Master of Applied Mathematics Program
Professional Master's Project Report
Department of Applied Mathematics
Illinois Institute of Technology

Modeling BTC Options and BTC Prices Using Stochastic Volatility, LSTM Neural Networks, and GRU Neural Networks

Submitted by
Farukh Sarkulov

In partial fulfillment of the requirements of Master's candidates in Master of Applied Mathematics Program

December 2021

Contents

1 Introduction

Cryptocurrencies and Bitcoin (BTC) are a hot topic currently because of various reasons. To some, cryptocurrencies are seen as the future of money as they are a form of decentralized currency whose value is fixed by complex mathematical algorithms that are part of the ”blockchain”. People can attain BTC by ”mining” which is a process in which computers run one algorithm continuously until a ”coin” is released from the blockchain. That coin and the effort that went into mining it is noted in the proof of work, which is essentially how all BTC are verified. Every coin has a unique documentation that shows each step taken for that coin to have been mined and claimed. Theoretically, there is a limit to how many BTCs can be mined which supposedly provides inherent value to each coin because they are finite resources. For others, BTC and cryptocurrencies are viewed as a purely speculative asset that is highly volatile and useful for anonymous transactions and money laundering. Most BTC transactions are purely speculative, that is, buying and selling thereby influencing market price [1]. Aside from the legal and economic aspects of cryptocurrencies, the largest issue with cryptocurrencies is the energy consumption associated with mining cryptocurrencies and conducting the ”proof of work” algorithms. Mining requires hundreds of computers running a set list of algorithms continuously in the hopes of attaining a coin. The proof of work method is computationally expensive and this is evident in the energy consumption of cryptocurrencies globally [3]. In recent years, BTC options trading has become popular and is only expanding. BTC options became widely accepted in 2017, when the Chicago Mercantile Exchange (CME) announced their plans to create BTC futures. BTC options operate the same as normal options but are more expensive due to the implied volatility inherent in BTC. Because BTC is so volatile, its implied volatility is very high which causes the BTC option prices to be very high [2].

The goal of this project is to accurately predict Bitcoin (BTC) option prices and closing prices using a combination of Black-Scholes Merton formulas, Neural Network Modeling, and other stochastic processes. First I will analyze the historical data for Bitcoin to determine if the data is mean reverting, how much volatility is in the market, and to derive important statistical values such as mean, standard deviation, skewness, and excess kurtosis. These derived parameters will be used in various ways to model the data in hopes of accurately predicting BTC options and BTC closing prices. The mean, variance, and volatility will be needed to apply the Black Scholes Merton and Heston Options Pricing Methods [4]. A stochastic volatility process will be applied to derive simulated daily returns using Markov Chain Monte Carlo (MCMC) sampling and the results will be compared to the true returns [12].

In part 1 of this project I will conduct a historical data analysis and use various statistical tests to analyze the data and to determine what model might be best for predicting option prices. I will obtain the data from Yahoo Finance using the R package *quantmod* and plot the overall historical data as a time series. We will also derive the daily returns and plot those to get a sense of if there is some mean reversion. To test how much mean reversion there is in our data, I will employ the Augmented Dickey Fuller (ADF) Test and the Hurst Test. I will also solve for mean reversion if the results of the ADF and Hurst Tests imply some stationarity in the returns. Figure 1 is the historical time series data for the BTC Closing price from 2018 to 2021. As we can see in the historical time series graph for BTC (Figure 1), there is an enormous jump in BTC price starting just after July 1, 2020. I anticipate that this jump in BTC price will make it near impossible to get an accurate prediction for models trained on the initial part of the data (2018-2020). However, I hope that the model can accurately predict the general trend in the BTC price.

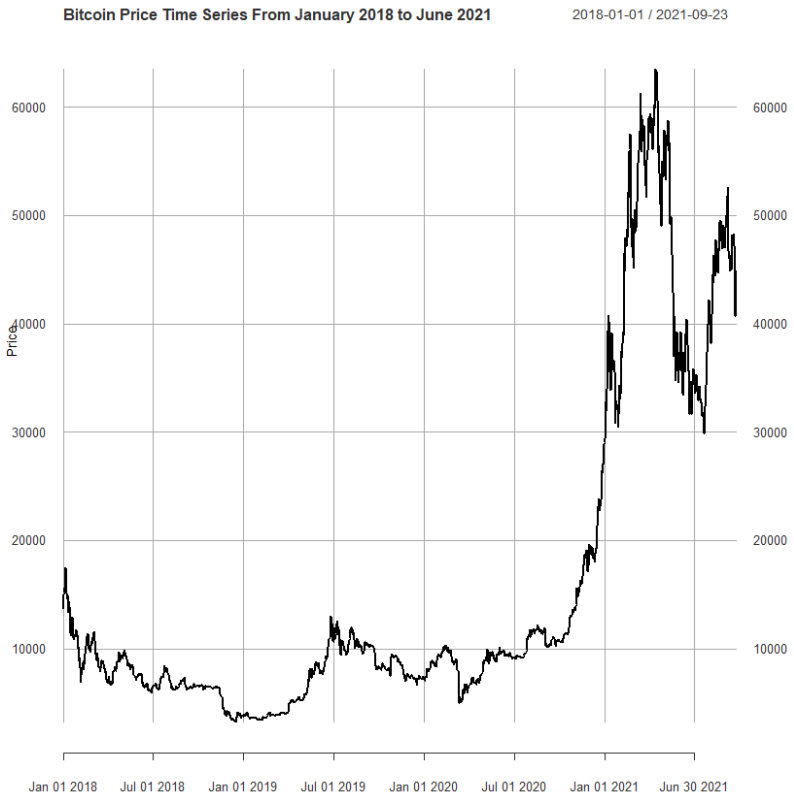

Figure 1: Historical BTC Closing Price January 1 2018 to September 23 2021

In part 2 of this project I will use stochastic volatility sampling to derive parameters from the BTC data. The derived parameters will be needed to implement the Heston Options Pricing Model, which will be used to improve on the Black Scholes Merton (BSM) Options Pricing results that were part of an earlier version of this project [6]. The discussion on BSM is limited because it is not the primary component of this project. The primary component is the stochastic volatility sampling that provides parameters for the Heston Model which will output predicted options prices. The goal of part 2 is to determine if the Heston Model does a better job of predicting options prices than the BSM. A secondary goal is to analyze the stochastic volatility process and consider methods to implement a predictive model in future applications.

In part 3 of this project I will implement neural networks to predict BTC closing prices. Neural networks are chosen because of the way they are trained, at each iteration loss is calculated and that loss is applied to parameter tuning before the next iteration begins. With each back propogation of loss, the model improves and converges to the true prices (ideally). Neural networks also introduce non-linearity into the model through specific activation functions, I think that neural networks will perform better at predicting BTC prices than other standard regressions like linear, log, and cubic spline polynomial regressions [16]. Part 2 and Part 3 are very different from each other because they are concerned with deriving different outcomes, so when reading this project, understand that the neural network parts are unrelated to the options pricing methods done in part 2.

2 Historical Data Analysis

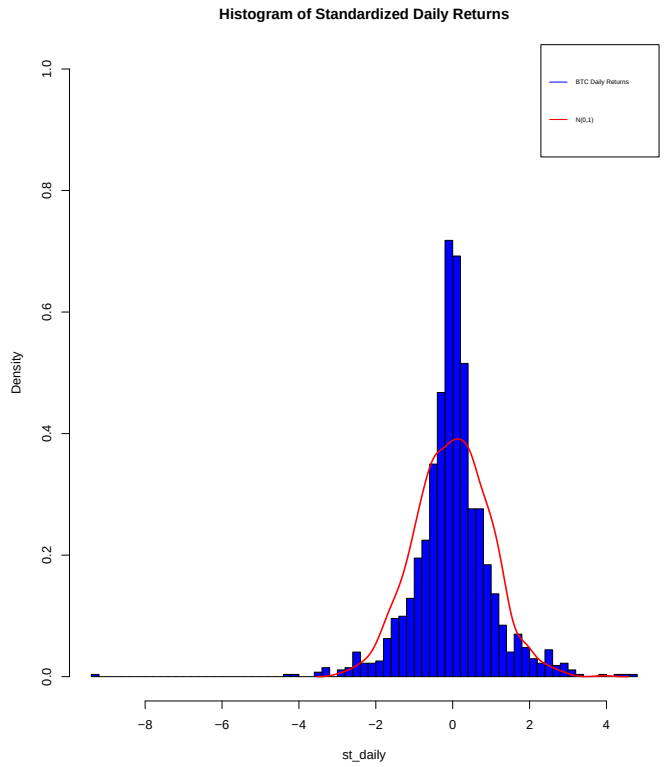

Figure 2: Distribution of Standardized Daily Returns

The histogram (Figure 2) shows the standardized daily returns for BTC prices from January 1, 2018 to September 23, 2021. Over it I plot a random normal variable over the same time interval (1362) to see how our data compares to a normal distribution. Overall the spread of the histogram aligns fairly well with the normal density but extends a little further because of a few outliers in the data. The peak in the daily returns data is nearly twice the size of the normal distribution implying that there is volatility in the data that is not captured accurately in a normal distribution. From my analysis I was also able to calculate the sample mean and sample variance, which I will use to solve for the instantaneous mean and instantaneous standard deviation. The instantaneous mean and standard deviation will be used in modeling the BTC options prices. The instantaneous mean and standard deviation are given by the following formulas:

$$\mu = \frac{\hat{\mu}}{\Delta t} + \frac{\sigma^2}{2}$$
$$\sigma = \frac{\hat{\sigma}}{\sqrt{\Delta t}}$$

where

- $\hat{\mu}$ is the sample mean of the log daily returns
- $\hat{\sigma}$ is the sample variance of the log daily returns

Below are the following statistics for the daily return data (Figure 3), as well as the calculated true mean and true standard deviation. From the excess kurtosis we seee that we have ”fat tails” meaning that the values on the ends of

the distribution are above the normal level, indicating that future returns will be either extremely large or extremely small.

μ	$\hat{\mu}$	$\hat{\sigma^2}$	σ^2	$\hat{\sigma}$	σ	Skewness	Excess Kurtosis
6.45e-06	0.001536	0.001636	4.482e-06	0.0404	0.00211	-0.44818	4.823

Figure 3: Table of Instantaneous Mean, Instantaneous Standard Deviation, Sample Mean, Sample Variance, Sample Standard Deviation, Skewness, Excess Kurtosis

2.1 Analyzing Data for Stationarity

The Augmented Dickey Fuller Test is a unit root test that tests for stationarity. A time series has stationarity if a shift in time does not cause a change in the shape of the distribution of the data. Stationarity correlates with mean, variance, and covariance remaining constant over time. The null hypothesis in an ADF test is that there is a unit root and the alternate hypothesis essentially says that the time series has stationarity. The presence of a unit root means that the time series is non-stationary. The ADF relies on selecting the appropriate regression model. The ADF is applied to the following model:

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} ... + \delta_{p-1} \Delta y_{t-p+1} + \epsilon_t$$

where α is a constant, β is the time trend coefficient, and p is the lag order of the autoregressive process. [5]

The other test that was used to test the mean-reversion and stationarity of our time series is the Hurst Test. The Hurst Test relies on the Hurst Exponent which is a measure of the long term memory of a time series. It is an effective measure for understanding the mean reversion within a time series and the average time it takes until the series reverts back to the mean. Mathematically, the Hurst Exponent, H, is defined in terms of the asymptotic behavior of the re-scaled range as a function of a given time span of a time series, more formally it is denoted:

$$E\left[\frac{R(n)}{S(n)}\right] = C_n^H \text{ as } n \to \infty$$

where

- $R(n)$ is the range of the first n cumulative deviations from the mean
- $S(n)$ is the series sum of the first n standard deviations
- n is the time interval of the observation
- C is a constant

A Hurst Exponent value between 0.5 and 1 indicates that the time series has a long term positive autocorrelation meaning that each successive observation further in time will likely be greater in value than the observation before it. A Hurst Exponent value between 0 and 0.5 indicates that there is oscillation between each subsequent observation meaning that a high value will be followed by a lower value and so on in that oscillating pattern. The oscillation would show that as the time interval progresses there would be some near convergence to a mean. [8]

From our ADF Test, our p-value is less than the significance level (0.05) but the ADF statistic is larger than any of our critical values. Because the p value is less than the significance level I can reject the null hypothesis and take the series to be stationary. However, these results slightly disagree with our Hurst Test results.

