UNIVERSITY OF LIVERPOOL

MATH391 Report

Report of Neural Network for Arbitrage-free Pricing

Jierui Li, Langming Liang, Menglei Zhang, Qi Huang, Sizhe Jiang and Shiyao Gu

Keywords: Neural Network, Call Option, Arbitrage-free

Project Supervisor: Boutaib Youness, Amiriyan Samira

Module Supervisor: Corina Constantinescu

Abstract

This report aims to incorporate the knowledge of machine learning to exploring the potential computational power of neural network in the pricing European call option under arbitrage-free condition.

We study the project to find the alternative approach to traditional method such as Black-Scholes which rely on certain unrealistic assumptions in option pricing, neural network model is designed around assumptions that reflect actual market behaviors. By integrating arbitrage-free pricing principles within the architecture of the learning algorithm, the neural network model produces option prices that naturally align with realistic market dynamics.

Contents

1. Introduction and Background

1.1. Related to finance

1.1.1. Explain the concept of European options and the objectives of the research

First, let's quickly understand the meaning of a call option through its origin story[1]. The ancient Greek philosopher Thales predicted that there would be a bountiful olive harvest in the summer. He believed that the demand for olive oil presses would far exceed supply during the harvest, but he didn't have enough money to purchase olive oil presses at the time. So, he approached the owners of the presses and offered to pay a small amount upfront to secure the right to rent the presses at the current price in the summer. When the harvest came, and the demand for the presses soared, Thales only had to pay the previously agreed-upon rent, which was significantly lower than the market rate. He then rented out the presses at a much higher price, making a substantial profit on the difference. This is one of the earliest examples of a call option.

In simple terms, a call option allows individuals to use their predictions and judgments about the future to set up a scenario where they can profit. More scientifically and rigorously, a call option can be defined as: The holder has the right to buy the underlying asset at the strike price on the expiration date.

The goal of this study is to determine how we can reasonably price an option. In other words, if we were the ancient Greek philosopher, how much of a premium should we pay upfront to ensure that the olive oil press owner agrees to rent us the presses in the summer at a predetermined price. This research focuses on identifying the right amount of premium that would make both the buyer and seller agree to the contract, similar to how we price an option in financial markets.

1.1.2. Importance of Arbitrage-Free Pricing for Financial Markets and the Necessity for New Model Inventions

Returning to the research objective of pricing options, it is essential to understand that option pricing is a challenging problem. The ancient Greek philosopher paid a premium based on intuition, but in modern exchanges, we cannot negotiate with traders. What we are trading is the right to make a decision in the future, not a tangible object like an apple. The rise and fall of stock prices are fundamentally driven by the number of people willing to buy the stock. If we knew a particular stock would rise tomorrow, we would buy it today, which would cause the stock to rise today. This feedback loop means that the act of prediction itself can influence the predicted outcome. In a complex and efficient market, it becomes nearly impossible to predict whether a stock will be popular tomorrow because we don't know how many people will be willing to buy it. This makes option pricing a difficult problem.

Moreover, establishing a fair price for options has significant implications for the market. Before explaining these implications, we must first define what constitutes a fair price. A fair price should not create risk-free arbitrage opportunities, where someone can profit without taking any risks.

However, many scholars throughout history have developed models to help price options. For example, Fischer Black and Myron Scholes introduced the famous Black-Scholes model in 1973, which provides a theoretical foundation for pricing European options and laid the groundwork for the development of financial engineering [2]. The model assumes that stock prices follow a geometric Brownian motion and

are traded in a frictionless market (i.e., no transaction costs or taxes are involved). These assumptions form the core foundation of the pricing formula. In the **Black-Scholes model**, the volatility of stock prices, the risk-free rate, time to maturity, strike price, and current stock price are the key factors affecting the price of options. Using these variables, the model provides a closed-form solution for pricing European call options and put options. The formulas are as follows:

$$C = S_0 N(d_1) - Ke^{-rT} N(d_2)$$

$$P = Ke^{-rT} N(-d_2) - S_0 N(-d_1)$$

Where:

- C is the price of the **call option**,
- P is the price of the **put option**,
- S_0 is the **current stock price**,
- K is the **strike price**,
- r is the **risk-free rate**,
- T is the **time to maturity**,
- $N(\cdot)$ is the **cumulative distribution function** of the standard normal distribution, and
- d_1 and d_2 are intermediate calculations that depend on the above variables.

The contribution of the Black-Scholes model is profound. First, it introduced the concept of no-arbitrage pricing, ensuring that option prices remain consistent and fair in an idealized market environment. This innovative pricing method prevents inconsistencies and anomalies in the option market, which was previously underdeveloped and lacked liquidity. The introduction of the Black-Scholes model also promoted a more transparent and efficient market environment, encouraging greater participation from both institutional investors and retail traders, thereby enhancing market depth and liquidity. The growth of the options market has significantly increased the trading volume and participants in the financial derivatives market, greatly enhancing the stability of the financial markets [3].

Performing no-arbitrage pricing for options is beneficial for the development of financial markets. In this research, the model we established is for pricing European options. In the future, it can be extended to no-arbitrage pricing for American options or even for options in other regions, which would benefit the financial markets of those regions. This could be particularly advantageous for emerging markets. For financial markets, introducing an effective pricing and forecasting model may potentially change certain market structures. In financial markets, the publication of pricing models can indeed influence market operations and alter investor behavior. The article "Sixty Years of Capital Asset Pricing Model: Research Agenda" (2023) explores the significant impact of models like the CAPM on the market. It emphasizes that as market dynamics evolve, pricing models must be regularly updated to remain effective. This highlights the necessity for financial models to stay current, ensuring the continued efficiency and stability of the market [4].It may also mean that researchers need to continuously develop new models or update existing ones. Therefore, the need for research into new pricing methods and models is essential in the financial markets. The contributions made with stronger computational power and more comprehensive datasets in the future will likely exceed current expectations. This will be further explained in next Section.

In summary, addressing how to price options while ensuring prices meet the no-arbitrage condition has profound implications for the market. The emergence of new approaches also holds significant research value. Building on prior theoretical studies, we propose a new method: utilizing Artificial Neural Networks (ANN) to learn the call option price function $C = C(T, K)$ based on empirical data, independent of traditional models, while ensuring the no-arbitrage condition. This method, along with the principles of neural networks, will be introduced in detail in the next section.

1.1.3. Tutorial of arbitrage

In this section, we will simulate the several stock market performance under arbitrage conditions, with expectation, this would demonstrate the importance of no-arbitrage pricing.

Now, let us assume that we are in a financial market. We will now demonstrate how to use mathematics to perform arbitrage in each of these four cases.(The examples we provide below all violate the rules specified in the situation)

Situation1

$$f \geq 0(\mathrm{NA}_0),$$

Assumptions:

A very foolish investor has introduced such an options product:

Option name	K(Dollars)	T(Days)	C(Dollars)
Foolish option 1	80	100	–1

Figure 1. *Situation1.*

Arbitrage Opportunity:

- Buying any amount of foolish option 1.

If the option price were negative, this implies that the buyer not only gains the right to purchase the underlying asset, but also receives income, which would constitute a form of risk-free profit. Therefore, the price of an option can only be non-negative.

Situation2

$$\frac{\partial f}{\partial K} \leq 0(\mathrm{NA}_1),$$

Assumptions:

There are currently two popular types of option combinations in the market:

Option name	K(Dollars)	T(Days)	C(Dollars)
Option 2	100	100	7
Option 3	110	100	10

Figure 2. *Situation 2.*

Arbitrage Opportunity

- Buy n Option 2.
- Sell n Option 3.

So that:

At Expiration:(Now we assume the price of oil is S Dollars/Tons)

- **If** $S \geq 110$
  - Exercise Option 2, buy the share of oil for 100 USD. Meanwhile,since the actual price of one share of oil has already been over 110 USD, the buyer of Option 3 would choose to exercise the option to buy the share for 110 USD, thus we will make a profit of 10 USD in this process.
  - So the profit we will make is: $(S-100)n-(S-110)n+3n=13n$
- **If** $100 \leq S < 110$
  - Exercise Option 2, buy one share of oil for 100 USD. Since the oil worth more than 100 USD, we can sell it at market price, and make some profit.
  - On the other hand, since the market price of the oil is lower than the agreed price, the buyer of Option 3 would not choose to exercise the option.
  - Thus, the profit we will make is: $(S-100)n+3n=nS-97n$
- **If** $S < 100$
  - Both the buyer of Option 3 and we would not choose to exercise the option.
  - Thus, the profit we will make is: $3n$

The specific portfolio formulas and images are as follows:

$$
\begin{aligned}
&\Pi \\
&= C_2 - C_3 \\
&= (S(T)-K_1,0)^+ - (S(T)-K_2,0)^+ \\
&= \begin{cases} 0 & S(T) \leq 100 \\ S(T)-100 & 100 < S(T) < 110 \\ 10 & 110 \leq S(T) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&P_r(T) \\
&= \Pi(T) + 10 - 7 \\
&= \begin{cases} 3 & S(T) \leq 100 \\ S(T)-97 & 100 < S(T) < 110 \\ 13 & 110 \leq S(T) \end{cases}
\end{aligned}
$$

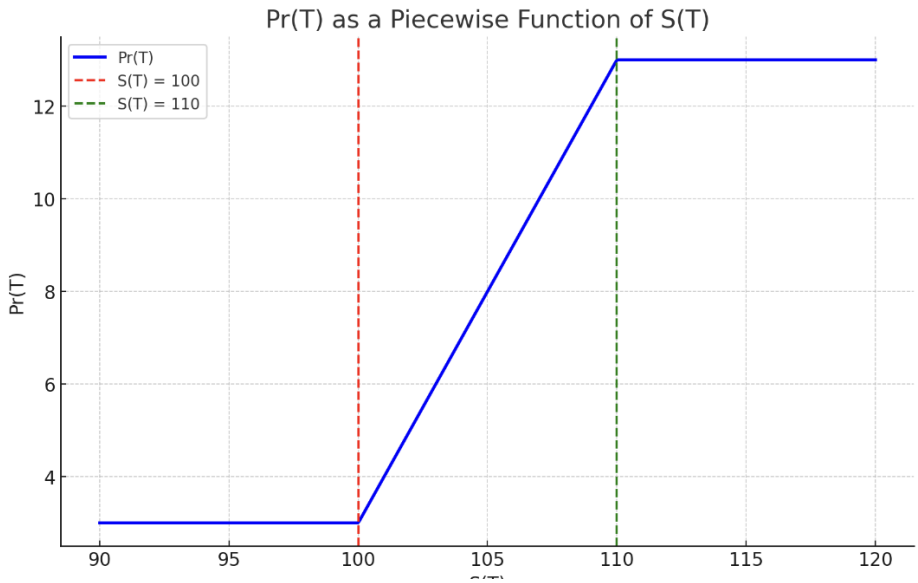


Figure 3. *Profit For Situation 2.*

Note that in the third case($S < 100$), you are supposed to lose money if you are in a reasonable investment market. However, in the first and second cases($S > 100$), due to your **more aggressive investment strategy** compared to the investors who buy Option 3, and since the market ultimately

reaches the situation you wanted to see, there is still a high probability that you can make money in these two cases, even in a reasonable investment market.

Situation3

$$\frac{\partial f}{\partial T} \geq 0(\mathrm{NA}_2),$$

Option name	K(Dollars)	T(Days)	C(Dollars)
Option 4	100	100	20
Option 5	100	120	10

Figure 4. *Situation 3.*

Further Assumption: There is a way to borrow oil for 20 days at a cost of less than 10 dollars per share[5].

Arbitrage Opportunity:

- Buy n Option 5.
- Sell n Option 4.

Now let us consider the worst-case scenario for us: 100 days later, Option 4 expires, and by then the oil market price is above 100 dollars. In this case, the buyer would exercise the option, and we would incur a loss from the price difference. However, 120 days later, our purchased Option 5 expires, and the oil price has dropped below 100 dollars, leaving us unable to profit from Option 5.

Even in this scenario, we still have a means to perform arbitrage:

- Borrow the corresponding quantity of oil, provide it to the buyer of Option 4, and pay the rental fee, assumed to be sn .
- Regardless of the oil market price at 120 days, exercise the option to buy oil and repay the borrowed oil.
- The profit we will make is: $(10 - s)n > 0$

Situation4

$$\frac{\partial^2 f}{\partial K^2} \geq 0(\mathrm{NA}_3),$$

Option name	K(Dollars)	T(Days)	C(Dollars)
Option 6	100	100	7
Option 7	110	100	6
Option 8	120	100	3

Figure 5. *Situation 4.*

Arbitrage Opportunity

We can create a **butterfly spread**:

- Buy n Option 6.
- Buy n Option 8.
- Sell $2n$ Option 7.

