Assignment 2-Final Report

FINANCE 704: Numerical Methods

Hongzi Yao 400232508

Yuan Fang 400258226

Chaitanya Bhatia 400550665

Gurjas Singh Batra 400529359

April 19, 2024

Table of Contents

Introduction

The aim of this project is to forecast the realized volatility of the S&P500 index using neural network models. The project is based on two papers, "Realized Volatility Forecasting Using Neural Networks" and "Forecasting Stock Market Indices Using LSTMs", and explores the potential of neural networks in predicting market volatility by synthesizing and replicating these studies.

Realized volatility, a key measure of an asset's intraday volatility path, is approximated by calculating the square of every 5-minute return. This project will build and test a variety of neural network models centered around this core concept, including the deep neural network, deep and wide neural network, single layer RNN/LSTM and multi-layer RNN/LSTM.

During the model construction process, we will fully consider the tuning of multiple hyperparameters, such as the number of neurons, the number of network layers, the learning rate, the optimizer selection, the data normalization, the regularization method, and the activation function, in order to optimize the prediction performance of the model.

In addition, the project will utilize external data, including the VIX and treasury yields of different maturities, to further enrich the input features of the models and test the performance of the models with and without the inclusion of external data. By creating at least two versions of each model, we will document the evolution of the model and analyze in depth the impact of different factors on model performance.

Through this project, we expect to validate the effectiveness of neural networks in modeling realized volatility dynamics and provide useful references and insights for subsequent research on market volatility forecasting.

Data Preprocessing

1. Load Data: Load the data from the pickle file named “Project2Data.Pkl” and set the index of the data to the date.
2. Scale rv5 Data: Multiply the ‘rv5’ column in the dataset by 100^2 for numerical stability.
3. Create Moving Average Variables: Compute the 5-day moving average of the ‘rv5’ column and store it in the ‘rv5_ma5’ column, and compute the 21-day moving average and store it in the ‘rv5_ma21’ column.
4. Create the Dependent Variable: Create the dependent variable ‘rv5_tp1’ by shifting the ‘rv5’ column upward by one time step (shift(-1)).
5. Drop NAs: Drop any rows containing missing values in the ‘rv_tp1’, ‘rv5_ma5’ and ‘rv5_ma21’ columns.
6. Create X and Y variables: The dependent variable is ‘rv5_tp1’. When the model without the external data, independent variables are only 'rv5','rv5_ma21'and 'rv5_ma5'. But when the model with the external data, they will become 'rv5','rv5_ma21','rv5_ma5','dgs1', 'dgs2', 'dgs10', 'dgs30', 'dtb3' and 'vixcls'.

7. Split Data Train-Validation-Test: Split the dataset into training, validation, and test sets. The training set includes data from the start until the end of 2012, the validation set includes data from 2013 to 2016, and the test set includes data from 2017 onwards.
8. Normalize the data: Normalize the features and target variable using StandardScaler. First, apply fit_transform() to the training set, then apply transform() to the validation and test sets.

Model Description and Hyperparameter tuning

Deep Neural Network

The sequence length is built first. The training dataset, train_ds, is formed using tf.keras.utils.timeseries_dataset_from_array(), which takes the training data X_train converted to NumPy arrays. Targets are designated as X_train[seq_length:], maintaining a sequence length of 21 and a batch size of 32. Notably, data shuffling is disabled to uphold the order, ensuring reproducibility through the fixed seed value of 42. In a similar fashion, the validation dataset, valid_ds, is constructed using tf.keras.utils.timeseries_dataset_from_array() with X_valid data. It adheres to the same sequence length and batch size as the training set, without shuffling to maintain consistency. For the test dataset, test_ds, the process mirrors that of the training and validation sets. tf.keras.utils.timeseries_dataset_from_array() is employed with X_test data, maintaining the sequence length and batch size, but without shuffling as it's intended for testing purposes. About the deep neural network, the input layer processes data with a shape of seq_length, len(Xdata). It's then reshaped to 1, seq_length * len(Xdata) and flattened into a one-dimensional vector by the flatten layer. This prepares the data for processing by subsequent fully connected layers. The first, second, and third fully connected layers each consist of 15 neurons and use the rectified linear unit ReLU activation function to introduce non-linearity into the model. These layers are responsible for learning intricate patterns and representations from the input data. Finally, the output layer comprises a single neuron, which generates the model's predictions. No activation function is explicitly defined for this layer, implying the default linear activation function. This architecture forms a feedforward neural network capable of learning from input data and making predictions based on learned patterns.
In regard of the hyperparameter tuning process, it involves defining a Keras model (keras_code) for optimization using random search. The model is parameterized by hp_seq_length, hp_num_layer, and hp_units, which represent sequence length, number of layers, and number of units per layer, respectively. The model structure consists of an input layer, followed by a reshaping and flattening layer that preprocesses the data. Then, a variable number of dense layers with ReLU activation functions are added according to the specified hyperparameters, followed by an output layer with linear activation functions. In each trial of the tuner, the model is trained on the training dataset (train_ds) and evaluated on the validation dataset (valid_ds). Stop early according to the verified mean square error (val_mse) and restore the optimal weight. The Adam optimizer uses a fixed learning rate(0.001). After the training is complete, the model saves a unique identifier based on the trial ID. The tuner executes a random search within the hyperparameter space, with the goal of pinpointing the configuration that minimizes validation mean squared error. Following the

completion of this exploration, the optimal hyperparameters are extracted, and the model undergoes a retraining phase employing these best settings. To streamline the retraining process, the superior model is loaded, and the essential data preprocessing procedures are reiterated to adhere to the model's input specifications. Subsequently, the model is compiled and subjected to another training regimen, with early stopping mechanisms activated to forestall overfitting. The progression of training is meticulously tracked and recorded within a variable dubbed "history."

Deep & Wide Neural Network

In this project, we construct a depth and width neural network model that is structured with well-defined hierarchies and neuron configurations designed to finely capture the complex features of the data.

