MSc in Mathematical Modeling

Deep Learning

Recurrent Neural Networks

Aimilios Potoupnis

Contents

Introduction to Recurrent Neural Networks

In this exploration, we delved into Recurrent Neural Networks (RNNs). But what exactly are RNNs?

Recurrent Neural Networks, or RNNs, represent a class of artificial neural networks designed to handle sequential data with temporal dependencies. Unlike traditional feedforward neural networks, RNNs possess loops within their architecture, allowing them to persist information over time. This inherent memory capability enables RNNs to process sequences of inputs, making them well-suited for tasks such as natural language processing, time series prediction, and speech recognition. Picture an RNN as a conveyor belt of information processing. Each input in a sequence is akin to an item moving along the belt, with the network's recurrent connections enabling it to retain and integrate past information as it progresses through the sequence.

When it comes to Recurrent Neural Networks (RNNs), there are several main architectures that are commonly used: Simple RNN, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU).

1. **Simple RNN (Recurrent Neural Network):** This is the fundamental architecture of RNNs. It processes sequential data by iterating through the elements while maintaining a state vector that serves as memory. However, simple RNNs suffer from the vanishing gradient problem, where gradients become too small to effectively update weights over long sequences, limiting their ability to capture long-term dependencies.
2. **Long Short-Term Memory (LSTM):** LSTM networks were introduced to address the limitations of simple RNNs, particularly in capturing long-term dependencies in sequential data. LSTMs incorporate memory cells and various gating mechanisms to control the flow of information, enabling them to retain information over long sequences and mitigate the vanishing gradient problem. They are well-suited for tasks requiring modeling of context over extended periods.
3. **Gated Recurrent Unit (GRU):** GRUs are a variation of LSTM networks that aim to simplify the architecture while maintaining similar performance. GRUs combine the forget and input gates of LSTMs into a single "update gate," reducing the number of parameters and computations compared to LSTM. Despite their simpler design, GRUs have shown effectiveness in various sequence modeling tasks and are preferred when computational resources are limited.
4. **Bidirectional LSTM (BiLSTM):**BiLSTM extends the capabilities of traditional LSTM networks by processing input sequences in both forward and backward directions. This allows the network to capture information from past and future contexts simultaneously, enhancing its ability to understand and model dependencies within the sequence. BiLSTMs are particularly useful in tasks where context from both

directions is important, such as natural language processing, speech recognition, and sentiment analysis.

5. **Echo State Networks (ESNs):** Echo State Networks are a type of recurrent neural network, although they have a slightly different architecture compared to traditional RNNs like LSTM and GRU. ESNs consist of a large reservoir of recurrently connected units with fixed random weights, which are then combined with a linear readout layer to produce the network's output. The key characteristic of ESNs is their ability to efficiently exploit the dynamics of the reservoir to perform various computational tasks, such as time series prediction, pattern recognition, and control tasks. While ESNs share some similarities with traditional RNNs, their unique architecture and training methodology distinguish them as a distinct class of recurrent neural networks.

This project

In this project I decided to work with time series prediction. In the first half of the project, I decided to work with synthetic data that have specific Seasonality and trend along with some noise, that are not random. On the second half I decided to work on real world problem, more specifically with the Bitcoin price over the years and see whether the models can accurately predict real timeseries.

In the initial phase of my project, I opted to engage with synthetic data characterized by distinctive seasonality, trend components, and structured noise, rather than random fluctuations. This deliberate choice allows for a controlled exploration of various time series forecasting techniques. By manipulating these synthetic datasets, I aim to gain insights into how different models perform under specific conditions, discerning their strengths and limitations in capturing patterns amidst structured variability.

Transitioning to the latter half of the project, I pivoted towards a real-world application by focusing on Bitcoin price data spanning multiple years. Here, the objective shifts from controlled experimentation to assessing the efficacy of time series prediction models in a dynamic and complex environment. By analyzing Bitcoin's historical price movements, I endeavor to ascertain the extent to which forecasting algorithms can accurately anticipate trends and fluctuations in a high-stakes financial market, offering valuable insights for investors and analysts alike.

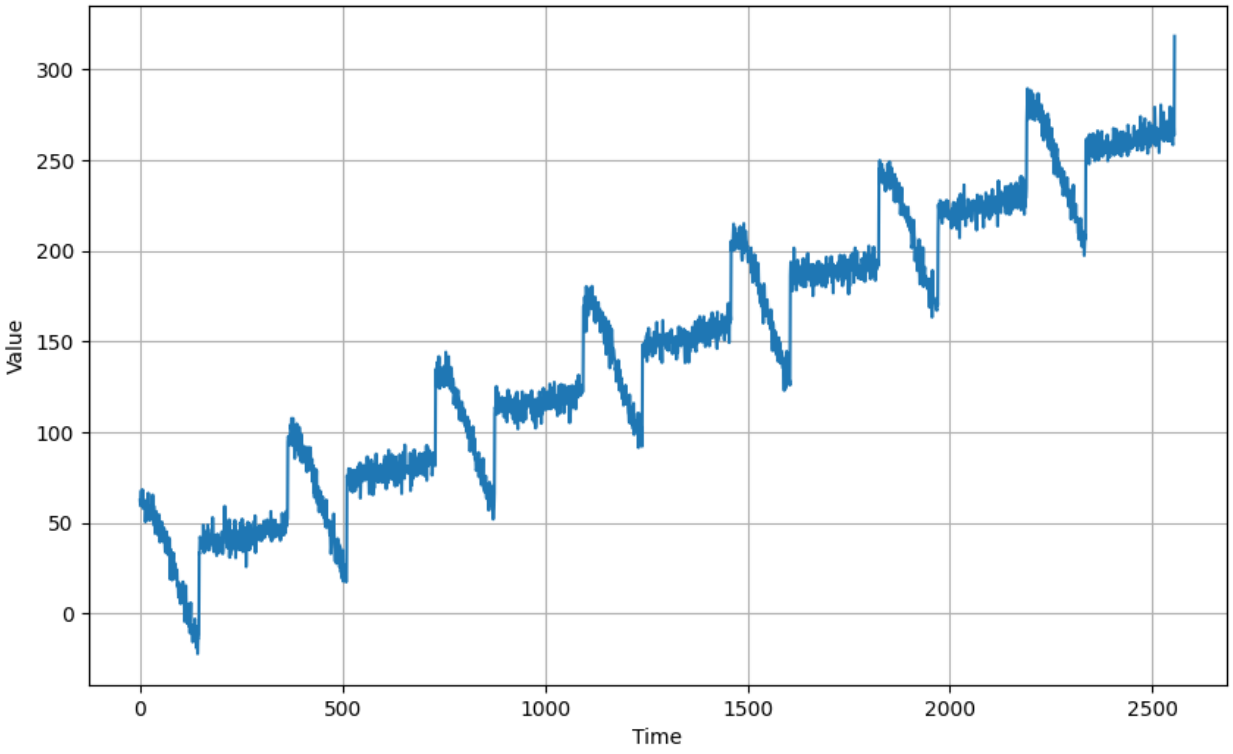
In both scenarios, I plan to explore a range of predictive models, encompassing both traditional statistical methods and cutting-edge neural network architectures. The roster includes simple Artificial Neural Networks (ANN), Recurrent Neural Networks (RNN), Long Short-Term Memory networks (LSTM), and Gated Recurrent Units (GRU). Each of these architectures brings its unique strengths to the table, allowing for a comprehensive evaluation of their performance across different types of time series data.

Synthetic Timeseries Data

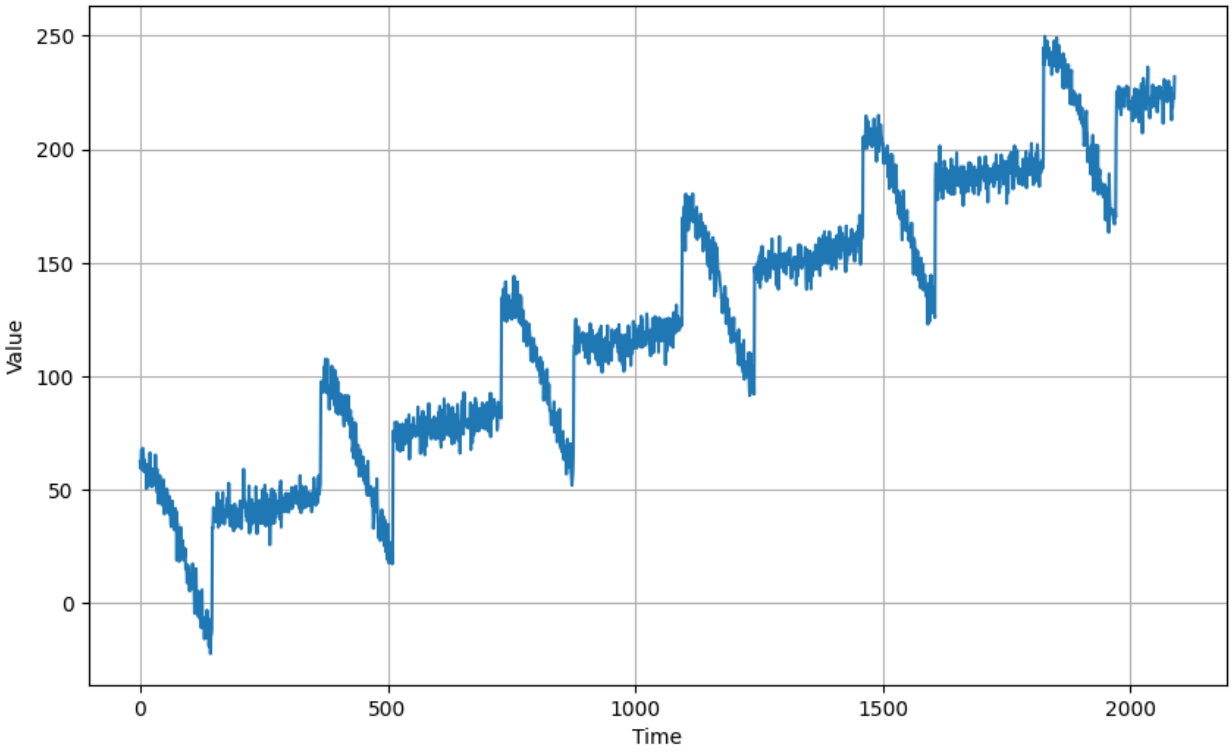
For the synthetic time series data, I developed a sophisticated generator capable of producing synthetic data tailored to specific parameters. This generator is designed to accommodate inputs such as the time period, desired trend, and prescribed seasonality. Leveraging this information, it meticulously crafts synthetic data sets spanning a time horizon equivalent to seven years, encompassing a total of 7 * 365 timesteps.

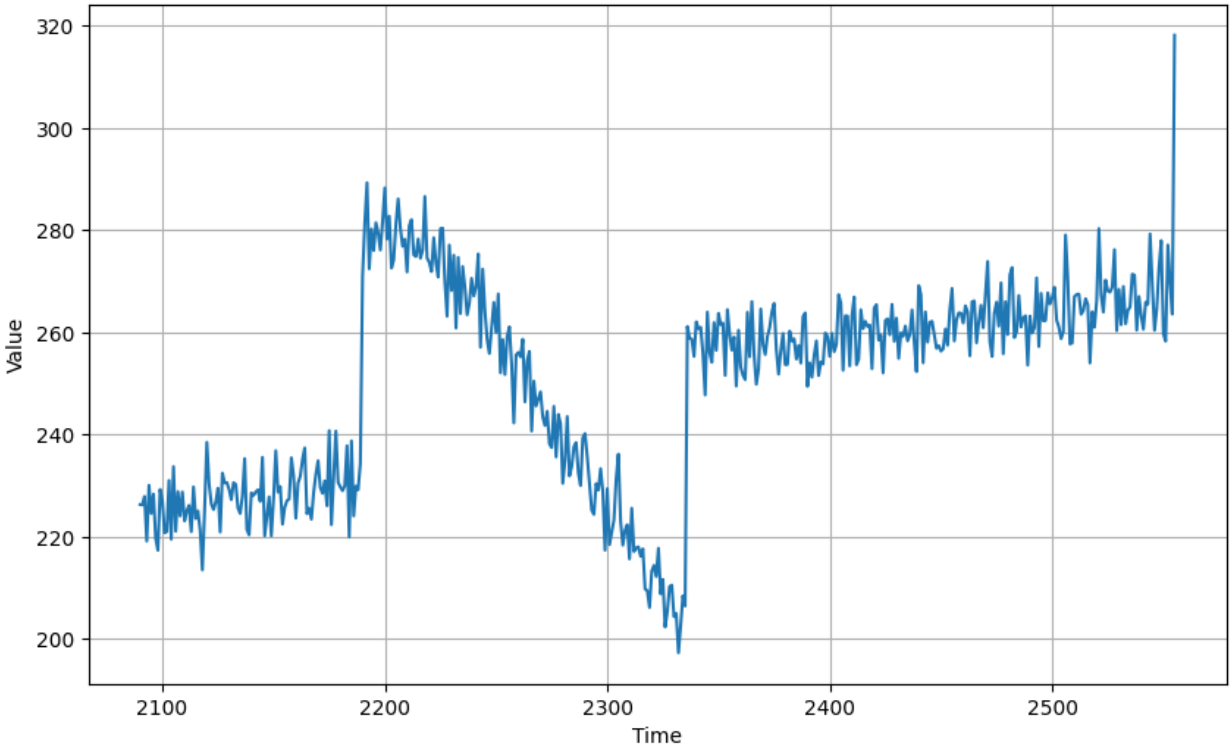
Furthermore, to imbue the synthetic data with a realistic touch, the generator incorporates specific noise levels as per user specifications. This noise, carefully calibrated to emulate real-world fluctuations, enhances the authenticity and complexity of the generated time series. With this robust framework in place, the resulting synthetic data captures the intricate interplay between trend, seasonality, and noise, facilitating comprehensive exploration and analysis of time series forecasting techniques.

