Computations in Option Pricing Engines

Vital Mendonca Filho, Pavee Phongsopa, Nicholas Wotton
Advisors: Yanhua Li, Qinshuo Song, Gu Wang

February 16th, 2020

WPI

Submited to
Worcester Polytechnic Institte
in fulfillment of the requirements for the
Degree of Bachelor of Science in Mathematical Sciences

Disclaimer

Abstract

As computers become more advanced and grow in their computational powers, machine learning techniques start to play more important roles in various industries. In our project, we look at how we can apply this method of data analysis and pattern identifying to complement extant financial models, specifically with regards to different option pricing methods. We first prove the model under discussion is arbitrage free to confirm that they will yield appropriate results. Next, we apply a neural network algorithm and study its ability to approximate the option prices from exting models. The results of our study show promising potential of applying machine learning to situations where traditional methods do not apply. As an example, we study the implied volatility surface of highly liquid stocks using real data which is computationally intensive, justifying the practical impact of the methods proposed in this project.

Contents

List of Figures

1 Introduction

In 1973, Black, Scholes and Merton derived a formula to calculate the theoretical value of a European option contract [BS73]. Since then, the underlying Black-Scholes model has been adapted to accommodate early exercises (American options), default risk, and some other exotic option forms seen in both exchanges and over the counter (OTC) contracts which usually have no closed form solutions and are computationally intensive [FPS03]. Nowadays, advancements in stochastic numerical methods further drive research on new and more efficient ways of performing these calculations. In this paper, we explore the possibility of applying machine learning to computations of option prices. We explore the computational efficiency and accuracy of deep and reinforcement learning algorithms comparing to traditional methods and analyzing its applicability and limitations.

First, we revisit the classical method of computation to set a benchmark for our comparison with machine learning. After proving the no-arbitrage condition in the discrete-time Binomial Tree (CRR) Model, we present the computation of European and American options in CRR model, and also the closed form formulas for them in the continuous-time Black-Scholes Model.

We demonstrate the machine learning methods, by approximating simple functions using neural networks. By comparing the difference between the value of the approximation and target function at the data points, we can calculate the loss function and determine how well our machine is "learning" the target function. Since neural networks come in varying sizes, we must also determine the optimal complexity of our system, for each approximation task, so it finds a balance between the accuracy of the approximation and the computational cost. We then apply this machine learning technique to approximate option prices derived in Section 2, for both CRR and Black-Scholes models, which shows promising results.

Finally, we examine the classical numerical methods of implied volatility as an example in option pricing, where no closed-form solutions are available, show how the implied volatility surface of one underlying stock can change due to different market conditions. Based on calculation with real market data, we show that calculation of these time varying quantities using traditional methods is very demanding in comutational resources, which demonstrates an area of promising application of machine learning techniques.

2 Traditional Methods in Option Pricing

In this section, we revisit the traditional option pricing models in both discrete time and continuous time settings. For the discrete time binomial tree model, we show the no arbitrage condition and the computational procedures for both European and American options. For the continuous time Black Scholes model, we give the Black Scholes formula for European options.

2.1 A Brief Overview of Options

In finance, a derivative is a contract whose value is reliant upon an underlying asset. Options are one type of derivative involving a pre-agreed upon price, known as the strike price, and a specific expiry date beyond which the option has no value and can no longer be exercised. While there are many types of options, the two most basic types are Calls and Puts. A Call option gives the owner the right, but not the obligation, to *buy* the underlying asset at the strike price K, while a Put option provides the right, but not the obligation to *sell* the underlying at the strike price. In terms of time to exercise, these two options can be further divided into two subgroups: American and European types. European options can only be exercised at their expiry T, while American options can be exercised at any time before, or on, the date of expiry. [Shr05]

Mathemtacially, we have the Payoff out of a Call option at time t is:

$$P_{Call}(T) = max\{0, S_T - K\},$$

because the option is in-the-money only if the underlying price (S_T) is greater than the previous stipulated strike. If the stock price(S_T) is less than the strike it makes no sense to pay the strike value for the option as it can be bought cheaper on the market, and the payoff is zero. Similarly, the payoff for a Put option is

$$P_{Put}(T) = max\{0, K - S_T\},$$

because the option is in-the-money only if the underlying price (S_T) is less than the previous stipulated strike (K). If the stock price is greater than the strike it makes no sense to sell the stock for the lower strike price as there is a better deal on the market and thus the payoff is zero. Graphically, Figure 1 shows the payoffs for Call and Put at expiry time T respectively.

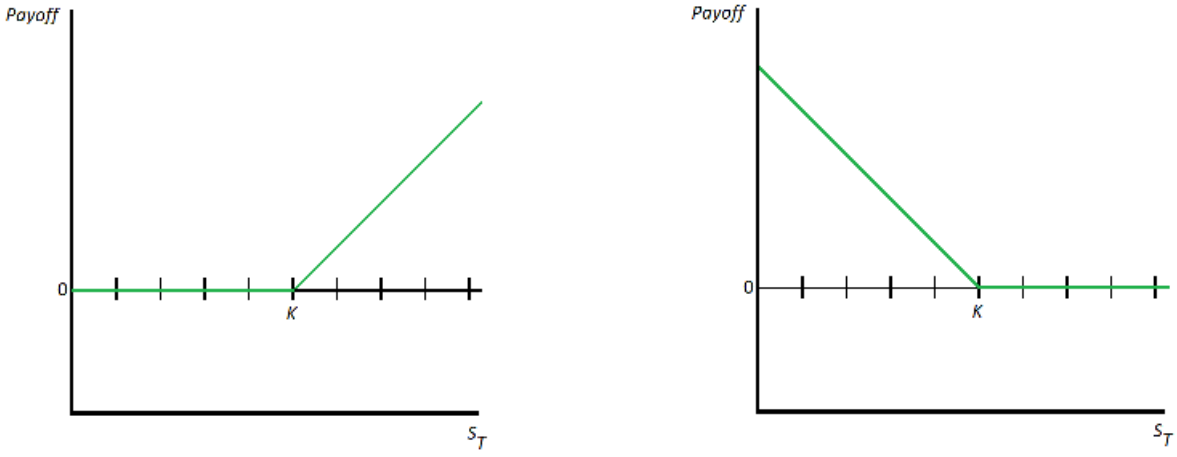
Note that it is possible to sell, or short, an option. In that case, the payoff diagram is just the same as for holding the option, but reflected on the S_t axis. Mathematically, the payoffs become $min\{0, K - S_T\}$, for the Call Option, and $min\{0, S_T - K\}$, for the Put Option.

2.2 Discrete Time Model

In this section, we discuss the no arbitrage conditions for the discrete time binomial tree model, which guarentees that all our results financially make sense.

Definition 2.1. *An n-step binomial tree model $B(n, p, S, u, d)$ describes a financial market which includes two assets; one is risk-free and earns a constant rate of return equal to r. The price of the risky asset is given by (as demonstrated in[1] Figure 2):*

[1]Retrieved from Wikipedia `https://upload.wikimedia.org/wikipedia/commons/2/2e/Arbre_Binomial_Options_Reelles.png`.



(a) European Call (b) European Put

Figure 1: Payoffs of European Call and Put Options

- *At $t = 0$, the price of the underlying asset is $S_0 = S$;*
- *$\{X_t : 1 \leq t \leq n\}$ are iid Bernoulli(p) random variables;*
- *For $1 \leq t \leq n$, the price of the underlying asset is*

$$S_t = S_{t-1}(\mathbf{1}_{\{X_t=1\}}u + \mathbf{1}_{\{X_t=0\}}d),$$

where $u < d$, and $\mathbf{1}_{\{X_t=i\}}$ is the indicator function taking value 1 when the condition specified is met and value 0 otherwise.

Definition 2.2. *An* **arbitrage opportunity** *exists when an investor can construct a trading strategy which starts with zero initial capital and has zero probability of losing money while maintaining a positive probability of strictly positive gain. [Rup06]*

Definition 2.3. *An asset pricing model is called* **arbitrage free** *if there exists no arbitrage opportunities within the model.*

In the next theorem we establish the necessary and sufficient conditions for the nonexistence of arbitrage in $B(n, p, S, u, d)$.

Theorem 2.1. *The model $B(n, p, S, u, d)$ is arbitrage free if and only if $d < (1 + r) < u$.*

Proof. ($\Rightarrow$) If the model is arbitrage free then $d < (1 + r) < u$:

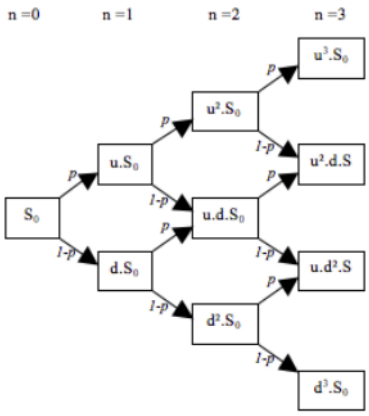


Figure 2: Binomial Price Tree

Suppose the given asset model is arbitrage free. This means a risk-neutral probability measure exists which assigns to the event $\{X_i = 1\}$ the probability [Rup06]

$$q = \frac{(1+r) - d}{u - d}$$

.

Since this is a probability we know $0 \leq q \leq 1$ which implies

$$0 \leq \frac{(1+r) - d}{u - d} \leq 1$$

However, note that if $q = 0$, $d = 1 + r$. In this case, we can construct an arbitrage opportunity as follows: borrow S from the Money Market Account and invest it in the risky asset. At expiry $t = 1$, we have either dS or uS in the risky asset. So after paying off the debt from the money market account, either we have a profit of $(u - d)S$ or zero dollars. Thus we have a positive probability of making money and zero probability of losing, starting from zero initial capital, which is an arbitrage opportunity. This violates our assumption that the asset model is arbitrage free.

Note also that if $q = 1$, $u = 1 + r$. In this case, we can again construct an arbitrage opportunity: short the risky asset and put the S dollars into the Money Market Account. Then at $t = 1$, we have uS dollars in the Money Market Account which we can use to close our short position in the risky asset leaving us with a profit of either $(u - d)S$ or zero dollars. Again, this is an arbitrage opportunity. Thus, we know $0 < q < 1$, and therefore $d < (1 + r) < u$.

$(\Leftarrow)$ If $d < (1 + r) < u$ then the model is arbitrage free:

If $d < (1+r) < u$, then

$$q = \frac{(1+r)-d}{u-d},$$

defines the risk neutral measure $\tilde{P}$ for the upward movement of S and $0 < q < 1$. Under this measure, every discounted portfolio process $D_t X_t$ is a martingale [Shr05], where $D_t = e^{-rt}$ is the discounting factor. Thus $\widetilde{\mathbb{E}}[e^{-rt}X_t] = X_0$, where $\widetilde{\mathbb{E}}$ is the expectation under the risk neutral measure [Shr10].

