8-2020

Black-Scholes and Neural Networks

Gabriel Adams
Utah State University

Black-Scholes and Neural Networks

Plan B Thesis

Gabriel Adams

Supervisor: Dr. Tyler Brough

Committee Members: Dr. Danjue Shang, Dr. Jacob Gunther

May 2020

1 Introduction

In the dictionary, the word "option" is defined as "a thing that is or may be chosen." When considering financial options, it is vitally important to remember the "choice" aspect of them. Over time, the need for flexibility, security, and speculation has led to the development of a variety of financial options. Options, in this sense, are defined as "financial instruments that are derivatives based on the value of underlying securities such as stocks. An options contract offers the buyer the opportunity to buy or sell—depending on the type of contract they hold—the underlying asset." Options are a subcategory of a broader class of financial assets: derivatives.

Options provide individual investors and firms alike with valuable tools to manage risk and make concrete plans that involve the future. In a volatile and constantly changing world, options can provide some measure of certainty, although this certainty certainly comes at a cost. This cost, or premium, is worth it to many, however. For example, firms can plan out certain expenses into the future instead of being subject to the whims of weather, unforeseen events, or just mere fate. Willing counterparties can take on the opposite side of this agreement and can potentially experience profit. In addition to providing certainty, options can also be used for speculation.

Options clearly are valuable to many. How are they priced, however? Just like almost anything else, options are priced on the market according to supply and demand. Individuals and groups buying and selling all come together to determine the price of each option. All else remaining constant, if more investors want a specific option, the price of that option will rise. Likewise, if demand decreases and supply remains the same, the price will decrease. This is how options are priced in actuality, but is there a way to determine how options *should* be priced? Other financial assets have formulas that, given the correct inputs, will return a specific price. Vanilla bonds, for example, can be priced using simple cash flow calculations. There are a variety of methods for pricing equities. One such method is the capital asset pricing model (CAPM). Options, however, have traditionally been a bit more difficult to analytically price.

The Black-Scholes-Merton (BSM) model for pricing options was developed by Fischer Black and Myron Scholes. The model was introduced to the world in 1973 [1]. Somewhat later, Robert Merton further developed the mathematics and theory behind this model [2]. This model, though not entirely accurate when compared to actual prices in the markets, is considered a cornerstone of financial economics and is technically sound. It provides a sort of answer to how much options *should* be priced. It is important to recognize that there exists a discrepancy between actual option prices and the theoretical prices from the Black-Scholes model. Despite this, the model is a valuable tool in evaluating options.

In this project, we investigate using artificial neural networks to price options. Specifically, we will attempt to use neural networks to approximate Black-Scholes using different simulated datasets. We will show that neural networks perform relatively well when provided with extensive training data.

When provided with relatively sparse data, or with training data that is slightly different than the test data, neural networks will underperform to varying degrees. Finally, we will propose potential future research to remedy some of the limitations of neural networks, thus providing a happy medium between neural networks and pure analytical solutions such as Black-Scholes.

2 Options

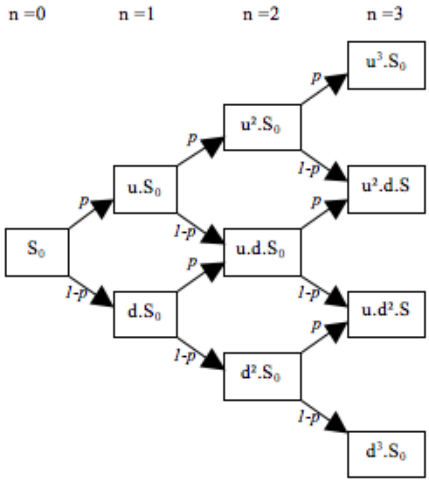
Options are an important financial asset in today's world. As mentioned in the introduction, financial options are assets that allow a holder to buy or sell an underlying asset at a specified strike price on a specified date or set of dates. There are two primary types of options. The first type of option is called a, well, call. Calls allow the buyer of an option to buy a specified asset (underlying) at a certain price (strike price) at specified time(s). The second type of option is a put. Puts allow the buyer to sell a specified asset (underlying) at a certain price (strike price) at specified time(s).

While puts and calls are relatively simple, they can be combined together, and with other assets, to create complex financial instruments designed for a variety of situations. Individuals and firms alike use options for hedging, insurance, and a variety of other purposes. Holbrook Working, in papers such as an article from 1953 [3], goes into more detail about the purposes of hedging. Options, and derivatives in general, provide flexibility in a variety of circumstances.

Within the realm of puts and calls, there are further subcategories. An important distinction is the difference between European and American style options. European options can only be exercised at the expiration date. That is, holders of European options can only "choose" on the expiration date. On the other hand, American options provide a bit more flexibility. They can be exercised at any moment up to and including the expiration date.

Various pricing models for options have been developed over the years. One such model that can be used to approximate the prices of both American and European options is the binomial pricing model. [4, 5]

Binomial Pricing Model



$$p = \frac{e^{rt/n} - d}{u - d}$$

$$u = e^{\sigma\sqrt{t/n}}$$

$$d = e^{-\sigma\sqrt{t/n}}$$

First, a tree of prices is constructed (from left to right in the diagram above). From an initial underlying price S_0, the tree branches up and down with probability p and $1 - p$ respectively. That is, p is the probability that the underlying price S_0 increases by a factor of u to uS_0 as time goes from $n = 0$ to $n = 1$. Likewise, $1 - p$ is the probability that the price of the underlying will decrease by

a factor of d. Formulas for p,u, and d can be found by the diagram above. Detailed explanations of their derivations can be found in other resources.

This branching process continues for n steps as shown above. On the far right of the tree, we can see the possible final states for the price of the underlying. Then, using this tree, we can backtrack and find the value of a specified option at each final node of the tree.

For a call option, this value is:

$$Max[(S_n - K), 0] \tag{1}$$

An investor will only exercise a call if the spot price is greater than the strike price. If the spot price is greater than the strike price, than the investor can exercise the call option, purchase the asset at K, sell at S_n, and make a profit of $S_n - K$, obviously ignoring transaction costs. The rational investor will not purposely lose money, so losses are, minus the premium paid for the option, are limited to 0.

