Comparing the Performance of Volatility Models

Quantitative Analysis Report

Anthony Li

August 2023

Contents

1 Introduction

1.1 Data

Our client primarily trades US equity derivatives and since the focus of the report is quite experimental, we have decided to work with the SP500 index due to the availability of daily market data.

We will be attempting to forecast the implied volatility of the SP500 between 06-02-2018 and 31-06-2023. This period allows us to investigate the volatility models in bullish markets as well as periods of high volatility and financial crisis. Additionally, the earliest available data available for the SP500 Twitter Sentiment Index is 06-02-2018 and I believe an investor sentiment variable would be a significant feature for the artificial neural network.

Our target data will be the CBOE Volatility Index, VIX. "The VIX Index is a calculation designed to produce a measure of constant, 30-day expected volatility of the U.S. stock market, derived from real-time, mid-quote prices of SP 500® Index (SPX℠) call and put options" (VIX Volatility Products, Accessed 07 July 2023). We will import the VIX adjusted closing price directly from Yahoo Finance.

GARCH(p,q)
To calibrate a Generalized Autoregression Condition Heteroskedastic (GARCH) model, we will need to obtain the daily returns for the SP500. We can then use the daily returns to calibrate the GARCH model via maximum likelihood estimation (MLE). We will use the Yahoo Finance API to directly import SP500 adjusted closing prices and calculate daily returns as

$$r_t = \frac{P_t - P_{t-1}}{P_{t-1}}. \tag{1}$$

Neural Network Approach
Having obtained the target variable, the VIX index, we must now decide on a list of significant predictive features. As discussed in M. Malliaris and L. Salchenberger work on 'Using neural networks to forecast the SP 100 implied volatility', including lagged variables as a predictor can significantly improve model performance and remedy autocorrelation in residuals. I have included up to 3-day lags of implied volatility alongside other macroeconomic factors.

Eghbal Rahmikia and Ser-Huang Poon, as discussed in Machine Learning for Realised Volatility Forecasting, investigated the use of LSTMs for realized volatility forecasting and included 147 input variables extracted from limit order book (LOB) data. They showed that the machine learning techniques significantly outperformed HAR class of models for the out-of-sample forecasting period. Unfortunately, we could not obtain the LOB dataset to include in our research but also the inclusion of many variables is far too computationally expensive for the purpose of this report. We resorted to using other daily data that were free to access.

Input Variables
21-day lagged realized volatility
Volume
Daily Returns
SP500 Twitter Sentiment Index
3-month treasury yield
1-day lagged Implied volatility
2-day lagged Implied volatility
3-day lagged Implied volatility

Table 1: List of Input Variables

We believed that including some macroeconomic features will improve the forecast of the neural network as macroeconomic indicators have previously been shown to predict financial crises relatively accurately. As volatility increases significantly during periods of the financial crisis, these macroeconomic factors can potentially help forecast periods of high volatility. We decided to use the 3-month treasury yield as daily data was easily accessible.

1.2 Implied Volatility and VIX

The implied volatility (IV) refers to the market's expectation of future volatility. It is derived using the prices of option contracts on an asset. To understand IV, we must first understand the Black-Scholes (B-S) options pricing model, developed by Fischer Black, Myron Scholes, and Robert Merton in the early 1970s. B-S model assumes that the price of an asset follows a geometric brownian motion with constant drift and diffusion.

One of the biggest problems that the B-S model faces is that it assumes constant volatility whereas volatility varies through time. Instead, the model can be used to solve for the IV based on the observed option prices. The asset price dynamics (Black, Scholes, and Merton 1973) is given by

$$dS = rSdt + \sigma SdW \tag{2}$$

and the closed-form solution for call and put option prices are

$$C = S \cdot N(d_1) - K \cdot e^{-rT} \cdot N(d_2) \tag{3}$$

$$P = -S \cdot N(-d_1) + K \cdot e^{-rT} \cdot N(-d_2) \tag{4}$$

where

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} \tag{5}$$

$$d_2 = d_1 - \sigma\sqrt{T} \tag{6}$$

and
- C is the price of the call option
- P is the price of the put option
- S is the current price of the stock
- K is the option's strike price
- σ is the volatility
- r is the risk-free interest rate
- T is the time to expiration of the option.

Following this, we can express the IV of a call option as $\sigma_{Call}^{IV} = \sigma(C, S, K, r, T)$, a function of C, S, K, r, and T. Without loss of generality, the same holds true for put options. The VIX, also known as the 'fear index' is a widely used measure of market volatility. The precise calculation of the VIX is proprietary however the general methodology is

1. **Selecting options**: the CBOE selects a range of options with different strikes and expiry dates.

2. **Calculating IV**: the IV for each option is calculated using the B-S model.

3. **Weighted Average**: a weighted average of the IVs is calculated according to the moneyness of the options.

4. **Interpolation**: the IVs are interpolated to estimate a complete IV curve.