Suppose there exists an arbitrage strategy, so that the corresponding portfolio X satisfies $X_0 = 0$, so $\widetilde{\mathbb{E}}[D_n X_n] = 0$. On the other hand, there is zero probability of losing money, so

$$\widetilde{P}\{X_n < 0\} = 0.$$

Together they imply that

$$\widetilde{P}\{X_n > 0\} = 0.$$

Since P is equivalent to $\tilde{P}$, this must also hold for P, which contradicts that X is an arbitrage. Therefore, the model must be arbitrage free. □

In the rest of the discussion we focus on the CRR$(n; S, r, \sigma, T)$ model which is a special case of the binomial tree model we defined above with interest rate $e^{-r\Delta t} - 1$, $u = e^{\sigma\sqrt{\Delta t}}$, $d = \frac{1}{u}$, and the risk neutral measure $q = \frac{e^{r\Delta t} - e^{-\sigma\sqrt{\Delta t}}}{e^{\sigma\sqrt{\Delta t}} - e^{-\sigma\sqrt{\Delta t}}}$, where $\Delta t = T/n$. This is the counterpart of the Black Scholes model in discrete time setting.

2.3 Computation of European and American Option Prices in CRR Model

In this section, we introduce the calculation of European and American option prices under the CRR model defined in the previous section [2]. The price $C_{t,i}$ of the European options (call or put) at the ith node at each time t is calcuated in an recursive way:

$$C_{t-\Delta t,i} = e^{-r\Delta t}(qC_{t,i} + (1-q)C_{t,i+1}), \tag{1}$$

where q is the risk-neutral probablity, and the terminal value (the starting point of the calculation) $C_{T,i} = (S_{T,i} - K)^+$ for the call option and $C_{T,i} = (K - S_{T,i})^+$ for the put option, is the payoff of the options at the expiration date.

Unlike European options, American options allow holders to exercise at any time up to and including the expiration date. To properly calculate the output tree, we follow the dynamic programming principle: first, we calculate the option value at each node if it is not exercised at this point, using the expected discounted payoff under the risk neutral measure, the same as for European options above. Then we update this value by allowing early exercising, i.e. finding the maximum between the option value if the holder wait to

[2] Wikipedia, `https://en.wikipedia.org/wiki/Binomial_options_pricing_model`

exercise later and the value if exercised now. Thus the value of American call option $A_{t,i}$ can be caluclated using the following recursive formula (for put options, we only need to change the value of immediate exercising accordingly):

$$A_{t-\Delta t,i} = max\left(S_{t-\Delta t} - K, e^{-r\Delta t}(qA_{t,i} + (1-q)A_{t,i+1})\right). \tag{2}$$

The terminal value is the same as the European options, because at the expiration date T, the holder of the option has to decide whether to exercise or not, and waiting is not an option anymore.

2.4 The Black-Scholes Model

The Black-Scholes model is a continuous time model of a financial market with one stock S which follows $\frac{dS_t}{S_t} = (\mu + r)dt + \sigma dW_t$ with initial value S_0, where the expected excess return μ, the interest rate r, and the volatility σ are positive constants and W_t is a brownian motion, and a risk-free asset, B_t which follows $\frac{dB_t}{B_t} = rdt$. The stock price follows lognormal distribution:

$$\ln \frac{S_T}{S_0} \sim \mathcal{N}((r - \frac{1}{2}\sigma^2)T, \sigma^2 T)$$

under the risk neutral measure.

The Call and Put price (C_0 and P_0 respetively) with maturity T and strike price K have closed form solution in this model [Shr10]:

$$C_0 = S_0\Phi(d_1) - Ke^{-rT}\Phi(d_2),$$

and

$$P_0 = Ke^{-rT}\Phi(-d_2) - S_0\Phi(-d_1),$$

where

$$d_1 = \frac{1}{\sigma\sqrt{(T-t)}}\left[\ln\frac{S_0}{K} + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right], \quad d_2 = d_1 - \sigma\sqrt{T-t},$$

and Φ is the cumulative distribution function of the Standard Normal Distribution:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{t^2}{2}}dt.$$

3 Exploring Modern Approach: Neural Networks

Inspired by the successes of machine learning in various industries in the recent years, the goal of our project is to apply machine learning to finance, focusing on the applications of neural networks in option pricing.

A neural network is a technique for approximating complex functions that do not have closed-form expressions. For any function f, starting from a batch of $(x, f(x))$ pairs, the

network iterates through compositions of simple functions, and steps towards the optimal composition, which minimizes the aggregate error of the approximiation from the bacch - the so called loss function.

A neural network is typically made up of layers of neurons. These "neurons" are simple functions with closed form expressions. A layer can have any number of neurons and a network can contain any number of internal layers. By nature, a network will always include an input and output layer. The input layer takes in the input and passes it into each neuron in the following layer. The output layer takes all values produced by the last layer and condenses them into the specified output size [PCa19]. For example, consider a neural network with 1 layer containing only 1 neuron - a linear function. The network tries to find the best linear function to approximate the relationship between the given batch of $(x, f(x))$ pairs, which is actually a linear regression problem

$$\min_{a,b}(f(x) - (a + bx)). \tag{3}$$

Note that in this case, a unique solution is guarenteed if there are only two data points. The relationship between any number of points greater than that can be uniquely solved if the functuon f itself is linearas explored in Section 3.1. If that is not the case, the relationship can be approximated.

For more complex problems, we can construct neural networks with more layers and neurons for better approximations. Consider another layer of one linear function added to the above moddel, so that the result from equation (3) is passed as the input to another linear function. Then the probelm expands to approximate f with the composition of two linear functions:

$$\min_{a_1,a_2,b_1,b_2}(f(x) - (a_1x + b_1)a_2 + b_2), \tag{4}$$

where the a_i's and b_i's are constants to be chosen for the i^{th} layer.

A single layer can also be made up of multiple neurons. In this case, each neuron is a different simple function, and rather than a composition of functions like Equation (4), this network processes multiple functions simultaneously and selects the best performing for the next iteration using an activation function, discussed below. It is worth noting that the number of neurons and layers directly impact the computational mass and time required for the network to execute the approximation.

Figure 3 shows a basic neural network with 2 internal layers composed of 3 neurons each. The network begins with the input-output pairs, $(x, f(x))$. The input layer passes x to each neuron in the first internal layer. Each a_i in this first layer is multiplied by x, which represents a linear function in x. Then, a_ix passes through an activation function. This activation function chooses whether to turn this neuron's result "on" or "off", i.e. whether to feed into the next layer of nuerons, based on if the approximation is good enough. The two most common types of activation functions are ReLU and Sigmoid,

$$ReLU : \phi(x) = \max\{0, x\},$$

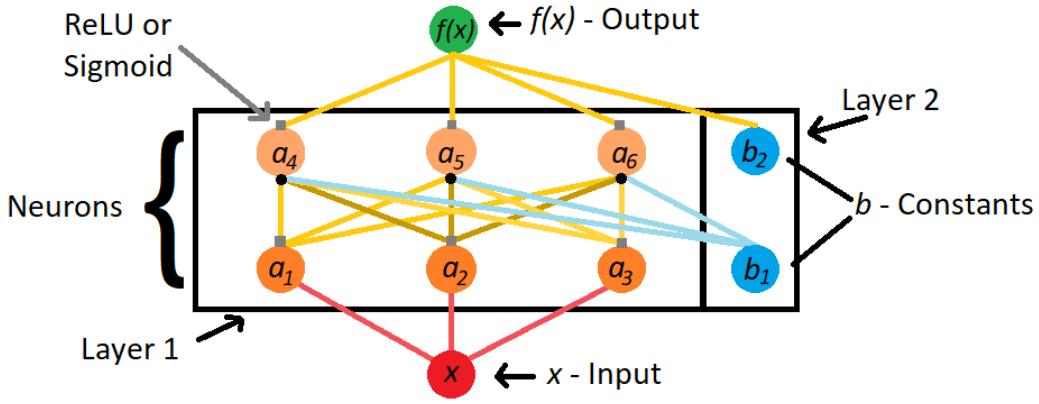


Figure 3: A Basic Neural Network Diagram

$$Sigmoid : \phi(x) = \frac{1}{1+e^{-x}}.$$

After $a_i x$ passes through the activation function, the constant b_i is added and the result is composed with each simple function of the next layer. Finally, each product of the neurons of the second layer passes through another activation function. The results of the activation functions are then passed into the output layer, where a single value is determined. During the training of the model, this value, $N(x)$ is then compared to the true given value, $f(x)$. The loss is assessed and adjustments are made by adjusting a_i's and b_i's to minimize the loss. Mathematically, the above procedure can be represented as:

$$N(x) = \phi_2 \left(\left(\phi_1 \left(\begin{bmatrix} x \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \end{bmatrix} \right) + \begin{bmatrix} b_1 & b_1 & b_1 \end{bmatrix} \right) \begin{bmatrix} a_{21} \\ a_{22} \\ a_{23} \end{bmatrix} \right) + b_2, \tag{5}$$

where ϕ_i is an activation function, a_{ij} is the coefficient associated with the j^{th} neuron on the i^{th} layer, and b_i is the constant associated with the i^{th} layer.

The above procedure is repeated for a large number of times. Each of such iteration is call an "epoch", which uses a newly generated data set, and further reduces the estimation error, until it is less than a pre-detemrined confidence level. The measure of accuracy for the model training is defined as the Mean Squared Error (MSE)

$$MSE = \sum_{i=1}^{n} (f(x) - N(x))^2,$$

where $N(x)$ is the estimated value for each x by the neural network. The MSE and learning rate which determine how fast our neural network changes itself are then used to optimize

```
In [0]: #model
        #nn.Linear
        in_dim = 1
        out_dim = 1

        model = nn.Sequential(
            nn.Linear(in_dim, 30),
        #    nn.ReLU(),
            nn.Linear(30, out_dim)
        )
```

Figure 4: The Neural Network in Python for Linear Functions

the neural network. In each epoch, the MSE is multiplied by the learning rate and the value is used to modify the multiplier of each neuron. The complete code for this example can be found in Appendix 7.2.1.

3.1 The Approximation of Linear and Non-Linear Functions via Neural Network

This section explores an example of implementing a neural network to approximate (or replication, because the approximation is exact in this case) a linear function.

Suppose the data - the batch of pairs of $(x, f(x))$ are drawn from the target function

$$f(x) = x + 2. \tag{6}$$

The neural network is defined using the Pytorch Python package, as shown in Figure 4.

There are 2 layers of 30 neurons each. Note that the step invoking the activation function ReLU is removed from the network because the target function itself is linear, and having ReLU in the network negatively impacts the accuracy and efficiency of the model. After initializing the model, the training data, $n = 1000$ pairs of $(x, f(x) = x + 2)$, are randomly generated. These newly generated pair are fed into the neural network for a total of 10000 epochs.

Once the model is trained, it is tested using another randomly generated batch of input-output pairs $(x, f(x))$. The closer the predicted value $N(x)$ from the neural network is to the actual value $f(x)$, the more accurate the network is. The results of the optimized model are illustrated in Figure 5. The graph shows that the model is accurate since the predicted results are linear and completely overlap the actual points.