An investor will exercise a put if the strike price is greater than the spot price. This can be expressed mathematically as:

$$Max[(K - S_n), 0] \tag{2}$$

After applying these to the final nodes of the tree, we can start to move backwards through the tree. This process is a bit more complicated and will be different for American, European, and Bermudan options. This process is continued until the first (leftmost) node of the tree is reached. The value of the option here is the price of the option.

Binomial trees are useful and flexible. Unlike Black-Scholes, they can be used to value American options. The value of early exercise can be quantified at the interior nodes of the tree. Binomial trees can also be used to price European options. Binomial trees, in a way, are discrete representations of the continous processed modeled by Black-Scholes.

Another option pricing model involves Monte Carlo simulation. Monte Carlo simulation can be used to price more complex options such as Asian options. Monte Carlo simulations are described in other resources.

3 The Black-Scholes Model

A third option pricing model is Black-Scholes. Black-Scholes provides an analytical solution to pricing European options. That is, given correct inputs, the Black-Scholes formula will give us an answer. As mentioned previously, Black-Scholes was developed by Fischer Black, Myron Scholes, and later Robert Merton. For their work, Scholes and Merton were awarded the Nobel Prize for Economics in 1997 [6]. Unfortunately, their colleague Black had already passed away and was thus unable to receive the prize.

The Black-Scholes formula for a call is as follows [7]. The price of a put can be priced using put-call parity or by a similar formula to the one below.

$$C_0 = N(d_1)S_t - N(d_2)Ke^{-rT} \tag{3}$$

$$d_1 = \frac{1}{\sigma\sqrt{T}}\left[ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T\right] \tag{4}$$

$$d_2 = \frac{1}{\sigma\sqrt{T}}\left[ln\left(\frac{S_t}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)T\right] \tag{5}$$

- C_0 is the price of the call option
- S_t is the stock price

- $N()$ is the CDF of the normal distribution
- T is the time to maturity (in years)
- K is the strike price
- r is the risk free rate
- σ is the volatility (standard deviation of log returns of underlying)

The value of a call option can be roughly divided into two parts: the amount the investor pays $(N(d_2)Ke^{-rT})$ and the amount the investor receives for selling the underlying $(N(d_1)S_t)$.

$N(d_2)Ke^{-rT}$ is the amount an investor pays when exercising a call option. The quantity Ke^{-rT} is the strike (or exercise) price K discounted back using the standard discounting factor e^{-rT}. r is the risk-free rate at which someone could invest funds over time T. r can be calculated using a variety of methods, including looking at US government bonds. This quantity is then multiplied by the $N(d_2)$, with d_2 defined above. This is the cumulative distribution function (CDF) of the normal distribution. Thus, $0 \leq N(d_2) \leq 1$. Essentially, the CDF returns the probability that a random variable is less than or equal to the input to the CDF. In rough terms, the larger d_1 and d_2 are, the more likely an investor is to exercise the option. As d_1 and d_2 increase, $N(d_1)$ and $N(d_2)$ also increase. This is because there is more area under the curve to the left of the input.

As S_t increases, the value of the option also increases. This makes sense because as S_t gets bigger, the difference between S_t and the strike price K become larger, thus leading to increased profits from exercising the option. Note that as S_t increases, the ratio $ln\left(\frac{S_t}{K}\right)$ also increases. This, in turn, will increase the inputs into the CDF.

Volatility (σ) is also important to examine. Higher volatility leads to higher option prices. This makes sense intuitively. Options provide some element of stability if an underlying asset is wildly volatile. Additionally, the upside potential of options is potentially great, while the downside potential is limited to just the premium spent on the option. Thus, increased volatility brings benefits but not many downsides. We can see this in the formulas above. σ can be found in the numerator and denominator of both d_1 and d_2 above. However, it is important to note that it is squared in the numerator and not in denominator. Thus, as σ increases, d_1 will also increase. As σ goes up, d_2 will decrease due to the negative sign in front of the σ^2 term. Then, we look at d_1 and d_2 together. d_1 will increase, leading the first term in the Black-Scholes equation to get bigger. d_2 will decrease, leading the second term in the Black-Scholes equation to get smaller. Thus, a bigger term will be subtracted by a smaller term, leading to a larger option price. Increased volatility leads to larger options prices.

There is much more complexity to the Black-Scholes formula. However, the previous paragraphs provide a basic intuitive understanding behind the formula. Given correct inputs, the formula will output a reasonable and technically sound price.

4 Neural Networks

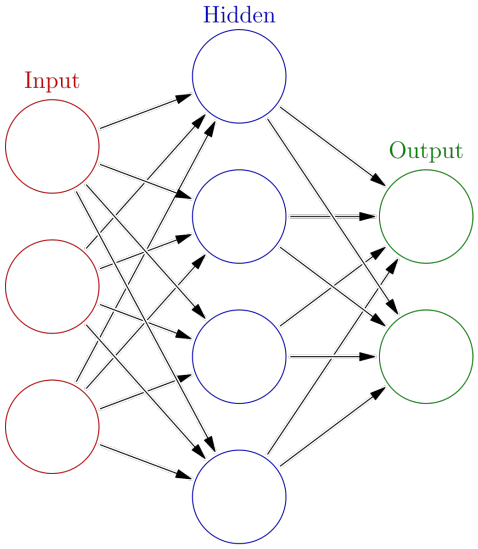
Machine learning has become increasingly in vogue over the past several years. As time goes by, computational abilities and speeds have increased more-than-linearly. Increased computational abilities have opened doors that were previously closed. Machine learning is one such door. Machine learning encompasses a variety of techniques that attempt to "learn" and predict from data.

In general, machine learning when compared to traditional econometric methods offers increased predictive ability at the cost of interpretability. A wide variety of machine learning techniques exist. Some are essentially more complicated versions of regression, while others are much more sophisticated. Neural networks are a type of machine learning that attempts to mimic the brain's processes. Simulated neurons pass information in a variety of ways in an attempt to glean understanding about underlying relationships. These neurons can be arranged in a variety of ways and layers. Training data is "fed" to the network, the network "learns," and then the network can be used to predict for

other data. Neural networks are traditionally quite well suited for prediction. In this case, we are interested in predicting options prices.

