_2) g \odot g$
- Correct bias in first and second moment: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$, $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
- Compute update: $\Delta\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$
- Update: $\theta \leftarrow \theta + \Delta\theta$

Adam's advantages include computational efficiency, low memory requirements, and suitability for large-scale problems. Its adaptability adjusts the learning rate based on the data's first and second moments, allowing it to handle non-stationary objectives effectively. The bias-corrected estimates and bounded effective stepsize contribute to its robustness, ensuring stable and reliable convergence [28].

4 Review of Related Work

In this section, the study provides a review of the usage of various methodologies in the pricing of different choices over the years, including a comparison of their advantages and disadvantages, accuracy, and robustness.

Traditional option pricing approaches, such as the Black-Scholes model (B-S), are based on a few assumptions about the underlying asset and market dynamics, including constant volatility and log-normal returns. These assumptions may not be valid in real-world markets, resulting in erroneous pricing and risk management [21].

Modern advances in mathematical analysis, processing hardware, and software, and the availability of large data have enabled the development of commoditized machines capable of learning to function as investment managers, financial analysts, and traders. By training a fully-connected feed-forward DL neural network it is possible to reproduce the Black and Scholes (1973) option pricing formula with remarkable accuracy. By providing a basic introduction to neural networks, as well as details on the many hyper-parameters that boost the model accuracy, DL nets can learn option pricing models from the markets and can be trained to mimic option pricing traders who specialize in a single stock or index [5].

Deep Learning (DL) is an advanced machine learning a promising branch of artificial intelligence, DL has received a lot of attention recently. Compared to traditional machine learning approaches, DL has the advantages of unsupervised feature learning, a great capability of generalization, and robust training power for massive data in finance[15].

According to L. Van Mieghem, A. Papapantoleon, and J. Papazoglou [34], neural network architectures for option pricing, such as MLPs have significant impacts on the performance in terms of accuracy, measured by MSE, and computational efficiency, measured by training time. MLPs, due to their simplicity, provide reasonable performance for standard problems like the Black-Scholes models.

According to Diederik P. Kingma and Jimmy Ba. Adam [29], deep learning can be used to forecast the future price of an underlying asset and then use that prediction to price the option. Another technique is to explicitly train the neural network to forecast the option price, given a collection of market characteristics such as the current asset price, volatility, and time till expiration.

According to Habib Zouaoui and Meryem-Nadjat Naas [44] deep learning models like LSTM and GRU networks can enhance the accuracy of option pricing models compared to traditional methods like the Black-Scholes model. However, it has some challenges such as data availability and quality, computational resources, model selection and validation, interpretability, and overfitting. The LSTM model can handle the vanishing gradient problem in recurrent neural networks. The LSTM model can have better pricing performance, with lower MAE, MSE, and RMSE values than the BSM and GRU models. The LSTM model has a superior ability to capture complex and nonlinear relationships in financial data and handle long-term dependencies. The GRU model can perform better than the BSM but is slightly less accurate than the LSTM model, while the GRU model offers a simpler and faster alternative. The performance of these models is heavily dependent on the availability of high-quality data and sufficient computational resources.

Overall, with the advancement of artificial intelligence in recent years, optimizing target values with deep learning approaches has become increasingly simple. Several scholars, investors, and traders began to use artificial intelligence for various types of option pricing. To better comprehend these methods, we offer recent research and the number of articles that use various deep learning models in exchange rate forecasting, as shown in Table 3.

Author(s)/Year	**Country**	**Methodology**	**Best accuracy**
Robert Culkin, Sanjiv R. Das (2017)	USA	BSM, ANN	ANN [9]
Gabriel Adams (2020)	USA	BSM, MLP, LSTM	MLP and LSTM [1]
Edward Chang (2022)	Canada	BSM, CNN-LSTM, ANN	CNN-LSTM [5]
Diogo Pinto Flórido (2022)	Spain	BSM, MLP, LSTM	MLP and LSTM [15]
Prateek Samuel Daniels (2022)	Australia	BSM, MLP, LSTM	LSTM [10]
Yan Liu, Xiong Zhang (2023)	China	BSM, SVM, LSTM, RNN	LSTM [32]
Li, Yan (2023)	China	BSM, MC, MLP	MLP [29]
Habib Zouaoui, Meryem Naas (2023)	Algeria	LSTM, GRU	LSTM [44]

Table 3: Reviewed previous studies.

Based on past research on Table 3, we conclude deep learning models have the potential to greatly enhance option pricing accuracy and profitability, especially when combined with vast volumes of high-quality data and thorough validation and interpretation. However, it is important to highlight that DL models are still relatively new methods of option pricing, and their performance may vary depending on the specific problem and data characteristics. These studies show that the LSTM and MLP models are one of the most effective ways of learning financial time series data. In our research, we utilize the Long Short-Term Memory (LSTM) and Multi-Layer Perceptron (MLP) models due to their high effectiveness. Additionally, we include the Gated Recurrent Unit (GRU) model as it has been less explored by researchers so this study use it as a something new contribution to the field of computational finance.

5 Method

This chapter outlines the comprehensive methodology employed in this research, from setting up the research environment and pre-processing data to developing, simulation, and evaluating advanced neural network architectures. By integrating traditional financial models with modern machine learning techniques, this study aims to enhance the accuracy and robustness of stock option pricing models. The first section covers the research environment, introducing the programming languages and libraries used. the second part details the simulation of stock prices and option prices using both the Geometric Brownian Motion (GBM) model and the Black-Scholes model. The third part of this chapter focuses on data pre-processing, an essential step in preparing the dataset for model training and evaluation. The dataset consists of historical stock prices and associated option prices for Tesla, Inc. (TSLA), covering the period from January 2023 to May 2024. The pre-processing process begins with data cleaning, where missing values and outliers are removed to maintain data integrity. Following this, normalization is applied to scale the features to a standard range, facilitating effective model training. Additionally, feature engineering is performed to create new features.

The neural network architectures implemented in this study are described in the next section. Three types of networks are developed and evaluated: Multi-Layer Perceptron (MLP), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU). This section describes the Neural Networks architecture implemented and the performance metrics are described.

The final section describes the performance metrics, the first performance metric consists of compared model performance including the GARCH model as one of the features. Additionally, the predictive accuracy of the models is evaluated using logarithmic transformations of the target variable.

5.1 Research environment

The models are built with Python programming language in Jupyter Notebook. The decision was made because of the language's simplicity and the availability of pre-built machine learning libraries, which enable faster workflow and easier debugging. Specifically, the libraries employed in the empirical phase of the study are:

Packages	**Description**
math	Used for mathematical functions like logarithms, trigonometric operations, etc.
NumPy	Uses arrays as the main data structure to perform computations.
Pandas	Used to read, write, and manipulate the dataset.
SciPy.stats	A library that includes various branches of science, used in this case for statistical tools.
sklearn.model_selection	Used for splitting data into training and testing sets.
time	Used to calculate the time of computation and training for the models.
Keras.models	Provides the Sequential model for building neural networks.
TensorFlow.keras.layers	Includes layers like Input, LSTM, Dense, Dropout for building neural networks.
Matplotlib.pyplot	Used to make graphs and plots to have a visual interpretation of the models.
Keras.layers	Includes layers like Dense, LeakyReLU and activations like elu for building neural networks.
Keras.optimizers	Provides optimizers like Adam for training neural networks.
sklearn.metrics	Used to calculate evaluation metrics like mean squared error, mean absolute error, and r2 score.
arch	Provides tools for modeling and analyzing conditional heteroskedasticity (ARCH models).
sklearn.preprocessing	Used for data preprocessing, like scaling features with StandardScaler.
datetime	Used for manipulating dates and times.

Table 4: The models implemented using Python in Jupyter Notebook

5.2 simulation

We employed the Geometric Brownian Motion (GBM) model to simulate stock prices and the Black-Scholes model to simulate option prices. The objective of these simulations was to determine the optimal data size and the most effective Activation Function for MLP type. To achieve this, three datasets with

varying sizes and architectures were created. The GBM model is selected for its ability to represent the continuous-time stochastic process of asset prices, reflecting the real-world log-normal behavior of stock prices. The Black-Scholes model is utilized for option pricing and is based on the assumption that the stock prices follow a log-normal distribution.

Simulating Option Prices Using Black-Scholes Model

The Black-Scholes formula for the price of a European call option is given by:

$$C = S_0 e^{-qT} \Phi(d_1) - K e^{-rT} \Phi(d_2) \tag{23}$$

For a put option, the formula is:

$$P = K e^{-rT} \Phi(-d_2) - S_0 e^{-qT} \Phi(-d_1) \tag{24}$$

where:

$$d_1 = \frac{\ln(S_0/K) + (r - q + \sigma^2/2)T}{\sigma\sqrt{T}} \tag{25}$$

$$d_2 = d_1 - \sigma\sqrt{T} \tag{26}$$

Here, S_0 is the current stock price, K is the strike price, T is the time to maturity, r is the risk-free interest rate, q is the dividend yield, σ is the volatility of the stock, and Φ is the cumulative distribution function of the standard normal distribution.

Python Code Implementation:

```
class EuropeanOptionBS:
    def __init__(self, S, K, T, r, q, sigma, Type):
        self.S = S
        self.K = K
        self.T = T
        self.r = r
        self.q = q
        self.sigma = sigma
        self.Type = Type
        self.d1 = self.d1()
        self.d2 = self.d2()
        self.price = self.price()

    def d1(self):
        d1 = (math.log(self.S / self.K) +
              (self.r - self.q + 0.5 * (self.sigma ** 2)) * self.T) / \
             (self.sigma * math.sqrt(self.T))
        return d1

    def d2(self):
        d2 = self.d1 - self.sigma * math.sqrt(self.T)
        return d2

    def price(self):
        if self.Type == "Call":
            price = self.S * math.exp(-self.q * self.T) * norm.cdf(self.d1) - \
                    self.K * math.exp(-self.r * self.T) * norm.cdf(self.d2)
        if self.Type == "Put":
            price = self.K * math.exp(-self.r * self.T) * norm.cdf(-self.d2) - \
                    self.S * math.exp(-self.q * self.T) * norm.cdf(-self.d1)
        return price
```

5.2.1 Simulating the InPut Dataset

To analyze the behavior of option prices under various market conditions, we generated a dataset by varying interest rates (r), strike prices (K), times to maturity (T), and volatilities (σ). The stock price (S_0) was fixed at 100.

To analyze the behavior of option prices under various market conditions, we generated a comprehensive dataset by systematically varying several key parameters within the Black-Scholes model. Specifically, we examined the effects of changes in interest rates, strike prices, times to maturity, and volatilities, while keeping the stock price constant at \$100.

The dataset was created to analyze the optimal size for various time-to-maturity intervals. The first dataset varied the time to maturity (T)from 0.1 to 2.0 years in increments of 0.1 years, resulting in a total size of 84,000 entries. The second dataset varied (T)from 0.1 to 1.0 years in increments of 0.1 years, with a total size of 42,000 entries. The third dataset varied (T)from 0.1 to 0.5 years in increments of 0.1 years, resulting in a total size of 21,000 entries. These datasets encompass both short-term and long-term options, providing comprehensive coverage for analysis.

The interest rates (r) were varied from 0.0 to 0.1 in increments of 0.01. This range captures a spectrum of possible market conditions, from extremely low to moderately high interest rates. The strike prices (K) ranged from \$50 to \$150 in increments of \$5, representing a wide variety of options from deep in-the-money to deep out-of-the-money scenarios. Finally, the volatility (σ) was varied from 0.1 to 2.0 in increments of 0.1, encompassing both low and high volatility environments.

For each combination of these parameters, the price of a European call option was calculated using the Black-Scholes model. The resulting option price was then normalized by dividing it by the stock price, to provide a ratio of the option price to the stock price. This normalization allows for easier comparison across different parameter settings. Additionally, the strike price was normalized by the stock price, yielding a strike-to-stock price ratio.

This methodical approach resulted in a robust dataset that captures the sensitivity of option prices to variations in interest rates, strike prices, times to maturity, and volatilities.

Python Code Implementation: The following code snippet demonstrates how the dataset was generated:

```
# Dataset
r = np.arange(0.0, 0.1, 0.01)  # interest rates
Strike = np.arange(50, 155, 5)  # strike prices
T = np.arange(0.1, 2.1, 0.1)  # times to maturity
sigma = np.arange(0.1, 2.1, 0.1)  # volatilities
stock_price = 100

data = []
for r_ in r:
    for Strike_ in Strike:
        for T_ in T:
            for sigma_ in sigma:
                price = EuropeanOptionBS(100, Strike_, T_, r_, 0, sigma_, "Call").price
                option_stock_ratio = price / stock_price
                strike_stock_ratio = Strike_ / stock_price
                data.append([r_, strike_stock_ratio, T_, sigma_, option_stock_ratio])

data = np.asarray(data)
```

5.2.2 Simulation of Stock Prices

The Geometric Brownian Motion (GBM) model is defined by the stochastic differential equation:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{27}$$

where S_t is the stock price at time t, μ is the drift coefficient, σ is the volatility coefficient, and W_t is a standard Brownian motion.

The solution to this differential equation is given by:

$$S_t = S_0 \exp\left((\mu - 0.5\sigma^2)t + \sigma W_t\right) \tag{28}$$

where S_0 is the initial stock price.

To simulate the stock price using the GBM model, we follow some steps. Define the initial stock price (S_0), the drift coefficient (μ), the volatility coefficient (σ), the time horizon (T), and the number of time steps (N). The time step (dt) is calculated as T/N. Create a sequence of random variables from a standard normal distribution, then compute the cumulative sum to obtain the Brownian motion. The standard Brownian motion is scaled by the square root of the time step. Using the parameters and the Brownian motion, simulate the stock price at each time step using the GBM formula.

Python Code Implementation: The following Python code demonstrates the simulation:

```
# Parameters
S0 = 85  # Initial value
mu = 0.1  # Drift coefficient
sigma = 0.3  # Volatility coefficient
T = 1  # Time horizon (1 year)
N = 252  # Number of time steps
dt = T / N  # Time step

# Generate the Brownian motion
np.random.seed(0)  # For reproducibility
W = np.random.standard_normal(size=N)
W = np.cumsum(W) * np.sqrt(dt)  # Standard Brownian motion

# Simulate GBM
t = np.linspace(0, T, N)
S = S0 * np.exp((mu - 0.5 * sigma**2) * t + sigma * W)
```

In this simulation, the initial stock price S_0 is set to 85. The drift coefficient μ is 0.1, representing the expected return of the stock. The volatility coefficient σ is 0.3, capturing the stock's price fluctuations. The time horizon T is 1 year, divided into $N = 252$ time steps, corresponding to the number of trading days in a year. The time step dt is calculated as T/N.

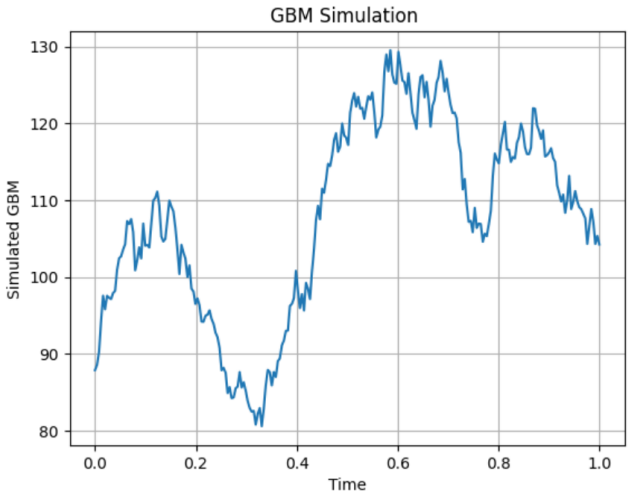


Figure 10: Simulation of stock price with GBM.