In order to approximate non-linear functuons, like option prices, we need to add activation functions back to the neural network constructed above for linear functions. We test the accuracy of this neural network for non-linear polynomials of different degrees, and the result was poor (as shown in Figure 6) due to the insufficient number of neurons

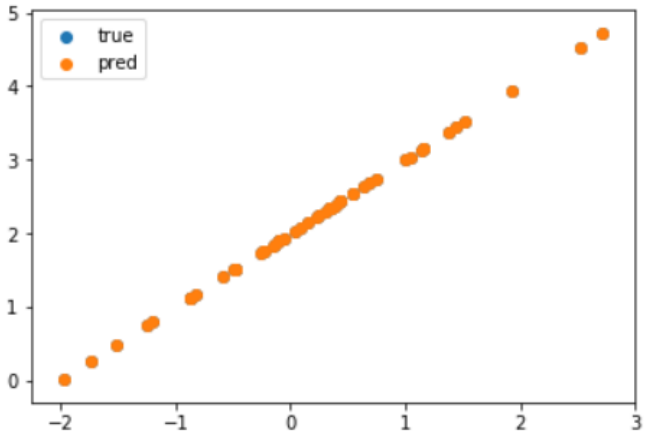


Figure 5: Actual and Predicted Values of $f(x) = x + 2$

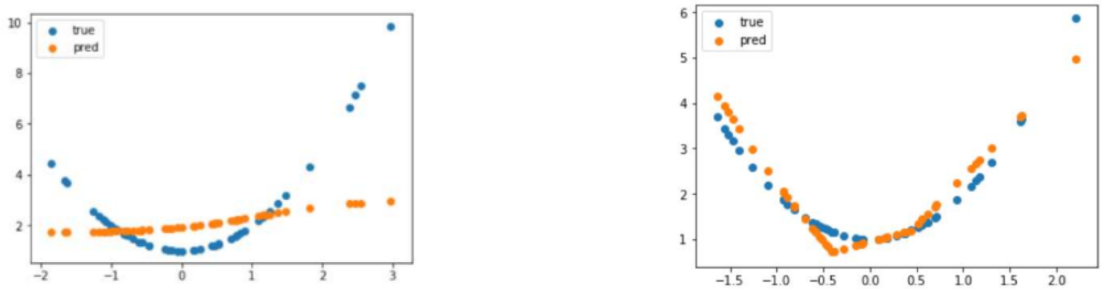


Figure 6: Approximation with insufficient neurons

and layers. We carry out a sequence of experiments by increasing the number of neurons and layers. The loss function decreases with the complexity of the neural network and the improvement of the fitting was visibly noticable in Figure 7, with results of Sigmoid activation function on the left and ReLU on the right. From our experiments, shown in Appendix 7.2.3 and Figure 7, the number of neurons plays a larger role than the number of layers in determining the accuracy of the predicted graph.

3.2 Neural Network Optimizer

To expand on our understanding of neural networks, a fascinating thing we find is that polynomials of different degrees react differently to the same neural network setup. Figure 8 shows that higher number of layers is not always more optimal both in term of accuracy and

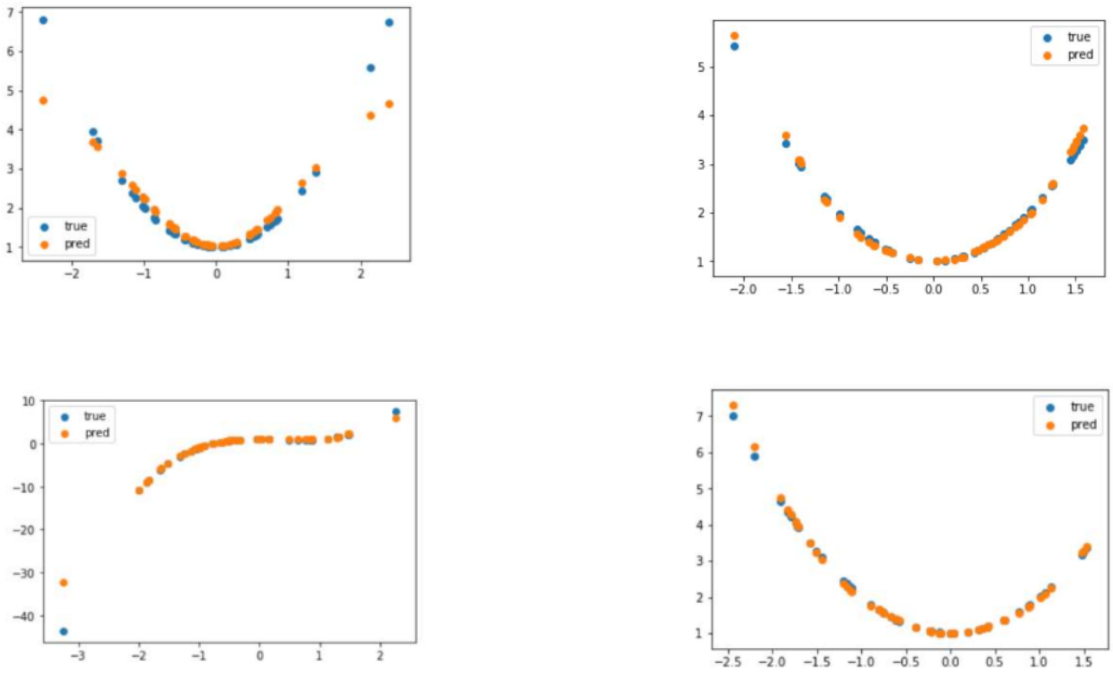


Figure 7: Approximation with 100 and 1000 neurons

```
Epoch [500/10000], Loss: 2.1053
Epoch [1000/10000], Loss: 0.4020
Epoch [1500/10000], Loss: 1.0503
Epoch [2000/10000], Loss: 0.4138
Epoch [2500/10000], Loss: 0.2278
Epoch [3000/10000], Loss: 0.1905
Epoch [3500/10000], Loss: 0.1372
Epoch [4000/10000], Loss: 0.1106
Epoch [4500/10000], Loss: 0.0987
Epoch [5000/10000], Loss: 0.0902
Epoch [5500/10000], Loss: 0.0840
Epoch [6000/10000], Loss: 0.0793
Epoch [6500/10000], Loss: 0.0761
Epoch [7000/10000], Loss: 0.0741
Epoch [7500/10000], Loss: 0.0713
Epoch [8000/10000], Loss: 0.0698
Epoch [8500/10000], Loss: 0.0684
Epoch [9000/10000], Loss: 0.0671
Epoch [9500/10000], Loss: 0.0661
Epoch [10000/10000], Loss: 0.0610
```

(a) 4th-Degree Polynomial with 3 layers

```
Epoch [500/10000], Loss: 0.2827
Epoch [1000/10000], Loss: 0.2442
Epoch [1500/10000], Loss: 0.2285
Epoch [2000/10000], Loss: 0.1762
Epoch [2500/10000], Loss: 0.1510
Epoch [3000/10000], Loss: 0.1263
Epoch [3500/10000], Loss: 0.1221
Epoch [4000/10000], Loss: 0.1204
Epoch [4500/10000], Loss: 0.1198
Epoch [5000/10000], Loss: 0.1193
Epoch [5500/10000], Loss: 0.1164
Epoch [6000/10000], Loss: 0.1120
Epoch [6500/10000], Loss: 0.1107
Epoch [7000/10000], Loss: 0.1100
Epoch [7500/10000], Loss: 0.1095
Epoch [8000/10000], Loss: 0.1092
Epoch [8500/10000], Loss: 0.1089
Epoch [9000/10000], Loss: 0.1087
Epoch [9500/10000], Loss: 0.1086
Epoch [10000/10000], Loss: 0.1085
```

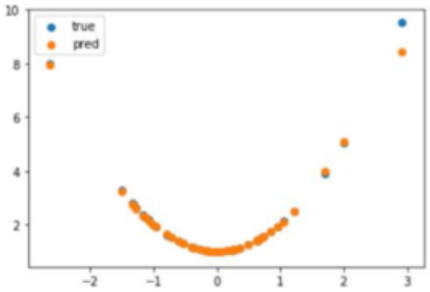
(b) 4th-Degree Polynomial with 4 layers

Figure 8: Loss Functions of Approximating 4th-Degree Polynomial with Different Neural Networks

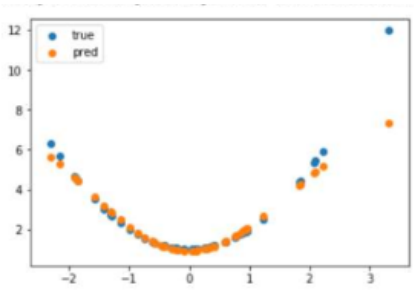
time efficiency. Further experiments with Sigmoid and ReLU as activation functions, also show that polynomials of different degrees work better with different setups. With Sigmoid function, the algorithm runs slightly faster compared to ReLU but left with larger loss with the same number of neurons and layers, mainly due to the fact that the approximation tends to diverge from the true value for x's with large absolute values. Thus it would be a good idea to include a model optimizer, which customize the structure of the neural network, which we could use at the later stage of our project.

The optimizer that we wrote requires 4 inputs. The maximum number of neurons, layers to test, acceptable loss and the actual function that we want to test neural network against. The optimizer then will run every combination starting from 2 neurons and 2 layers up to the input amount. Two setup statistic will be returned. One is the setup with the lowest loss and the other is the setup with the shortest execution time and loss less than the acceptable level 0.001. For the later stage of our project, we mostly use the second setup due to its practicality. The code for the setup-optimizer can be found in the Appendix 7.2.5.

Lastly, just to see how close we can get to 0 loss, we leave our machine running for a few hours with over 10000 neurons in each layer. We are able to get the loss function to 0 up to more than 5 decimal places but the process is impractical due to the amount of time



(a) Approximated Graph vs True Graph with ReLU



(b) Approximated Graph vs True Graph with Sigmoi]d

Figure 9: Approximated Graph vs True Graph with Different Activation Functions

needed. Nevertheless, we may see different result given a more powerful machine such as super computer.

4 Computation of Option Prices via Neural Networks

In this section, we apply neural network techniques for non-linear functions, which is developed in the last section, to approximate the non-linear options prices in both Black-Schoels Model and Binomial Tree (CRR) model.

4.1 European Option Prices in Black-Scholes Model

First, we try to use neural networks to approximate the Black-Scholes formula for European call option price. The first step is to obtain a set of training data. $n = 21$ random numbers are sampled from the standard normal distribution and multiplied by 100 to serve as the sample of underlying stock prices x. Then $f(x)$, the call option price, is calculated using the Black-Scholes formula for each x, with the strike price $K = 110$. This creates a set of input-output pairs for use in training the model.

The neural network consists of four layers: a layer that takes in 1 input and outputs 50, a layer that takes in 50 and outputs 12, a layer that takes in 12 and outputs 2, and a layer that takes in 2 and outputs 1. Note that in each case, the layer contains a number of neurons equal to the input number, and these neurons combine and simplify to create the ouput [PCa19]. For example, the first internal layer has 50 neurons that produce a total of 12 results. This model is illustrated in Figure 10, where H1 is 50 and H2 is 12. Notice that the input and output dimensions for a single layer do not have to agree, while the dimensions between layers must agree.