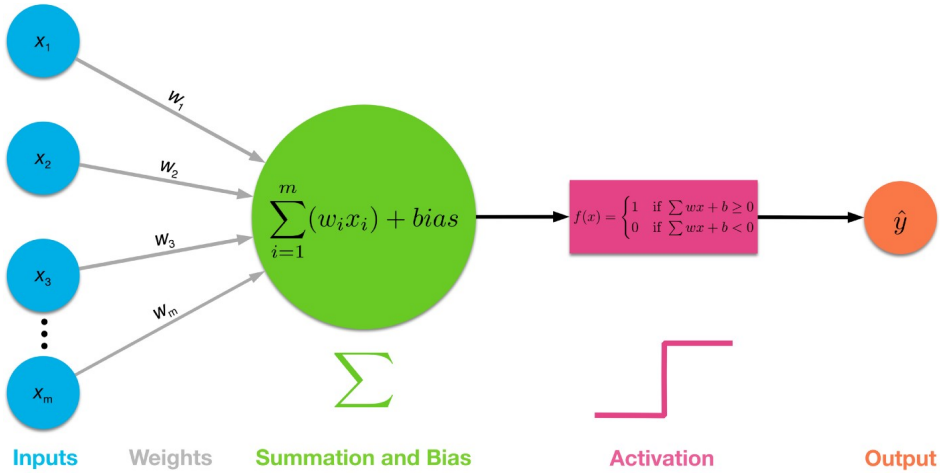
A simple neural network is shown in the figure below. Inputs are fed into the red nodes on the left. These inputs are then passed onto the hidden layer. From there, the inputs are weighted and passed through activation functions. A bias could also potentially be added. The results are then passed to the output layer, where a similar process takes place before the final result is calculated [8].

A Simple Neural Network



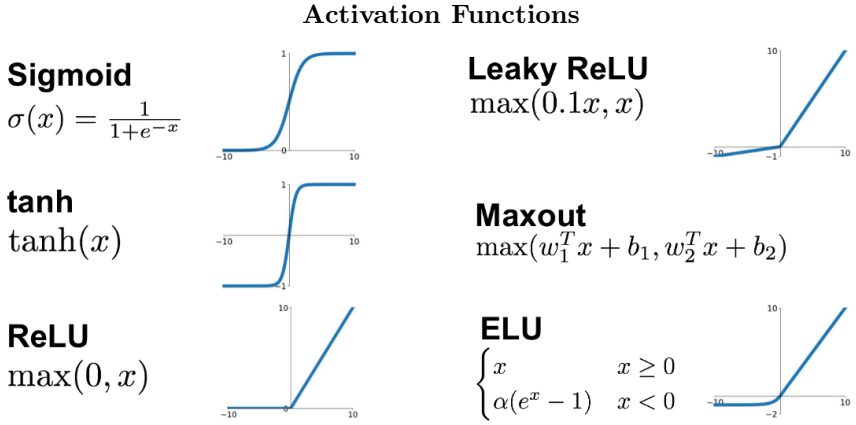
The following figure shows the process from inputs to output in more detail [9].

Another Representation



Activation functions help neural networks model potential non-linearities in the relationship between the input variables and the output. There are a wide variety of activation functions in use with neural networks. One common activation function is the sigmoid function, which is explored a bit in the next section. It maps inputs to an output between 0 and 1. Another common activation function, and the activation function used in this project, is the rectified linear unit function (or ReLU). This

function returns 0 for any inputs below 0, and returns the the input for any input greater than 0. These and other examples of activation functions are shown in the figure below [10].



Using a neural network to predict options prices has been done before. Research done before has shown that a neural network can, without any theoretical knowledge, approximate the relationship between the various parameters of Black-Scholes and the eventual price of the option. Instead of simply replicating this previous research, this project is exploring the relationships between the quality and quantity of the training data and the performance of the neural network. In other aspects, it is quite similar to previous research.

Neural networks are extremely powerful. In a later section, we will look at the Universal Approximation Theorem. According to this theorem, neural networks are capable of approximating essentially any continuous function, given a correctly formed structure. This is incredibly powerful. It provides us with the theoretical backing to approximate Black-Scholes with a neural network in this project.

Neural networks, once trained, can provide quick and accurate predictions. Neural networks are in use throughout industry. Image processing, natural language processing, and a variety of other fields use neural networks for prediction and classification. For example, Google uses deep neural networks to identify multiple languages on its Google Home devices.

Downsides of neural networks include the necessity of large computational resources and large amounts of training data. It can take quite some time for a neural network to train properly. Simpler models, including classical regression, can provide quicker results. In addition, neural networks are essentially black boxes. Thus, they make it difficult to quantify uncertainty and relationships between inputs and the output. Traditional regression techniques are much better at this. In finance, it may be necessary to explain to stakeholders why certain decisions were made. This explanation could potentially be much easier with traditional techniques, but more difficult with black box approaches such as neural networks.

5 Basic Mathematics of Neural Networks

Most practitioners today use software packages to train and use neural networks. This "black box" approach is quite helpful for quickly putting models into production. It is helpful, however, to delve a bit into the math behind the neural networks, as well as code a small example in Python to demonstrate core principles. This code sample is not included here, but some of the math is. This effort to better understand what is going on behind the scenes with neural networks is informed by a small selection of sources [11, 12, 13].

At their core, most machine learning algorithms are simply optimization problems. Neural networks are no different. Throughout the entire process of creating a neural network, the prime objective

will be to reduce the error from a specified loss function. A common loss function for use in neural networks is mean-squared error:

$$Loss(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2 \tag{6}$$

where y is the actual value, $\hat{y}$ is the value predicted by the neural network, and n is the number of observations.

