UNIVERSITY OF HERTFORDSHIRE

Department of Computer Science

7COM1075 - Data Science and Analytics Masters Project

Final Project Report

27/08/2021

Cryptocurrency Price Prediction using RNNs

MSc Final Project Declaration

This report is submitted in partial fulfillment of the requirement for the degree of Master of Science in 7COM1075 Data Science and Analytics Masters Project at the University of Hertfordshire (UH).
It is my own work except where indicated in the report.
I did not use human participants in my MSc Project.

I hereby give permission for the report to be made available on the university website, provided the source is acknowledged.

Abstract

Investing in cryptocurrency, particularly Bitcoin, has been popular for many years since it is one of the most popular and decentralized cryptocurrencies. Bitcoin has more than 5.8 million active users and around 111 exchanges across the globe which has made them an alternate form of investment and gains profit in a relatively short investment time compared to other means. Cryptocurrencies can be used for both payments worldwide as well as for investment. Due to its recent price boom and breakdown, Bitcoin has attracted much attention from the media, ordinary people, and big investors. It has also caught the attention of many researchers that wanted to inspect the various factors that influence its prices and the trends that drive its fluctuations, explicitly using machine learning algorithms.

This research aims to identify the best efficient and accurate model for Bitcoin price prediction using deep learning techniques such as LSTM, GRU in sequential and parallel multilayer architectures. In this project, the Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) models are used in one layer, two layers, and three layers with standard, Bidirectional architecture with sequential and parallel mode to predict the accuracy of models for bitcoin prices in USD. The experiment results indicate that two layers of Bidirectional GRU based prediction models outperformed the other models. In single-layer sequential architecture, GRU models performed the best. However, the hyperparameter tuning process can be applied to the current models to achieve improved results.

The results derived from these experiments might help people interested in investment in digital currencies but are unsure about the process and future risks. Although the results will predict outcomes based on supplied parameters and may not be accurate, it will be a close estimate rather than complete uncertainty. This proposed approach can also be applied to some other popular cryptocurrencies like Ethereum (ETH), Litecoin (LTC), Binance Coin (BNB), and Monero (XMR). The practical application of this project can be the automated trading systems/bots.

Acknowledgement

I express my deepest gratitude to my supervisor Mr Harpreet Singh, who has supported me in all project phases. Mr Harpreet's guidelines helped me achieve the goals on time, and his immediate feedback provided me with an opportunity to rectify the issues with the project.

Acronyms

Adam Adaptive Moment Estimation.
ANN Artificial Neural Network.
ARIMA Autoregressive Integrated Moving Average.
CNN Convolutional Neural Network.
CRNN Convolutional Recurrent Neural Network.
DLNN Deep Learning Neural Network.
GRNN General Regression Neural Network.
GRU Gated recurrent unit.
LSTM Long short-term memory.
MAPE Mean Absolute Percentage Error.
MLP Multilayer Perceptron.
MSE Mean Squared Error.
ResNet Residual Neural Network
RMSE Root Mean Squared Error.
RNN Recurrent Neural Network.
SVM Support Vector Machines.
TCN Temporal Convolutional Networks.
TS Time series.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 Background

The digitalization era had many technological improvements in different areas, which benefited many consumers and businesses. Cryptocurrencies are one such example that has gained popularity due to the digital transformation in the financial sector over time. Bitcoin is a highly safe and reliable global cryptocurrency in which peer-to-peer payment transactions are validated by nodes in the network and registered on a publicly accessible ledger.

Bitcoin is the first decentralized digital currency as a central bank or government does not manage it. More than 40 countries (Germany, Croatia, Switzerland, Canada) accept Bitcoin as cryptocurrency [3]. It was first introduced in 2009, but it became popular in 2017. Bitcoin can be used for both payments and investments all around the world. Before investing in bitcoin, it is always advised that investors conduct a precise risk analysis to avoid any losses. Bitcoins can be acquired through mining or exchanging goods, services, or any other available modes of payment. The historical data generated for cryptocurrencies transactions are in the form of time-series data. Time series data is a collection of the records which are recorded over time. If we plot the collected data points on a graph, one of its axes will be time. Time series are used in Natural language processing(NLP), pattern recognition, weather forecasting, stock prediction, earthquake prediction, and almost any other applied science and engineering area involving temporal measurements.

The process of releasing bitcoin into the public is known as bitcoin mining. It is the method of adding and verifying transaction records across the Bitcoin network. In general, mining involves solving computationally challenging riddles to discover a new block, which is then added to the blockchain. The reward for miners is bitcoin, which is half every 210,000 blocks. The smallest unit of bitcoin is called a Satoshi, and it is divisible to eight decimal places (100 millionths of a bitcoin). Bitcoin could someday be made divisible to even more decimal places if required and if the involved miners accept the change.

Although several kinds of research have been conducted using machine learning and deep learning techniques to predict bitcoin prices, this project focuses on investigating the effect of multilayer sequential and parallel architecture implementation to predict the cryptocurrency price accurately. The experiments are conducted for multiple architectures, and the results are published in section 6. The practical application of this project is automated trading systems/bots. Also, this proposed system will help people interested in digital currencies investment but are unsure about the process and future risks.

1.2 Problem Statement

With the recent popularity of cryptocurrencies among people as a form of investment, it is crucial to predict the prices of such commodities for a shorter time frame. The data

used for this is dependent on time and is very similar to the stock market data, which has been in existence for many years now. Therefore, we can use the stock market approaches for data transformation with multilayer architectures in a sequential and parallel manner.

From the literature review, the above techniques were not widely used in the existing papers published on the cryptocurrency price prediction. Hence, in this report, the above-discussed approaches will be used to analyse and find the answer to the research questions.

1.3 Research Questions

i) Do the GRU models provide better prediction results in multilayer architecture compared to LSTM models?

ii) Can multilayer deep neural network architecture be used to enhance the model efficiency?

iii) How effective GRU in Bidirectional mode compared to standard GRUs in multilayer architecture?

iv) Does the number of units used in the hidden layers improve model performance?

1.4 Research aim

The aim of this project is to build and compare efficiency of the deep learning models with single and multilayer architectures for predicting the cryptocurrency prices.

1.5 Objectives

The objectives for this project are as follows:

- Conducting research on existing technologies
- Analyze the price data using the deep learning models
- Predicting the results obtained from project Implementation
- Evaluate the model based on the test dataset.
- Compare the evaluation results and identify the best architecture

1.6 Tools and Techniques Used

Table 1: Technical specification

Programming Language	Python
Libraries	NumPy, pandas, matplotlib,seaborn, scikit-learn, Keras (deep learning)
IDE	Jupyter Notebook, Google colab

2. Literature Review

Machine learning models like recurrent neural networks (RNN) and long short-term memory (LSTM) have been shown to perform better than traditional time series models in cryptocurrency price prediction [13]. The comparative analysis study of RNNs shows that the GRU model results were better than the LSTM model [4]. The Bidirectional LSTM was more reliable than the standard unidirectional LSTM [14]. The study suggests that the GRU and LSTM networks in standard and Bidirectional mode will be more reliable to work with the time-series data analysis.

The cryptocurrency price prediction is similar to the stock market prediction with similar features like High, Low, Open and Close. So, the data can be transformed to a new training set with the number of features equal to the number of days considered in time_steps, and the final prediction variable will have the value of time_steps+1 [5], [15]. Research on existing literature recommends using the RNN techniques like LSTM and GRU for the time-series analysis but shows a research gap on using different architecture models configurations like parallel mode processing using Keras functional APIs. Therefore, this project focuses on the effects of using different architecture models with the transformed dataset in a single, stacked, and parallel manner using standard and bidirectional LSTM and GRU. For a regression problem, the performance of prediction models is often measured in terms of the root-mean-square error (RMSE) or the mean absolute percentage error (MAPE) [1]. Therefore, the MAPE values are used to compare models performance for various architectures in this project.

Table 2: Lists the details of the background research conducted in project

PAPER	YEAR	DATASET	MODELLING	EVALUATION
A Comparative Study of Bitcoin Price Prediction Using Deep Learning	2019	https://bitcoincharts.com/charts/bitstampUSD#rg60ztgSzm1g10zm2g25zv	DNN, LSTM, CNN, ResNet, CRNN, Ensemble, SVM	MAPE,RMSE
Bitcoin Price Prediction Using Machine Learning	2019	https://www.cryptodatadownload.com/data/bittrex/	Logistic Regression , SVM , ARIMA , RNN	Accuracy
Bitcoin Price Alert and Prediction System using various Models	2021	https://data.cryptocompare.com/	Linear Regression, Theil-Sen Regressor, Huber Regressor, LSTM and GRU	Linear Regression, Theil-Sen Regressor and Huber Regressor - Accuracy LSTM and GRU - MSE, R2
Bitcoin price prediction using Deep Learning Algorithm	2019		RNN, LSTM,GRU and ARIMA	Accuracy,MSE, R2

Applying Deep Learning to Better Predict Cryptocurrency Trends	2018		Stochastic Gradient Descent, RMSProp and Adam optimizer	Accuracy
Short Term Stock Price Prediction Using Deep Learning	2017	https://www.nyse.com/index	LSTM, MLP	RMSE
Cryptocurrency forecasting with deep learning chaotic neural networks	2018		DLNN, GRNN	RMSE
Project Based Learning: Predicting Bitcoin Prices using Deep Learning	2019	https://poloniex.com	Multilayers CNN and RNN	Performance comparison of layers
Ether Price Prediction Using Advanced Deep Learning Models	2021	https://www.cryptodatadownload.com/	LSTM, GRU, and TCN	Accuracy
A Deep Learning-based Cryptocurrency Price Prediction Scheme for Financial Institutions	2020	https://www.investing.com/	LSTM, GRU and LSTM-based hybrid model	MSE, RMSE, MAE and MAPE
Machine Learning Models Comparison for Bitcoin Price Prediction	2018	https://www.okcoin.com/en	Theil-SenRegression, HuberRegression, LSTM and GRU	MSE and R-Square (R2)
Deep Learning Approach to Determine the Impact of Socio Economic Factors on Bitcoin Price Prediction	2019	https://poloniex.com/	CNN, LSTM and GRU	MSE
A Gated Recurrent Unit Approach to Bitcoin Price Prediction	2020	https://bitcoincharts.com/	Neural Network, LSTM, GRU, GRU-Dropout and GRU-Dropout-GRU	RMSE
Evaluation of bidirectional LSTM for short-and long-term stock market prediction	2018		MLP-ANN, LSTM, SLSTM and BLSTM	RMSE, MAE, and R2

3. Social, Ethical and Legal Issues

The project analyses the past prices of the cryptocurrencies and then predicts the next set of records using Deep Learning. Therefore, the prices predicted using this project will not be the same as market prices since we have many prediction parameters. Multiple external factors also affect the prices, which are not considered and have a high impact on the surge and fall of the prices. Statements passed by the big investors on the news or social media platforms are examples of external factors. Apart from this, the economic situations also impact the prices where the world's economy drops due to attacks, cybercrime or pandemic situations where the overall economy suffers enormous losses.