5. **VIX Calculation**: the VIX is calculated as the weighted average across all of the squared IVs.

1.3 Properties of Returns and Implied Volatility

Returns
Many popular models in finance assume that returns follow a Gaussian distribution due to the attractive statistical inference that comes with it. However, returns are not normal and have much fatter tails. Some of the stylized facts about financial return include zero mean, no linear autocorrelation, volatility

clustering, unconditional heavy tails, gain/loss asymmetry, conditional heavy tails, and aggregational gaussianity. We will not explore these properties as the report will be focused on forecasting IV but it is important to consider these properties when building models.

Implied Volatility
Since the Black-Scholes assumes the normality of returns and constant volatility, this leads to some unique properties of IV. Properties of IV include

- **Market Expectations**: since IV is calculated based on observed option prices, the IV is the market's expectations of future price volatility.

- **Forward-Looking**: IV looks ahead since the market anticipates price fluctuations up until expiry. This differs from historical volatility which only considers past price movements.

- **Volatility Skew**: the IV is not uniform across all strikes and expiry dates. This is a reflection of market sentiment and can be explained by investors being risk-averse. It is also the result of an imperfect model as B-S option pricing model does not perfectly price option contracts.

1.4 History of Volatility Models

Stochastic volatility models go as far back as 1973, when the B-S option pricing model was developed. Newer models have evolved over time to better predict the fluctuations in financial returns and accurately price financial derivatives. Here are a few popular examples...

1. ARCH Model (1982): Engle suggested that volatility itself could be time-varying and introduced the concept of conditional volatility. The ARCH(p) model (Engle 1982) is given by

$$\epsilon_t | \psi_{t-1} \sim N(0, h_t) \tag{7}$$

$$h_t^2 = \alpha_0 + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2 \tag{8}$$

where

- h_t is the conditional volatility at time t
- ϵ_t is the asset return at time t
- α_i are the ARCH parameters

2. GARCH Model (1986), an extension to ARCH: Bollerslev suggested the inclusion of lagged conditional volatility terms allowed the model to capture lagged relationships. More details on GARCH will be included in section 2.1.

3. Heston Model (1993): Heston's model is an extension of the B-S model. It aims to capture changing volatility in financial markets. Heston suggested that the volatility itself should have its own stochastic process and evolves over time alongside the asset price. The dynamics of the model (Heston 1993) is given by

$$dS_t = \mu S_t dt + \sqrt{v_t} S_t dW_t^S \tag{9}$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t^v \tag{10}$$

where
- S_t is the stock price at time t
- μ is the average expected return of the asset
- v_t is the instantaneous volatility of the asset
- W_t^S is the asset's brownian motion
- W_t^v is the volatility's brownian motion
- κ is the rate of mean-reversion
- θ is the long-term average volatility
- σ is the volatility of the volatility

Other types of volatility models include jump-diffusion type models and rough volatility models. We will not be discussing these in this report.

2 Methodology

2.1 GARCH

The GARCH model is among the most commonly used volatility models and many extensions have been introduced, such as GJR-GARCH, EGARCH, and DCC GARCH. In this report, we will be using GARCH(1,1) to forecast conditional volatility. The specification of a GARCH(p,q) model (Bollerslev 1986) is given by

$$\epsilon_t|\psi_{t-1} \sim N(0, h_t) \tag{11}$$

$$h_t = \alpha_0 + \sum_{i=1}^{q} \alpha_i \epsilon_{t-i}^2 + \sum_{i=1}^{p} \beta_i h_{t-i}^2 \tag{12}$$

where

$$p \geq 0,\ q > 0$$

$$\alpha_0 > 0,\ \alpha_i \geq 0,\ i = 1, .., q$$

$$\beta_i \geq 0, \; i = 1, ..., p$$

and
- h_t is the conditional volatility at time t
- ϵ_t is the asset return at time t
- α_i and β_i are the GARCH parameters.

MLE is the most common practice for fitting a GARCH model using returns. It involves selecting parameters that maximise the probability of observing the sample given the underlying assumptions. Before fitting the model, we will split the return data into an 80:20 training and testing set ratio. We will use the training data to fit a GARCH(1,1).

	Coefficient	Std Error	p-value
ω	3.7060e-06	9.249e-12	0.000
α	0.2000	3.675e-02	5.248e-08
β	0.7800	2.845e-02	1.798e-165

Table 2: GARCH Model Summary

The calibrated model is

$$\sigma_t^2 = 0.0000037060 + 0.2000 r_{t-1}^2 + 0.8000 \sigma_{t-1}^2 \tag{13}$$

Using an iterative method, we can forecast 1-day forward conditional volatility for the test data. Using these forecasts, we can conduct out-of-sample performance analysis by comparing it to the observed IV.