From the Hurst Test I am given a corrected empirical Hurst Exponent value of 0.59 which is greater than 0.5, indicating a trending series where high values are often followed by high values. This suggests to us that our data is not very mean reverting. From the Hurst and ADF test, we know that we have a trending time series with some stationarity. When we see the plot of the daily returns (Figure 5), there does seem to be some mean reversion in the time series.

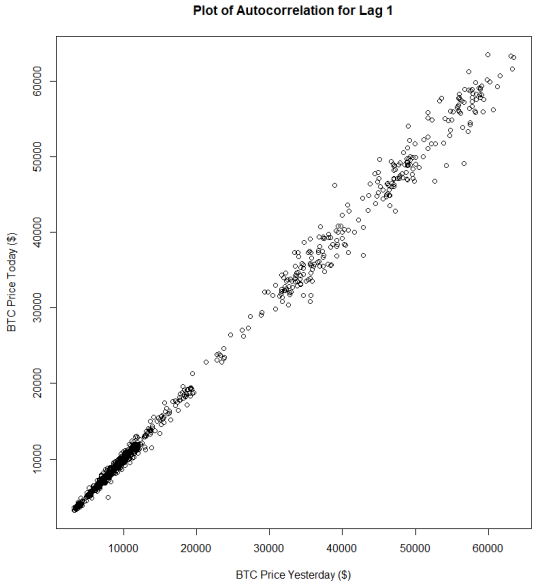

Figure 4: Lag 1 Autocorrelation Plot for BTC Closing Price

The autocorrelation plot (Figure 4) between BTC Price Today and BTC Price Yesterday shows a strong correlation in the initial stages of BTC when it's price was under $20,000. As the price increases the correlation becomes less strong.

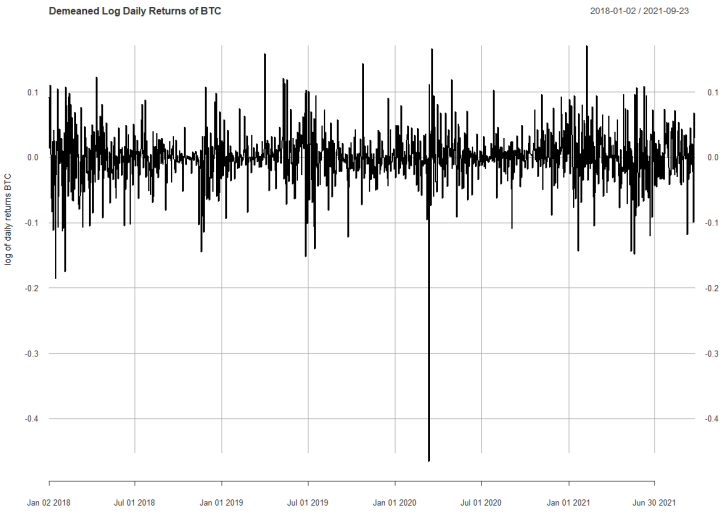

Figure 5: Centered Daily Log Returns of BTC

I derive the half life parameter to determine how long it takes for the series to revert back to a mean (if there is one), our Hurst and ADF test results imply some weak stationarity in the daily returns, and the plot of the daily returns shows some mean reversion throughout the time series.

$$t_{1/2} = \frac{ln(2)}{\lambda} = \tau ln(2)$$

λ is the decay constant

τ is the mean lifetime

$t_{1/2}$ is the half life

λ is the inverse of the mean lifetime which was found through a linear regression between two variables, the change between each observation in the lagged daily returns, and the difference between each observation in the daily returns. The two values are fit to one another and λ is the second coefficient in this formula. The results of this analysis gave us a half life value of 6.194, meaning that the returns revert back to some mean roughly every six days. The half life is a required parameter for the Heston Options Pricing Formula.

3 Black Scholes Merton (BSM) Implied Volatility Analysis, Motivations for Improvement

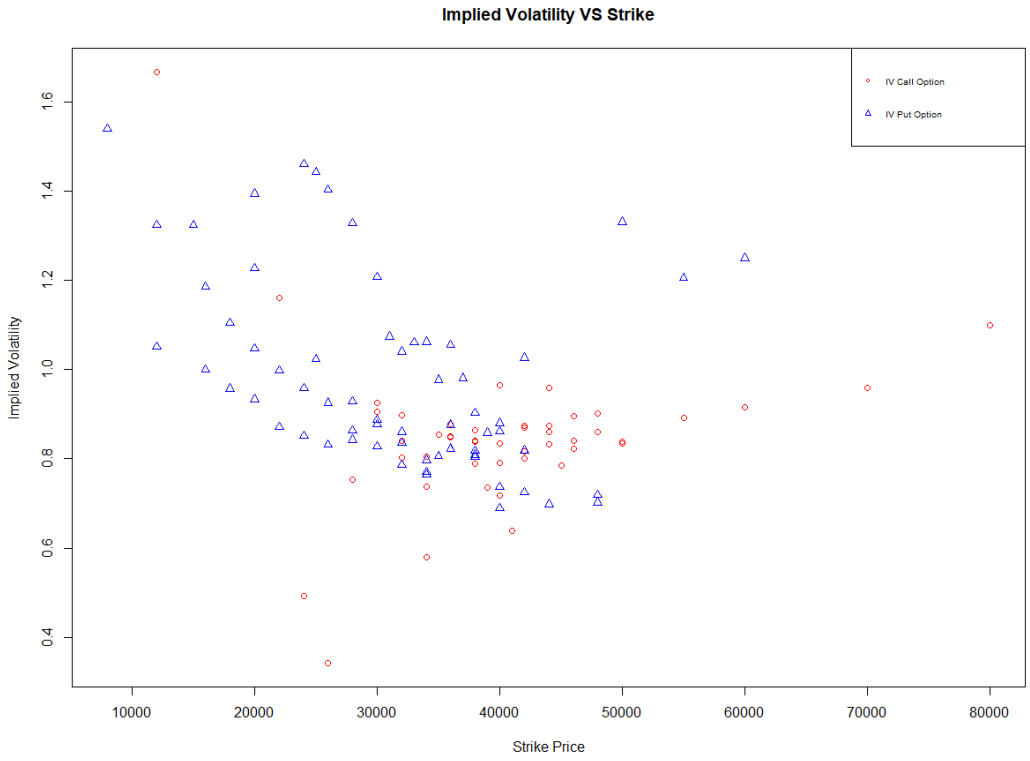

Figure 6: BSM Implied Volatility Plot (Volatility Smile)

In prior versions of this project, options prices were derived from historical BTC data using the Black Scholes Merton Options Pricing Equations [6]. The results can be seen in Figure 12. The implied volatility is derived from the

results of the BSM model and are plotted against the strike price, from Figure 6 we see a "volatility smile" present. This implies that the mean of our data changes as the strike price increases. Implied volatility increases when the underlying asset of an option is out of the money, or in the money, but not when it is at the money. The volatility in the data indicates that the BSM fails to accurately model BTC options because the BSM assumes constant volaility. To improve on the BSM, we will implement the Heston Options Pricing Model which assumes that volatility changes with strike price and asset price [11]. The formula for the Heston Options Pricing model is shown below:

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t^s \tag{3.1}$$

$$dv_t = k(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t^v \tag{3.2}$$

where

- S_t = asset price at time t
- r = risk free interest rate
- $\sqrt{v_t}$ = volatility (sd)
- σ = volatility of volatility
- θ = long term price variance
- k = rate of reversion to θ (half-life)
- W_t^S = Brownian Motion of the assets price
- W_t^v = Brownian Motion of the assets variance

In order to properly implement the Heston Model I will need to derive the necessary parameters using a stochastic volatility sampling that provides values for parameters σ, μ, θ, and ϕ. Before implemeting the Heston Model, the derived parameters will be tested for accuracy by plotting simulated daily returns against the true daily returns. The simulated daily returns will also be used to derived estimated closing prices and will be compared against the true closing prices. The goal of this analysis is to ensure that the parameters the SV simulation provides are accurately representing the true data. One area of concern is the correlation between W_t^S (Brownian Motion of the Assets Price) and W_t^v (Brownian Motion of the Assets Variance). If they are correlated then the implied volatility will increase with the variance of the returns. If that is the case then the outputs from the Heston Model will not be much better than the output of the BSM Model. The Heston Model usually has these two parameters correlated and this will be proven true once we derive the options prices.

3.1 Deriving SV Parameters and Testing for Accuracy

I analyze the Stochastic Volatility (SV) Model to account for shortcomings in the Black-Scholes Merton (BSM) Model. BSM assumes that the underlying volatility of an asset is constant over the life of the derivative thereby unaffected by changes in the price level of of the underlying security. In the SV model I assume that the volatility of the underlying price is a stochastic process instead of being constant. Below is the mathematical representation of the SV model [9] [11]:

- $\mathbf{y} = (y_1, y_2, ..., y_n)^T$

- **y** is a vector of returns with mean zero
- $y_t|h_t \sim N(0, e^{h_t})$
- $h_t|h_{t-1}, \mu, \phi, \sigma_n \sim N(\mu + \phi(h_{t-1} - \mu), \sigma_n^2)$
- $h_0|\mu, \phi, \sigma_n \sim N(\mu, \frac{\sigma_n^2}{(1-\phi^2)})$
- $N(\mu, \sigma_n^2)$ is normal distribution with mean μ and variance σ_n^2
- $\theta = (\mu, \phi, \sigma_n)^T$ is a vector of parameters where:
- μ = level of log variance
- ϕ = persistence of log variance
- σ_n = volatility of log variance
- $\mathbf{h} = (h_0, h_1, ..., h_n)$ = latent time varying process (log-variance process)

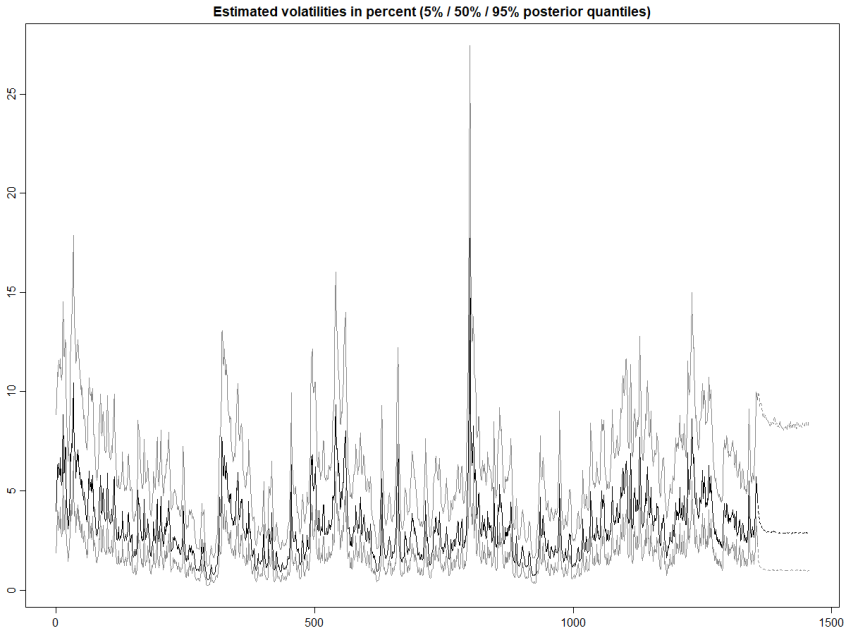

Figure 7: Estimated Volatilties Derived from Markov Chain Monte Carlo Method

To better understand the nature of the volatility inherent in the daily returns of BTC prices we resample the returns. I call the function *svsample* which simulates from the joint posterior distribution of the SV parameters μ, ϕ, & σ along with latent log volatilities. The function returns the Markov Chain Monte Carlo (MCMC) draws. We fed the daily return data into *svsample* and the function is able to do Bayesian regression to find our desired parameters and then outputs MCMC draws. I create a stocahstic volatility sample from the given daily return data and use that data to create a volatility plot, trying to predict volatility for 365 days. From the volatility graph (Figure 7) we get the following estimated volatilities for 5%/50%/95% posterior quantiles (1%,3%,8%). These values should be taken with a serious degree of skepticism as we forecast for 365 days as a theoretical exercise. In reality, any prediction that far out will be very off the mark in real life.