While bitcoin is widely accepted, some countries are suspicious of it due to its volatility, decentralised nature, perceived threat to existing financial markets, and connections to illegal activities such as drug trafficking and money laundering. Due to its decentralised nature, the third parties are unable to add transactional charges to its usage. Some countries like Russia, China, and Vietnam have explicitly restricted the use of digital currency. Because bitcoin is unregulated, if you do not have the keys to access a relative's digital wallet, you will not be able to access their cash if they pass away. Bitcoins price varies with time, just like any other investment causing market risk. Because Bitcoin exchanges are digital, they are subject to hacking, system failures, and viruses.

The mining of cryptocurrencies consumes more resources like electricity which cause emissions like carbon dioxide in the environment. Apart from this, the cryptocurrency traders remain entirely anonymous, encouraging money laundering and tax evasion problems. Countries had started to legalise cryptocurrencies as a form of payment. El Salvador is the first country to legalise bitcoin as a form of payment.

General Data Protection Regulations(GDPR): The project adheres to the seven principles of the GDPR law of the general data protection regime in the UK. The data source used for analysis purposes, and the software/IDE/Programming Language has the licence which grants limited, personal, non-exclusive, non-sub-licensable and non-transferable license to use the Content.

4. Methodology

The dataset used in this report is taken from the below references and is available on the website https://coinmarketcap.com/currencies/bitcoin/historical-data/. The data used in the analysis is in the CSV file format. The dataset has a license that grants a limited, personal, non-exclusive, non-sublicensable and non-transferable license to use the Content and to use the Service. Bitcoin market (USD) time-series data has the following features and is generally represented in candle chart form.

Dataset Features :
Date: The reported date of the cryptocurrency pricing.
Open: The cryptocurrency's price at the start of a business day.
High: The cryptocurrency's highest price at the given date.
Low: The cryptocurrency's lowest price at the given date.
Close: The cryptocurrency's price at the end of a business day.
Volume: The total amount of coins traded on that day.
Market Cap: The market value of a publicly traded cryptocurrency unit.

4.1 Data preprocessing

Normalization: To avoid the differences between the ranges of the values, the MinMaxScaler from the sklearn library is used, which updates the values of numeric features in the dataset to a common scale, i.e. in our case, between [0-1].

$$Xnormalized = \frac{X - Xmin}{Xmax - Xmin}$$

Data partitioning
The dataset is divided into two parts, i.e. train set and the test set. 80% of the data is used to train the model using deep learning, whereas the remaining 20% is used to test the model accuracy.

Feature selection
The feature **'Close'** is used to create a labeled data for future prediction. The feature **'Close'** determines how well or poorly a cryptocurrency performed, which is a big deal for investors and financial institutions and other stakeholders.

Data Transformation
To predict the values with accuracy, the dataset is transformed by using the historical 'close' values in the dataset and is used in generating new records for the training set. A 30-day prediction transform dataset will have values (V0-V29) in Xtrain, whereas the prediction Ytrain will be V30. This process of transformation is applied to the entire dataset.

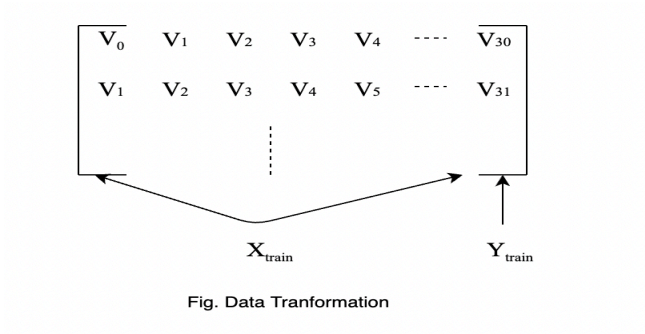

Figure 1: Data Transformation

Libraries

The python libraries that are used in this project for importing, transforming, implementation and analysis are as follows:

Pandas
This high-level library provides various methods to perform different operations on data frames [3].

Numpy
This library provides various methods to perform different operations on large datasets, n-dimensional arrays, and matrices [3].

SciKit Learn
This library provides various algorithms for predictive data analysis [3].

Matplotlib
A library that provides various methods to plot different types of graphs [3].

Seaborn
Seaborn is a matplotlib based Python data visualization package. It has a high-level interface for creating visually appealing and instructive statistics visuals [23].

Keras
This library provides functionalities for developing different deep learning models, and if required, we can also use Tensorflow, CNTK, and Theano libraries with Keras for developing a high-level neural network program [3].

4.2 Recurrent Neural Network (RNN)

A Recurrent Neural Network (RNN) is a network that contains at least one feedback connection so that the activations can flow round in a loop [17]. This helps the neural networks to perform processing and learn sequences like the output from the previous step (x□-1) is fed as an input to the next step(x□). RNN's are used in speech recognition, Language translation, stock prediction and image recognition to describe the content in the corresponding pictures. They perform well in processing the time

series and the sequential data by breaking them into smaller chunks and implementing sequential memory. Audio and text are examples of sequential data. It can be thought of as a feed-forward neural network with a loop to pass previous information.

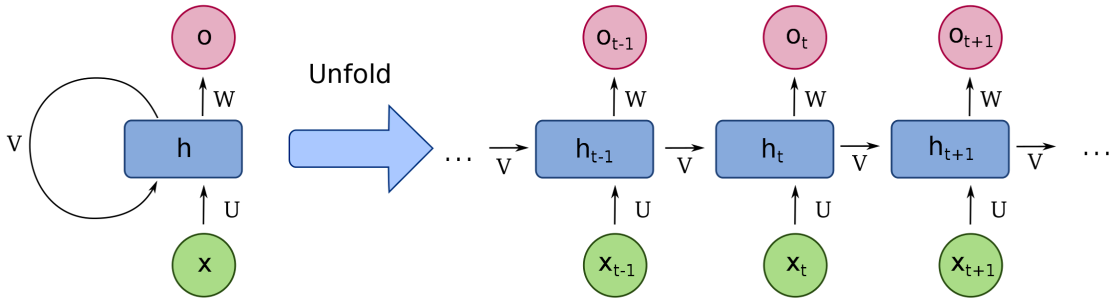

Figure 2: Recurrent Neural Network

The training of the neural network is done in the following steps:
1. The network makes a forward pass and makes the predictions.
2. It then uses the loss function to compare the predicted values against the actual values. The output of a loss function is an error value which is an estimate of network performance.
3. The error value will then be used in gradient calculation during the backpropagation step for each node in the network. The network uses the gradient values for internal weight adjustment while the network learns—a higher value of gradient results in higher weights and vice versa.

Vanishing gradient problem: Gradients are values used to update a neural network's weights. During the backpropagation, every node computes their gradients w.r.t. the gradient values in the previous layers. As a result, the network fails to learn during backpropagation. Therefore, smaller gradient values in previous layers will result in much smaller values for subsequent layers in the network, which causes the exponential shrinking of the gradient. The equation calculates the process of updating the weights in neural networks and is given by the equation:

new weight = *weight - learning rate * gradient*

The RNNs offer an advantage of training faster and utilising fewer computational resources due to fewer TensorFlow operations to compute. The problem of vanishing gradients in RNNs makes it challenging to learn the large data sequences. The LSTM and GRU networks perform better for longer sequences and long-term dependencies, which are discussed below.

Activation Functions:
The activation function of a neuron in a deep neural network determines the output of a neuron for a set of input data. It mimics our human brain, where the neurons are fired or activated by different stimuli.
The Activation Functions can be classified into two categories.

1. Linear Activation Function
Linear Activation Function is also known as Identity Activation Function, is a line or has linear output. The functions output will not be limited to a specific range. It makes no difference to the complexity or other data characteristics that are input to the neural networks.

2. Nonlinear Activation Functions
Nonlinear functions tackle the limitations of a linear activation function. They have a derivative function connected to the inputs, allowing backpropagation. They make it possible to form a deep neural network by stacking numerous layers of neurons and are the most commonly used activation functions in the neural networks. The two most important terms for the nonlinear function are Differential or Derivative and the Monotonic function.
Differential or Derivative is the change in the y-axis with the x-axis, also known as the slope. A monotonic function is either completely non-increasing or completely non-decreasing.

4.3 Long Short Term Memory (LSTM)

The LSTM network model has working characteristics that are similar to recurrent neural networks, which is a feedback neural network. It analyzes the data and transfers information as it progresses forward, but the operations within the cells of the LSTM differ, which helps the LSTM retain useful information or discard ineffective information. The cell state can carry relevant data throughout the sequence's processing. The gates are various neural networks that determine which inputs about the cell state are allowed. Through training, the gates can learn what content is essential to keep or discard. The Gates contains the sigmoid activation functions, which is similar to the Tanh activation, but instead of converting numbers ranging from -1 to 1, it converts the values ranging from 0 to 1. As multiplying any integer value by 0 equals 0, values vanish or are "forgotten" this is useful for updating or forgetting data.

