University of California Santa Barbara

Department of Statistics

PSTAT 199(Independent Studies)

# DEEP LEARNING FOR EUROPEAN CALL OPTIONS

Thiha Aung, Mentor: Professor Tomoyuki Ichiba

December 21, 2020

# Contents

# 1 Introduction

## 1.1 Abstract

Current advancements in computational power, mathematical analysis and computer hardware have made it possible to explore computationally expensive models in Mathematical Finance. We explored how deep learning can be used for Option Pricing, a subset of Mathematical Finance in a general way. Revisiting the revolutionary work for option pricing by Black and Scholes from the 1990s (1973) , we explored a deep learning framework which can learn to price European options more accurately than Black-Scholes formula. We train a fully-connected Multi-Layer Perceptron(MLP) neural network to reproduce the Black and Scholes (1973) option pricing formula to a higher accuracy. We also offer a brief introduction to neural networks and details on Hyper-parameter Optimization that make the model as accurate as possible while avoiding from over-fitting. Moreover, we explored the literature review on Neural Networks for Option Pricing by Johannes Ruf to explore modifications needed for architecture and input features for better predictive behavior. [3]

## 1.2 Goals & Questions

For our model, we simulated the input parameters of Black Scholes formula. Then, we computed respective parameters to compute option prices using Black-Scholes formula. Using this data set, we will train our Multi-Layer Perceptron(MLP) network. Moreover, these are the questions we will try to answer as part of the project.

- How accurate can a simple neural network learn Black-Scholes pricing?
- To explore error distribution of in the money and out of the money options.
- How can we use more advanced neural networks to improve our results?

# 2 Methods

The Black Scholes model is used to price put and call options by estimating the variation over time. The model is based on the assumption that the markets are highly efficient (i.e., Efficient Market Hypothesis), which suggests that stock prices are uncorrelated to one another across time. As a result, Geometric Brownian Motion (GBM) also has been assumed. However, the assumption is often violated in practice, leading to numerous variations of the Black-Scholes model.

The Black-Scholes formula for European call and put options are:

$$C(S_0, t) = S_0 e^{-q(T-t)} N(d_1) - K e^{-r(T-t)} N(d_2)$$
$$P(S_0, t) = K e^{-r(T-t)} N(-d_2) - S_0 e^{-q(T-t)} N(-d_1)$$

where $S_0$: Stock Price, $C(S_0, t)$: Price of the Call Option, $K$: Exercise Price, $(T-t)$: Time to Maturity, where T is Exercise Date ,$\sigma$: Underlying Volatility (a standard deviation of log returns) (30, 60, 252 days returns,etc), $r$: Risk-free Interest Rate (i.e., T-bill Rate).

The $d_i$ variables are defined as:

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

$$d_2 = d_1 - \sigma\sqrt{T-t} = \frac{\ln \frac{S_0}{K} + (r - q - \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$

Finally, $N(x)$ is cumulative distribution function for the standard normal distribution.

In this project, we will be training our neural network to learn Vanilla(European) Call Options. To train a neural network to learn the call option pricing equation, Culkin and Das (2017) simulated a range of call

option prices with ranges of different parameters. We repeated the simulation using the parameter range below.

| Parameter | Range |
|---|---|
| Stock Price (S) | \$10 - \$50 |
| Strike Price ($K$) | \$7 - \$650 |
| Maturity ($T-t$) | 1 day to 3 years |
| Dividend Rate ($q$) | 0% - 3% |
| Risk Free Rate ($r$) | 1% - 3% |
| Volatility ($\sigma$) | 5% - 90% |
| Call Price ($C$) | \$0 - \$328 |

Each parameter is drawn from a uniform distribution of given end points in the table. In total, the data-set contains 300,000 observations. Using these parameter combinations, we computed Call Price using Black-Scholes' formula. The Call Price range is given in the table above. In our neural network, Call Price is the output and the remaining parameters are inputs. Moreover, we optimized our hyper-parameters via Random Search. After optimizing our model, we trained our model and discussed error distribution and sensitivity. Then, we explored how we can improve our model's performance by feature-engineering and different neural network architecture.