Figure 10 depicts a simulation of a GBM model over a time span of one year, as indicated by the x-axis labeled 'Time'. The y-axis, labeled 'Simulated GBM', represents the simulated values in dollars. This plot shows the stochastic movements typical of financial instruments or assets modeled using GBM,

which incorporates both drift and volatility components. The trace begins just below the 100 dollar mark and exhibits significant fluctuations, peaking above 120 dollars, demonstrating the model's ability to capture the erratic behavior of asset prices over time.

5.2.3 Model Architecture

To construct and compile an Artificial Neural Network (ANN) model with specified parameters, we define a function. This function accepts two arguments: neurons, which specify the number of neurons in each layer, and activation function, which determines the type of activation function used in the hidden layers.

The function begins by initializing a sequential model, ANN, which allows us to build the neural network layer by layer. We start by adding the input layer with the number of neurons specified in the first element of the neurons list and an input dimension of 4, corresponding to the four input features.

Next, we add the first hidden layer with the specified activation functions 'relu', 'elu', and 'leaky relu'. We then add additional hidden layers as specified in the neurons list 10, 30, and 60. For each subsequent neuron count in the list, we add a dense layer followed by the specified activation function. This process ensures that the network architecture can be easily adjusted by changing the number of neurons and the activation function. Finally, we add the output layer, which consists of a single neuron, as we are predicting a single value. The model is then compiled using the mean squared error loss function and the Adam optimizer with 150 epochs and 16 batch Sizes, which is well-suited for training neural networks. The resulting ANN model is returned by the function, ready to be trained on the data.

Python Code Implementation: The following Python code demonstrates the simulation:

```
def create_model(neurons, activation_function):
    ANN = Sequential()
    ANN.add(Dense(neurons[0], input_dim=4))
    if activation_function == 'relu':
        ANN.add(Dense(neurons[0], activation='relu'))
    elif activation_function == 'elu':
        ANN.add(Dense(neurons[0], activation=elu))
    elif activation_function == 'leaky_relu':
        ANN.add(LeakyReLU())
    for neuron in neurons[1:]:
        ANN.add(Dense(neuron))
        if activation_function == 'relu':
        ANN.add(Dense(neuron, activation='relu'))
        elif activation_function == 'elu':
        ANN.add(Dense(neuron, activation=elu))
        elif activation_function == 'leaky_relu':
        ANN.add(LeakyReLU())
    ANN.add(Dense(1))
    ANN.compile(loss='mean_squared_error', optimizer='adam')
    return ANN

neurons = [10, 30, 60]
activation_functions = ['relu', 'elu', 'leaky_relu']
histories = {}
training_times = {}
y_pred1 = {}
for activation_function in activation_functions:
    model = create_model(neurons, activation_function)
    start_time = time.time()
    history = model.fit(X_train, y_train, epochs=150, batch_size=16, verbose=0)
    end_time = time.time()
    histories[activation_function] = history
    training_times[activation_function] = end_time - start_time
    y_pred1[activation_function] = model.predict(X_test)
```

This flexible design allows for easy experimentation with different network architectures and activation functions to optimize the model's performance in predicting option prices. The performance of each

activation is in Table 5.

Activation	Training Time (s)	Log Loss	MSE
relu	601.08	0.0421	0.031
elu	514.40	0.0338	0.023
leaky relu	0.0160	450.45	0.016

Table 5: Performance Metrics of Activation

The table provides a comparison of performance metrics for different activation functions used in a neural network model. The data used in these models is simulated, as described in the "Simulating Input Dataset" and "Simulating Option Price" sections. The input variables include the interest rate (r), the ratio of strike price to stock price ($\frac{\text{Strike}}{\text{Stock}}$), time to maturity, and volatility. The output variable is the logarithm of the ratio of option price to stock price, denoted as $\log\left(\frac{\text{price}}{\text{stock\_price}}\right)$. The dataset consists of 42,000 samples. For model evaluation,The dataset was divided into training and test sets using a random sampling approach. This division was performed using the *traintestsplit* function from the *scikitlearn* library, which ensures that the samples are randomly shuffled before splitting. In this analysis, 80% of the data was allocated to the training set, and the remaining 20% was designated as the test set. A fixed random seed ($randomstate = 0$) was used to maintain consistency and reproducibility of the split across different runs. By randomly selecting samples for the test and training sets, we aim to ensure that both sets are representative of the entire dataset, minimizing biases that could arise from temporal or other systematic patterns in the data. The activation functions compared are ReLU, ELU, and Leaky ReLU. For each activation function, three metrics are presented: Training Time in seconds, Log Loss, and MSE. The ReLU activation function is recognized for its overall balance between computational efficiency and error metrics, maintaining a moderate profile across all categories. On the other hand, the ELU activation function excels in reducing both log loss and MSE, suggesting it might be the best choice for applications where prediction accuracy and generalization are paramount. Lastly, Leaky ReLU shows a significant advantage in minimizing training time, making it particularly suitable for scenarios where rapid model training is crucial.

Python Code Implementation: The following Python code demonstrates the final model:

```
model_ML = Sequential()
model_ML.add(Dense(30, input_dim=4))
model_ML.add(LeakyReLU())
model_ML.add(Dense(60))
model_ML.add(ELU())
model_ML.add(Dense(90))
model_ML.add(LeakyReLU())
model_ML.add(Dense(1))
```

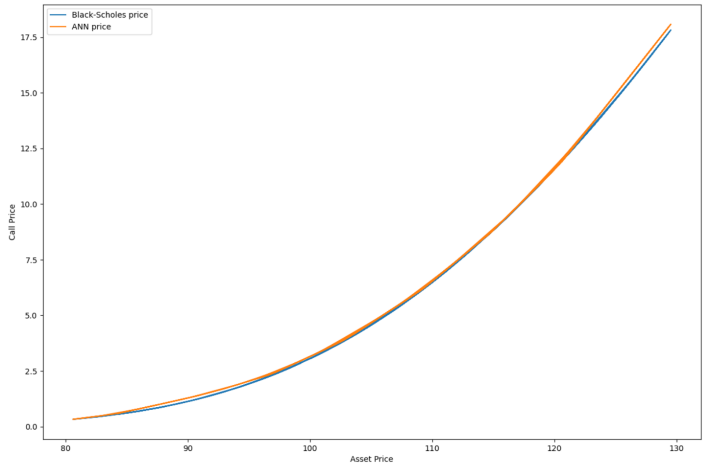


Figure 11: The simulated call option price calculated using the Black-Scholes compared to the option price predicted by an Artificial Neural Network (ANN) .

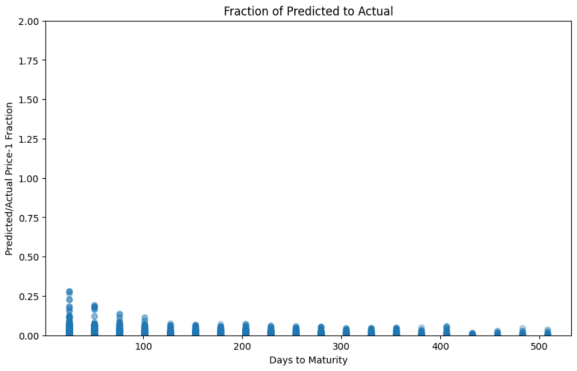
Figure 11 provided displays a comparison of simulated call option prices as calculated using two distinct methods: the Black-Scholes model and predictions made by an Artificial Neural Network (ANN). The blue line represents the B-S price, and the orange line represents the ANN price. The x-axis represents the asset price which is simulated by GBM, ranging from 80 to 130 $. The y-axis measures the call price by $. The two curves demonstrate the behavior of each pricing method across the range of asset prices. The graph demonstrates that both models closely align over the range of asset prices, indicating that the ANN is capable of accurately predicting the call option price in comparison to the traditional B-S model. The close alignment of the two curves suggests a high level of accuracy in the ANN's predictions, providing an alternative method for option pricing.

5.2.4 Data size

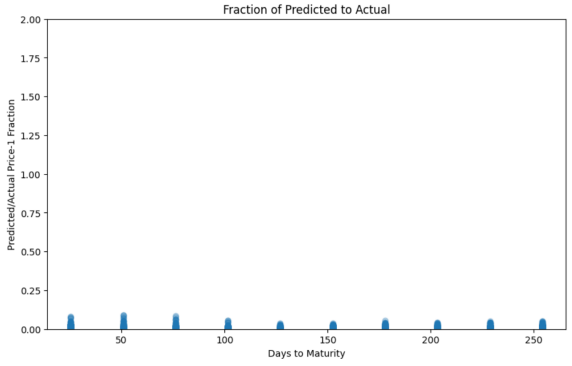
Here is the summary of the result in Table 6 and Figure 12 to improve the understanding of the model's performance across different dataset sizes and to realize the optimal data size when the target is $\log\left(\frac{\text{Call}}{\text{Stock}}\right)$.

Data size	Mean Fraction Error	Log Loss	MSE
84,000	-0.03	0.0.024	0.017
42,000	-0.01	0.001	0.017
21,000	-0.009	0.01	0.10

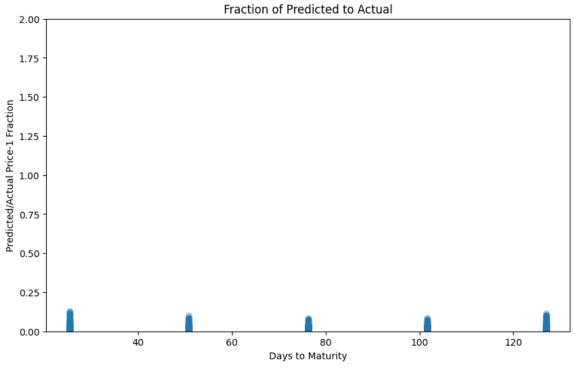
Table 6: Performance Metrics of Different data size



(a) The fractional error for $\log\left(\frac{\text{Call}}{\text{Stock}}\right)$ for a dataset size of 84,000.



(b) The fractional error for $\log\left(\frac{\text{Call}}{\text{Stock}}\right)$ for a dataset size of 42,000.



(c) The fractional error for $\log\left(\frac{\text{C}}{\text{S}}\right)$ for a dataset size of 21,000.

Figure 12: The fractional error for $\log\left(\frac{\text{Call}}{\text{Stock}}\right)$ for different dataset sizes.

The Figure 12 presented illustrates the fractional error for the logarithm of the ratio between the call option price and the stock price, denoted as $\log\left(\frac{\text{Call}}{\text{Stock}}\right)$, across different dataset sizes and plotted against

days to maturity. The x-axis represents the days to maturity, and the y-axis depicts the fractional error of the predicted to actual values.

In subplot 12a, the fractional error is shown for a dataset size of 84,000. The graph reveals that the fractional error remains relatively low and stable over the entire range of days to maturity, with minimal variations.

Subplot 12b displays the fractional error for a smaller dataset size of 42,000. This plot also demonstrates that the error remains low and consistent across different maturities, although the overall trend appears slightly less stable compared to the larger dataset.

Finally, subplot 12c depicts the fractional error for the smallest dataset size of 21,000. The results indicate a low fractional error but with even more noticeable variations compared to the datasets of 84,000 and 42,000. This suggests that the smallest dataset might introduce higher variability in predictions.

According to the results, the optimized size of dataset could be about 42.000.

5.3 Data description

An American-style dataset of historical call option data from TSLA is analyzed, which was carefully chosen. This implies that these options are not limited to exercising on the dates that they expire, namely April 19, 2024, May 3, 2024, May 17, 2024, and June 21, 2024. The website [3], which is renowned for its extensive financial data and analytical tools, provided the dataset used in this study. Through the examination of the anticipated IV at various strike prices, the call options included in the dataset are classified as out-of-the-money (OTM), at-the-money (ATM), and in-the-money (ITM). All possibilities are covered in order to gather important information regarding the mood and expectations of the market. The dataset is diversified to cover a wider range of market situations and responses, which improves the analysis.

The initial dataset comprised multiple CSV files, each representing options data with varying strike prices and expiration dates. The file columns are provided in the Table 7.

Column	**Description**
Time	The date of the record.
Open	The opening Option price in dollars on the given date.
High	The highest Option price in dollars on the given date.
Low	The lowest Option price in dollars on the given date.
Last	The last recorded Option price in dollars on the given date.
Change	The change in price from the previous closing Option price.
%Chg	The percentage change in price from the previous closing Option price.
Volume	The total number of shares or contracts traded on the given date.
Open Int	The open interest, represents the total number of outstanding contracts that are held by market participants at the end of the day.
IV	Implied Volatility, indicating market expectations of future volatility.
Delta	A measure of the sensitivity of the option's price to changes in the price of the underlying asset.
Gamma	The rate of change of delta relative to the price of the underlying asset.
Theta	The rate of decline in the value of an option due to the passage of time.
Vega	A measure of an option's sensitivity to changes in the volatility of the underlying asset.
Rho	The rate of change in the value of an option relative to a change in the interest rate.
Theo	Theoretical price of the option, calculated using an option pricing model.
Price	The Underlying stock price in dollars adjusted for certain conditions or assumptions.
Strike	The strike price in dollars of the option
Bid	The highest price a buyer is willing to pay for the security.
Ask	The lowest price a seller is willing to accept for the security.

Table 7: Columns and their Description

5.3.1 Data pre-processing

This section presents the most important phase of machine learning. This process aims to detect and correct (or remove) corrupt or inaccurate observations from the data set. The full data set consists of 50,287 observations. The observations steps ensure that the dataset is more representative of actual market conditions, enhancing the model's accuracy and reliability. To achieve this purpose we have to do some steps:

- Options with very low prices, specifically where the call price C_t is less than 0.1, have been excluded from analysis. This exclusion is due to the fact that options are often traded at integer values, which can cause significant deviations between observed and theoretical option prices when the option values are extremely low. By removing these low-value options, the accuracy of the analysis is enhanced [2].

- The data has been excluded with maturity (T-t) of more than about two years from the dataset in order to refine it and increase the performance accuracy of our pricing model. Due to the possibility of variances between theoretical and actual options pricing, this criterion eliminates options with extremely high time to maturity.

- To improve the accuracy of our pricing model's performance, we refine the dataset by removing data points where the trading volume is zero and by eliminating option data. Data points with zero volume represent assets that haven't been traded, which can introduce noise and inaccuracies into the model since they don't reflect real market activity.

- New columns are added to our data representing the data that is needed such as the days to Maturity column which calculates the difference between the 'Exchange Date' and a fixed maturity date. An additional risk-free rate of 0.05 was obtained from the 1-year maturity Triple-A rated government also the Moneyness columns which means that the ratio between underlying asset price and strike price have been added to describe the intrinsic value of an option in its current state. It indicates whether exercising an option would result in a profit or loss at the current price of the underlying asset.

- NaN Values in important columns such as 'Time', 'Last', 'Change', 'IV', 'Price ', and 'Strike' has been checked and removed to ensure model reliability.

- I categorized the "Time to Maturity" of options into distinct groups to facilitate analysis. Specifically, options with a time to maturity greater than 180 days were classified as long-term, those between 60 and 180 days were designated as medium-term, and those with less than 60 days were considered short-term. This classification aids in better understanding and analysis of the data, allowing for more nuanced insights into the behavior of options across different time horizons.