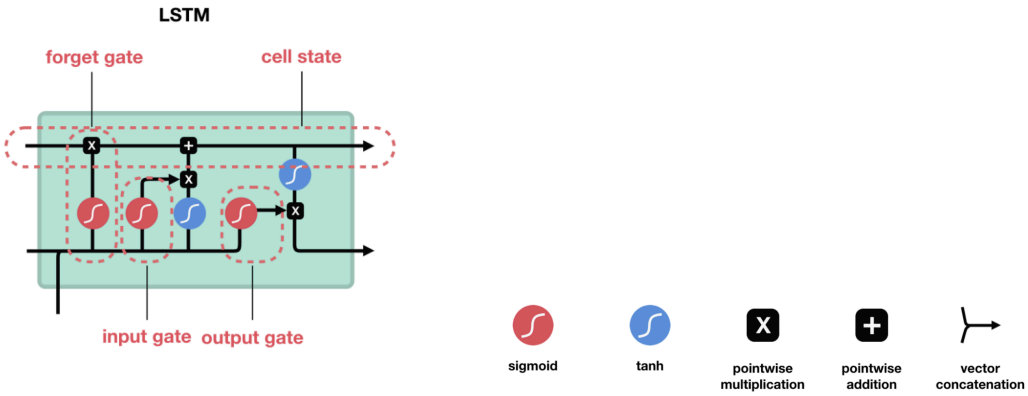

Figure 3: LSTM Neural Network

The LSTM network solves the vanishing gradient problem by remembering the information over long periods. The network's activations act as its short-term memory, whereas its weights act as its long-term memory.
It works in a three-step process :
1. **Forget gate**
The forget gate determines whether data should be rejected or retained. The sigmoid function passes data from the previously hidden layer as well as the current input data. The results are between 0 and 1. The values closer to 0 are forgotten, whereas those closer to 1 remain in the memory.

2. **Input Gate**
The input gate is responsible for updating the cell state.

3. **Output Gate**
The output gate determines the next hidden state in the network. The hidden state contains data from previous inputs, and the predictions are also performed using the hidden state.

The nonlinear activation functions used in the LSTM and GRU's are as follows:
1. **Sigmoid or Logistic Activation Function**
The Sigmoid Function curve has an S-shape graph. The output range of this function is between 0 and 1, which resembles the output range of probability. The logistic sigmoid function has the possibility of causing a neural network to become stuck during training. The softmax function is a multiclass classification function that is more generalised than the logistic activation function.

2. **Tanh** (hyperbolic tangent Activation Function) Tanh function is also s-shaped and is similar to the logistic sigmoid but a bit better as it has a range of values between -1 to 1.

4.4 Gated Recurrent Units (GRU)

The GRU is a newer type of recurrent neural network that can be considered as a variant of LSTM. The GRU does not have the cell state but instead use the hidden state to transfer data. There are only two gates in GRU which are a reset gate and an update gate. As the GRUs have fewer tensor computations, they are somewhat faster to train than LSTMs. They use an update gate and a reset gate to tackle the vanishing gradient problem.

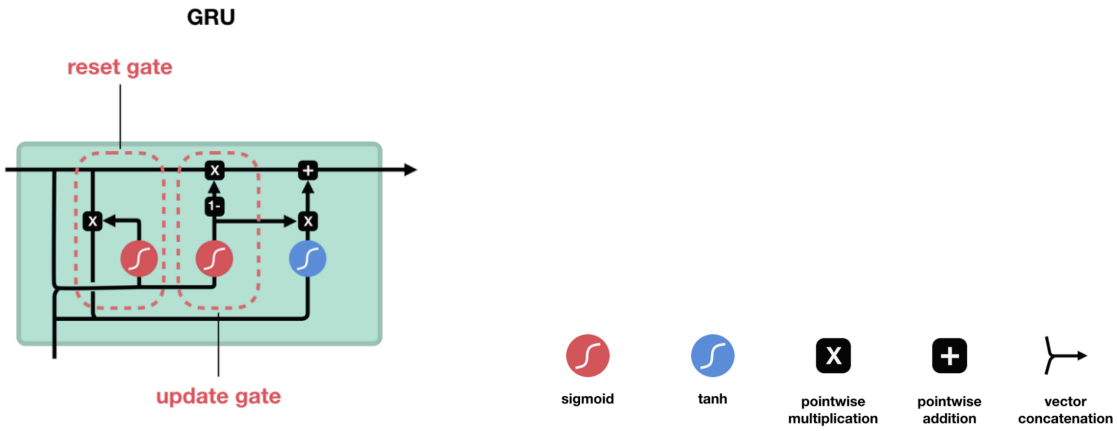

Figure 4: GRU Neural Network

Update Gate

The update gate works similar to an LSTM forget and input gate. It determines what data should be discarded and what should be retained.

Reset Gate

The second gate is the reset gate, which determines how much data from the previous information should be removed.

The significant differences between the GRU and the LSTM networks are as follows:
1. The LSTM network has three gates, i.e. Input gate, Forget gate, and Output gate, while the GRU has only two gates, i.e. Reset gate and an Update gate.
2. In LSTMs, the Long term memory is stored in the cell state, and the short term memory is stored in the hidden state. However, the GRU has only one state, i.e. Hidden state (Ht).

4.5 Bidirectional RNNs

Bidirectional recurrent neural networks (RNN) are just the combination of two separate RNN networks. The input sequence is supplied in regular time order to one network, whereas it is provided in reverse time order for another network. At each step, the outputs of the two networks are normally concatenated, although there are other choices, such as summation. This approach allows the networks to have both backward and forward information about the sequences at each time step.

4.6 Keras Functional API

The Keras functional API allows us to design more flexible models than the tf.keras.Sequential API. Models with non-linear topologies share layers, and the functional API supports even multiple inputs and outputs. A deep learning model is typically a directed acyclic graph (DAG) of layers. As a result, the functional API allows us to create multilayer layer parallel graphs.

4.7 Performance Evaluation:

In order to evaluate the model performance after fitting, the error is obtained by comparing model predictions values to the actual data.

Error = $\widehat{yi}$ - y_i

where $\widehat{yi}$ is the prediction value and y_i is the actual value in the above equation.

Mean Absolute Error (MAE)

Mean Absolute Error is obtained by calculating the model predictions and the actual values. Smaller is the value of MAE, which signifies better results.

$$MAE = \frac{1}{n}\sum_{i=1}^{n} |\widehat{yi} - yi|$$

Root Mean Squared Error (RMSE)

RMSE represents the standard deviation of the residuals, i.e. variance among the predictions and the training data of the model. When compared to mean square error (MSE), RMSE is easier to interpret because its units match the output units. The RMSE gives a rough indication of how distributed the data points are. The lower the value of RMSE, the better the results.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (\widehat{yi} - yi)^2}$$

Mean Absolute Percentage Error (MAPE)

Mean Absolute Percentage Error (MAPE) depicts the percentage error and provides better insight than other performance evaluations, making the results simple to understand. Because the division operation is used in the equation, MAPE may have certain limits if the data point value is 0. However, this is not the typical case because we will always have a positive amount of data points to analyse. The lower the MAPE value, the better the results will be.

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n} |\widehat{yi} - yi) / yi|$$

5. Implementation

This section describes the approach used to perform the practical implementation of the project. It is divided into subsections about accessing the dataset, operations performed on the dataset to achieve desired results, building the models, performing prediction and evaluation. The implementation of this project is in a python programming language using Jupyter notebook / Google colab environment for code execution. The libraries Pandas, Numpy, SciKit Learn, Matplotlib, Seaborn and Keras, are used in this project.

5.1 Data Collection

In this step, the data is obtained by importing with function read_csv function in pandas library, which takes the file path URL as the input parameter in the form of a string object. The dataset used in this project is stored in the personal GitHub repository (https://github.com/247pankaj/crypto-dataset).

IMPORTING DATA

```
[3]  # Fetching the csv dataset from GitHub repository
     crypto_dataframe = pd.read_csv("https://raw.githubusercontent.com/247pankaj/crypto-dataset/main/Bitcoin_price.csv")
```

The available features and their respective datatypes in our dataset are displayed where it contains seven features, i.e. Date, Open, High, Low, Close, Volume and MarketCap. Apart from the date, which is an object, all the other features contain float values.

```
[5]  # Feature DataTypes
     crypto_dataframe.dtypes
```

```
Date          object
Open         float64
High         float64
Low          float64
Close        float64
Volume       float64
MarketCap    float64
dtype: object
```

5.2 Data Preprocessing

Before we can begin building our model, several necessary steps need to be performed on the imported dataset. The first step in preprocessing is converting the Date object into DataTime, as this field will then be used as an index of our dataset.

```
[6]  # Converting Date Object into Datetime
     crypto_dataframe['Date'] = crypto_dataframe['Date'].astype('datetime64[ns]')
```

The next step is to move our result variable one time step-up (-1). The goal is to organise the data so that the prediction variables can be used to forecast the next day's closing price. The empty field is replaced with 0 then removed that value from the dataset because it might be considered in the prediction.