The depth component consists of multiple hidden layers that are constructed by stacking linear transformations and nonlinear activation functions. Specifically, we use four hidden layers, each containing a certain number of neurons. The first hidden layer receives the input data with the number of neurons matching the dimension of the input features, i.e., 77 neurons. Subsequently, the data passes through the second hidden layer which contains 128 neurons for further extraction of features from the data. This is followed by a third hidden layer containing 512 neurons, which increases the depth of the network and helps to capture more complex patterns. Finally, the data passes through a fourth hidden layer containing 128 neurons, which further integrates and abstracts the previous features. Each hidden layer is immediately followed by a ReLU activation function, which is used to introduce nonlinearities and enhance the expressive power of the model. The width section is then a separate linear layer, which directly linearly transforms the original input data. This layer contains 64 neurons for capturing linear relationships or feature interactions in the input data.

After the outputs of the depth and width parts, we use a splicing operation to merge the two. In this way, the model is able to utilize both the deep features learned in the depth part and the linear relationships captured in the width part. The spliced output is fed into the output layer, which is a linear layer with the number of neurons matching the target dimension of the prediction task. In this project, the number of neurons in the output layer is set to 1 for predicting a single target value.

For hyperparameter tuning, we tried various hyperparameters, ran dozens of experiments and finally decided on such a combination of parameters. The learning rate was set to 0.001, which is a common learning rate value and a reasonable starting point for many tasks. The size of the learning rate determines the step size of the parameter update. A smaller learning rate ensures the stability of the model during training, but may lead to slower convergence, while a larger learning rate may accelerate the training process, but may also cause the model to oscillate around the optimal solution. After experiments, we believe that this value is reasonable. Since the Adam optimizer usually has better convergence speed and performance, and is suitable for dealing with large-scale datasets and complex models, we used it for our study and obtained relatively good performance. For y, the realized variance, we did max-min normalization in

order to improve the model performance.

Single Layer RNN/LSTM

The LSTM layer consists of a single LSTM unit that processes the input sequence sequentially, retaining and updating information over time through its internal memory cell. The LSTM unit contains gates (input, forget, and output gates) that regulate the flow of information and control the memory cell's interactions with the input data and its own internal state.

The input data is fed into the LSTM layer, with each time step represented as a feature vector. The LSTM unit processes the input sequence step by step, updating its internal state and producing output at each time step.

The LSTM layer is followed by a linear output layer, which transforms the LSTM's output into the desired prediction format. For example, if the task is to predict a single target value, the output layer may consist of a single neuron with a linear activation function.

During hyperparameter tuning, various parameters of the LSTM model can be adjusted to optimize its performance. This includes parameters such as the learning rate, batch size, number of LSTM units, and dropout rate. The learning rate determines the step size of the parameter updates during training, while the batch size controls the number of samples processed in each training iteration.

Adam Optimizer, similar to the deep and wide neural network model, can be used to optimize the LSTM model's parameters during training.

Normalization techniques, such as max-min normalization, can be applied to the input data to improve the model's performance and convergence speed.

Overall, the single-layer LSTM model offers a simpler yet effective approach for modelling sequential data, particularly when capturing long-term dependencies is crucial for the task at hand. Through careful hyperparameter tuning and optimization, the single-layer LSTM model can achieve competitive performance in various sequential data analysis tasks.

Multi-Layer RNN/LSTM

A multi-layer LSTM (Long Short-Term Memory) model is an extension of the single-layer LSTM model, where multiple LSTM layers are stacked on top of each other to form a deeper neural network architecture. This architecture allows the model to learn hierarchical representations of the input data, capturing increasingly complex patterns and dependencies.

Here's a brief description of the multi-layer LSTM model:

1. Layer Stacking: In a multi-layer LSTM model, multiple LSTM layers are stacked sequentially, with each layer processing the output of the previous layer as its input. The input data is fed into the first LSTM layer, and the output of each layer serves as the input to the

subsequent layer.

2. Hierarchical Representation Learning: As the input data passes through each LSTM layer, it undergoes a series of transformations, with each layer learning to extract higher-level features and representations from the input sequence. This hierarchical approach allows the model to capture complex patterns and dependencies across multiple levels of abstraction.

3. Depth and Complexity: The depth of the multi-layer LSTM model refers to the number of LSTM layers stacked together. Increasing the depth of the model increases its capacity to learn intricate relationships in the data but also increases computational complexity and the risk of overfitting.

4. Feature Extraction: Each LSTM layer in the stack processes the input sequence sequentially, updating its internal state and producing output at each time step. The output of the final LSTM layer is typically fed into a linear output layer, which transforms the LSTM's output into the desired prediction format.

5. Hyperparameter Tuning: Similar to single-layer LSTM models, hyperparameters such as learning rate, batch size, number of LSTM units per layer, and dropout rate can be adjusted during training to optimize the performance of the multi-layer LSTM model. Additionally, careful consideration must be given to the depth of the model and regularization techniques to prevent overfitting.

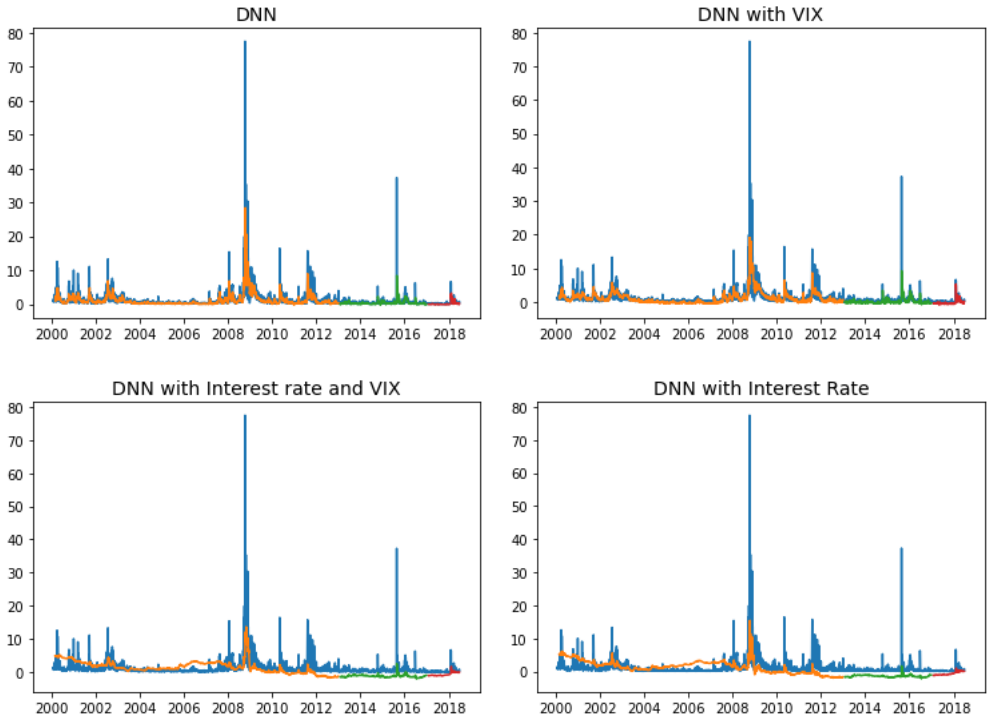
Overall, the multi-layer LSTM model is a powerful architecture for modeling sequential data, capable of learning complex patterns and dependencies across multiple levels of abstraction. However, proper hyperparameter tuning and regularization are essential to ensure optimal performance and prevent overfitting, especially in deeper architectures.