In Figure 1, we can see that the GARCH(1,1) model underestimates the IV for the majority of the period however overshoots during volatility shocks. A general drawback to structural models, such as GARCH, is that it relies on too many assumptions and does not allow for flexibility to fit the data. Structural models are useful for understanding the effects of variables on the behaviour of volatility however non-structural models tend to perform better for forecasting purposes as we will see.

2.2 Artificial Neural Networks

Artificial neural networks date back to the 1940s when it was mostly theoretical due to a lack of computational power. As computing power improved and data became more granular, neural networks also increased in popularity amongst academia, and more complex architectures were developed.

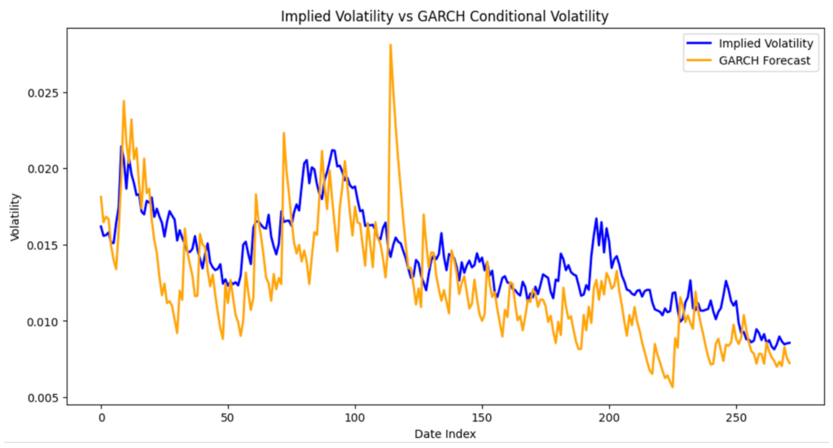

Figure 1: GARCH: Forecast vs Observed Values (OOS)

Zhang and Gupta (p.1340, 2003) explained that an artificial neural network is composed of "an input layer, an output layer, and one or more hidden layers". These layers consist of a number of neurons that are interconnected to every neuron in neighbouring layers. These multiple layers map an input tensor into an output tensor through multiple weights, biases, and activation functions. Artificial neural networks are trained by minimising a loss function via back-propagation, more on this in section 2.4.

One of the key aspects of artificial neural networks is selecting the correct activation function. This is the transformation that produces the output tensor for each layer. Here is a list of activation functions we will use

- **Linear**: $f(x) = x$

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

- **Tanh**: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

2.3 Long-Short Term Memory (LSTM)

We will use more complex architectures to handle sequential data. LSTMs are a type of recurrent neural network (RNN). RNN architectures are capable of handling sequential data however standard RNNs face the vanishing gradient problem. LSTMs are a more complex architecture that controls the flow of old memory and new inputs which remedies the vanishing gradient problem. They consist of a series of gates with activation functions to control the flow of information. Here are the gates and states involved in an LSTM layer (Hochreiter and Schmidhuber 1997)

1. Cell State (also known as memory cell): The Cell States contain information from the past LSTM cells. It captures and remembers long-term relationships in the data. This is protected by the Forget Gate and updated by the Input Gate.

2. Hidden State (also known as the output of the LSTM cell): The Hidden State captures the output of previous LSTM cells and is used in the forget gate, input gate, and output gate to generate a new updated Hidden State as the output of the LSTM cell.

3. Forget Gate: This gate controls the amount of information that is kept from the Cell State.

4. Input Gate: The input gate controls the amount of new information added to the cell state.

5. Output Gate: This gate decides the amount of information from the Cell State that should be used in generating the output of the LSTM cell (also known as the Hidden State).

2.4 Parameter Optimization

We will be constructing a neural network consisting of 2 LSTM layers followed by 2 dense layers. Before we fit the model, the input variables must all be standardized. This ensures equalized scaling, faster convergence, and avoiding the vanishing gradient problem. Standardizing now will also be crucial during the regularization step which penalises parameters with large magnitudes. We will be scaling using Min-Max scaling,

$$X^{scaled} = \frac{X - min(X)}{max(X) - min(X)} \tag{14}$$

We sought to split the data into training and test sets so that we can investigate the out-of-sample performance of the model. We will use an 80:20 split for training and testing respectively. It is important to emphasize that we must not shuffle the data as we are dealing with sequential data with temporal relationships.