So that:

At Expiration:(Now we assume the price of oil is S Dollars/Tons)

The principle here is similar to the above. To avoid repetition, we will directly provide the portfolio formulas and images:

$$\begin{aligned}&\Pi\\&= C_6 + C_8 - 2C_7\\&= (S(T) - K_6, 0)^+ + (S(T) - K_8, 0)^+ - 2(S(T) - K_7, 0)^+\\&= \begin{cases}0 & S(T) \le 100\\ S(T) - 100 & 100 < S(T) < 110\\ 120 - S(T) & 110 \le S(T) < 120\\ 0 & 120 \le S(T)\end{cases}\end{aligned}$$

$$\begin{aligned}&P_r(T)\\&= \Pi(T) - 7 - 3 + 2 \cdot 6\\&= \Pi(T) + 2\\&= \begin{cases}2 & S(T) \le 100\\ S(T) - 98 & 100 < S(T) < 110\\ 122 - S(T) & 110 \le S(T) < 120\\ 2 & 120 \le S(T)\end{cases}\end{aligned}$$

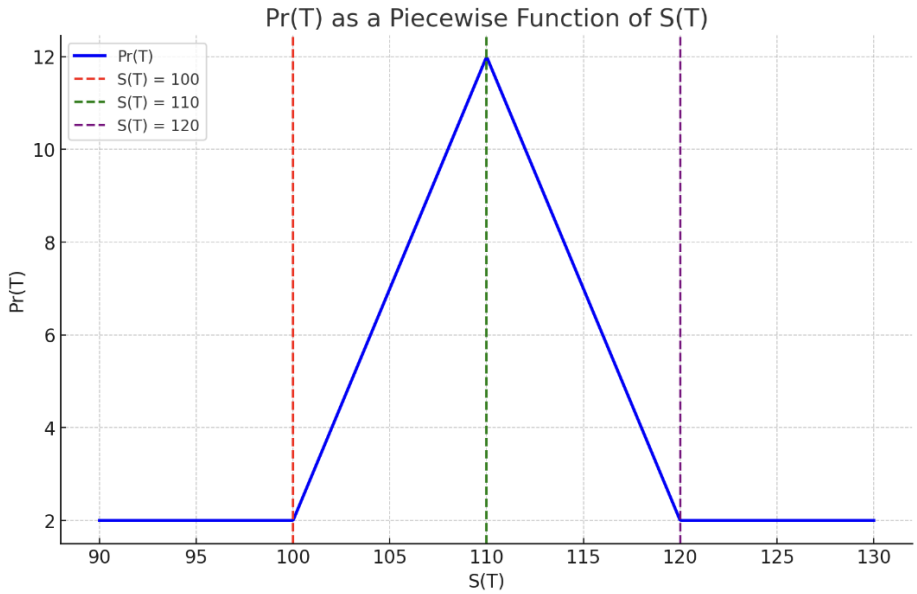


Figure 6. *Profit For Situation 4.*

From the above cases, we can see that if the pricing model does not satisfy the above conditions, many arbitrage opportunities will arise, leading to significant economic losses. Next, we will introduce the use of neural networks to address the rationality of no-arbitrage pricing.

1.2. Theoretical background of using neural networks for arbitrage-free pricing

1.2.1. What is neural network?

One week before we began writing this report, the 2024 Nobel Prize in Physics was announced, awarded to John J. Hopfield and Geoffrey E. Hinton in recognition of their outstanding contributions to the field

of deep learning and neural networks. This is a quote from the Nobel Prize website, which we believe is very fitting to include here.[6]

Machine learning differs from traditional software, which works like a type of recipe. The software receives data, which is processed according to a clear description and produces the results, much like when someone collects ingredients and processes them by following a recipe, producing a cake. Instead of this, in machine learning the computer learns by example, enabling it to tackle problems that are too vague and complicated to be managed by step by step instructions.

Neural networks are a crucial method in deep learning. As mentioned above, the most significant difference and advantage—compared to traditional programming is that we don't need to care about the exact 'recipe' for solving a problem. In other words, we're not actually 'teaching' the computer how to make a 'cake'. What we need to do is provide the computer with enough data and powerful computing capabilities. The neural network will then enable the computer to create 'recipes' that may be beyond human understanding.

From a mathematical perspective, a neural network can be understood as a function approximator, meaning that for any function from R^m to R^n (any function, with arbitrary m and n , not even limited to R), a neural network can approximate this function with sufficient precision (we will explain just how sufficient this precision is in the later Universal Approximation Theorem). The details of how this is achieved will be explained in **Section 2**.

1.2.2. Why neural network?

During the week 6 presentation, Dr. Azmoodeh asked us a tricky question:

"Which do you think is better, your model or the Black-Scholes formula?"

This interesting question sparked a lot of reflection among us afterward. Since the Black-Scholes formula was proposed in the early 1970s, and it has been widely used in the following years, serving as a milestone in financial mathematics and financial engineering. To reward such contribution it has won the Nobel Prize in Economics in 1997. Although the research paper we studied with was published almost 40 years later than the Black-Scholes formula, the contributions of the method proposed in this paper cannot be compared to the Black-Scholes formula.

We started to wondering: why did the authors choose to develop a neural network model for arbitrage-free pricing when such a efficient model like the Black-Scholes formula already exists?

Now, we may have a better answer to this question and the thought-provoking question that Dr. Azmoodeh posed to us. Before introduce this, let's discuss some relevant background. We will use our current answer to this question as the conclusion for this section.

Universal Approximation Theorem[7]

Let σ be any continuous, non-constant, bounded, and non-linear activation function. Let $C([a,b]^n)$ denote the space of continuous functions on the n -dimensional unit hypercube $[a,b]^n$. Then, for any continuous function $f \in C([a,b]^n)$ and any $\epsilon > 0$, there exists a feedforward neural network function of the form,

$$F(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + \theta_j)$$

where α_j ,w_j , and θ_j are real-valued parameters (weights and biases), such that,

$$\sup_{x \in [a,b]^n} |f(x) - F(x)| < \epsilon$$

We can make a simple extension to the above theorem to obtain a form that is better suited to the context of our problem. Specifically, this involves transforming the continuous functions here into a loss function (MSE) for the discrete data in our problem.

Let's pick $x_1, x_2, x_3, x_4 \cdots x_n \in [a, b]^2$, in our question $x_i = (K_i, T_i)$
For $\forall\epsilon'$, we pick: $\epsilon = \sqrt{\epsilon'}$. We let $F(x)$ here, be the real-world price of an option.
By Universal Approximation Theorem:

$$\sup_{x\in[a,b]^n} |f(x) - F(x)| < \sqrt{\epsilon'}$$
$$\Rightarrow |f(x_i) - F(x_i)| \leq \sup_{x\in[a,b]^n} |f(x) - F(x)| < \sqrt{\epsilon'}$$

for $i = 1, 2, 3 \cdots n$

We consider these x_i as the data in the training dataset, and $f(x_i)$ as the predicted values obtained from the model through forward propagation. Now, let's calculate the Mean Squared Error (MSE) for this neural network model,

$$\text{MSE } = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - F(x_i))^2 \leq \frac{1}{n}\sum_{i=1}^{n}(\sqrt{\epsilon'})^2 = \epsilon'$$

In simple terms, this proves that **for any arbitrarily small error ϵ, there exists a neural network model such that the loss function of the neural network is smaller than that error.**

The above conclusion is intriguing: it suggests that we can treat the aforementioned arbitrarily small error as the loss function of the Black-Scholes formula. This indicates that there exists a sufficiently well-designed neural network model capable of surpassing the performance of the Black-Scholes formula. Thus, our problem now shifts to how to construct a sufficiently well-designed neural network model as mentioned above.

1.2.3. The conditions that affect the performance of a neural network

These are the main factors affecting the loss function of the neural network, and they are also the primary focus of our efforts in this project:[8]

1. Definition of Forward Propagation

We adopted two types of neural network structures: an initial one and an arbitrage-free one. In addressing the no-arbitrage problem, we attempted to implement the neural network developed by Dugas et al.[9]

Initial: $\hat{\mathcal{N}} = \left\{f(x) = w_0 + \sum_{i=1}^{H} w_i \cdot h\left(b_i + \sum_j v_{ij}x_j\right)\right\}$

Arbitrage-free: $\hat{\mathcal{N}}_{++}^{c,n} = \left\{f(x) = e^{w_0} + \sum_{i=1}^{H} e^{w_i}\left(\prod_{j=1}^{c}\zeta\left(b_{ij} + e^{v_{ij}}x_j\right)\right)\left(\prod_{j=c+1}^{n} h\left(b_{ij} + e^{v_{ij}}x_j\right)\right)\right\}$

2. Training Dataset and Testing Dataset

We used Pandas to preprocess over 2,200 data points for Apple and Tesla, and randomly split them into training and testing datasets. We achieve the fitting through a training dataset and test the fitting results using a test dataset. We will explain this in detail later and demonstrate the loss function of our model on both the training and test datasets separately.

3. Optimization Algorithm and Computational Hardware Efficiency

Fortunately, PyTorch, developed by Facebook's AI Research lab, provides the Adam optimization algorithm, allowing us to train neural networks efficiently.

Our detailed implementation will be presented later in the report. However, returning to the comparison with the Black-Scholes formula, we believe that in order to achieve a model superior to the Black-Scholes formula, certain conditions must be met, such as having a sufficiently large training dataset, highly efficient optimization algorithms, and advanced computational hardware capabilities. These are conditions we may not possess. Additionally, the Black-Scholes Formula has certain limitations that are difficult to overcome. For example, one of its issues is that the model makes assumptions that don't align with the real world, such as the assumption of no transaction costs and constant volatility in the market. In conclusion, returning to the question raised by Dr. Azmoodeh, our current answer is:

"*Our model does indeed have the potential to surpass the Black-Scholes formula, as ensured by the Universal Approximation Theorem. Furthermore, the Black-Scholes model has certain limitations in its assumptions. Therefore, adopting a machine learning approach for option pricing, while moving away from the assumptions of traditional models, may offer a more market-relevant and innovative solution.*"

In the following sections of the report, we will also provide a brief comparison between our model and the Black-Scholes model.

1.2.4. Neural network with an arbitrage-free structure

With the help of the paper we referred to[10], we implemented a neural network model with an arbitrage-free structure. Specifically, it fulfilled the three important conditions we mentioned:

$$f \geq 0(\mathrm{NA}_0), \frac{\partial f}{\partial K} \leq 0(\mathrm{NA}_1), \frac{\partial f}{\partial T} \geq 0(\mathrm{NA}_2), \frac{\partial^2 f}{\partial K^2} \geq 0(\mathrm{NA}_3)$$

In fact, if the training data is sufficiently high-quality (meeting all the conditions mentioned above) and the neural network is trained effectively, an arbitrage-free structure would NOT be necessary (according to the Universal Approximation Theorem). In reality, we can view the arbitrage-free structure as a kind of insurance; it ensures that even if some conditions for arbitrage-free pricing are not met in the training data, the model's final results can still be guaranteed to be arbitrage-free. In subsequent tests, we found that the arbitrage-free structure significantly affects the convergence of the loss function. However, this structure can prevent the occurrence of arbitrage in the pricing model, thus avoiding substantial economic losses.

2. Implementation Methodology

2.1. Data generation

2.1.1. How to import data into python

The first step we should do was to look for eligible data online, Then we saved the data in Excel format, in this program we were looking for Apple and Tesla option prices. After that, we read option data from the Excel file, filtered the data for specific tickers (Tesla and Apple) and options (Put and call). The data used in our training are call options on Apple.

Here we have written a code example for reading data with two warnings:

- Openpyxl package required which is used to process Excel file.
- When using this method,we should correct the path to the path of the spreadsheet on the current computer.

```
import pandas as pd
from datetime import datetime

def read_option_data(file_path, ticker, current_date):

    # Read the Excel file
    df = pd.read_excel(file_path)

```

```
    # Filter data for the specified ticker and for Call options
    df_filtered = df[(df['Ticker'] == ticker) & (df['Type'] == 'Call')].copy()

    # Convert Expiration(T) to datetime
    df_filtered['Expiration(T)'] = pd.to_datetime(df_filtered['Expiration(T)'])

    # Calculate the difference in days between Expiration(T) and current_date
    df_filtered['Days to Expiration'] = (df_filtered['Expiration(T)'] - current_date).dt.days

    # Extract K, Days to Expiration and T columns to form one matrix
    K_T_matrix = df_filtered[['Strike(K)', 'Days to Expiration']].values

    # Extract S column to form another matrix
    c_matrix = df_filtered[['Last Option Price (c)']].values

    return K_T_matrix.T, c_matrix.T
```

This Python script reads option data from an Excel file, and calculates the days until expiration from the current date. It then returns two matrices: one containing the strike prices and the days until expiration, and the other containing the most recent option prices. The main components of the code are as follows:

- **Libraries:** The script imports `pandas` to handle data processing and `datetime` for working with dates.
- **Function Definition** (`read_option_data`)
  - **Parameters:** The function takes three inputs:
    1. `file_path`: The path to read the Excel file.
    2. `ticker`: The stock symbols used to filter the data.
    3. `current_date`: The current date used to calculate days until expiration. We choose 2024.6.26 as the current data.
  - **Data Reading:**
    The data is loaded from the provided Excel file using `pandas.read_excel()`.
  - **Date Conversion and Calculation:**
    The expiration dates are converted into `datetime` format. Then The difference between the expiration date and the current date is calculated to determine the days until expiration.
  - **Matrices:**
    The first matrix (`K_T_matrix`) contains the strike prices and the days until expiration. The second matrix (`c_matrix`) contains the most recent option prices.
  - **Return Values:**
    The function returns the transposed versions of both matrices for further processing or analysis.