The density plots display plots for the density estimates for the posterior distributions of μ,σ, and ϕ. The mean values of these posterior distibutions will be used to simulate stochastic returns and we will compare these returns to the true returns. The plots are for our stochastic volatility samples that were calculated using our daily return data. From the density plots, we see that all three parameters are fairly normal but have varying levels of skewness (Figure 8). I use these derived parameters to simualte a stochastic volatility process for the same length of time that our dataset looks at (1358 days). The simulation then provides us with simulated daily returns that can be used to derive BTC Closing Prices. I will then take the mean squared errror (MSE) of both the simulated daily returns and the derived BTC Closing prices to see how accurate the stochastic volatility model was.

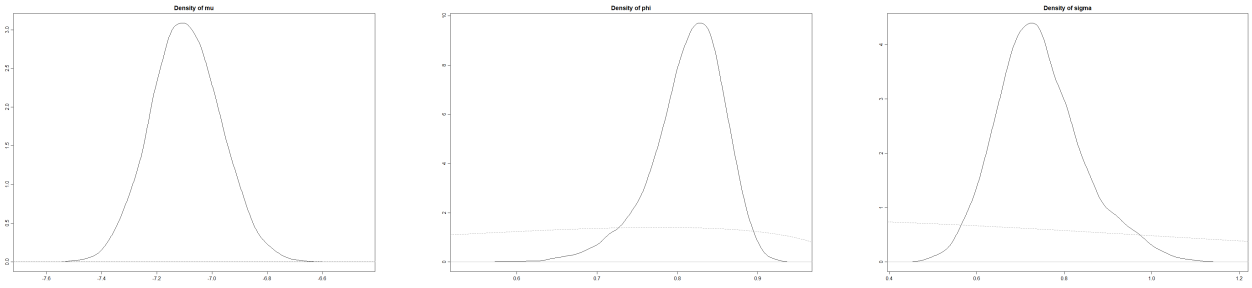

Figure 8: Paradensity Plots MCMC SV Sampling

Using the derived parameters, the true daily returns are sampled using a stochastic volatility process, which simulates from a joint posterior distribution from parameters μ, ϕ, and σ. The sampling also provides latent-log volatilities $(h_0, h_1, ..., h_n)$. The derived parameters are then used to simulate returns for 1358 days. The simulated daily returns are then plotted against the true daily returns and the MSE is calculated to test the accuracy of the simulation. The daily returns are then used to calculate simulated BTC Closing prices using the defined formula for daily returns:

$$\text{Daily Returns} = \frac{\text{Opening Price} - \text{Closing Price}}{\text{Opening Price}} \tag{3.3}$$

$$\text{Closing Price = Opening Price(1 - Daily Returns)} \tag{3.4}$$

The above derivation of the closing price is not rigorous and serves merely to test how well our derived parameters model the data. The next step in this analysis will be to try and develop a preditctive model that can be trained and tested on data.

Analysis of SV Predictions The derived simulated daily returns when plotted against the true returns in Figure 9 are near identical except for the fact that the simulated returns are shifted by an interval (Figure 9). This is obvious because we sampled our data with Lag 1, i.e the model only used one data point prior to calculate the return at the next point in time. The simulated daily returns are then used to calculate simulated BTC Closing Prices which are then plotted against the true BTC Closing prices (Figure 10). The results again show that the data is near identical except for the shift between the simulated data and the true data.

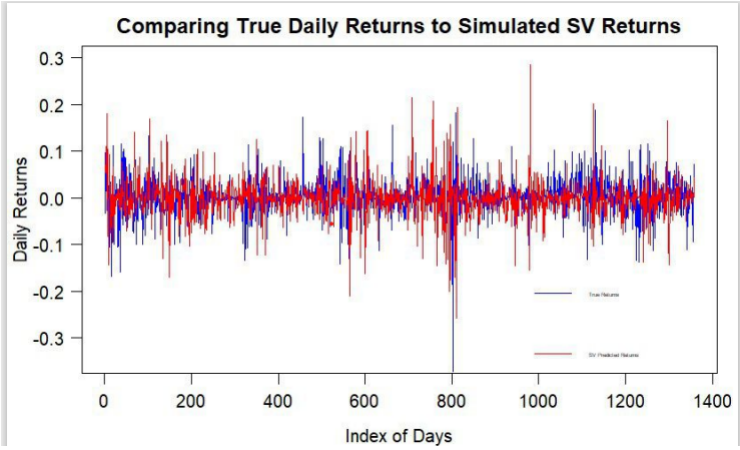

Figure 9: SV Sim Daily Returns VS True Daily Returns

Blue = True Returns, Red = SV Predicted Returns

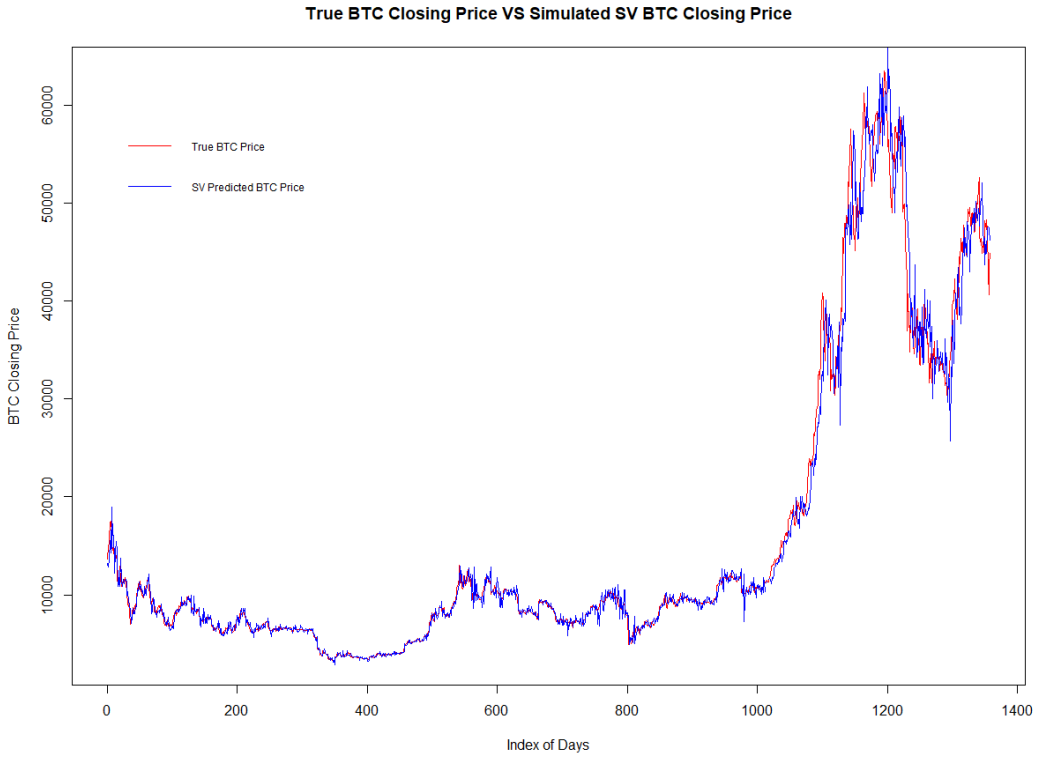

Figure 10: SV Derived BTC Closing Prices VS True Closing Prices

3.2 Implementing Heston, Comparing Results to BSM Results

Using the parameters derived from the SV simulation I simulate some BTC call option prices. To derive call option prices using the SV parameters I take advantage of the Heston Model for pricing options, which differs a little from the classical Black-Scholes Model (BSM). The Heston Model assumes that volatility is arbitrary and not constant as

assumed in the BSM. The Heston Model is set up as follows [11]:
I then take advantage of an existing global function called *CallHestoncf* from the publicly available package *NMOF* to calculate the Heston call option prices given the parameters. The volatility data is taken from our SV simulation over our BTC data and the value for k was calculated prior when we were testing for mean stationarity. The long term variance is calculated using the global function *lrvar*. Once I calculate the Heston call option values we then plot the values against the true call option prices and the BSM derived call options prices. The results show little improvement in precision when using the Heston Model versus BSM (Figure 11). In fact, when we analyze the Mean Absolute Percentage Error (MAPE) for both Heston and BSM we get values of 0.0129707 (Heston) and 0.01298701 (BSM). When I look at the absolute difference between the results we see that the difference is 1.627e-05 which is very small. The plots and the calculated MAPE values tell us that the Heston Model does not improve on the BSM much at all for our data (Figure 12).

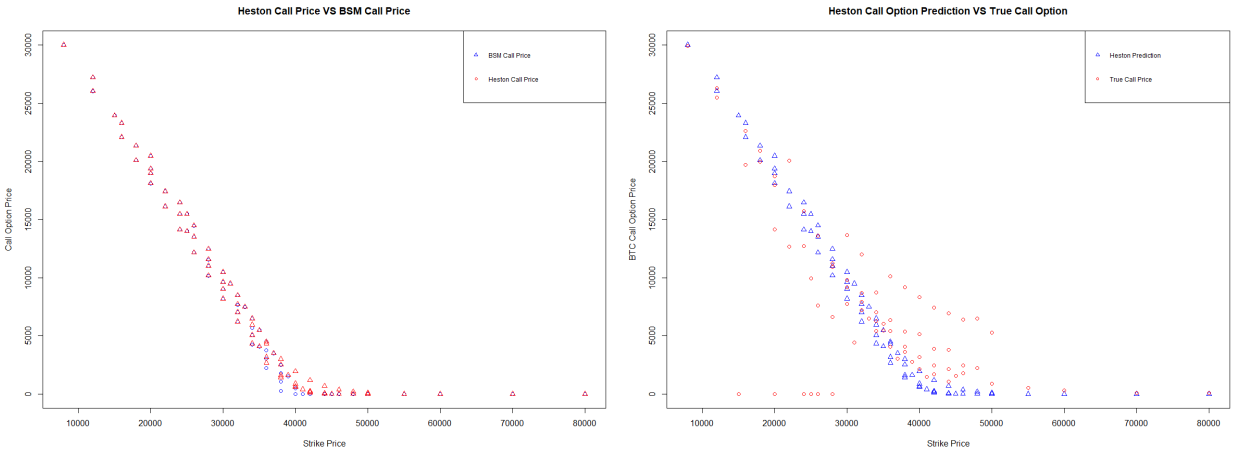

Figure 11: Heston Predicted Call Options VS BSM Predicted Call Options, Heston Predicted Call Options VS True Call Options