Since the target variable is continuous, we will use the Mean Squared Error (MSE) as our error function. The goal is to identify the set of parameters that minimise the error function via backpropagation. To improve the efficiency of the parameter optimisation, we will split the data into batch sizes of 32 and

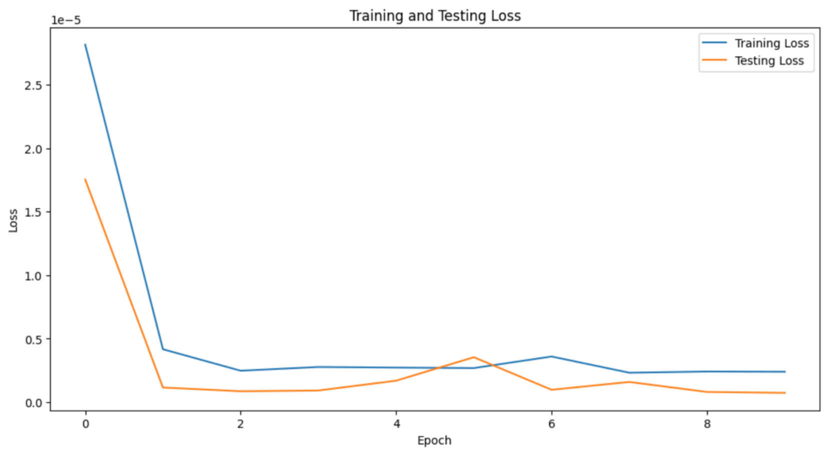

Figure 2: Training vs Testing Error for each Epoch

iterate for 10 epochs. This will increase the speed of convergence and reduce the computational cost by only using a subset of the training data for each iteration. The mean squared error is given by

$$J(W) = MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{15}$$

where
- n is the sample size.
- y_i is the observed value of the target variable.
- $\hat{y}_i$ is the predicted value of the target variable.

Backpropagation is an algorithm used to train artificial neural networks. It involves adjusting the weights and biases of each neuron after each iteration with the goal of minimising the error function. The size of each update is determined by the learning rate, η, which is a hyperparameter. The learning rate can either be optimised or pre-defined carefully. Here are the generalized formulas to show how the weights, $\hat{w}_k$, are updated

$$J(\hat{w}) = \frac{1}{N}\|\hat{X}\hat{w} - Y\|_2^2 \tag{16}$$

$$\hat{w}_{k+1} \leftarrow \hat{w}_k - \eta\nabla_{\hat{w}} J(\hat{w}_k) \tag{17}$$

We can see that, in Figure 2, the training and test loss decreases after each Epoch. The test error starts to revert after the 3rd epoch which may be a sign of overfitting. An early stop can be introduced to terminate the iteration early to avoid overfitting.

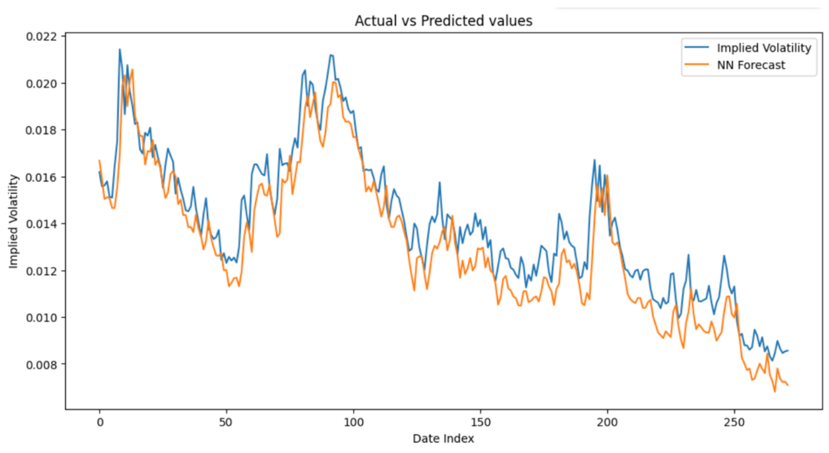

Figure 3: Base Model Forecast vs Observed Values (OOS)

The artificial neural network clearly outperforms the GARCH(1,1) model in out-of-sample (OOS) forecasting, as shown in Figure 3. However, we can clearly see a lag between the forecast IV and the observed IV. The performance also decreases after date index 125 which shows a sign of model decay. A solution could be to optimise the model more frequently but could also be computationally expensive as we increase the complexity of the model.

2.5 Hyperparameter Optimization

When building a neural network, the number of neurons should be carefully selected. One way to do this is by optimizing it using a GridSearch method to identify the hyperparameters which minimise the cross-validation (CV) loss. CV involves splitting the training set into K equal-sized sets and picking one set out for validation. Since we are dealing with sequential data, we must use a rolling window method instead.

For example, the data can be split into 5 equal-sized sets (in order). The first set can be used to train the model and then the model is tested on the second set. This can roll forwards such that the second set is now the training set and so on. Here are the values of hyperparameters we will be searching

- LSTM 1: 32, 64, 128
- LSTM 2: 32, 64, 128
- Dense Layer: 16, 32, 64

The optimal parameters for LSTM 1, LSTM 2, and the Dense layer are 64, 64, and 32 respectively. In Figure 4, We can see that the updated forecast