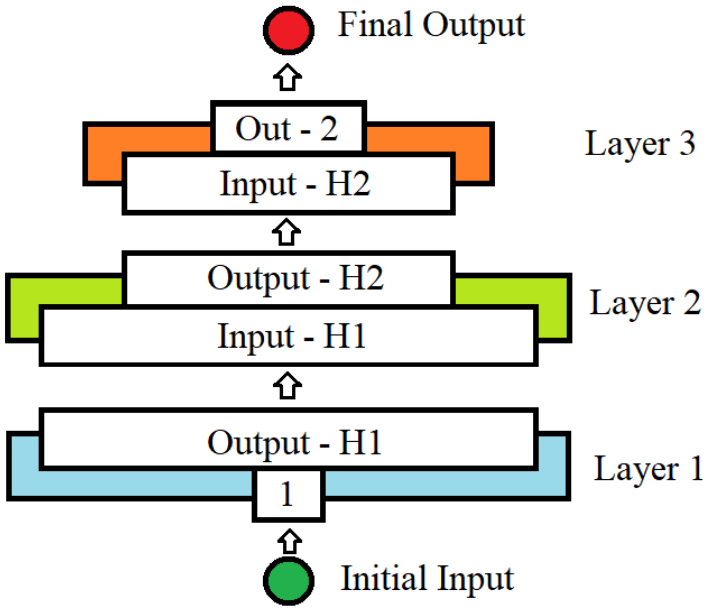


Figure 10: Illustration of the BSM Neural Network

Once the model is trained, a second set of simulated stock prices is generated, which are used both to calculate the true option price using the Black-Scholes formula, and the approximation using the trained neural network. The true and predicted prices are plotted against the underlying stock price in Figure 11. As illustraed in the graph, this first version of the program performs poorly. Not only is the error large, the predictions are completely wrong - it is a linear function of the stock price, with a slope of nearly zero.

Naturally, this leads to an investigation on why the model was behaving this way. There are possibly many approaches to improving the approxmiation (see e.g. [Lu+19]). After some experimentations, we discover that uniformly transforming the data leads to better results. Regarding all data points as a vector x, the transformed data vector, x_N is defined as

$$x_N = \frac{u-l}{\max(x) - \min(x)}(x - \min x) + l,$$

where u and l are the upper and lower scaling parameters for the transform respectively. Difference choices of l and u produce difference accuracy in the approximation using neural networks. After some experimentation, it was deduced that a normalization of the $f(x)$ vector ($l = 0, u = 1$) and a scaling of the x vector ($l = -1, u = 1$) corresponds to the best accuracy. The exact reason for this is unclear. However, it is possible that the network is more able to detect the pattern $f(x)$ with a less noisy data set. As seen in the left panel of Figure 12, after the above linear transformation, the model is able to accurately calculate

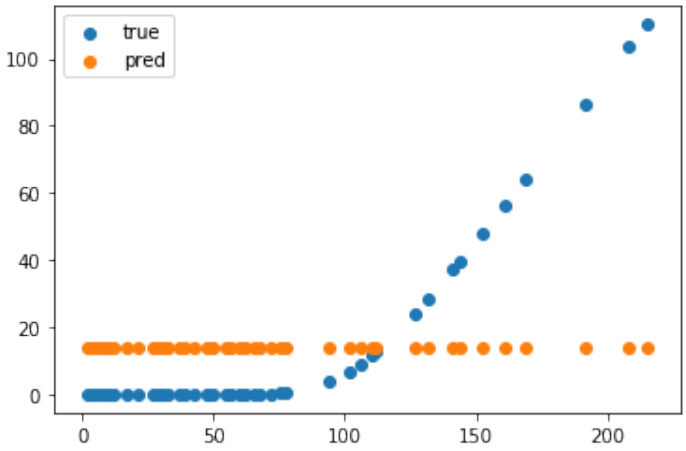
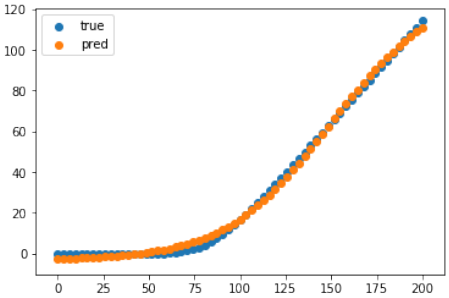
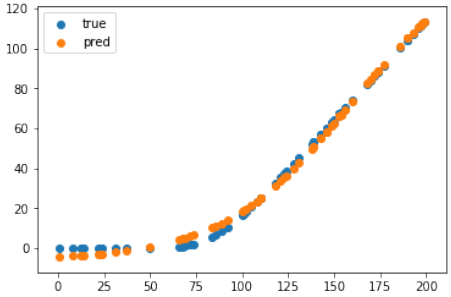


Figure 11: Initial Result for the Black-Scholes Neural Network



(a) Normalized Training Data Results



(b) Normalized Random Data Results

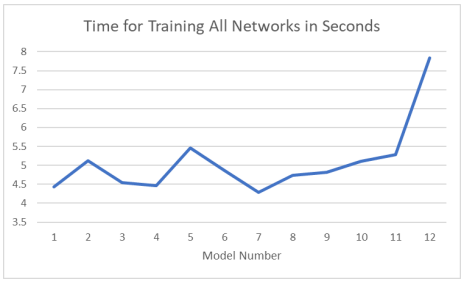
Figure 12: Results with Normalized Data

the values of the Call options for the training data. Additionally, as seen in the right panel of Figure 12, the model is able to much more accurately predict the prices of the options for random, normalized data.

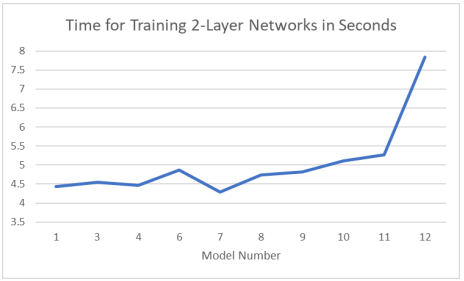
While the result after the data transformation is vastly superior to the original one, further experiments are performed by varying the number of neurons and layers. From the data summarized in Table 1 and the figures in Appendix 7.3, it is clear that, in most cases, increasing the number of neurons increases accuracy, but increasing the number of layers increases computation time and yields a higher loss. In the table, the Model Column denotes the number of neurons in the first, second, and third layer respectively. The Loss column contains the remaining loss after 5000 iterations of training, and the Time Column indicates the time for the training of each network in seconds. The total elapsed time for each model is also plotted in Figure 13a. The two peaks in the graph corresponds to

	Model	Loss	Time
1	[50, 12]	0.00057448	4.441021919
2	[50, 12, 10]	0.00055705	5.121997118
3	[50, 30]	0.00072565	4.549135208
4	[50, 50]	0.00055633	4.46467185
5	[50, 50, 100]	0.00111636	5.457895279
6	[50, 100]	0.00067071	4.867865562
7	[80, 20]	0.00055201	4.295755148
8	[100, 80]	0.00044758	4.736923695
9	[100, 100]	0.00052513	4.812922478
10	[100, 120]	0.00042859	5.106760263
11	[100, 150]	0.00049361	5.278343201
12	[500, 100]	0.00030057	7.834435225

Table 1: Comparison of Different Networks



(a) All Models



(b) Only 2-Layer Models

Figure 13: Time for Model Trainings

Networks 2 and 5, which have three layers, and Figure 13b shows the times with these two models removed. Typically, increasing neurons and adding another layer to an otherwise identical model increases the computation time. The code of the Black-Scholes Neural Network can be found in Appendix 7.2.4.

4.2 CRR Model with Neural Network

In this section, we test the effectiveness of neural network in estimating the options prices in the CRR model, which can be calculated using the algorithm described in Section 2.3. We run our neural network against the American Put prices, as a function of the initial stocl price, the same way as the approximation of Black-Scholes formula with data normalization in the previous section. 100 test values are fed into each epoch. The result is promising as the final loss function converges to 0.0001 as shown on Figure 14.

```
Epoch [500/10000], Loss: 0.0017
Epoch [1000/10000], Loss: 0.0012
Epoch [1500/10000], Loss: 0.0010
Epoch [2000/10000], Loss: 0.0008
Epoch [2500/10000], Loss: 0.0007
Epoch [3000/10000], Loss: 0.0006
Epoch [3500/10000], Loss: 0.0006
Epoch [4000/10000], Loss: 0.0005
Epoch [4500/10000], Loss: 0.0004
Epoch [5000/10000], Loss: 0.0004
Epoch [5500/10000], Loss: 0.0003
Epoch [6000/10000], Loss: 0.0003
Epoch [6500/10000], Loss: 0.0003
Epoch [7000/10000], Loss: 0.0002
Epoch [7500/10000], Loss: 0.0002
Epoch [8000/10000], Loss: 0.0002
Epoch [8500/10000], Loss: 0.0002
Epoch [9000/10000], Loss: 0.0002
Epoch [9500/10000], Loss: 0.0002
Epoch [10000/10000], Loss: 0.0001
```

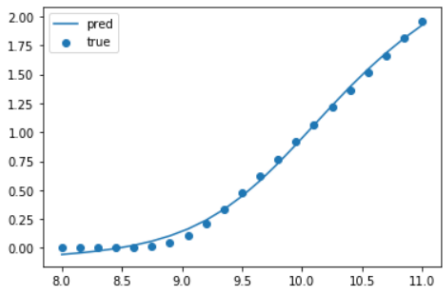
Figure 14: Loss Function of American Put Option per 500 Epoch

We run more tests on the neural network because even though our input data was randomly generated, they do not have the same volatility nor the magnitude of the real data. As such, the loss function may have been scaled down and we end up with a misinterpretation of the results. It turns out that the machine is in factworking better than we anticipated. The reason for this unusually low loss function is probably due to the large number of epochs. When we scaled out the total number and print out 10 times as often, we can clearly see the initial inaccuracy of the neural network and how fast it is correcting itself. Only 5000 epoch is needed to get an acceptable result from the predicted graph as shown in Figure 15 for American Call and Put options. Aside from the slight difference for intermediate intial stoch price, the predicted closely resembles the true graph. In most cases, the neural network was able to quickly converge to a loss less than 0.001 and in rare occasion down to 0.0001. Full Code and output is provided in the Appendix 7.2.2 and Appendix 7.4.

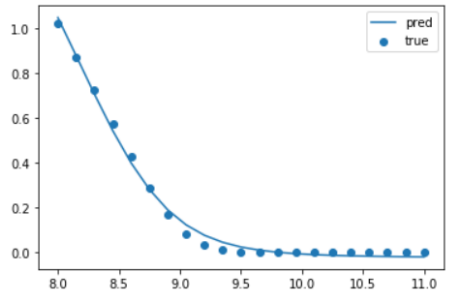
4.3 Universal Approximation Theorem

In mathematical theories behind artificial neural networks, universal approximation theorem[3] suggests that with enough neurons, a single hidden layer of neural network can approximate any continuous function on compact subset of real numnbers $\mathbb{R}$. Specifically, any n+1 width single layer deep neural network can approximate any continuous function of n-dimensional variables. In the case of deep neural networks, universal approximator

[3]Wikipedia `https://en.wikipedia.org/wiki/Universal_approximation_theorem`.