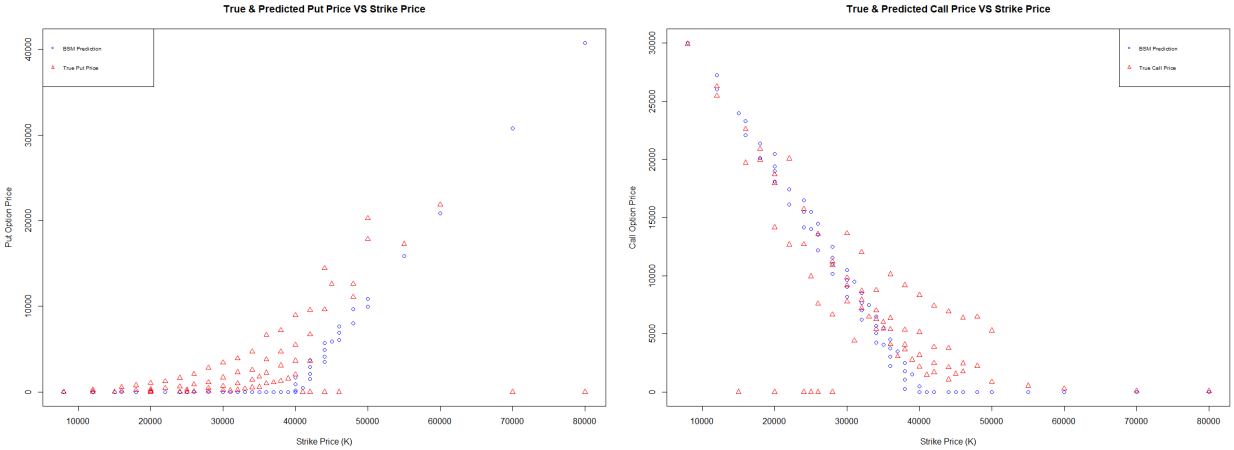

Figure 12: BSM Predicted Put Options VS True Put Options, BSM Call Options VS True Call Options

The BSM Model tracks fairly well with the actual BTC call Option Prices but it is clear from the plot that the model is not accurately modeling the volatility of BTC Call Option Prices. The true standard deviation that we derived from

our sample mean and sample variance does not adequately model the volatility inherent in the underlying. We use the daily risk free interest rate listed on the Federal Reserve's website. The Fed had r = 5% [10].
Looking at the BSM predicted put prices compared to the real put prices we see that the BSM did a reasonable job of modelinng Put Option Prices. The BSM predicted put prices gave us zero for most of the predictions, for T=10, this tracked well with the real put prices but as the strike price K increased the model became less accurate.The BSM followed the trend of the real data but not closely, and some predictions were very different from the true options price at that strike price.
To check the put call parity I take our predicted put and call option prices and compare them with the real put and call options prices. I re-write the put call parity as follows:

$$C - P = S - Ke^{-rT} \tag{3.5}$$

I calculate the LHS of (3.5) using both the predicted and real prices and plot the two sets of data on the same plot as a function of strike price. We see that put call parity does not necessarily hold but the differences in the LHS values are not drastic.

4 RNN-LSTM Model

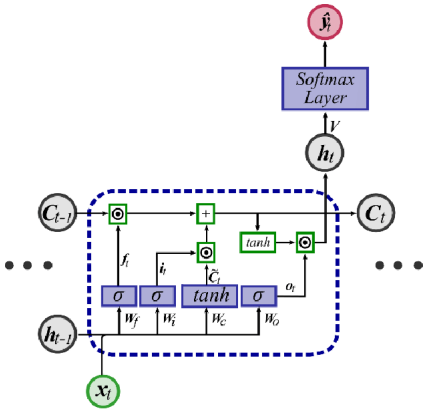

Figure 13: LSTM Model Diagram [14]

A defining feature of cryptocurrencies are their volatility. Investing in cryptocurrencies carries greater risk than investing in the stock market due to the 24/7 trading structure of cryptocurrencies and because of the legal challenges that cryptocurrencies must deal with on a frequent basis [7]. To better forecast BTC prices and BTC options we will employ a Recurrent Neural Network (RNN) model that utilizes the idea of Long Short Term Memory (LSTM). A large benefit of RNN-LSTM models is that they are able to derive current observations using previous observations which allows information to persist throughout the network. By having an inherent "memory" LSTMs are well suited to forecasting models that are dependent on time series. An issue that arises with standard ("vanilla") RNNs is that it reiterates one layer many times over without ever storing the previous data. A vanilla RNN takes data, iterates that value multiple times through the same layer, each iteration wipes the previous observation and replaces it with a new value. This process becomes computationally heavy and skews the information over a time series since it has no memory. The errors in such models can explode as the network updates with each iteration, this problem is seen in the "vanishing gradient problem", whereby the gradient used to update the loss in each layer explodes as it back propogates through the network [20].

LSTM networks work well classifying, processing, and predicting given a set of time-series data. LSTMs were designed as a response to the ”vanishing gradient problem” which occurs with standard RNNs. When training a standard RNN, the back propagation that updates the error in each layer can tend to zero or infinity due to the large number of computations required to fully back propogate through the network. LSTMs are utilized because they diminish the possibility of having a vanishing gradient occur in the back propogation. In the standard RNN, one neural network is repeated a set number of times but each neural network is comprised of one layer, often a tanh activation function. LSTMs have a similar structure to a standard RNN but differ in that each neuron is composed of four layers with different activation functions. These layers are connected by a ”cell state” which sends information through between each neuron thus serving as the ”memory” within the network acting as a conveyor belt that provides memory to the model [15]. There are various papers that attempt to improve model performance by manipulating the parameters and number of layers in the model. In this project we do a similar examination to determine the modeling performance of RNN-LSTM on BTC prices. However, the examination will show that the model marginally improves and sometimes worsens as you add hidden layers and change parameters. The best way to improve the model would be to make the data more robust. One way that can be done is by feeding the model more parameters like the hash rate, the trading volume, and other statistical measures that can be easily pulled from the internet.
To test the performance of the model I subset the data into four training and four testing subsets [13]. The data was split into the following subsets:

	Training	Testing
Subset 1	[1,200]	[201,250]
Subset 2	[150,349]	[350,399]
Subset 3	[300,499]	[500,549]
Subset 4	[450,649]	[650,699]

Figure 14: Model Performance on Training Subsets

The model was analyzed using a standard LSTM model with two hidden layers. The performance of the training and testing models are listed below:

Training	Average Loss	Average Validation Loss
Subset 1	0.01197	0.009103
Subset 2	0.01181	0.079359
Subset 3	0.021359	0.05739
Subset 4	0.010241	0.005484

Figure 15: Model Performance on Training Subsets

Prediction	Mean Squared Error	Mean Absolute Error
Subset 1	0.00015	0.011955
Subset 2	0.00407	0.05009
Subset 3	0.37679	0.53547
Subset 4	0.15516	0.3611

Figure 16: Model Prediction VS Real Data

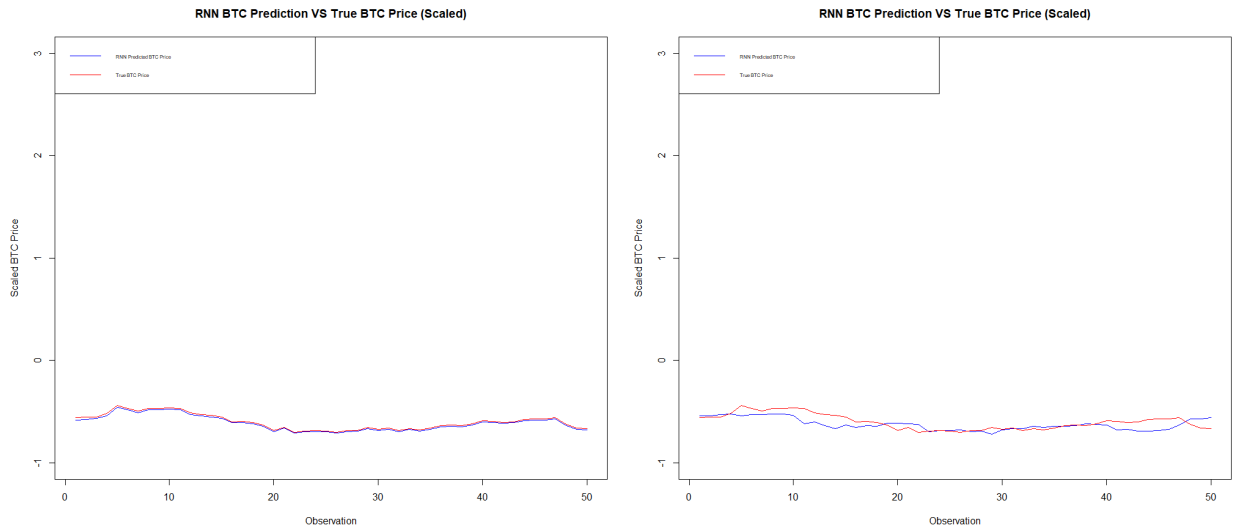

Figure 17: Predictions VS True BTC Price Subsets 1 & 2

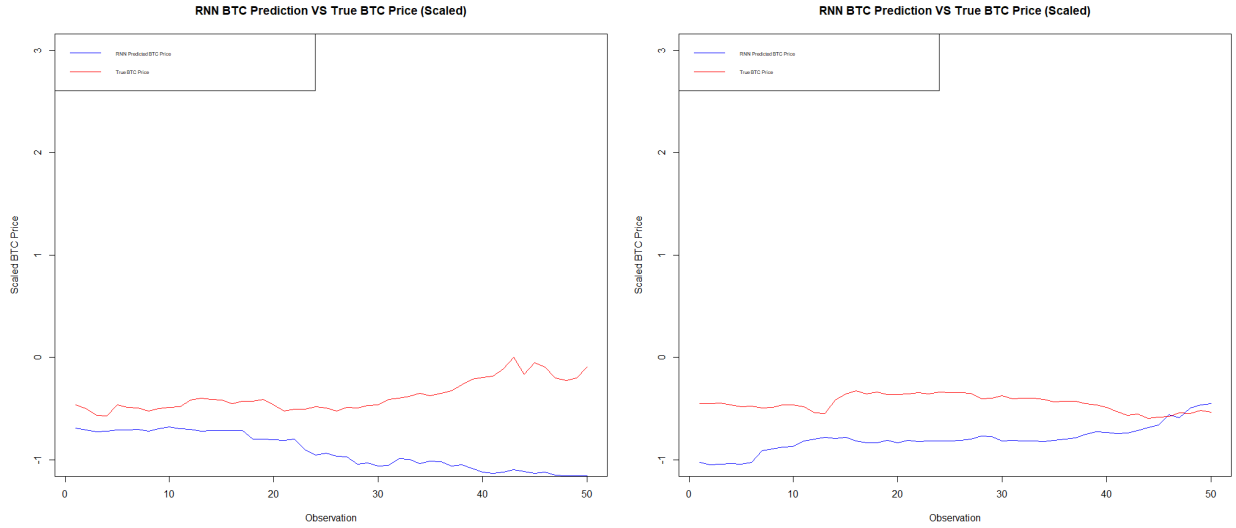

Figure 18: Predictions VS True BTC Price Subsets 3 & 4

In Figure 17 I show the RNN prediction over the subsets vs the true data (Figure 17, Figure 18)). The subset was drawn from original training data with 800 observations and original testing data with 550 obvservations. The train/test split of the data results in about a 60/40 split with 60% of the data in the training set and 40% of the data in the testing set. Figure 19 is the prediction over the testing set.