Then, we know that we should have:

$$f \geq 0, \quad \frac{\partial f}{\partial K} \leq 0, \quad \frac{\partial f}{\partial T} \geq 0, \quad \frac{\partial^2 f}{\partial K^2} \geq 0$$

So we take $\mathbf{K}$ as the first element, and take $\mathbf{K}$ as $-\mathbf{K}$ in this code.

Through python, using `read_option_data` method to read the data from excel, get two matrices that read the data for the function methods used subsequently:

One for strikes and days to expiration:

$$\begin{pmatrix} X_1 & X_2 & \dots & X_n \end{pmatrix} = \begin{pmatrix} (-K_1, T_1) & (-K_2, T_2) & \dots & (-K_n, T_n) \end{pmatrix}$$

$$\begin{pmatrix} X_1^T & X_2^T & \dots & X_n^T \end{pmatrix} = \begin{pmatrix} -K_1 & -K_2 & \dots & -K_n \\ T_1 & T_2 & \dots & T_n \end{pmatrix}$$

One for stock prices:

$$(C_1 \ C_2 \ \dots \ C_n)$$

2.1.2. How to randomly splits the data into training and testing datasets

How to divide all our data into training set and test set is very important, for example, there are 30 days of options data, we need to take 20 samples of them, but we can not just take the first 20 days, because this does not reflect the changes in the last ten days, although the final model can be fitted well by the first twenty, but there is no prediction for the last ten days, which is going to cause a lot of This will cause a big error, because if I choose a sample this way, my changes for the next ten days can be whatever I want them to be.

In our project, the price of the option is closely related to the delivery time, so we should design a method that can randomly split the whole dataset into a training set and a test set, with the following code example.

```python
import torch

def split_train_test(matrix1, matrix2, ratio):

    # Ensure the input ratio is between 0 and 1
    if not (0 < ratio < 1):
        raise ValueError("The ratio must be a float between 0 and 1")

    # Calculate the number of columns for training based on the ratio
    n = int(matrix1.shape[1] * ratio)

    # Ensure the input matrices have enough columns
    if matrix1.shape[1] < n or matrix2.shape[1] < n:
        raise ValueError("The input matrices must have enough columns to split based on the ratio")

    # Randomly shuffle the indices of columns
    indices = torch.randperm(matrix1.shape[1])
    train_indices = indices[:n]
    test_indices = indices[n:]

    # Split the first matrix
    train_matrix1 = matrix1[:, train_indices]
    test_matrix1 = matrix1[:, test_indices]

    # Split the second matrix
    train_matrix2 = matrix2[:, train_indices]
    test_matrix2 = matrix2[:, test_indices]

    return train_matrix1, test_matrix1, train_matrix2, test_matrix2
```

This function splits two matrices (`matrix1`, `matrix2`) into training and testing sets based on a provided ratio.

- **Libraries:** The script imports `pandas` to handle data processing and `datetime` for working with dates.
- **Function Definition** (`split_train_test`)
  - **Parameters:** The function takes three inputs:
    1. `matrix1 (torch.Tensor)`: The first input matrix.
    2. `matrix2 (torch.Tensor)`: The second input matrix.
    3. `ratio (float)`: The ratio of columns to include in the training data (0 < ratio < 1).
  - **Shuffling and Splitting:**
    The columns of both matrices are randomly shuffled to ensure randomness in the split. Based on the ratio, the columns are split into training and testing sets.
  - **Consistency:**
    Both matrices are split using the same indices, ensuring corresponding columns between the two matrices remain aligned.
  - **Return Values:**
    Four matrices are returned:
    1. `train_matrix1`, `test_matrix1` (for `matrix1`)
    2. `train_matrix2`, `test_matrix2` (for `matrix2`)

In general, the training dataset is seventy percent of the overall dataset, i.e. the `ratio` is 0.7, but due to our limited access to data, the size of the Apple dataset for training is close to 1,000 in total, so the `ratio` is set to 0.9.

It is important to note that the total dataset is divided into two parts: the **Training set** and the **Test set**:

- The **Train Set** is used to train the model, a collection of samples used to train the neural network, through the training set, the neural network can learn the characteristics and patterns of the data. So that the model can make correct predictions about the input data.
- Whereas the **Test Set** is a collection of samples used to evaluate the performance of the neural network on new data, independent of the training set, to check if the model performs well on unseen data. This can help determine if the model is overfitting or underfitting, and the loss function is usually used to assess how well the model fits the test set.

Mean Squared Error (MSE) is one of the commonly used loss functions in neural networks. It is mainly used to assess the difference between the predicted and true values of a model.

The mean square error is defined as the average of the squares of the differences between the predicted and true values[11].

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

- y_i denotes the first true value for the i sample.
- $\hat{y}_i$ denotes the first predict value for the i sample.
- n denotes the number of samples.

A detailed description of the loss function will be mentioned in the subsequent backward propagation section.

2.2. Forward propagation

2.2.1. Basic model

Now we are going to give a simple example to demonstrate how forward propagation works in neural networks.

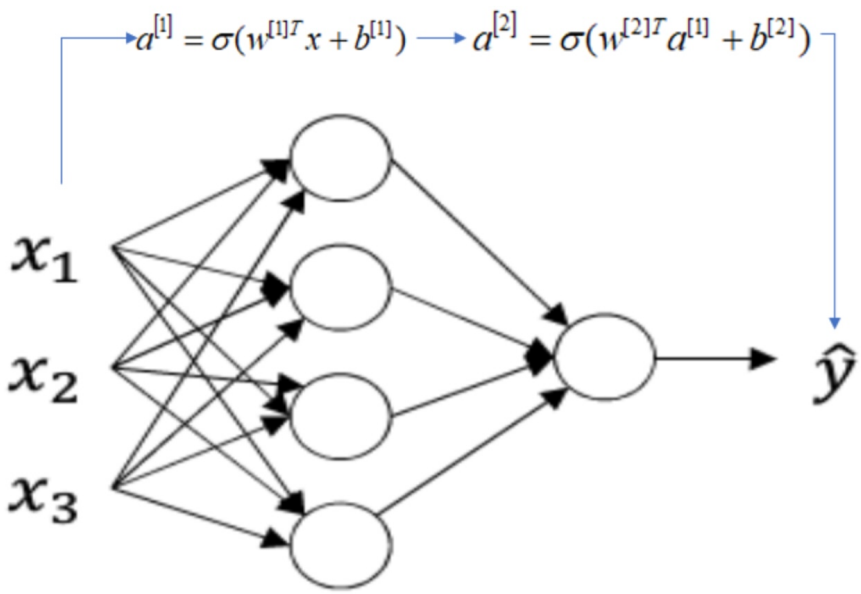


Figure 7. *Forward propagation.*

It can be noticed that this is a neural network with three layers, the Input layer at the beginning, the Hidden layer in the middle and the Output layer at the end.

Having three nodes in the input layer, The input layer passes external data to the neural network. The input layer usually does not perform any calculations that it is just a matter of taking the input data directly to the next layer. We can express it in the following equation:

$$a^{[0]} = x = (x_1, x_2, x_3)$$

In the hidden layer, the neuron performs a weighted summation of the data from the input layer and adds a bias term with the following formula:

$$z^{[i]} = \sum_{j=1}^{n} w^{[j]} x + b^{[i]}$$

Each neuron in the hidden layer performs a similar calculation as follows:

$$z^{[1]} = \begin{bmatrix} w_1^{[1]} \\ w_2^{[1]} \\ w_3^{[1]} \\ w_4^{[1]} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} \cdot x + b_1^{[1]} \\ w_2^{[1]T} \cdot x + b_2^{[1]} \\ w_3^{[1]T} \cdot x + b_3^{[1]} \\ w_4^{[1]T} \cdot x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

In summary, we can express this in the following equation:

$$\begin{aligned} z_1^{[1]} &= w_1^{[1]T} x + b_1^{[1]} \\ z_2^{[1]} &= w_2^{[1]T} x + b_2^{[1]} \\ z_3^{[1]} &= w_3^{[1]T} x + b_3^{[1]} \\ z_4^{[1]} &= w_4^{[1]T} x + b_4^{[1]} \end{aligned}$$

The purpose of the activation function is to allow the neural network to fit complex nonlinear functions, combine the outputs of the linear combinations $z^{[i]}$ translated into a nonlinear output, Then we can have the following expression:

$$\begin{aligned} a_1^{[1]} &= \sigma(z_1^{[1]}) \\ a_2^{[1]} &= \sigma(z_2^{[1]}) \\ a_3^{[1]} &= \sigma(z_3^{[1]}) \\ a_4^{[1]} &= \sigma(z_4^{[1]}) \end{aligned}$$

Where we can get:

$$\begin{aligned} a^{[1]} &= \sigma(w^{[1]T} x + b^{[1]}) \\ &= (a_1^{[1]}, a_2^{[1]}, a_3^{[1]}, a_4^{[1]}) \end{aligned}$$

Similarly, we can get:

$$\hat{y} = a^{[2]} = \sigma(w^{[2]T} x + b^{[2]})$$

In our project, we build a neural network model with only two input nodes, with uncertain hidden layers.

$$\hat{y} = c(k, T)$$

2.2.2. Initial model

At the beginning of our project, we used the following model[10]:

$$\hat{\mathcal{N}} = \left\{ f(x) = w_0 + \sum_{i=1}^{H} w_i \cdot h\left(b_i + \sum_j v_{ij} x_j \right) \right\}$$

In the initial phase of model development, we used a basic artificial neural network (ANN) with a fully connected structure. The following Python code illustrates the structure and training process of the initial ANN model:

```python
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam

def build_model():
    # Initialize Sequential model
    ANN = Sequential()

    # Add layers
    ANN.add(Dense(10000, input_dim=2, activation='relu'))
    ANN.add(Dense(1000, activation='relu'))
    ANN.add(Dense(100, activation='relu'))
    ANN.add(Dense(10, activation='relu'))
    ANN.add(Dense(1, activation='linear'))

    # Compile model
    ANN.compile(loss='mean_squared_error', optimizer=Adam(learning_rate=0.0001))

    # Train the model
    ANN.fit(X.T, Y.T, epochs=1000, batch_size=32, verbose=1)

    return ANN
```

2.2.3. Modified model

Since our project is required to be under the no-arbitrage principle, we improve the model as follows[10]:

$$c, n\hat{\mathcal{N}}_{++} = \left\{ f(x) = e^{w_0} + \sum_{i=1}^{H} e^{w_i} \left(\prod_{j=1}^{c} \zeta\left(b_{ij} + e^{v_{ij}} x_j \right) \right) \left(\prod_{j=c+1}^{n} h\left(b_{ij} + e^{v_{ij}} x_j \right) \right) \right\}$$

Based on this model expression, we can draw a neural network diagram for this model:

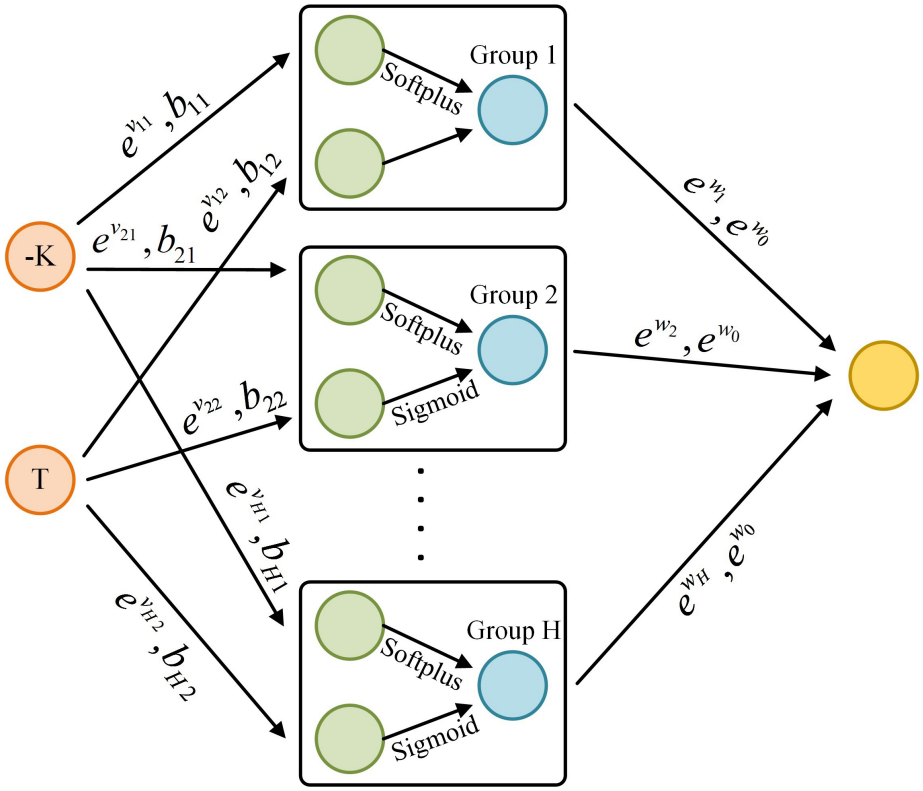


Figure 8. *Modified model.*

Compared to the initial model, it makes the following improvements:

- Connections from the input level to individual groups and between groups are weighted by exponential operations, which ensures that NA_0 to NA_4 are greater than 0.
- This neural network model demonstrates an architecture with multiple Groups, It is to map the two nodes (-K and T) in the input layer to the second layer, and then use two different activation functions, Softplus and Sigmoid, respectively, to nonlinearly vary the data, and finally the product of the two is used as a new node, which is passed to the next layer of hidden layer.