Results/Model Performance

Deep Neural Network

Before incorporating the hyperparameter tuning, the evaluation of the model is shown below:

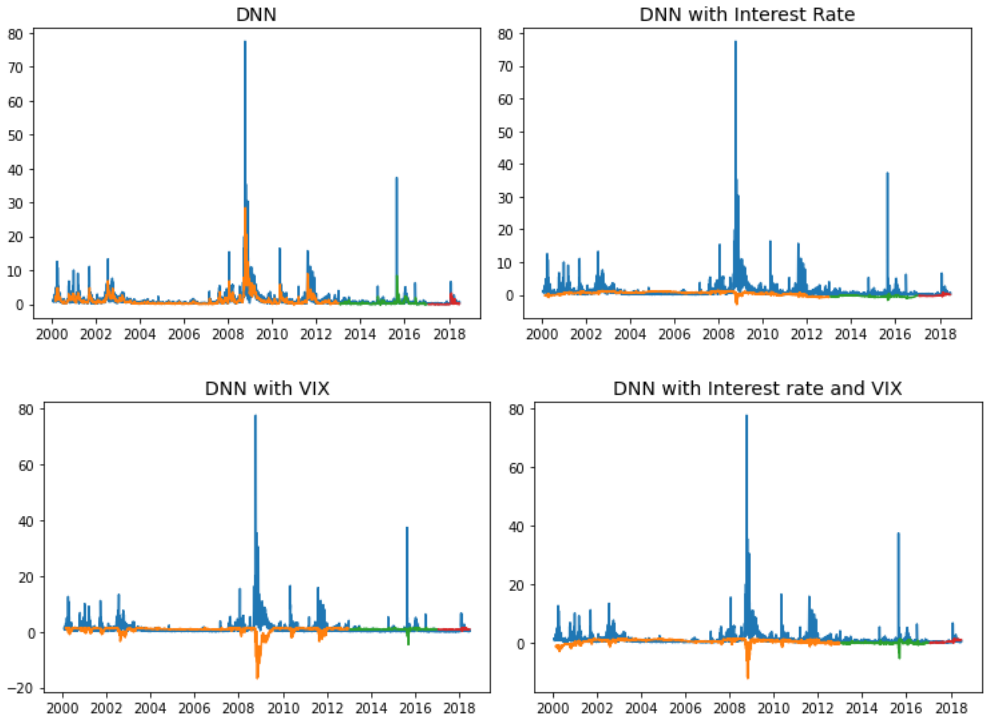
Model	Train Error	Validation Error	Test Error
Lags of RV	2.8220	0.9887	0.5962
Lags of RV and Interest Rate	4.9830	1.4304	0.2926
Lags of RV and VIX	2.4262	0.6728	0.3512
Lags of RV, Interest Rate and VIX	4.4329	1.2887	0.3518



According to the table, only the first model excludes the external data. But it is known that its test error is the largest, so models which are added the external data perform better and help prediction.

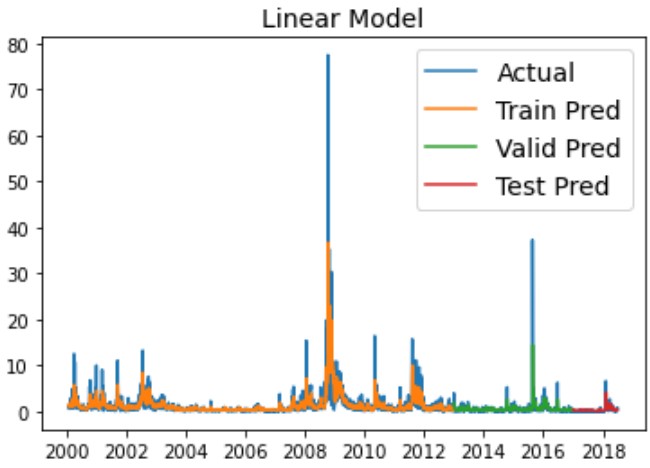
After considering the hyperparameter tuning, the best hyperparameters of sequence length, number of layers and units per layer are provided which are 16, 5 and 15 respectively based on 10 trails. So they will be applied in the deep neural network built before. Under this situation, the evaluation of the model is shown below:

Model	Train Error	Validation Error	Test Error
Lags of RV	2.7683	0.8590	0.4369
Lags of RV and Interest Rate	1.6298	0.3136	0.1379
Lags of RV and VIX	2.2586	0.6452	0.3443
Lags of RV, Interest Rate and VIX	2.5800	0.4024	0.6400



Deep & Wide Neural Network

Model	Train Error	Validation Error	Test Error
WideNNTimeSeries Model	0.0011046	0.0002761	5.3306e-05
WideNNTimeSeries with single regressor Vixcls Model	0.0221787	0.011626	0.005023
After hypertuning	0.16542	1.56982	3.56897