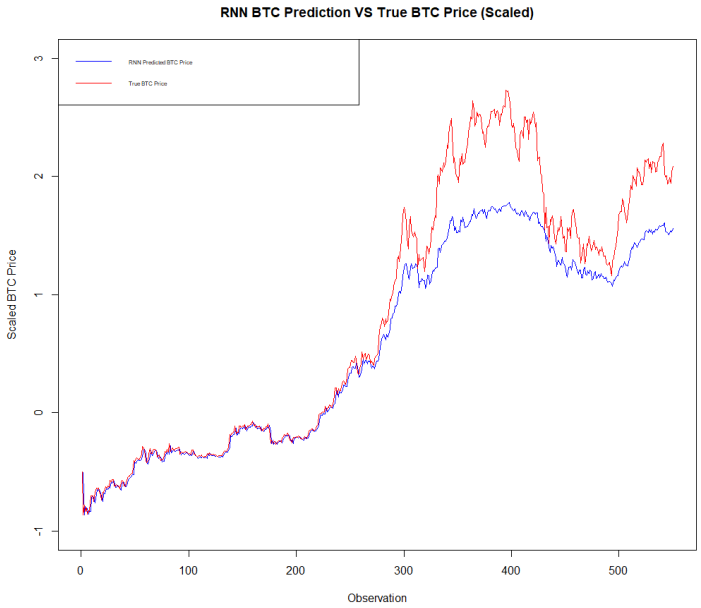

Figure 19: Prediction over Test Data, 2 Hidden Layer LSTM

Overall we see that the prediction over the testing set does well in predicting BTC prices for the first 300 observations in the testing set. However from observations 300-500 accuracy decreases as the predicted price is consistenly less than the true BTC price. Analytically this is expected because of the sharp increase in BTC's price that began around July 2020. The prices in the later part of the time series are more than double the prices that the model was trained on in the first 60% of the historical time series data.

5 Improving on LSTM, Implementation of RNN-GRU Model

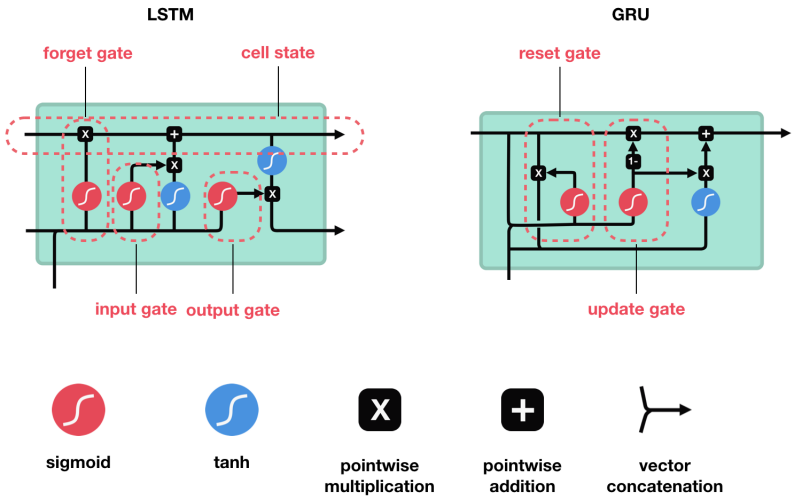

Figure 20: LSTM Diagram Compared to GRU Diagram [17]

5.1 Understanding Gated Recurrent Units (GRU)

To improve on the LSTM NN model implemented in the previous section we implement Gated Recurrent Units (GRU). GRUs are considered a more streamlined and efficient version of LSTM NNs because it is less costly computationally and provides greater nuance in how memory is stored in the neural network. The main features of GRUs occur in the hidden unit between the input and output layers. Within the hidden layer there is a method to pass on "significant/important" info and a method to "reset" and drop out outputs that contribute little to the overall solution [16].

The hidden layer is composed of an update gate and reset gate that determine what information is carried over from the prior state and what information is removed. The gates mathematically are vectors with values between 0 and 1, the constraints allow us to perform convex operations which is crucial to allow for back propogation and stochastic gradient descent. The reset gate carries short term memory through the network while the update gate carries long term memory through the network. Both work in tandem to create a candidate hidden state that determines how much subsequent output will be influenced by short term and long term memory. Mathematically the reset and update gate are defined as follows:

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r) \tag{5.1}$$

$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z) \tag{5.2}$$

where

- $R_t \in \mathbb{R}^{n \times h}$ is the reset gate
- $Z_t \in \mathbb{R}^{n \times h}$ is the update gate
- $W_{xr}, W_{xz} \in \mathbb{R}^{d \times h}$ are weight parameters
- $W_{hr}, W_{hz} \in \mathbb{R}^{h \times h}$ are weight parameters
- $b_r, b_z \in \mathbb{R}^{1 \times h}$ are bias terms
- $\sigma()$ is a sigmoid function

The candidate hidden state is then constructed by the integrating the reset gate with the standard network updating mechanism, meaning that the hidden state is activated just as any other input would in that layer. The hidden state is then defined as:

$$\tilde{H}_t = \tanh(X_t W_{xh} + (R_t \circ H_{t-1}) W_{hh} + b_h) \tag{5.3}$$

where

- $W_{xh} \in \mathbb{R}^{d \times h}$ is a weight parameter
- $W_{hh} \in \mathbb{R}^{h \times h}$ is a weight parameter
- $b_h \in \mathbb{R}^{1 \times h}$ is a bias term
- $\circ$ is a Hadarmard element wise multiplication operator

The output is a "candidate" which is used to determine the final output of the hidden state, that output will then be sent through an activation function and output a final prediction. The hidden state is determined once we incorporate

the effect of the update gate, this final step determines how much value is placed into short term memory and how much value is placed in long term memory. Mathematically, the final update step of the GRU network is defined as:

$$H_t = Z_t \circ H_{t-1} + (1 - Z_t) \circ \tilde{H}_t \tag{5.4}$$

When Z_t (the update gate) is close to 1 we retain the old/previous state, when it is closer to 0 we take choose to use the updated state. This update step allows us to filter prior time series data out of the network to create more accurate predictions. This selection step also helps mitigate the vanishing gradient problem that occurs with vanilla RNNs because due to the structure of the network, any back propogation will be bounded between (-1,1) or (0,1) thereby negating the impact of a vanishing/exploding gradient [17].

5.2 GRU NN Implementation Results

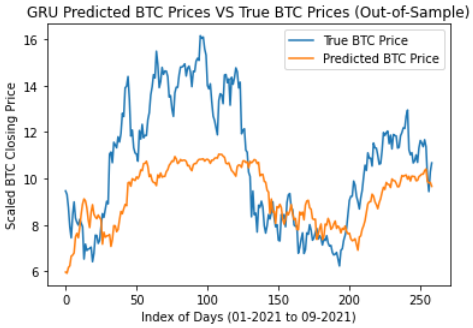

Figure 21: GRU Predicted BTC CLosing Prices VS True BTC Closing Prices (Out of Sample)

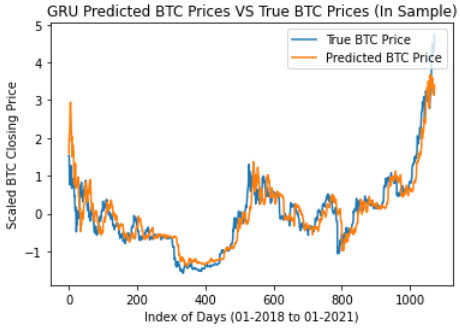

Figure 22: GRU Predicted BTC CLosing Prices VS True BTC Closing Prices (In Sample)

The in-sample mean squared error (MSE) was very low (0.14) which tells us that our GRU NN learned the training data very well (Figure 22). However when we make predictions out of sample we get an MSE of 8.41 which is very high. Looking at the above out of sample plot we see that this makes sense, our model performed terribly against the true values. A big reason for this has to due to with the BTC price time sequence. We trained our model on data that came before the big price spikes that BTC saw during the pandemic. No matter how well the model learned the training data, it would be very difficult for the model to predict a jump of $20k in 2020 and that is why we obtain such a poor MSE value out of sample.

6 Conclusion

From my work we see that the BTC option prices derived using the Black-Scholes Merton (BSM) Option Pricing Method tracked the trend of the real option prices well but was not an adequate representation of the true values. The BSM call prices were more accurate than the BSM put prices. The BSM call prices accurately predicted many call option prices and the amount of variance from the true value remained relatively consistent when comparing the numbers and the plots.

BSM and Heston Option Price Analysis The BSM put option prices gave us many zero and near zero values, far more than in the call option estimates. However, there were many instances of put option prices being zero in the Derebit dataset, and the BSM correctly predicted those values. But looking at the trend lines of both the predicted and true values, it is clear that the predicted values from the BSM are far more linear in their trend. This implies that the volatility inherent in BTC option prices was not properly captured through the BSM pricing model. The Heston model was used to try to account for the volatility in the BTC prices because the Heston model uses volatility data as inputs to find options prices. However, the Heston predicted options prices provided no significant improvement from the BSM options prices. The Heston model used the simulated volatilities that I derived using MCMC stochastic sampling. Although the Heston model provided no significant improvements over the BSM, the volatilities that were used as parameters worked very well when used them to model BTC closing prices given our crude implementation (Figure 10). The outputs are not a model but serve as good sanity check to ensure that the volatilities that were derived from the MCMC sampling are representative of the true data. The paper [21] we referenced in class found success modeling BTC options using a SV and SVCJ model that properly accounted for the jumps using a combination of poisson, normal, and exponential distributions. Next steps would be creating and training a predictive model using the stochastic volality processes, similar to the SVCJ paper that used a stochastic volatilty process in conjunction with random Poisson distributions to get predicted prices [21]. Moving forward with this part of the project, it will be important to implement a stochastic volatility model to derive proper statistical parameters. The SV sampling and rudimentary modeling that we did showed that the derived parameters from the SV sampling are indicative of the true behavior of BTC prices, next steps would be to figure out a way to design a model that can be applied to a training set and then tested to predict prices on an unknown set of data.

Analysis of Neural Network Performance The second half of this project was implementing two types of neural networks and analyzing their performance. A Long-Short-Term-Memory Neural Network (LSTM-NN) and a Gated Recurrent Unit Neural Network (GRU-NN) were implemented because they are designed to store memory throughout the model thereby being well suited for time series data. Both models worked very well in-sample but struggled out-of-sample. The out of sample predictions tracked the trends of the true prices well but could not match the magnitude of the prices. This can be attributed to the training data being drastically different from the testing data. The testing data contained BTC prices between $20,000 and $60,0000 and thet training data was below $20,000. The model had no indication of this drastic increase in prices and was assuming that prices would not explode past $20,000 in the way that it did in real life. Implementing a random jump process into the neural networks could improve performance but it is doubtful it would improve much if we use the same set of training data.

Next Steps, Considerations for Improvement To improve the performance of the neural networks I would need to better prepare the data so that the model can better predict drastic jumps in prices. Adjusting the training data by segmenting them into subsets might improve the model's performance over smaller intervals but might not change anything when I test the model on the out of sample data. Cross validation would be the preferred method as it is generally known as a great way of improving model accuracy. Cross validation is a resampling method that utilizes

different portions of the data to test and train a model in different iterations. You continuously partition the data into complementary subsets and train the model on each subset, we then test the model by predicting on a subsequent complementary subset and comparing results. The idea is to do this enought times to combine/average measures of fitness in prediction to derive more accurate estimates of model prediction performance [18]. Also, more time can be invested in tuning the parameters and altering some components of the neural networks to try and get better results, however this is unlikely to improve the results much as playing with the layers of neural networks is more arbitrary and has been shown to marginally improve performance from standard neural network set ups. Truthfully, a better type of neural network is needed to accurately model BTC prices. Looking further ahead, Deep Reinforcement Learning would probably be the best method for accurately modeling the prices because there is constant feedback within the network that allows the model to improve faster than in a traditional supervised or unsupervised learning environment [22].