I will now describe the mathematical structure of a simple neural network. We will imagine it has a input layer with two nodes, a hidden layer with three nodes, and an output node of a single node. In simple terms, our training data (each observation consisting of 2 values (1 for each explanatory variable) will be fed into the input layer. Each node in the hidden layer will then take the corresponding values from the input layer, multiply it by a weight w_{1i} and add a corresponding bias b_{1i}, where 1 represents the layer and $i \in \{1, 2\}$ represents the corresponding node. In each node, this value will be passed through an activation function. In our case, we will use a sigmoid activation function. The result from this calculation will be passed onto the next layer of nodes. Thus, the values passed onto the second layer are the following:

$$Value = \sigma(w_{1i}x + b_{1i}) = \frac{1}{1 + e^{-(w_{1i}x + b_{1i})}} \tag{7}$$

The second (or output) layer takes the values from the hidden layer and performs a similar process. The values are multiplied by weights, biases are added, and the results are again passed through an activation function. The results of these activation functions are then passed to the final (output) layer for final processing. Thus, the values passed from the second layer to the output layer are the following:

$$Value = \sigma(w_{2i}z + b_{2i}) = \frac{1}{1 + e^{-(w_{2i}z + b_{2i})}} \tag{8}$$

where $i \in \{1, 2, 3\}$ is the corresponding node and z is the results from the previous (input) layer. We can thus see there is a recursive nature to this. The final prediction, then, has a value of:

$$\hat{y} = \sigma(w_2(\sigma(w_1x + b_1) + b_2)) \tag{9}$$

This equation works for our simple example. For more complicated neural networks, this equation would be more involved, having additional recursive parts for each successive layer.

Now that we have a rough mathematical structure for the neural network, we explore how this model is "trained" and used in practice. Two processes are vital here: feedforward and backpropogation. Feedforward is when data is fed through the network to produce a prediction. Backpropogation then takes the error from this prediction (from the loss function) and uses it to change the weights and biases throughout the network in order to reduce the error.

Feedforward is the process by which data is fed through the network to produce a prediction. The input values are multiplied by the appropriate weights then are passed along to the next layer. Another round of multiplication takes place and then the values passed along to the final (output) layer. Here, a prediction is made.

Backpropogation uses this prediction and the resulting error (based on the loss function) to make changes to the weights throughout the network. This process involves some simple differential calculus. We first calculate the derivative of the loss function with respect to the weights and biases. The derivatives thus calculated will tell us how to update the weights and biases throughout the network using a method called gradient descent. Below we present the appropriate derivative for the weights, as well as split it up into individual parts that can be easily calculated. For the sake of simplicity, we will ignore the derivative for the bias, although it is similar.

This equation gives the partial derivative of the loss with respect to the weights. Note below how the derivative is split into three separate parts. Again, this simplifies calculations and helps with intuitive understanding.

$$\frac{\partial Loss(y,\hat{y})}{\partial w_i} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_i} \tag{10}$$

Here we look at the first part of the derivative. Using the power rule, we bring down the exponent.

$$\frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} = \frac{\partial}{\partial \hat{y}} \left(\frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2 \right) = \frac{2}{n} \sum_{i=1}^{n} (y - \hat{y}) \tag{11}$$

Here we look at the second part of the derivative. Interestingly, it simplifies to an expression that contains itself.

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial}{\partial z} \sigma(z) = \frac{\partial}{\partial z} \left(\frac{1}{1+e^{-z}} \right) \tag{12}$$

$$= \frac{1}{(1+e^{-z})} \frac{e^{-z}}{(1+e^{-z})} = \left(\frac{1}{1+e^{-z}} \right) \left(1 - \frac{1}{1+e^{-z}} \right) = \sigma(z)(1-\sigma(z)) \tag{13}$$

Here we look at the third part of the derivative. It is very simple to solve.

$$\frac{\partial z}{\partial w_i} = \frac{\partial}{\partial w_i} (w_i x + b) = x \tag{14}$$

Now, we combine all three parts of the derivative back together to get an overall answer.

$$\frac{\partial Loss(y,\hat{y})}{\partial w_i} = \frac{2}{n} \sum_{i=1}^{n} (y - \hat{y}) \sigma(z)(1-\sigma(z))(x) \tag{15}$$

The feedforward and backpropgation processes are repeated many times until the errors are sufficiently small. Batching can increase the efficiency of this process. Batching basically combines multiple observations together during the training process. In this project, we use a batch size of 64. We use 20 epochs. Each epoch is one "pass through" the training data.

Care must be taken to not overfit the model. There are a variety of techniques to help with this, some more complicated than others. In this project, we use a relatively simple method. We hold out validation data to ensure that the neural networks being trained are sufficiently generalizable. There are much more interesting techniques to prevent overfitting. Some involve taking out certain nodes in the network. These could be worth exploring at a future stage, but we do not discuss them further here.

After a neural network is trained, it can be used to make predictions for new data that comes in. If the neural network has been trained properly, these predictions should be reasonable accurate. If the neural network has been overfit or underfit, these predictions could be widely off.

This example, although relatively simple, provides a basic illustration of how neural networks work.

6 Solving Black-Scholes with a Neural Network

By their nature, neural networks are very flexible. Traditional regression models, while useful, come with a variety of assumptions and restrictions that limit their applicability [14]. Regression techniques can be modified to account for non-linearity, autocorrelation, collinearity, and a variety of other issues. These modifications, however, must be calculated by knowledgeable practitioners. Neural networks, on the other hand, are purely data-driven and do not have any similar restrictions [15]. Neural networks are relatively easy to develop and put in practice.

By the Universal Approximation Theorem, it has been shown that neural networks can essentially approximate any continuous function on compact subsets of real numbers, even with just a single hidden layer. Various versions of this theorem have been proven throughout the past few decades. Hornik, Stinchombe, and White showed this to be true in 1989 for multi-layer networks [16]. One issue with earlier proofs of the Universal Approximation Theorem is that the single hidden layer could potentially have a very large (if finite) amount of neurons. This could be a very real limitation on the usefulness of the theorem. In 2017, it was shown that a width-restricted neural network using Relu activation functions could also serve as a universal approximator [17]. Looking at these results together, it is clear that neural networks are incredibly powerful in solving a wide variety of functions.