The time series look like



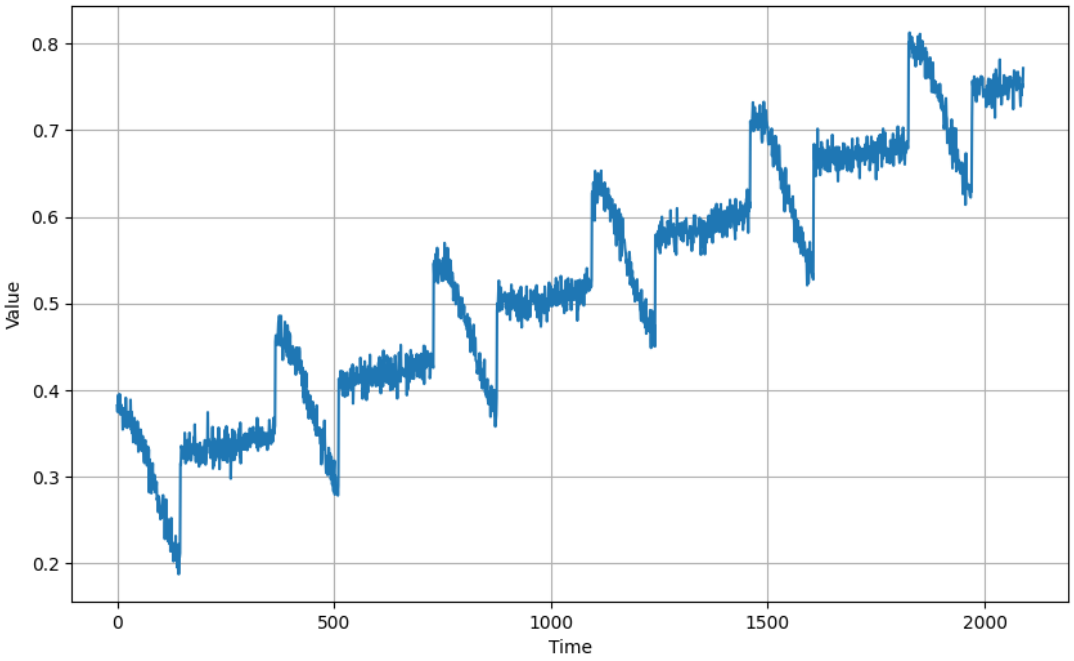
Then it is split into training and testing dataset. I decided to keep as testing dataset the last year, plus 100 days. The two datasets are shown below.

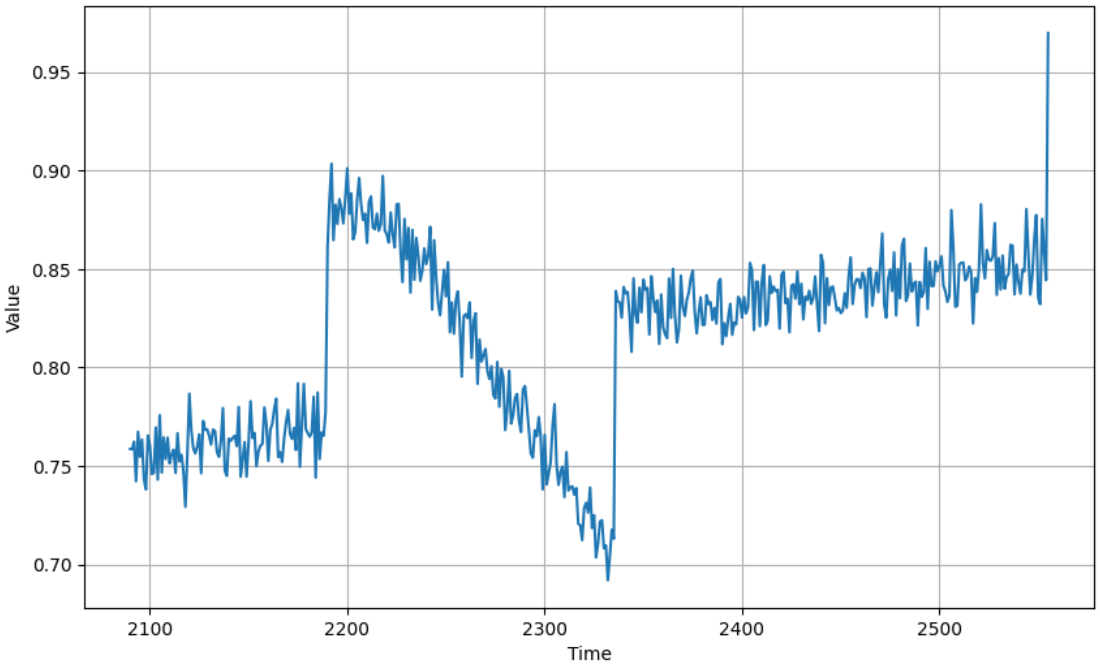

250
200
150
100
50
0
Value
0
500
1000
1500
2000
Time


320
300
280
260
240
220
200
Value
2100
2200
2300
2400
2500
Time

After observing that the tanh activation function struggled with unnormalized data, the next step was to normalize the data. To achieve this, I assumed that the test set should not exceed 50% higher than the full range of the train set and normalized the data to a range between 0 and 1.

With the data normalized, the plots now depict:





After preprocessing the dataset, each data point now corresponds to a one-month frame, represented by a 30-day window. Thus, to predict any value, we analyze the preceding 30 values within the sequence, effectively capturing the temporal dynamics of a month.

Note

My initial objective was to determine the optimal number of layers and the most suitable activation function for this task. Although the ReLU activation function is generally not optimal due to potential issues like exploding or vanishing gradients, in simple neural networks like these, such problems may not arise. Nevertheless, for experimentation purposes, I decided to compare ReLU with another activation function.

Artificial Neural Network

The initial approach focused on employing a basic artificial neural network without memory or recurrent layers to evaluate its performance. This served as a fundamental reference point for comparing other models. Starting with a single input of 50 neurons and a solitary output layer, subsequent layers were added to increase complexity and functionality.

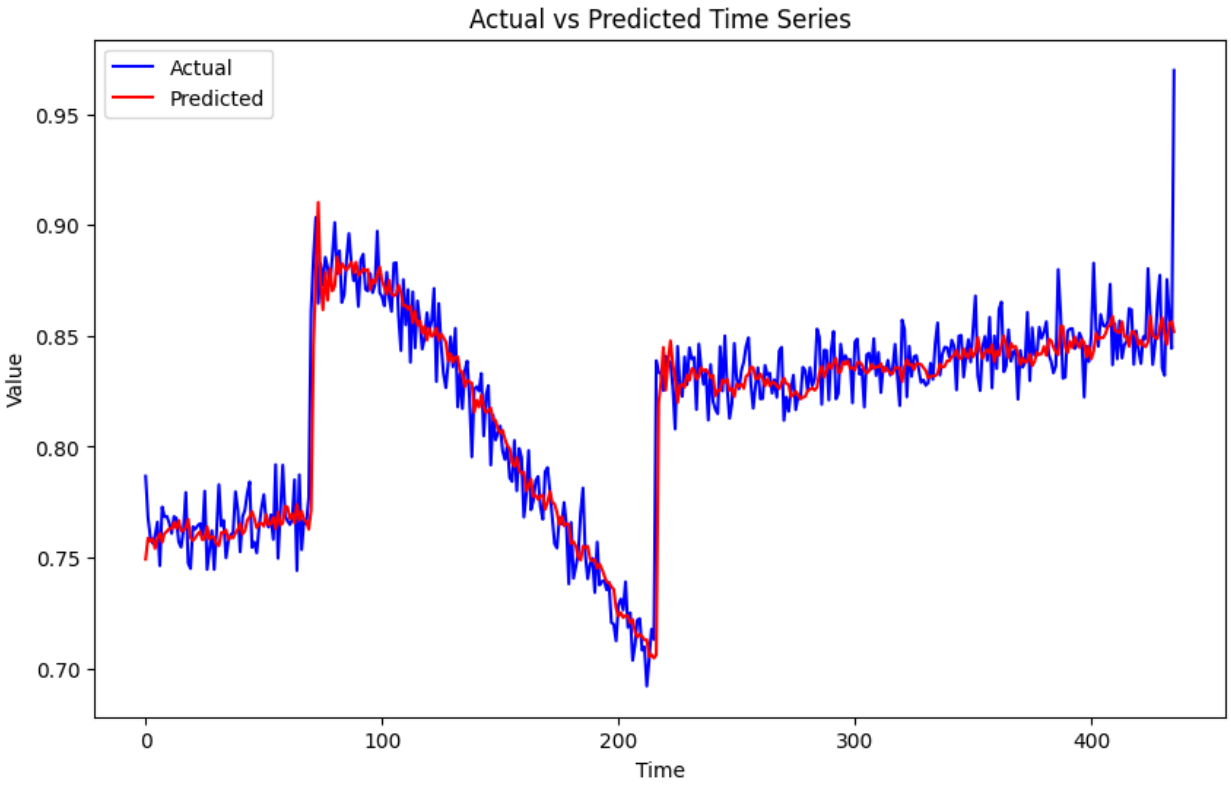
As the process unfolded, additional hidden layers were systematically introduced, enhancing the network's ability to learn and abstract information. This methodical approach enabled a thorough assessment of each modification's influence on performance metrics, yielding valuable insights into the effectiveness of various model architectures.

Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

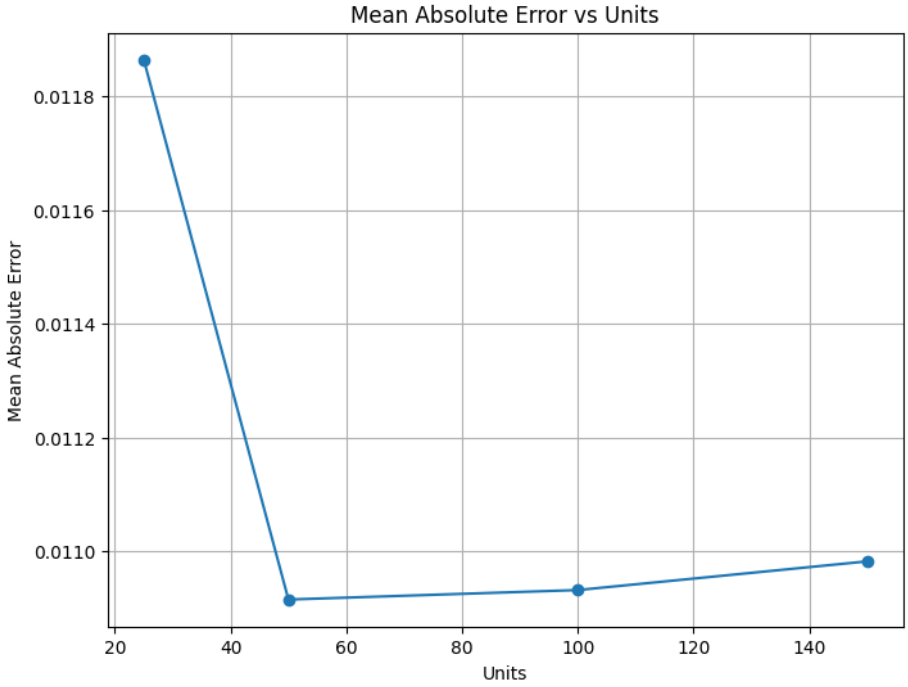
- 0 hidden layers, 50 neurons, tanh activation:loss:2.8610e-04 - mean_absolute_error: 0.0119
- 1 hidden layer, 50 neurons, tanh activation: loss: 2.7323e-04 - mean_absolute_error: 0.0116
- 2 hidden layers, 50 neurons, tanh activation:loss: 2.8707e-04 - mean_absolute_error: 0.0122
- 3 hidden layers, 50 neurons, tanh activation:loss: 3.0519e-04 - mean_absolute_error: 0.0125
- 0 hidden layers, 50 neurons, ReLU activation: loss: 2.3608e-04 - mean_absolute_error: 0.0108
- 1 hidden layer, 50 neurons, ReLU activation: loss: 2.8957e-04 - mean_absolute_error: 0.0118
- 2 hidden layers, 50 neurons, ReLU activation: loss: 4.3986e-04 - mean_absolute_error: 0.0155
- 3 hidden layers, 50 neurons, ReLU activation: loss: 0.0135 - mean_absolute_error: 0.1140

```
Best model found:
Number of Layers: 0
Units: 50
Activation: relu
Validation Loss: 0.00025394276599399745
```

And the predicted timeseries is shown below



The last thing I wanted to see is if different number of neurons in the best model could potentially decrease the error. After training for 25, 50, 100, 150 neurons the MAE error diagram with the units is the below.



We can see that indeed 50 neurons was the optimal approach with mean_absolute_error: 0.0108.

Simple RNN

I began by constructing a neural network with one input Simple RNN layer comprising 50 neurons, followed by another Simple RNN layer comprising 50 neurons at the output, aiming to avoid sequence output, along with a dense layer for regression. I then progressively added one hidden layer at a time to observe the impact on performance. Once I identified the optimal number of layers, my next step involved tuning the number of neurons within each layer to further enhance the model's performance.