7 References

[1] Voigt K. Rosen A. "What is Bitcoin? Here's How BTC Works" *NerdWallet* 2021
https://www.nerdwallet.com/article/investing/what-is-bitcoin

[2] Hankin A. "Bitcoin Options Are Headed to the US" *Investopedia* 2019
https://www.investopedia.com/articles/investing/033115/it-possible-trade-bitcoin-options.asp

[3] Huang J. O'Neill C. Tabuchi H. "Bitcoin Uses More Electricity Than Many Countries. How is That Possible?" *The New York Times* 2021
https://www.nytimes.com/interactive/2021/09/03/climate/bitcoin-carbon-footprint-electricity.html

[4] Stampfli J. Goodman V. "The Mathematics of Finance: Modeling and Hedging *Brooks/Cole* 2001

[5] Prabhakaran S "Augmented Dickey Fuller Test(ADF Guide) - Must Read Guide" *machinelearningplus.com*
https://www.machinelearningplus.com/time-series/augmented-dickey-fuller-test/

[6] Sarkulov F. "MATH523 Summer Project: Modeling BTC Options" *Department of Applied Mathematics, Illinois Institute of Technology* 2021

[7] Iansiti M. Lakhani K. "The Truth About Blockchain" 2017 *Harvard Business Review*
https://hbr.org/2017/01/the-truth-about-blockchain

[8] Mansukhani S. "The Hurst Exponent: Predictability of Time Series" 2019 *Analytics Informs*
https://pubsonline.informs.org/do/10.1287/LYTX.2012.04.05/full/

[9] Stan Development Team. "2.5 Stochastic Volatility Models" *Stan's User Guide*
https://mc-stan.org/docs/2_21/stan-users-guide/stochastic-volatility-models.html

[10] The Federal Reserve "Selected Interest Rates (Daily)" 2021 *The Federal Reserve*.
https://www.federalreserve.gov/releases/h15/

[11] Dunn R. Hauser P. Seibold T. Gong H. "Estimating Option Prices with Heston's Stochastic Volatility Model" *Kenyon College, College of New Jersey, Western Kentucky University, Valparaiso University*

[12] Kastner G. "Dealing with Stochastic Volatility in Time Series Using the R Package stochvol" *WU Vienna University of Economics and Business* https://cran.r-project.org/web/packages/stochvol/vignettes/article.pdf

[13] Xi Sun. "Stock Price Prediction with LSTM Model" *Department of Applied Mathematics, Illinois Institute of Technology* 2019

[14] Diaz Achanccaray M.P "Block Diagram of the LSTM RNN Cell Unit" *Research Gate* 2017
https://www.researchgate.net/figure/Block-diagram-of-the-LSTM-recurrent-neural-network-cell-unit-Blue-boxes-means-sigmoid_fig2_328761192

[15] Hochreiter S. Schmidhuber J. "Long Short-Term Memory" *Neural Compuation, Volume 9, Issue 8* 1997

[16] Yadav S. "Intro to Recurrent Neural Networks LSTM — GRU" *Kaggle* 2018
https://www.kaggle.com/thebrownviking20/intro-to-recurrent-neural-networks-lstm-gru

[17] Phi M. "Illustrated Guide to LSTM's and GRU's: A step by step explanation" *Towards Data Science - Medium* 2018
https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

[18] Bruno Spilak ”Deep Neural Networks for Cryptocurrencies Prediction” 2019 *Zuberlin Humboldt Universitat* https://d-nb.info/1185667245/34

[19] Kejsi Sturga. Olti Qirici ”Bitcoin Price Prediction with Neural Networks” 2019 *University of Tirana* http://ceur-ws.org/Vol-2280/paper-06.pd

[20] Ana Lucia Lima ”Bitcoin Price Prediction Using Recurrent Neural Networks and LSTM” 2021 *Analytics Vidhya* https://www.analyticsvidhya.com/blog/2021/05/bitcoin-price-prediction-using-recurrent-neural-networks-and-lstm/

[21] Jun Hou A. Wang W. Chen C. Hardle W. ”Pricing Cryptocurrency Options” *Journal of Financial Econometrics* 2020

[22] Tatsat, Hariom, et al. ”Machine Learning and Data Science Blueprints for Finance: From Building Trading Strategies to Robo-Advisors Using Python.” O’Reilly Media, Inc., 2020.