Despite their numerable benefits, neural networks also come with some downsides. Interpretability is especially limited when compared to more traditional methods. Neural networks require a very large amount of data for training. Despite these downsides, neural networks are a very effective tool in a variety of fields.

Neural networks have been used to price options in the past. In 1993, Malliaris and Salchenberger used a neural network to estimate option prices [18]. In 2003, a comparison between neural networks and the traditional Black-Scholes model was conducted by Amilon [19]. In 2005, it was shown that a neural network was superior to the Black-Scholes model in most circumstances [20]. In 2018, researchers showed that using a neural network could improve computational speed when solving for the price of an option.

This project is not attempting to exactly replicate the previously mentioned papers. Instead, we will again show the ability of neural networks to approximate Black-Scholes. We will do this by generating a selection of artificial datasets directly from Black-Scholes. We will then train neural networks on a randomly selected subsets of these datasets. We will then show that the neural network can accurately predict the prices from the held out artificial data.

7 Data Collection and Simulation

We ran the entire data simulation, training, and testing processes on Google Colab. Google Colab provides a hosted resource where individuals can run Jupyter notebooks. For this project, we used a Python notebook. Python has been used for previous research in the space. In addition, Python provided some helpful packages with regard to neural networks, plotting, and using Black-Scholes. Packages used for this research project included:

- numpy
  - data manipulation capabilities
- pandas
  - data manipulation capabilities
- matplotlib
  - helps plot the validation loss of each neural network
- scipy
  - percentile functions for the gamma, beta, and uniform distributions
- keras
  - provides neural network framework using TensorFlow in the background
- sklearn

  - provides easy train/test splitting capability, parameter grid capabilities

- py_vollib
  - analytical Black-Scholes solutions

- progressbar
  - helpful progress bar when running data generation processes

We simulated data using a grid process. The parameters of Black-Scholes are the spot price (S), strike price (K), dividend rate (D), time (t), risk-free rate (R), and a measure of volatility (σ). We simulated these parameters under the following distributions (for the full and sparse datasets). For the extremes dataset, all of the distributions described here were used except for the spot price distribution. Note that the parameterizations noted are for the scipy package in python and are not standard parameterizations.

$$
\begin{gathered}
S \sim gamma(a = 100, scale = 1) \\
K \sim Unif(50, 200) \\
R \sim Unif(0.01, 0.18) \\
D \sim Unif(0.01, 0.18) \\
t \in \{0.25, 0.5, 0.75, 1\} \\
\sigma \sim Beta(a = 2, b = 5) + 0.001
\end{gathered}
$$

Spot prices were simulated using a gamma distribution roughly centered at 100. Strike prices were distributed uniformly from 50 to 200. Both the risk-free and dividend rates had the same uniform distribution form 0.01 to 0.18. Time-to-maturity took only four values (each value expressed in years ($0.25 = 3$ months). Volatility was simulated with a beta distribution with 0.001 added on. We added 0.001 to the simulated σ values in order to avoid having "zero" variance, an impossibility.

For the purposes of this project, we simulated three separate datasets. The first, called "full," was a "as complete as possible" simulated dataset with almost a million observations. Due to constraints on computational resources, this was as a big a dataset we could simulate under reasonable circumstances. The second dataset was a "sparse" version of the full dataset. This set covered the same ranges for each of the parameters, but simply had fewer observations for each. This dataset ended up having 12,500 observations. The third and final dataset, named "extremes," was similar to the full dataset. The only difference was that the spot prices, instead of being generated from the gamma distribution mentioned above, were instead distributed uniformly from 90 to 110. The purpose of this dataset is to test whether the neural network can generalize from this set of limited data to more "extreme" situations. This set also had almost a million observations. These datasets are briefly summarized in the table below.

Dataset	**# Increments**	**Total # of Observatons**
Full	12	995328
Sparse	5	12500
Extremes	12	995328

The increments are taken at evenly spaced percentiles of the corresponding distributions, whether they be beta, gamma, or uniform. Again, the time to maturity is selected from one of four options.

The data generation process is quite time intensive with available computing resources. To minimize having to run the data generation process, each dataset is exported to a corresponding comma-separated values (csv) file for easy retrieval. These csv files are used in the next step where the neural networks are actually trained.

8 Training the Neural Networks

Using the datasets described in the previous section, we train three corresponding neural networks. The keras framework provided in Python provides a relatively simple way to accomplish this.

Each neural network consists of an input layer, two hidden layers, and an output layer. The input layer has six nodes (one for each input parameter). The hidden layers have ten nodes each. The output layer has one node (for the predicted price). The activation function used in throughout is the rectified linear unit, which is shown in a previous figure. Prior to training, 10% of each dataset is randomly selected and set aside to test the neural networks after training. The remaining 90% goes on the the training process. Of the remaining 90%, 80% is actively used in the training process. The other 20% is used for validation. This is described in more detail in the results section. Finally, the batch size is set to 64 and the number of epochs is set to 20. This means that the training will essentially averaged over groups of 64 observations and that training will go over the entire dataset 20 times before finishing.

9 Results

As we train the neural networks, we can look at how the loss is affected as the neural network goes through the training process. We can look at the training error, which is the error of the training data. This can be interesting to look at, it is not really an accurate representation of how well the neural network is doing. Instead, we can look at the validation error. This is how well the network is performing on a set of data that is not included in the training data. In our case, we used an 80/20 split (after taking the test set out). That is, after the test set is taken out, the remaining data is split again. 80% is used to train the network, while the other 20% is used for validation.

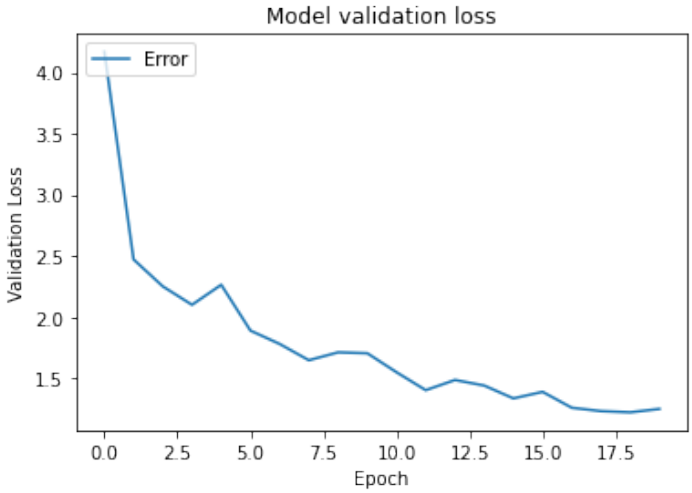
In the table below, we can see how each of the three neural networks performed while training over 20 epochs. To reemphasize, the numbers in the table are values of validation error, not training error.