Another approach would be to reduce all of the predictor values by one i.e. +1. This would have resulted in the insertion of a 0 at the start of the dataset, and we can use the same process of removing that record as its value would not exist and can not be used in prediction.

```
[8]  crypto_dataframe[next_close] = crypto_dataframe['Close'].shift(-1,fill_value=0)
```

```
[9]  crypto_dataframe.drop(crypto_dataframe.tail(1).index,inplace=True)
```

Splitting dataset

The dataset can now be divided into a training and a testing set. The dataset is divided into 80 and 20 split for train and test data, where 80% of records are in the training set, which will be used to train the model, and 20% of data is in the test set, which will then be used in model evaluation.

```
[13] # Train-Test Split(80%-20%)
     training_size = int(len(crypto_dataframe) * 0.80)
     training_dataset, testing_dataset = crypto_dataframe.iloc[:training_size],crypto_dataframe.iloc[training_size:]
```

The train and test datasets obtained are now separated into the predictor variables (X) and the outcome variable (y). This is achieved by dropping the outcome feature “next_close” from the dataset in X_train and X_test arrays and then assigning it to respective y_train and y_test arrays for the dataset.

```
[15] # Splitting training_dataset into Predictors and outcomes
     X_train = training_dataset.drop(next_close, axis = 1)
     y_train = training_dataset.loc[:,[next_close]]

     # Splitting testing_dataset into Predictors and outcomes
     X_test = testing_dataset.drop(next_close, axis = 1)
     y_test = testing_dataset.loc[:,[next_close]]
```

Scaling Dataset

The feature values in the dataset mostly contain highly varying magnitudes, units, and ranges, resulting in the neglect of specific values. The best practice for this problem in machine learning is to apply scaling between specified ranges to the numeric variables in the dataset. The MinMaxScaler from the sklearn library is used to achieve this. Scaling is done to fit the values in the range from -1 to 1. The below steps are performed on the data:

a. Fitting the scaler to the train set with fit() function
b. Applying the scale to train set by the transform() function
c. Applying the scale to the test set

```
min_max_scaler = MinMaxScaler(feature_range = (-1,1))

# Fitting MinMaxScaler to Trainset(X_train,y_train)
predictor_scaler = min_max_scaler.fit(X_train)
outcome_scaler = min_max_scaler.fit(y_train)

# Scaler in trainset
y_norm_train = outcome_scaler.transform(y_train)
x_norm_train = predictor_scaler.transform(X_train)

# Scaler in testset
y_norm_test = outcome_scaler.transform(y_test)
x_norm_test = predictor_scaler.transform(X_test)
```

Data Transformation

In the data transformation step, a python function is defined, i.e. transform_dataset, which takes the predictor variables array and outcome variable array as input parameters. The output of this function is the transformed array of X_array and y_array, which are returned after the execution of the function. X_transform and y_transform are the two empty arrays created before processing to collect the transformed data. A for loop is used to iterate the elements inside the input X_array until its (length - time_steps) as the final records equal to the time steps will be appended to the X_transform array. X_transform is the array of records with feature count equal to the time steps, whereas the y_transform is just one feature i.e. outcome value for the prediction . In our case, a global variable defined for the project as TIME_STEPS used in processing.

```
[17] def transform_dataset (X_array, y_array, time_steps = 1):
         X_transform, y_transform = [], []
         for i in range(len(X_array) - time_steps):
             output = X_array[i:i+time_steps, :]
             X_transform.append(output)
             y_transform.append(y_array[i+time_steps])

         return np.array(X_transform), np.array(y_transform)


     X_test, y_test = transform_dataset(x_norm_test, y_norm_test, TIME_STEPS)
     X_train, y_train = transform_dataset(x_norm_train, y_norm_train, TIME_STEPS)

     print('X_train: ', X_train)
```

5.3 Building the DL Model

Following the data preprocessing, the subsequent step is to build the model, train it using the train set, and then test its performance on the test set. As per our experiment, which is conducted in the next section, we will be creating a separate model for each experiment and check its performance by predicting the Bitcoin prices. The models are built with single and multilayer architecture, with each layer containing 64 units and a dense layer with one unit. For these models, adam is used as the optimizing function, and loss function is mean square error(mse). The create model function takes the number of units(neurons) and the model type, i.e. mt, as input parameters where the model type can be LSTM or GRU. The Bidirectional networks are used in sequential and parallel modes for multilayers. In parallel architecture, the inputs are concatenated using keras.layers.concatenate method.

```
[18] # BiRNN Model using Functional API of Keras (Parallel)

    def build_model_birnn_func(units, mt):
        input1 = Input(shape=(X_train.shape[1], X_train.shape[2]))
        hidden1 = Bidirectional(mt(units = units) )(input1)

        hidden2 = Bidirectional(mt(units = units))(input1)

        hidden3 = keras.layers.concatenate([hidden1, hidden2]) # Concat
        output = Dense(1)(hidden3)

        parallelModel = Model(inputs=input1, outputs=output)

        parallelModel.compile(optimizer='adam',loss='mse')
        return parallelModel

    # Building the Sequential Bidirectional model
    def build_model_birnn(units, mt):
        sequentialModel = Sequential()
        sequentialModel.add(Bidirectional(mt(units), input_shape=(X_train.shape[1], X_train.shape[2])))
        sequentialModel.add(Dense(1))

        sequentialModel.compile(optimizer='adam',loss='mse')
        return sequentialModel
```

Model Fitting

The model will now be fitted to check the bitcoin price prediction. The models will run for the 100 epochs. EarlyStopping is used with the patience value as 10, it prohibits the model from running if the validation error has not improved in 10 epochs. It will reduce the execution time, particularly for the huge size datasets. As we are dealing with the analysis of time-series data where the order of records is of utmost importance, therefore, the shuffled value in the model should always be set to False in such scenarios. The CSV logger is also used for logging the output details.

```
[24] # Existing models Fit function
     def fit_mType(mType):
         early_stopping = keras.callbacks.EarlyStopping(monitor = 'val_loss',
                                                        patience = 10)

         # shuffle = False because the order of the data matters
         modelHistory = mType.fit(X_train, y_train, epochs = 10, validation_split = 0.2,
                         batch_size = 32, shuffle = False, callbacks = [csv_logger,early_stopping])
         return modelHistory
```

5.4 Model Prediction

The inverse transform should be applied to reverse the scaling effect on the dataset in order to obtain their actual values. Otherwise, the prediction values will remain in the scaled range, i.e. -1 to 1.

```
[25] y_test = min_max_scaler.inverse_transform(y_test)
     y_train = min_max_scaler.inverse_transform(y_train)
```

The prediction function is defined, which takes a model to predict as the input and performs the prediction on the test set.

```
[26] # Prediction
     def perform_prediction(mType):
         predict_values = mType.predict(X_test)
         predict_values = min_max_scaler.inverse_transform(predict_values)
         return predict_values

     predicted_parallel_biGru = perform_prediction(func_biGruModel)
     predicted_parallel_biLstm = perform_prediction(func_biLstmModel)
```

5.5 Model Evaluation

To evaluate the model's performance, RMSE, MAE and MAPE are calculated for each model, and the values are then compared with the other models used in experimentation.

```
[28] def prediction_perfomance(predicted_values, actual_values, model_type):
         errors = predicted_values - actual_values
         mse = np.square(errors).mean()

         rmse = np.sqrt(mse)
         mae = np.abs(errors).mean()
         mape = np.mean(np.abs((actual_values - predicted_values) / actual_values)) * 100
```

6. Experimentation and Results

The experiments with a single hidden layer, two hidden layers and three hidden layers architectures were performed with Bidirectional and standard LSTMs and GRUs units in a sequential and parallel mode. These experiments were performed on the dataset by dividing it into the training set and testing set. The performances of the experiments are evaluated on the test set by determining their Mean Absolute Percent Error(MAPE) values. Based on the type of RNNs used and their corresponding parameter values, the experiments are further divided into 20 sub experiments. The results obtained are then compared among the different layer architectures to determine the architecture with the best performance.
For all the different hidden layer architectures, the below model configurations are used:
1. Bidirectional GRU in parallel mode using Keras functional APIs
2. Bidirectional LSTM in parallel mode using Keras functional APIs
3. Bidirectional GRU in sequential mode
4. Bidirectional LSTM in sequential mode
5. GRU in sequential mode
6. LSTM in sequential mode

Experiment 1: Single layer Architecture

6.1.1: Single layer Architecture (Bidirectional GRU: Parallel model)

Objective: This experiment intends to study the effect of using Bidirectional GRU in parallel mode with Keras functional APIs in the Single-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only one hidden layer is placed in parallel with another hidden layer, where both the parallel bidirectional GRUs have 64 units in their hidden layer. The output is then concatenated passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam” optimizer are used for model compilation.

The architectural diagram with the inputs as mentioned above is represented in the figure below:

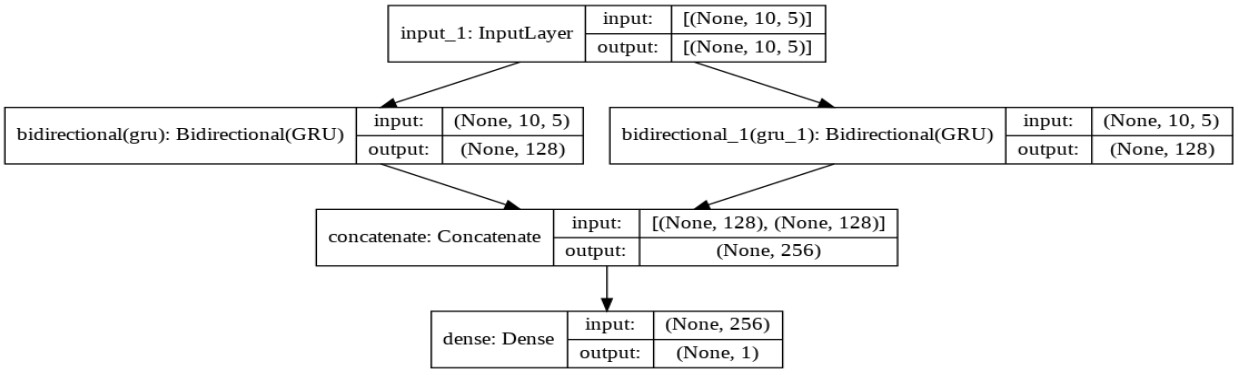

Figure 5: Single layer Architecture - Bidirectional GRU in Parallel mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 749.74
Root Mean Square Error: 1576.74
Mean Absolute Percentage Error: 5.97

The MAPE value signifies that the model has an error rate of 5.97%.

6.1.2: Single layer Architecture (Bidirectional LSTM: Parallel model)

Objective: This experiment aims to study the effect of using Bidirectional LSTM in parallel mode with Keras functional APIs in the Single-layer architecture and to compare its performance with other architectures.

Implementation: In this experiment, only one hidden layer is placed in parallel with another hidden layer, where both the parallel bidirectional LSTMs have 64 units in their hidden layer. The output is then concatenated passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam” optimizer are used for model compilation.