After making the above improvements, the no-arbitrage principle can be well satisfied, and based on this model, we wrote the following code example.

The first step we need to take is to prepare the two activation functions required for the forward propagation[9], the reason why we choose these two activation functions is that the sigmoid $h(s)$ has a positive first derivative, its primitive, which we call Softplus, is convex.

- **Sigmoid:**

$$h(s) = \frac{1}{1 + e^{-s}}$$

- **Softplus:**

$$\zeta(s) = \ln(1 + e^{s})$$

- **Note that:**

$$\frac{d\zeta(s)}{ds} = h(s) = \frac{1}{1 + e^{-s}}$$

```python
import torch
import torch.nn.functional as F

# Define the softplus function
def softplus(x):
    return F.softplus(x)
```

```

# Define the sigmoid function
def sigmoid(x):
    return torch.sigmoid(x)
```

This is what our parameters look like:

$$V = \begin{pmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \\ \vdots & \vdots \\ v_{H1} & v_{H2} \end{pmatrix} \quad b = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ \vdots & \vdots \\ b_{H1} & b_{H2} \end{pmatrix}$$

$$w = (w_1, w_2, w_3, w_4, \cdots, w_H)$$

$$w_0 \in \mathbb{R}$$

```
def forward_propagation(V, b, w, w0, x):

    # Matrix Reshaping
    V_first_column = V[:, 0].reshape(-1, 1)
    V_second_column = V[:, 1].reshape(-1, 1)
    x_first_row = x[0, :].reshape(1, -1)
    x_second_row = x[1, :].reshape(1, -1)
    b_first_column = b[:, 0].reshape(-1, 1)
    b_second_column = b[:, 1].reshape(-1, 1)

    # Matrix Multiplications and Activation Functions
    parameters_ready_to_apply_softplus = torch.mm(torch.exp(V_first_column), x_first_row) + b_first_column
    parameters_ready_to_apply_sigmoid = torch.mm(torch.exp(V_second_column), x_second_row) + b_second_column

    after_softplus = softplus(parameters_ready_to_apply_softplus)
    after_sigmoid = sigmoid(parameters_ready_to_apply_sigmoid)

    # Layer Combination
    layer1 = after_softplus * after_sigmoid

    # Final Output Calculation
    result = torch.mm(torch.exp(w).unsqueeze(0), layer1) + torch.exp(w0)

    return result
```

- **Function Definition** (`forward_propagation`)
  Parameters: The function takes five inputs:

  1. V: Weight matrix for the first layer.
  2. b: Bias matrix for the first layer.
  3. w: Weight vector for the output layer.
  4. w_0: Bias scalar for the output layer.
  5. x: Input data matrix.

- **Matrix Multiplications and Activation Functions**:
  - The function calculates two intermediate results using matrix multiplication and the exponential of the weight and input matrices.
  - These intermediate values are processed through the **Softplus** and **Sigmoid** activation functions.
  - `parameters_ready_to_apply_softplus` applies the Softplus activation to the first column of `V` and `x`, while `parameters_ready_to_apply_sigmoid` applies the Sigmoid activation to the second column of `V` and `x`.

- **Layer Combination**:
  - The processed results of the two activation functions (`after_softplus` and `after_sigmoid`) are multiplied element-wise to produce `layer1`.

We introduce the forward propagation and the neural network architecture below. Next we aim to contribute to minimize the loss value during the training, which is contained in the backward propagation. One of the most significant process is finding the minimal loss value of the loss function. Based on the conventional approach which called gradient descent, we will apply a more advanced one called adam, actually it's a We first explain the relationship between optimization and deep learning.[12]

2.3. Optimization Algorithms

This section will discuss optimization and deep learning, introducing several different optimization algorithms and how to apply them in our project. For deep learning problems, we usually start by defining a loss function. Once we have the loss function, we can use optimization algorithms to try to minimize the loss.[13]

2.3.1. Back-propogation

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector.[12]
Here is a figure illustration.

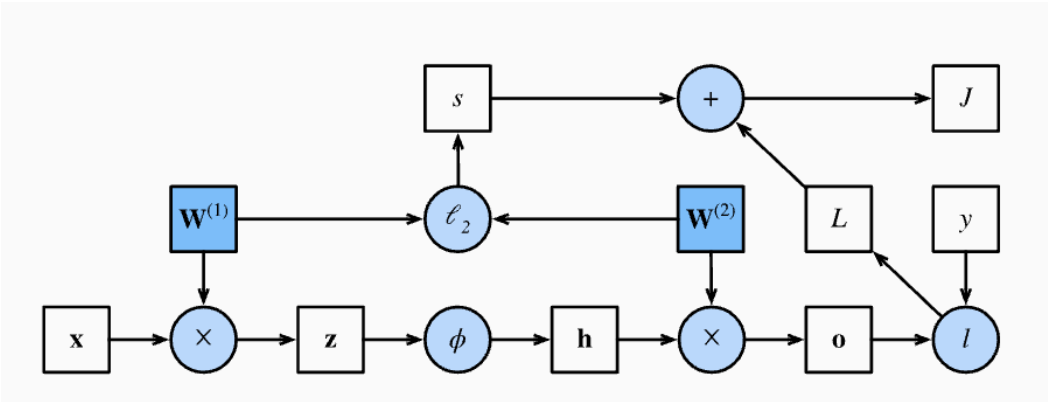


Figure 9. *back-propogation.*

Assume the objective function $J = L + s$, where L represents the loss term and s the regularization term. The parameters of a single-hidden-layer network are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The goal of back-propagation is to compute the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To achieve this, we apply the chain rule and calculate the gradients for each intermediate variable and parameter in turn. The calculation order is opposite to that in forward propagation, as we need to start from the output of the computational graph and work backward toward the parameters.

2.3.2. Loss functions

There are a few kinds of loss function, but we will focus on the mean squared error and the binary cross-entropy loss.

Before thatwe first introduce some basic properties of the loss funcitons.

Properties of Loss Functions

Loss functions possess several essential properties that should be considered when choosing one for a particular task:[14]

1. **Convexity**: A loss function is convex if any local minimum also serves as the global minimum. Convex loss functions are preferred for their compatibility with gradient-based optimization methods.
2. **Differentiability**: A differentiable loss function has a continuous derivative with respect to the model's parameters. Differentiability is crucial for enabling gradient-based optimization.
3. **Robustness**: Loss functions should be resilient to outliers, meaning they are not overly influenced by a small number of extreme values.
4. **Smoothness**: A smooth loss function has a continuous gradient without abrupt changes or spikes.
5. **Sparsity**: A loss function promoting sparsity encourages the model to produce sparse outputs, which is especially useful in high-dimensional data settings or when the number of significant features is limited.
6. **Monotonicity**: A loss function is monotonic if its value decreases as the model's predictions approach the true output. This ensures that the optimization is progressing towards the optimal solution.

Mean Squared Error (MSE)

The MSE measures the average of the squared differences between the predicted values and the true values. The MSE loss function can be defined mathematically as [14]:

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

The MSE loss function has the following properties:

- **Non-negative:** MSE is always non-negative because the differences between the predicted and actual values are squared. A value of 0 indicates a perfect fit.
- **Sensitive to outliers:** MSE is a quadratic function of the prediction errors, placing more emphasis on larger errors.
- **Differentiable:** MSE is smooth and continuous, allowing efficient gradient computation.
- **Convex:** MSE has a unique global minimum, simplifying optimization.
- **Scale-dependent:** MSE depends on the target variable's scale; RMSE or MSPE are often used for comparison.

Binary Cross-Entropy Loss (BCE)

Binary Cross-Entropy Loss is used in binary classification tasks to quantify the difference between the predicted probability and the actual class label. The cross-entropy measures the difference between two probability distributions.

In binary classification, where classes are labeled as 0 or 1, the BCE loss for an individual prediction can be expressed as:

$$L(y, p) = -(y\log(p) + (1-y)\log(1-p))$$

This formula is intuitive:

$$L(y, p) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{if } y = 0 \end{cases}$$

where y is the true class label and p is the predicted probability of the class being 1. The BCE loss minimizes when the predicted probability p aligns with the actual label y.

BCE is beneficial as it's differentiable, providing a probabilistic interpretation for gradient-based optimization. It penalizes high-confidence incorrect predictions, supporting logistic regression's goal to use BCE as its loss function. [14]

2.3.3. Optimization and deep learning

Optimization is at the heart of deep learning, ensuring that neural networks can learn effectively from data by minimizing loss functions. The choice of optimizer and tuning its hyperparameters like learning rate are critical for the successful training of deep neural networks. Advanced methods like Adam and learning rate scheduling help improve the optimization process, allowing models to converge more efficiently. [15]

2.3.4. Adam optimizer

We introduce Adam, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments.[13]Firstly, let's introduce some prior knowledge:

- **SGD is more effective than regular GD in solving optimization problems.**
  Gradient descent(GD) performs a parameter update for the entire dataset, computing the gradient of the cost function $J(\theta)$ with respect to all training examples:

  $$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

  Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:[16]

  $$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

  Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

- **Using a larger set of observations in a small batch to provide an additional efficiency boost through quantification, key for high-efficiency multi-machine.**

- **A mechanism to summarize historical gradients for faster convergence.**
  Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It increases for dimensions whose gradients point in the same direction and reduces updates for dimensions whose gradients change directions. [17]As a result, we gain faster convergence and reduced oscillation.[16]

  $$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

  $$\theta = \theta - v_t$$

- **Achieved efficient computation of the preprocessor by shrinking each coordinate**
  AdaGrad (Adaptive Gradient Algorithm) adjusts the learning rate for each parameter θ_i based on the cumulative sum of past squared gradients. Parameters that receive large gradients will have their learning rates decrease, while parameters with smaller gradients will retain larger learning rates.

The update rule is as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

where G_t is the sum of the squared gradients up to time t, η is the global learning rate, and ϵ is a smoothing term to avoid division by zero. [18]

- **Separated the shrinking of each coordinate by adjusting the learning rate**
  RMSProp Algorithm is an optimization algorithm that maintains a moving average of the squared gradients to adjust the learning rate for each parameter. The update rule for RMSProp is as follows:[19]

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$$

  where:

  - $E[g^2]_t$ is the exponentially decaying average of squared gradients,
  - γ is the decay rate (commonly set to 0.9),
  - g_t is the gradient at time step t,
  - η is the global learning rate (often set to 0.001),
  - ϵ is a small constant (e.g., 10^{-8}) to prevent division by zero.

Adam Algorithm [13] brings together various techniques into a powerful and efficient learning algorithm. It is very popular, especially in deep learning. However, it is not without issues. In particular, [20] demonstrated that Adam sometimes fails to converge due to poor variance control. In their improvement work,[21] proposed a variant called Yogi to address these issues. Below, we explore the Adam algorithm in more detail.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Figure 10. *Adam algorithm[13].*

The Adam optimizer can usually be called directly in deep learning frameworks, such as TensorFlow or PyTorch. Most frameworks have the Adam optimizer implemented internally, and users only need to configure the parameters to use it.

There are more things contained in the optimization algorithms which included in the backward propagation. For example, The application of regularization, learning rate adjustment, noise reduction in the data, and other methods to reduce the value of the loss function. Next, we will analysis the model.