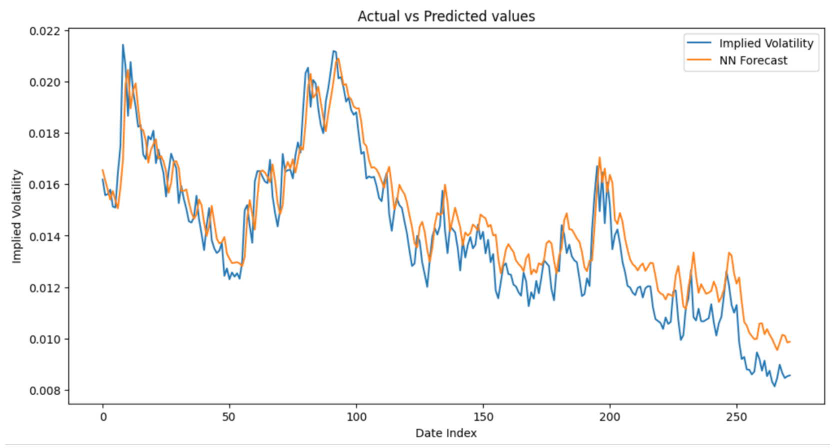

Figure 4: Hyperparameter Optimized Forecast vs Observed Values (OOS)

IV is closer to the observed values compared to the base model. However, the performance clearly drops after Date Index 100 again. Similarly to before, this could represent a decay of the model. Frequently optimising parameters and hyperparameters is very time-consuming therefore the frequency must be carefully selected.

2.6 Regularization

When fitting neural networks, it can be easy to over-fit data since the model allows for non-linear relationships between input variables and output variables. The goal of a machine learning model is to be able to fit the relationship between input variables and output variables without fitting the noise that occurs by chance. By fitting these residuals, the out-of-sample forecasting will be influenced negatively and hence poorly forecast unseen data.

Regularization is a method to prevent over-fitting and hence improve the out-of-sample performance of a model. Most regularization methods involve adding a penalty term to the loss function of a model which discourages overly complex models. The most common regularization techniques are Lasso (L1), Ridge (L2), dropout, elastic net, and early stopping. We will be using Lasso in our work as it can drive some coefficients to exactly 0 rather than just shrinking. The Lasso error term (Tibshirani 1996) is given by

$$\lambda \sum_{j=1}^{p} |\beta_j| \tag{18}$$

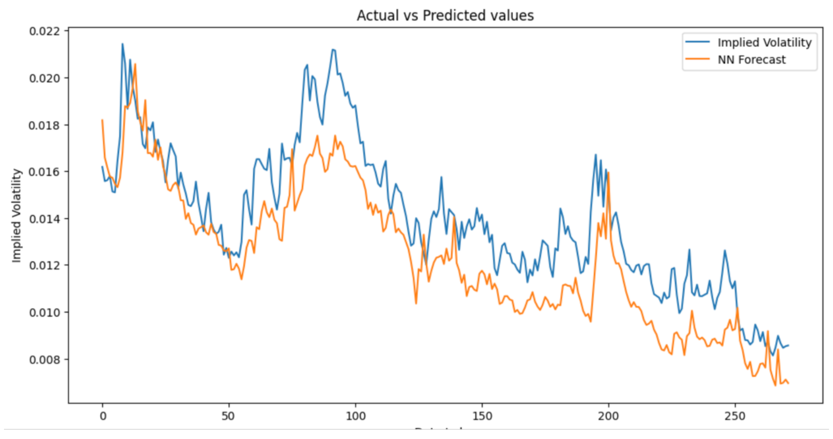

Figure 5: Regularization Forecast vs Observed Values (OOS)

Note: we have used the optimised hyperparameters identified in section 2.5.

In Figure 5, it is evident that the performance drops significantly when introducing the lasso regularization. The forecast consistently underestimates the IV which can be explained by the shrinkage of the parameters towards 0. Although it is good practice to implement regularization techniques for better OOS performance, it was not beneficial in this case.

3 Results

3.1 Results and Analysis

We report the out-of-sample performance metrics for each of the models covered. The performance metrics will be root mean squared error (RMSE), R-squared, and adjusted R-squared.

	GARCH	Base Model	Hyperparameter	Regularization
RMSE_{IS}	0.004641	0.001544	0.001474	0.001782
RMSE_{OOS}	0.002976	0.001311	0.001109	0.002023
R^2_{OOS}	0.01374	0.8088	0.8632	0.5442
$\overline{R}^2_{OOS}$	0.01009	0.8030	0.8590	0.5303

Table 3: Table of OOS Performances

From Table 3, it is apparent that the hyperparameter-optimized artificial neural network performs the best across all 3 metrics and all 3 neural network models outperform GARCH significantly. This is expected as the base neural network model hyperparameters are just one of the combinations included within the grid search. It is important to note that fitting a neural network involves setting randomized initial parameters therefore each execution of the code will produce slightly different results. The optimal solutions are local optima and not necessarily global optima. We have presented the results obtained from our execution of the code and based our conclusions on this.