The architectural diagram with the inputs as mentioned above is represented in the figure below:

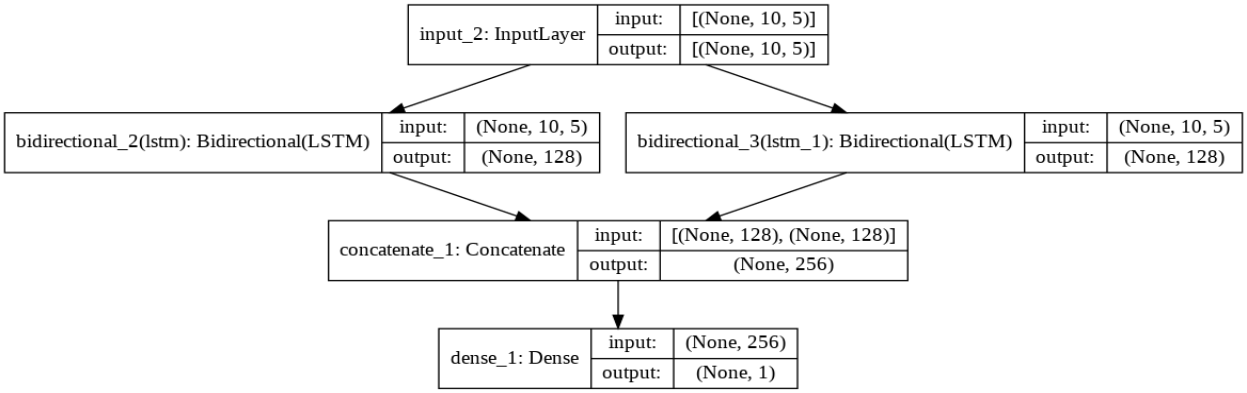

Figure 6: Single layer Architecture - Bidirectional LSTM in Parallel mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1235.28
Root Mean Square Error: 2700.12
Mean Percentage Error: 8.59

The MAPE value signifies that the model has an error rate of 8.59%.

6.1.3: Single layer Architecture (Bidirectional GRU: Sequential model)

Objective: This experiment aims to study the effect of using Bidirectional GRU in sequential mode with the Single-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only one hidden layer is placed sequentially, where the bidirectional GRU in the hidden layer has 64 units. The output is then passed to the dense layer consisting of a single neuron. The "mse" loss function and "adam" optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

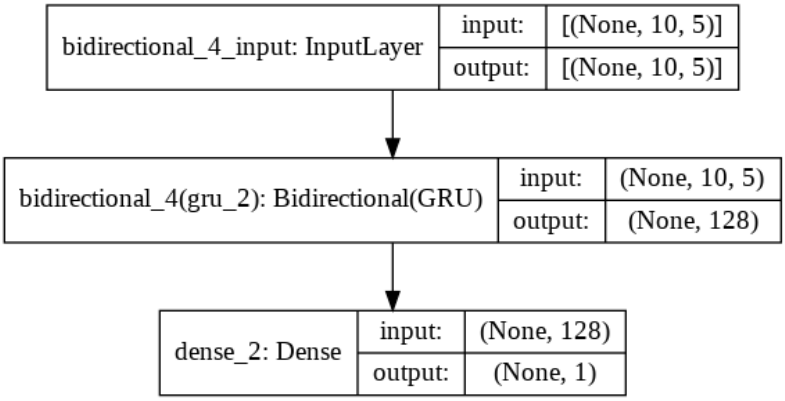

Figure 7: Single layer Architecture - Bidirectional GRU in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 814.30
Root Mean Square Error: 1683.03
Mean Percentage Error: 6.49

The MAPE value signifies that the model has an error rate of 6.49 %.

6.1.4: Single layer Architecture (Bidirectional LSTM: Sequential model)

Objective: This experiment aims to study the effect of using Bidirectional LSTM in sequential mode with the Single-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only one hidden layer is placed sequentially, where the bidirectional LSTM in the hidden layer has 64 units. The output is then passed to the dense layer consisting of a single neuron. The “mse” loss function and “adam” optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

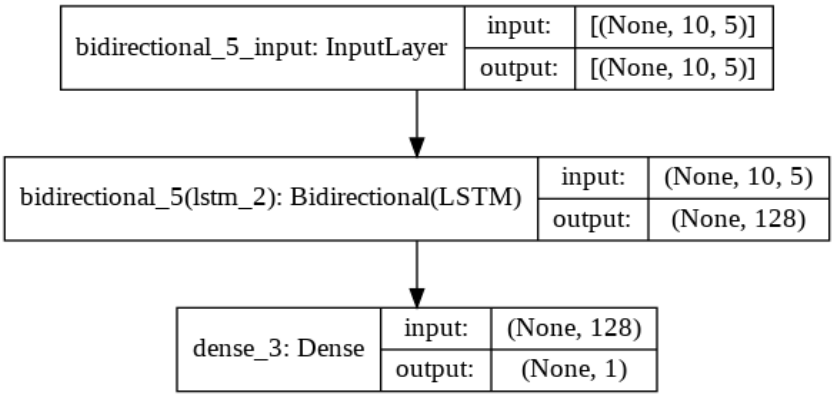

Figure 8: Single layer Architecture - Bidirectional LSTM in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1215.36
Root Mean Square Error: 2996.99
Mean Percentage Error: 8.12

The MAPE value signifies that the model has an error rate of 8.12 %.

6.1.5: Single layer Architecture (GRU: Sequential model)

Objective: This experiment aims to study the effect of using standard GRU in sequential mode with the Single-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only one hidden layer is placed sequentially, where the standard GRU in the hidden layer has 64 units. The output is then passed to the dense layer consisting of a single neuron. The “mse” loss function and “adam” optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

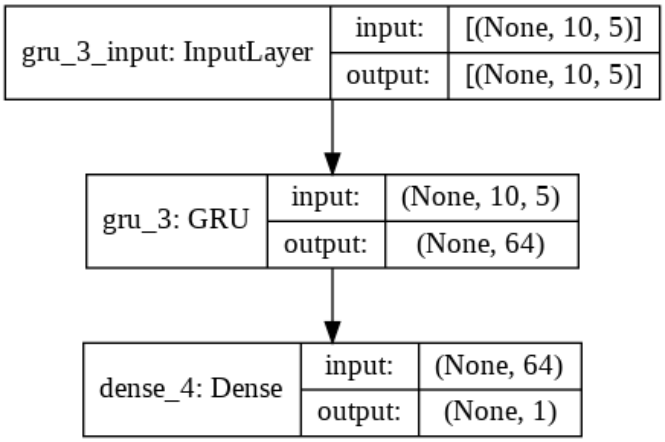

Figure 9: Single layer Architecture - GRU in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 770.89
Root Mean Square Error: 1738.87
Mean Percentage Error: 5.78

The MAPE value signifies that the model has an error rate of 5.78 %.

6.1.6: Single layer Architecture (LSTM: Sequential model)

Objective: This experiment aims to study the effect of using standard LSTM in sequential mode with the Single-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only one hidden layer is placed sequentially, where the standard LSTM in the hidden layer has 64 units. The output is then passed to the dense layer consisting of a single neuron. The "mse" loss function and "adam" optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

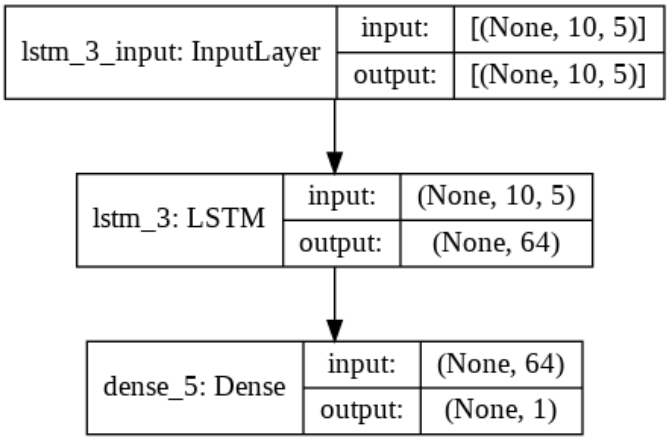

Figure 10: Single layer Architecture - LSTM in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1173.52
Root Mean Square Error: 2926.27
Mean Percentage Error: 7.83

The MAPE value signifies that the model has an error rate of 7.83 %.

Experiment 2: Two layer Architecture

6.2.1 Two layer Architecture (Bidirectional GRU: Parallel model)

Objective: This experiment aims to study the effect of using Bidirectional GRUs in parallel mode with Keras functional APIs in the Two-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only two hidden layers are placed parallel with another pair of hidden layers; the bidirectional GRUs in all hidden layers have 64 units each. The output is then concatenated and passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam“ optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

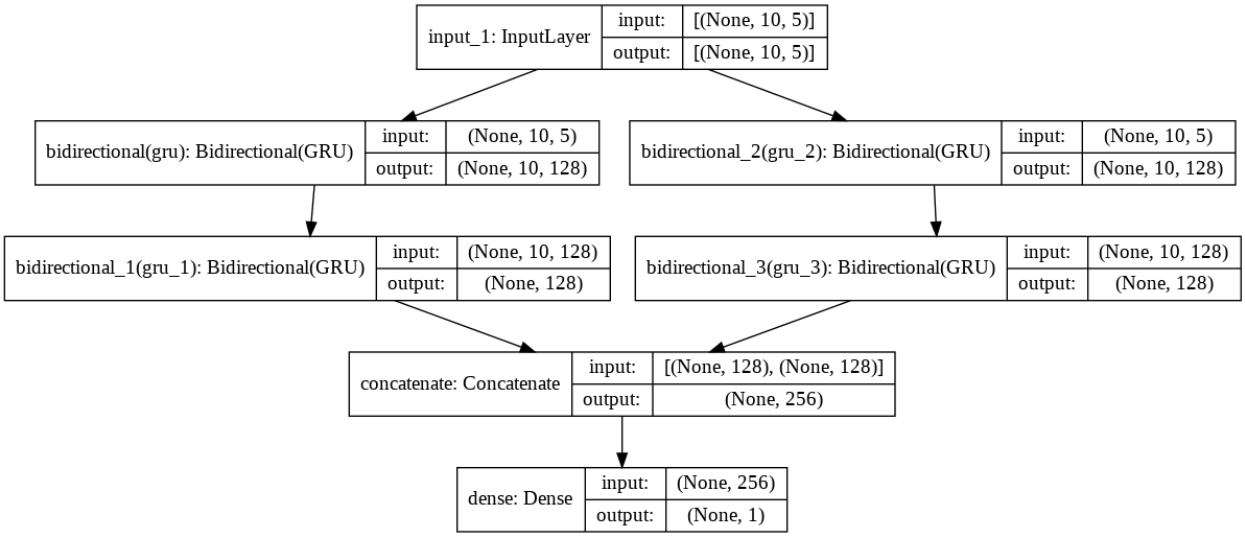

Figure 11: Two layer Architecture - Bidirectional GRU in Parallel mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 743.55
Root Mean Square Error: 1582.34
Mean Percentage Error: 5.69

The MAPE value signifies that the model has an error rate of 5.69 %.