- We enhance our dataset by adding two crucial columns: 'log return' and 'cond vol'. These additions are pivotal in capturing the nuanced behavior of financial time series data. The 'log return' column represents the logarithmic returns of the underlying stock prices. log returns can better capture the continuous compounding effect in financial markets, providing a more accurate and interpret able measure of stock price changes. The 'cond vol' column captures the conditional volatility obtained from fitting a GARCH model to the log returns. Conditional volatility represents the expected volatility of returns at a given time, conditioned on past information. The GARCH model is specifically designed to model and forecast time-varying volatility, a common characteristic in financial time series data. By including conditional volatility as a feature, we effectively account for the clustering of volatility, the tendency for periods of high volatility to be followed by high volatility and periods of low volatility to be followed by low volatility. The inclusion of 'cond vol' allows our models to adapt to the changing risk environment over time, enhancing their predictive power and robustness. By adding the log return and 'cond vol' columns, we enrich our dataset with valuable information that captures both the return dynamics and the volatility structure of the underlying financial instruments.

After removing all the non-representative observations, the data set consists of 31,827 samples which according to the simulation is a reasonable number to train the neural network examined in this thesis. Figures 13 summarize the characteristics of the data set.

	Time	option_price	Volume	IV	underlying_stockprice	Strike	r	Maturity	C/S	Monyness	K/S	log_return	cond_vol
count	31827	31827.000000	31827.000000	31827.000000	31827.000000	31827.000000	3.182700e+04	31827.000000	31827.000000	327.000000	31827.000000	31827.000000	31827.000000
mean	2023-11-02 13:47:01.566594560	26.766472	494.989851	0.552730	208.355209	285.390235	5.160000e-02	0.839542	0.125124	0.874560	1.391830	-0.000009	0.004011
min	2022-10-26 00:00:00	0.100000	1.000000	0.111900	108.100000	50.000000	5.160000e-02	0.000000	0.000457	0.144133	0.177004	-0.130590	0.002892
25%	2023-08-23 00:00:00	2.500000	6.000000	0.488900	179.830000	190.000000	5.160000e-02	0.373016	0.012437	0.591619	0.939236	0.000000	0.003981
50%	2023-12-05 00:00:00	13.650000	29.000000	0.520700	202.640000	260.000000	5.160000e-02	0.674603	0.065394	0.788825	1.267708	0.000000	0.003981
75%	2024-02-20 00:00:00	40.550000	150.000000	0.575000	239.740000	350.000000	5.160000e-02	1.202381	0.195103	1.064696	1.690277	0.000000	0.003981
max	2024-06-11 00:00:00	232.000000	224714.000000	7.632400	293.340000	750.000000	5.160000e-02	2.396825	0.857955	5.649600	6.938020	0.142427	0.014789
std	NaN	32.430714	4249.107492	0.190091	36.174820	129.380676	6.807162e-15	0.610085	0.145518	0.425017	0.660341	0.004020	0.000270

Figure 13: The description of TSLA data between 2023-2024.

Figure 13 The table presents a detailed summary of data you can find count, mean, min, 25%, 50%, 75%, max, and std of each column, according to the table dataset containing 31,827 records from October 26, 2022, to June 11, 2024. *option_price* variable has an average of 26.77 with a minimum of 0.10 and a maximum of 232. The volume of options traded has an average of 494.99, but it varies widely, with a minimum of 1 and a maximum of 224,714. The implied volatility (IV) has an average of 0.553, ranging from 0.112 to 7.63, indicating varied market expectations. The underlying stock price averages 208.36, with values between 108.10 and 293.34. The *Strike* price, a critical option parameter, averages 285.39, with a wide range from 50 to 750. The risk-free rate (r) is consistently 5.16%, and the *Maturity* of the options averages 0.84 years, ranging from immediate expiry to nearly 2.4 years.

Figure 14: TSLA underlying stock price between 2023-2024.

Figure 14 shows the underlying stock price of TSLA in the period between 2022 and 2024. The range of the underlying price per one share of TSLA is about between 108 dollars and 293 dollars. The plot shows significant fluctuations and multiple peaks and valleys. Initially, there is a downward trend until early 2023, followed by a recovery and subsequent volatility throughout the year. The price reaches a high point in mid-2023, then experiences fluctuations with a general downward trend into 2024. This indicates a volatile market with multiple short-term fluctuations and overall long-term trends that could reflect broader economic factors or company-specific events.

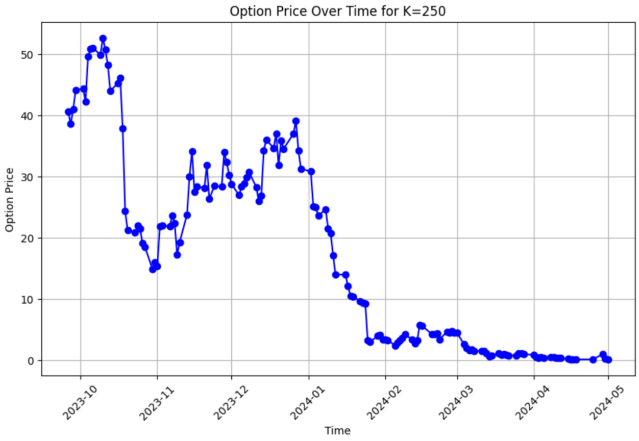
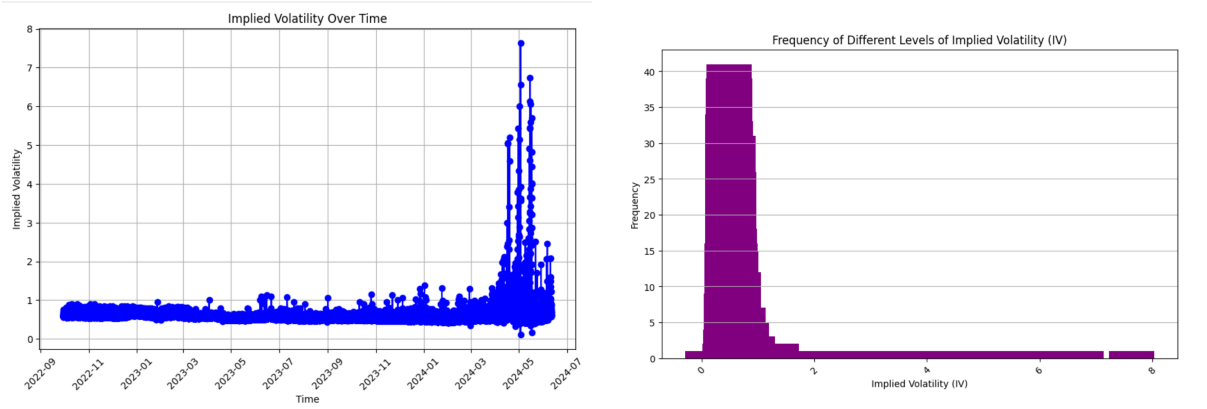


Figure 15: TSLA Option Price Over Time for K=250 with an Expiration Date of 2024/05/17

Figure 15 for a strike price (K) of 250 with an Expiration Date of 2024/05/17 shows the option price from late 2023 to mid-2024. Initially, the option price fluctuates with notable peaks, but it begins a downward trend around early 2024, eventually stabilizing at a lower level. This decrease could reflect a drop in implied volatility, underlying asset price movements, or approaching expiration, reducing the option's time value. The overall trend indicates diminishing expectations of significant price movements in the underlying asset as the expiration date approaches.

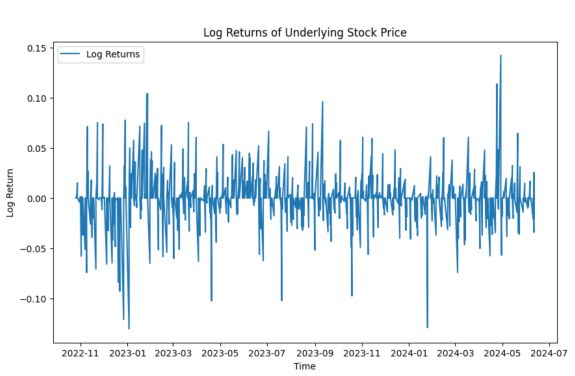


(a) TSLA Implied Volatility Over Time 2022-2024.

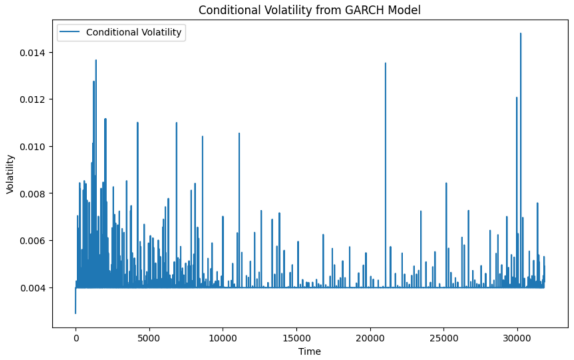
(b) TSLA Frequency of Different Levels of Implied Volatility.

Figure 16: TSLA Implied Volatility.

Figure 16 show implied volatility over time, 16a and 16b reveal a significant increase in volatility beginning around early 2024, with a sharp spike leading up to the expiration date between April and June 2024. This suggests that market uncertainty or expected fluctuations in the option price increased dramatically as the expiration approached. The earlier period shows relatively stable and low implied volatility, indicating consistent market conditions until the sudden rise.



(a) TSLA Log Returns of Underlying Stock Price.



(b) TSLA Conditional Volatility from GARCH Model.

Figure 17: Historical Volatility.

Figure 17a illustrates the log returns of an underlying stock price over a period from November 2022 to July 2024. Log returns, which are used to measure the relative change in stock price, are plotted on the vertical axis, while the time is represented on the horizontal axis. The time series data exhibit considerable volatility, with log returns fluctuating significantly over the observed period. This volatility is indicative of the stock's varying performance, with frequent spikes and drops in log returns.

The graph 17b illustrates the conditional volatility of a time series as estimated by a GARCH model. The vertical axis represents volatility, while the horizontal axis denotes time. The plot indicates varying levels of volatility over the observed period, with noticeable spikes representing periods of high volatility and more stable periods where volatility is relatively low. Initially, there are frequent and significant spikes, suggesting periods of intense market activity or instability. As time progresses, the frequency and magnitude of these spikes tend to decrease, indicating a reduction in volatility. However, occasional large spikes persist throughout the timeline, reflecting sporadic market shocks or events.

5.4 Network architecture and Model Development

Figure 18 outlines a comprehensive process for developing and evaluating a neural network model, divided into four main stages: Data Preparation, Data Preprocessing, Neural Network, and Strategy Analysis. The Data Preparation phase begins with importing data, defining features, dropping NaN values, and defining the target variable. For model evaluation,The dataset was divided into training and test sets using a random sampling approach. This division was performed using the *traintestsplit* function from the *scikitlearn* library, which ensures that the samples are randomly shuffled before splitting. In this analysis, 80% of the data was allocated to the training set, and the remaining 20% was designated as the test set. A fixed random seed ($randomstate = 0$) was used to maintain consistency and reproducibility of the split across different runs. The X_train dataset serves as the model's input for training, while y_train represents the target variable for fitting the model. For prediction, X_test is used as the input, and y_test provides the corresponding output. The Neural Network phase involves designing the model architecture, compiling the model, and training it using the training dataset. Finally, in the Strategy Analysis phase, the model is evaluated for its performance, and predictions are made using the test dataset. This workflow ensures a structured approach to handling data, building, and validating the neural network model, leading to reliable and actionable predictions.

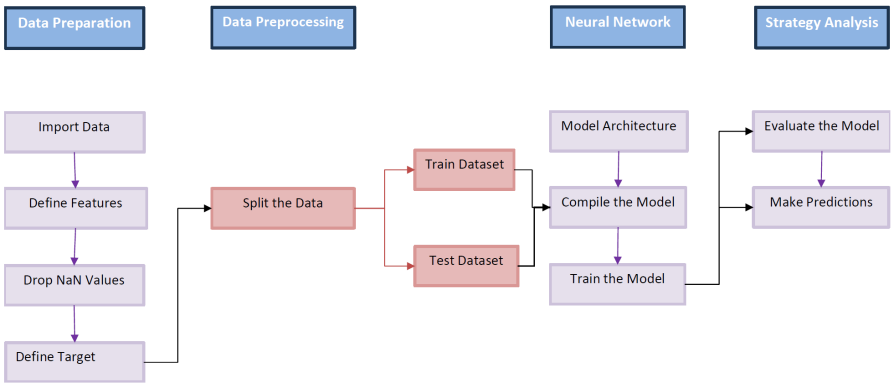


Figure 18: Neural Network Model.

The Artificial Neural Networks implemented in this thesis are MLP, LSTM, and GRU. The input parameters used consist of: the underlying asset's price series St, the strike price K, the time to maturity T-t, the risk-free interest rate r, the implied volatility, and Both including and excluding GARCH volatility. The network's goal is to approximate the market's option pricing function f(x) and log (f(x))such that:

$$C_t = f\{S, K, T-t, r, IV\} \tag{29}$$

$$C_t = f\{S, K, T-t, r, IV, GARCH\} \tag{30}$$

$$log(C_t) = f\{S, K, T-t, r, IV\} \tag{31}$$

$$log(C_t) = f\{S, K, T-t, r, IV, GARCH\} \tag{32}$$

The pricing function f(x) that is used to calculate the price of an option C_t is not affected by the return of the underlying asset, which is independent of the level of the underlying asset's price St. The price is homogeneous of degree one in the strike price K as well as the underlying asset S_t, as the theorem shows. Huchinson et al. (1994) [22] standardize their data by dividing their inputs by the stock price, following Merton's example. As a result, the equation above becomes:

$$\frac{C_t}{S_t} = f\left\{1, \frac{K}{S_t}, T-t, , IV\right\} \tag{33}$$

$$\frac{C_t}{S_t} = f\left\{1, \frac{K}{S_t}, T-t, , IV, Garch\right\} \tag{34}$$

$$(\log\left(\frac{C}{S}\right)) = f\left\{1, \frac{K}{S_t}, T-t, IV\right\} \tag{35}$$

$$(\log\left(\frac{C}{S}\right)) = f\left\{1, \frac{K}{S_t}, T-t, , IV, Garch\right\} \tag{36}$$

Two sets of data are created: 80% for training with the size of 25,461 and 20% for testing with the size of 6366. Using this method, observations are taken one at a time from the data set and then selected, then returned to the data set. This makes it possible for a specific observation to appear more than once in the training set. The neural network model's performance is assessed by training it on the train test and assessing it on a test set that isn't part of the train test.

5.4.1 MLP

For finding a better LSTM model different Activation Functions have been used in a model training process to make a model with ReLU, ELU, and Leaky Relu to compare how different activation functions affect training efficiency and model performance. According to the result in Table 8, all three activation functions were trained for 150 epochs and 16 catch sizes. In terms of training time, ReLU took the longest, followed by ELU, and Leaky ReLU was the fastest. The Log Loss values indicate that Leaky ReLU achieved the lowest value, suggesting better classification performance compared to ReLU and ELU, which had Log Loss values, respectively. Similarly, the MSE values show that ELU had the lowest MSE, followed by Leaky ReLU and ReLU, indicating that ELU had the best performance in terms of regression accuracy.