3. Result

3.1. Initial VS Arbitrage-free

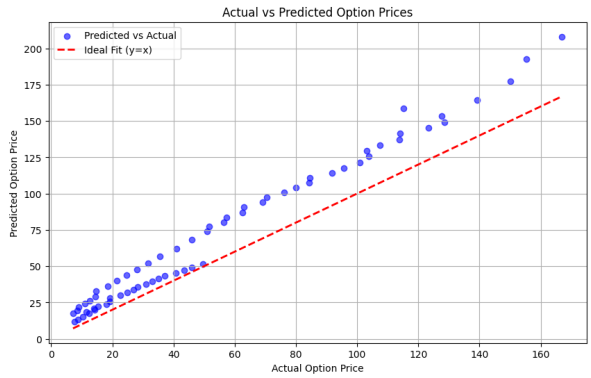


Figure 11. *Initial.*

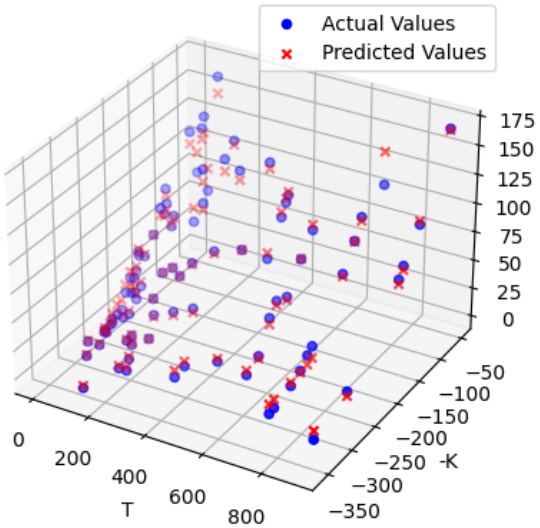


Figure 12. *Arbitrage-free.*

Here, we used two different methods to demonstrate the results of our model.

In Figure 11, blue dots represent the ratio between actual values and theoretical values, that is: $\frac{\tilde{c}_i}{c_i}$. We also plotted a red line($y = x$) to show the gap between the actual values and the model's predicted values. In Figure 12, we directly plotted the mapping relationship between K , T , and c .

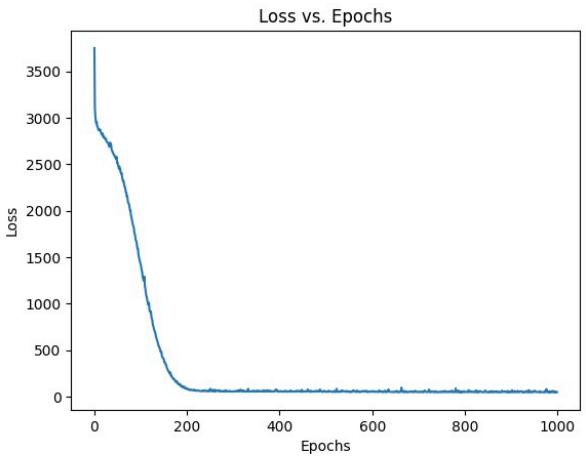


Figure 13. *Initial.*

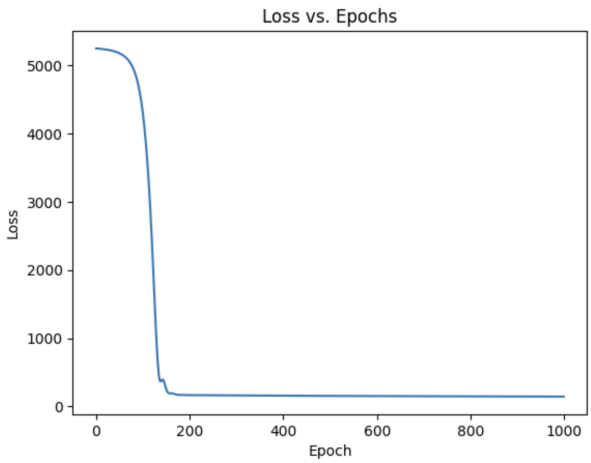


Figure 14. *Arbitrage-free.*

During the training process of these two models, we observed an interesting phenomenon: the final loss function of the initial model was lower than that of the modified model. Here are some possible reasons:

In real financial markets, there may indeed be some arbitrage opportunities that reflect market imperfections and complexity of pricing strategies. The **Initial model** has no constraints and is free to fit these data where arbitrage opportunities exist, so it is able to fit the training data more closely, resulting in a relatively small value of its losses.

In contrast, the **Arbitrage-free model** avoids arbitrage opportunities by imposing no-arbitrage constraints that force the model's outputs to conform to the fundamental principles of financial markets. As a result of these constraints, the Arbitrage-free model is restricted in its degrees of freedom when fitting the data, and therefore the value of losses will be relatively larger.

So, the **Initial model has smaller losses** because it is attempting to fit all the data perfectly, including those parts that represent imperfections in the market, which may contain exactly the arbitrage opportunities. The **Arbitrage-free model, on the other hand, has a larger loss** because it forces the model to not simply follow these arbitrage opportunities, ensuring that it meets the assumption that the market is arbitrage-free, thus limiting to some extent the degree of freedom to fit the data.

As a result, it is realistic to expect that the Initial model will typically have smaller losses than the Arbitrage-free model in the presence of market arbitrage opportunities.The Arbitrage-free model is more concerned with pricing rationality and the robustness of the financial markets, rather than just minimising losses.

So both the initial neural network and the modified one have their advantages and disadvantages:

- **Initial Model:**
  - We can find that this method is fast and effective in minimising losses, but may over-fit the data without considering real-world funding constraints.
  - But our initial model is ideally suited to minimising losses without worrying about the consequences of arbitrage.
- **Arbitrage-Free Model:**
  - It takes more time to converge and results in a higher final loss, but it ensures that the model is financially sound and follows no-arbitrage rules.
  - It is better suited for practical financial applications where the avoidance of arbitrage opportunities is a necessity, making it more robust and reliable for real-world use.

Therefore, we can conclude the following:

- If our goal is to achieve higher predictive accuracy in a stable market, the initial model may be preferable.
- If the goal is to ensure consistent and arbitrage-free pricing in dynamic and volatile markets, the arbitrage-free model is the better choice.

3.2. Loss function VS Quantity of Neurons

H	Loss
1	64.7333
2	62.379
3	45.1903
4	45.0423
5	44.711
10	44.7529
100	44.8589
1000	41.4791

Table 1. *Loss Function and H.*

In the figure, H represents the number of neuron groups in the neural network (see Section 2). We can observe that as H increases, the loss function gradually decreases. Furthermore, the Universal Approximation Theorem indicates that as H increases, the loss function will gradually converge to zero.

3.3. Arbitrage-free VS Black-Scholes

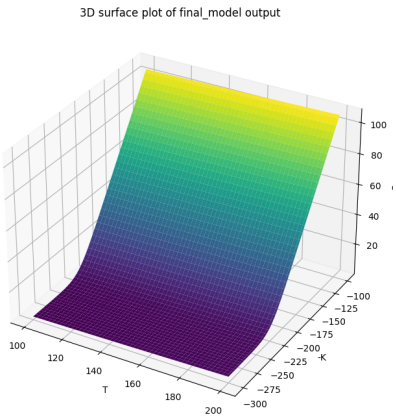


Figure 15. *Our model.*

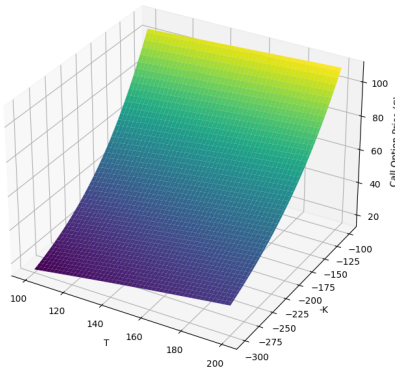


Figure 16. *BS Formula.*

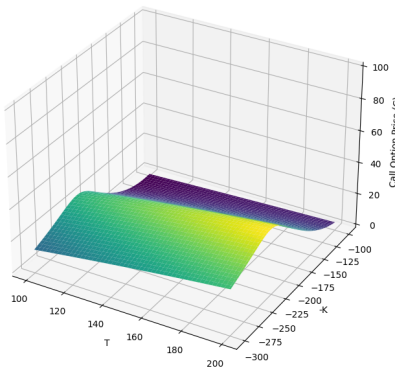


Figure 17. *Difference .*

- First, we can see that in most regions, the predictions of the two models are consistent, with option prices following the expected trend as expiry times increase and strike prices change.
- We can observe in the image above that there are some differences between the models under certain extreme conditions (e.g. relatively high strike prices and long expiry times). This suggests that the neural network model captures certain market characteristics that are not fully described by the Black-Scholes model. To take the simplest example, the Black-Scholes model assumes that asset prices are continuously changing and follow a Geometric Brownian Motion. However, in real markets, asset prices sometimes make sudden and large jumps, especially after important economic events or market news announcements, and this discontinuity cannot be described by the continuous process in the model.
- So from the three images above we can infer that if the market conditions are stable and the assumptions of the Black-Scholes model are met (e.g., normal volatility, frictionless market, etc.),

the difference between the two results is relatively small. However, under complex or extreme market conditions, the neural network model may have higher adaptability and accuracy.

4. Conclusion

At end the study of the strategy of European call options by using neural network, much like the ancient Greek philosopher Thales' foresight with olive presses to predict future market conditions and make informed decisions based on that prediction. Just as Thales secured a favourable rent price by paying a small premium upfront, investors today use call options to gain the right to buy an asset at a predetermined price in the future, aiming to profit from favourable market movements. Our efforts of using neural networks for arbitrage-free pricing can be a significant step forward in financial modeling. The main goal was to develop a pricing model that stays consistent with the no-arbitrage principle, while arbitrage opportunities may seem like quick ways to make profits, they can create inefficiencies in the market. Our model aims to maintain stability within its pricing predictions.

4.1. Role of Data and Optimization

One of the most important aspects of this project was how we handled data and optimization. By carefully managing the data generation and splitting, we ensured that the neural network was able to generalize well based on historical option data, although during this project the dataset used might not performed significantly regarding the efficiency and accuracy of the prediction due the small sample size, however the techniques such as forward/backward propagation and the use of the Adam optimizer further improved the model's convergence , striking a balance between speed and accuracy. In financial markets, where speed and precision are vital, these optimization strategies gave our neural network a significant advantage, allowing it to quickly find solutions that align with real-world conditions.

4.2. Challenges and Limitations

Note that we cannot use data from different stocks to train our neural network; instead, we must select a specific stock. In fact, our limitation lies in the number of data points for that particular stock. This means we need more data from the Apple stock market. While the project has demonstrated promising results, several challenges were encountered during the research process.

4.2.1. Limited Sample Size

Although our team has achieved the promising outcomes, several challenges occurred during the research. A key limitation was the small dataset, for both training and testing of our model we applied the dataset from Apple and Tesla, both are highly traded and representative of certain market behaviors, however, both dataset are relatively too small, for model working with such a limited dataset can lead to issues like under-fitting. With fewer epochs and smaller batch sizes used during training and testing, there's a risk that the neural network may not have captured the underlying patterns in the data effectively. This under-fitting could result in the model failing to fully learn the complex relationships within the options market, leading to less accurate predictions. Moreover, the lack of variety in the dataset could cause the model to over-fit to the specific characteristics of Apple's stock, limiting its applicability to broader market conditions.

4.2.2. Long Computational Time and High Resource Demands

Finally, the project required significant computational power and time. Neural networks, especially those working with large datasets or complex models, demand substantial processing capabilities. Training our model, even on a relatively small dataset, took considerable time. As the dataset size increases, the need for more efficient hardware and software solutions required. This aspect will be particularly important

for scaling the model to handle more extensive, real-world financial data. Future projects might require access to more powerful computing resources or cloud-based solutions to speed up training times.

4.3. Final Thoughts

In summary, this research has demonstrated the power of neural networks in arbitrage-free pricing, offering a flexible and accurate alternative to traditional models. While Black-Scholes will likely continue to play a role in financial modeling, the advantages of machine learning—particularly its ability to handle complexity and variability—position neural networks as a key tool for the future of finance. This project lays a strong foundation for further exploration and innovation in the field of financial engineering, and the potential applications of these technologies will only grow as markets become more complex.

References

[1] W. N. Goetzmann and K. G. Rouwenhorst, *The origins of value: The financial innovations that created modern capital markets*. Oxford University Press, USA, 2005.
[2] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.
[3] Z. Janková, "Drawbacks and limitations of black-scholes model for options pricing," *Journal of Financial Studies & Research*, vol. 2018, 2018.
[4] S. Kumar, A. Kumar, K. U. Singh, and S. K. Patra, "The six decades of the capital asset pricing model: A research agenda," *Journal of Risk and Financial Management*, vol. 16, no. 8, p. 356, 2023.
[5] A. Chertov, "How to exploit calendar arbitrage?" https://quant.stackexchange.com/questions/15215/how-to-exploit-calendar-arbitrage, 2014, [Online; accessed 29-October-2024].
[6] N. P. Outreach, "The nobel prize in physics 2024 - popular science background," https://www.nobelprize.org/prizes/physics/2024/popular-information/, 2024, [Online; accessed 29-October-2024].
[7] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.
[8] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[9] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia, "Incorporating second-order functional knowledge for better option pricing," *Advances in neural information processing systems*, vol. 13, 2000.
[10] C. Dugas, Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia, "Incorporating functional knowledge in neural networks," *Journal of Machine Learning Research*, vol. 10, no. 6, 2009.
[11] A. Trivedi. (2024, July) Mean squared error: Definition and formula. Accessed: 2024-10-29. [Online]. Available: https://www.analyticsvidhya.com/blog/2024/07/mean-squared-error/
[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: https://arxiv.org/abs/1412.6980
[14] J. Terven, D. M. Cordova-Esparza, A. Ramirez-Pedraza, E. A. Chavez-Urbiola, and J. A. Romero-Gonzalez, "Loss functions and metrics in deep learning," 2024. [Online]. Available: https://arxiv.org/abs/2307.02694
[15] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," 2018. [Online]. Available: https://arxiv.org/abs/1606.04838
[16] S. Ruder, "An overview of gradient descent optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1609.04747

[17] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html

[18] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research*, vol. 12, no. 7, 2011.