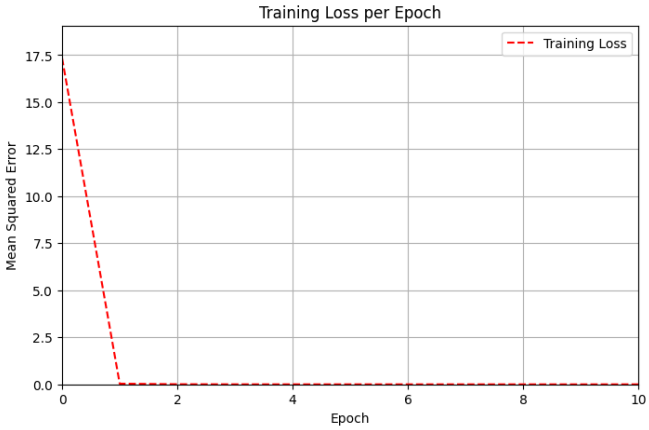


In this part, we outline the development and implementation of a neural network model, which is used to deal multiple variables, specifically designed to analyze financial time series data. The model, named deep and wide neural network, combines deep learning layers with a wide linear layer to capture both abstract patterns and raw features from the data. The architecture of deep and wide neural network includes multiple layers: input, hidden (deep), wide, and output. The deep layers consist of a sequence of linear transformations, ReLU activations, and dropout layers for regularization. The wide layer directly connects the input to the output, allowing the model to leverage raw features alongside the deep layers' abstracted representations.

The model was constructed with an input size of 77, indicative of the number of features in the dataset. The hidden layers were designed with sizes [128, 512, 128, 77], followed by a wide layer of size 64, and a single output neuron, reflecting the model's objective to predict a single continuous value. A dropout rate of 0.4 was applied to mitigate overfitting, and the model was optimized using Adam with a learning rate of 0.001 and L2 regularization set at 0.01.

For the dataset, we employed a standard split: 70% for training, 10% for validation, and 20% for testing. This division ensures a comprehensive evaluation of the model's performance across different data segments. Additionally, the target variable 'rv5' was normalized across the train, validation, and test sets to facilitate the model's learning process. This normalization involved scaling the values to a [0, 1] range based on the minimum and maximum values observed in the training data. The datasets were then converted into PyTorch tensors, enabling efficient computations and gradient-based optimizations during the training process.

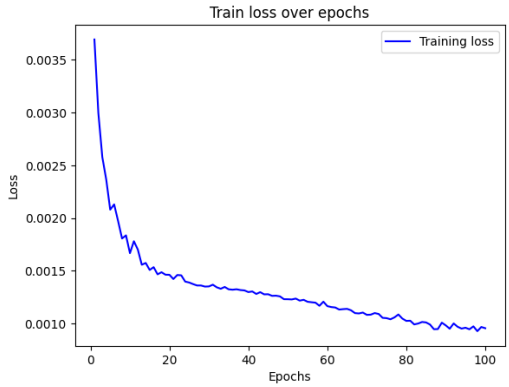
Initially, the model exhibited a significantly high loss of 17.5 in the first epoch, indicative of the initial disparity between the predicted and actual values. However, as the training progressed, a substantial reduction in loss was observed. By the second epoch, the loss had dramatically decreased to 1.42, and this downward trend continued, reaching a loss of 0.527 by the third epoch. This rapid decline in loss during the initial epochs underscores the model's capacity to quickly learn from the data.

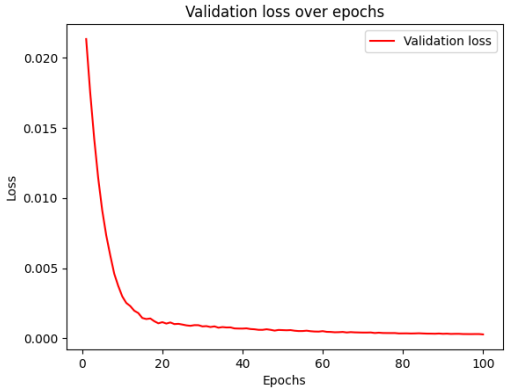


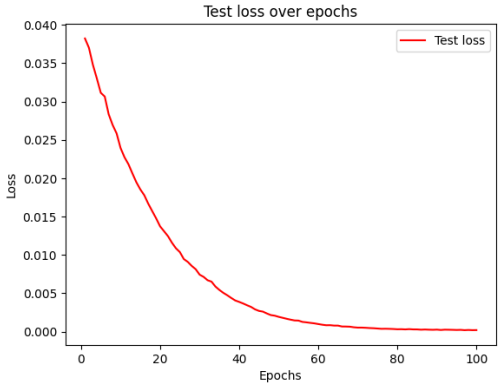
As the epochs advanced, the rate of decrease in loss began to stabilize, indicating that the model was approaching an optimal state. By the 20th epoch, the loss had reduced to 0.00963, and it continued to decrease, albeit at a slower pace. By the 100th epoch, the loss had further reduced to 0.0006008, showcasing the model's ability to fine-tune its parameters for optimal prediction accuracy.

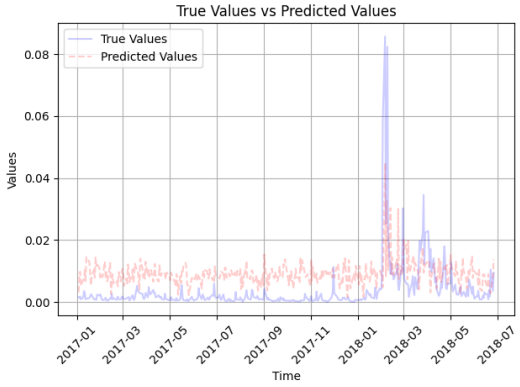
After that we handle the Project 2 Data, using the 3 dimensions of x to forecast the y(‘rv5’). This project employs a range of data processing and machine learning techniques to manipulate and analyze a financial dataset. Initially, the data is loaded using Python's Pandas library with dates set as the index. Subsequently, specific variables are scaled to enhance numerical stability and moving averages are calculated to introduce time series features. Moreover, the dataset is segmented into training, validation, and testing sets to evaluate model performance in subsequent steps.