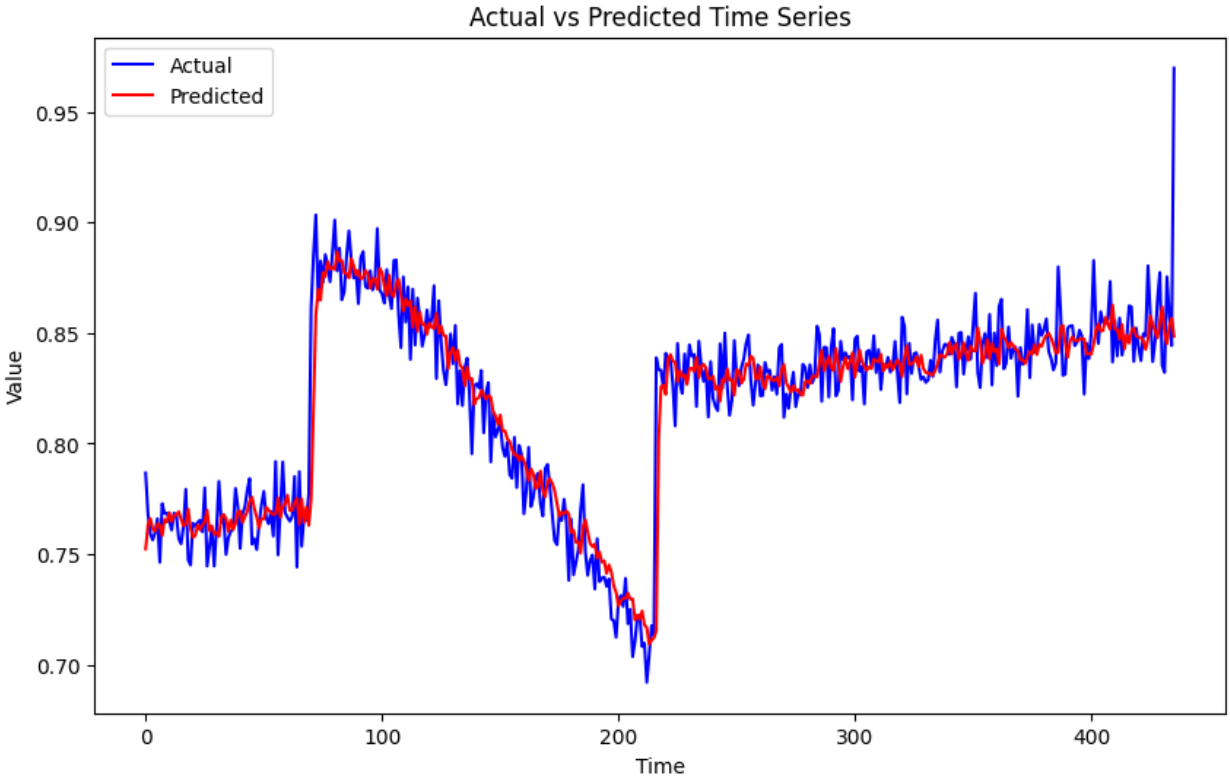
Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

- 0 hidden layers, 50 neurons, tanh activation:loss: 2.8090e-04 - mean_absolute_error: 0.0120
- 1 hidden layer, 50 neurons, tanh activation: loss: 2.8426e-04 - mean_absolute_error: 0.0119
- 2 hidden layers, 50 neurons, tanh activation: loss: 2.8001e-04 - mean_absolute_error: 0.0119
- 3 hidden layers, 50 neurons, tanh activation:loss: 3.1729e-04 - mean_absolute_error: 0.0125
- 0 hidden layers, 50 neurons, ReLU activation: loss: 2.5895e-04 - mean_absolute_error: 0.0112
- 1 hidden layer, 50 neurons, ReLU activation: loss: 2.4769e-04 - mean_absolute_error: 0.0110
- 2 hidden layers, 50 neurons, ReLU activation: loss: 0.0015 - mean_absolute_error: 0.0353
- 3 hidden layers, 50 neurons, ReLU activation: loss: 0.0050 - mean_absolute_error: 0.0670

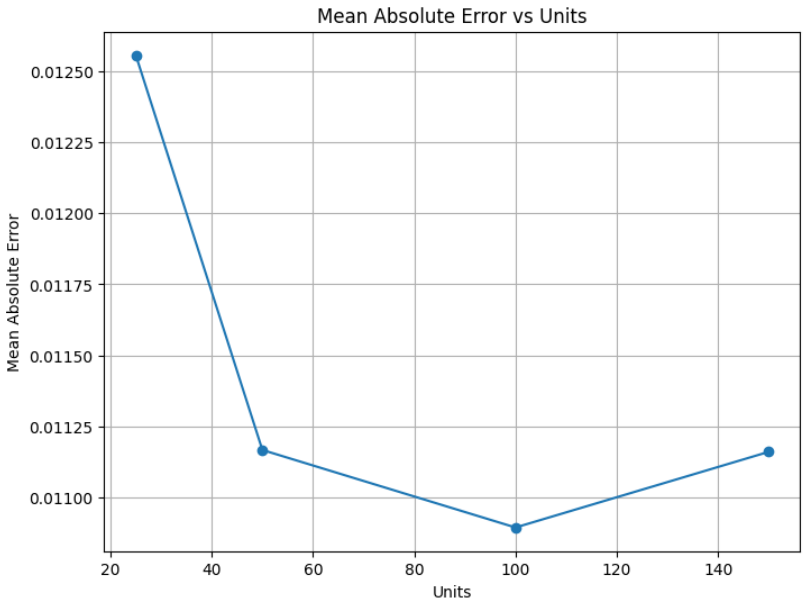
We can observe that the smallest loss is on

```
Best model found:
Number of Layers: 1
Units: 50
Activation: relu
Validation Loss: 0.00026752290432341397
```

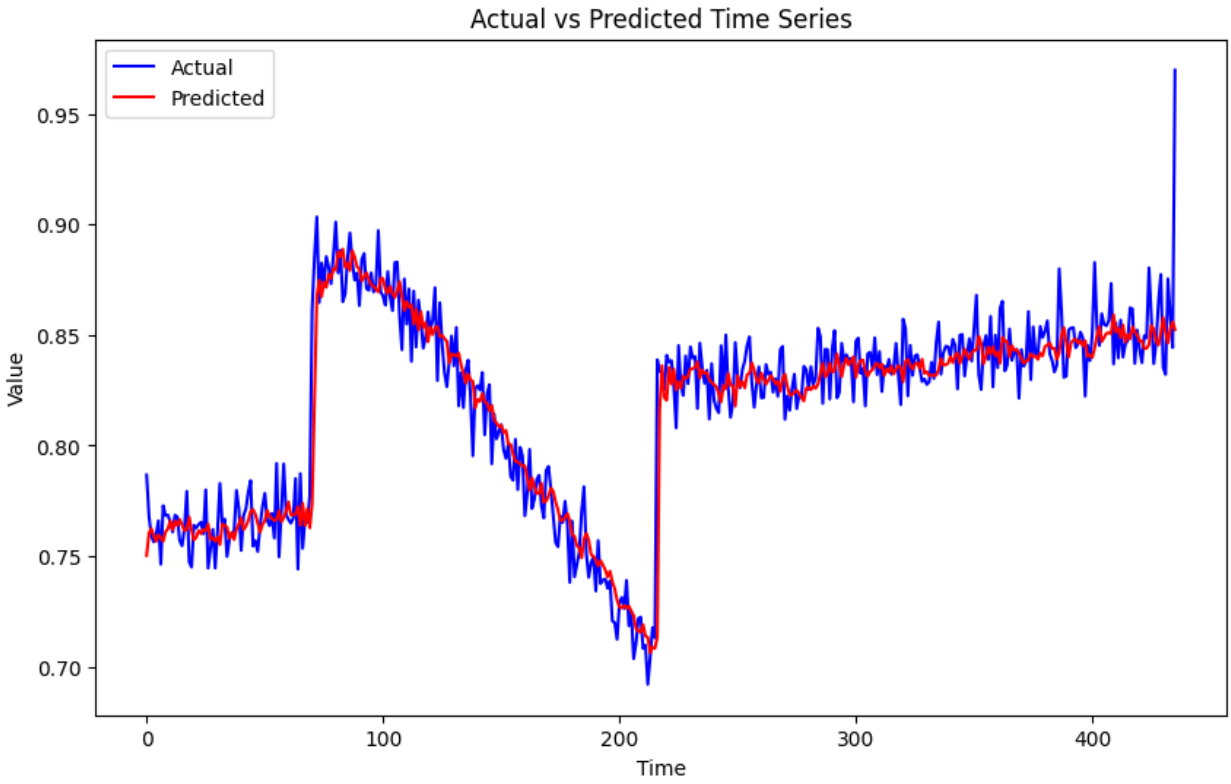
And the predicted timeseries is the one below:



It is a good prediction. The last thing I wanted to see is if different number of neurons in the best model could potentially decrease the error. After training for 25, 50, 100, 150 neurons the MAE error diagram with the units is the below.



We can see that the minimum mae is found for 100 neurons and it is Smallest Validation Mean Absolute Error: 0.010894229. The predicted time series is:



LSTM

I began by constructing a neural network with one input LSTM layer comprising 50 neurons, followed by another LSTM layer comprising 50 neurons at the output, aiming to avoid sequence output, along with a dense layer for regression. I then progressively added one hidden layer at a time to observe the impact on performance. Once I identified the optimal number of layers, my next step involved tuning the number of neurons within each layer to further enhance the model's performance.

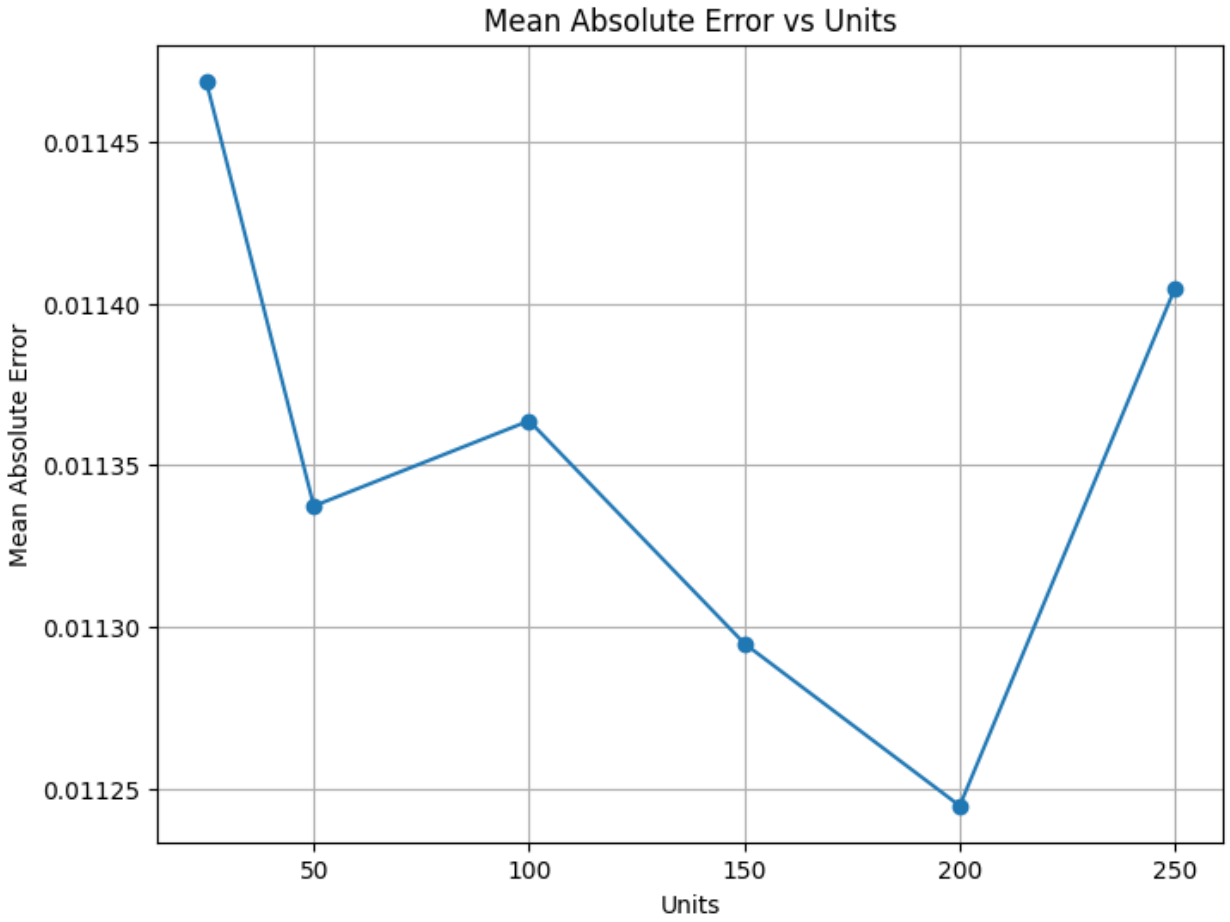
Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

- 0 hidden layers, 50 neurons, tanh activation:loss2.4350e-04 - mean_absolute_error: 0.0110
- 1 hidden layer, 50 neurons, tanh activation: loss: 2.4251e-04 - mean_absolute_error: 0.0109
- 2 hidden layers, 50 neurons, tanh activation: loss: 2.4525e-04 - mean_absolute_error: 0.0110
- 3 hidden layers, 50 neurons, tanh activation:loss: 2.5503e-04 - mean_absolute_error: 0.0112
- 0 hidden layers, 50 neurons, ReLU activation:loss: 3.0809e-04 - mean_absolute_error: 0.0120
- 1 hidden layer, 50 neurons, ReLU activation: loss: 3.2885e-04 - mean_absolute_error: 0.0121
- 2 hidden layers, 50 neurons, ReLU activation: loss: 9.6451e-04 - mean_absolute_error: 0.0195
- 3 hidden layers, 50 neurons, ReLU activation:loss: 9.4135e-04 - mean_absolute_error: 0.0188

We can observe that the smallest loss is on

```
Best model found:
Number of Layers: 1
Units: 50
Activation: tanh
Validation Loss: 0.0002653776609804481
```

Last but not least is that, I wanted to see, if different number of neurons in the best model could potentially decrease the error. After training for 25, 50, 100, 150, 200, 250 neurons the MAE error diagram with the units is the below.