[19] G. Hinton, "Neural networks for machine learning - lecture 6a - overview of mini-batch gradient descent," http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, 2012, coursera Lecture.

[20] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryQu7f-RZ

[21] E. Kowalski, P. Michel, and W. Sawin, "Stratification and averaging for exponential sums: bilinear forms with generalized kloosterman sums," 2020. [Online]. Available: https://arxiv.org/abs/1802.09849

A. Appendix(Codes)

Model to Solve Non-arbitrage Pricing

This is a model established by the Neural Networks Group of the MATH391 course at the **University of Liverpool** for the year 2024, aimed at solving the no-arbitrage problem. The primary reference is: “Incorporating Functional Knowledge in Neural Networks.”

Author: ShiYao Gu, JieRui Li, LangMing Liang, MengLei Zhang, SiZhe Jiang, Qi Huang.

Project Supervisor: Youness Boutaib, Samira Amiriyan

Module Supervisor: Corina Constantinescu

Principle of the Model

According to the artical, the model we want to implement is:

$C = C(K, T)$ (Here, we are no interest in other parameters)

- C is the price of this call option
- K is the strick price
- T is the time to maturity

And here, in order to insure the property of "Non-arbitrage", we need the following property:

$$f \geq 0, \quad \frac{\partial f}{\partial x_1} \geq 0, \quad \frac{\partial f}{\partial x_2} \geq 0, \quad \frac{\partial^2 f}{\partial x_1^2} \geq 0$$

that is: (take $-K$)

$$f \geq 0, \quad \frac{\partial f}{\partial T} \geq 0, \quad \frac{\partial f}{\partial K} \leq 0, \quad \frac{\partial^2 f}{\partial K^2} \geq 0$$

Therefore, according to the artical: "Incorporating Functional Knowledge in Neural Networks". The following structure of NN will be used.

$$c, n\hat{\mathcal{N}}_{++} = \left\{ f(x) = e^{w_0} + \sum_{i=1}^{H} e^{w_i} \left(\prod_{j=1}^{c} \zeta \left(b_{ij} + e^{v_{ij}} x_j \right) \right) \left(\prod_{j=c+1}^{n} h \left(b_{ij} + e^{v_{ij}} x_j \right. \right. \right.$$

- Notice that here, we only have two parameters, that is K, T, thus we should take $n = 2, c = 1$

More detailed structure is shown on the PPT.

Let’s now take a look at our upcoming implementation plan:

- Step01: Define the forward propagation

- Step02: Import dataset
- Step03: Randomly splits training and testing datasets
- Step04: Optimize the model using PyTorch
- Step05: Make prediction model

Here's all the package we need:

In [1]:
```
# pip install torch
# pip install pandas
# pip install openpyxl
```

Step01: Define forward propagation

The first step we need to take is to prepare the two activation functions required for the forward propagation.

- sigmoid: $h(s) = 1/\left(1+e^{-s}\right)$
- softplus: $\zeta(s) = \ln(1+e^{s})$

In [2]:
```
import torch
import torch.nn.functional as F

def softplus(x):
    return F.softplus(x)

def sigmoid(x):
    return torch.sigmoid(x)
```

Based on the above model, we provide the implementation method for forward propagation.

$$c, n\hat{\mathcal{N}}_{++} = \left\{ f(x) = e^{w_0} + \sum_{i=1}^{H} e^{w_i} \left(\prod_{j=1}^{c} \zeta\left(b_{ij} + e^{v_{ij}} x_j\right) \right) \left(\prod_{j=c+1}^{n} h\left(b_{ij} + e^{v_{ij}} x_j \right. \right. \right.$$

This is what our parameters look like: (Pay very very much attention to deminsion)

$$V = \begin{pmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \\ \vdots & \vdots \\ v_{H1} & v_{H2} \end{pmatrix} \quad b = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ \vdots & \vdots \\ b_{H1} & b_{H2} \end{pmatrix}$$

$w = (w, w_2, w_3, w_4 \cdots w_H)$

$w_0 \in R$

In [3]:
```
def forward_propagation(V, b, w, w0, x):
    """
    Perform forward propagation using PyTorch.

    Arguments:
    V -- weight matrix for the first layer
    b -- bias matrix for the first layer
    w -- weight vector for the output layer
    w0 -- bias scalar for the output layer
    x -- input data matrix

    Returns:
    result -- the output of the forward propagation
    """
    V_first_column = V[:, 0].reshape(-1, 1)
    V_second_column = V[:, 1].reshape(-1, 1)
    x_first_row = x[0, :].reshape(1, -1)
    x_second_row = x[1, :].reshape(1, -1)
    b_first_column = b[:, 0].reshape(-1, 1)
    b_second_column = b[:, 1].reshape(-1, 1)

    parameters_ready_to_apply_softplus = torch.mm(torch.exp(V_first_colum
    parameters_ready_to_apply_sigmoid = torch.mm(torch.exp(V_second_colum

    after_softplus = softplus(parameters_ready_to_apply_softplus)
    after_sigmoid = sigmoid(parameters_ready_to_apply_sigmoid)

    layer1 = after_softplus * after_sigmoid

    result = torch.mm(torch.exp(w).unsqueeze(0), layer1) + torch.exp(w0)

    return result
```

Define the Lost Function

Here we just compute MSE:

$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(AL_i - y_i)^2$

In [4]:
```
def mean_squared_error(AL, Y):
    """
    Calculate Mean Squared Error (MSE) using PyTorch.

    Parameters:
    AL -- predicted values tensor
    Y -- true labels tensor

    Returns:
    mse -- mean squared error tensor
    """
    mse = torch.mean((AL - Y) ** 2)
    return mse
```

Step02: Import Dataset

Here we have written a method for reading data.

Warnings:

- openpyxl package required
- When using this method, please correct the path to the path of the spreadsheet on the current computer.

In [5]:
```
import pandas as pd
from datetime import datetime

def read_option_data(file_path, ticker, current_date):

    """
    Reads option data from an Excel file, filters it based on the given t
    calculates the days to expiration from the current_date, and returns
    and stock prices.

    Parameters:
    file_path (str): The path to the Excel file.
    ticker (str): The ticker symbol to filter the data.
    current_date (datetime): The current date for calculating days to exp

    Returns:
    tuple: Two matrices, one for strikes and days to expiration, and one
    """


    # Read the Excel file
    df = pd.read_excel(file_path)

    # Filter data for the specified ticker and for Call options
    df_filtered = df[(df['Ticker'] == ticker) & (df['Type'] == 'Call')].c

    # Convert Expiration(T) to datetime
    df_filtered['Expiration(T)'] = pd.to_datetime(df_filtered['Expiration

    # Calculate the difference in days between Expiration(T) and current_
    df_filtered['Days to Expiration'] = (df_filtered['Expiration(T)'] - c

    # Extract K, Days to Expiration and T columns to form one matrix
    K_T_matrix = df_filtered[['Strike(K)', 'Days to Expiration']].values

    # Extract S column to form another matrix
    c_matrix = df_filtered[['Last Option Price (c)']].values

    return K_T_matrix.T, c_matrix.T
```

Then, we know that we should have :

$$f \geq 0, \quad \frac{\partial f}{\partial T} \geq 0, \quad \frac{\partial f}{\partial K} \leq 0, \quad \frac{\partial^2 f}{\partial K^2} \geq 0$$

So we take K as the first element, and take K as $-K$

In [6]:
```
def negate_first_row(matrix):
    """
    Negates the elements of the first row of the given matrix.
```

```
    Parameters:
    matrix (numpy.ndarray): The input matrix whose first row elements wil

    Returns:
    numpy.ndarray: The modified matrix with the first row elements negate
    """

    # Negate the first row of the matrix
    matrix[0] = -matrix[0]

    return matrix
```

Next, we read the relevant data, but in this program, we only analyzed the data related to Apple Inc.

In [7]:
```
x_apple, y_apple = read_option_data('/Users/shiyaogu/Documents/Modules in
x_tesla, y_tesla = read_option_data('/Users/shiyaogu/Documents/Modules in
x_apple_final = torch.tensor(negate_first_row(x_apple), dtype = torch.flo
x_tesla_final = torch.tensor(negate_first_row(x_tesla), dtype = torch.flo
y_apple_final = torch.tensor(y_apple, dtype = torch.float32)
y_tesla_final = torch.tensor(y_tesla, dtype = torch.float32)
```

Step03: Randomly splits training and testing datasets

Here, we have improved the previous method of separating training data and test data. We have adopted a method of randomly splitting the data.

In [8]:
```
import torch

def split_train_test(matrix1, matrix2, ratio):
    """
    Randomly splits two given matrices into training and testing datasets

    Parameters:
    matrix1 (torch.Tensor): The first input matrix.
    matrix2 (torch.Tensor): The second input matrix.
    ratio (float): The ratio of columns to include in the training data (

    Returns:
    tuple: Four matrices: train_matrix1, test_matrix1, train_matrix2, tes
    """
    # Ensure the input ratio is between 0 and 1
    if not (0 < ratio < 1):
        raise ValueError("The ratio must be a float between 0 and 1")

    # Calculate the number of columns for training based on the ratio
    n = int(matrix1.shape[1] * ratio)

    # Ensure the input matrices have enough columns
    if matrix1.shape[1] < n or matrix2.shape[1] < n:
        raise ValueError("The input matrices must have enough columns to

    # Randomly shuffle the indices of columns
```

```
    indices = torch.randperm(matrix1.shape[1])
    train_indices = indices[:n]
    test_indices = indices[n:]

    # Split the first matrix
    train_matrix1 = matrix1[:, train_indices]
    test_matrix1 = matrix1[:, test_indices]

    # Split the second matrix
    train_matrix2 = matrix2[:, train_indices]
    test_matrix2 = matrix2[:, test_indices]

    return train_matrix1, test_matrix1, train_matrix2, test_matrix2
```

In [9]:
```
train_x_apple, test_x_apple, train_y_apple, test_y_apple = split_train_te
train_x_tesla, test_x_tesla, train_y_tesla, test_y_tesla = split_train_te
print(train_x_apple.shape)
print(train_y_apple.shape)
```

```
torch.Size([2, 824])
torch.Size([1, 824])
```

Step04: Optimize the model using PyTorch

In [10]:
```
import torch.optim as optim

# Hyperparameters
H = 10
input_dim = 2  # Dimension of x_apple

# Initialize parameters
V = torch.randn(H, 2, requires_grad=True, dtype=torch.float32)
b = torch.randn(H, 2, requires_grad=True, dtype=torch.float32)
w = torch.randn(H, requires_grad=True, dtype=torch.float32)
w0 = torch.randn(1, requires_grad=True, dtype=torch.float32)

# Optimizer
optimizer = optim.Adam([V, b, w, w0], lr=0.01)

# Training loop
num_epochs = 50000
for epoch in range(num_epochs):
    optimizer.zero_grad()

    # Forward propagation
    y_pred = forward_propagation(V, b, w, w0, train_x_apple)

    # Compute loss
    loss = mean_squared_error(y_pred, train_y_apple)

    # Backward propagation and optimization
    loss.backward()
    optimizer.step()

    # Print loss every 1000 iterations
    if (epoch + 1) % 1000 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}
```

```
print("Optimization finished")
optimal_V = V
optimal_b = b
optimal_w = w
optimal_w0 = w0
```

```
Epoch [1000/50000], Loss: 82.5075
Epoch [2000/50000], Loss: 62.7903
Epoch [3000/50000], Loss: 53.3472
Epoch [4000/50000], Loss: 48.9517
Epoch [5000/50000], Loss: 47.7352
Epoch [6000/50000], Loss: 47.1710
Epoch [7000/50000], Loss: 46.8830
Epoch [8000/50000], Loss: 46.7401
Epoch [9000/50000], Loss: 46.6483
Epoch [10000/50000], Loss: 46.6008
Epoch [11000/50000], Loss: 46.5722
Epoch [12000/50000], Loss: 46.5536
Epoch [13000/50000], Loss: 46.5405
Epoch [14000/50000], Loss: 46.5307
Epoch [15000/50000], Loss: 46.5254
Epoch [16000/50000], Loss: 46.5170
Epoch [17000/50000], Loss: 46.5120
Epoch [18000/50000], Loss: 46.5070
Epoch [19000/50000], Loss: 46.5010
Epoch [20000/50000], Loss: 46.4956
Epoch [21000/50000], Loss: 46.4919
Epoch [22000/50000], Loss: 46.5027
Epoch [23000/50000], Loss: 46.4912
Epoch [24000/50000], Loss: 46.4863
Epoch [25000/50000], Loss: 46.4854
Epoch [26000/50000], Loss: 46.4847
Epoch [27000/50000], Loss: 46.4841
Epoch [28000/50000], Loss: 46.4838
Epoch [29000/50000], Loss: 46.4833
Epoch [30000/50000], Loss: 46.4830
Epoch [31000/50000], Loss: 46.4828
Epoch [32000/50000], Loss: 46.4828
Epoch [33000/50000], Loss: 46.4823
Epoch [34000/50000], Loss: 46.4822
Epoch [35000/50000], Loss: 46.4827
Epoch [36000/50000], Loss: 46.4819
Epoch [37000/50000], Loss: 46.4950
Epoch [38000/50000], Loss: 46.4854
Epoch [39000/50000], Loss: 46.4817
Epoch [40000/50000], Loss: 46.4816
Epoch [41000/50000], Loss: 46.4816
Epoch [42000/50000], Loss: 46.4814
Epoch [43000/50000], Loss: 46.4814
Epoch [44000/50000], Loss: 46.4814
Epoch [45000/50000], Loss: 46.4813
Epoch [46000/50000], Loss: 46.5202
Epoch [47000/50000], Loss: 46.4812
Epoch [48000/50000], Loss: 46.4889
Epoch [49000/50000], Loss: 46.4811
Epoch [50000/50000], Loss: 46.4811
Optimization finished
```

Step05: Make prediction model

Here, our method will return a model using the optimized parameters.