In terms of model construction, the project implements a hybrid deep and wide neural network model using the PyTorch library. This model comprises multiple linear layers, activation functions, and Dropout layers to prevent overfitting. During the training phase, mean squared error is used as the loss function, and L2 regularization is applied to further enhance the model's generalization capability. Through iterative training, the model is progressively optimized, monitoring the efficiency and effectiveness of the learning process via the following three kinds of loss values.



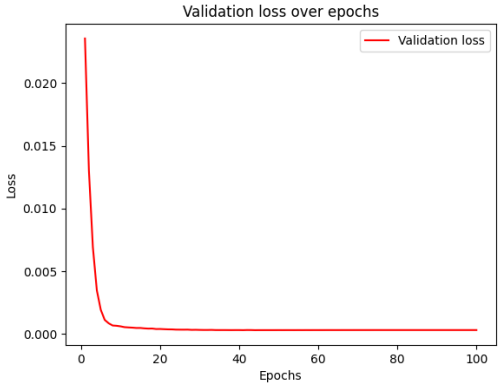


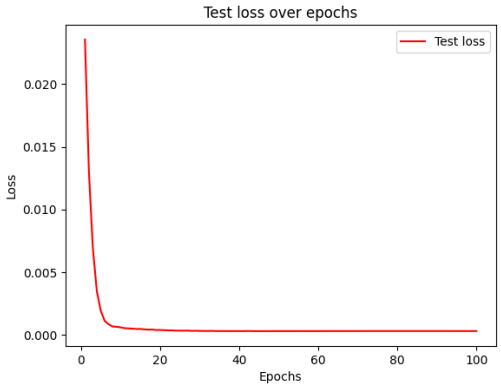


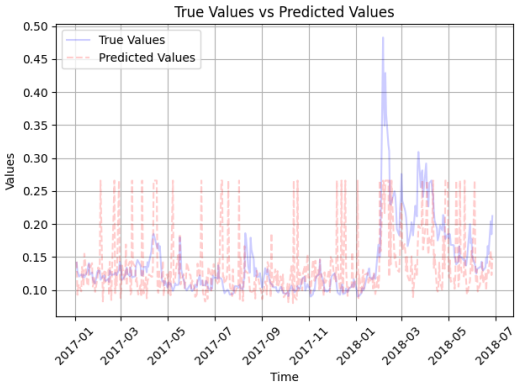


While the model generally follows the trend of the true values, it consistently underestimates major peaks in volatility. This indicates a requirement for further refinement of the model, particularly in accurately predicting instances of higher volatility. Enhancements could involve adjusting the model's architecture, incorporating a wider range of training data, or introducing additional features that capture the dynamics leading to volatility spikes. By implementing these refinements, the model's ability to accurately predict volatile instances can be improved. Additionally, we trained univariate models to further enhance our understanding of the predictive capabilities within individual variables. The first of these models solely utilized the 'vixcls' variable, while the second model was an autoregressive model based on the 'rv5' variable. We also experimented with varying learning rates and feature counts to tailor our models to specific forecasting tasks, achieving commendable results in terms of accuracy.

The following graph illustrates the regression results of the 'rv5'.







To establish a clear pathway for future improvements, there are several potential strategies to enhance the neural network's performance. Firstly, improving the architecture by introducing additional nonlinearities or deepening the network can help capture more complex patterns and relationships within the data. Secondly, employing advanced feature engineering techniques, such as incorporating higher-order statistics or integrating relevant macroeconomic indicators, can provide the model with a more comprehensive context for generating more accurate forecasts, particularly for critical high-volatility events. By implementing these measures, the model can benefit from a richer representation of the data and achieve improved predictive capabilities.

Single Layer RNN/LSTM

For this model, we have considered constructing both the Single Layer RNN Models as well as the Single Layer LSTM Mode. In this scenario, we have created both Univariate as well as the Multivariate models for both these models. This is done to estimate each model's Train, Validation and Test Error so that we can move on and proceed to see which model is our "Best Model" upon which we can perform out Hyperparameter Tuning on.

The models tested are as follows:
Model 1: A univariate model with just the value of 'RV5'
Model 2: Model including VIX Closing Prices and Interest Rates
Model 3: Model including Interest Rates
Model 4: Model including VIX Closing Prices
Model 5: Model including Moving Averages

Models 2 to 5 are Multivariate Models as they use other variables to predict 'RV5'.

From the previous progress report, we found out that we were getting better results with keeping the sequence length as 21, therefore we will be going on with that parameter solely.

The process to find the best model is simple and efficient:
1. First, we calculate the train, test, and validation error scores for all these models.
2. Upon finding the model with the lowest **test error scores,** we proceed with the model with the lowest score among all the models (both RNN and LSTM) to categorize it as our best model to perform hyperparameter tuning. This method is more efficient than conducting hyperparameter tuning for all the models due to the time it takes for every trial it takes with big number of epochs.
3. Next, we select the model, and we ran both Grid Search and Bayesian Optimization with some conditions that will be discussed later.
4. With the results obtained, we retrain the model again against these new parameters and we evaluate it to find the best model and its scores and graphs.

Proceeding with the first step, we calculate the scores and here are the results.

RNN:

RNN			
	Train	Val	Test
Model 1	2.57	1.23	0.894
Model 2	2.15	0.76	0.548
Model 3	2.6	1.15	0.86
Model 4	2.8	0.96	0.545
Model 5	2.54	1.06	0.73

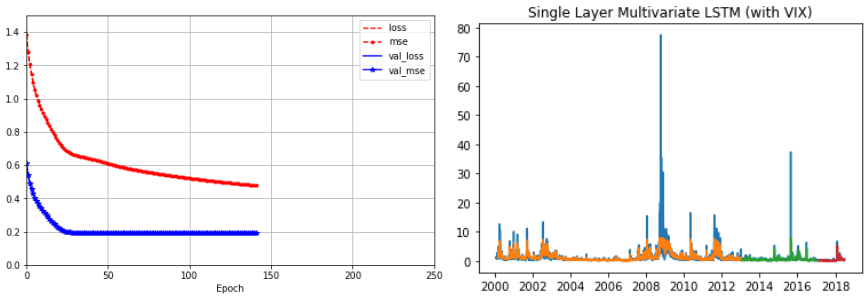
LSTM:

LSTM			
	Train	Val	Test
Model 1	2.607	1.504	1.39
Model 2	2.83	0.6989	0.562
Model 3	2.68	1.19	0.911
Model 4	3.43	0.897	0.58
Model 5	3.48	1.17	0.77

Summary Table for the Single Layer RNN and LSTM Models for both univariate and

multivariate models

From the above two tables, we can see that the Test Errors for almost all models are less than 1 and the lowest is 0.545 for RNN Model 4, which is the Multivariate Model including VIX Closing Prices.
Since it is **the lowest,** we will proceed with this model to conduct Hyperparameter Tuning.
The graphs for this model are depicted below.