GRU

I began by constructing a neural network with one input GRU layer comprising 50 neurons, followed by another GRU layer comprising 50 neurons at the output, aiming to avoid sequence output, along with a dense layer for regression. I then progressively added one hidden layer at a time to observe the impact on performance. Once I identified the optimal number of layers, my next step involved tuning the number of neurons within each layer to further enhance the model's performance.

Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

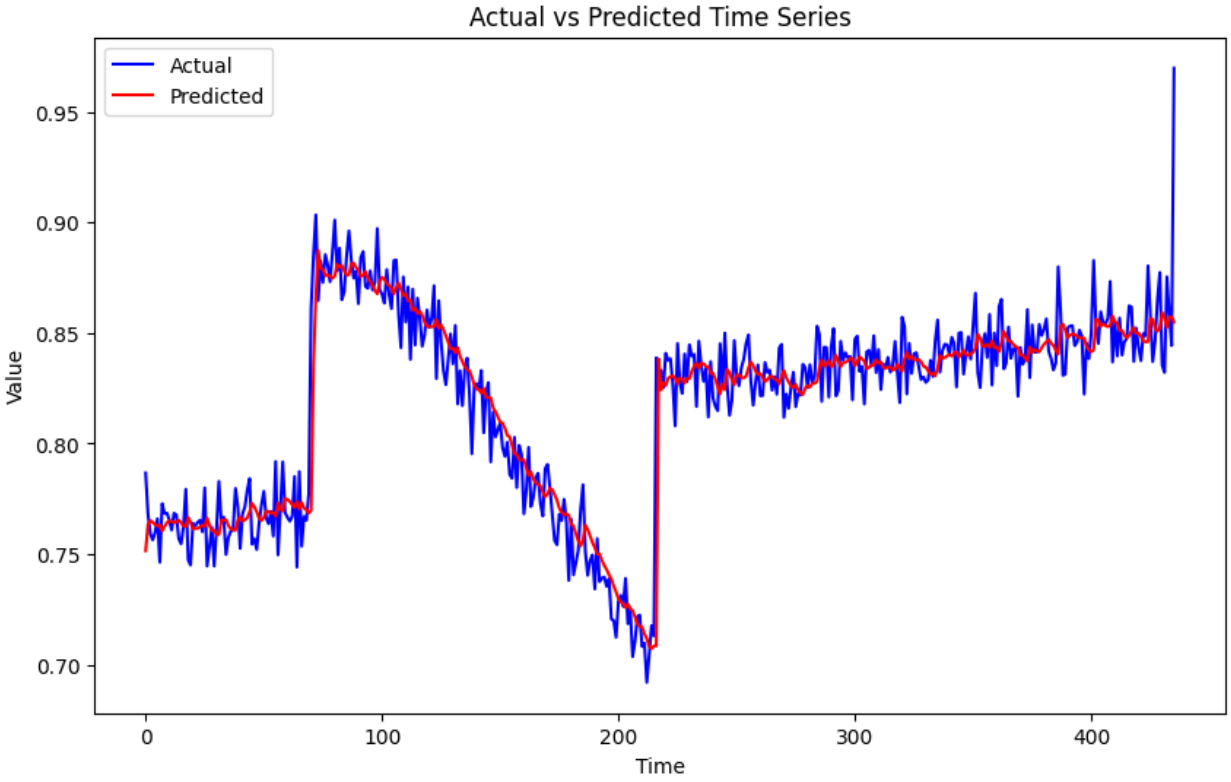
One note is that this kind of Neural Network takes 4-5 times more time to get trained than the rest.

- 0 hidden layers, 50 neurons, tanh activation:loss: 2.3867e-04 - mean_absolute_error: 0.0108
- 1 hidden layer, 50 neurons, tanh activation: loss: 2.3621e-04 - mean_absolute_error: 0.0107
- 2 hidden layers, 50 neurons, tanh activation: loss: 2.3607e-04 - mean_absolute_error: 0.0107
- 3 hidden layers, 50 neurons, tanh activation: loss: 2.3774e-04 - mean_absolute_error: 0.0107
- 0 hidden layers, 50 neurons, ReLU activation:loss: 3.6244e-04 - mean_absolute_error: 0.0125
- 1 hidden layer, 50 neurons, ReLU activation: loss: 4.4565e-04 - mean_absolute_error: 0.0136
- 2 hidden layers, 50 neurons, ReLU activation: loss: 4.4565e-04 - mean_absolute_error: 0.0136
- 3 hidden layers, 50 neurons, ReLU activation: loss: 6.5695e-04 - mean_absolute_error: 0.0154

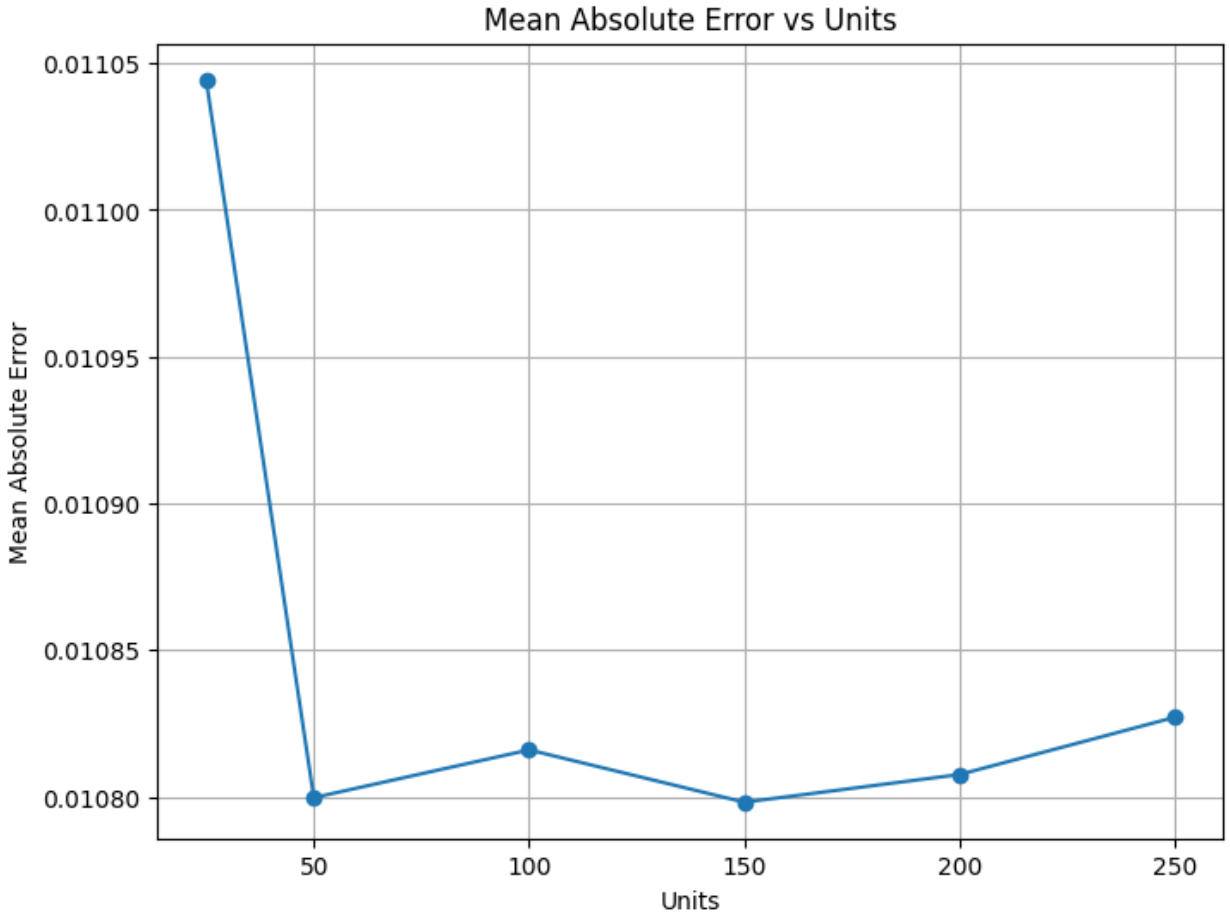
We can observe that the smallest loss is on

```
Best model found:
Number of Layers: 2
Units: 50
Activation: tanh
Validation Loss: 0.00025384765467606485
```

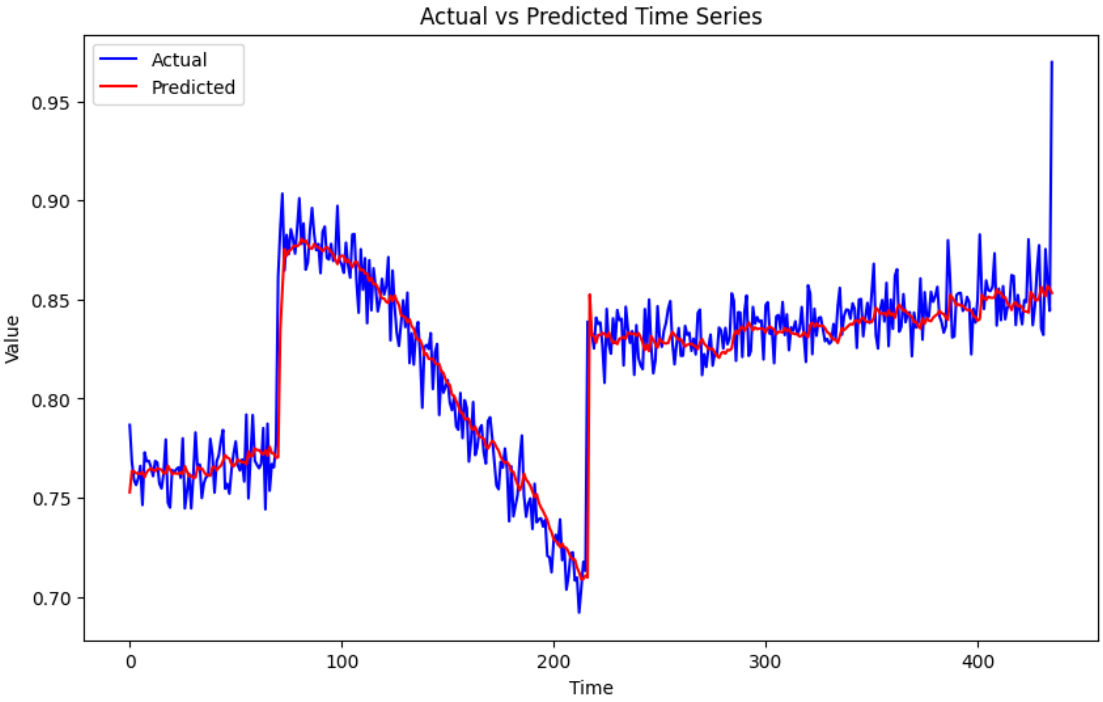
And the predicted timeseries is shown below:



Last but not least is that, I wanted to see, if different number of neurons in the best model could potentially decrease the error. After training for 25, 50, 100, 150 neurons the MAE error diagram with the units is the below.



And the time series produced is :



Best model and conclusions

In our investigation, the GRU (Gated Recurrent Unit) emerged as the optimal neural network architecture for Bitcoin price prediction, showcasing robust efficiency across all tested configurations. Despite its effectiveness, it's worth noting that GRU models necessitated a longer training duration compared to other architectures. This trade-off between training time and performance warrants consideration in practical applications.

Moreover, our analysis underscored the superior performance of the tanh activation function over relu activation across various architectures. The choice of tanh not only contributed to better overall results but also mitigated issues related to exploding or vanishing gradients, ensuring stable training dynamics.

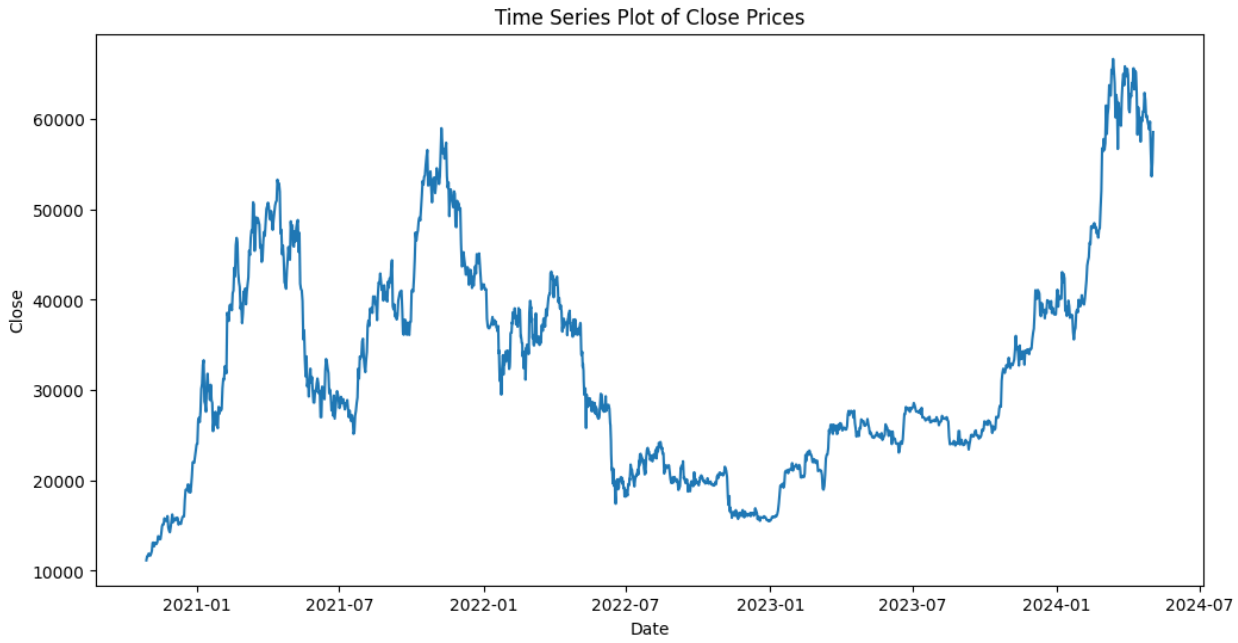
Additionally, the simplicity of the neural network architectures played a crucial role in their efficiency. By adopting parsimonious designs, we avoided unnecessary complexity and potential overfitting, resulting in models that were robust and effective in capturing the underlying patterns of Bitcoin price dynamics.