Activation	Training Time(s)	Log Loss	MSE
relu	601.08	0.0421	0.0312
elu	514.40	0.0338	0.0117
leaky relu	450.45	0.0160	0.0236

Table 8: Performance Metrics of Different Activation Functions

According to the this and the result that we had in simulation part, the neural network model has been designed with four layers, including three hidden layers and one output layer. The first hidden layer is a Dense layer with 30 neurons, followed by a LeakyReLU activation function. The second hidden layer is a Dense layer with 60 neurons, followed by an ELU activation function. The third hidden layer is a Dense layer with 90 neurons, followed by another LeakyReLU activation function. The output layer is a Dense layer with a single neuron, indicating that the model is designed for a regression task. Overall, the network architecture effectively combines Dense layers with different activation functions to capture complex patterns in the data, and the model is compiled using the Adam optimizer with mean squared error as the loss function to optimize its performance.

Table 9 is the summary of the model that details the architecture, outlining each layer's type, output shape, and the number of parameters. The first layer, labeled "dense 30" (Dense), has an output shape of (None, 30) and consists of 150 parameters. Following this, the LeakyReLU layer also outputs (None, 30) but does not add any parameters. The third layer, "dense 31" (Dense), has an output shape of (None, 60) with 1,860 parameters. This is followed by the "elu 3" (ELU) activation layer, which outputs (None, 60) and similarly does not contribute additional parameters. The subsequent "dense 32" (Dense) layer outputs (None, 90) and includes 5,490 parameters. The second LeakyReLU activation layer, which outputs (None, 90), does not add parameters. Finally, the "dense 33" (Dense) layer, which outputs a single value (None, 1), contains 91 parameters. This breakdown of layers and parameters highlights the structure and complexity of the neural network model.

Layer (type)	Output Shape	Param
dense_30 (Dense)	(None, 30)	150
leaky_re_lu_8 (LeakyReLU)	(None, 30)	0
dense_31 (Dense)	(None, 60)	1,860
elu_3 (ELU)	(None, 60)	0
dense_32 (Dense)	(None, 90)	5,490
leaky_re_lu_9 (LeakyReLU)	(None, 90)	0
dense_33 (Dense)	(None, 1)	91

Table 9: Model Summary

After choosing the Activation Functions and the number of layers and neurons, the choice of Epochs and Bach size have been clarified.

Epochs	Batch Size	Training Time(S)	Log Loss	MSE
150	16	601.08	0.0421	0.0312
150	16	514.40	0.0338	0.0117
150	160	450.45	0.0160	0.0236
50	10	214.36	0.0400	0.0063
50	15	154.30	0.0483	0.0265
50	30	83.73	0.0825	0.0097
50	40	60.22	0.0306	0.0118
100	10	447.00	0.0381	0.0383
100	15	303.00	0.0492	0.0036
100	30	161.08	0.0386	0.0025
100	40	117.98	0.0335	0.0033
150	10	663.50	0.0230	0.0266
150	15	2786.52	0.0106	0.0137
150	30	223.23	0.0326	0.0059
150	40	177.32	0.0133	0.0115

Table 10: Performance Metrics of Different Epochs and Bach size

Table 10 The table presents a summary of performance metrics for different combinations of epochs and batch sizes used during the training of a machine learning model. These metrics include training time, log loss, and mean squared error (MSE), providing insights into how each configuration affects the model's learning efficiency and predictive accuracy.

In reviewing the performance data, it becomes evident that the choice of the best epoch and batch size configuration depends on a balance between computational efficiency (training time) and model performance (log loss and MSE). A lower log loss and MSE indicate better predictive accuracy, while a shorter training time suggests greater computational efficiency.

From the data, the configuration of 100 epochs with a batch size of 15 stands out as the most effective setup. This configuration achieves a significantly lower log loss compared to other settings, suggesting better model accuracy without an excessive increase in training time. The MSE value under this setup is also among the lower scores, further confirming its effectiveness in producing a robust model.

Python Code Implementation: The following Python code demonstrates the model:

```
model_ML = Sequential()
model_ML.add(Dense(30, input_dim=4))
model_ML.add(LeakyReLU())
model_ML.add(Dense(60))
model_ML.add(ELU())
model_ML.add(Dense(90))
model_ML.add(LeakyReLU())
model_ML.add(Dense(1))
model_ML.compile(loss='mean_squared_error', optimizer='adam')
history_model_ML = model_ML.fit(X_train, y_train, epochs=100, batch_size=15,verbose=0)
```

5.4.2 LSTM

For finding a better LSTM model different architectures have been applied. The final model begins with an LSTM layer consisting of 100 units, set to return sequences, and configured to handle input shapes of (1, 4). This is followed by a second LSTM layer, also with 100 units, but this time not returning sequences, effectively summarizing the information. A Dropout layer with a rate of 0.2 is added to prevent overfitting by randomly setting 20% of the input units to zero during training. The network then includes a Dense layer with 50 units and an ELU activation function to introduce non-linearity and enable the model to learn more complex patterns. The final layer is a Dense layer with a single unit, making it suitable for regression tasks. The model is compiled with the Adam optimizer, using a learning rate of 0.001, and the mean squared error as the loss function to measure prediction accuracy. The model is trained on the provided training data for 150 epochs with a batch size of 16, as indicated by the fit method, which captures the training process in LSTM history.

Attribute	Third Model	Second Model	First Model
Number of Layers	5	5	4
LSTM Layers	2 LSTM (100 units each)	2 LSTM (100 units each)	2 LSTM (50 units each)
Dense Layers	2 Dense (50 units, 1 unit)	2 Dense (50 units, 1 unit)	1 Dense (1 unit)
Dropout Layers	1 Dropout (0.2)	1 Dropout (0.3)	1 Dropout (0.2)
Activation Function	ELU	ReLU	non
Training Time (s)	320.825	340.738	302.269
Last Log Loss	0.0000485370	0.0000486713	0.0001425965
R-squared	0.999	0.997	0.998
RMSE	0.0052425483	0.0074433834	0.0062721198
MAE	0.0024172009	0.0053625320	0.0033799996
MSE	0.0000274843	0.0000554040	0.0000393395

Table 11: Comparison of Two LSTM Models

Table 11 illustrates a detailed comparison of three LSTM-based machine learning models, evaluating their architectural components and performance metrics. Each model comprises different configurations of LSTM layers, dense layers, dropout layers, and activation functions, which significantly influence their performance outcomes such as training time, log loss, R-squared, RMSE, MAE, and MSE.

The first model, with the fewest layers, exhibits the shortest training time and achieves the lowest values in MSE and log loss, suggesting high predictive accuracy and efficiency. This model's architecture lacks an activation function in the dense layer, which could contribute to its streamlined efficiency. Its superior R-squared value further underscores its robustness in fitting the data. The second model incorporates a slightly higher dropout rate and uses the ReLU activation function. Despite slightly longer training times, this model shows commendable performance, although it does not surpass the first model in the lowest metrics of log loss and MSE. The third model, similar in structure to the second but using the ELU activation function, demonstrates the longest training time. it maintains high accuracy, as reflected in its R-squared and RMSE, and the log loss. The third model emerges as the most efficient and effective in this set, balancing computational efficiency with high accuracy, Therefore the final model is :

Python Code Implementation: The following Python code demonstrates the model:

```
def create_improved_lstm_model():
    model = Sequential()
    model.add(LSTM(units=100, return_sequences=True, input_shape=(1, 4)))
    model.add(LSTM(units=100, return_sequences=False))
    model.add(Dropout(0.2))
    model.add(Dense(units=50, activation='elu'))
    model.add(Dense(units=1))
    model.compile(optimizer=Adam(learning_rate=0.001), loss='mean_squared_error')
    return model
```

5.4.3 GRU

For the GRU model, we utilize the same parameters that were previously employed for the LSTM model except the dropout rate since it shows more overfilling in the log loss plot. The model begins with a GRU layer consisting of 100 units, set to return sequences, and configured to handle input shapes of (1, 4). This is followed by a second GRU layer, also with 100 units, but this time not returning sequences, effectively summarizing the information. A Dropout layer with a rate of 0.3 is added to prevent overfitting. The network then includes a Dense layer with 50 units and an ELU activation function to introduce non-linearity and enable the model to learn more complex patterns. The final layer is a Dense layer with a single unit, making it suitable for regression tasks. The model is compiled with the Adam optimizer, using a learning rate of 0.001, and the mean squared error as the loss function to measure prediction accuracy. The model is trained on the provided training data for 150 epochs with a batch size of 16, as indicated by the fit method, which captures the training process in GRU history.

Python Code Implementation: The following Python code demonstrates the model:

```
def create_gru_model():
    model = Sequential()
    model.add(GRU(units=100, return_sequences=True, input_shape=(1, 4)))
    model.add(GRU(units=100, return_sequences=False))
    model.add(Dropout(0.3))
    model.add(Dense(units=50, activation='elu'))
    model.add(Dense(units=1))
    model.compile(optimizer=Adam(learning_rate=0.001), loss='mean_squared_error')
    return model
```

5.5 performance metrics

5.5.1 Volatility performance

In this analysis, the aim is to incorporate volatility clustering into predictive models by utilizing a GARCH model to estimate conditional volatility. Volatility clustering refers to the phenomenon where high-volatility events tend to cluster together, indicating that the current level of volatility is informative about future volatility. We start by preprocessing the data to calculate the log-returns of the underlying stock prices, as these returns are a fundamental input for the GARCH model. The GARCH model is then fitted to the log returns to capture the time-varying volatility. The fitted model provides us with conditional volatility, which is a measure of the expected future volatility based on past information.

To calculate log returns and fit a GARCH(1,1) model, we begin by computing the log-returns of the underlying stock prices. The log return for each period is determined by taking the natural logarithm of the ratio of consecutive stock prices. This calculation can be expressed mathematically as:

$$\text{log\_return}_t = \ln\left(\frac{\text{price}_t}{\text{price}_{t-1}}\right)$$

where price_t is the stock price at time t and price_{t-1} is the stock price at the previous time period. Specifically, we create a new column in dataset to store the log returns, and the use of the `shift()` function allows you to align each price with its preceding price, ensuring the correct calculation of returns. After calculating log returns, any rows containing `NaN` values, which result from the shifting operation at the beginning of the data, are removed.

Once the log returns are calculated and cleaned, we proceed to fit a GARCH(1,1) model. The GARCH(1,1) model is specified with one lag each for both the autoregressive and moving average terms(check Section 3.6 to know how it will be calculate). In the context of fitting the model, the returns array is extracted from the dataset to serve as the input for the GARCH model.

The `arch` library is typically used to implement the GARCH model. You instantiate a model by specifying the type of volatility model (`'Garch'`) and the orders for the lagged terms, with $p = 1$ and $q = 1$ representing the GARCH(1,1) configuration. After defining the model, we fit it to the data, which estimates the model parameters based on the log return data.

Upon fitting the model, the conditional volatility is calculated and stored as a new column in the dataset. Conditional volatility represents the estimated volatility at each time point, conditional on past returns and volatilities, providing insights into the dynamic behavior of volatility over time.

Next, conditional volatility has been included along with other features to train models.

Python Code Implementation: The following Python code:

```
#Calculate log-returns of the underlying stock prices
data['log_return'] = np.log(data['underlying_stockprice'] /data['underlying_stockprice'].shift(1))
data = data.dropna()
# Fit a GARCH(1,1) model
returns = data['log_return'].values
model = arch_model(returns, vol='Garch', p=1, q=1)
garch_fit = model.fit(disp='off')
data['cond_vol'] = garch_fit.conditional_volatility
```

5.5.2 Log performance

Sophisticated performance has been employed as a metric to enhance the accuracy and efficacy of my machine-learning models. This metric, denoted as the logarithm of the target variable (Log(C/S)), where C represents the call option price and S represents the stock price, plays a pivotal role in capturing the intricate dynamics between options and their underlying stocks.

The choice of using the logarithm of the target variable stems from its ability to stabilize variance and normalize the distribution of data, which is often skewed in financial datasets. Transforming the target variable using the natural logarithm ensures that extreme values are compressed, thereby reducing the potential impact of outliers on the model's performance. This transformation is particularly beneficial in financial modeling, where prices can exhibit significant volatility and unpredictability.

Moreover, the logarithmic transformation aids in achieving linearity, a crucial assumption for many machine learning algorithms. The relationship between call option prices and stock prices is inherently non-linear, and applying the logarithm helps in approximating this relationship more linearly, facilitating better model training and prediction.

Incorporating Log(C/S) as a performance metric also aligns with the principle of dimensional consistency in financial modeling. Since both the call option price and stock price are measured in monetary units, taking their ratio and subsequently the logarithm ensures that the resulting metric is dimensionless, allowing for more meaningful comparisons and interpretations.

6 Results

In this chapter, the results obtained from the empirical study are presented. We discuss different conditions under which the models were evaluated:

1. The performance of different models: MLP, LSTM, GRU.
2. item The performance of models with $\frac{C}{S}$ (call option/stock price) and $\log\left(\frac{C}{S}\right)$ (logarithm of call option/stock price) as the target variable.
3. The performance of models incorporating GARCH volatility versus those without it as a feature.

6.1 Performance of different models in predicting call option price over the stock price, incorporating GARCH volatility versus those without it as a feature.

GARCH model	Model	Last log loss	MSE	RMSE	MAE	R^2	Time(S)
Without GARCH	MLP	0.00003810	0.00004319	0.00657251	0.0044427307	0.998	97.03
With GARCH	MLP	0.00003857	0.00003249	0.00570003	0.0029176797	0.998	233.72
Without GARCH	LSTM	0.00004867	0.00005540	0.00744338	0.005362532	0.997	340.73
With GARCH	LSTM	0.00004917	0.00005495	0.00741299	0.00441310	0.997	370.93
Without GARCH	GRU	0.00004596	0.00005534	0.00743975	0.004917299	0.997	311.14
With GARCH	GRU	0.00004598	0.00005495	0.00741299	0.00441310	0.997	349.71

Table 12: Comparison of Neural Network Model when C/S is target

Table 12 provides a detailed analysis of the performance of various neural network models, MLP, LSTM, and GRU models, both with and without the integration of the GARCH volatility as part of features, focusing on their performance metrics for a specific dataset where 'C/S' is the target variable. Models are evaluated based on several metrics, namely Last log loss, Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), R-squared (R^2), and computational time measured in seconds.

Across all models, the incorporation of the GARCH methodology results in improvements or stabilizations in the performance metrics, suggesting better model accuracy in probabilistic terms.

The computational time for each model increases when GARCH is incorporated, reflecting the additional computational overhead required for implementing the GARCH method. This increase is most notable in the MLP model, where the time nearly triples, suggesting that the integration of GARCH into MLP requires significant additional processing. Meanwhile, the LSTM and GRU models also show increased computational times, but the increases are less dramatic compared to MLP.

Overall, the integration of GARCH into these neural network models enhances their predictive accuracy at the cost of increased computational time. This trade-off between accuracy and efficiency is crucial in scenarios where either metric is a priority, depending on the specific application and requirements of the task.