Epoch	Full	Sparse	Extremes
1	4.1698	434.85	20.862
2	2.4709	310.99	7.1893
3	2.2505	147.21	5.1229
4	2.0998	57.971	5.6529
5	2.2636	25.569	5.032
6	1.8889	17.967	4.8148
7	1.7795	16.233	4.7485
8	1.6462	16.474	4.559
9	1.7109	15.396	5.0435
10	1.7035	15.197	4.5602
11	1.5482	15.056	4.6663
12	1.4014	14.88	4.2512
13	1.4847	14.838	4.4657
14	1.4395	14.579	3.8682
15	1.3341	14.575	3.913
16	1.3877	14.286	4.1381
17	1.2574	14.121	4.4358
18	1.23	14.085	4.2378
19	1.2203	13.911	3.7568
20	1.2478	14.004	3.6434

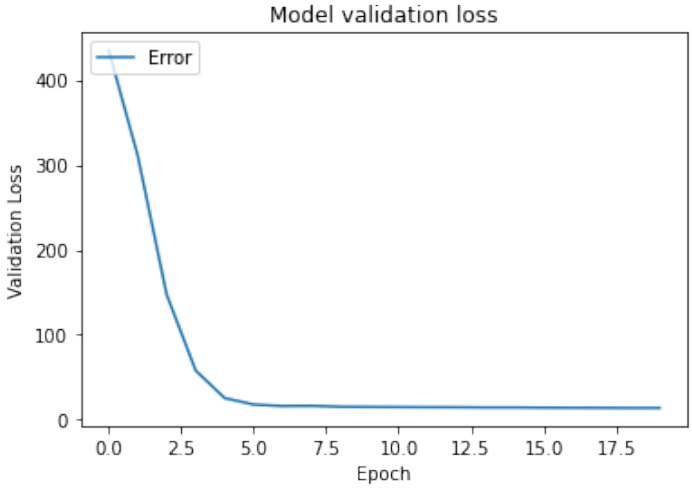
The plots below help us visualize the results included in the table above. Each plot shows how

validation error decreases as the neural network is trained. When looking at the plots, it is useful to pay attention to the units on the vertical axes.

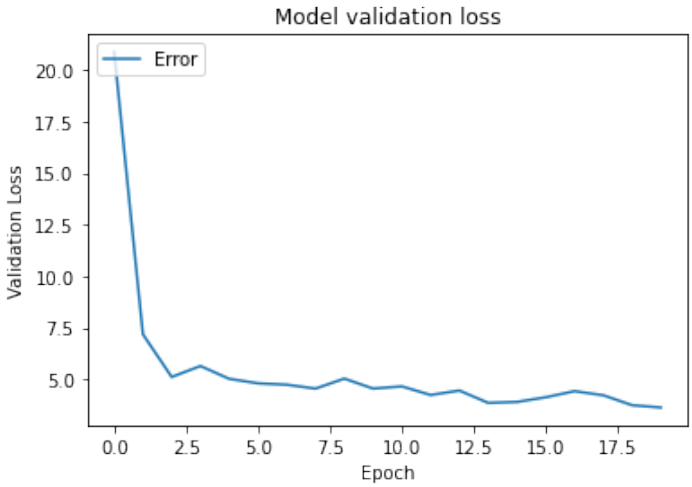
Full



Sparse



Extremes



In all three cases, the validation error generally decreases as epochs increase (as training moves along). While the curves in the plots look similar, the vertical axes tell a different story. The neural network training on the full dataset quickly converges to a validation error under 2. On the other hand, the sparse neural network has a much larger validation error over its life, ending above 13.5. Finally, the extremes dataset performs somewhere between the other two, ending with a validation error of just above 3.5.

It is clear that having sparse data is very detrimental to proper neural network training. Despite having the same "even" coverage of the input parameters as the full dataset, the sparse neural network is unable to generalize and perform as well as the full neural network. Interestingly, while the extremes neural network underperforms the full network, it stills performs somewhat adequately, especially when compared to the sparse network.

We now look at the performance of each network on the same test set. These results are included in the table below.

Full	**Sparse**	**Extremes**
1.24215	10.324	2.3341

The results from the validation error analysis above are essentially repeated. We can see that the network performed on the full dataset performs very well. It has an error of 1.24215. The extremes network comes in second place with an error of 2.3341, while the sparse network comes in last with an error of 10.324.

10 Conclusions

From the results in the previous section, we can draw some interesting conclusions. Neural networks perform best when the training data is relatively complete. The neural network trained on the full data clearly performed the best as measured by both validation errors and the eventual test error.

Neural networks perform relatively poorly when training data is relatively concentrated (at least on one input). This is shown by the results for the extremes dataset. Here, the validation errors and testing error were greater than the corresponding full errors, but were smaller than the corresponding errors for the sparse network.

Neural networks perform very poorly when training data is sparse. With sparse data, neural networks have trouble generalizing and making accurate predictions with other inputs.

When using neural networks, it is apparent that collecting a complete dataset for training is vitally important. With incomplete, sparse, or otherwise inadequate data, the performance of the neural network will suffer. Reliable results require thorough inputs.

If improperly trained neural networks are used in production, it is easy to imagine the potential damage that could occur. Large prediction errors could be a result. Collecting data can be a costly and difficult process, but it is often worth it. Inadequate data collection could render an otherwise solidly-constructed neural network useless.