In [11]:
```
def get_predict_function(V, b, w, w0):
    """
    Returns a prediction function using the learned parameters.

    Parameters:
    V (torch.Tensor): Learned parameter V.
    b (torch.Tensor): Learned parameter b.
    w (torch.Tensor): Learned parameter w.
    w0 (torch.Tensor): Learned parameter w0.

    Returns:
    function: A function that takes input data x and returns predictions.
    """
    def predict(x):
        """
        Predict using the learned parameters.

        Parameters:
        x (torch.Tensor): Input data of shape (n_samples, 2).

        Returns:
        torch.Tensor: The predicted values.
        """
        return forward_propagation(V,b,w,w0,x)

    return predict
```

In [12]:
```
final_model = get_predict_function(optimal_V, optimal_b, optimal_w, optim
```

In [13]:
```
y_predict = final_model(test_x_apple)
```

In [14]:
```
print(test_y_apple)
```

```
tensor([[7.3000e-01, 6.1380e+01, 4.0250e+01, 6.0000e-02, 9.0500e+00, 8.930
0e+00,
         1.3546e+02, 4.5000e-01, 1.5293e+02, 6.0000e-02, 1.1490e+01, 5.090
0e+01,
         8.7400e+01, 2.0000e-02, 4.7750e+01, 6.1000e-01, 1.2600e+01, 4.500
0e-01,
         1.0200e+01, 5.1800e+01, 1.4185e+02, 1.3043e+02, 1.2982e+02, 1.140
0e+00,
         6.3000e+01, 1.6700e+02, 1.2760e+02, 6.4000e-01, 6.5570e+01, 1.412
5e+02,
         7.3250e+01, 9.6350e+01, 5.3730e+01, 9.8700e+01, 1.6070e+02, 1.490
0e+01,
         1.3450e+01, 1.1005e+02, 7.7300e+01, 3.0100e+01, 5.7050e+01, 1.104
1e+02,
         2.4800e+01, 1.3300e+02, 1.1200e+02, 4.9500e+00, 4.1000e-01, 7.555
0e+01,
         9.6900e+01, 9.4000e-01, 1.1795e+02, 1.8605e+02, 1.8182e+02, 8.665
0e+01,
         1.6623e+02, 1.9340e+01, 1.4450e+02, 2.3300e+00, 1.1700e+00, 2.641
0e+01,
         5.4400e+01, 1.6465e+02, 8.5090e+01, 5.7110e+01, 1.3500e+00, 3.382
0e+01,
         8.1200e+01, 4.6000e+01, 2.5620e+01, 8.3900e+00, 6.8340e+01, 7.350
0e+01,
         1.4600e+00, 1.1050e+01, 7.9200e+00, 1.0570e+02, 2.0000e-02, 4.550
0e+00,
         8.2000e-01, 9.7000e+01, 1.4070e+01, 9.3450e+01, 1.2247e+02, 1.061
0e+02,
         3.1640e+01, 4.6960e+01, 2.0700e+01, 6.4500e+00, 6.1550e+01, 9.100
0e-01,
         4.3550e+01, 4.8520e+01]])
```

In [15]:
```
print(mean_squared_error(y_predict, test_y_apple))
```

```
tensor(56.0273, grad_fn=<MeanBackward0>)
```

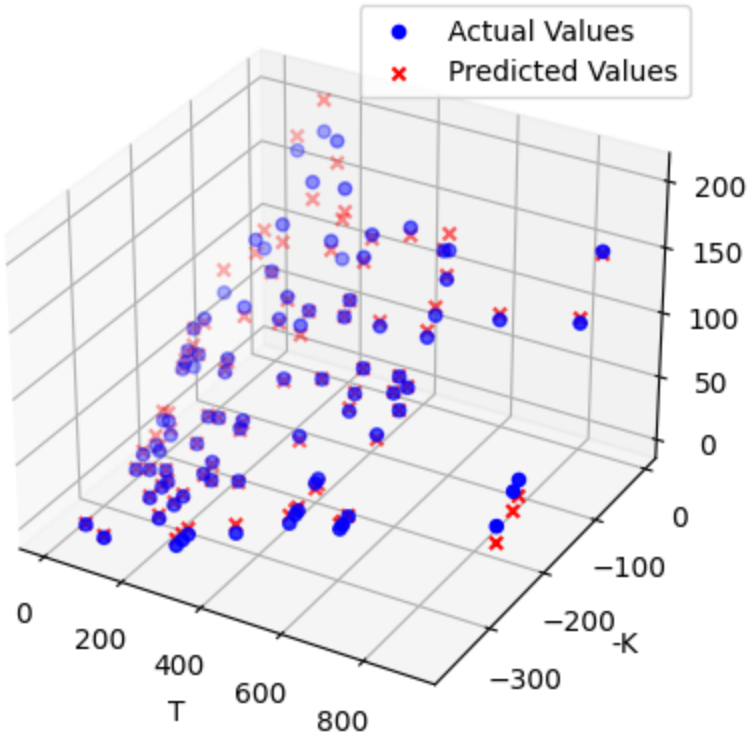
Extra Step: Analysis of the model

First, we will plot an image to roughly observe how the model fits on the test data set.

In [16]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
```

In [17]:
```
test_x_apple_np = test_x_apple.detach().numpy()
test_y_apple_np = test_y_apple.detach().numpy()
y_predict_np = y_predict.detach().numpy()

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')


ax.scatter(test_x_apple_np[1], test_x_apple_np[0], test_y_apple_np, c='bl


ax.scatter(test_x_apple_np[1], test_x_apple_np[0], y_predict_np, c='red',
```

```
ax.set_xlabel('T')
ax.set_ylabel('-K')
ax.set_zlabel('c')


ax.legend()


plt.show()
```



We can observe that the predicted values of the data points are quite close to the actual values.

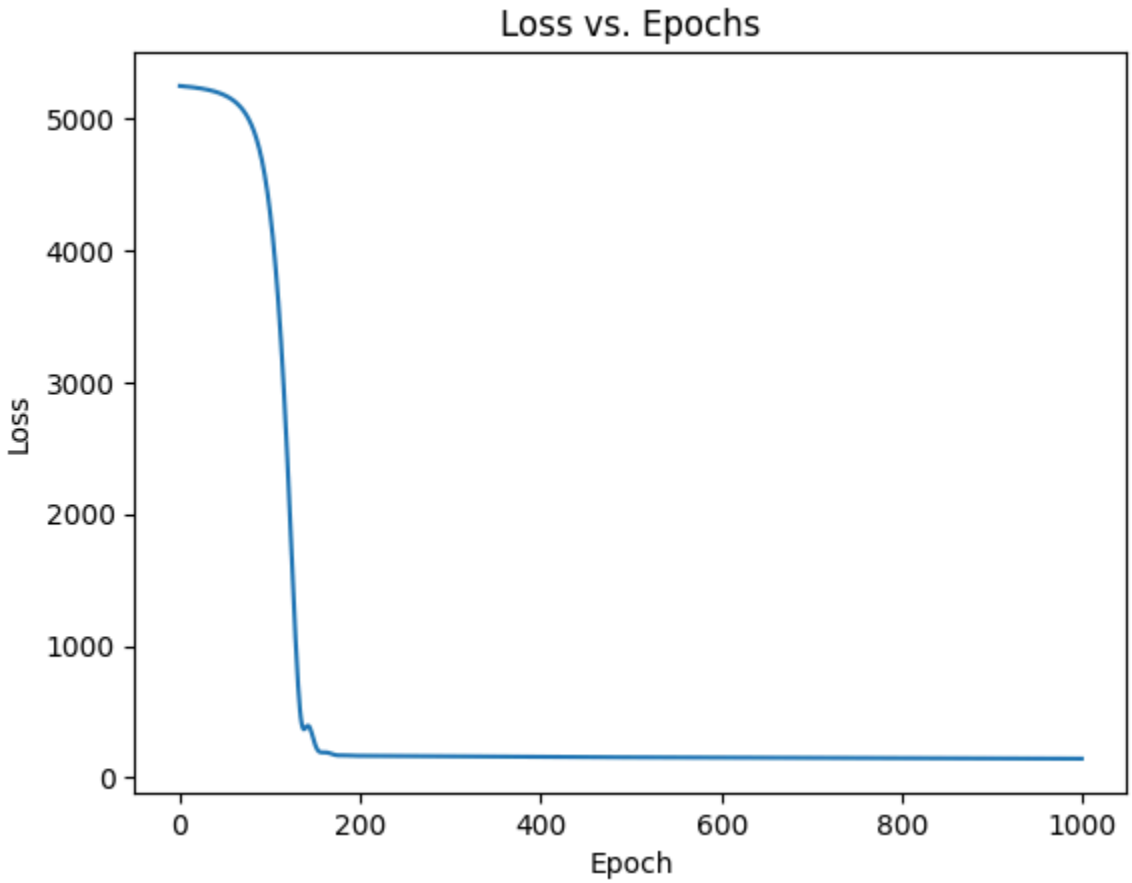
Next, let's analyze the differences in the loss function exhibited by the model for different numbers of neurons.

First, we noticed an issue: during the model training process, we found that the loss function converges to a value that is not zero. We believe this is because this value represents the best result the model can achieve given the current structure. That is, it is limited by the model structure (in this case, the number of neurons).

In [18]:
```
import torch
import torch.optim as optim
import matplotlib.pyplot as plt

# Hyperparameters
H = 10
input_dim = 2  # Input dimension

# Initialize parameters
V = torch.randn(H, 2, requires_grad=True, dtype=torch.float32)
b = torch.randn(H, 2, requires_grad=True, dtype=torch.float32)
w = torch.randn(H, requires_grad=True, dtype=torch.float32)
w0 = torch.randn(1, requires_grad=True, dtype=torch.float32)
```

```
# Optimizer
optimizer = optim.Adam([V, b, w, w0], lr=0.01)

# Training loop
num_epochs = 1000
losses = []

for epoch in range(num_epochs):
    optimizer.zero_grad()

    # Forward propagation
    y_pred = forward_propagation(V, b, w, w0, train_x_apple)

    # Compute loss
    loss = mean_squared_error(y_pred, train_y_apple)

    # Backward propagation and optimization
    loss.backward()
    optimizer.step()

    # Save loss value
    losses.append(loss.item())

    # Print loss every 1000 iterations
    if (epoch + 1) % 1000 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}

print("Optimization finished")

# Plot loss vs. epochs
plt.plot(range(num_epochs), losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs. Epochs')
plt.show()
```