GARCH model	Model	Min	Max	Mean	Std
Without GARCH	MLP	-2.89	13.27	0.05	0.35
With GARCH	MLP	-0.5	0.15	-0.01	0.02
Without GARCH	LSTM	-6.68	11.08	-0.09	0.39
With GARCH	LSTM	-0.37	0.77	0.01	0.02
Without GARCH	GRU	-0.31	16.74	0.18	0.59
With GARCH	GRU	-1.10	12.75	0.23	0.57

Table 13: Comparison fraction error of Neural Network Model when C/S is target

Table13 provides a comparison of the fractional error of different neural network models when the target variable is the ratio of the option call price (C) to the stock price (S), denoted as S/C. The table evaluates the performance of three models: MLP, LSTM, and GRU, both with and without the inclusion

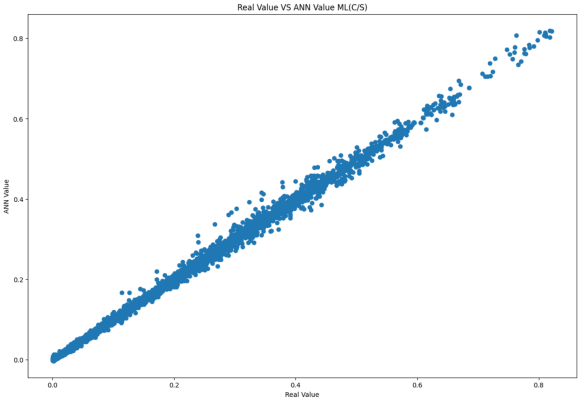
of a GARCH as a feature. The metrics considered for evaluation are the minimum, maximum, mean, and standard deviation of the fractional error.

For the MLP and LSTM model, the implementation of GARCH narrows the span of error, indicating a drastic improvement in model stability. The mean error approaches zero, which an average prediction closer to actual values and a reduction in bias. Similarly, the standard deviation, a measure of error variability, is markedly lower, illustrating that predictions are not only more accurate on average but also more reliable, with less variability in error.

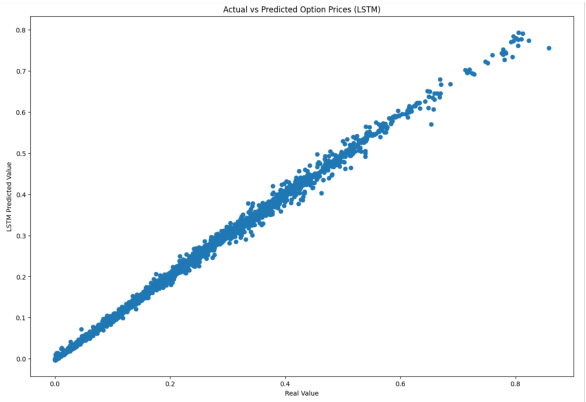
The GRU models exhibit similar trends in Std. The addition of GARCH results in a contraction in the range of errors and a reduction in the standard deviation, indicating enhanced prediction consistency.

In summary, the incorporation of the GARCH model into the MLP, LSTM, and GRU models results in a marked improvement in prediction accuracy and stability as evidenced by a reduction in the range of errors.

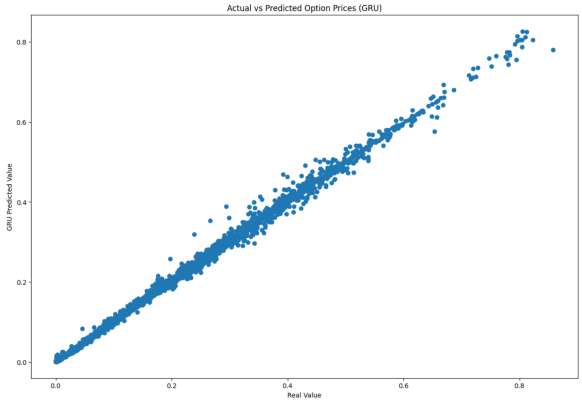
- Figures that illustrating the performance of different models in predicting call option price over the stock price, excluding GARCH as a feature.



(a) Actual vs Predicted MLP(C/S)



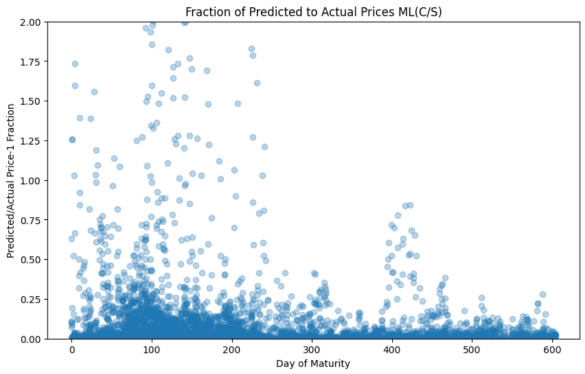
(b) Actual vs Predicted LSTM(C/S)



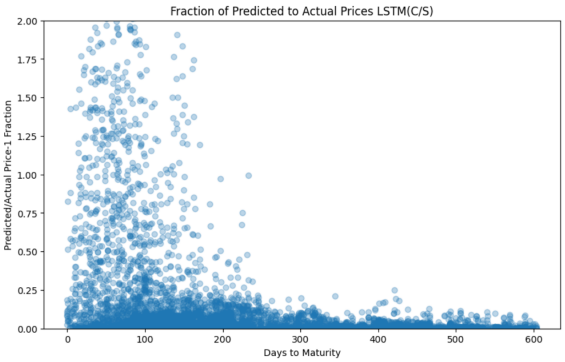
(c) Actual vs Predicted GRU(C/S)

Figure 19: Actual vs Predicted of $\frac{C}{S}$ for different models, excluding GARCH in features.

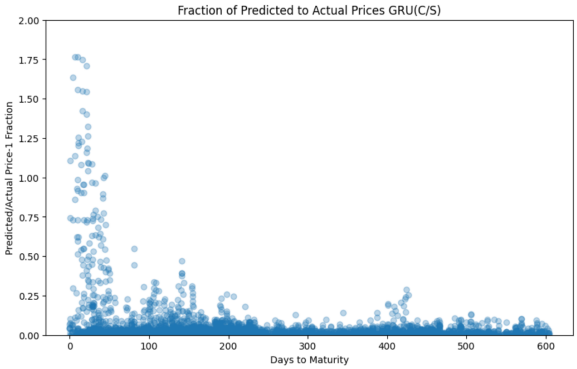
Figure 19 illustrates the actual versus predicted option prices for the MLP, LSTM, and GRu models that show a strong linear relationship between the real values and the predicted values, indicating high accuracy in the model's predictions.



(a) Fraction error Using MLP Model over days to maturity.



(b) Fraction error Using LSTM Model over days to maturity.

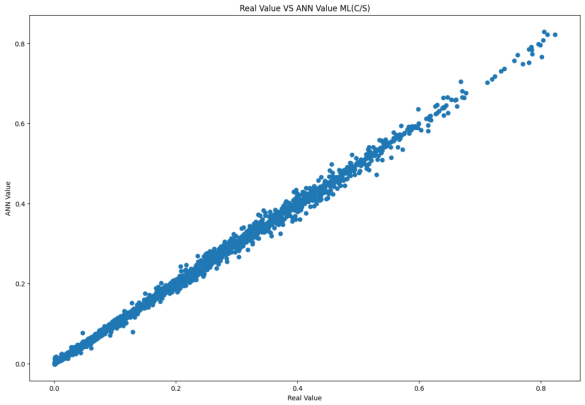


(c) Fraction error Using GRU Model over days to maturity.

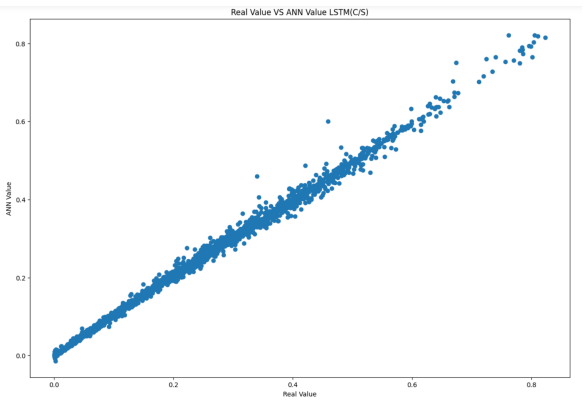
Figure 20: Fraction error different Models over days to maturity, $\frac{C}{S}$ as target, excluding GARCH in features.

Figure 20 depicts the fraction of predicted to actual prices minus one for the different models over days to maturity when $\frac{Calloption}{Stockprice}$ is target, without incorporating GARCH in features. Subfigure 20a, 20b, and 20c demonstrates high variability and errors in the initial days, with predictions becoming more consistent as maturity extends and becomes smaller in MLP, LSTM, and GRU models.

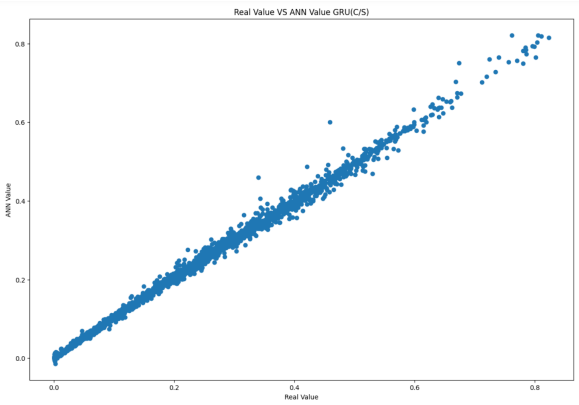
- Figures show The performance of different models in predicting call option price over the stock price incorporating GARCH in features.



(a) Actual vs Predicted MLP(C/S)



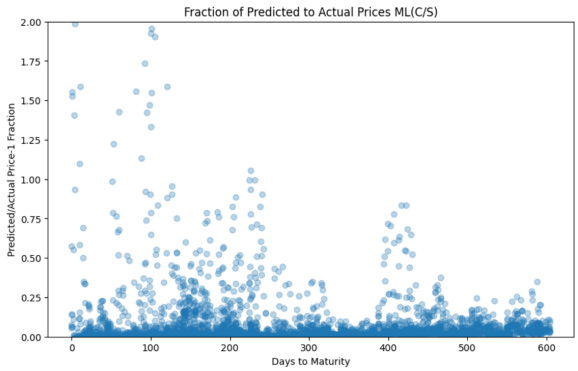
(b) Actual vs Predicted LSTM(C/S)



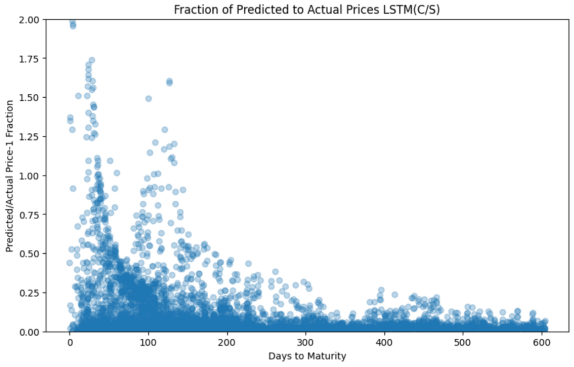
(c) Actual vs Predicted GRU(C/S)

Figure 21: Actual vs Predicted of $\frac{C}{s}$ for different models, incorporating GARCH in features.

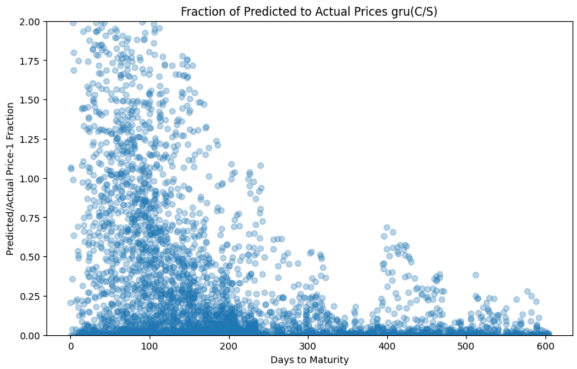
Figure 21 illustrates the actual versus predicted values for the same set of MLP, LSTM, and GRU models that show the actual versus predicted values, where the points closely align along the diagonal, indicating high predictive accuracy.



(a) Fraction error Using MLP Model over days to maturity.



(b) Fraction error Using LSTM Model over days to maturity.



(c) Fraction error Using GRU Model over days to maturity.

Figure 22: Fraction error Using different Models over days to maturity, $\frac{C}{S}$ as target, incorporating GARCH in features.

Figure 22 shows the fraction of predicted to actual prices minus one for the different models over days to maturity when $\frac{Calloption}{Stockprice}$ is a target, incorporating GARCH in features. Subfigure 20a, 20b, and 20c demonstrates high variability and errors in the initial days, with predictions becoming more consistent as maturity extends and becomes smaller in MLP, LSTM, and GRU models.

6.2 Performance of different models in predicting the logarithm of call option price over the stock price, incorporating GARCH volatility versus those without it as a feature.

GARCH model	Model	Last log loss	MSE	RMSE	MAE	R^2	Time(S)
Without GARCH	MLP	0.004387	0.002937	0.054199	0.031008	0.999	212.19
With GARCH	MLP	0.004182	0.003373	0.058081	0.037200	0.999	372.57
Without GARCH	LSTM	0.002255	0.002301	0.047969	0.029447	0.999	750.03
With GARCH	LSTM	0.004129	0.006465	0.080409	0.048214	0.998	870.93
Without GARCH	GRU	0.003781	0.241651	0.376399	0.491580	0.928	725.86
With GARCH	GRU	0.003902	0.518758	0.720248	0.572516	0.997	849.71

Table 14: Comparison of Neural Network Models in predicting the logarithm of call option price over the stock price.

Table 14 provides a detailed comparison of MLP, LSTM, and GRU models in predicting the logarithm of the call option price over the stock price, denoted as log(C/S). The models are evaluated with and without the integration of the GARCH in features.

The columns in the table include key performance metrics such as Last log loss, Mean Squared Error (MSE), Root Mean Square Error (RMSE), Mean Absolute Error (MAE), R-squared (R^2), and

the computation time in seconds (Time(S)). These metrics provide a comprehensive assessment of the models' accuracy and efficiency.

The integration of the GARCH method is shown to have varying impacts on these models but generally makes not much difference in performance accuracy but a notable increase in computational time, suggesting a trade-off between accuracy and efficiency.

Overall, Each model displays high levels of accuracy in predictions, with very high R-squared values, indicating that a substantial proportion of variance in the target variable is captured by the models with or without incorporation of GARCH. However, the incorporation of GARCH comes at the cost of increased computational resources and time.

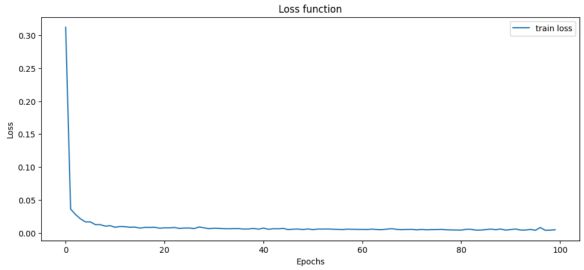
GARCH model	**Model**	**Min**	**Max**	**Mean**	**Std**
Without GARCH	MLP	-0.38	0.33	-0.003	0.023
With GARCH	MLP	-0.55	0.15	-0.01	0.02
Without GARCH	LSTM	-0.26	0.70	0.0002	0.024
With GARCH	LSTM	-0.37	0.77	0.010	0.027
Without GARCH	GRU	-0.15	0.75	0.002	0.023
With GARCH	GRU	-0.21	0.86	-0.004	0.032

Table 15: Comparison fraction error of Neural Network Models in predicting the logarithm of call option price over the stock price.