6.2.2 Two layer Architecture (Bidirectional LSTM: Parallel model)

Objective: This experiment aims to study the effect of using Bidirectional LSTMs in parallel mode with Keras functional APIs in the Two-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only two hidden layers are placed parallel with another pair of hidden layers; the bidirectional LSTMs in all hidden layers have 64 units each. The output is then concatenated and passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam“ optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

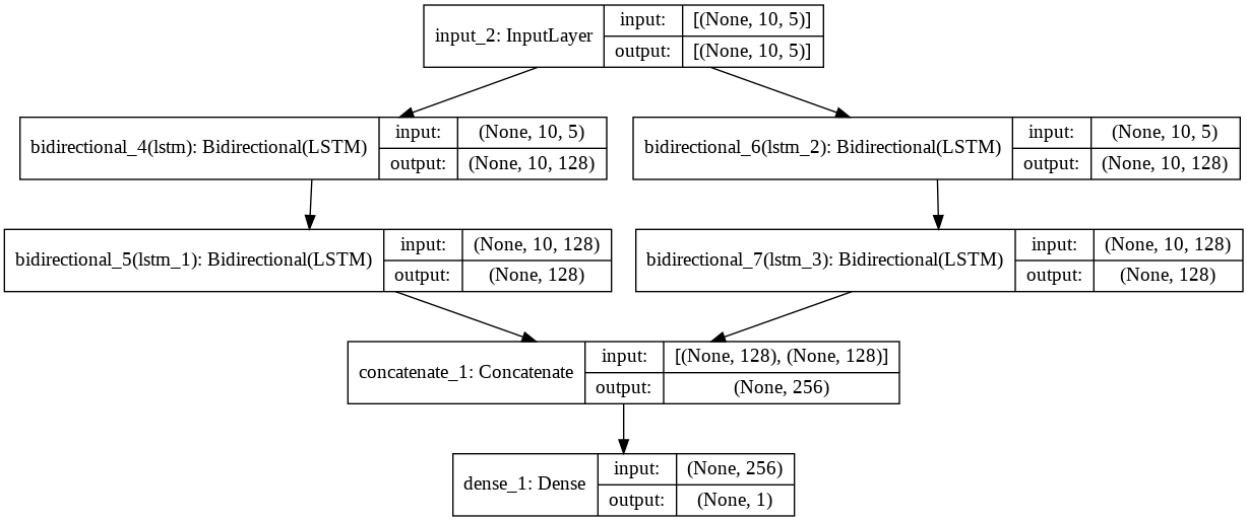

Figure 12: Two layer Architecture - Bidirectional LSTM in Parallel mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 933.11
Root Mean Square Error: 1973.65
Mean Percentage Error: 6.96

The MAPE value signifies that the model has an error rate of 6.96 %.

6.2.3 Two layer Architecture (Bidirectional GRU: Sequential model)

Objective: This experiment aims to study the effect of using Bidirectional GRUs in sequential mode with the Two-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only two hidden layers are placed sequentially, with the bidirectional GRUs in both hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam“ optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

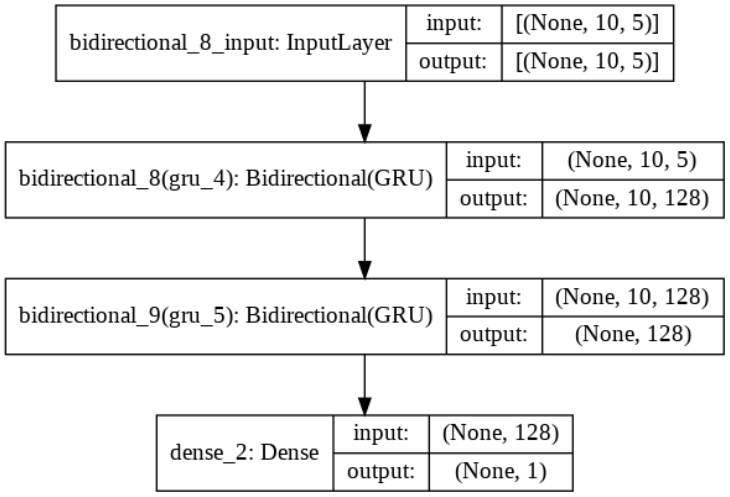

Figure 13: Two layer Architecture - Bidirectional GRU in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 678.07
Root Mean Square Error: 1361.69
Mean Percentage Error: 5.55

The MAPE value signifies that the model has an error rate of 5.55 %.

6.2.4 Two layer Architecture (Bidirectional LSTM: Sequential model)

Objective: This experiment aims to study the effect of using Bidirectional LSTMs in sequential mode with the Two-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only two hidden layers are placed sequentially with the bidirectional LSTMs in both hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam“ optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

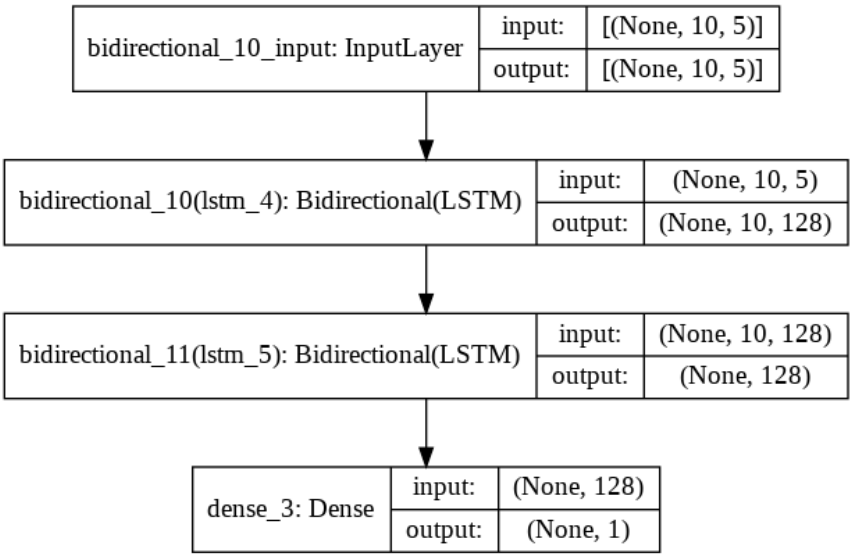

Figure 14: Two layer Architecture - Bidirectional LSTM in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 944.44
Root Mean Square Error: 2087.12
Mean Percentage Error: 6.99

The MAPE value signifies that the model has an error rate of 6.99 %.

6.2.5 Two layer Architecture (GRU: Sequential model)

Objective: This experiment aims to study the effect of using standard GRUs in sequential mode with the Two-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only two hidden layers are placed sequentially, with the standard GRUs in both hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam“ optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

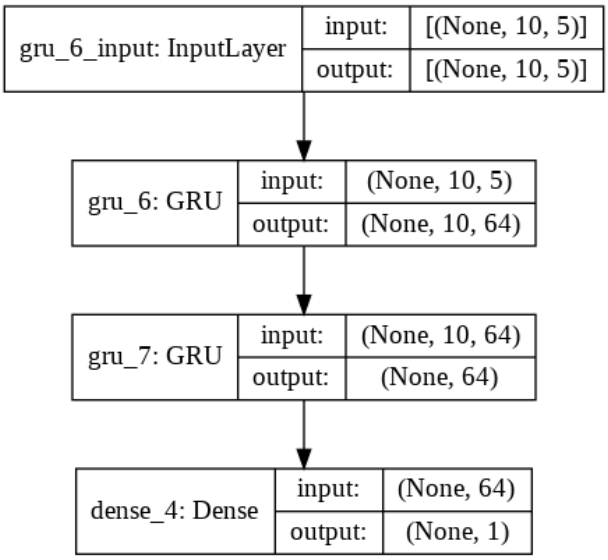

Figure 15: Two layer Architecture - GRU in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 932.70
Root Mean Square Error: 1677.66
Mean Percentage Error: 7.58

The MAPE value signifies that the model has an error rate of 7.58 %.

6.2.6 Two layer Architecture (LSTM: Sequential model)

Objective: This experiment aims to study the effect of using standard LSTMs in sequential mode with the Two-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only two hidden layers are placed sequentially with the standard LSTMs in both hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. In this process, the “mse” loss function and “adam“ optimizer are used for model compilation.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

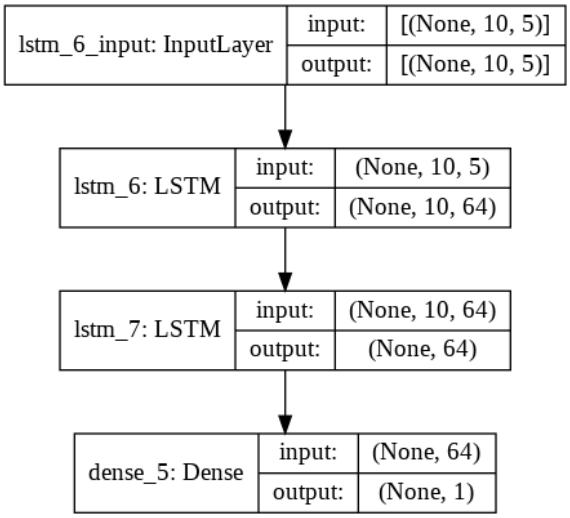

Figure 16: Two layer Architecture - LSTM in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1171.78
Root Mean Square Error: 2393.81
Mean Percentage Error: 8.55

The MAPE value signifies that the model has an error rate of 8.55 %.