11 Theory-Informed Machine Learning

A more sophisticated version of a neural network has emerged in recent years. This type of neural network incorporates the same things as a naïve neural network but adds a few additional complexities that can contribute to relatively superior performance. Traditional neural networks use the traditional loss function of mean-squared error. While this is suitable in many situations, it also has its limitations. Consider the effect of one single outlier. A large outlier's errors is squared. Large errors are penalized more heavily than smaller errors. This can lead to inaccuracies and suboptimal predictions. Outliers certainly occur in finance, especially during black swan events. One such event is currently going on

throughout the United States, and the world in general. Coronavirus is wreaking havoc on the world and pushing financial markets to their limits. Risk and uncertainty are rampant everywhere. As such, prices are extremely volatile, and no one really knows what assets should be priced. Traditional methods probably have trouble keeping up. An enhanced neural network could provide some valuable insight into what prices should be.

In addition to handling outliers better, an improved neural network can provide a few other benefits. In situations where there is biased or insufficient data, an improved network can still make valuable insights into prices.

An improved neural network will keep the same skeleton as a naïve neural network but will add a theory-informed part to the loss function [21]. The mean-squared error part will remain but will be augmented with a version of the BSM. Why not just use BSM directly? Because, again, while it is theoretically sound, actual prices are usually a bit different. By using an enhanced neural network, we can take an interesting approximate Bayesian approach to pricing. In basic terms, we can vary how much we rely on actual pricing data and how much we rely on theory such as BSM. For example, we could rely entirely on just the data by disregarding completely the error term based on BSM. On the other hand, we can focus more on BSM and less on the data by weighting the BSM error term more. This relative weighting would allow us to investigate some interesting research questions.

We do not investigate this possibility further, but it is an interesting option for future research.

12

References

[1] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[2] Robert C. Merton. Theory of rational option pricing. *The Bell Journal of Economics and Management Science*, 4(1):141–183, 1973.

[3] Holbrook Working. Hedging reconsidered. *Journal of Farm Economics*, 35(4):544–561, 1953.

[4] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3):229 – 263, 1979.

[5] Binomial options pricing model, Mar 2020.

[6] The sveriges riksbank prize in economic sciences in memory of alfred nobel 1997.

[7] Introduction to the black-scholes formula (video).

[8] A super simple introduction to neural networks.

[9] Nahua Kang. Multi-layer neural networks with sigmoid function- deep learning for rookies (2), Feb 2019.

[10] Pawan Jain. Complete guide of activation functions, May 2020.

[11] Arunava. Derivative of the sigmoid function, Jul 2018.

[12] backpropagation calculus — deep learning, chapter $4_2017, 2017$.

[13] James Loy. How to build your own neural network from scratch in python, Mar 2020.

[14] Michael A. Poole and Patrick N. O'Farrell. The assumptions of the linear regression model. *Transactions of the Institute of British Geographers*, (52):145–158, 1971.

[15] Vasilescu Dumitru, Ciobanu; Maria. Advantages and disadvantages of using neural networks for predictions. *Ovidius University Annals, Series Economic Sciences*, 13(1):444–449, 2013.

[16] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[17] Boris Hanin and Mark Sellke. Approximating continuous functions by relu nets of minimal width, 2017.

[18] Mary Malliaris and Linda Salchenberger. A neural network model for estimating option prices. *Applied Intelligence*, 3(3):193–206, Sep 1993.

[19] Henrik Amilon. A neural network versus black–scholes: a comparison of pricing and hedging performances. *Journal of Forecasting*, 22(4):317–335, 2003.

[20] Julia Bennell and Charles Sutcliffe. Black–scholes versus artificial neural networks in pricing ftse 100 options. *Intelligent Systems in Accounting, Finance and Management*, 12(4):243–260, 2004.

[21] M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686 – 707, 2019.

[22] Jahnavi Mahanta. Introduction to neural networks, advantages and applications, Jul 2017.

[23] Balázs Csanád Csáji. Approximation with artificial neural networks, 2001.