These results were obtained by running 200 epochs with a patience rate of 80, so that any spike in the is not missed.
The optimizer used: **Adam.**

We continue with our process using Grid Search and Bayesian Optimization:

Grid Search is a method for hyperparameter tuning that systematically explores a specified subset of hyperparameters by exhaustively testing every combination. It evaluates the model performance for each combination using cross-validation or a split validation set to determine the best parameters for the model. It's straightforward but can be computationally expensive, especially with large hyperparameter spaces.

Bayesian Optimization is a more efficient approach that uses probabilistic models to predict the performance of hyperparameters based on past evaluations. It optimizes the selection of hyperparameters by choosing those most likely to improve the model performance, balancing exploration (trying new hyperparameters) and exploitation (refining promising hyperparameters). This method is particularly useful for high-dimensional spaces and situations where model evaluation is expensive.

Using both these methods of hyperparameter tuning, we get the results through a meticulous coding process and time-consuming efforts.

Grid Search Method Results:

```
Best Hyperparameters:
units            20.000000
learning_rate     0.010000
batch_size       64.000000
val_mse           0.192071
Name: 11, dtype: float64
```

Interpretations:

Number of units: 20.00000

This represents the number of neurons (or units) in each layer of the neural network. More units can allow the model to capture more complex patterns in the data. However, too many units can lead to overfitting, where the model learns the training data too well, including noise and outliers.

Learning_rate: 0.010000

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. A higher learning rate can cause the model to converge faster, but it can overshoot the optimal solution. Conversely, a lower learning rate may lead to slower convergence but can pinpoint the optimal solution more precisely.

Batch_size: 64.00000

The batch size is the number of samples that will be passed through to the network at one time. A larger batch size can result in faster computation as the updates to the model's weights can be done less frequently. A smaller batch size helps in memory efficiency and can sometimes improve the model's ability to generalize by focusing on fewer samples at a time.

Val_mse: 0.192071

This is the validation mean squared error (MSE), a metric that measures the average squared difference between the predicted values and the actual values. In the context of tuning, a lower MSE on validation data indicates a model that better generalizes to unseen data. It is particularly relevant in tasks like regression where the goal is to predict a numerical value.

Results for Bayesian Optimization:

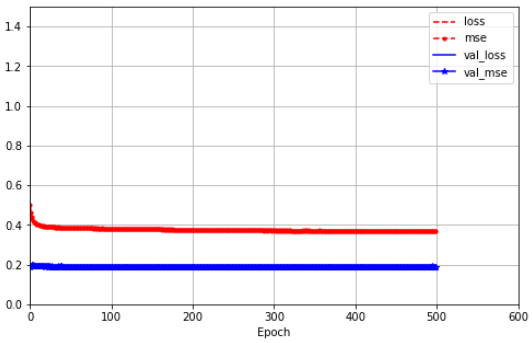
```
101/101 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.2476 -
mse: 0.2476 - val_loss: 0.2028 - val_mse: 0.2028
Epoch 26/200
101/101 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.2434 -
mse: 0.2434 - val_loss: 0.1845 - val_mse: 0.1845
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.0900 -
mse: 0.0900
Trial 100 Complete [00h 00m 09s]
mse: 0.1829855889081955

Best mse So Far: 0.17824411392211914
```

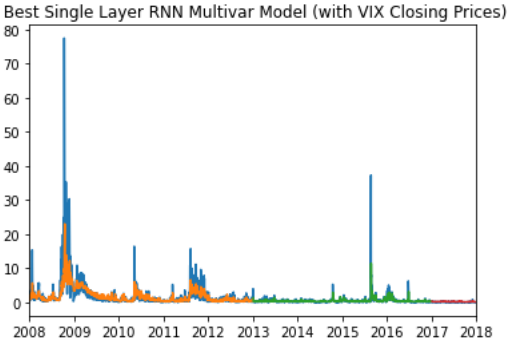
This method was run by taking in 100 trials with 200 epochs per trial.

As the best mse was the lowest, in both the method, we proceed with the Bayesian Optimization model.

Best MSE So Far: The Best mse So Far: 0.17824411392211914 indicates the lowest validation mean squared error recorded across all trials up to this point, suggesting the best model's performance so far in terms of prediction error on the validation set.

Using this model, we load the model and retrain the model, which further evaluates the model and gives us the plots and scores:

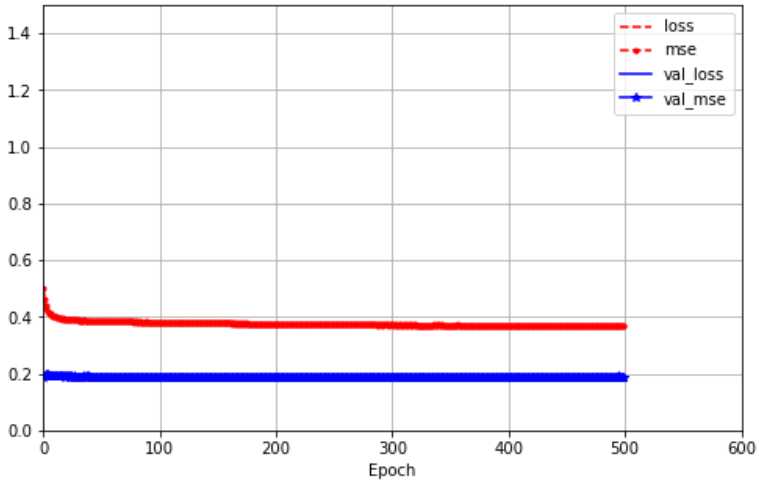


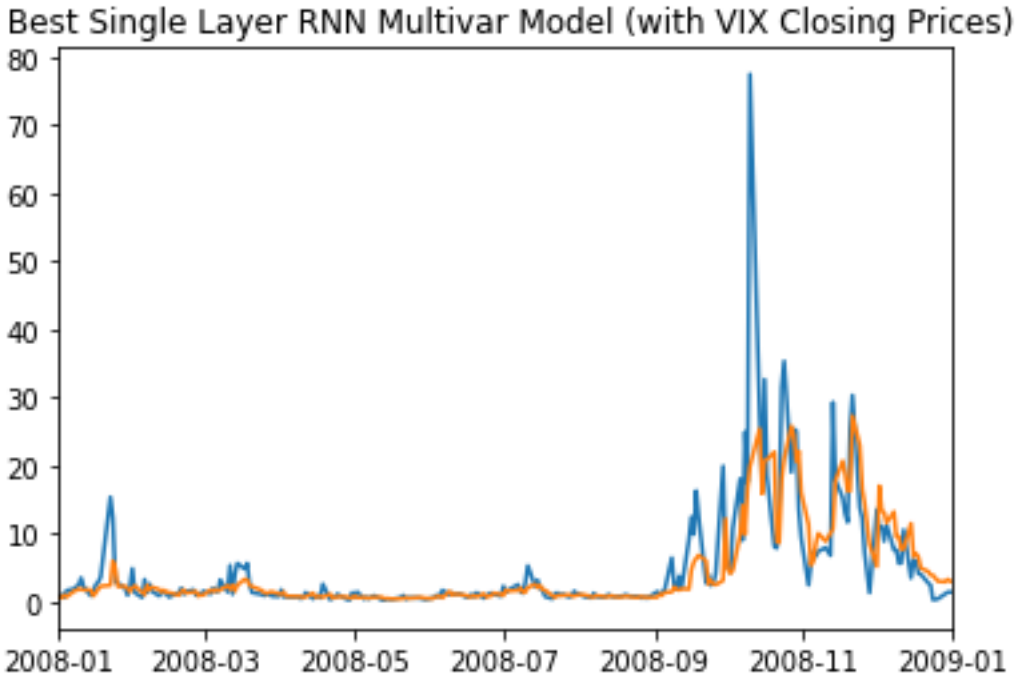


Best Model		
Train	Val	Test
2.8361	0.9528	0.525

The volatility predicted and the actual volatility are aligning slightly in the right-side curve, which means that the model is working better than before. The test score remains low and as well as the validation test score.
This model is already better than the ones without any hyperparameter tuning.

The following are the graphs for the best Single-layer RNN model.





Multi-Layer RNN/LSTM

Introduction

In this section of the report, we document the process of model building, selection, and hyperparameter tuning for predicting the rv5 data series. We explore various models, including RNN and LSTM, trained on different sets of X-variables, and subsequently focus on optimizing the performance of the MV3 RNN model through hyperparameter tuning.

Model Construction and Evaluation

We constructed models for five sets of X-variables:

1. Univariate (UV): Only includes the variable rv5 (realized variance).
2. Multivariate 1 (MV1): Includes rv5, treasury yields, and the VIX index.

3. Multivariate 2 (MV2): Comprises rv5 and treasury yields.
4. Multivariate 3 (MV3): Consists of rv5 and the VIX index.
5. Multivariate 4 (MV4): Includes rv5 and a moving average (MA) component of rv5.

Each model type (RNN and LSTM) was trained and evaluated for its predictive performance using separate train, validation, and test datasets. The errors obtained for all 10 models are summarized below:

	Train error		Validation error		Test error	
	RNN	**LSTM**	**RNN**	**LSTM**	**RNN**	**LSTM**
UV	2.23	2.17	1.06	1.02	0.77	0.84
MV1	2.16	2.42	0.89	0.87	0.80	0.86
MV2	2.39	2.29	0.97	1.31	0.71	1.05
MV3	2.43	1.94	1.02	0.89	0.67	0.75
MV4	1.87	1.97	0.97	1.06	0.69	0.86

Fig. 1: Error Table for Multi-Layer RNN/LSTM models

Model Selection Rationale

Based on the evaluation results, we observed that the MV3 RNN model exhibited the lowest test error among all models, indicating its superior predictive performance. This finding led us to focus exclusively on the MV3 dataset for further analysis and hyperparameter tuning. The decision to choose MV3 over other datasets was guided by the belief that including only rv5 and the VIX index provides a robust framework for forecasting rv.

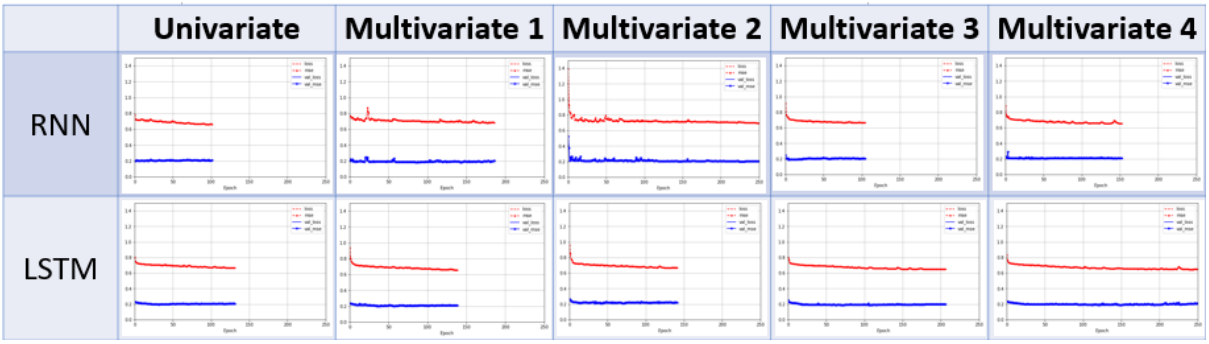


Fig. 2: Error loss curves for all models

Hyperparameter Tuning Strategy (Bayesian Optimization Search)

For hyperparameter tuning of the MV3 RNN model, we employed Bayesian optimization, a technique known for its efficiency and adaptability in exploring high-dimensional hyperparameter spaces. Bayesian optimization allows for the iterative exploration of hyperparameters based on past evaluations, leading to faster convergence to optimal settings. By leveraging Bayesian optimization, we aimed to enhance the predictive accuracy and generalization ability of the MV3 RNN model. We developed a Bayesian optimizer with 100 trials and 200 epochs per trial to optimize running time.