6.2.7 Two layer Architecture (Adding dropout layer)

Objective: This experiment aims to study the effect of using dropout layers in a Two-layer architecture and compare its performance with other architectures.

Implementation: The above experiments 6.2.5 and 6.2.6 are repeated by adding a dropout layer to avoid overfitting of data. A dropout layer with the value of 0.2 is used after each sequential layer in the above architectures. The results obtained from this experiment use the MAPE values for comparison.

Table 3: Results of adding a dropout layer

	With dropout layer (0.2)	Without dropout layer
GRU	10.15	7.58
LSTM	11.04	8.55

It is observed from the above results that adding the dropout layer in the model worsen its performance. Therefore, it is a hyperparameter and can be tuned by checking the results with its other set of values and using the optimum value.

6.2.8 Two layer Architecture (Increasing number of units in hidden layer)

Objective: This experiment aims to study the effect of increasing the number of neurons in the hidden layers of a Two-layer architecture and then comparing their performance with other architectures.

Implementation: The above experiments 6.2.1 and 6.2.6 are repeated by doubling the number of neurons, i.e. 128 units are used in the hidden layers instead of 64 units. The results obtained from this experiment use the MAPE values for comparison.

Table 4: Results obtained after increasing number of units in hidden layers

	2-Layer Architecture (units = 128)	2-Layer Architecture (units = 64)
Func Bidirectional GRU	5.46	5.69
Func Bidirectional LSTM	6.39	6.96
Bidirectional GRU	5.57	5.55
Bidirectional LSTM	7.2	6.99
GRU	5.45	7.58
LSTM	6.23	8.55

The above results depicts that as the number of units in the hidden layer increases then the model becomes more accurate. The performance improvement is seen for almost all types of RNNs used in the above experiment.

Experiment 3: Three layer Architecture

6.3.1 Three layer Architecture (Bidirectional GRU: Parallel model)

Objective: This experiment aims to study the effect of using Bidirectional GRUs in parallel mode with Keras functional APIs in the Three-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only three single hidden layers are placed in parallel with each other; the bidirectional GRUs in all hidden layers have 64 units each. The output of all three parallel layers are then concatenated and passed through a dense layer consisting of a single neuron. The “mse” loss function and “adam“ optimizer are used for model compilation in this method.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

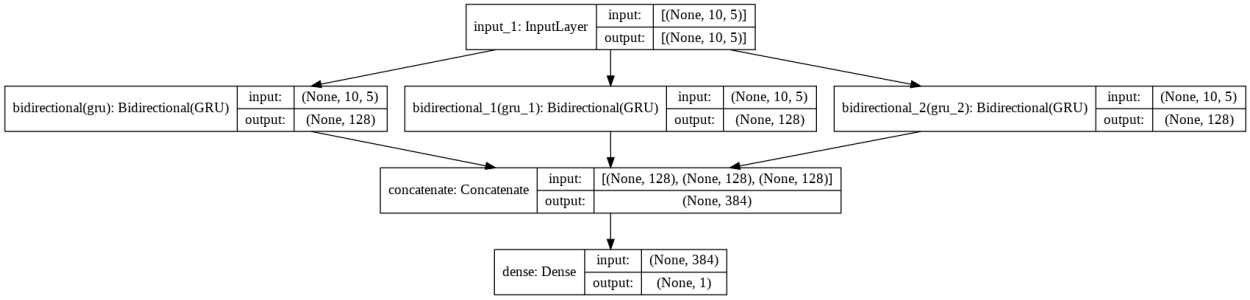

Figure 17: Three layer Architecture - Bidirectional GRU in Parallel mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 832.57
Root Mean Square Error: 1322.06
Mean Percentage Error: 7.77

The MAPE value signifies that the model has an error rate of 7.77 %.

6.3.2 Three layer Architecture (Bidirectional LSTM: Parallel model)

Objective: This experiment aims to study the effect of using Bidirectional LSTMs in parallel mode with Keras functional APIs in the Three-layer architecture and to compare its performance with other architectures.

Implementation: In this experiment, only three single hidden layers are placed in parallel with each other; the bidirectional LSTMs in all hidden layers have 64 units each. The output of all three parallel layers are then concatenated and passed through a dense layer consisting of a single neuron. The “mse” loss function and “adam“ optimizer are used for model compilation in this method.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

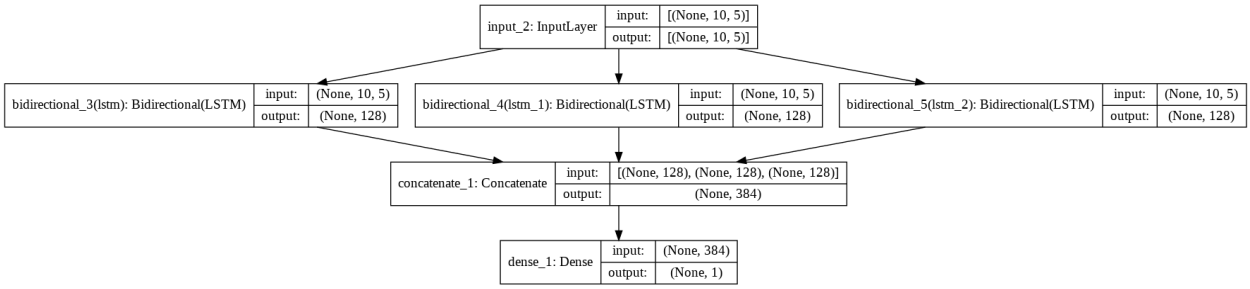

Figure 18: Three layer Architecture - Bidirectional LSTM in Parallel mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1098.90
Root Mean Square Error: 2260.79
Mean Percentage Error: 8.93

The MAPE value signifies that the model has an error rate of 8.93 %.

6.3.3 Three layer Architecture (Bidirectional GRU: Sequential model)

Objective: This experiment aims to study the effect of using Bidirectional GRUs in sequential mode with the Three-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only three hidden layers are placed sequentially, with the bidirectional GRUs in all three hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. The “mse” loss function and “adam“ optimizer are used for model compilation in this method.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

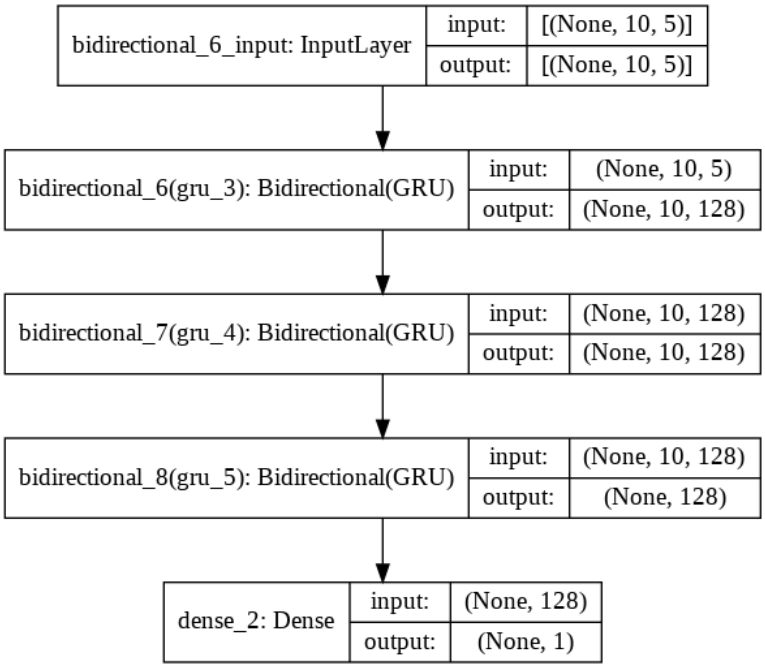

Figure 19: Three layer Architecture - Bidirectional GRU in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1094.67
Root Mean Square Error: 2111.94
Mean Percentage Error: 8.48

The MAPE value signifies that the model has an error rate of 8.48 %.

6.3.4 Three layer Architecture (Bidirectional LSTM: Sequential model)

Objective: This experiment aims to study the effect of using Bidirectional LSTMs in sequential mode with the Three-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only three hidden layers are placed sequentially with the bidirectional LSTMs in all three hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. The “mse” loss function and “adam“ optimizer are used for model compilation in this method.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

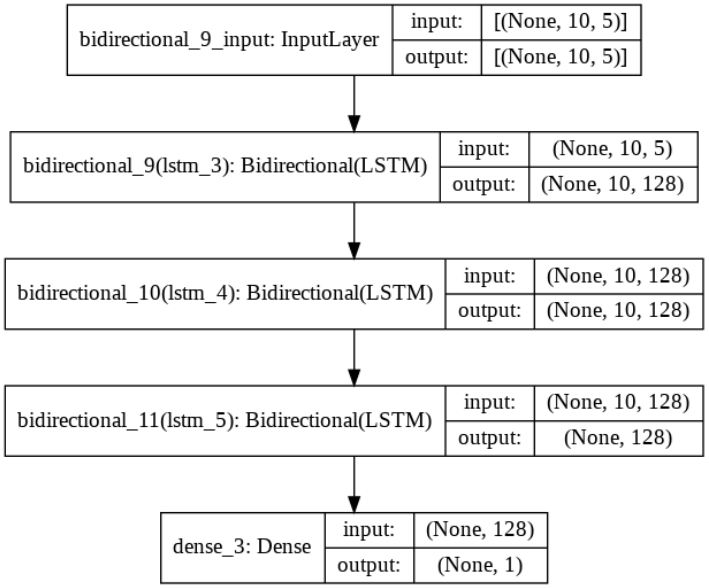

Figure 20: Three layer Architecture - Bidirectional LSTM in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1596.16
Root Mean Square Error: 3048.13
Mean Percentage Error: 12.01

The MAPE value signifies that the model has an error rate of 12.01 %.