(a) American Call True vs Predicted Graph



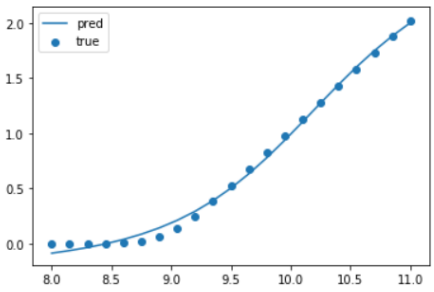
(b) American Put True vs Predicted Graph

Figure 15: American Call and Put Approximation with 5000 Epoch

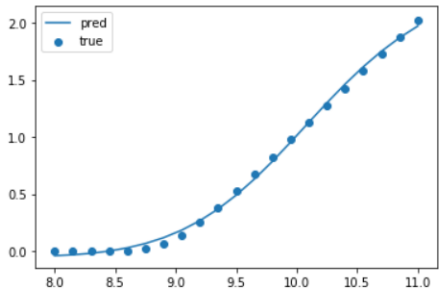
works if and only if the activation function is not polynomial. In this section, we test this theory with some examples.

Knowing that given enough time, neurons, and layers, any graph can be approximated, we want to test the efficiency of neural networks of differnt structures given limited resources. To do so, we choose a fixed number of neurons that we further divide among 3 layers in one case, while keeping all of them in the same layer in the other.

From the approximation, both have roughly the same loss of 0.0003, as shown in Figure 16. However, the speed of the calculation for the single layer is much faster at 3.02s while the 3 layers network takes 4.41s. In the case that we switch Sigmoid to ReLU in a single layer, the loss drops to 0, giving us a perfect prediction in Figure 17. With the above demonstration of the Universal Approximation Theorem, for any financial quantities that we want to approximate, given that they closely resemble polynomial functions, we can be confident in accurately generating a predicted graphs using neural networks with only one layer of sufficiently large number of nuerons.



(a) Predicted Graph of 3 Layers with 6, 5, and 3 neurons in each layer



(b) Predicted Graph of 1 layer with 14 neurons

Figure 16: Predicted Graph with 1 and 3 layers of neurons

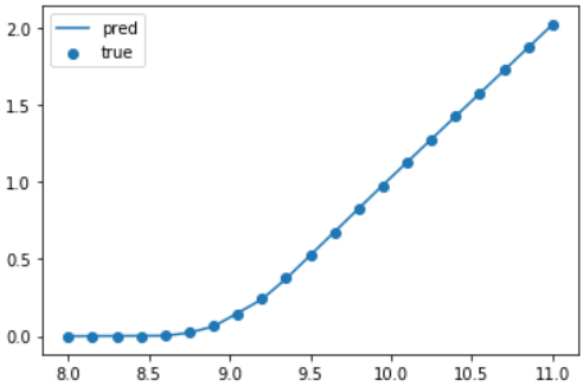


Figure 17: Predicted graph of 1 layer and 14 neurons with ReLU instead of Sigmoid

5 Applications: Implied Volatility

In this section, we explore further applications of machine learning to other quantities associated to option contracts, which like many other problems in mathematical finance, do not have a closed form solution and are computationally expensive. One of such problems is the estimation of implied volatility surface [Shr10].

5.1 Implied Volatility

To find the fair premium of an option contract, we apply the Black-Scholes equation. This operation takes into account 5 parameters of the contract and the underlying asset: current price (S), strike (K), time to expiration (T), risk-free rate (r) and volatility (σ). The first 4

are easy to retrieve from the market. Volatility describes the variability of the stock prices, and the past practices have used the value estimated from historical data as a proxy. This backward looking method attracts criticism in that what happens in the past does not necessarily indictes the future.

Implied volatility is a foward looking way, in that we can use the market price of an option contract and backtrack the Black-Scholes formula to obtain the market's expected value of the underlying's volatility for the duration of the contract. To do so, we have to calculate the Vega of the option.

Vega measures the sensitivity of the option's premium to volatility. It is important to notice that the higher the volatility the better for both Calls and Puts as it increases the probability of having the option end up in the money, which agree with the following closed-form calculation from the Black-Scholes formula that it is always strictly positive, and the same for both call and put options:

$$SN'(d_1)\sqrt{T}. \tag{7}$$

In the following, we will demonstratet the derivation of the implied volatility using call option prices, and the calculations for the put option follows the same procedure. With Vega being strictly positive, the option price as a function the volatility is invertible, and we can calculate the implied volatitlity by matching the market price C_mkt and the theoretical value $C_BSM(\sigma)$.

The equation $C_mkt - C_BSM(\sigma) = 0$ does not have a closed-form solution, and we need help from numerical methods. In Numerical Analysis, Newton's method is used for approximating the roots of differentiable functions. It is an iterative method, and in each step uses the intersection of the tangent line of a given point x_n to determine the next value x_{n+1}.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{8}$$

The procedure continues until the absolute value of $x_{n+1} - x_n$ is smaller than a pre-set threshold. Figure 18 illustrates this process, where x_n's are the sequence of approximations for the implied volatility and $f(x) = C_mkt - C_BSM(x)$. The Python code of this calcaultion is in Section 7.5.

5.2 Volatility Surface

It is important to study the behavior of implied volatility, specifically how it changes when varying the strike and the time of maturity of an option. Deep out-the-money and in-the-money options tend to have a higher implied volatility as the market expects that there is still a slight probability of the underlysing asset moving towards the strike (at-the-money). The same holds in terms of time to maturity. The longer the expiry, the more uncertainty the underlying asset has, and thus the higher implied volatility.

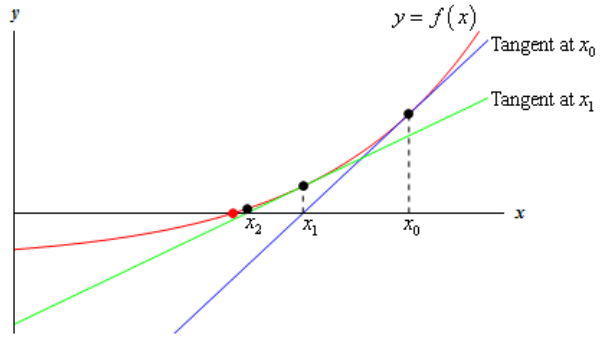
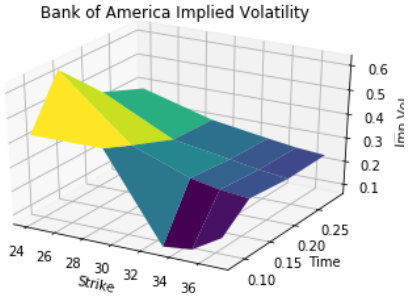
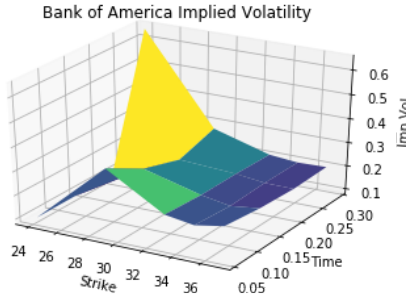


Figure 18: Illustration of Newton's Method



(a) Volatility Surface of BAC Dec 2019



(b) Volatility Surface of BAC Jan 2020

Figure 19: olatility Surface of BAC

The following are some examples of how the implied volatility surface, as a function of underlying asset price and time to maturity. Figure 19 compares the surface for call options on stocks of Bank of America (NYSE: BAC), in December 2019 and January 2020. The surface's shape for the two contracts are very different. This change happen due to different market expectations of the performance of the BAC stock. An implied volatility surface is not stable as news, changes in economic policy and balance sheet, which affect both liquidity and pricing of option contracts.

The next example in Figure 20 shows the volatility surface for call options on stocks of Microsoft (NYSE: MSFT) in January 2020. It highlights that despite the majority of deep out-of-money and deep in-the-money contract have relatively higher prices, or equivalently, higher implied volatilities, the surface shape does not form the expected "smile". As mentioned above, cases like this are not uncommon due to changes in the option prince in accordance to market's beliefs.

Analysis of the implied volaitlity surface assist traders, risk managers and mathematicians on a daily basis. Approximating implied volatility surfaces could auxiliate traders, specially high frenquency traders to obtain accurate estimations, allowing finance pro-

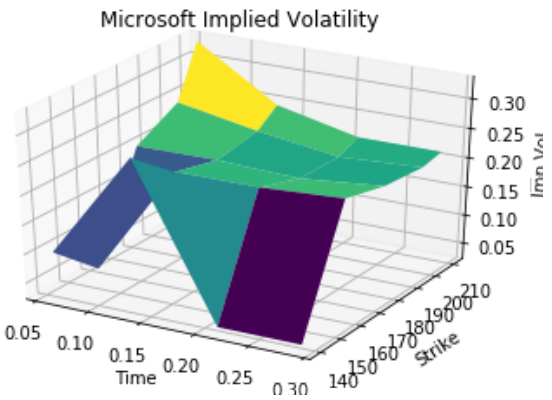


Figure 20: Volatility Surface MSFT Jan 2020

fessionals to leverage new mathematical models when devising and executing investment strategies. However, the need for faster computations highlight the limitations brought by traditional numerical methods. The use of machine learning may help estimating the volatilty surface from fewer data points (all of which needs numerical calculations) than what is needed for the traditionally used interpolation method, and ease the computationally heavy burden brought by large number of strikes and maturities, and the ever changing option prices.

6 Conclusion

Our result shows that neural network is an effective tools in predicting the trend of many data sets. By training it against CRR and BSM models, it was able to reach desirable result relatively quickly within 5 to 10 seconds, on estimating the option prices. This solution can help approximate functions or models that can not be solved using traditional method due to their complexity or the limited resources available for the computation. The machine learning techniques can act as a maleable model that helps us get a "good enough" approximation of a function in a reasonable timeframe. Even though there are still some limitation due to it being a model-based machine learning, we believe that neural network can lay the foundation for future development in computational finance.