8 Appendix

```
library(quantmod)
library(fBasics)
library(tidyr)
library(TSA)
library(urca)
library(pracma)
library(zoo)
library(ggplot2)
library(pins)
library(skimr)
library(prophet)
library(stochvol)
library(MLmetrics)
library(sandwich)
library(NMOF)
library(readxl)
library(derivmkts)



#pulling BTC historic daily data
getSymbols("BTC-USD",src = 'yahoo', from =as.Date("2018-1-01"),
        to =as.Date("2021-09-23") )
btc <- `BTC-USD`
BTCClose <- btc$`BTC-USD.Close`
BTCClose <- na.omit(BTCClose)
BTCOpen <- btc$`BTC-USD.Open`


#Graphics
#i.) timeseries of BTCAdj
plot.xts(BTCClose,
       main = "Bitcoin Price Time Series From January 2018 to June 2021",
       xlab = "Date", ylab = "Price")

#chart series for daily returns of BTC
chart_Series(dailyReturn(btc))
stats_btc <- basicStats(btc)

daily_return <- dailyReturn(btc)
basicStats(daily_return)
```

```
#deriving the true mean and true sd from the sample mean and sample variance
samp_mean <- mean(daily_return)
samp_sd <- sqrt(var(daily_return))

tru_sd <- as.numeric(samp_sd/(sqrt(365)))
tru_mean <- as.numeric((samp_mean/365)+(0.5)*(tru_sd^2))


#plotting the histogram of standardized daily returns with a standard normal
   ↪ distribution N(0,1)
t <- seq(1,1271,1)
theor <- rnorm(t,mean = 0, sd=1)
st_daily <- (daily_return-mean(daily_return))/stdev(daily_return)
hist(st_daily, breaks = 50, col ="blue", ylim = c(0,1),
    freq=F,main = "Histogram of Standardized Daily Returns")
legend("topright",legend= c("BTC Daily Returns","N(0,1)"),
     col = c("blue","red"), lty = 1, cex = 0.5)
par(new=T)
lines(density(theor), col = "red",lwd=2 )
basicStats(st_daily)

#analyzing correlation between yesterday and today price
btc.ts <- ts(as.vector(btc$`BTC-USD.Close`),start=c(2018,2021), frequency =
   ↪ 365)
plot(y=btc.ts, x=zlag(btc.ts), xlab = "BTC Price Yesterday ($)",
    ylab = "BTC Price Today ($)",
    main = "Plot of Autocorrelation for Lag 1")

x = zlag(btc.ts)
index <- 2:length(x)
cor(btc.ts[index],x[index])
basicStats(x)

#testing mean reversion using ADF test and Hurst Test
ADF_results <- summary(ur.df(daily_return,type = "drift",lags=1))
Hurst_results <- hurstexp(daily_return)

#calculating half-life for mean reversion
dr.lag <- lag(daily_return,-1)
delta.dr <- diff(daily_return)
df <- cbind(daily_return,dr.lag,delta.dr)
df <-df[-1,]
regress.results <- lm(delta.dr ~dr.lag,data=df)
lambda <- summary(regress.results)$coefficients[2]
```

```
half.life <- -log(2)/lambda

#here we sample our data to determine parameters for our SV model
logreturns <- logret(BTCClose, demean = TRUE)
plot(logreturns, type= "l",ylab = "log of daily returns BTC", xlab = "Date",
    ↪ main ="Demeaned Log Daily Returns of BTC")
res_sv <- svsample(logreturns, designmatrix = "ar1")

#here we calculate the estimated volatilities for a 100 day out prediction
volplot(res_sv,forecast = 100, dates = daily_return$date[seq_along(res_sv)])

#These plots show us the density of our SV parameters mu, phi, and sigma
paradensplot(res_sv,showobs = FALSE)
residual <- resid(res_sv)
sv_parameters <- summary(res_sv)
plot(residual, seq(1,1358,1))

sv_sim <- svsim(length(daily_return),mu = -7.087,phi = .812, sigma = .741, nu
    ↪ = Inf,rho=0)
par(mfrow = c(2,1))
plot(sv_sim)

summary(sv_sim)
par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0), las=1,xpd =TRUE)
plot(x= seq(1,1358,1),y = daily_return,ylim= c(-0.35,0.3), xlab = "Index of
    ↪ Days",ylab="Daily Returns", main = "Comparing True Daily Returns to
    ↪ Simulated SV Returns",type = "l",col = "blue")
lines(sv_sim$y, col ="red")
legend(x=100, y = -1, inset = c(0,-0.75), xpd =TRUE, legend = c("True Returns
    ↪ ","SV Predicted Returns"), col = c("blue","red"),bty="n",lty = 1:1, cex
    ↪ = .3, pt.cex=20)

#Comparing MSE and assessing accuracy of the simulated returns
returns_compare <- cbind(daily_return,sv_sim$y)
MSE_returns <- mean((daily_return - sv_sim$y)^2)

#We will derive the associated simulated closing prices using the simualted
    ↪ daily returns. We will asumme we know the opening price of that day and
    ↪ using that infomration will derive the closing price. We will then plot
    ↪ those prices and see how close they get to the actual prices
sim_returns <- sv_sim$y
sv_btc_prices <- rep(0, length(daily_return))
for (i in 1:length(daily_return)){
  sv_btc_prices[i] <- BTCOpen[[i]]*(1-sim_returns[[i]])
}
```

```
par(mar=c(5.1,4.1,4.1,2.1),mgp=c(3,1,0),las=0)
plot(x = seq(1,1358,1),y = BTCClose, type = "l", col = "red", xlab = "Index of
    ↪  Days", ylab = "BTC Closing Price", main = "True BTC Closing Price VS
    ↪ Simulated SV BTC Closing Price")
lines(sv_btc_prices, col = "blue")
legend(x = 1, y= 60000, legend = c("True BTC Price", "SV Predicted BTC Price")
    ↪ , col =c("red","blue"),lty = 1:1,cex =0.75,box.lty = 0,bg = "transparent
    ↪ ")

closeprice_compare <- cbind(BTCClose,sv_btc_prices)
closingprices <- 0

#Calculating MSE between the SV predicted prices and the true prices
for (i in 1:length(sv_btc_prices)){
  closingprices <- (1/length(sv_btc_prices))*(sum((BTCClose[[i]]-sv_btc_prices
      ↪ [[i]])^2))
}

Options_Data_Derebit_Compiled <- readxl::read_xlsx("
    ↪ Options_Data_Derebit_Compiled.xlsx")
predicted_calls <- rep(0,nrow(Options_Data_Derebit_Compiled))
predicted_puts <- rep(0,nrow(Options_Data_Derebit_Compiled))

s0_list <- Options_Data_Derebit_Compiled[[1]]
t_list <- Options_Data_Derebit_Compiled[[2]]/365
k_list <- Options_Data_Derebit_Compiled[[3]]
callprice <- Options_Data_Derebit_Compiled[[5]]
putprice <- Options_Data_Derebit_Compiled[[7]]

heston_predicted_call_options <- rep(0,77)
longrun_var_svsim <- sv_sim$vol0
for(i in 1:length(heston_predicted_call_options)){

  heston_predicted_call_options[i] <- callHestoncf(s0_list[i],k_list[i],t_list
      ↪ [i],r=.05,q=0,v0 =.0527,vT =longrun_var_svsim,rho =0,k = half.life,
      ↪ sigma = sv_sim$vol[i])

}

plot(x=k_list,y = heston_predicted_call_options,xlab = "Strike Price", ylab =
    ↪ "BTC Call Option Price", main = "Heston Call Option Prediction VS True
    ↪ Call Option",col = "blue",pch=2)
points(x=k_list, y= callprice, col = "red")
legend('topright',legend = c("Heston Prediction","True Call Price"),col = c("
    ↪ blue","red"),pch = 2:1,cex=0.75)
```

```
plot(x=k_list,y=predicted_calls, col ="blue", xlab = "Strike Price",ylab = "
    ↪ Call Option Price", main = "Heston Call Price VS BSM Call Price")
points(x=k_list, y=heston_predicted_call_options, col = "red",pch=2)
legend("topright", legend = c("BSM Call Price","Heston Call Price"), col = c("
    ↪ blue", "red"), pch =2:1 ,cex = 0.75)

#See here that the SV Heston Model does not improve our results greatly. The
    ↪ predicted call prices are near identical with the exception of a few
    ↪ points around K = 40k. We will now test MAPE between both predictions to
    ↪  see which one give us better results

#Calculating MAPE between the SV Heston Calls and the true prices
for (i in 1:length(heston_predicted_call_options)){
  Heston_call_MAPE <- (1/length(heston_predicted_call_options))*(sum((abs(
      ↪ heston_predicted_call_options[i]-callprice[i]))/callprice[i]))
}

#Calculating MAPE between the BSM and true prices (CALL)
for (i in 1:length(predicted_calls)){
  BSM_call_MAPE <- (1/length(predicted_calls))*(sum((abs(predicted_calls[[i]]-
      ↪ callprice[i]))/callprice[i]))
}
MAPE_diff <- abs(BSM_call_MAPE- Heston_call_MAPE)

#From the Mean Absolute Percentage Error we see the Heston Model provides
    ↪ minimal improvements to our predicted call prices.

#seeing how BSM model holds up
library(ragtop)
library(qrmtools)


for (i in 1:77) {
  predicted_calls[i] <- blackscholes(callput = 1,s0_list[i],k_list[i],
                             r=.05,t_list[i],vola=tru_sd)
}
for (j in 1:77) {
   predicted_puts[j] <- blackscholes(callput =-1,s0_list[j],k_list[j],
                             r=.05,t_list[j],vola=tru_sd)
}

#Plotting T=10 Days for Calls
pred_t10 <- cbind(k_list[1:19],predicted_calls[1:19],predicted_puts[1:19])
```

```
plot(x= k_list[1:19], y = callprice[1:19],col= "red", ylim = c(0,40000),type =
    ↪  "p", xlab = "Strike Price (K)", ylab = "Call Option Price", main = "BSM
    ↪  Call Price T= 10 Days")
par(new=T)
points(x = k_list[1:19], y = predicted_calls[1:19], col = "blue", pch = 2)
legend("topright",legend = c("True Call Price","BSM Call Price"), col = c("red
    ↪ ","blue"),pch = c(1,2), cex=0.5)

#Plotting T=31 Days for Calls
pred_t31 <- cbind(k_list[20:39],predicted_calls[20:39],predicted_puts[20:39])
plot(x= k_list[20:39], y = callprice[20:39],col= "red", ylim = c(0,40000),type
    ↪  = "p", xlab = "Strike Price (K)", ylab = "Call Option Price", main = "
    ↪ BSM Call Price T= 31 Days")
par(new=T)
points(x = k_list[20:39], y = predicted_calls[20:39], col = "blue", pch = 2)
legend("topright",legend = c("True Call Price","BSM Call Price"), col = c("red
    ↪ ","blue"),pch = c(1,2), cex=0.5)

#Plotting T=59 Days for Calls
pred_t59 <- cbind(k_list[40:59],predicted_calls[40:59],predicted_puts[40:59])
plot(x= k_list[40:59], y = callprice[40:59],col= "red", ylim = c(0,40000),type
    ↪  = "p", xlab = "Strike Price (K)", ylab = "Call Option Price", main = "
    ↪ BSM Call Price T= 59 Days")
par(new=T)
points(x = k_list[40:59], y = predicted_calls[40:59], col = "blue", pch = 2)
legend("topright",legend = c("True Call Price","BSM Call Price"), col = c("red
    ↪ ","blue"),pch = c(1,2), cex=0.5)

#Plotting T=157 Days for Calls
pred_t157 <- cbind(k_list[60:77],predicted_calls[60:77],predicted_puts[60:77])
plot(x= k_list[60:77], y = callprice[60:77],col= "red", ylim = c(0,40000),type
    ↪  = "p", xlab = "Strike Price (K)", ylab = "Call Option Price", main = "
    ↪ BSM Call Price T= 157 Days")
par(new=T)
points(x = k_list[60:77], y = predicted_calls[60:77], col = "blue", pch = 2)
legend("topright",legend = c("True Call Price","BSM Call Price"), col = c("red
    ↪ ","blue"),pch = c(1,2), cex=0.5)

#Plotting all Call Prices and Predictions
pred_allt <- cbind(k_list,predicted_calls,predicted_puts)
plot(x=k_list, y = predicted_calls, col ="blue", xlab = "Strike Price (K)",
    ↪ ylab = "Call Option Price", main = "True & Predicted Call Price VS
    ↪ Strike Price")
legend("topright", legend = c("BSM Prediction","True Call Price"), col = c("
    ↪ blue","red"),pch = c(1,2), cex = 0.6)
```

```
par(new = T)
points(x=k_list, y=callprice, col = "red", pch = 2)

#Plotting T=10 Days for Puts
plot(x= k_list[1:19], y = putprice[1:19],col= "red", ylim = c(0,40000),type =
    ↪ "p", xlab = "Strike Price (K)", ylab = "Put Option Price", main = "Put
    ↪ Price T= 10 Days")
par(new=T)
points(x = k_list[1:19], y = predicted_puts[1:19], col = "blue", pch = 2)
legend("topright",legend = c("True Put Price","BSM Put Price"), col = c("red
    ↪ ","blue"),pch = c(1,2), cex=0.5)

#Plotting T=31 Days for Calls
plot(x= k_list[20:39], y = putprice[20:39],col= "red", ylim = c(0,40000),type
    ↪ = "p", xlab = "Strike Price (K)", ylab = "Put Option Price", main = "Put
    ↪  Price T= 31 Days")
par(new=T)
points(x = k_list[20:39], y = predicted_puts[20:39], col = "blue", pch = 2)
legend("topleft",legend = c("True Put Price","BSM Put Price"), col = c("red","
    ↪ blue"),pch = c(1,2), cex=0.5)

#Plotting T=59 Days for Calls
plot(x= k_list[40:59], y = putprice[40:59],col= "red", ylim = c(0,40000),type
    ↪ = "p", xlab = "Strike Price (K)", ylab = "Put Option Price", main = "Put
    ↪  Price T= 59 Days")
par(new=T)
points(x = k_list[40:59], y = predicted_puts[40:59], col = "blue", pch = 2)
legend("topleft",legend = c("True Put Price","BSM Put Price"), col = c("red","
    ↪ blue"),pch = c(1,2), cex=0.5)

#Plotting T=157 Days for Calls
plot(x= k_list[60:77], y = putprice[60:77],col= "red", ylim = c(0,40000),type
    ↪ = "p", xlab = "Strike Price (K)", ylab = "Put Option Price", main = "Put
    ↪  Price T= 157 Days")
par(new=T)
points(x = k_list[60:77], y = predicted_puts[60:77], col = "blue", pch = 2)
legend("topleft",legend = c("True Put Price","BSM Put Price"), col = c("red","
    ↪ blue"),pch = c(1,2), cex=0.5)

#Plotting all Call Prices and Predictions
plot(x=k_list, y = predicted_puts, col ="blue", xlab = "Strike Price (K)",ylab
    ↪  = "Put Option Price", main = "True & Predicted Put Price VS Strike
    ↪ Price")
legend("topleft", legend = c("BSM Prediction","True Put Price"), col = c("blue
    ↪ ","red"),pch = c(1,2), cex = 0.6)
```

```
par(new = T)
points(x=k_list, y=putprice, col = "red", pch = 2)

#Checking Put Call Parity

putcallparP <- function(s,T,K,C,r){
  (C-s)+(K*exp(-r*T))
}
putcallparC <- function(s,T,K,P,r){
  (P+s)-(K*exp(-r*T))
}
priceminus_strike <- function(s,K,T,r){
  s-K*exp(-r*T)
}
callminusput <- function(c,p){
  c-p
}
parity_put <- rep(0,length(predicted_calls))
parity_call <- rep(0,length(predicted_puts))
s_kexp <- rep(0,length(predicted_calls))
c_p <- rep(0,length(predicted_calls))
c_p_true <- rep(0, length(predicted_calls))

for (i in 1:length(predicted_puts)) {
  parity_put[i] <- putcallparP(s0_list[[i]], t_list[[i]],k_list[[i]],
     ↪ predicted_calls[[i]],.05)
  parity_call[i] <- putcallparC(s0_list[[i]], t_list[[i]],k_list[[i]],