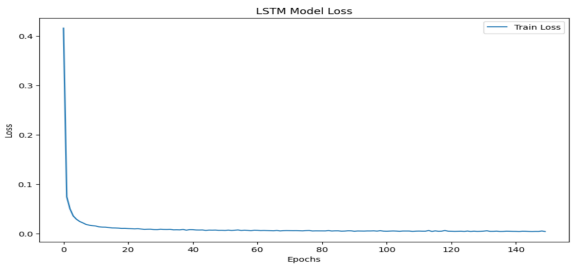
Table 15 provides a comparison of the fractional error of Neural Network models when the target is the logarithm of the ratio of the option call price (C) to the stock price (S), denoted as log(C/S). It evaluates three models: MLP, LSTM, and GRU, both with and without the incorporation of a GARCH in features. The performance metrics considered are the minimum, maximum, mean, and standard deviation (Std) of the fractional error.

The data indicates that the use of GARCH does not consistently lead to a narrowing of the range of prediction errors across all models. Each neural network model shows notable high accuracy in predicting and low error close to zero. While the MLP and GRU models perform well, the LSTM models show significant improvements in error reduction and consistency. This highlights the effectiveness of LSTM models in managing the inherent volatility and non-linear patterns often present in financial time series data, making them valuable for predictive analytics in finance-related applications.

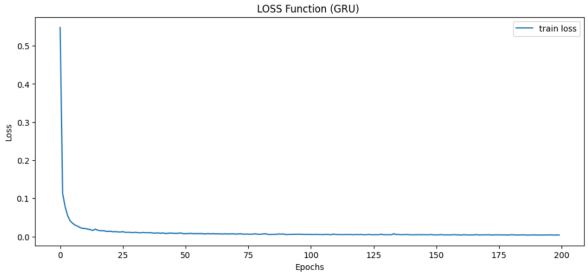
- Figures show the performance of models in predicting the logarithm of call option price over the stock price. Notably, this analysis is conducted without incorporating GARCH as one of the features.



(a) The Loss function of MLP(log(C/S))



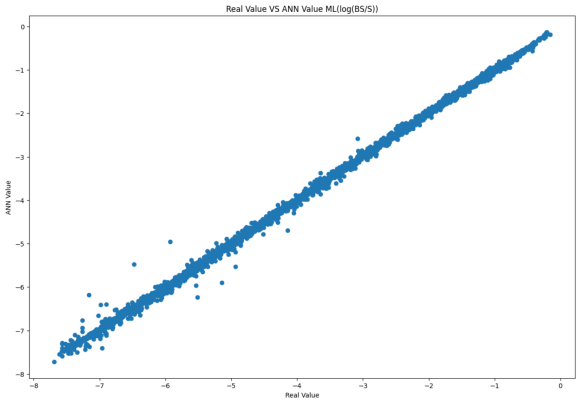
(b) The Loss function of LSTM(log(C/S))



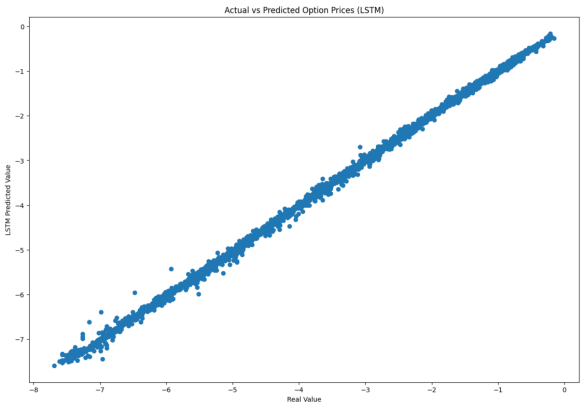
(c) The Loss function of GRU(log(C/S))

Figure 23: The Loss function of different in predicting the logarithm of call option price over the stock price, excluding GARCH in features.

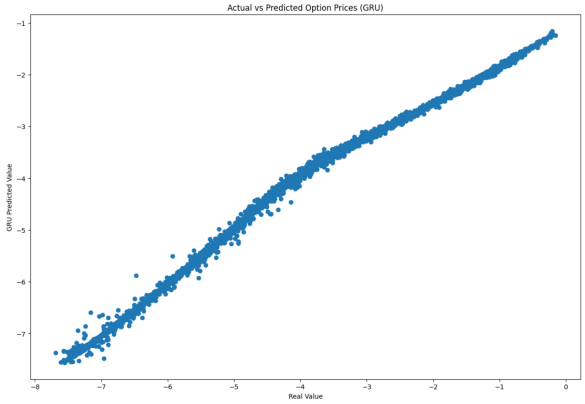
Figure 23 shows the loss functions for different models predicting log(C/S), where C is the option price and S is the underlying stock price excluding, GARCH in features. These loss functions, representing mean squared error (MSE), provide insights into how well each model learns during training. Subplots 23a, 23b, and 23c in order show MLP, LSTM, and GRU, all show a rapid decrease in the loss within the first few epochs confirming their ability to minimize prediction errors over epochs.



(a) Actual vs Predicted MLP(log(C/S))



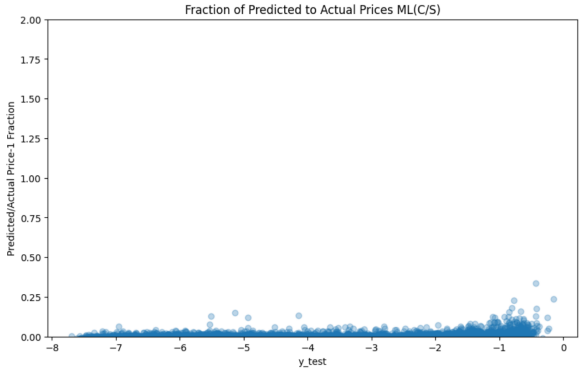
(b) Actual vs Predicted LSTM(log(C/S))



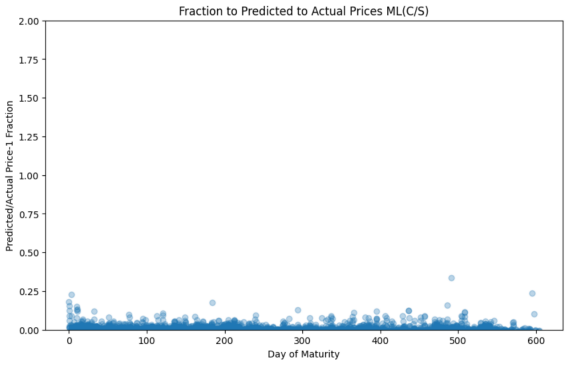
(c) Actual vs Predicted GRU(log(C/S))

Figure 24: Actual vs Predicted in predicting the logarithm of call option price over the stock price for different models, excluding GARCH in features.

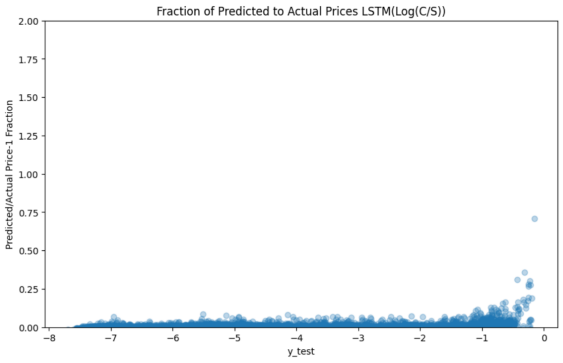
Figure 24 illustrates the actual versus predicted values of log(C/S) for the three models, excluding GARCH in features. Subplots 24a, 24b, 24c in order show MLP, LSTM, and GRU models, all presents points closely align along the diagonal, indicating high accuracy in predictions.



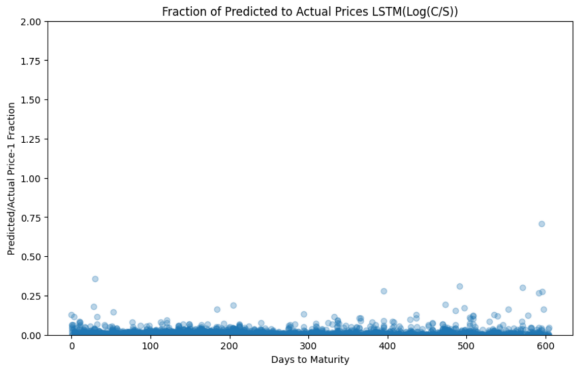
(a) Fraction error of MLP(log(C/S)) over y_test



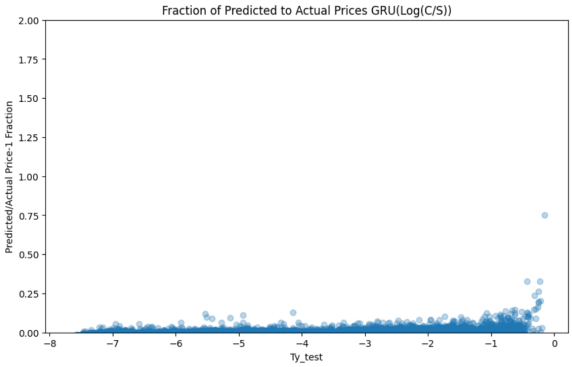
(b) Fraction error of MLP(log(C/S)) over days to maturity



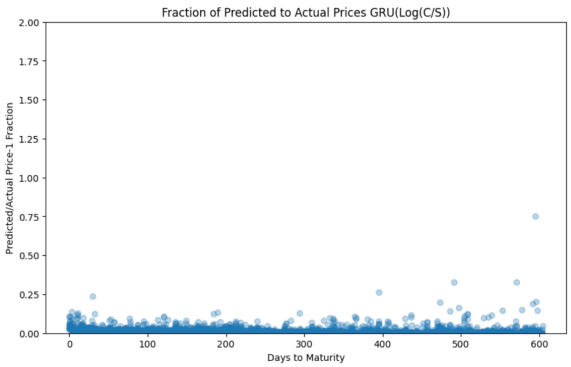
(c) Fraction error of LSTM(log(C/S)) over y_test



(d) Fraction error of LSTM(log(C/S)) over days to maturity



(e) Fraction error of GRU(C/S) over y_test



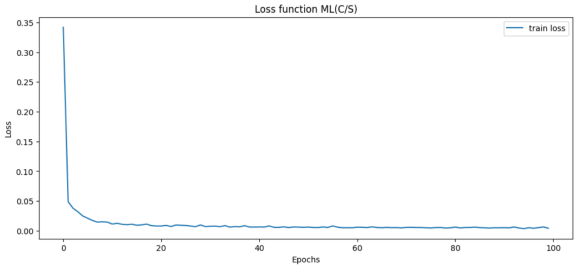
(f) Fraction error of GRU(C/S) over days to maturity

Figure 25: Fraction error different Models in predicting the logarithm of call option price over the stock price, excluding GARCH in features.

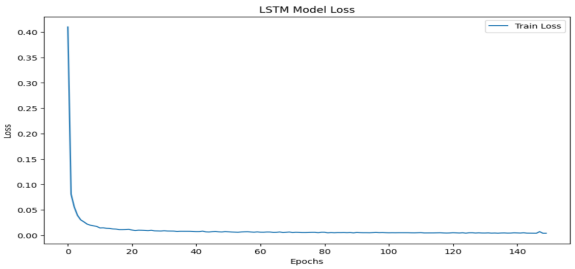
Figure 25 shows the fraction of Predicted to Actual Prices minus one Using in order MLP, LSTM, and GRU Model in predicting $\log\left(\frac{C}{S}\right)$ and excluding GARCH in features.

Subplots 25a, 25c, and 25e illustrate the fraction error over the y_test values for in order MLP, LSTM, and GRU model, with errors remaining low and distributed uniformly, the error is higher when the $\log\left(\frac{C}{S}\right)$ is near negative one, suggesting consistent model performance. Subplot 25b, 25d, and 25f depicts the fraction error over days to maturity for in order MLP, LSTM, and GRU model, showing minimal error variations, confirming that the model maintains accuracy regardless of time to maturity. Overall, all plots show less reaction error and high accuracy for the model.

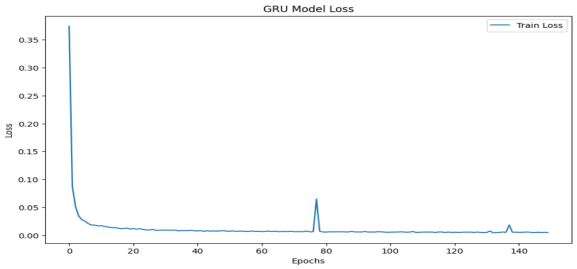
- Figures show the performance of various models in predicting the logarithm of call option price over the stock price. Notably, this analysis is conducted by incorporating GARCH as one of the features.



(a) MLP(log(C/S)) Loss function



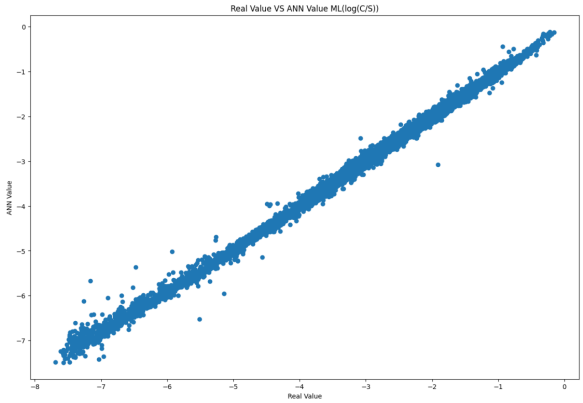
(b) LSTM(log(C/S)) Loss function



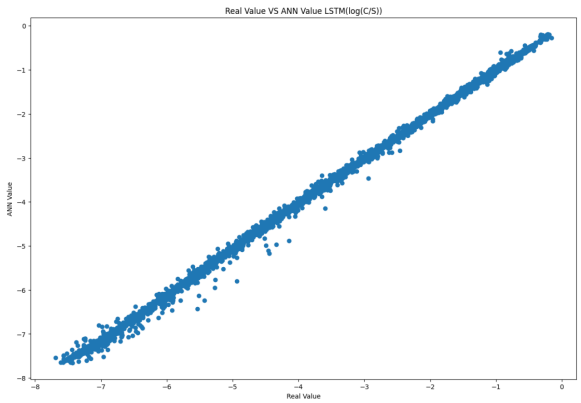
(c) GRU(log(C/S)) Loss function

Figure 26: The Loss function of different models in predicting the logarithm of call option price over the stock price, incorporating GARCH as features.

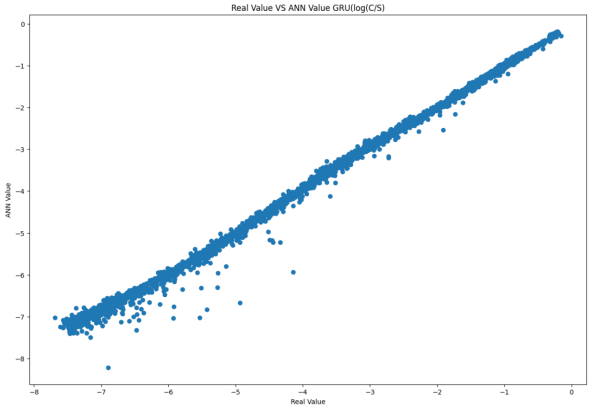
Figure 26 shows the loss functions over epochs for different models predicting log(C/S), where C is the option price and S is the underlying stock price including GARCH in features. In each subplot (26a, 26b, and 26c), in order shows MLP, LSTM, and GRU models, all show training loss decreases rapidly initially and then plateaus, indicating that the models converge and stabilize after a certain number of epochs. Notably, the GRU model's loss function exhibits a spike, suggesting potential instability or overfitting at certain epochs since our GRU model dropout is more than LSTM.