7 Appendix

7.1 Derivation of Vega

$$\frac{\partial C}{\partial \sigma} = S\frac{\partial N(d_1)}{\partial \sigma} - Ke^{-rT}\frac{\partial N(d_2)}{\partial \sigma} \tag{9}$$

$$= N(d_1) + S\frac{\partial N(d_1)}{\partial d_1}\frac{\partial d_1}{\partial \sigma} - Ke^{-rT}\frac{\partial N(d_2)}{\partial d_2}\frac{\partial d_2}{\partial \sigma} \tag{10}$$

$$= S\frac{1}{\sqrt{2\pi}}e^{\frac{-d_1^2}{2}}\left(\frac{\sigma^2 T^{\frac{3}{2}} - \left[\ln\frac{S}{K} + \left(r + \frac{\sigma^2}{2}\right)T\right]T^{\frac{1}{2}}}{\sigma^2 T}\right)$$
$$- Ke^{-rT}\left(S\frac{1}{\sqrt{2\pi}}e^{\frac{-d_1^2}{2}}\frac{S}{K}e^{rT}\right)\left(\frac{-\left[\ln\frac{S}{K} + \left(r + \frac{\sigma^2}{2}\right)T\right]T^{\frac{1}{2}}}{\sigma^2 T}\right) \tag{11}$$

$$= S\frac{1}{\sqrt{2\pi}}e^{\frac{-d_1^2}{2}}\left(\frac{\sigma^2 T^{\frac{3}{2}} - \left[\ln\frac{S}{K} + \left(r + \frac{\sigma^2}{2}\right)T\right]T^{\frac{1}{2}}}{\sigma^2 T}\right)$$
$$- S\frac{1}{\sqrt{2\pi}}e^{\frac{-d_1^2}{2}}\left(\frac{-\left[\ln\frac{S}{K} + \left(r + \frac{\sigma^2}{2}\right)T\right]T^{\frac{1}{2}}}{\sigma^2 T}\right) \tag{12}$$

$$= SN'(d_1)\sqrt{T} \tag{13}$$

7.2 Python Code

7.2.1 Approximating a Linear AND Non-Linear Function Using a Neural Network

```
In [0]: import torch
        import torch.nn as nn
        import numpy as np
        import torch.nn.functional as F
        import matplotlib.pyplot as plt
```

Figure 21: Python Imports

```
In [0]: # target function
        a = 1.
        b = 2.
        f = lambda x: a*x+b

        #test
        f(2.)

Out[0]: 4.0
```

Figure 22: Target function

```
In [0]: #model
        #nn.Linear
        in_dim = 1
        out_dim = 1

        model = nn.Sequential(
            nn.Linear(in_dim, 30),
        #    nn.ReLU(),
            nn.Linear(30, out_dim)
        )
```

Figure 23: Neural Network

```
In [0]: #loss function
        criterion = nn.MSELoss()
```

Figure 24: Loss Function

```
In [0]: #optimizer
        learning_rate = 0.001
        optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Figure 25: Learning Rate and Optimizer

In [0]:
```
#training data

batch_size = 1000

x_train = torch.randn(batch_size, 1)
y_train = f(x_train)
```

Figure 26: Randomized Training Data

In [0]:
```
# Train the model

num_epochs = 500

for epoch in range(num_epochs):

    # Forward pass
    outputs = model(x_train)
    loss = criterion(outputs, y_train)

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 10 == 0:
        print ('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1,
                                                    num_epochs, loss.item()))
```

Figure 27: Training Loop

```
Epoch [10/500], Loss: 3.6593
Epoch [20/500], Loss: 2.1498
Epoch [30/500], Loss: 1.2767
Epoch [40/500], Loss: 0.7678
Epoch [50/500], Loss: 0.4685
Epoch [60/500], Loss: 0.2906
Epoch [70/500], Loss: 0.1835
Epoch [80/500], Loss: 0.1180
Epoch [90/500], Loss: 0.0773
Epoch [100/500], Loss: 0.0515
Epoch [110/500], Loss: 0.0348
Epoch [120/500], Loss: 0.0239
Epoch [130/500], Loss: 0.0166
Epoch [140/500], Loss: 0.0116
Epoch [150/500], Loss: 0.0082
Epoch [160/500], Loss: 0.0058
Epoch [170/500], Loss: 0.0041
Epoch [180/500], Loss: 0.0030
Epoch [190/500], Loss: 0.0021
Epoch [200/500], Loss: 0.0015
Epoch [210/500], Loss: 0.0011
Epoch [220/500], Loss: 0.0008
Epoch [230/500], Loss: 0.0006
Epoch [240/500], Loss: 0.0004
Epoch [250/500], Loss: 0.0003
Epoch [260/500], Loss: 0.0002
Epoch [270/500], Loss: 0.0002
Epoch [280/500], Loss: 0.0001
Epoch [290/500], Loss: 0.0001
Epoch [300/500], Loss: 0.0001
Epoch [310/500], Loss: 0.0000
Epoch [320/500], Loss: 0.0000
Epoch [330/500], Loss: 0.0000
Epoch [340/500], Loss: 0.0000
Epoch [350/500], Loss: 0.0000
Epoch [360/500], Loss: 0.0000
Epoch [370/500], Loss: 0.0000
Epoch [380/500], Loss: 0.0000
Epoch [390/500], Loss: 0.0000
Epoch [400/500], Loss: 0.0000
Epoch [410/500], Loss: 0.0000
Epoch [420/500], Loss: 0.0000
Epoch [430/500], Loss: 0.0000
Epoch [440/500], Loss: 0.0000
Epoch [450/500], Loss: 0.0000
Epoch [460/500], Loss: 0.0000
Epoch [470/500], Loss: 0.0000
Epoch [480/500], Loss: 0.0000
Epoch [490/500], Loss: 0.0000
Epoch [500/500], Loss: 0.0000
```

Figure 28: Example of Loss Printed per 10 epoch

```
In [0]: #test
        x_ = torch.randn(50,1)
        y_ = f(x_)
        plt.scatter(x_.detach().numpy(), y_.detach().numpy(), label='true')

        y_pred = model(x_)
        plt.scatter(x_.detach().numpy(), y_pred.detach().numpy(), label='pred')

        plt.legend()
```

Figure 29: Code for Graphing True vs Predicted

7.2.2 Approximating the CRR Model Using a Neural Network

```
#Neural Network
a = 20
model = nn.Sequential(
    nn.Linear(1, a),
    nn.ReLU(),
    nn.Linear(a, a),
    nn.Linear(a, 1)
)
```

Figure 30: Number of Neurons and Layers

```
def CRRAmericanOption(type, n, T, S, K, r, sigma, q, tree):
  #type 'call' or 'put'
  # Variables and Initialization
  # n = Steps/height of bionomial tree
  # T = Time until maturity
  # S = Base price
  # K = Strike price
  # r = Interest
  # q = Dividend
  # sigma = volatility
  # tree = show option tree if True doesn't show if False

  dt = T/n #delta t for each step
  u = numpy.exp(sigma*numpy.sqrt(dt)) # Price multiplier if it goes up
  d = 1/u                             # Price multiplier if it does down
  p = (numpy.exp((r-q)*dt)-d)/(u-d)       # Formula for calculating probability for each price

  # Binomial tree
  # Constructing the tree
  binomial_tree = numpy.zeros([n+1, n+1])

  # Initializing the tree
  for i in range(n+1):
      for j in range(i+1):
        binomial_tree[j, i] = S*(d**j)*u**(i-j)


  # Exercise tree
  # Constructing the tree
  exercise_tree = numpy.zeros([n+1, n+1])

  # Initializing the tree
  for i in range(n+1):
      for j in range(i+1):
        exercise_tree[j, i] = K
```

Figure 31: Python Code for Option Model

```
# Options
# Option value
option = numpy.zeros([n+1, n+1])
# From wiki page call option value is Max [ (Sn - K), 0 ]
if type is 'call':
  option[:, n] = numpy.maximum(numpy.zeros(n+1), binomial_tree[:, n]-exercise_tree[:, n])

if type is 'put':
  option[:, n] = numpy.maximum(numpy.zeros(n+1), exercise_tree[:, n]-binomial_tree[:, n])

# Calculatig the price at t = 0
for i in numpy.arange(n-1, -1, -1):
   for j in numpy.arange(0, i+1):
      option[j, i] = numpy.exp(-r*dt)*(p*option[j, i+1]+(1-p)*option[j+1, i+1])

if type is 'call':
  option[:, n] = numpy.maximum(option[:, n], binomial_tree[:, n]-exercise_tree[:, n])

if type is 'put':
  option[:, n] = numpy.maximum(option[:, n], exercise_tree[:, n]-binomial_tree[:, n])


# Return value
if tree:
  print(numpy.matrix(option))
  print("Option Value: ")
  return option[0, 0]
else:
  print("Option Value: ")
  return option[0, 0]
```

Figure 32: Python Code for Option Model

```
# Train the model

num_epochs = 1000

for epoch in range(num_epochs):

    # Forward pass
    outputs = model(x_train)
    loss = criterion(outputs, y_train.float())

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch == 0 or (epoch+1) % 50 == 0:
        print ('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1,
                                                    num_epochs, loss.item()))
```

Figure 33: Python Code for Training Model

```
#training data
#training size per epoch
batch_size = 100
#creating a list
a = []
#generate randomize data
x_train = torch.randn(batch_size, 1)

#run loop on training to create output array
for i in range(batch_size):
  y = CRRAmericanOption('put', 10, 100, 200, x_train[i], 0.05, 0.11, 0.1, False)
  a.append(y)

#transform list to array
b = numpy.array(a)
#transform array into tensor set
y_train = torch.from_numpy(b)
```

Figure 34: Python Code for Training Data

7.2.3 Layers vs Neurons

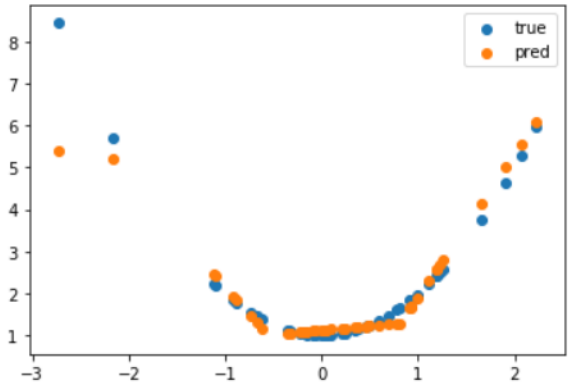


Figure 35: 2 layers and 3 neurons in each layer

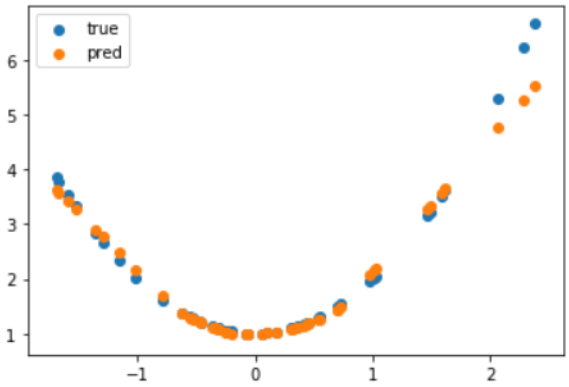


Figure 36: 2 layers and 9 neurons in each layer

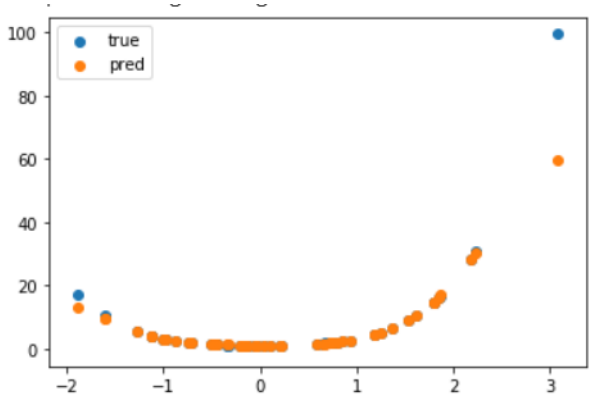


Figure 37: 2 layers and 15 neurons in each layer

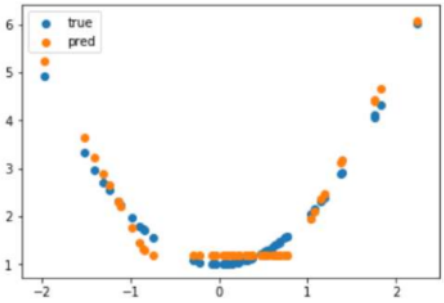


Figure 38: 4 layers with 3 neurons in each layer

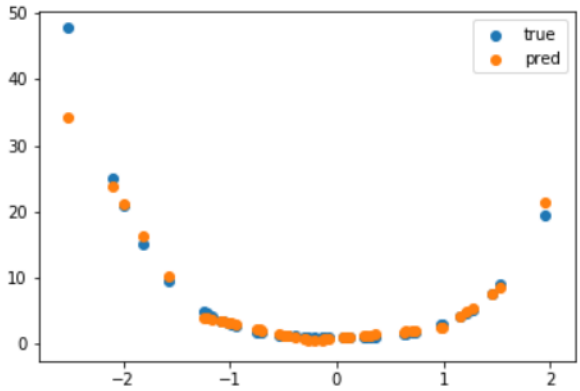


Figure 39: 5 layers and 3 neurons in each layer

7.2.4 Approximating the BSM using a Neural Network