## 2.1 Brief Introduction to Neural Networks

MLPs were first introduced in the paper by McCulloch and Pitts (1943)[2].The application of back-propagation in the 1980s[4], and current advancements in software and hardware had made it possible for training neural networks for different applications. Multi-Layer Perceptron(MLP) neural network, also known as Artificial Neural Network has one of the simplest architectures among all other neural networks. Inspired by biological neurons, MLPs imitate neurons in the brain, each of which applies receives a number of inputs $x_i$ , applies a function $h_i(x_i)$, and produces an output o, shown in Figure 1. Each neuron applies weights to its inputs and includes an activation function. The activation function introduces non-linearities to the output, which results in non-linear regression. Mathematically, neural networks can be thought of as stacks of piece-wise linear functions connected together by non-linear activation functions.

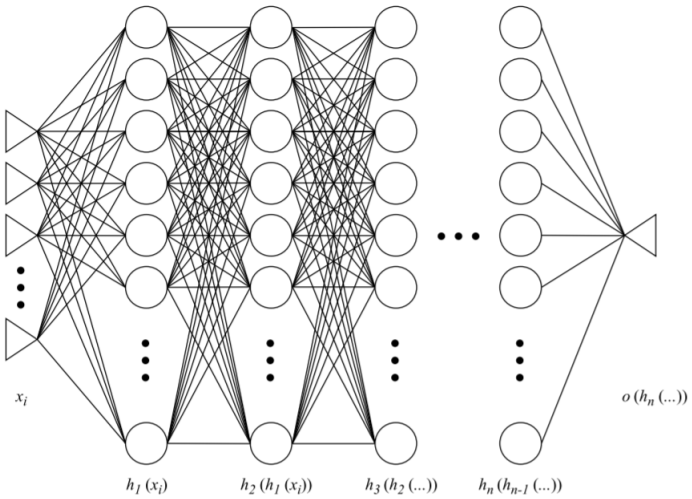


Figure 1: Architecture of Feed-Forward Neural Network

These neurons connect the input and output of the network in different ways. A multi-layer perceptron (MLP) is structured into a series of layers, each of which containing at least one neuron. The first layer is the input and the last layer is the output. Every layer between the input layer and the output layer is

defined as the hidden layer. Each neuron in the hidden layers and the output layer receives each output from the neurons of the layer preceding it as input, and passes its output to every neuron in the following layer.

Each node $h_n^{(j)}$ in layer n of the Neural Network, takes in inputs $h_{n-1}^{(i)}, i = 1, 2, ..., M_{n-1}$ from the previous layer, creating a function called net-input:

$$a_n^{(j)} = \sum_{i=1}^{M_{n-1}} w_{ij} h_{n-1}^{(i)} + b_n^{(j)} \tag{1}$$

where

$M_{n-1}$ =number of nodes at layer (n-1)
i,j = Node number at any arbitrary layer
$b_n^{(j)}$ = bias of the j-th node
$w_{ij}$ = weight at node j of n-th layer, taking input from node i at n-1 layer.
i = 1,2,...,$M_{n-1}$, j = 1,2,...,$M_n$

By using basic principles of combinatorics, between layer (n-1) and n, there are $M_n * M_{n-1} + M_{n-1}$ parameters. Moreover, the net input $a_n^{(j)}$ is passed into a non-linear function to generate a single output such that :

$$h_n^{(j)} = h(a_n^{(j)})$$

This function is known as activation function. In this paper, we use the most common activation function known as ReLU(Rectified Linear Unit).

- ReLU (rectified linear unit). The function is h(x) = max(x, 0). When x is negative, the gradient of the function is zero, and hence, the neuron will not be active in such case.

Initial weights for each neuron are randomly generated. Each weight is updated to lower the error. The user defines what is the error(objective) function. The goal is to find the global minimum of the error(objective) function. Mean Squared Error(MSE) is the most common error function. To minimize MSE, we compute the gradient of MSE at each neuron with respect to its weights, and then moving its weights in the opposite direction, by a factor known as the learning rate. This is an optimization problem. The most common known algorithm is Gradient Descent. These gradients are computed efficiently by the back propagation algorithm.