Overall, our findings highlight the importance of selecting appropriate neural network architectures and activation functions while considering trade-offs such as training time and complexity. These insights provide valuable guidance for developing accurate and reliable models for cryptocurrency market analysis.

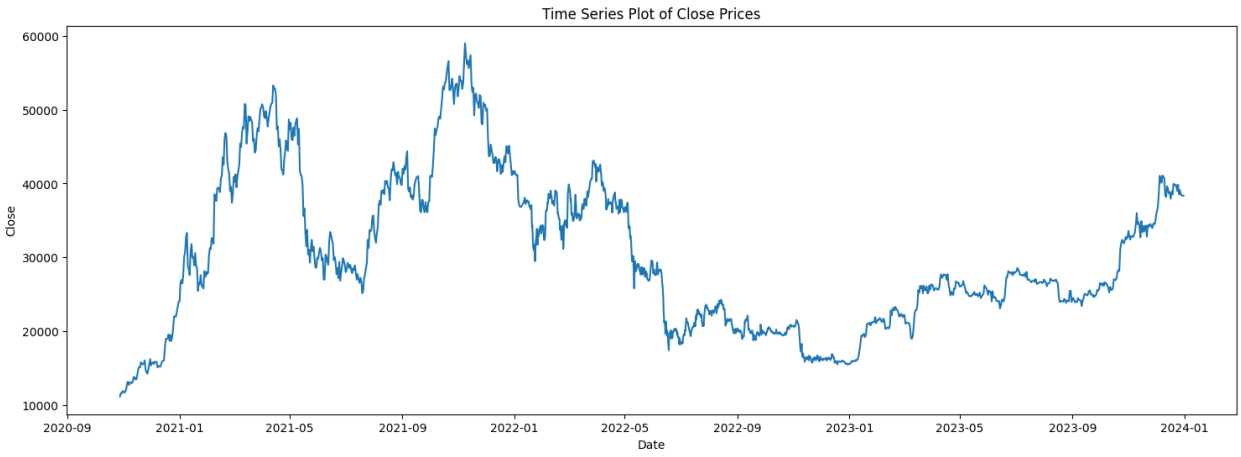
Bitcoin Timeseries Prediction

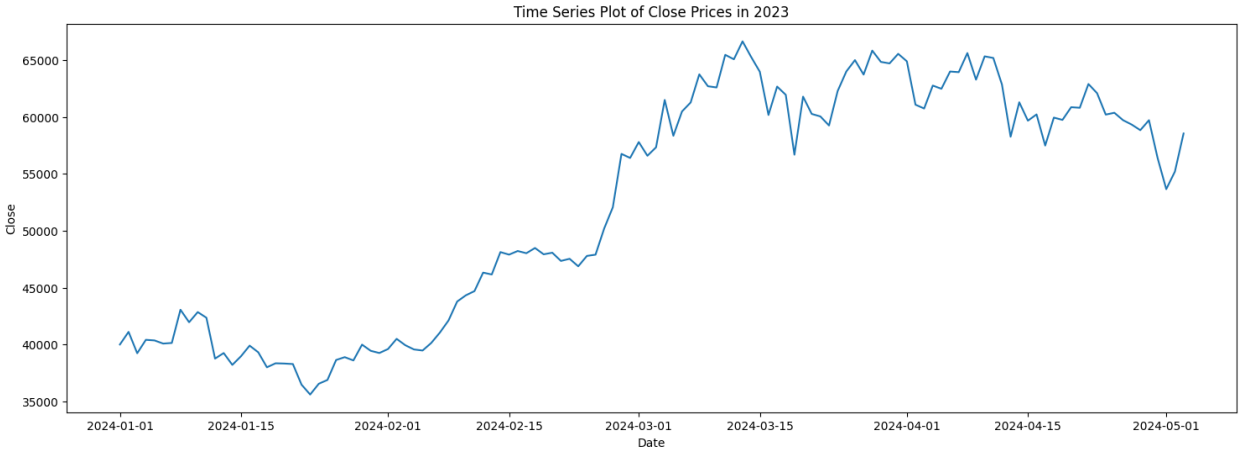
For this phase, the objective shifted towards evaluating model performance using real-world data, departing from synthetic data with predetermined trends and seasonality. The challenge here is to work with a smaller dataset, only for 4 years. The chosen dataset for this purpose was sourced from: https://www.cryptodatadownload.com/. It begins on October 28, 2020, with Bitcoin's closing price recorded at $11,153.76. This dataset is continually updated, providing an ongoing stream of data for analysis and prediction of Bitcoin's closing price.

We can see the graph of closing price of the bitcoin in time.



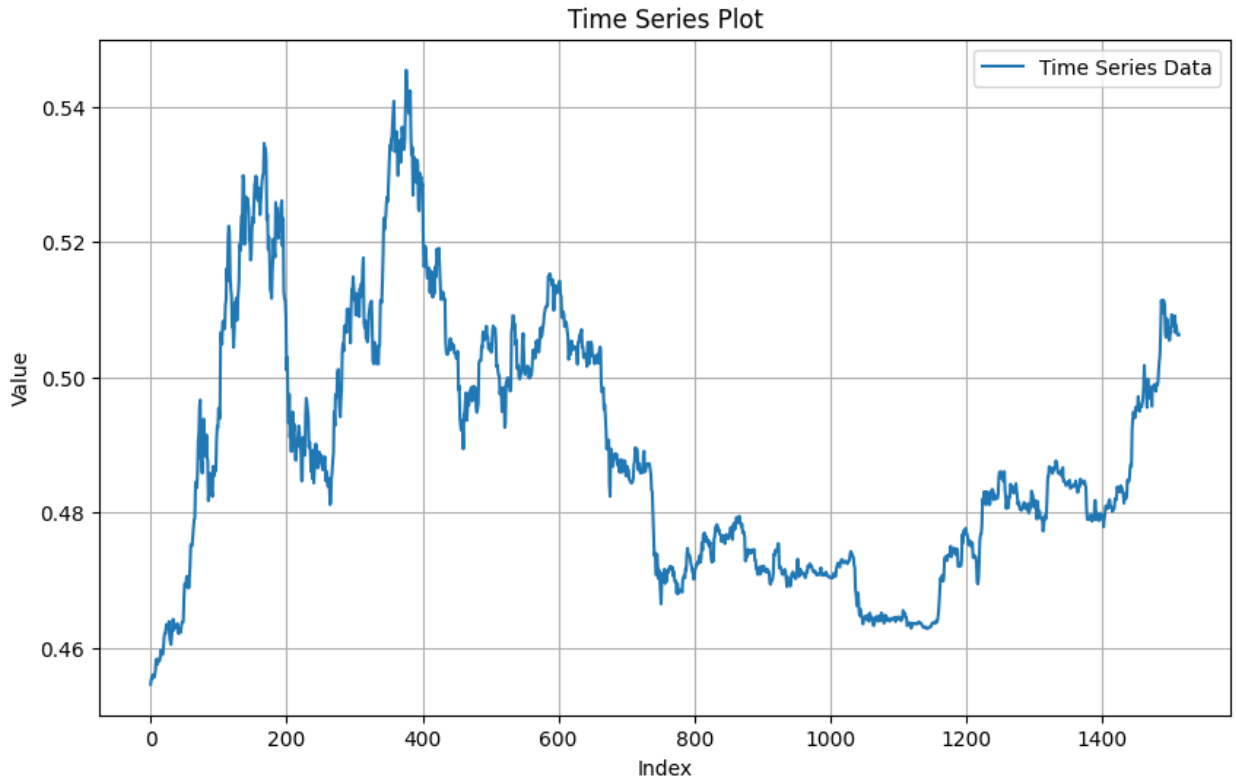
This time I decided to predict for the year 2024. So, I split the training set and test set accordingly.

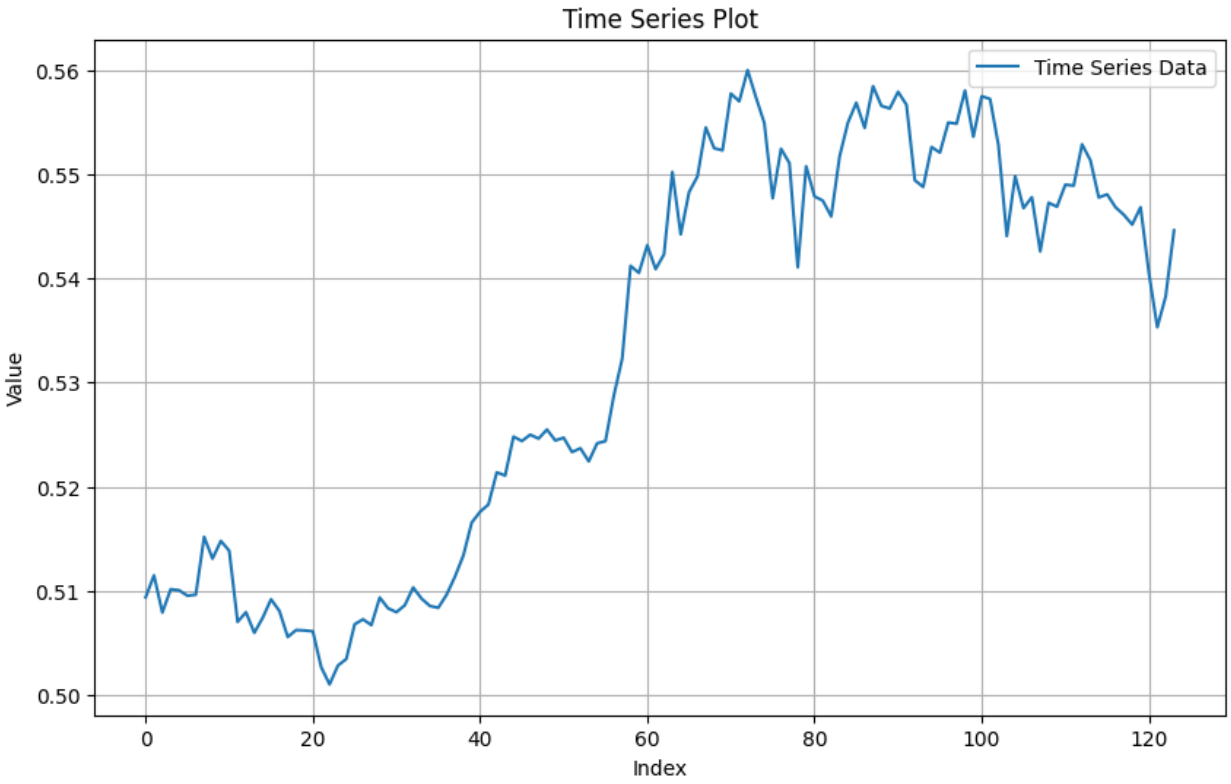




I opted for a broader normalization margin this time due to Bitcoin's inherent volatility. To achieve this, I calculated the range of the training set and multiplied it by 5, establishing this expanded range as the new 0-1 normalization scale. This decision was motivated by the need to accommodate Bitcoin's significant price fluctuations and ensure that the model effectively captures the full spectrum of price movements for more accurate predictions.

According to this the train set and test set are shown below:





Finally the window size of this dataset is set again to 30.

Artificial Neural Network

Once again I decided to start working with a simple RNN, in order to evaluate its performance, without any memory layers. This served as a fundamental reference point for comparing other models. Starting with a single input of 50 neurons and a solitary output layer, subsequent layers were added to increase complexity and functionality.

As the process unfolded, additional hidden layers were systematically introduced, enhancing the network's ability to learn and abstract information. This methodical approach enabled a thorough assessment of each modification's influence on performance metrics, yielding valuable insights into the effectiveness of various model architectures.

Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

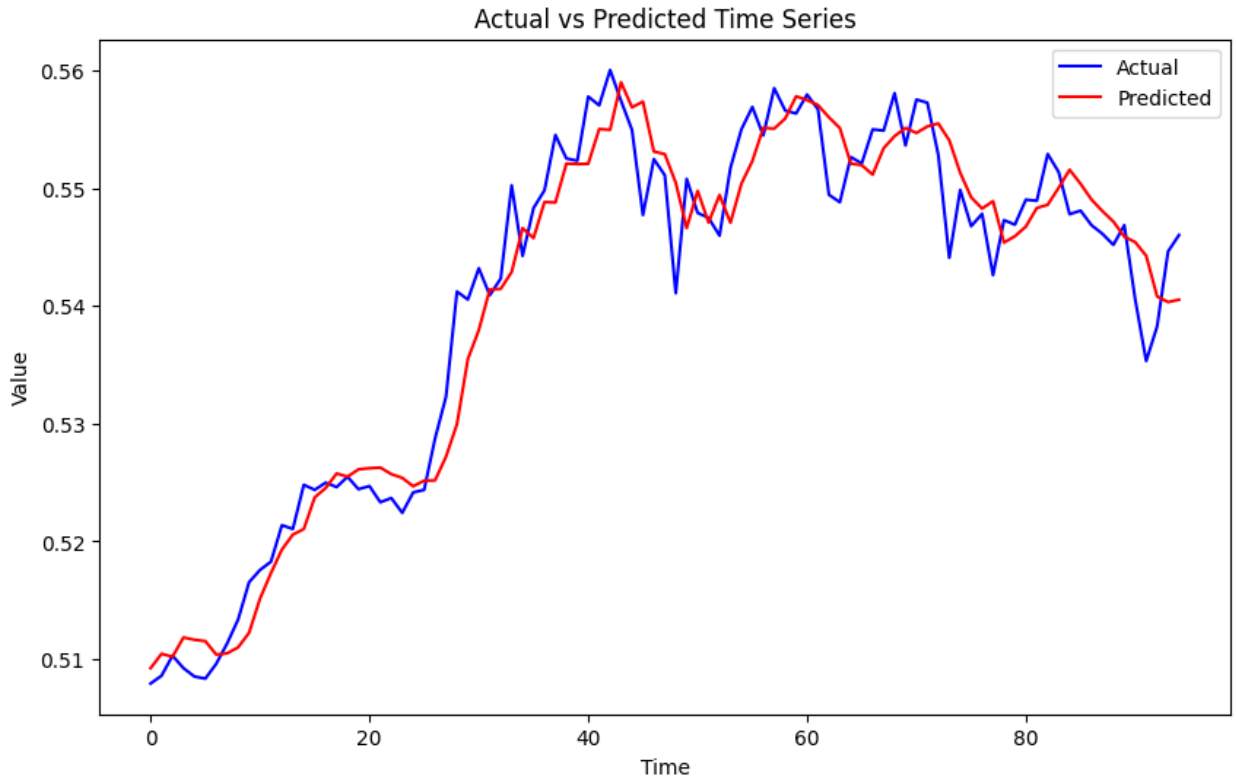
- 0 hidden layers, 50 neurons, tanh activation: loss: 1.3085e-05 - mean_absolute_error: 0.0027
- 1 hidden layer, 50 neurons, tanh activation: loss: 1.2859e-05 - mean_absolute_error: 0.0027
- 2 hidden layers, 50 neurons, tanh activation: loss: 1.2998e-05 - mean_absolute_error: 0.0026
- 3 hidden layers, 50 neurons, tanh activation: loss: 1.3847e-05 - mean_absolute_error: 0.0028
- 0 hidden layers, 50 neurons, ReLU activation: loss: 1.4418e-05 - mean_absolute_error: 0.0029

- 1 hidden layer, 50 neurons, ReLU activation: loss: 1.5563e-05 - mean_absolute_error: 0.0029
- 2 hidden layers, 50 neurons, ReLU activation: loss: 2.0242e-05 - mean_absolute_error: 0.0033
- 3 hidden layers, 50 neurons, ReLU activation: loss: 0.0014 - mean_absolute_error: 0.0367

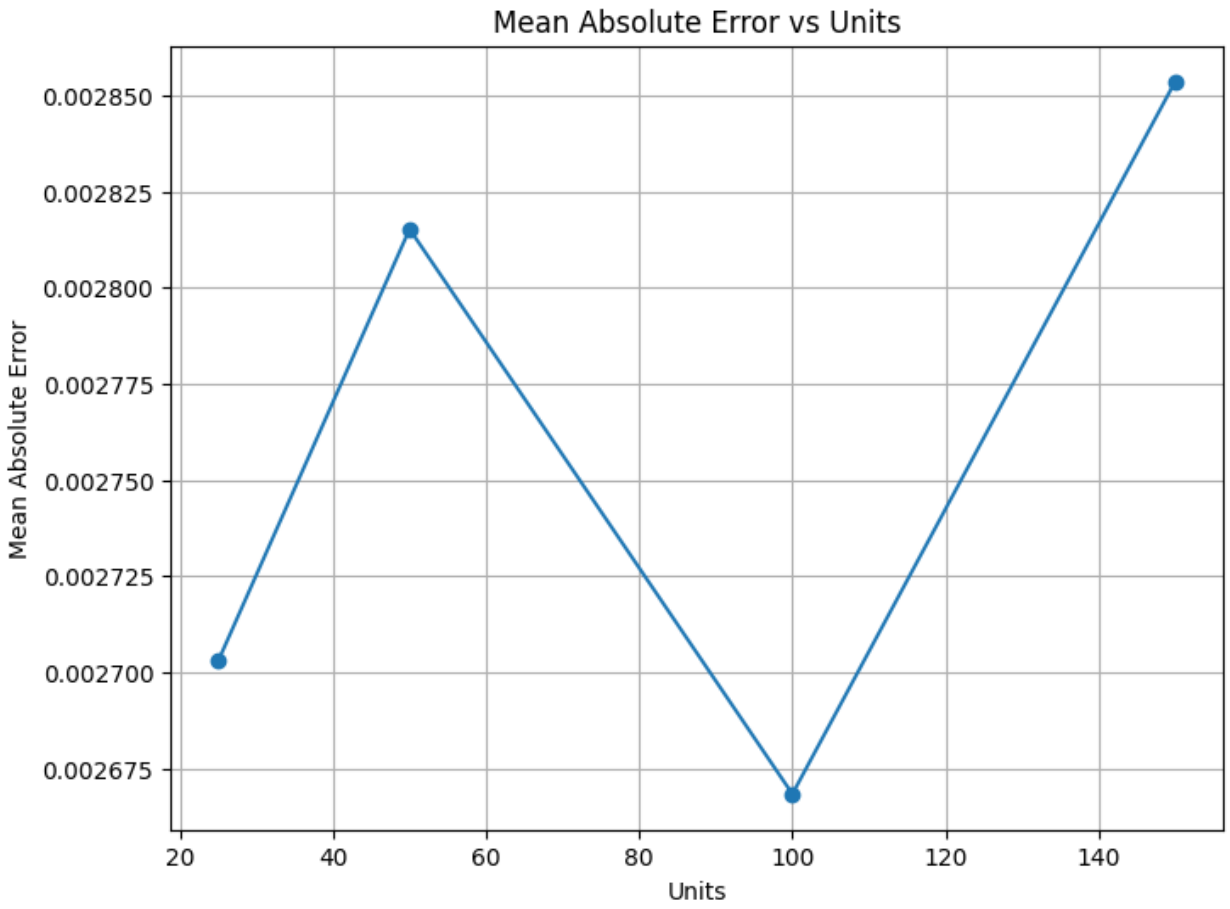
According to these the best model is the one shown below:

```
Best model found:
Number of Layers: 1
Units: 50
Activation: tanh
Validation Loss: 1.3685216799785849e-05
```

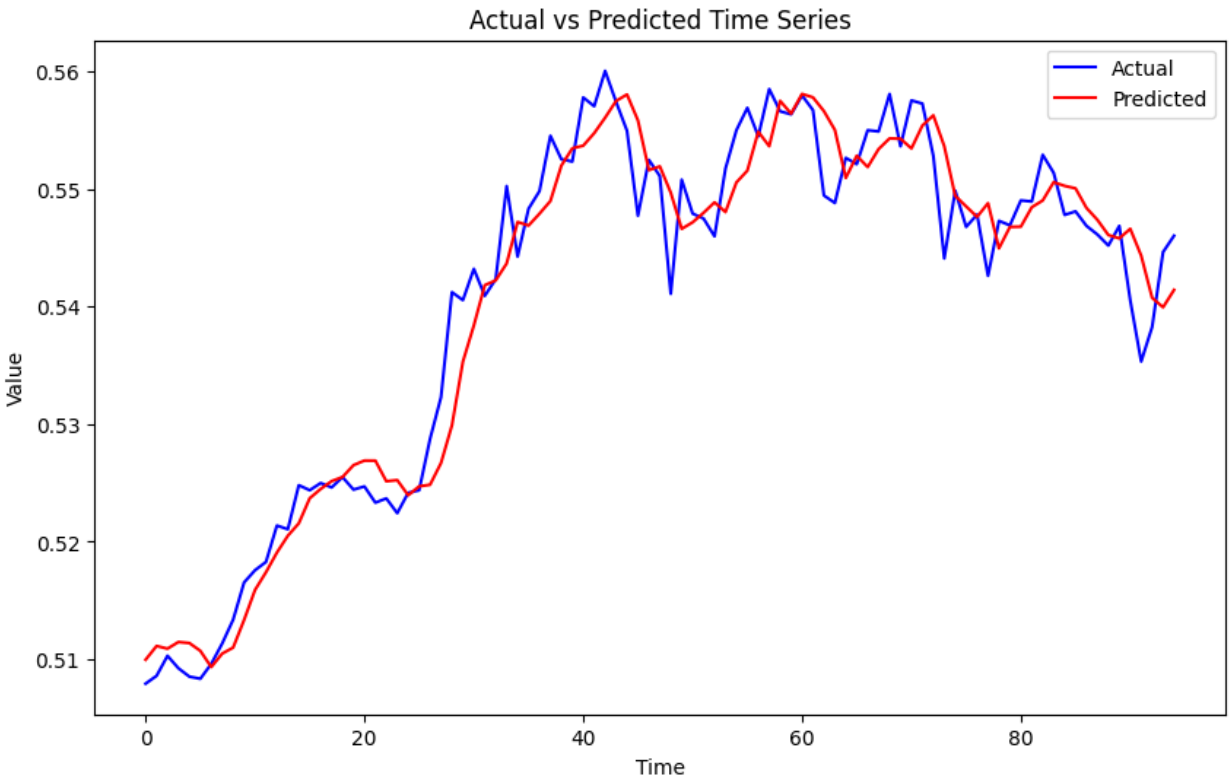
And the time series produced



Now let's see what number of neurons is the optimal for one layer. I tried 25, 50, 100, 150 units per layer and the plot of mean absolute error per units is shown below:



So the best model is the one with 100 units, and the predicted time series is shown below:



Simple RNN

I began by constructing a neural network with one input Simple RNN layer comprising 50 neurons, followed by another Simple RNN layer comprising 50 neurons at the output, aiming to avoid sequence output, along with a dense layer for regression. I then progressively added one hidden layer at a time to observe the impact on performance. Once I identified the optimal number of layers, my next step involved tuning the number of neurons within each layer to further enhance the model's performance.

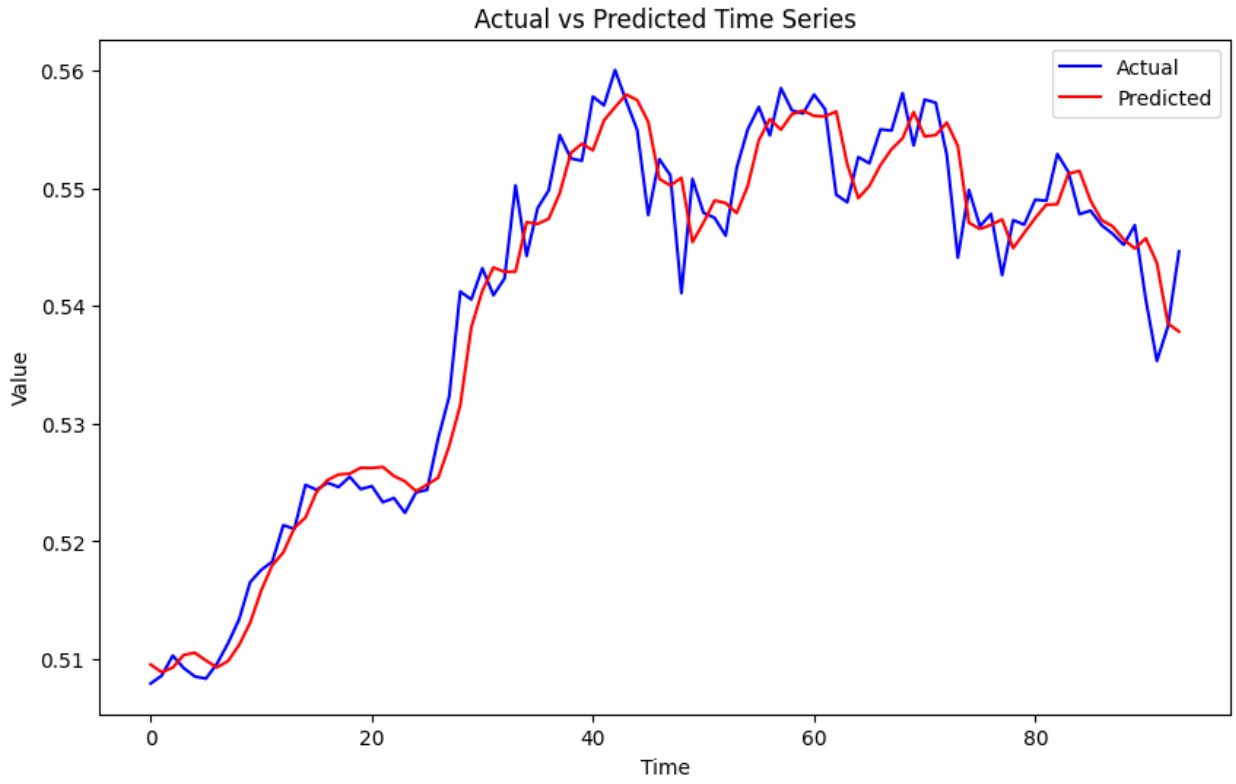
Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

- 0 hidden layers, 50 neurons, tanh activation: loss: 1.0341e-05 - mean_absolute_error: 0.0023
- 1 hidden layer, 50 neurons, tanh activation: loss: 1.0684e-05 - mean_absolute_error: 0.0023
- 2 hidden layers, 50 neurons, tanh activation: loss: 9.8651e-06 - mean_absolute_error: 0.0023
- 3 hidden layers, 50 neurons, tanh activation: loss: 9.8765e-06 - mean_absolute_error: 0.0023
- 4 hidden layers, 50 neurons, tanh activation: loss: 9.8549e-06 - mean_absolute_error: 0.0024
- 5 hidden layers, 50 neurons, tanh activation: loss: 1.0905e-04 - mean_absolute_error: 0.0083
- 0 hidden layers, 50 neurons, ReLU activation: loss: 1.1026e-05 - mean_absolute_error: 0.0024
- 1 hidden layer, 50 neurons, ReLU activation: loss: 1.0701e-05 - mean_absolute_error: 0.0023
- 2 hidden layers, 50 neurons, ReLU activation: loss: 3.0462e-05 - mean_absolute_error: 0.0045