```
import torch
import torch.nn as nn
import numpy as np
import torch.nn.functional as F
import matplotlib.pyplot as plt
import scipy.stats as ss
```

Figure 40: Import Statements for Code in Python

```
'''=========
option class init
==========='''
class VanillaOption:
    def __init__(
        self,
        otype = 1, # 1: 'call'
                   # -1: 'put'
        strike = 110.,
        maturity = 1.,
        market_price = 10.):
      self.otype = otype               # Put or Call
      self.strike = strike             # Strike K
      self.maturity = maturity         # Maturity T
      self.market_price = market_price #this will be used for calibration


    def payoff(self, s): #s: excercise price
      otype = self.otype
      k = self.strike
      maturity = self.maturity
      return np.max([0, (s - k)*otype])
```

Figure 41: Definition of the Vanilla Option Class

```
'''============
Gbm class
============'''

class Gbm:
    def __init__(self,
                 init_state = 100.,
                 drift_ratio = .0475,
                 vol_ratio = .2
                ):
        self.init_state = init_state
        self.drift_ratio = drift_ratio
        self.vol_ratio = vol_ratio
```

Figure 42: Definition of the Geometric Brownian Motion Class

```
'''========
Black-Scholes-Merton formula.
=========='''

def bsm_price(self, vanilla_option):
    s0 = self.init_state
    sigma = self.vol_ratio
    r = self.drift_ratio

    otype = vanilla_option.otype
    k = vanilla_option.strike
    maturity = vanilla_option.maturity

    d1 = 1/(sigma*np.sqrt(maturity))*(np.log(s0/k) + (r + np.power(sigma,2)/2)*(maturity))
    d2 = 1/(sigma*np.sqrt(maturity))*(np.log(s0/k) + (r - np.power(sigma,2)/2)*(maturity))
    return (otype * s0 * ss.norm.cdf(otype * d1) #line break needs parenthesis
            - otype * np.exp(-r * maturity) * k * ss.norm.cdf(otype * d2))

Gbm.bsm_price = bsm_price
```

Figure 43: Definition of the Black-Scholes-Merton Formula

```
'''=======
Get BSM prices given an option and a Tensor
======='''

def prices_bsm(self, vanilla_option, data):

  # Create the list
  a = []

  # Get tensor size
  sizeT = list(data.size())[0]

  for i in range(sizeT):
    self.init_state = data[i].item()
    callPrice = gbm1.bsm_price(vanilla_option)
    a.append(callPrice)

  # Convert to array then tensor
  arrayOut = np.array(a)
  outputData = torch.from_numpy(arrayOut)
  return outputData

Gbm.prices_bsm = prices_bsm
```

Figure 44: Definition of a Function to Calculate a Group of BSM Prices Given a Tensor

```
def f(s, k = 90):
  gbm = Gbm(init_state=s)
  option = VanillaOption(strike=k)
  return gbm.bsm_price(option)

batch_size = 63
x_list = np.linspace(0, 200, batch_size)
y_list = np.array([f(x) for x in x_list])
```

Figure 45: Definition of a function to Calculate BSM Price given a Single Underlying and Strike Price And Creation of a Random List of Strike Prices For Model Training

```
#model
#nn.Linear
H1 = 80; H2 = 20 #number of hidden layer
model = nn.Sequential(
    nn.Linear(1, H1),
    nn.Sigmoid(),
    nn.Linear(H1, H2),
    nn.Sigmoid(),
    nn.Linear(H2,2),
    nn.Sigmoid(),
    nn.Linear(2,1)
)
```

Figure 46: Definition of a Neural Network Model

```
#loss function
criterion = nn.MSELoss()
```

Figure 47: Definition of the Loss Function as Mean Squared Error

```
#optimizer
learning_rate = 0.1
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.7)
```

Figure 48: Definition of the Method of Optimization for the Network

```
batch_size = np.size(x_list)
x_train0 = torch.from_numpy(x_list).reshape(batch_size,1).float()
y_train0 = torch.from_numpy(y_list).reshape(batch_size,1).float()
```

Figure 49: Creation of Tensors of Training Data

```
# Normalization

def linear_transform(xx, l = 0, u= 1):
  M = torch.max(xx)
  m = torch.min(xx)
  return (u-l)/(M-m)*(xx-m)+l, m, M, l, u
x_train, x_m, x_M, x_l, x_u = linear_transform(x_train0, -1, 1)
y_train, y_m, y_M, y_l, y_u = linear_transform(y_train0, 0, 1)
```

Figure 50: Definition of the Linear Transform Function

```
# Train the model

num_epochs = 5000

for epoch in range(num_epochs):

    # Forward pass
    outputs = model(x_train.float())
    loss = criterion(outputs, y_train.float())

    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch == 0 or (epoch+1) % 50 == 0:
        print ('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1,
                                                    num_epochs, loss.item()))
        #print(outputs[1:10])
```

Figure 51: Loop for Training the Model

```
Epoch [4000/5000], Loss: 0.0004
Epoch [4050/5000], Loss: 0.0004
Epoch [4100/5000], Loss: 0.0004
Epoch [4150/5000], Loss: 0.0004
Epoch [4200/5000], Loss: 0.0004
Epoch [4250/5000], Loss: 0.0004
Epoch [4300/5000], Loss: 0.0004
Epoch [4350/5000], Loss: 0.0004
Epoch [4400/5000], Loss: 0.0004
Epoch [4450/5000], Loss: 0.0004
Epoch [4500/5000], Loss: 0.0003
Epoch [4550/5000], Loss: 0.0003
Epoch [4600/5000], Loss: 0.0003
Epoch [4650/5000], Loss: 0.0003
Epoch [4700/5000], Loss: 0.0003
Epoch [4750/5000], Loss: 0.0003
Epoch [4800/5000], Loss: 0.0003
Epoch [4850/5000], Loss: 0.0003
Epoch [4900/5000], Loss: 0.0003
Epoch [4950/5000], Loss: 0.0003
Epoch [5000/5000], Loss: 0.0003
```

Figure 52: Loss Per Epoch while Training the Model

```
def learnedfun(x):
  out = (1-(-1))/(x_M-x_m)*(x-x_m)+(-1.)
  out = model(out)
  out = (y_M- y_m)*out+y_m
  return out

y_pred = learnedfun(x_train0)
```

Figure 53: Function for Linearly Transforming Model Output

```
# Test with training data
plt.scatter(x_train0.detach().numpy(), y_train0.detach().numpy(), label='true')
plt.scatter(x_train0.detach().numpy(), y_pred.detach().numpy(), label='pred')

plt.legend()
plt.show()
```

Figure 54: Testing of the Model Versus the Training Data. See Figure 12b for an Example of Trained Data

```
# Test the Model with Random data

# Generate random data
x_0 = np.random.randint(0, 200, batch_size)

# Get BSM Prices as determined by the formula
y_0 = np.array([f(x) for x in x_0])

# Transform to tensor
x_0 = torch.from_numpy(x_0).reshape(batch_size,1).float()
y_0 = torch.from_numpy(y_0).reshape(batch_size,1).float()

# Normalize
x_, x_m, x_M, x_l, x_u = linear_transform(x_0)
y_, y_m, y_M, y_l, y_u = linear_transform(y_0)

# Plot x_ versus formula prices
plt.scatter(x_0.detach().numpy(), y_0.detach().numpy(), label='true')

# Get BSM Prices as determined by the model
y_pred = learnedfun(x_0)

# Plot x_ versus the model prices
plt.scatter(x_0.detach().numpy(), y_pred.detach().numpy(), label='pred')

plt.legend()
```

Figure 55: Testing of the Model Using Randomized Data. See Appendix 7.3 for Example Output

7.2.5 Setup-Optimizer

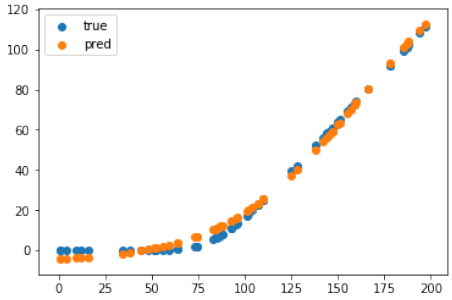
```
#Best set up for least loss
bestLoss = 100
bestNeurons = 0
bestLayers = 0
bestTimeForLoss = 1000000 #i dont know what Max_int equivalent is

#Best set up for least time within < 0.001
neuronsBestTime = -1
layersBestTime = -1
lossBestTime = 1000000 #same as the above

#temporary variables
tempTime = 0
tempLoss = 0

#main code/model
for i in range(2, maxNeuron+1): #Loop through each # of neurons
  for j in range(2,maxLayer+1): #Loop through each # of layers
    model = nn.Sequential(
      nn.Linear(1, i),
      nn.Sigmoid(),
      nn.Linear(i, i),
      #loop for the layers inside
      #need help fixing this**
      #for k in range(2, j):
       # nn.Linear(i, i),
      nn.Linear(i, 1)
    )

    #run the rest of the code with the current neurons and layers set up

    #start the timer
    start = time.time()

    # Train the model
    for epoch in range(10000):

      # Forward pass
      outputs = model(x_train)
      loss = criterion(outputs, y_train)

      # Backward and optimize
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
```

Figure 56: Setup-Optimizer Code

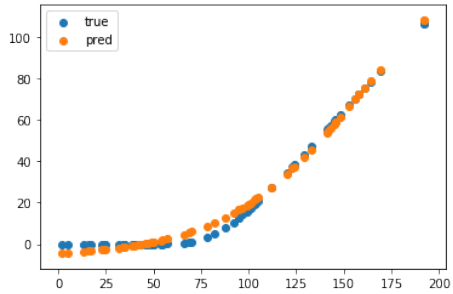
```
      if (epoch+1) % 10000 == 0:
        print ('Loss: ', loss.item())
    #end trainging loop

    #stop the timer
    end = time.time()

    #temp variables
    tempLoss = loss.item()
    tempTime = end - start

    #Get time and other statistics if the requirements are met
    #if statement for least loss
    if bestLoss > tempLoss :
      bestLoss = tempLoss
      bestNeurons = i
      bestLayers = j
      bestTimeForLoss = tempTime

    #if statement for best time when loss is < 0.001
    if tempLoss < lossThreshold and lossBestTime > tempTime:
      neuronsBestTime = i
      layersBestTime = j
      lossBestTime = tempTime
      LossLoss = tempLoss
#end of Test Runs