     ↪ predicted_puts[[i]],.05)
  s_kexp[i] <- priceminus_strike(s0_list[[i]],k_list[[i]],t_list[[i]],.05)
  c_p[i] <- callminusput(predicted_calls[[i]],predicted_puts[[i]])
  c_p_true[i] <- callminusput(callprice[i],putprice[i])
}
cbind(c_p_true,c_p,s_kexp)
plot(x = k_list,y = c_p, col = "red", main = "call minus puts VS strike", xlab
    ↪  = "Strike Price (K)", ylab = "calls minus puts")
par(new=T)
points(x=k_list,y=c_p_true, pch =2, col = "blue")
legend("bottomleft", legend = c("(BSM Call, BSM Put)","(Real Call, Real Put)")
    ↪ , pch = 1:2, col = c("red","blue"), cex = 0.7, text.width = 2, bty = "n
    ↪ ", pt.cex = 1.25)

#calculating implied volatility and graphing volatility smile
imply_vol_call <- rep(0,length(k_list))
imply_vol_put <- rep(0,length(k_list))
for (i in 1:length(k_list)) {
```

```
  imply_vol_call[i] <- bscallimpvol(s0_list[[i]],k_list[[i]],.05,t_list[[i
    ↪ ]],0,callprice[i])
  imply_vol_put[i] <- bsputimpvol(s0_list[[i]],k_list[[i]],.05,t_list[[i]],0,
    ↪ putprice[i])

}
imply_vol_call <- as.numeric(imply_vol_call)
imply_vol_put <- as.numeric(imply_vol_put)
plot(x = k_list, y = imply_vol_call, ylab = "Implied Volatility", xlab = "
   ↪ Strike Price", main = "Implied Volatility VS Strike", col = "red")
legend("topright", legend = c("IV Call Option","IV Put Option"), pch = 1:2,
     col = c("red","blue"), cex=0.6)
par(new=T)
points(x=k_list,y=imply_vol_put, col = "blue", pch = 2)

##LSTM Implementation in R (NOTE YOU NEED TO CREATE A RETICULATE ENVIRONMENT
   ↪ THAT HAS TENSORFLOW TO MAKE THIS RUN)
library(devtools)
library(tensorflow)
library(keras)
library(reticulate)
#use_condaenv('r-reticulate',required = TRUE)

library(timetk)
library(recipes)
library(dplyr)


colnames(BTCClose) <- "closeprice"
btcclose_tib <- tk_tbl(BTCClose[1:1350])

#normalize, scale, and center the closing price tibble
normed <- recipe(closeprice~., btcclose_tib) %>% step_sqrt(closeprice) %>%
   ↪ step_center(closeprice) %>% step_scale(closeprice) %>% prep()

normed_data <- bake(normed, btcclose_tib)


train_x <- normed_data$closeprice[1:800]
train_x <- array(train_x, dim = c(length(train_x),1,1))

train_y <- normed_data$closeprice[2:801]
train_y <- array(train_y, dim = c(length(train_y),1))

test_x <- normed_data$closeprice[800:1349]
```

```
test_x <- array(test_x, dim = c(length(test_x),1,1))

real_y <- normed_data$closeprice[801:1350]
real_y <- array(real_y, dim = c(length(real_y),1))
#Model Construction

#split into 3 subsets, we will train the models on intervals and then compile
   ↪ the predictions

trainx_1 <- train_x[1:200]
trainx_1 <- array(trainx_1, dim = c(length(trainx_1),1,1))

trainx_2 <- train_x[150:349]
trainx_2 <- array(trainx_2, dim = c(length(trainx_2),1,1))

trainx_3 <- train_x[300:499]
trainx_3 <- array(trainx_3, dim = c(length(trainx_3),1,1))

trainx_4 <- train_x[450:649]
trainx_4 <- array(trainx_4, dim = c(length(trainx_4),1,1))

trainy_1 <- train_y[2:201]
trainy_1 <- array(trainy_1, dim = c(length(trainy_1),1,1))
trainy_2 <- train_y[151:350]
trainy_2 <- array(trainy_2, dim = c(length(trainy_2),1,1))

trainy_3 <- train_y[301:500]
trainy_3 <- array(trainy_3, dim = c(length(trainy_3),1,1))

trainy_4 <- train_y[451:650]
trainy_4 <- array(trainy_4, dim = c(length(trainy_4),1,1))

testx_1 <- normed_data$closeprice[201:250]
testx_1 <- array(testx_1, dim = c(length(testx_1),1,1))

testx_2 <- normed_data$closeprice[350:399]
testx_2 <- array(testx_2, dim=c(length(testx_2),1,1))

testx_3 <- normed_data$closeprice[500:549]
testx_3 <- array(testx_3, dim=c(length(testx_3),1,1))

testx_4 <- normed_data$closeprice[650:699]
testx_4 <- array(testx_4, dim=c(length(testx_4),1,1))

realy_1 <- normed_data$closeprice[201:250]
```

```
realy_1 <- array(realy_1, dim = c(length(realy_1),1,1))

realy_2 <- normed_data$closeprice[350:399]
realy_2 <- array(realy_1, dim = c(length(realy_2),1,1))

realy_3 <- normed_data$closeprice[500:549]
realy_3 <- array(realy_3, dim = c(length(realy_3),1,1))

realy_4 <- normed_data$closeprice[650:699]
realy_4 <- array(realy_4, dim = c(length(realy_4),1,1))
#Here we will do k-fold cross validation to determine the best model to run, i
   ↪ .e number of neurons and hidden layers
library(caret)

trctrl <- trainControl(method = "cv", number = 10, savePredictions = TRUE)
fitcheck <- train(factor(closeprice)~., data =btcclose_tib, method = "
   ↪ naive_bayes", trControl = trctrl)
#test/train split



set.seed(1350)
batch <- 10
epoch <- 75

model <- keras_model_sequential() #initiate module
step <- 2
#this is a 2 hidden layer one output layer RNN
nn_model <- model %>%
  layer_lstm(units = 75, input_shape=c(1,1),batch_size = batch,
     ↪ return_sequences = TRUE, stateful = TRUE) %>%
  layer_lstm(units = 50,batch_size = batch,return_sequences = TRUE,activation
     ↪ = "tanh", stateful = TRUE) %>%
  layer_dense(units =1) %>%
  compile(loss = "mse", optimizer = "adam", metric = "accuracy")

model2 <- nn_model %>% fit(x=train_x,y = train_y, batch_size = batch, epochs =
   ↪  epoch, validation_split=0.1, verbose =1 )
plot(model2)

prediction <- model %>% predict(test_x,batch_size = batch)

plot(prediction, type = "l", ylim = c(-1, 3), main = "RNN BTC Prediction VS
   ↪ True BTC Price (Scaled)", ylab = "Scaled BTC Price", xlab = "Observation
   ↪ ",col= "blue")
```

```
lines(x= seq(1:length(prediction)), y=real_y , col= "red")
legend("topleft", legend = c("RNN Predicted BTC Price", "True BTC Price"), col
    ↪  =c("blue","red"), lty = 1:1, cex = 0.5,pt.cex=2)

MAE(as.numeric(prediction), as.numeric(real_y))
#this is a 3 hidden layer one ouput layer RNN
model <- keras_model_sequential() #initiate module
nn_model2 <- model %>%
  layer_lstm(units = 100, input_shape=c(1,1),batch_size = batch,
     ↪ return_sequences = TRUE, stateful = TRUE) %>%
  layer_lstm(units = 75,input_shape = c(1,1), batch_size = batch,
     ↪ return_sequences = TRUE,activation = "softmax", stateful = TRUE) %>%
  layer_lstm(units = 50,batch_size = batch,return_sequences = TRUE,activation
     ↪ = "tanh", stateful = TRUE) %>%
  layer_dense(units =1) %>%
  compile(loss = "mse", optimizer = "adam", metric = "accuracy")

model3 <- model %>% fit(x = train_x,y = train_y, batch_size = batch, epochs =
    ↪ epoch, validation_split=0.1, verbose =1)
prediction2 <- model %>% predict(test_x, batch_size = batch)

plot(prediction2, type = "l", ylim = c(-1, 3), main = "RNN BTC Prediction VS
    ↪ True BTC Price (Scaled)", ylab = "Scaled BTC Price", xlab = "Observation
    ↪ ",col= "blue")
lines(x= seq(1:length(prediction2)), y= real_y, col= "red")
legend("topleft", legend = c("RNN Predicted BTC Price", "True BTC Price"), col
    ↪  =c("blue","red"), lty = 1:1, cex = 0.5)


#4 hidden layer model with one output layer
model <- keras_model_sequential() #initiate module
nn_model3 <- model %>%
  layer_lstm(units = 100, input_shape=c(1,1),batch_size = batch,
     ↪ return_sequences = TRUE, stateful = TRUE) %>%
  layer_lstm(units = 75,input_shape = c(1,1), batch_size = batch,
     ↪ return_sequences = TRUE,activation = "softmax", stateful = TRUE) %>%
  layer_lstm(units = 50,batch_size = batch,return_sequences = TRUE,activation
     ↪ = "tanh", stateful = TRUE) %>%
  layer_lstm(units = 25,batch_size = batch,return_sequences = TRUE,activation
     ↪ = "tanh", stateful = TRUE) %>%
  layer_dense(units =1) %>%
  compile(loss = "mse", optimizer = "adam", metric = "accuracy")

model4 <- model %>% fit(x = train_x,y = train_y, batch_size = batch, epochs =
    ↪ epoch, validation_split=0.1, verbose =1)
```

```
prediction3 <- model %>% predict(test_x, batch_size = batch)

plot(prediction3, type = "l", ylim = c(-1, 3), main = "RNN BTC Prediction VS
    ↪ True BTC Price (Scaled)", ylab = "Scaled BTC Price", xlab = "Observation
    ↪ ",col= "blue")
lines(x= seq(1:length(prediction3)), y= real_y, col= "red")
legend("topleft", legend = c("RNN Predicted BTC Price", "True BTC Price"), col
    ↪  =c("blue","red"), lty = 1:1, cex = 0.5)

##GRU NN implementation in Python (NOTE YOU NEED TO SET UP TENSORFLOW
    ↪ ENVIRONMENT TO RUN)
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 22 12:53:19 2021

@author: farus
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
import yfinance as yf
from yahoofinancials import YahooFinancials

import tensorflow as tf
from datetime import timedelta
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold, TimeSeriesSplit, GridSearchCV

import keras.initializers
from keras.layers import Dense, Layer, GRU, Dropout, Activation
from keras.models import Sequential
from keras.models import load_model
from keras.regularizers import l1, l2
from keras.callbacks import EarlyStopping
from keras.wrappers.scikit_learn import KerasRegressor
from keras import regularizers


raw_data = yf.download('BTC-USD',
                 start = '2018-01-01',
                 end = '2021-09-23',
```

```
                  progress = False)

raw_data.head()

btc_close = raw_data["Close"]

adf, p ,usedlag, nobs, cvs, aic = sm.tsa.stattools.adfuller(btc_close)
adf_results_string = 'ADF: {}\np-value: {},\nN: {}, \ncritical values: {}'
print(adf_results_string.format(adf, p, nobs, cvs))


#our p value and ADF value show that we cannot reject the null hypothesis so
   ↪ the data is non-stationary. NOTE however that the returns themselves
   ↪ have some stationarity!

#taking the pacf and plotting
#pacf = sm.tsa.stattools.pacf(btc_close, nlags=30)

#T = len(btc_close)
#sig_test = lambda tau_h: np.abs(tau_h) > 2.58/np.sqrt(T)

#for i in range(len(pacf)):
   #if sig_test(pacf[i]) == False:
     # n_steps = i-1
      #print('n_steps set to', n_steps)
     # break

#this tells us 4 is the max lag until the pacf says there is no correlation. 4
   ↪  lags is very heavy computationally however

#plt.plot(pacf, label='pacf')
#plt.plot([2.58/np.sqrt(T)]*30, label='99% confidence interval (upper)')
#plt.plot([-2.58/np.sqrt(T)]*30, label='99% confidence interval (lower)')
#plt.xlabel('number of lags')
#plt.legend();

#our pacf plot shows that the optimum lag is closer to two than four, to
   ↪ reduce computational intensity we will assume lag of 2 will work fine
   ↪ enough for our desires


#split into train and test
train_ratio = 0.8
split = int(len(btc_close)*train_ratio)
```

```
train_df = btc_close.iloc[:split]
test_df = btc_close.iloc[split:]


#scale and standardize
mu = np.float(train_df.mean())
sd = np.float(train_df.std())

standardize = lambda x: (x-mu)/sd

train_df = train_df.apply(standardize)
test_df = test_df.apply(standardize)

#We create overlapping time subintervals to have one-step ahead time
   ↪ prediction. Intuitively, the formula for y_t is dependent on y_t-1 so
   ↪ the subsequences create this correlation by overlapping the subintervals

def lag_my_features(df, n_steps, n_steps_ahead):
   lag_list = []
   for lag in range(n_steps+n_steps_ahead-1,n_steps_ahead-1, -1):
      lag_list.append(df.shift(lag))
      lag_array = np.dstack([i[n_steps_ahead+n_steps-1:] for i in lag_list])

      lag_array = np.swapaxes(lag_array,1,-1)
      return lag_array

#apply lag features to our test and train set
n_steps_ahead = 10 # forecasting horizon
n_steps =4

x_train = lag_my_features(train_df, n_steps, n_steps_ahead)
y_train = train_df.values[n_steps + n_steps_ahead-1:]
y_train_timestamps = train_df.index[n_steps + n_steps_ahead-1:]


x_test = lag_my_features(test_df, n_steps, n_steps_ahead)
y_test = test_df.values[n_steps + n_steps_ahead-1:]
y_test_timestamps = test_df.index[n_steps + n_steps_ahead-1:]
print([tensor.shape for tensor in (x_train, y_train, x_test, y_test)])

#transpose to amke the dimensions match for model fitting

x_test = np.transpose(x_test)
x_train = np.transpose(x_train)
```

```
print([tensor.shape for tensor in (x_train, y_train, x_test, y_test)])


model = Sequential()
model.add(GRU(50,return_sequences=True, input_shape = (x_train.shape[1],1)))
model.add(Dropout(0.2))
model.add(GRU(100, return_sequences = False))
model.add(Dropout(0.2))
model.add(Dense(1, activation = "linear"))
model.compile(loss = 'mse', optimizer = 'adam')

model.fit(x_train, y_train, epochs=50, batch_size=32)
in_samp_pred = model.predict(x_train)
in_samp_compare = [[y_train],[in_samp_pred]]
prediction = model.predict(x_test)
compare_tens = [[y_test],[prediction]]

in_samp_loss = mean_squared_error(y_train, in_samp_pred)
out_samp_loss = mean_squared_error(y_test, prediction)

ind = list(range(0,259))
ind2= list(range(0,1073))
fig, ax = plt.subplots()
ax.plot(ind, y_test, label = 'True BTC Price')
ax.plot(ind, prediction, label = 'Predicted BTC Price')
ax.legend(loc='upper right')
plt.title('GRU Predicted BTC Prices VS True BTC Prices (Out-of-Sample)')
plt.xlabel('Index of Days (01-2021 to 09-2021)')
plt.ylabel('Scaled BTC Closing Price')
plt.show()

fig, ax = plt.subplots()
ax.plot(ind2,y_train , label = 'True BTC Price')
ax.plot(ind2, in_samp_pred, label = 'Predicted BTC Price')
ax.legend(loc='upper right')
plt.title('GRU Predicted BTC Prices VS True BTC Prices (In Sample)')
plt.xlabel('Index of Days (01-2018 to 01-2021)')
plt.ylabel('Scaled BTC Closing Price')
plt.show()
```