Since our neural network is a large nested function, we can utilize the back propagation algorithm effectively. The gradient at each neuron can be calculated using the chain rule. After the gradient of each parameter is calculated, the parameters are moved towards the direction that lowers the loss function. In order to prevent divergence, we move them by a small factor, known as the learning rate. Each time all parameters are updated using gradients, we say "one epoch" is completed.

Training to only reduce the loss function will result in over-fitting. To prevent over-fitting, we will (i) choose the number of epochs, (ii) add dropout rate(for example, dropout rate of 0.20 at layer j will only use 80% of nodes at the j-th layer), (iii) use batch-gradient, which is a method of using only a subset of training data to compute the gradient and update the parameters (iv) choose the number of layers, (v) choose the number of nodes, (vi) choose the learning rate and optimization algorithm to minimize loss(objective) function. These settings are known as hyper-parameters. Batch-gradient requires the use of batch size. For instance, for training the neural network with 10000 data points, let us use a batch size of 500. Using 500 data points, the model will compute the gradient with respective learning rate, optimization algorithm and update the parameters once. After 20 iterations, we will have used all 10000 data points and updated the parameters 20 times. Then, we say 1 epoch completed. Hence, for training data with 10000 data points, using batch size 500 and epoch 10 will update the parameters of the model 200 times (20 updates per epoch * 10 epoches =200 updates).

## 2.2 Multi-Layer Perceptron Neural Network for European Call Options

We used 70% of approximately 300,000 observations as training data and the rest as training data. Our inputs are Stock Price, Strike Price, Maturity, Dividend Rate , Risk-free Interest Rate, Volatility and our

output is Call Price. Moreover, we added moneyness as one of the inputs. Initial model fitting suggests us that adding an indicator for moneyness as input gives a better error distribution. So, we encoded In The Money(ITM) options with indicator value 1 and Out of The Mone(OTM) options as 0. Using these 8 inputs and 1 output, we built our model.

### 2.2.1 Hyper-parameter Optimization

It is obvious that the user cannot define arbitrary combination of hyper-parameters. So, in order to build our model, we used Random-Search Cross Validation. Grid-Search is a more popular method of Cross-Validation for finding hyper-parameters. Grid-Search finds the best hyper-parameter combination by searching through all possible combinations of hyper-parameters through the whole space, then gives the combination that minimizes the loss function,in this case MSE. Random-Search does not search through the whole parameter space. Depending on the number of iterations user define, random search finds the best parameter combination randomly through a subset of the whole space. This is useful when we have large data-set and limited training time. Moreover, it has been shown that random-search is as effective as grid-search even though it only searches through a subset of the parameter space [1].

### 2.2.2 Architecture of MLP

Random-Search cross validation through 30% of the whole parameter space gives us the following combination of hyper-parameters.

| Parameter | Range | Best Value |
|---|---|---|
| Number of layers | 1 - 8 | 6 |
| Number of Nodes(Neurons) | 25,50,..., 600 | 425 |
| Optimizer | SGD, Adam | Adam |
| Learning rate | 3e-2,3e-3,...,3e-6 | 3e-5 |
| Dropout rate | 0.0,0.1,...,0.4 | 0.0 |

Hence, we defined our model using the best values of hyper-parameter above. Moreover, in the model, we used ReLU activation function between each layer, MSE loss function and normalized after the input layer. While training the data, we split the training data into 75% actual training and 25% validation data to plot the MSE error of both training and validation after each epoch. Finally, we used batch size of 250 and 50 epoches.

# 3 Analysis

After training the model, we observed the the graph of training and validation error vs epoch, R squared value and MSE on test data, error distribution and error sensitivity.