Optimizing the number of neurons in each layer with 3 possible values reduced the OOS RMSE from 0.001311 to 0.001109 (a 16.9 percent decrease) and increased the OOS R-squared and Adjusted R-Squared. There are many more hyperparameters that can be optimised to further improve the model. For example, the learning rate, number of layers, epochs, batch size, and activation functions are all hyperparameters.

3.2 Conclusions

The correct use of machine learning models can significantly improve implied volatility forecasting by optimizing the parameters and hyperparameters. Although artificial neural networks are much better for forecasting implied volatility, it is difficult to establish any causality and understand the relationships between the input variables and the implied volatility.

Although we cannot confidently predict IV, the work in this report can be enhanced further by optimizing the other hyperparameters as mentioned earlier but comes with a high computational cost. We were limited by the computational power and access to paid data services which can potentially improve the model.

3.3 Improvements and Modifications

The use of artificial neural networks certainly has the potential to forecast implied volatility however there is a clear lagged relationship between the forecast values and the observed values. It can be interesting to investigate how neural networks can model the entire implied volatility curve today instead of forecasting future VIX. This can help identify incorrectly priced options if we are able to correctly model the implied volatility curve.

Please note that the word count is 3087 due to the need to explain artificial neural networks and the correct procedure for fitting the model.

4 Bibliography

[1] Black, F, Scholes, M, and Merton, R 1973, 'The Pricing of Options and Corporate Liabilities', Journal of Political Economy, 81(3), 637-654.

[2] Engle, R 1982, 'Autoregressive Conditional Heteroskedasticity with Estimates of the Variance of United Kingdom Inflation', Econometrica, 50(4), 987-1008.

[3] Bollerslev, T 1986, 'Generalized Autoregressive Conditional Heteroskedasticity', Journal of Econometrics, 31(3), 307-327.

[4] Malliaris M, Salchenberger L 1996, 'Using neural networks to forecast the SP 100 implied volatility', Neurocomputing 10, 188-191.

[5] Tibshirani, R 1996, 'Regression Shrinkage and Selection via the Lasso', Journal of the Royal Statistical Society: Series B (Methodological), 58(1), 267-288.

[6] Heston, S 1993, 'A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options', The Review of Financial Studies, 6(2), 327-343.

[7] Hochreiter, S, and Schmidhuber, J 1997, 'Long Short-Term Memory', Neural Computation, 9(8), 1735-1780.

[8] Hagan, P, Kumar, D, Lesniewski, A, and Woodward, D 2002, 'Managing Smile Risk', Wilmott Magazine, 84-108.

[9] Eghbal Rahmikia and Ser-Huang Poon 2023, Machine Learning for Realised Volatility Forecasting.

[10] VIX Volatility Products, https://www.cboe.com/tradable products/vix/, CBOE, 2023.