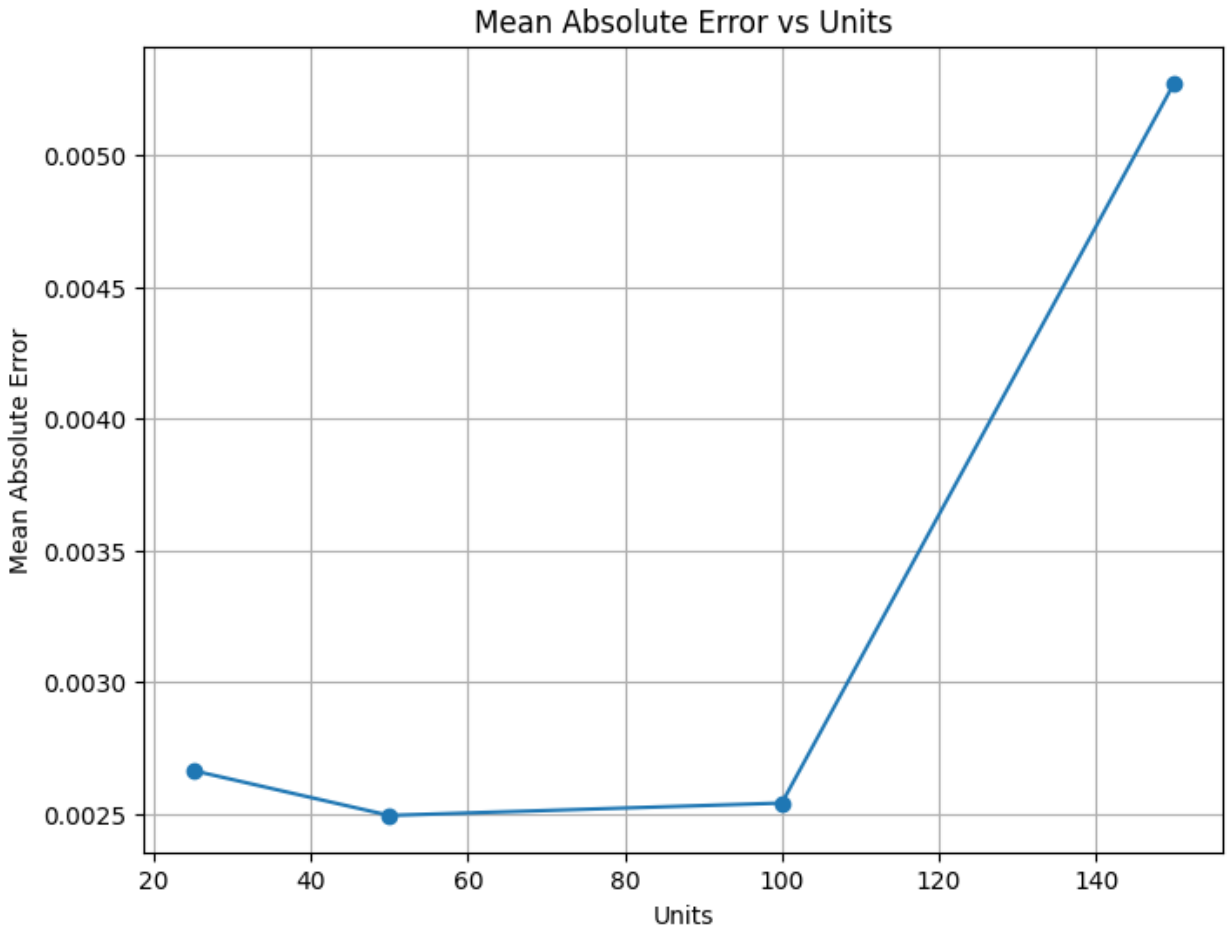
- 3 hidden layers, 50 neurons, ReLU activation: loss: 3.2531e-04 - mean_absolute_error: 0.0173
- 4 hidden layers, 50 neurons, ReLU activation: loss: 5.7655e-04 - mean_absolute_error: 0.0230
- 5 hidden layers, 50 neurons, ReLU activation: loss: 0.0012 - mean_absolute_error: 0.0329

```
We can observe that the smallest loss is on
Best model found:
Number of Layers: 3
Units: 50
Activation: tanh
Validation Loss: 1.1229667506995611e-05
```

And the time series produced is



Now I tried to find the optimal number of neurons in each layer. After training for 25, 50, 100, 250 neurons the plot is shown below :



LSTM

I began by constructing a neural network with one input LSTM layer comprising 50 neurons, followed by another LSTM layer comprising 50 neurons at the output, aiming to avoid sequence output, along with a dense layer for regression. I then progressively added one hidden layer at a time to observe the impact on performance. Once I identified the optimal number of layers, my next step involved tuning the number of neurons within each layer to further enhance the model's performance.

Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

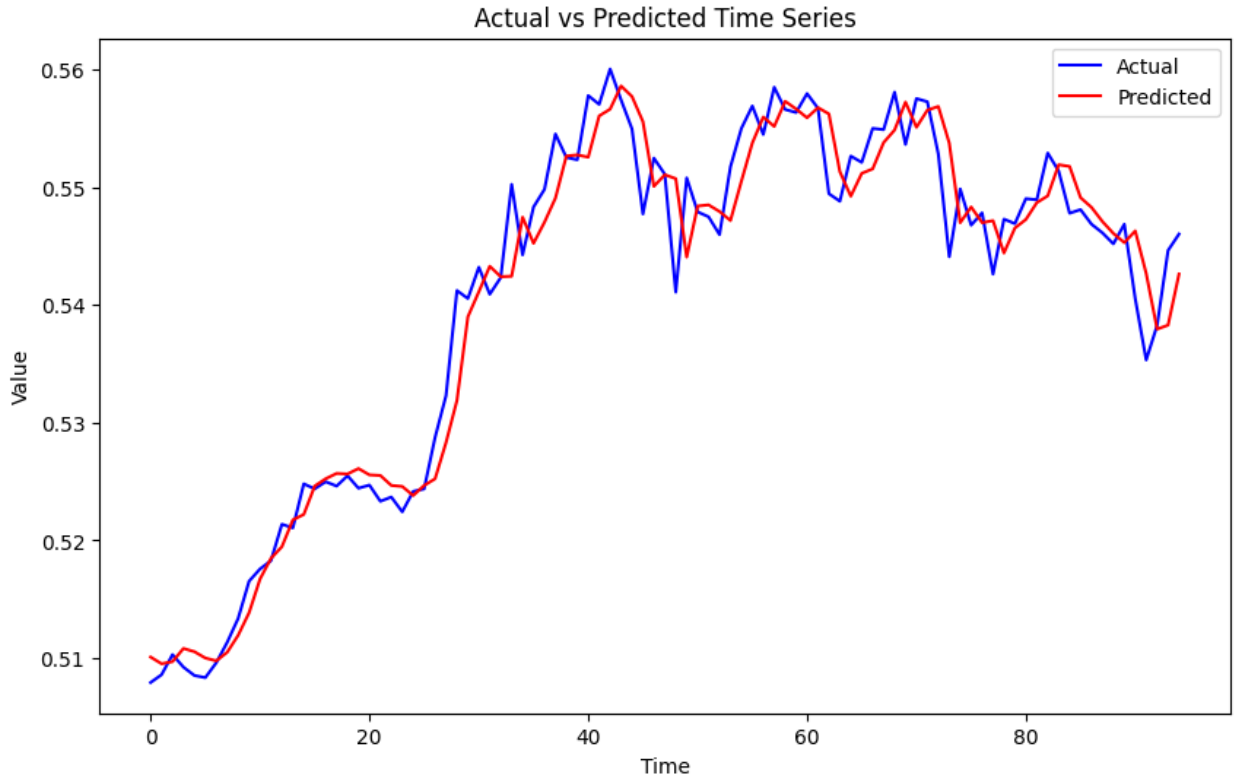
- 0 hidden layers, 50 neurons, tanh activation: loss: 9.8168e-06 - mean_absolute_error: 0.0022
- 1 hidden layer, 50 neurons, tanh activation: loss: 9.8617e-06 - mean_absolute_error: 0.0022

- 2 hidden layers, 50 neurons, tanh activation: loss: 9.6432e-06 - mean_absolute_error: 0.0022
- 3 hidden layers, 50 neurons, tanh activation: loss: 1.0201e-05 - mean_absolute_error: 0.0023
- 0 hidden layers, 50 neurons, ReLU activation: loss: 1.0644e-05 - mean_absolute_error: 0.0024
- 1 hidden layer, 50 neurons, ReLU activation: loss: 1.1409e-05 - mean_absolute_error: 0.0025
- 2 hidden layers, 50 neurons, ReLU activation: loss: 6.2369e-05 - mean_absolute_error: 0.0063
- 3 hidden layers, 50 neurons, ReLU activation: loss: 4.1021e-04 - mean_absolute_error: 0.0194

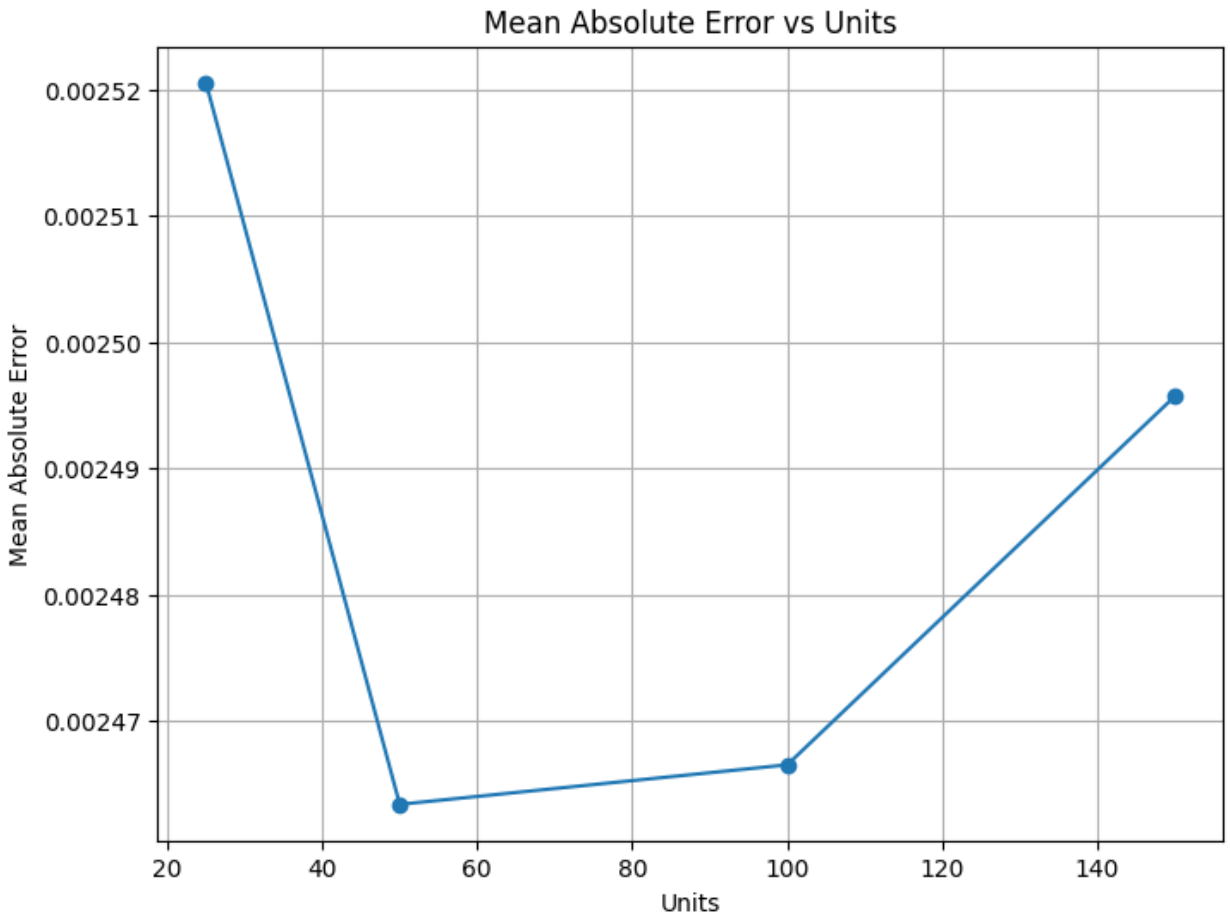
We can observe that the smallest loss is on

```
Best model found:
Number of Layers: 2
Units: 50
Activation: tanh
Validation Loss: 1.1172970516781788e-05
```

And the predicted time series



Last but not least is that, I wanted to see, if different number of neurons in the best model could potentially decrease the error. After training for 25, 50, 100, 150 neurons the MAE error diagram with the units is the below.



GRU

I began by constructing a neural network with one input GRU layer comprising 50 neurons, followed by another GRU layer comprising 50 neurons at the output, aiming to avoid sequence output, along with a dense layer for regression. I then progressively added one hidden layer at a time to observe the impact on performance. Once I identified the optimal number of layers, my next step involved tuning the number of neurons within each layer to further enhance the model's performance.

Subsequently, I trained the model for 1000 epochs, with a batch size of 64, and utilized the Adam optimizer with a learning rate of 0.001. I chose mean squared error as the loss function to be minimized, while monitoring the mean absolute error during training. Additionally, I implemented a mechanism to save the best model throughout every epoch.