## 3.1 Model Analysis

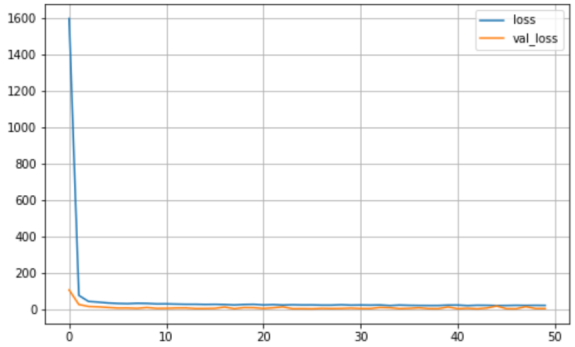


Figure 2: Training and Validation error vs epoch

As seen in figure above, our model error starts to converge after 6 or 7 Epoches. Moreover, we observe validation is not fluctuating randomly at each epoch. So, our model is not over-fitting. Using the test data, we analyzed the R-squared value. Since our R-squared value is 0.9989, we obtained an excellent fit. Moreover, our Mean Squared Error(MSE) is 3.261, given that true European call prices range from 0$ to 328$. Hence, our MSE is relatively low. However, we will analyze the residuals(difference) in the next sections to understand the error distribution better.

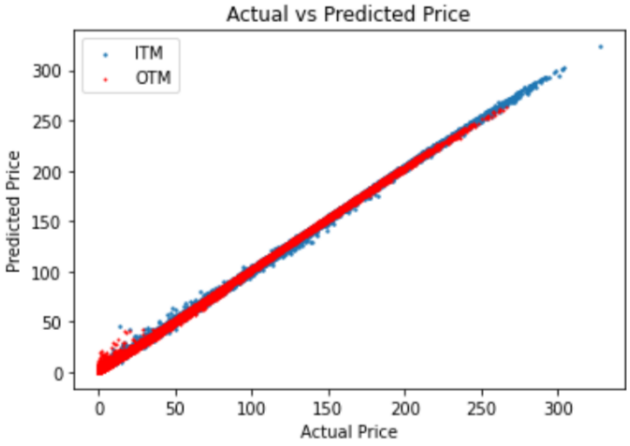


Figure 3: Actual vs Predicted Price on test data

In the figure above, we plotted the actual vs predicted price, grouped by moneyness. First, we observe that as the price increases, the plot becomes less scattered. Hence, we have slight problem with predicting options with low call price. Moreover, we observe that we predicted Out of The Money(OTM) options better than In The Money(ITM) options. In the next section, we will explore how the price of call option and how much moneyness affect prediction error.

## 3.2 Error Distribution by Moneyness

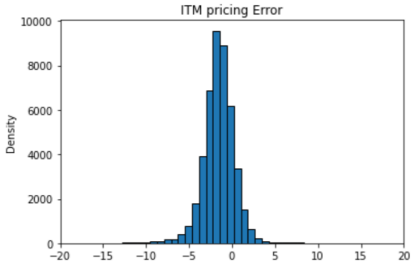


Figure 4: In the money(test data) pricing difference.

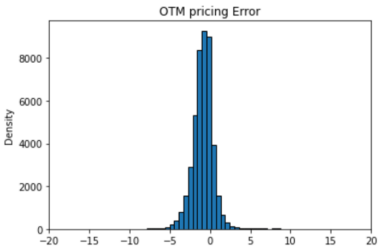


Figure 5: Out of the money(test data) pricing difference.

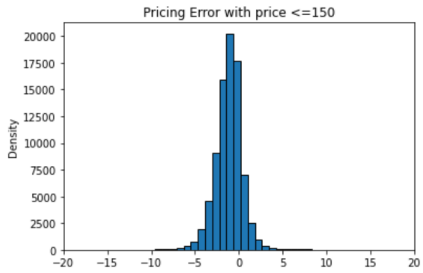


Figure 6: Pricing difference on test data with price $\leq 150$

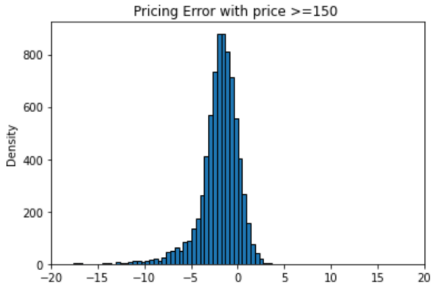


Figure 7: Pricing difference on test data with price $\geq 150$

First, we computed the difference by subtracting actual price from predicted price. For both ITM and OTM options of test data, we observe that the model is mostly under-pricing. However, the standard deviation of ITM options is greater than the standard deviation of OTM options. Then, again on test data, we splitted the data by Call price. We analyzed the pricing error when the price is less than 150 dollars or otherwise. Again, in both distribution, we observe our model is mostly under-pricing. As expected, the distribution of pricing difference is smaller for options with prices lower than 150 dollars. And we know that Black-Scholes equation under-prices deep ITM and OTM options. For our model, in general, it under-prices the options.