5 Appendix A: Code

```
import numpy as np
import pandas as pd
import yfinance as yf

# Downloading SP500 adjusted close prices
data = yf.download("^GSPC", start='2018-01-05', end='2023-07-01')['Adj Close']

# Downloading Implied Volatility adjusted close prices
Implied_Vol = yf.download("^VIX", start='2018-02-01', end='2023-07-01')['Adj Close']
```

```
# Downloading 3 Month Treasury Yield data
Treasury_Yield = yf.download("^IRX", start='2018-02-06', end='2023-07-01')['Adj Close']

# Downloading SP500 Volume data
Volume = yf.download("^GSPC", start='2018-02-06', end='2023-07-01')['Volume']

# Access files from Google Drive - all code was done on Google Colaboratory
from google.colab import drive
drive.mount('/content/gdrive')

import os
os.chdir("./gdrive/My Drive/Colab Notebooks/Applied Project/")

# Importing .csv file containing the S&P500 Twitter Sentiment Index
Sentiment_Index = pd.read_csv("Sentiment_Index.csv", index_col=0)

# Changing annualized percentage Implied Volatility to daily format
Implied_Vol = Implied_Vol/(100*np.sqrt(252))

# Calculating returns
returns = data/data.shift(1) - 1

returns = returns.iloc[1:]
returns = pd.DataFrame(returns.values)

# Calculating 21-day lagged realized volatility of returns
lagged_realized_volatility = returns.rolling(window=21).std().dropna()

# Removing initial 20 days as we needed it for calculating the realized volatility only
returns = returns.iloc[20:]

# Spliting into 80:20 train and test datasets
returns_train = returns[:round(0.8*len(returns))]
returns_test = returns[round(0.8*len(returns)):]

# Originally extracted 3 days earlier for the lagged input for Neural Network Model
Implied_Vol_train = Implied_Vol[3:round(0.8*len(returns))+3]
Implied_Vol_test = Implied_Vol[round(0.8*len(returns))+3:]

from arch import arch_model

# Defining and fittting a GARCH(1,1) model
garch_model = arch_model(returns_train, vol='Garch', p=1, q=1)

garch_results = garch_model.fit()
```

```
# Print the summary of the model
print(garch_results.summary())

# Extracting the parameters of the GARCH model
parameters = garch_results.params

# Extracting the last conditional volatility from the Training set for forecasting
last_volatility = garch_results.conditional_volatility.values[-1]

# Defining a list to store conditional volatilities for the Test set
garch_forecasted_volatilities = np.array([last_volatility])

# Using the returns in the Test set to forecast the conditional volatilities
for i in range(len(returns_test[:-1])):
    forecasted_volatility = np.sqrt(parameters['omega'] +
                                    parameters['alpha[1]'] * returns_test.iloc[i]**2 +
                                    parameters['beta[1]'] * last_volatility**2)
    # Storing the forecasted volatility
    garch_forecasted_volatilities = np.append(garch_forecasted_volatilities,
                                              forecasted_volatility)

    # Updating the 'last_volatility' for the next iteration
    last_volatility = forecasted_volatility

# Calculating OOS-RMSE
GARCH_MSE = np.mean((Implied_Vol_test - garch_forecasted_volatilities) ** 2)
GARCH_RMSE = np.sqrt(GARCH_MSE)
print("The Root Mean Squared Error for the GARCH model is: ", GARCH_RMSE)

# Calculating R-Squared and Adjusted R-Squared
from sklearn.metrics import r2_score

r_squared = r2_score(Implied_Vol_test, garch_forecasted_volatilities)

# Calculating adjusted R-squared
n = len(Implied_Vol_test)
p = 1
adjusted_r_squared = 1 - (1 - r_squared) * (n - 1) / (n - p - 1)

print('The OOS R-squared is: ', r_squared)
print('The OOS Adjusted R-squared is: ',adjusted_r_squared)

import matplotlib.pyplot as plt

# Plotting graph to compare actual IV against GARCH conditional volatilities for test set
```

```
x = range(len(Implied_Vol_test))

y1 = Implied_Vol_test
y2 = garch_forecasted_volatilities

plt.figure(figsize=(12, 6))

plt.plot(x, y1, label='Implied Volatility', color='blue', linestyle='-', linewidth=2)

plt.plot(x, y2, label='GARCH Forecast', color='orange', linestyle='-', linewidth=2)

plt.xlabel('Date Index')
plt.ylabel('Volatility')
plt.title('Implied Volatility vs GARCH Conditional Volatility')
plt.legend()

plt.show()

# Defining a dataframe to store all of the input variables and target variable
data = {
    # Excluding last day as we only want to forecast 1-day ahead Implied Volatility
    'Yield': Treasury_Yield.values[:-1],
    'Volume': Volume.values[:-1],
    'returns': returns.squeeze().values[:-1],
    'realized_vol': lagged_realized_volatility.squeeze().values[:-1],
    'Sentiment': Sentiment_Index.squeeze().values[:-1],
    # Carefully extracting the correct lags of IV
    'Implied_Vol_LAG1': Implied_Vol.values[3:-1],
    'Implied_Vol_LAG2': Implied_Vol.values[2:-2],
    'Implied_Vol_LAG3': Implied_Vol.values[1:-3],
    'Implied_Vol': Implied_Vol.values[4:]
}

df = pd.DataFrame(data)
df.head()

# Rescaling the input variables
from sklearn.preprocessing import MinMaxScaler

# Scaling by Min-Max
scaler = MinMaxScaler()

# Scaling the data ready for Neural Network input Tensors
scaled_data = scaler.fit_transform(df.iloc[:,:-1])

# Create a new DataFrame containing the scales values
```

```
scaled_df = pd.DataFrame(scaled_data, columns=df.columns[:-1])

# Creating the first LSTM model with pre-defined hyperparameters

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, LSTM, Dense
from sklearn.model_selection import train_test_split


# Pre-defining hyperparameters
d_1 = 128 # output dimension of the first LSTM layer
d_2 = 128 # output dimension of the second LSTM layer