```
Epoch [1000/1000], Loss: 142.6894
Optimization finished
```



Finally, the table below shows the relationship between the number of neurons and the convergence results of the loss function.

H	Loss
1	64.7333
2	62.379
3	45.1903
4	45.0423
5	44.711
10	44.7529
100	44.8589
1000	41.4791

Extra Step2: Mapping relationship and comparation with Black-Sholes Formula

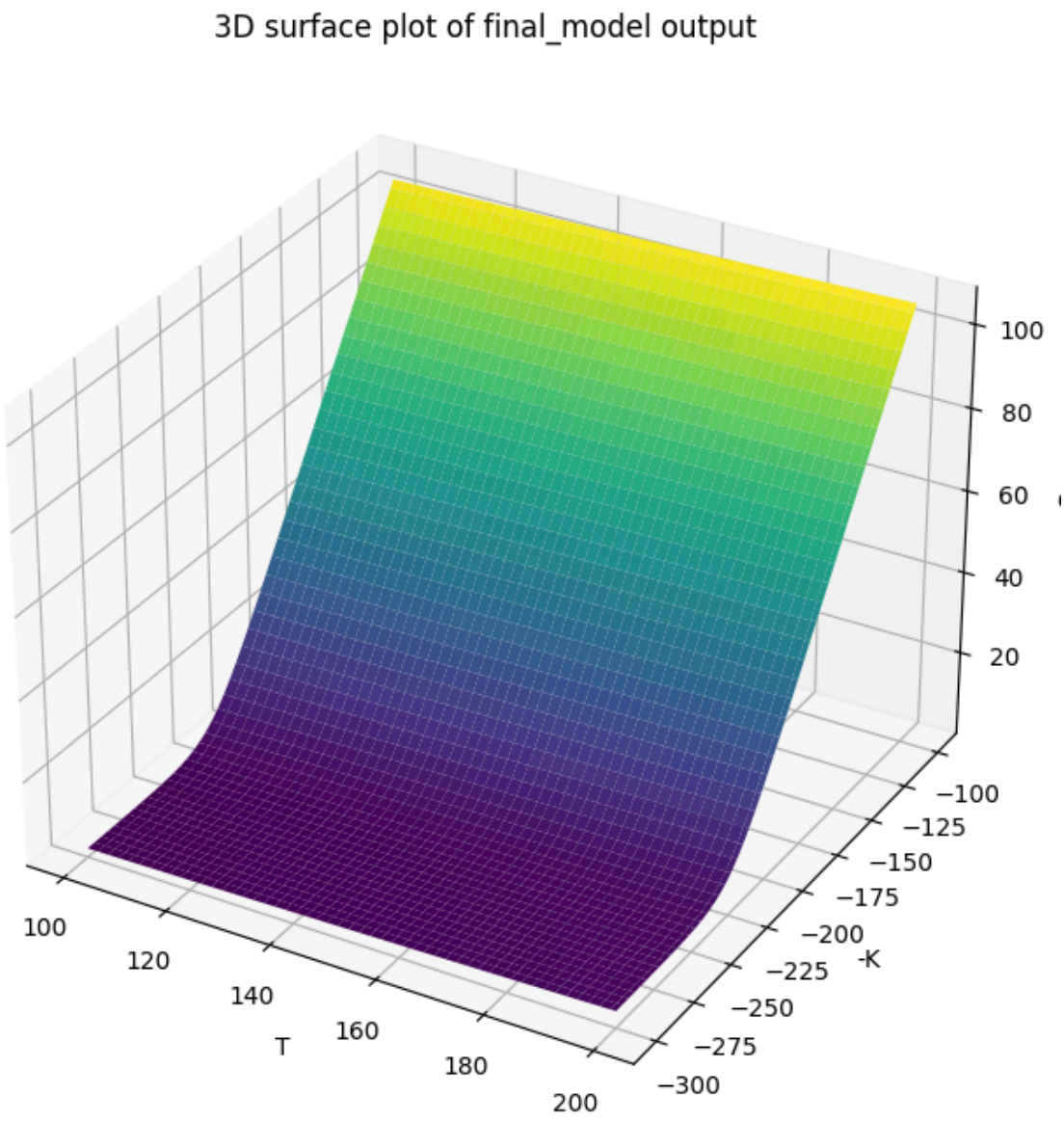
We know that what we want to do, in simple, is just to find a mapping relationship:

$\hat{c} = f(K, T)$

and we need f to satisfy some condition to make sure "Non-arbitrage"

So this plot shows this kind of relationship:

In [19]:
```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from mpl_toolkits.mplot3d import Axes3D

# Define the range for K and T
K_range = np.linspace(-300, -100, 100)
T_range = np.linspace(100, 200, 100)

# Create a tensor containing all combinations of K and T
K, T = np.meshgrid(K_range, T_range)
tensor = np.column_stack((K.ravel(), T.ravel())).T

# Convert NumPy array to PyTorch tensor
tensor = torch.tensor(tensor, dtype=torch.float32)

# Use final_model to compute the results
c = final_model(tensor)

# Convert the result to a NumPy array and reshape
C_model = c.detach().numpy().reshape(K.shape)

# Plot the 3D surface
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(T, K, C_model, cmap='viridis')

ax.set_title('3D surface plot of final_model output')
ax.set_xlabel('T')
ax.set_ylabel('-K')
ax.set_zlabel('C')

plt.show()
```



Then we compare it with Black-Sholes formula:

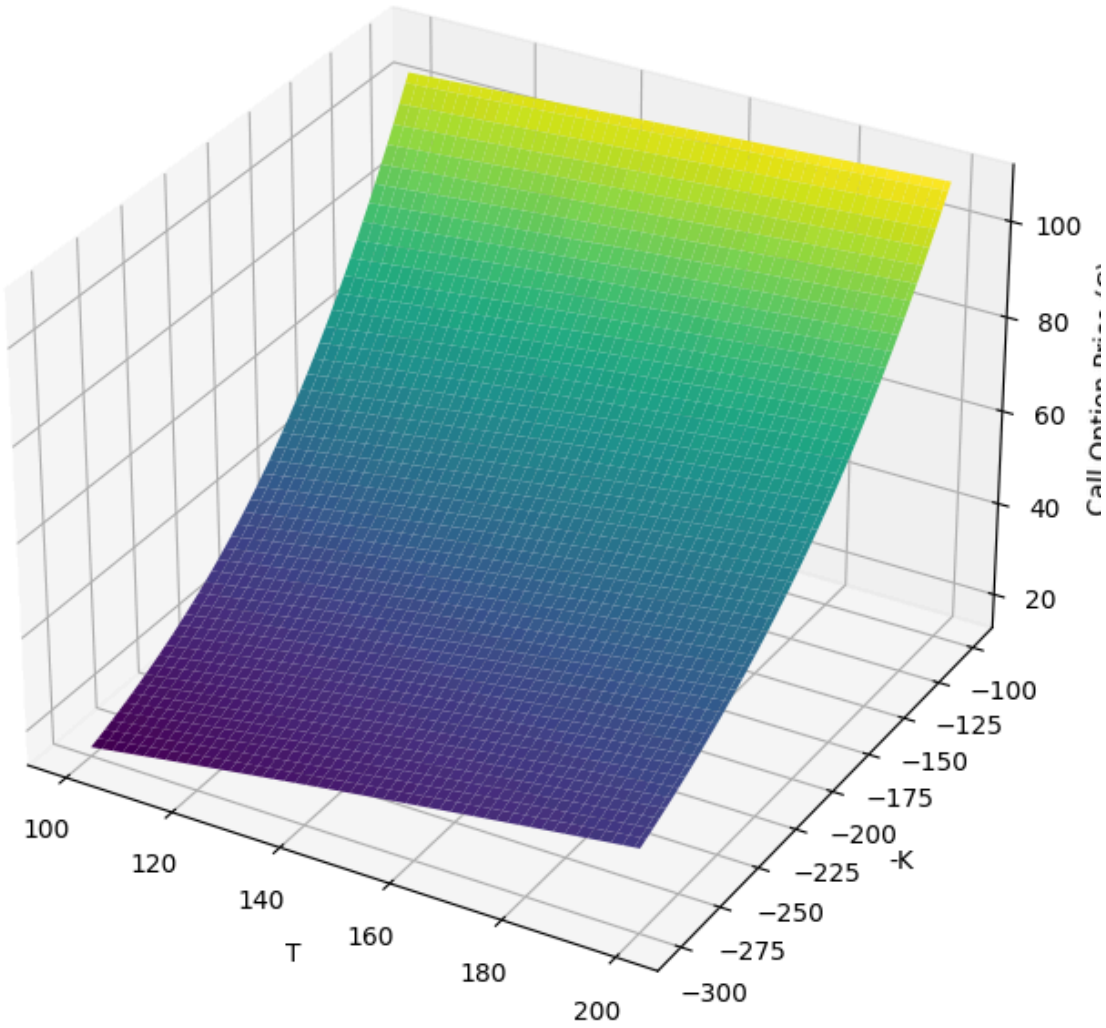
Here's the Black-Scholes Fomula:

$$C = S_0 N(d_1) - X e^{-rT} N(d_2)$$

where:
$$d_1 = \frac{\ln(S_0/X) + \left(r + \sigma^2/2\right) T}{\sigma\sqrt{T}}$$
$$d_2 = d_1 - \sigma\sqrt{T}$$

In [20]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from mpl_toolkits.mplot3d import Axes3D

# Black-Scholes formula
def black_scholes(S, K, T, r, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(
    d2 = d1 - sigma * np.sqrt(T)
    call_price = S * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    return call_price
```

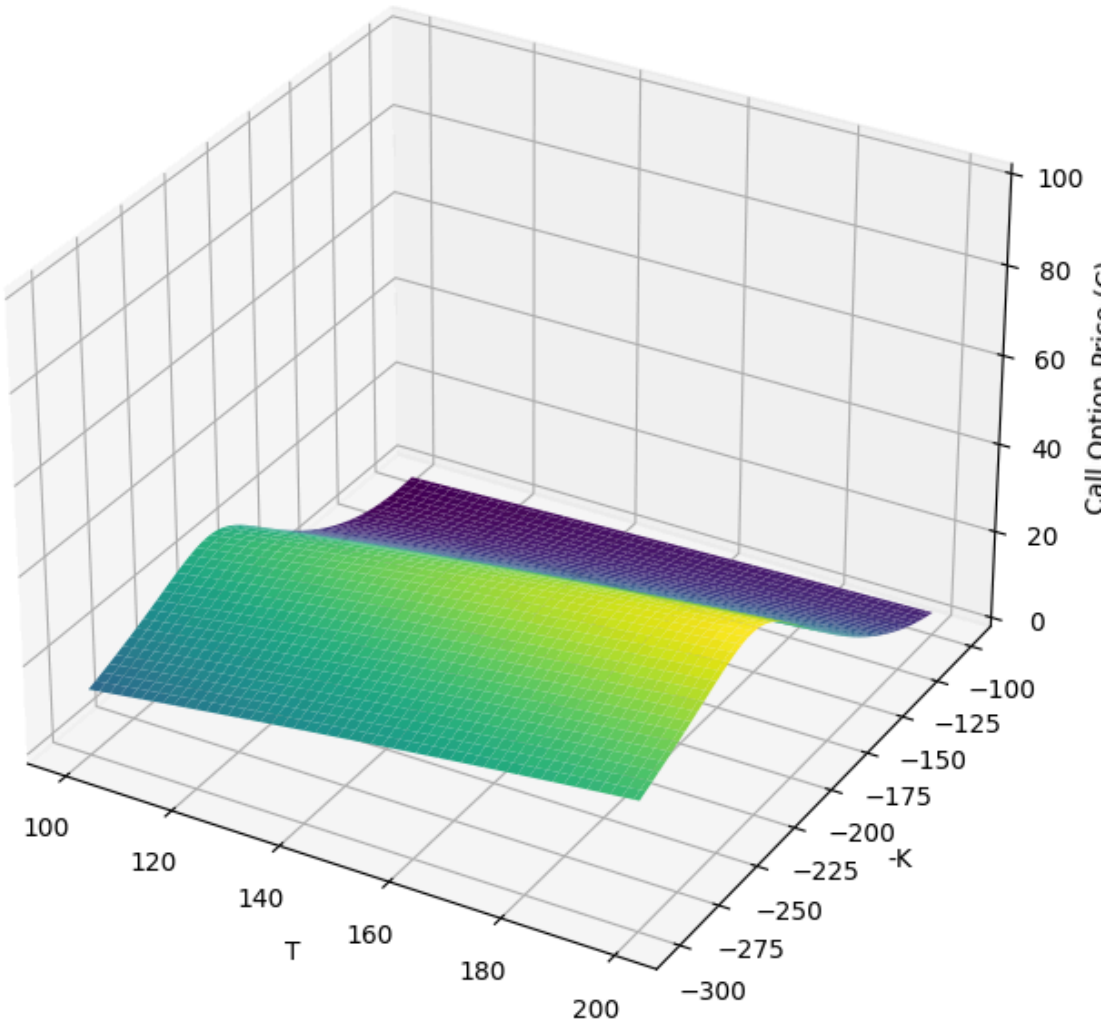
```
# Parameter settings
S = 200  # Current price of the underlying asset
r = 0.0001  # Risk-free interest rate
sigma = 0.05  # Volatility

# Calculate option price
C_bs = black_scholes(S, -K, T, r, sigma)

# Plot the 3D graph
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(T, K, C_bs, cmap='viridis')

# Set axis labels
ax.set_xlabel('T')
ax.set_ylabel('-K')
ax.set_zlabel('Call Option Price (C)')

plt.show()
```



Then we plot the difference between our model and Black-Sholes formula:

In [21]:
```
fig = plt.figure(figsize=(12, 8))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(T, K, C_bs- C_model, cmap='viridis')
```

```
ax.set_xlabel('T')
ax.set_ylabel('-K')
ax.set_zlabel('Call Option Price (C)')
ax.set_zlim(0, 100)

plt.show()
```



Thank you very much.