#print out saved statistic
print ('-------------------------------------------------')
print ('Setting for least loss')
print ('Least Loss: ', bestLoss)
print ('Neurons: ', bestNeurons)
print ('Layers: ', bestLayers)
print ('Time: ', bestTimeForLoss,'s')
print (' ')
print ('Setting for least time under 0.001 loss')
print ('Loss: ', LossLoss)
print ('Neurons: ', neuronsBestTime)
print ('Layers: ', layersBestTime)
print ('Time: ', lossBestTime, 's')
```

Figure 57: Setup-Optimizer Code

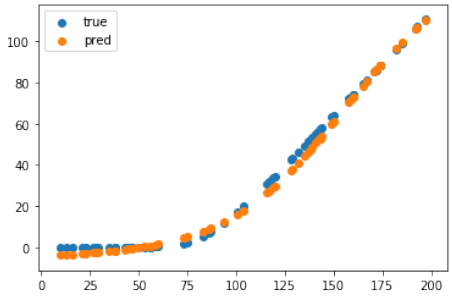
7.3 True vs Prediction Graphs



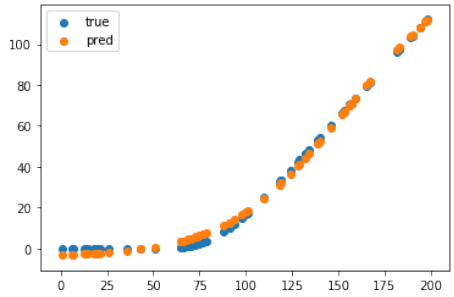
(a) Results with 2 Internal Layers, H1 = 50 and H2 = 30



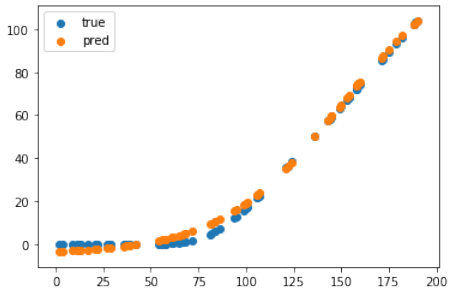
(b) Results with 2 Internal Layers, H1 = 50 and H2 = 50



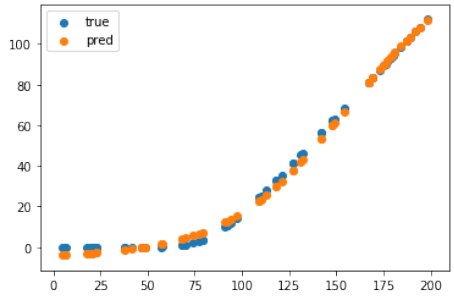
(c) Results with 2 Internal Layers, H1 = 50 and H2 = 100



(d) Results with 2 Internal Layers, H1 = 80 and H2 = 20

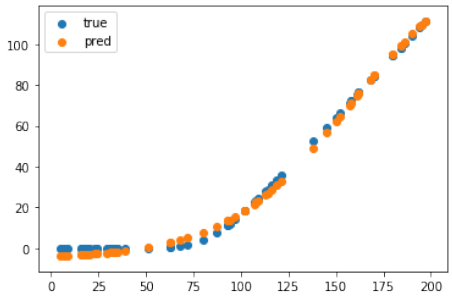


(e) Results with 2 Internal Layers, H1 = 100 and H2 = 80

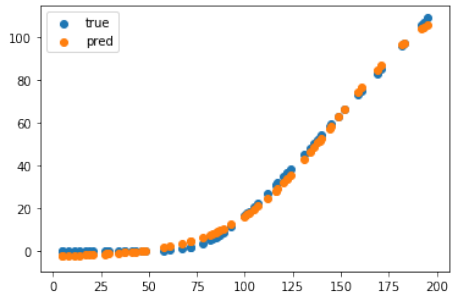


(f) Results with 2 Internal Layers, H1 = 100 and H2 = 100

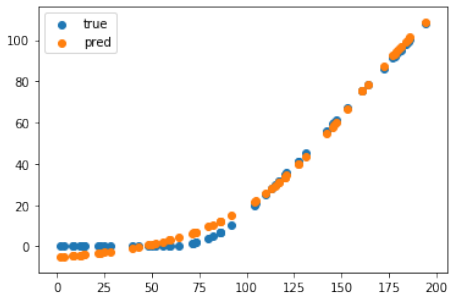
Figure 58: Black-Scholes Approximation Output Graphs With Two Layers



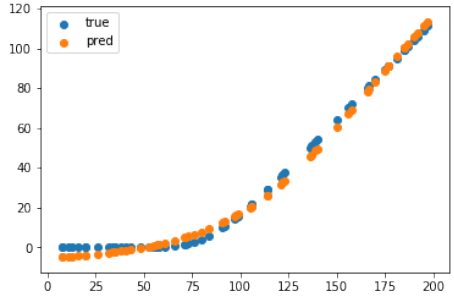
(a) Results with 2 Internal Layers, H1 = 100 and H2 = 150



(b) Results with 2 Internal Layers, H1 = 500 and H2 = 100



(c) Results with 3 Internal Layers, H1 = 50, H2 = 12, and H3 = 100



(d) Results with 3 Internal Layers, H1 = 50, H2 = 50, and H3 = 21

Figure 59: Black-Scholes Approximation Output Graphs With Three Layers

7.4 Loss Function

```
Epoch [1/1000], Loss: 0.0684
Epoch [50/1000], Loss: 0.0326
Epoch [100/1000], Loss: 0.0177
Epoch [150/1000], Loss: 0.0114
Epoch [200/1000], Loss: 0.0084
Epoch [250/1000], Loss: 0.0068
Epoch [300/1000], Loss: 0.0058
Epoch [350/1000], Loss: 0.0051
Epoch [400/1000], Loss: 0.0046
Epoch [450/1000], Loss: 0.0041
Epoch [500/1000], Loss: 0.0037
Epoch [550/1000], Loss: 0.0034
Epoch [600/1000], Loss: 0.0031
Epoch [650/1000], Loss: 0.0029
Epoch [700/1000], Loss: 0.0026
Epoch [750/1000], Loss: 0.0024
Epoch [800/1000], Loss: 0.0023
Epoch [850/1000], Loss: 0.0021
Epoch [900/1000], Loss: 0.0020
Epoch [950/1000], Loss: 0.0018
Epoch [1000/1000], Loss: 0.0017
```

Figure 60: Loss Function of European Put Option per 50 Epoch

```
Epoch [1/1000], Loss: 40141.1211
Epoch [50/1000], Loss: 0.3636
Epoch [100/1000], Loss: 0.1111
Epoch [150/1000], Loss: 0.0499
Epoch [200/1000], Loss: 0.0262
Epoch [250/1000], Loss: 0.0149
Epoch [300/1000], Loss: 0.0090
Epoch [350/1000], Loss: 0.0056
Epoch [400/1000], Loss: 0.0036
Epoch [450/1000], Loss: 0.0025
Epoch [500/1000], Loss: 0.0017
Epoch [550/1000], Loss: 0.0013
Epoch [600/1000], Loss: 0.0010
Epoch [650/1000], Loss: 0.0008
Epoch [700/1000], Loss: 0.0007
Epoch [750/1000], Loss: 0.0006
Epoch [800/1000], Loss: 0.0005
Epoch [850/1000], Loss: 0.0005
Epoch [900/1000], Loss: 0.0004
Epoch [950/1000], Loss: 0.0004
Epoch [1000/1000], Loss: 0.0004
```

Figure 61: Loss Function of European Call Option per 50 Epoch

```
Epoch [1/1000], Loss: 0.0487
Epoch [50/1000], Loss: 0.0187
Epoch [100/1000], Loss: 0.0077
Epoch [150/1000], Loss: 0.0036
Epoch [200/1000], Loss: 0.0021
Epoch [250/1000], Loss: 0.0014
Epoch [300/1000], Loss: 0.0011
Epoch [350/1000], Loss: 0.0010
Epoch [400/1000], Loss: 0.0009
Epoch [450/1000], Loss: 0.0008
Epoch [500/1000], Loss: 0.0008
Epoch [550/1000], Loss: 0.0007
Epoch [600/1000], Loss: 0.0007
Epoch [650/1000], Loss: 0.0007
Epoch [700/1000], Loss: 0.0006
Epoch [750/1000], Loss: 0.0006
Epoch [800/1000], Loss: 0.0006
Epoch [850/1000], Loss: 0.0006
Epoch [900/1000], Loss: 0.0006
Epoch [950/1000], Loss: 0.0005
Epoch [1000/1000], Loss: 0.0005
```

Figure 62: Loss Function of American Call Option per 50 Epoch

7.5 Implied Volatility Calculation

```python
def impliedvolatility(option, marketprice):
  def loss(v):
    option.vol = v
    return (option.bsm_price() - marketprice)**2
  return so.fmin(loss, option.vol, disp = 0)[0]
```

Figure 63: Implied Volatility Function

```python
#Setup Strike and Time to Maturity
T = np.array([1/12., 7/52., 11/52., 15/52.])
K = np.array([24., 29., 33., 35., 37.])
```

```python
#Create rectangular grid
T, K = np.meshgrid(T, K)
```

Figure 64: Implied Volatility Setup

```
#Extract different market prices for time to expiration and strike
market_price = np.array([[9.55, 9.78, 9.73, 9.75], [4.78, 4.90, 4.95, 5.125],
        [0.64, 1.72, 1.97, 2.215],[0.015, 0.74, 0.97, 1.21],[0.01, 0.25, 0.385, 0.585]])
```

Figure 65: Implied Volatility Market Price

```
#initiate vol matrix of zeros
vol = np.zeros([5,4])
```

Figure 66: Implied Volatility Matrix

```
#Explore market price array and get implied vol of each option
for i in range(5):
  for j in range(4):
    option3.K = K[i,j]
    option3.t = T[i,j]
    vol[i,j] = impliedvolatility(option3,market_price[i,j])
```

Figure 67: Implied Volatility Calculation

```
#Plot implied Vol Surface
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(K, T, vol, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_title('Bank of America Implied Volatility');
ax.set_xlabel('Strike')
ax.set_ylabel('Time')
ax.set_zlabel('Imp Vol');
```

Figure 68: Implied Volatility Surface Plot

References

[BS73] Fischer Black and Myron Scholes. "The Pricing of Options and Corporate Liabilities". In: *The Journal of Political Economy* 81.3 (1973), pp. 637–654.

[FPS03] Jean-Pierre Fouque, George Papanicolaou, and K. Ronnie Sircar. *Derivatives in Financial Markets with Stochastic Volatility*. Cambridge University Press, 2003, p. 201.

[Shr05] Steven Shreve. *Stochastic Calculus for Finance I: The Binomial Asset Pricing Model*. Springer Finance, 2005.

[Rup06] David Ruppert. *Statistics and Finance: An Introduction*. Springer, 2006.

[Shr10] Steven Shreve. *Stochastic Calculus for Finance II: Continuous Time Models*. Springer Finance, 2010.

[Lu+19] Lu Lu et al. "Dying relu and initialization: Theory and numerical examples". In: *arXiv preprint arXiv:1903.06733* (2019).

[PCa19] Adam Paszke, Soumith Chintala, and Edward Yang et al. *Pytorch Documentation*. 2019.