K = 1 # Output dimension of the last layer
N_epochs = 10
N_batch_size =32

X = scaled_df.iloc[:, :].values
y = df.iloc[:, -1].values

# Splitting data into Training and Test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)

# Reshaping X into suitable LSTM input tensors
X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])

# Creating a neural network with 2 LSTM layers and 2 dense layer
model = Sequential([
    LSTM(d_1, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True),
    LSTM(d_2),
    Dense(32, activation = 'linear'),
    Dense(K, activation = 'linear')
])

# Compiling the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Training the model on the Training dataset
history = model.fit(X_train, y_train, epochs=N_epochs, batch_size=N_batch_size, validation_

# Predicting the OOS Implied Volatility using NN model
NN_predictions = model.predict(X_test)

# Calculating OOS RMSE
NN_mse = np.mean((y_test - NN_predictions.flatten()) ** 2)
NN_rmse = np.sqrt(NN_mse)
```

```
print("The Root Mean Squared Error for the Neural Network model is: ", NN_rmse)

NN_predictions = model.predict(X_test)

NN_r_squared = r2_score(y_test, NN_predictions)

n = len(y_test)
p = 8
NN_adjusted_r_squared = 1 - (1 - NN_r_squared) * (n - 1) / (n - p - 1)

print('The OOS R-squared is: ', NN_r_squared)
print('The OOS Adjusted R-squared is: ', NN_adjusted_r_squared)

# Plotting a graph of the Training and Testing loss for each Epoch
plt.figure(figsize=(12, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Testing Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Testing Loss')
plt.legend()
plt.show()

# Plot the true values and predicted values
plt.figure(figsize=(12, 6))
plt.plot(y_test, label='Implied Volatility')
plt.plot(NN_predictions, label='NN Forecast')
plt.xlabel('Date Index')
plt.ylabel('Implied Volatility')
plt.title('Actual vs Predicted values')
plt.legend()
plt.show()

import tensorflow as tf
from sklearn.model_selection import TimeSeriesSplit
from sklearn.model_selection import GridSearchCV

# Creating a function that creates our NN model with different hyperparameters
def create_model(lstm_1,lstm_2,dense):
    model = Sequential()
    model.add(LSTM(lstm_1, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequence
    model.add(LSTM(lstm_2))
    model.add(Dense(dense, activation='linear'))
    model.add(Dense(1, activation='linear'))
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model
```

```
# Defining the grid search of parameters
param_grid = param_grid = {
    'lstm_1': [32, 64, 128],
    'lstm_2': [32, 64, 128],
    'dense': [16, 32, 64]
}


# Creating a TimeSeriesSplit for crossvalidation when optimising hyperparameters
tscv = TimeSeriesSplit(n_splits=5)

from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
model = KerasRegressor(build_fn=create_model, epochs=10, batch_size=32, verbose=0)

# Perform grid search with cross-validation
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=tscv)
grid_result = grid.fit(X_train, y_train)

# Print the best parameters and corresponding accuracy
print("The Best parameters: ", grid_result.best_params_)

best_model = grid_result.best_estimator_.model

# Calculating OOS RMSE
NN_predictions = best_model.predict(X_test)

NN_mse = np.mean((y_test - NN_predictions.flatten()) ** 2)
NN_rmse = np.sqrt(NN_mse)
print("The Root Mean Squared Error for the Neural Network model is: ", NN_rmse)

NN_r_squared = r2_score(y_test, NN_predictions)

n = len(y_test)
p = 8
NN_adjusted_r_squared = 1 - (1 - NN_r_squared) * (n - 1) / (n - p - 1)

print('The OOS R-squared is: ', NN_r_squared)
print('The OOS Adjusted R-squared is: ', NN_adjusted_r_squared)

# Plotting actual IV against NN forecasted IV for test set
plt.figure(figsize=(12, 6))
plt.plot(y_test, label='Implied Volatility')
plt.plot(NN_predictions, label='NN Forecast')
plt.xlabel('Date Index')
plt.ylabel('Implied Volatility')
plt.title('Actual vs Predicted values')
plt.legend()
plt.show()

# Importing L2 regularization function
from tensorflow.keras.regularizers import L2

# Extracting the optimal hyperparameters
LSTM1 = grid_result.best_params_['lstm_1']
LSTM2 = grid_result.best_params_['lstm_2']
DENSE = grid_result.best_params_['dense']

# Defining NN model with the addition of L2 Regularization
model_REG = Sequential([
    LSTM(LSTM1, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True, ke
    LSTM(LSTM2),
    Dense(DENSE, activation = 'linear'),
    Dense(K, activation = 'linear')
])

# Compiling the model
model_REG.compile(optimizer='adam', loss='mean_squared_error')

# Training the model on the Training dataset
history_REG = model_REG.fit(X_train, y_train, epochs=N_epochs, batch_size=N_batch_size, vali

# Predicting the OOS Implied Volatility using NN model
NN_REG_predictions = model_REG.predict(X_test)

# Calculating OOS RMSE
NN_REG_mse = np.mean((y_test - NN_REG_predictions.flatten()) ** 2)
NN_REG_rmse = np.sqrt(NN_REG_mse)
print("The Root Mean Squared Error for the Neural Network model is: ", NN_REG_rmse)

NN_r_squared = r2_score(y_test, NN_REG_predictions)

n = len(y_test)
p = 8
NN_adjusted_r_squared = 1 - (1 - NN_r_squared) * (n - 1) / (n - p - 1)

print('The OOS R-squared is: ', NN_r_squared)
print('The OOS Adjusted R-squared is: ', NN_adjusted_r_squared)

# Plot the true IV against the NN forecasted IV for test data
plt.figure(figsize=(12, 6))
plt.plot(y_test, label='Implied Volatility')
plt.plot(NN_REG_predictions, label='NN Forecast')
plt.xlabel('Date Index')
plt.ylabel('Implied Volatility')
plt.title('Actual vs Predicted values')
plt.legend()
plt.show()
```