13 Code

```
# -*- coding: utf-8 -*-
"""Untitled2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/19rdt-0nU4pWYb4xYXNNhAKRcNCM2P_qp
"""

# Commented out IPython magic to ensure Python compatibility.
# %%capture
#
# import numpy as np
# import pandas as pd
# import matplotlib.pyplot as plt
# import scipy.io as sio
# import keras
# import keras.backend as K
#
# !pip install py_vollib  # not included in Google Colab, have to install for each instance
#
# from sklearn.model_selection import ParameterGrid
# from py_vollib import black_scholes_merton as bsm
# from progressbar import ProgressBar
# from scipy.stats import gamma
# from scipy.stats import beta
# from scipy.stats import uniform
# from keras.models import Model
# from keras.layers import Input, Dense
# from sklearn.model_selection import train_test_split

# S (spot price)
# gamma
def thisS (q):
  return gamma.ppf(q, a = 100, scale = 1)

# K (strike price)
# uniform (lower = 50, upper = 200)
def thisK (q):
  return uniform.ppf(q, 50, 200)

# (interest rate)
# uniform (lower = 0.01, upper = 0.18)
def thisR (q):
  return uniform.ppf(q, 0.01, 0.18)

# D (dividend)
# uniform (lower = 0.01, upper = 0.18)
def thisD (q):
  return uniform.ppf(q, 0.01, 0.18)
```

```
# t (time-to-maturity)
# t will be 3, 6, 9, 12 months for all examples (0.25, 0.5, 0.75, 1 year)

# sigma (volatility)
# beta (add small amount so volatility cannot be zero)
def thisSigma (q):
  return (beta.ppf(q, a = 2, b = 5) + 0.001)

num_increment = 12
percentiles = pd.Series(np.linspace(0, 0.99, num_increment))

S = percentiles.apply(thisS)
K = percentiles.apply(thisK)
q = percentiles.apply(thisD)
t = np.array([.25, .5, .75, 1])
r = percentiles.apply(thisR)
sigma = percentiles.apply(thisSigma)

param_grid = {'S': S, 'K' : K, 'q' : q, 't' : t, 'r' : r, 'sigma' : sigma}
grid = ParameterGrid(param_grid)

pbar = ProgressBar()
fullDF = pd.DataFrame()
prices = []
for params in pbar(grid):
    prices.append(bsm.black_scholes_merton(flag = 'p', S = params['S'], K = params['K'],\
    q = params['q'], t = params['t'],r = params['r'], sigma = params['sigma']))
    fullDF = fullDF.append(pd.Series(params), ignore_index = True)

# swap price to first column
fullDF['price'] = prices

# output to csv
fullDF.to_csv('dataFull.csv', index = False)
print(fullDF.head())
print(fullDF.tail())

num_increment = 5
percentiles = pd.Series(np.linspace(0, 0.99, num_increment))

S = percentiles.apply(thisS)
K = percentiles.apply(thisK)
q = percentiles.apply(thisD)
t = np.array([.25, .5, .75, 1])
r = percentiles.apply(thisR)
sigma = percentiles.apply(thisSigma)

param_grid = {'S': S, 'K' : K, 'q' : q, 't' : t, 'r' : r, 'sigma' : sigma}
grid = ParameterGrid(param_grid)
```

```
pbar = ProgressBar()
sparseDF = pd.DataFrame()
prices = []
for params in pbar(grid):
    prices.append(bsm.black_scholes_merton(flag = 'p', S = params['S'], K = params['K'],\
    q = params['q'], t = params['t'],r = params['r'], sigma = params['sigma']))
    sparseDF = sparseDF.append(pd.Series(params), ignore_index = True)

# swap price to first column
sparseDF['price'] = prices

# output to csv
sparseDF.to_csv('dataSparse.csv', index = False)
print(sparseDF.head())
print(sparseDF.tail())

def this_extremes_S (q):
  return uniform.ppf(q, 90, 110)
S = percentiles.apply(this_extremes_S)

K = percentiles.apply(thisK)
q = percentiles.apply(thisD)
t = np.array([.25, .5, .75, 1])
r = percentiles.apply(thisR)
sigma = percentiles.apply(thisSigma)

param_grid = {'S': S, 'K' : K, 'q' : q, 't' : t, 'r' : r, 'sigma' : sigma}
grid = ParameterGrid(param_grid)

pbar = ProgressBar()
extremesDF = pd.DataFrame()
prices = []
for params in pbar(grid):
    prices.append(bsm.black_scholes_merton(flag = 'p', S = params['S'], K = params['K'],\
    q = params['q'], t = params['t'],r = params['r'], sigma = params['sigma']))
    extremesDF = extremesDF.append(pd.Series(params), ignore_index = True)

# swap price to first column
extremesDF['price'] = prices

# output to csv
extremesDF.to_csv('dataExtremes.csv', index = False)
print(extremesDF.head())
print(extremesDF.tail())

# testing neural network (full data)

fullDF = pd.read_csv("dataFullMain.csv")

def baseline_model():
  # create model
```

```
  i = Input(shape=(6,))
  x = Dense(10, activation='relu')(i)
  y = Dense(10, activation='relu')(x)
  o = Dense(1)(y)
  model = Model(i, o)
  model.compile(loss="mse", optimizer= "adam")
  return model

model_full = baseline_model()
X = fullDF[['S','K','q','r','sigma','t']]
y = fullDF[['price']]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state = 7)
history_full = model_full.fit(X_train, y_train, batch_size = 64, epochs = 20, \
verbose = 2, validation_split=0.2)  # set batch size to 1, otherwise there are errors when trying to

plt.plot(history_full.history['val_loss'])
plt.title('Model validation loss')
plt.ylabel('Validation Loss')
plt.xlabel('Epoch')
plt.legend(['Error', 'Test'], loc='upper left')
plt.show()
X_test_full = X_test
y_test_full = y_test
model_full.evaluate(x=X_test, y=y_test)

# testing neural network (sparse data)

sparseDF = pd.read_csv("dataSparseMain.csv")

def baseline_model():
  # create model
  i = Input(shape=(6,))
  x = Dense(10, activation='relu')(i)
  y = Dense(10, activation='relu')(x)
  o = Dense(1)(y)
  model = Model(i, o)
  model.compile(loss="mse", optimizer= "adam")
  return model

model_sparse = baseline_model()
X = sparseDF[['S','K','q','r','sigma','t']]
y = sparseDF[['price']]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state = 7)
history_sparse = model_sparse.fit(X_train, y_train, batch_size = 64, epochs = 20, \
verbose = 2, validation_split=0.2)  # set batch size to 1, otherwise there are errors when trying to

plt.plot(history_sparse.history['val_loss'])
plt.title('Model validation loss')
plt.ylabel('Validation Loss')
```

```
plt.xlabel('Epoch')
plt.legend(['Error', 'Test'], loc='upper left')
plt.show()

model_sparse.evaluate(x=X_test_full, y=y_test_full)

# testing neural network (extremes data)

extremesDF = pd.read_csv("dataExtremesMain.csv")

def baseline_model():
  # create model
  i = Input(shape=(6,))
  x = Dense(10, activation='relu')(i)
  y = Dense(10, activation='relu')(x)
  o = Dense(1)(y)
  model = Model(i, o)
  model.compile(loss="mse", optimizer= "adam")
  return model

model_extremes = baseline_model()
X = extremesDF[['S','K','q','r','sigma','t']]
y = extremesDF[['price']]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state = 7)
history_extremes = model_extremes.fit(X_train, y_train, batch_size = 64, epochs = 20, \
verbose = 2, validation_split=0.2)  # set batch size to 1, otherwise there are errors when trying to

plt.plot(history_extremes.history['val_loss'])
plt.title('Model validation loss')
plt.ylabel('Validation Loss')
plt.xlabel('Epoch')
plt.legend(['Error', 'Test'], loc='upper left')
plt.show()

model_extremes.evaluate(x=X_test_full, y=y_test_full)

tableOutput = pd.DataFrame({'Full':history_full.history['val_loss'], \
'Sparse':history_sparse.history['val_loss'], \
'Extremes':history_extremes.history['val_loss']}, columns=['Full', 'Sparse', 'Extremes'])
tableOutput.to_csv("tableResultsValidaton.csv")

print(len(fullDF.index))
print(len(sparseDF.index))
print(len(extremesDF.index))
```