## 3.3 Error sensitivity Analysis

OLS Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | Difference | **R-squared:** | 0.550 |
| **Model:** | OLS | **Adj. R-squared:** | 0.550 |
| **Method:** | Least Squares | **F-statistic:** | 1.377e+04 |
| **Date:** | Mon, 21 Dec 2020 | **Prob (F-statistic):** | 0.00 |
| **Time:** | 03:01:50 | **Log-Likelihood:** | -1.6591e+05 |
| **No. Observations:** | 90000 | **AIC:** | 3.318e+05 |
| **Df Residuals:** | 89991 | **BIC:** | 3.319e+05 |
| **Df Model:** | 8 | | |
| **Covariance Type:** | nonrobust | | |

| | coef | std err | t | P>\|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **const** | -6.5492 | 0.068 | -96.483 | 0.000 | -6.682 | -6.416 |
| **Stock Price** | 0.0003 | 0.000 | 1.466 | 0.143 | -0.000 | 0.001 |
| **Strike Price** | 0.0069 | 0.000 | 34.607 | 0.000 | 0.007 | 0.007 |
| **Maturity** | 0.7925 | 0.006 | 134.236 | 0.000 | 0.781 | 0.804 |
| **Dividends** | -9.4117 | 0.589 | -15.981 | 0.000 | -10.566 | -8.257 |
| **Risk-free** | 5.6636 | 0.196 | 28.854 | 0.000 | 5.279 | 6.048 |
| **Volatility** | 3.3251 | 0.021 | 160.441 | 0.000 | 3.284 | 3.366 |
| **moneyness** | 4.9658 | 0.067 | 73.702 | 0.000 | 4.834 | 5.098 |
| **Indicator** | 0.3767 | 0.019 | 19.480 | 0.000 | 0.339 | 0.001 |

Figure 8: Regression analysis on Error(Difference)

The distribution tells us that the model is under-pricing in general and the standard deviation of ITM options are greater than OTM options. So, in order to understand the distribution better, we performed regression analysis. From the p-values, we observed that only three variables:the indicator for moneyness, Stock Price and Strike Price, are affecting the difference between predicted Price and actual price. This is similar to phenomenon of Implied Volatility(IV) smile that arises from Black-Scholes formula's mis-pricing deep ITM

and OTM options. Note that our Neural Network doesn't have any definied equation like Black-Scholes. In fact, our model is just trying to find the best non-linear fit that predicts the price most accurately. Moreover, even though our model only uses same input as Black-Scholes model, we have an unexpected behavior of similar pattern.

# 4 Conclusion

We went over a brief overview of the ideas in deep learning, and then proceeded to show how it may be applied in the field of Mathematical Finance, in particular, Option Pricing. Moreover, we observed that how a simple neural network such as MLP can be used to predict prices that are very close to true prices. We observed that our model is mostly under-pricing. Moreover,we learned that the pricing error may come from Indicator function of Moneyness, Stock Price and Strike Price by regression analysis. This indicates the possibility of missing information from moneyness. One possible explanation is to explore Implied Volatility. However, since this is a simulated data-set, it is not possible to compute the implied volatility. We will continue to explore how this can be applied to real world European Call options and learn how the implied volatility affects prediction. Moreover, we plan to explore the applications of deep learning towards American Options. This is a more complex problem since American Options can be executed before expiration. One possible approach is deep reinforcement learning.

# 5 Source Code

For details reference to the analysis, please visit the GitHub repository "here".

# 6 Acknowledgements

Thank you to Professor Tomoyuki Ichiba for mentorship throughout this project.

## References


[1] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: A big comparison for nas, 2019.

[2] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, December 1943.

[3] Johannes Ruf and Weiguan Wang. Neural networks for option pricing and hedging: a literature review, 2020.

[4] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.