(a) Actual vs Predicted MLP(log(C/S))



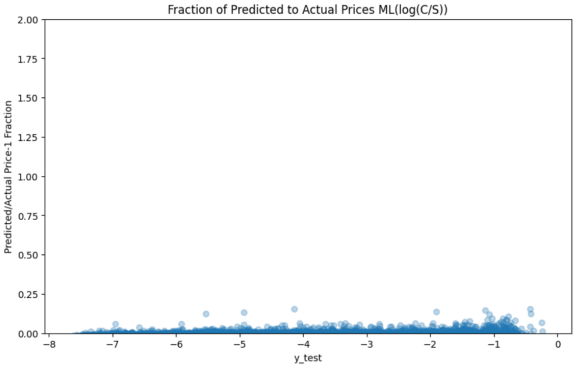
(b) Actual vs Predicted LSTM(log(C/S))



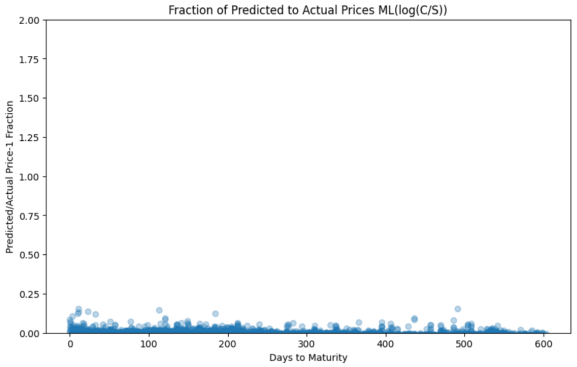
(c) Actual vs Predicted GRU(log(C/S))

Figure 27: Actual vs Predicted in predicting the logarithm of call option price over the stock price for different models, incorporating GARCH in features.

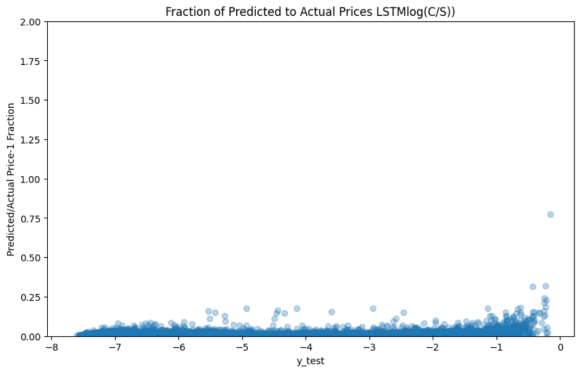
Figure 27 compares the actual versus predicted values of log(C/S) for the three models. Each scatter plots (27a, 27b, and 27b) in order show MLP, LSTM, and GRU model, all have a strong linear relationship, indicating that all models perform well in predicting the target variable. The points are closely aligned along the diagonal line, demonstrating that the predictions are generally accurate with few outliers.



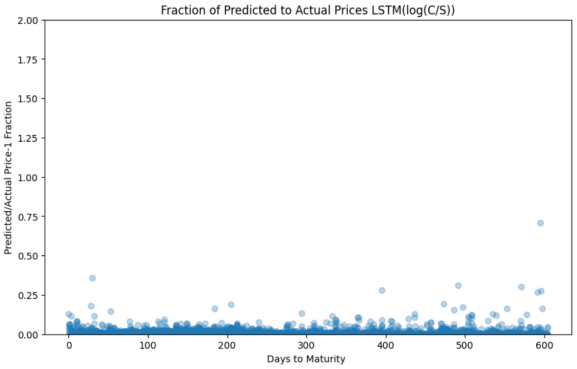
(a) Fraction error of MLP(log(C/S)) over y_test



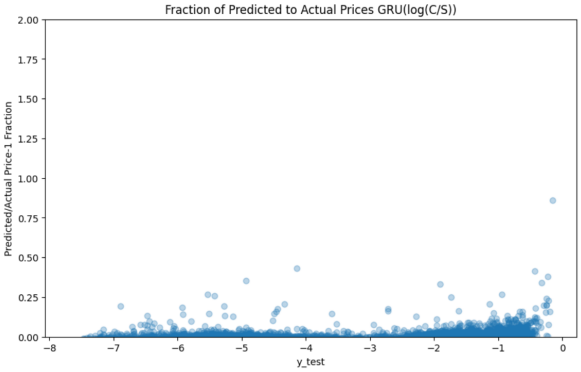
(b) Fraction error of MLP(log(C/S)) over days to maturity



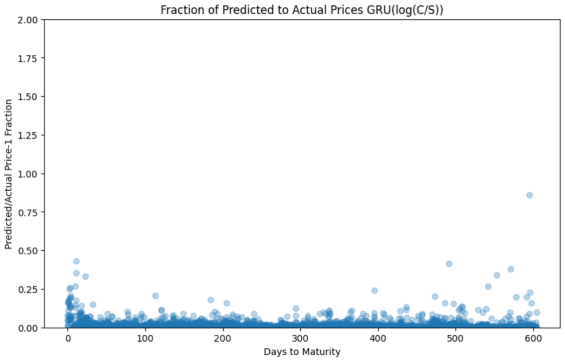
(c) Fraction error of LSTM(log(C/S)) over y_test



(d) Fraction error of LSTM(log(C/S)) over days to maturity



(e) Fraction error of GRU(C/S) over y_test



(f) Fraction error of GRU(C/S) over days to maturity

Figure 28: Fraction error Using different models in predicting the logarithm of call option price over the stock price, incorporating GARCH in features.

Figure 28 illustrates the fraction of Predicted to Actual Prices minus one for MLP, LSTM, and GRU models in predicting the logarithm of call option price over the stock price, incorporating GARCH in features, providing a comprehensive view of its predictive performance. Subplots 28a, 28c, and 28e in order show MLP, LSTM, and GRU models fraction error over the y_test values, maintaining a pattern of mostly low errors but with noticeable higher error outliers, the higher error can be seen near the negative on. Lastly, subplots 28b, 28d, and 28f in order show MLP, LSTM, and GRU models fraction error over days to maturity, revealing a stable error distribution with minimal higher error instances, the higher error can be seen near the zero. Overall, all models occasional higher errors, but they generally performs very well, maintaining low error rates across different dimensions.

6.3 Introducing New Data for Model Validation

In this section, our aim is to assess how well the model handles expiration dates that are earlier than the input expiration date provided. This focused evaluation will help us determine the model's effectiveness and reliability in predicting outcomes for earlier expiration dates. Therefore, in this section, we introduce

a new dataset, the new dataset mirrors the original data structure but includes entries with a distinct expiration date. This variation ensures that the model is exposed to a different temporal context. As test data, we analyze an American-style dataset of historical call option data for Tesla, Inc. (TSLA). The specific focus is on options that expired on March 15, 2024. This data, which the machine learning model has never seen before, consists of 378 entries with a strike price of 250. The trades considered in this analysis have expiry dates ranging from March 2023 to March 2024, as illustrated in Figure 29.

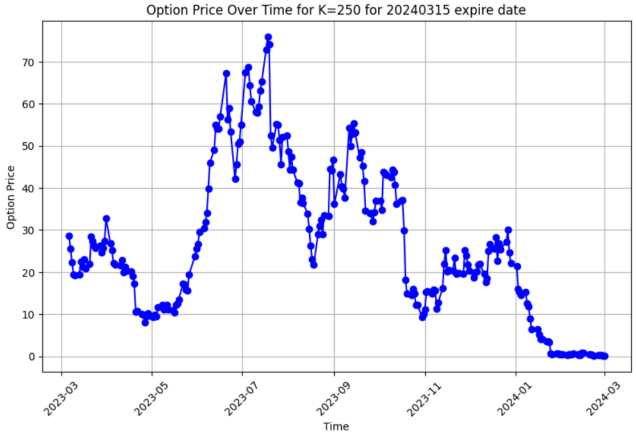


Figure 29: TSLA Option Price Over Time for K=250 for 20240315 expire date.

The analysis provides valuable insights into the behavior and characteristics of TSLA call options over this period. The period under consideration captures a full year of market activity, offering a comprehensive view of how these options performed as they approached their expiration date.

In our data preparation, as detailed in the data preparation section, we meticulously prepared the dataset to ensure its suitability for machine learning analysis. We selected MLP machine learning models that demonstrated optimal performance for our chosen features to predict $\log\left(\frac{C}{S}\right)$ as target. These features include Implied Volatility, the ratio of the strike price to the current stock price, the time remaining until the option's expiration date, the risk-free interest rate, and the conditional volatility that is derived using the GARCH (1,1) model.

By incorporating these five features, the machine learning model can better understand the complex relationships and dynamics within the options market, leading to more accurate predictions and valuable trading insights.

Time terms	**Min**	**Max**	**Mean**	**Std**
Total terms	-0.01	0.88	0.35	0.21
Short term	0.045	0.59	0.32	0.14
Medium term	-0.01	0.88	0.42	0.25
Long term	0.002	0.50	0.25	0.15

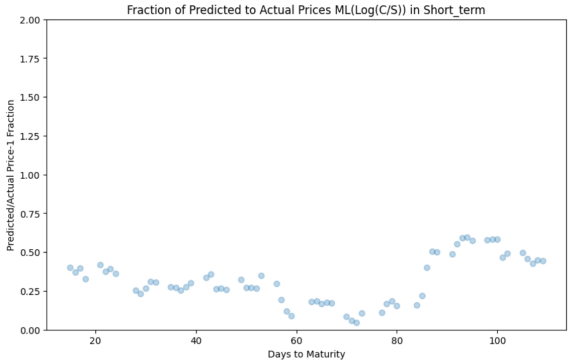
Table 16: Comparison fraction error.

The table 16 compares the fraction error of the MLP model when the log ratio of $\frac{C}{S}$ is our target, across different time terms over days to maturity, with a strike price $K = 250$ for options expiring on 20240315. The columns in the table represent the minimum (Min), maximum (Max), mean (Mean), and standard deviation (Std) of the fraction error. The analysis categorizes predictions into three distinct groups based on their temporal distance from the expiry date: short-term, medium-term, and long-term.

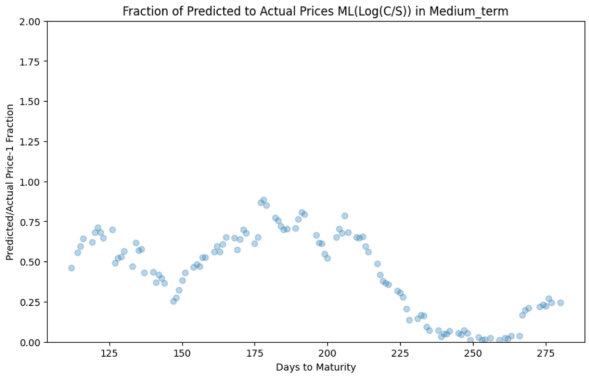
In the analysis, it is evident that the variation in prediction errors differs across these time terms. The MLP model shows variations in the range of prediction errors and their distribution across the different terms. Generally, the predictions for medium and long-term horizons tend to show less extreme errors and a more centered mean, indicating a consistency in predictive accuracy that is preferable in financial forecasting.

Short-term predictions, on the other hand, exhibit a narrower range of error but with a slightly higher average error. This might reflect the challenges associated with capturing immediate market volatilities or the specific conditions at shorter forecasting horizons, which can be more susceptible to

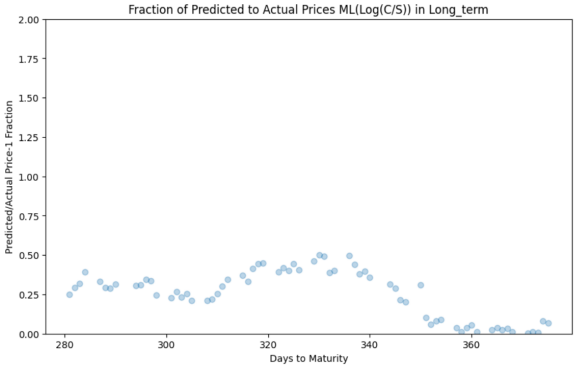
sudden market shifts or less predictable events. The standard deviation across the terms illustrates the variability of the prediction errors, with medium-term predictions showing higher variability. This could indicate fluctuations in model performance depending on the specific characteristics of the data set or market conditions at these different periods.



(a) Fraction error in Short term over days to maturity.



(b) Fraction error in Medium term over days to maturity.



(c) Fraction error in Long term over days to maturity.

Figure 30: Fraction error for different time terms over days to maturity.

Figure 30 illustrates the fraction of Predicted to Actual Prices minus one of the logarithms of call option price over the stock price ($\log\left(\frac{C}{S}\right)$) using MLP over days to maturity for strike price of $K = 250$ expiring on 20240315, categorized into short, medium, and long-term maturities. The x-axis in each subplot represents the days to maturity, while the y-axis indicates the fraction error, which is the ratio of predicted to actual prices minus one.

Subfigure 30aa focuses on the short term, showing that the fraction error remains relatively low and stable, particularly within the first 50 days to maturity. Beyond this period, there is a slight increase in variability, but the errors remain relatively contained. Subfigure 30bb depicts the medium term, where the fraction error exhibits more pronounced variability. The fraction error increases significantly between 100 to 200 days to maturity, indicating larger deviations of predicted prices from actual prices during this period. This suggests that the prediction model may be less accurate for medium-term maturities. Subfigure 30cc illustrates the long term, with the fraction error showing a lower and more stable pattern, particularly towards the end of the maturity period. The error remains relatively low after 300 days to maturity, suggesting better prediction accuracy for long-term maturities compared to the medium-term.

Breaking down the performance by time terms, the short-term predictions show a mean error of 0.32, the medium-term a mean error of 0.42, and the long-term a mean error of 0.25. In summary, the model performance highlights the model's varying effectiveness across different time horizons, with a tendency for better performance in shorter-term predictions.

7 Conclusions

This dissertation explored the performance of various models for option pricing, specifically comparing different advanced machine learning techniques such as Multi-Layer Perceptron (MLP), Long Short Term Memory networks (LSTM), and Gated Recurrent Unit (GRU) neural networks. The analysis was conducted on two main targets: C/S and $\log(C/S)$, both with and without incorporating the GARCH feature.

In analyzing the performance of the models in predicting the targets using both log(c/s) and c/s, the results highlight significant differences in error metrics, specifically the error fraction and mean squared error (MSE). The mean squared error (MSE) is lower when the target is c/s, indicating that the absolute differences between the predicted and actual values are smaller when predicting c/s directly. However, MSE is more sensitive to outliers due to the squaring of errors, which can disproportionately affect the metric if there are extreme values in the dataset. Conversely, when the target variable is log(c/s), the error fraction is notably lower compared to when the target is c/s. The error fraction is a critical measure in this context because it provides a normalized view of the error relative to the target values, offering insight into the proportion of the prediction error in relation to the actual values. The lower error fraction in the case of log(c/s) suggests that the model's predictions are closer to the true values in a proportional sense. This normalization helps in mitigating the impact of larger values, making it a more reliable metric for evaluating model performance in such scenarios.

The importance of the error fraction over MSE in this context stems from the need to ensure consistent predictive performance across the range of target values. Since the error fraction adjusts for the magnitude of the target values, it provides a more balanced and comparable measure of error. This is because the error fraction accounts for the relative scale of predictions, offering a more meaningful evaluation metric in contexts where target values vary widely. Thus, prioritizing the error fraction ensures a more reliable and generalized model performance evaluation. Therefore, despite the better MSE for c/s, the lower error fraction for log(c/s) is more significant, indicating a model that generalizes better across varying scales of data.