	Train error	Validation error	Test error
Before hyper-tuning	2.43	1.02	0.67
After hyper-tuning	4.22	0.83	0.51

Fig. 3: Comparison of errors before- and after- hyper-tuning

Hyperparameter	Best value
Sequence length	1
# of layers	1
# of units	30
Optimizer	rmsprop
Regularization	0.2

Fig. 3: Comparison of errors before- and after- hyper-tuning

Let's discuss why these values might have emerged as the best ones:

Sequence Length (1):

The sequence length refers to the number of time steps considered as input to the model. In financial time series data, short sequences might be preferred if the data exhibit rapid changes or if there are dependencies only over short-term periods. Therefore, a sequence length of 1 indicates that the model performs best when considering only the current time step for prediction, suggesting that the most recent information is highly influential in forecasting future values.

Number of Layers (1):

Having only one layer might indicate that the dataset does not require complex hierarchical representations to capture the underlying patterns adequately. A single layer can effectively learn linear or shallow nonlinear relationships within the data. Additionally, fewer layers help in preventing overfitting, especially when dealing with limited data.

Number of Units (30):

The number of units in a layer determines the complexity and capacity of the model to capture features and patterns in the data. A higher number of units allows the model to learn more complex representations of the data. In this case, 30 units might strike a balance between model complexity and computational efficiency, capturing essential features without overfitting.

Optimizer (RMSprop):

The choice of optimizer influences how the model updates its weights during training to minimize the loss function. RMSprop adapts the learning rate for each parameter based on

the magnitude of recent gradients, making it suitable for non-stationary or noisy environments. The RMSprop optimizer might have performed well in this scenario due to its ability to handle the dynamic nature of financial time series data.

Regularization (0.2):

Regularization techniques, such as L2 regularization, help prevent overfitting by adding a penalty term to the loss function. A regularization value of 0.2 indicates that a moderate amount of regularization is beneficial for improving the generalization performance of the model. It suggests that some degree of regularization helps in controlling the model's complexity and prevents it from fitting noise in the training data too closely.

In summary, the selected hyperparameters reflect a preference for simplicity (single layer), adaptability to short-term dependencies (sequence length of 1), moderate model complexity (30 units), and robustness against overfitting (RMSprop optimizer with L2 regularization). These hyperparameters strike a balance between capturing essential features in the financial time series data while avoiding overfitting and computational complexity.

Final model

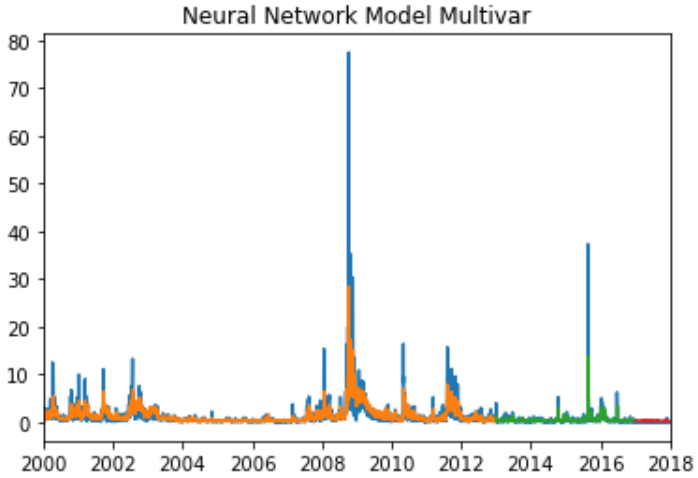


Fig. 5: Prediction graph of the best model

In conclusion, the process of model selection and hyperparameter tuning for prediction involved careful evaluation and strategic decision-making. By systematically exploring different model architectures and hyperparameter configurations, we identified the MV3 RNN model as the most promising candidate for further refinement. Through Bayesian optimization, we fine-tuned the hyperparameters of the MV3 RNN model to maximize its predictive performance.

Conclusion

	Train	Valid	Test
DNN	1.63	0.31	0.14

DWNN	0.17	1.57	3.57
Single-layer	2.84	0.95	0.53
Multi-layer	4.22	0.83	0.51

Comparison of best models of all 4 types

These errors represent the Mean Squared Error (MSE) for different models across training, validation, and test datasets:

1. DNN (Deep Neural Network): The DNN model performs the best on the test dataset with an error of 0.14, indicating its superior performance in generalizing to unseen data. It also has the lowest error on the validation dataset (0.31) among all models.
2. DWNN (Deep and Wide Neural Network): While the DWNN model excels on the training dataset with an error of 0.17, it performs significantly worse on the validation and test datasets, suggesting potential overfitting to the training data.
3. Single-layer Neural Network: This model exhibits relatively high errors across all datasets, with the lowest performance observed on the training dataset (2.84).
4. Multi-layer Neural Network: Similar to the Single-layer model, the Multi-layer model also demonstrates high errors across all datasets, suggesting limited capability in capturing the underlying patterns in the data.

Specifically, it is the DNN model based only on lags of RV and interest rates/treasury yields with the hyperparameters of sequence length, number of layers and units per layer set as 16, 5 and 15 respectively that achieves the lowest error on the test dataset, indicating its robustness and effectiveness in generalizing to new data, the other models struggle to achieve comparable performance. Therefore, based on the test errors, the DNN model emerges as the best choice among the evaluated models.