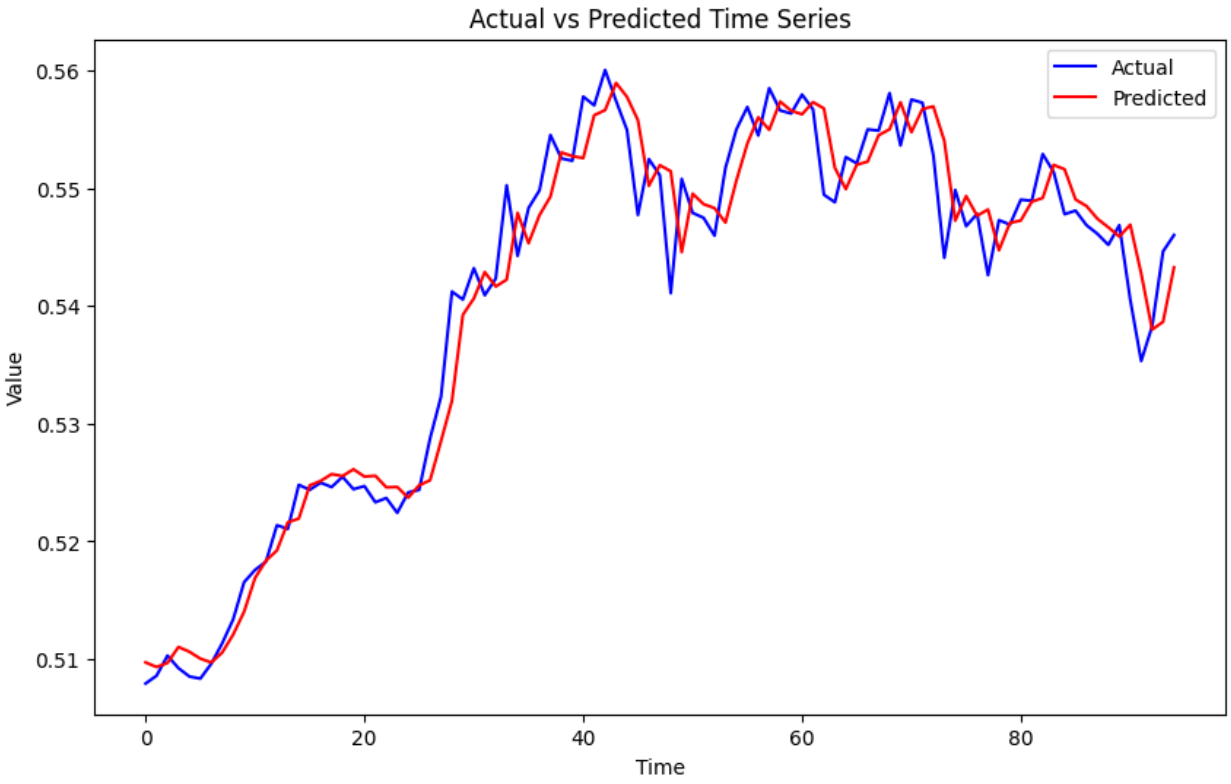
- 0 hidden layers, 50 neurons, tanh activation: loss: 9.9002e-06 - mean_absolute_error: 0.0023
-
- 1 hidden layer, 50 neurons, tanh activation: loss: 1.0215e-05 - mean_absolute_error: 0.0023
-
- 2 hidden layers, 50 neurons, tanh activation: loss: 1.0646e-05 - mean_absolute_error: 0.0023
- 3 hidden layers, 50 neurons, tanh activation: loss: 1.0166e-05 - mean_absolute_error: 0.0023
- 0 hidden layers, 50 neurons, ReLU activation: loss: 1.1843e-05 - mean_absolute_error: 0.0024

- 1 hidden layer, 50 neurons, ReLU activation: loss: 1.0911e-05 - mean_absolute_error: 0.0024
- 2 hidden layers, 50 neurons, ReLU activation: loss: 1.4355e-05 - mean_absolute_error: 0.0029
- 3 hidden layers, 50 neurons, ReLU activation: loss: 3.0186e-05 - mean_absolute_error: 0.0043

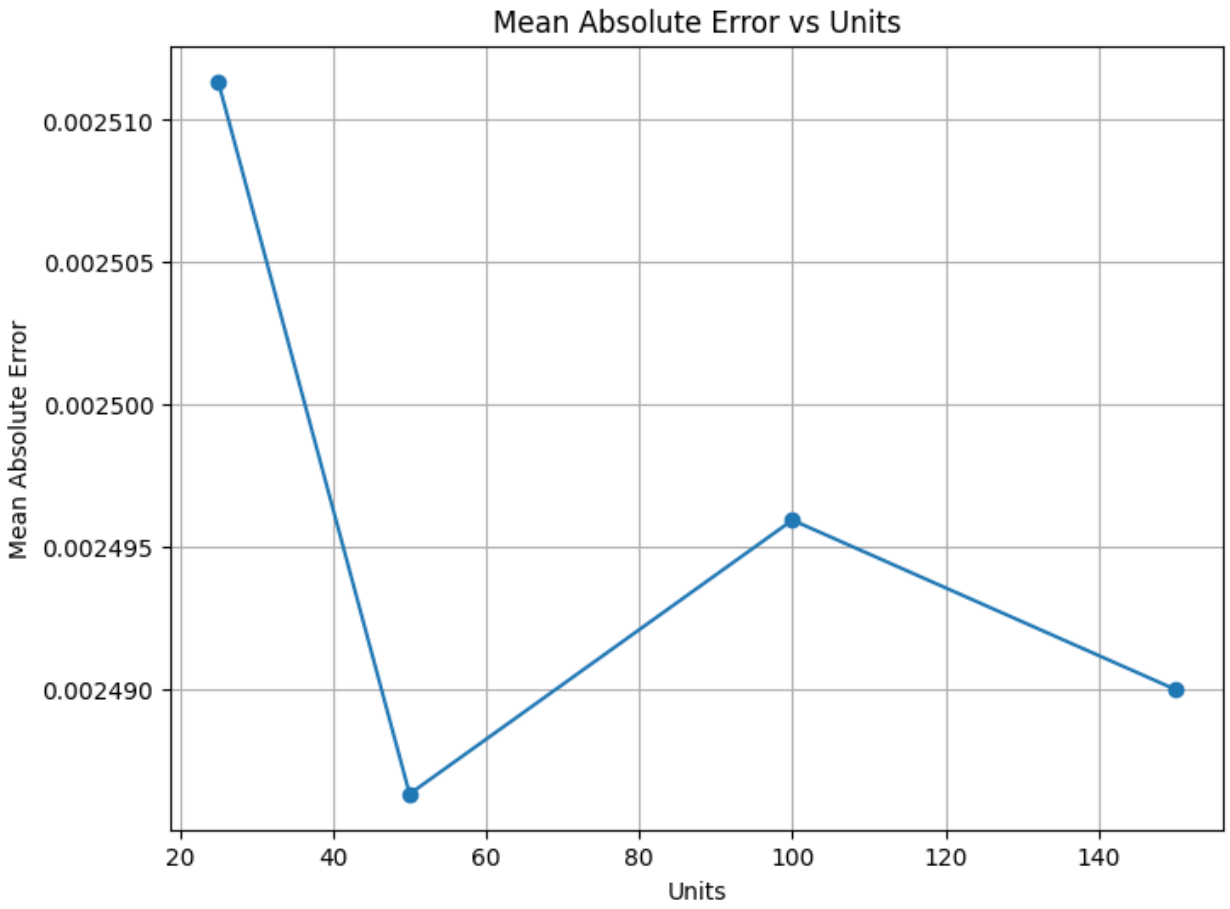
We can observe that the smallest loss is on

```
Best model found:
Number of Layers: 0
Units: 50
Activation: tanh
Validation Loss: 1.1526753041835036e-05
```

And the predicted timeseries is shown below:



Last but not least is that, I wanted to see, if different number of neurons in the best model could potentially decrease the error. After training for 25, 50, 100, 150 neurons the MAE error diagram with the units is the below.



Best model and conclusions

In our examination of Bitcoin data, the LSTM model emerged as the most effective for forecasting. We observed that the tanh activation function yielded superior results compared to the relu function, particularly suited for the intricate patterns within cryptocurrency price movements. However, due to the relatively small size of the Bitcoin dataset, neural network models faced limitations in achieving optimal performance. This underscores the importance of considering dataset characteristics when selecting modeling approaches. Overall, our findings emphasize the significance of LSTM models and tailored activation functions for accurate time series analysis in cryptocurrency markets.

Bitcoin classification

To explore LSTM's efficacy in classification and future prediction, we employed the Bitcoin dataset, forecasting the mean value for the subsequent 7 days in each window. We then classified the forecasted values into three classes:

- Class 0 for values lower than -0.33%,

- Class 1 for values between -0.33% and +0.33%,
- Class 2 for the remaining values.

In this study, LSTM neural networks were systematically explored with varying architectural configurations. Specifically, we tested LSTM architectures with 0 to 4 layers and neuron counts ranging from 50 to 1000. Additionally, we experimented with a progressive increase in neuron count by a factor of two in each layer.

Among the various LSTM architectures explored, the model with a single hidden layer consisting of 50 neurons exhibited the highest efficiency, achieving an accuracy of 62%. This finding underscores the significance of simplicity in architecture, suggesting that a more complex network may not necessarily yield superior results in this context.

The other configurations with increased or less layers or neuron counts did not exhibit comparable performance with accuracy varying from 20-40%, the simplicity of the 1-layer, 50-neuron LSTM model suggests that a parsimonious approach may be more effective for Bitcoin price prediction.

Results

Below we can see the following accuracies for different architectures of LSTM:

Model	Architecture	Layers	Units per Layer	Activation	Validation Accuracy
LSTM_0_50_relu	LSTM	0	50	ReLU	0.41747573
LSTM_1_50_relu	LSTM	1	50	ReLU	0.41747573
LSTM_2_50_relu	LSTM	2	50	ReLU	0.41747573
LSTM_3_50_relu	LSTM	3	50	ReLU	0.41747573
LSTM_0_500_relu	LSTM	0	500	ReLU	0.41747573
LSTM_0_1000_relu	LSTM	0	1000	ReLU	0.41747573
LSTM_1_500_relu	LSTM	1	500	ReLU	0.42718446
LSTM_1_1000_relu	LSTM	1	1000	ReLU	0.41747573
LSTM_2_500_relu	LSTM	2	500	ReLU	0.47572815
LSTM_2_1000_relu	LSTM	2	1000	ReLU	0.41747573
LSTM_3_50_tanh	LSTM	3	50	Tanh	0.3883495

Model	Architecture	Layers	Units per Layer	Activation	Validation Accuracy
LSTM_2_50_tanh	LSTM	2	50	Tanh	0.5533981
LSTM_1_50_tanh	LSTM	1	50	Tanh	0.592233
LSTM_1_500_tanh	LSTM	1	500	Tanh	0.2524272
LSTM_3_500_tanh	LSTM	3	500	Tanh	0.2524272
LSTM_2_500_tanh	LSTM	2	500	Tanh	0.2524272
LSTM_1_relu	LSTM	1	N/A	ReLU	0.41747573
LSTM_2_relu	LSTM	2	N/A	ReLU	0.41747573
LSTM_3_relu	LSTM	3	N/A	ReLU	0.41747573
LSTM_1_1000_tanh	LSTM	1	1000	Tanh	0.41747573
LSTM_2_1000_tanh	LSTM	2	1000	Tanh	0.2524272
LSTM_3_1000_tanh	LSTM	3	1000	Tanh	0.2524272
LSTM_3_500_relu	LSTM	3	500	ReLU	0.2524272
LSTM_4_relu	LSTM	4	N/A	ReLU	0.5145631
LSTM_5_relu	LSTM	5	N/A	ReLU	0.41747573

Conclusion:

The evaluation of various LSTM models with different configurations shows significant variation in validation accuracy. Among the models tested, the one with a single layer, 50 units, and the Tanh activation function (LSTM_1_50_tanh) achieved the highest validation accuracy of 0.592233. This indicates that the Tanh activation function is more effective in this context than ReLU, particularly for models with fewer units.

Interestingly, while increasing the number of units generally did not improve performance, a model with two layers, 500 ReLU units (LSTM_2_500_relu), reached a relatively higher validation accuracy of 0.47572815, although it was still lower than the best Tanh model. Some models with larger architectures and Tanh activation, particularly those with 500 or 1000 units, performed poorly with validation accuracies dropping to around 0.2524272. This suggests that over-parameterization and potential overfitting issues might arise with more complex models. Therefore, simpler models with appropriate activation functions tend to generalize better on the validation set in this particular setup.

Conclusion

In conclusion, Recurrent Neural Networks (RNNs) have demonstrated significant potential in the realms of time series forecasting and natural language processing (NLP). Their ability to capture temporal dependencies and process sequential data makes them uniquely suited for these tasks. In time series analysis, RNNs can model complex patterns and predict future values with remarkable accuracy, enabling advancements in fields such as finance, weather forecasting, and healthcare. In NLP, RNNs have revolutionized the understanding and generation of human language, facilitating improvements in machine translation, sentiment analysis, and conversational agents. Despite their challenges, such as vanishing gradients and computational intensity, ongoing research and innovations, like the development of Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, continue to enhance their efficacy and expand their applicability.

Looking ahead, the integration of RNNs with other emerging technologies promises to further advance their capabilities. Hybrid models combining RNNs with attention mechanisms or transformers are already yielding superior results in both time series and NLP tasks. Additionally, the application of RNNs in real-time data analysis and adaptive learning systems highlights their growing importance in an increasingly data-driven world. As research progresses, the continual refinement of RNN architectures and training methods will undoubtedly unlock new possibilities, making RNNs an indispensable tool in the toolkit of data scientists and engineers tackling the complexities of sequential data across various domains.