Based on the analysis of the performance of models with GARCH as one of the features in predicting the call option price over the stock price, there is an improvement in predictive accuracy. However, this enhancement does not extend in predicting the logarithm of call option price over the stock price, as indicated by the lack of significant improvement in MSE and fractional error metrics. This discrepancy may stem from the use of implied volatility as a feature, which is often derived from the market price of options. This creates a circular dependency, as the model uses implied volatility to estimate option prices, while implied volatility itself is contingent on those prices. To address this issue, future research should focus on developing models that do not incorporate implied volatility, thereby eliminating this dependency.

Based on the comprehensive analysis, while MLP and GRU models with a mean fraction of -0.003 and 0.002 as previously shown in Table 15 perform well, the LSTM model with a mean fraction of -0.0002 emerges as the nominated model for options pricing, LSTM is an ideal choice for dynamic financial modeling because of Its ability to effectively handle time series data. The fraction errors of all models are mostly low but with higher error outliers when the logarithm of call option over stock price is near negative one and when we are really near to the expiration date. These enhanced models were particularly effective in capturing the nonlinear relationships in financial markets, leading to a more reliable estimation of call option prices. This research contributes to the development of more dynamic financial models, suggesting that integrating modern machine learning approaches can substantially improve financial decision-making and risk management strategies.

The evaluation of the MLP models on the test data provides a comprehensive insight into their predictive capabilities and robustness. The performance of the machine learning model in predicting ($\log(C/S)$) was evaluated across different time terms to maturity. Table 16 reveals that the overall performance of the model, indicated by the metrics across different time terms, varies notably. For the total terms, the model shows a mean fraction error of 0.35 which is about 100 times bigger than fraction error the trained MLP model. This result indicates that for better performance, as an input to our model we have to choose an expiration date that is near the time that we want to predict since the market volatility could be different from time to time and can affect our prediction.

8 Limitations and Challenges

While the methodologies applied in this research provide valuable insights into option pricing and predictive modeling, some limitations and challenges will highlight the need for caution when interpreting the results. Here, I highlight the most important points, (with further details provided in Sections 5 and 3).

- Limited Scope of Data

The analysis was based on a dataset comprising historical call option data for Tesla (TSLA) over one year period from 2023 to 2024, I collected the data by purchasing a premium account with limited data permissions, as the option without limitations was too expensive for me. The choice to focus on a single stock introduces a limitation in terms of generalizability. The findings may not necessarily extend to other stocks or different market conditions. Stock-specific factors, such as corporate events or sector-specific trends, could have influenced the results, making it difficult to generalize the conclusions to a broader market context. Future research could benefit from including a diverse range of stocks and longer time horizons to assess the robustness of the model across different scenarios. Another significant challenge was sourcing adequate and high-quality data. The datasets I initially encountered were either insufficient in size or contained numerous missing values (NaNs), which posed considerable obstacles in ensuring robust model training and reliable outcomes. Many publicly available datasets lacked the depth and breadth required for a thorough analysis, making it difficult to achieve the desired level of accuracy and generalization in the model. Furthermore, some datasets were burdened with excessive noise and irrelevant information, necessitating extensive pre-processing to render them useful for model training. When data is limited, simulating synthetic data can augment the training set, improving the model robustness that I use in this study too.

- Implied Volatility Derivation

Implied volatility is a critical input for pricing options and reflects the market's expectation of future volatility. In the context of options pricing models, implied volatility is often derived from the option's market price using models such as Black-Scholes. However, this derivation introduces a circular dependency because the model relies on implied volatility to calculate the option price, while implied volatility itself is derived from the option price. This feedback loop can cause inconsistencies, especially in rapidly changing markets where implied volatility might react to price changes rather than inform them. To address this issue, it's essential to implement iterative techniques that converge on a stable implied volatility value. These techniques recalibrate the model dynamically, using market data to refine predictions without assuming static volatility levels. By acknowledging and addressing this circular dependency, the model's predictive accuracy can be significantly enhanced, reducing bias and improving reliability in different market conditions. This aspect raises questions about the objectivity of the input data used for predictive modeling, I add GARCH volatility to my features to have better understanding about this volatility,but to address this dependency, future research should focus on developing models that do not incorporate implied volatility, thereby eliminating this issue.

- Time Limits

In the pursuit of developing an effective model for my research, I encountered several challenges that significantly impacted both the model selection process and data acquisition. I experimented with various models, each with distinct architectural frameworks, in an effort to determine which would best suit the requirements of my research objectives. This iterative process was not only time-consuming but also computationally intensive, as each model demanded extensive resources for training and validation. For overcome this challenge I used optimizer, and activation functions, and learning rate methods in the architecture of my models.

- Neural Network Challenges

The application of neural networks, while offering improved prediction capabilities, introduces challenges related to model complexity and data requirements. I try to handle these challenges but still, Neural networks require substantial amounts of data for effective training, and the limited dataset used in this study could have constrained the model's ability to generalize. Furthermore, neural networks can suffer from issues like overfitting and Vanishing. For overcoming the vanishing problem, I use batch normalization and to combat overfitting, the dropout technique is employed. These strategies ensure that neural networks maintain their predictive power without succumbing to common pitfalls.

9 Future Research

While the research demonstrates the potential of advanced neural networks in options pricing, several limitations warrant attention.

One promising direction is to use alternative historical volatility, which is calculated based on the past price movements of the underlying asset. By analyzing the standard deviation of returns over a specified period, historical volatility offers an objective measure derived directly from historical price data. This approach could mitigate the circular dependency associated with IV, providing a more stable and straightforward measure for forecasting option prices. Historical volatility allows models to base predictions on actual market behavior rather than speculative expectations, which can be beneficial in capturing trends and patterns inherent in the asset's historical performance. Realized volatility, computed using high-frequency data, offers yet another measure by capturing actual market activity over short intervals that could enhance their responsiveness to sudden market changes, improving the timeliness and accuracy of option price predictions during periods of heightened market activity or stress. This approach leverages the granularity of high-frequency data to reflect short-term market dynamics, offering a more accurate representation of recent volatility. Future research would be to create a model with these volatilizes instead of implied volatility.

Additionally, incorporating additional financial features and market indicators can capture more nuanced aspects of market dynamics. This could also consider the integration of sentiment analysis from news and social media, as this could provide additional insights into market dynamics and investor behavior, further enhancing model performance.

Another area of interest is the improvement of the model architecture. Advancements in neural network architectures, such as Transformers and Convolutional Neural Networks (CNNs), offer promising avenues for future exploration. These architectures can capture complex patterns and dependencies within financial data, potentially outperforming traditional models in predicting option prices.

Finally, expanding the dataset to include more extensive historical data of different stocks and diverse market conditions can help in making the model more robust. The focus on a single stock, such as Tesla, introduces potential biases that may limit the generalizability of findings across different market conditions and asset classes. Future research should explore a broader range of stocks and incorporate diverse market data to validate and extend the applicability of the proposed models. Future research should also focus on identifying the optimal expiration date to provide input for our model, enabling accurate predictions of our target within the desired time frame.

By pursuing these future research directions, it is expected that the accuracy and reliability of machine learning models in financial forecasting will continue to improve, thereby contributing to more informed decision-making in the financial industry.

References

[1] Garrett Adams. *Black-Scholes and Neural Networks*. Tech. rep. 1486. All Graduate Plan B and other Reports, 2020. URL: https://doi.org/10.26076/133e-2777.

[2] U. Anders, O. Korn, and C. Schmitt. "Improving the pricing of options: a neural network approach". In: *Journal of forecasting* 17.5–6 (1998), pp. 369–388. DOI: 10.1002/(SICI)1099-131X(1998090)17:5/6<369::AID-FOR702>3.0.CO;2-S.

[3] Barchart. *Commodity, Stock, and Currency Quotes, Charts, News & Analysis*. Accessed: 2024-06-27. 2024. URL: https://www.barchart.com/.

[4] Fischer Black and Myron Scholes. "The Pricing of Options and Corporate Liabilities". In: *Journal of Political Economy* 81.3 (1973), pp. 637–654. URL: https://www.cs.princeton.edu/courses/archive/fall09/cos323/papers/black_scholes73.pdf.

[5] Emily Chang. *CNN-LSTM vs ANN: Option Pricing Theory*. Tech. rep. Western University, 2022. URL: https://ir.lib.uwo.ca/usri/usri2022/ReOS/321/#:~:text=an%20ANN%20model.-,The%20results%20from%20this%20paper%20show%20that%20the%20CNN%2DLSTM,accuracy%20when%20predicting%20option%20prices..

[6] Wikipedia contributors. *Activation function — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-July-2024]. 2024. URL: https://en.wikipedia.org/wiki/Activation_function.

[7] Wikipedia contributors. *Black–Scholes model*. Accessed: 2024-05-29. 2024. URL: https://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model.

[8] Wikipedia contributors. *Call option*. Accessed: 2024-05-29. 2024. URL: https://en.wikipedia.org/wiki/Call_option#:~:text=Price%20of%20options&text=the%20expected%20intrinsic%20value%20of,delay%20to%20the%20payout%20time.

[9] Robert Culkin and Sanjiv R Das. "Machine Learning in Finance: The Case of Deep Learning for Option Pricing". In: *Journal of Investment Management* 15 (2017), pp. 92–100. URL: https://srdas.github.io/Papers/BlackScholesNN.pdf.

[10] Prateek Samuel Daniels. "Machine Learning Techniques for Pricing, Hedging and Statistical Arbitrage in Finance". PhD Thesis. PhD thesis. Sydney, Australia: University of Technology Sydney, 2022. URL: https://www.proquest.com/dissertations-theses/machine-learning-techniques-pricing-hedging/docview/2901816039/se-2.

[11] A.-A. Encean and D. Zinca. "Cryptocurrency Price Prediction Using LSTM and GRU Networks". In: *2022 International Symposium on Electronics and Telecommunications (ISETC)*. IEEE, 2022, pp. 1–4. DOI: 10.1109/ISETC56213.2022.10010329.

[12] Robert F. Engle. "Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation". In: *Econometrica* 50.4 (1982), pp. 987–1007. DOI: 10.2307/1912773.

[13] W. Farida Agustini, I.R. Affianti, and E.R. Putri. *Stock price prediction using geometric Brownian motion*. 2018. DOI: 10.1088/1742-6596/974/1/012047.

[14] Stephen Figlewski. "Forecasting Volatility Using Historical Data". In: (May 1994). URL: https://www.researchgate.net/publication/228140936_Forecasting_Volatility_Using_Historical_Data.

[15] Diogo P. Flórido. "Estimate European vanilla option prices using artificial neural networks". MA thesis. Universidade de Lisboa, 2022. URL: https://repositorio.ul.pt/bitstream/10451/53641/1/TM_Diogo_Florido.pdf.

[16] Kunihiko Fukushima. "Neocognitron: A self organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (1980), pp. 193–202. DOI: 10.1007/bf00344251.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Available from: ProQuest Ebook Central. [4 June 2024]. Cambridge: MIT Press, 2016.

[18] Richard HR Hahnloser et al. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". In: *Nature* 405.6789 (2000), pp. 947–951. DOI: 10.1038/35016072.

[19] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852 [cs.CV].

[20] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". eng. In: *Neural computation* 9.8 (1997), pp. 1735–1780. ISSN: 0899-7667.

[21] J. Huang and Z. Cen. "Cubic Spline Method for a Generalized Black-Scholes Equation". In: *Mathematical Problems in Engineering* (2014). DOI: 10.1155/2014/484362. URL: https://doi.org/10.1155/2014/484362.

[22] J. M. Hutchinson, A. W. Lo, and T. Poggio. "A Nonparametric Approach to Pricing and Hedging Derivative Securities Via Learning Networks". In: *The Journal of Finance* 49.3 (1994), pp. 851–889. DOI: 10.1111/j.1540-6261.1994.tb00081.x.

[23] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[24] A Itkin. *Deep learning calibration of option pricing models: some pitfalls and solutions*. 2019. arXiv: 1906.03507 [q-fin.CP]. URL: https://arxiv.org/abs/1906.03507.

[25] Zuzana Janková. "Drawbacks and Limitations of Black-Scholes Model for Options Pricing". In: *Journal of Financial Studies and Research* (2018). URL: https://api.semanticscholar.org/CorpusID:159387768.

[26] Zuzana Janková. "Drawbacks and Limitations of Black-Scholes Model for Options Pricing". In: *Journal of Financial Studies and Research* (2018). URL: https://api.semanticscholar.org/CorpusID:159387768.

[27] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[28] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[29] Y. Li and K. Yan. "Prediction of Barrier Option Price Based on Antithetic Monte Carlo and Machine Learning Methods". In: *Cloud Computing and Data Science* (2023), pp. 77–86. DOI: 10.37256/ccds.4120232110. URL: https://doi.org/10.37256/ccds.4120232110.

[30] L. Liang and X. Cai. "Time-sequencing European options and pricing with deep learning – Analyzing based on interpretable ALE method". In: *Expert Systems with Applications* 187 (2022), p. 115951. DOI: 10.1016/j.eswa.2021.115951. URL: https://doi.org/10.1016/j.eswa.2021.115951.

[31] Chang Liu et al. "Forecasting copper prices by decision tree learning". In: *Resources Policy* 52 (2017), pp. 427–434. ISSN: 0301-4207. DOI: https://doi.org/10.1016/j.resourpol.2017.05.007. URL: https://www.sciencedirect.com/science/article/pii/S0301420716302501.

[32] Y. Liu and X. Zhang. "Option Pricing Using LSTM: A Perspective of Realized Skewness". In: *Mathematics* 11 (2023), p. 314. DOI: 10.3390/math11020314. URL: https://doi.org/10.3390/math11020314.

[33] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[34] Laurens Van Mieghem, Antonis Papapantoleon, and Jonas Papazoglou-Hennig. *Machine learning for option pricing: an empirical investigation of network architectures*. 2023. arXiv: 2307.07657 [q-fin.CP].

[35] Christopher Olah. *Understanding LSTMs*. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2024-07-16. 2015.

[36] Frank Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: 10.1037/h0042519.

[37] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". eng. In: *Nature (London)* 323.6088 (1986), pp. 533–536. ISSN: 0028-0836.

[38] Dev Shah, Haruna Isah, and Farhana Zulkernine. "Predicting the Effects of News Sentiments on the Stock Market". In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 4705–4708. DOI: 10.1109/BigData.2018.8621884.

[39] Leslie N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. 2018. arXiv: 1803.09820 [cs.LG].

[40] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[41] Wikipedia contributors. *Geometric Brownian Motion — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Geometric_Brownian_motion. Accessed: 2024-07-16. 2024.

[42] Wikipedia contributors. *Neuron — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Neuron. Accessed: 2024-07-16. 2024.

[43] Bing Xu et al. *Empirical Evaluation of Rectified Activations in Convolutional Network*. 2015. arXiv: 1505.00853 [cs.LG].

[44] Habib Zouaoui and Meryem-Nadjat Naas. "Option pricing using deep learning approach based on LSTM-GRU neural networks: Case of London stock exchange". In: *Data Science in Finance and Economics* 3.3 (2023), pp. 267–284. ISSN: 2769-2140. DOI: 10.3934/DSFE.2023016. URL: https://www.aimspress.com/article/doi/10.3934/DSFE.2023016.