6.3.5 Three layer Architecture (GRU: Sequential model)

Objective: This experiment aims to study the effect of using standard GRUs in sequential mode with the Three-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only three hidden layers are placed sequentially, with the standard GRUs in all three hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. The “mse” loss function and “adam“ optimizer are used for model compilation in this method.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

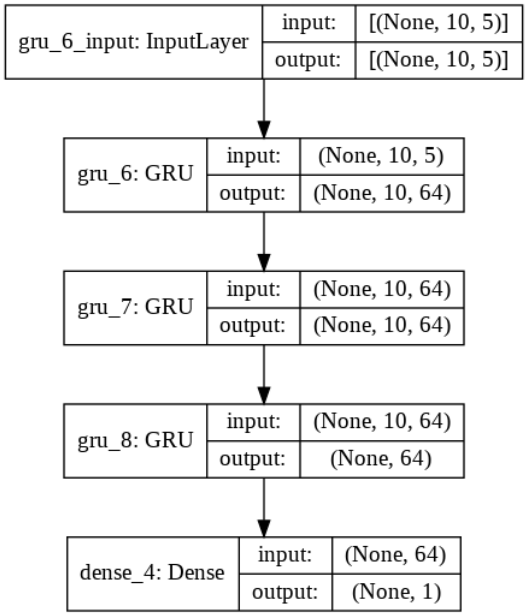

Figure 21: Three layer Architecture - GRU in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 972.89
Root Mean Square Error: 1942.86
Mean Percentage Error: 7.52

The MAPE value signifies that the model has an error rate of 7.52 %.

6.3.6 Three layer Architecture (LSTM: Sequential model)

Objective: This experiment aims to study the effect of using standard LSTMs in sequential mode with the Three-layer architecture and compare its performance with other architectures.

Implementation: In this experiment, only three hidden layers are placed sequentially with the standard LSTMs in all three hidden layers having 64 units. The output is then passed through a dense layer consisting of a single neuron. The “mse” loss function and “adam“ optimizer are used for model compilation in this method.

The architectural diagram with the above-mentioned inputs is represented in the figure below:

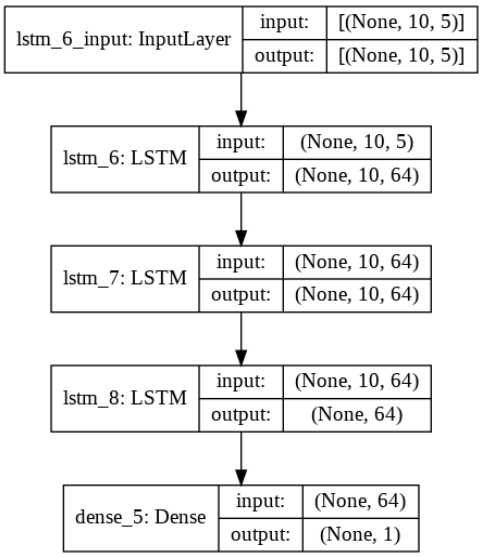

Figure 22: Three layer Architecture - LSTM in Sequential mode

The performance results obtained from the test data predictions are as follows:
Mean Absolute Error: 1424.56
Root Mean Square Error: 3339.09
Mean Percentage Error: 9.35

The MAPE value signifies that the model has an error rate of 9.35 %.

Results

The model's overall performance was found to be better in the two hidden layers architecture, where the Bidirectional GRUs had the lowest error percentage compared to other configurations. In the three hidden layers architecture, the results were worse than the other two architectures for all the LSTM and GRU configurations.
The standard GRUs have the lowest error rate of 5.78% for single-layer architecture, whereas the worst performance is observed for Bidirectional LSTMs in a parallel configuration. The Bidirectional GRUs performed the best for two-layer architecture with a 5.55% MAPE, but the standard LSTM gave the lowest precision with a MAPE value of 8.55%. The three-layer architecture in Bidirectional sequential LSTM showed the worst performance with a MAPE of 12.01%, whereas the Bidirectional GRU in parallel configuration has a MAPE of 7.77%.

The below table shows the overall MAPE result values for the Sequential, Parallel and standard configurations of GRU and LSTM networks in 1-layer, 2-layer and 3-layer architectures.

Table 5: Represents the experimental results for comparison

	1 - Layer Architecture	2 - Layer Architecture	3 - Layer Architecture
Parallel Bidirectional GRU	5.97	5.69	7.77
Parallel Bidirectional LSTM	8.59	6.96	8.93
Bidirectional GRU	6.49	5.55	8.48
Bidirectional LSTM	8.12	6.99	12.01
GRU	5.78	7.58	7.52
LSTM	7.83	8.55	9.35

Apart from the above experiments, two more sub-experiments were performed in two Layer architecture. The first was to examine the effect of the dropout layer to avoid overfitting the model. The results revealed that adding the dropout layer had deprived the performance for the standard GRU and LSTM configurations. The results obtained by performing this experiment are shown in Table 3. The other experiment was to investigate the impact of adding more neurons in the hidden layer. For this experiment, the number of units in each layer was doubled to 128 units from 64 units. The increase in the number of units also increased the time required for execution as the network did more data processing. It also increased the performance of models for all the configurations of Bidirectional, GRU and LSTMs. Therefore, it is advised to use more neurons in hidden layers to improve the learning process during the training of the network, but as the processing time also increases, a threshold value should be considered while building the network.

The results obtained from above experiments are plotted with the barplot to summarise all the conducted experiments which is helpful in comparing the results and choosing the best performance model based on the MAPE values in the graph. The best performance is found for the Bidirectional GRU configuration in the two layer architecture.

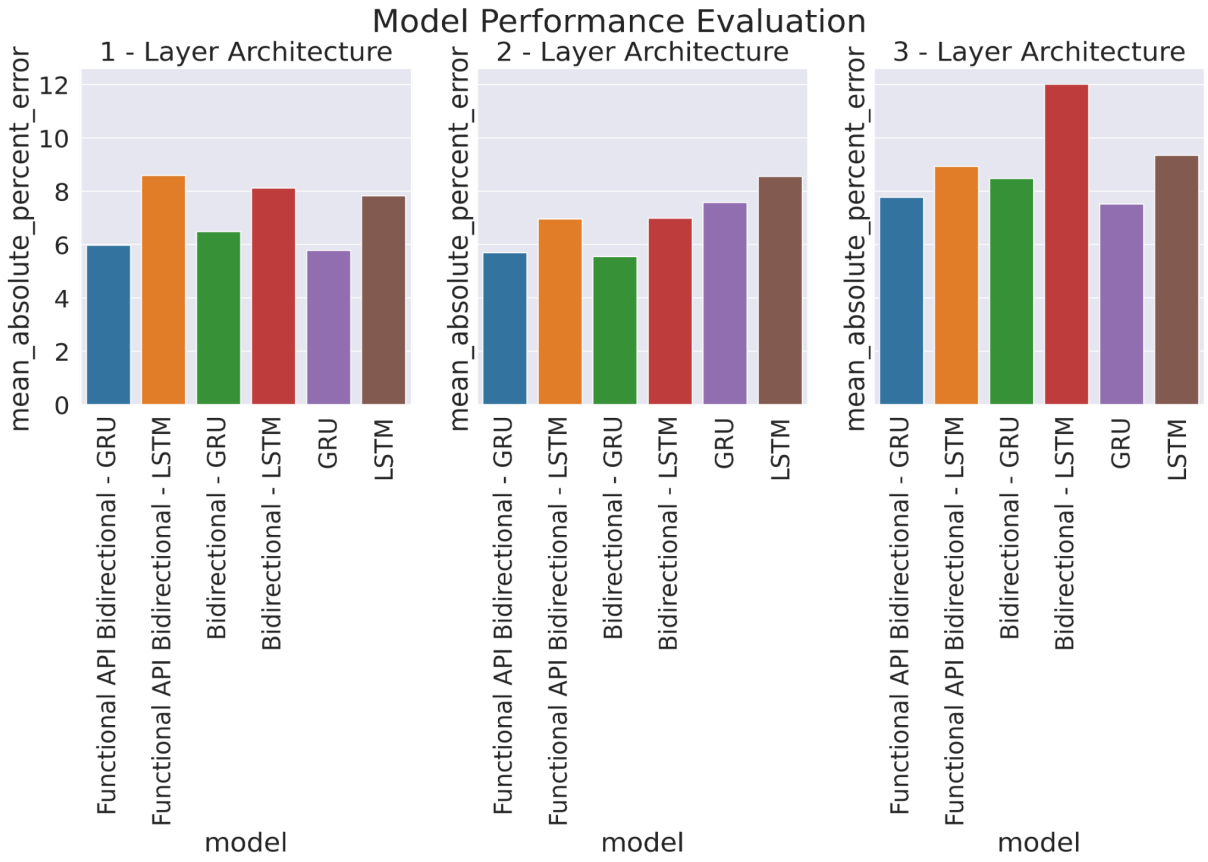

Figure 23: Comparison of model MAPE results

7. Conclusion

In this project, we have used the single and multilayer architecture for the price prediction for Bitcoin, which is the most traded cryptocurrency in the market. The same approach is applicable for other cryptocurrencies as well as the parameters for prediction remain the same. It has been observed from the literature review that RNNs provide the best performance while dealing with the time series data analysis. Here, we have used the LSTM and GRU neural networks that overcome the gradient descent problem from the RNNs and, therefore, can work efficiently with large data sequences with our aim to predict prices for the short term and reduce the risks faced by the regular investor while dealing with the cryptocurrencies. The aim of this project is achieved by performing the above experiments by building and comparing deep learning model efficiencies in multilayer architecture.

The outcomes of the performed experiments suggest that the GRUs perform better than the LSTM neural networks. Multilayer use improves the model efficiency in some instances, but adding more hidden layers in the deep learning model does not always improve the model efficiency. This can be thought of as a tradeoff value that depends upon the individual project requirements and can be determined by similar kinds of experimentations. Comparison of unidirectional and bidirectional GRUs in the single and multilayer architecture also states that its value does not depend upon the number of hidden layers introduced in the network as it can either improve or might get worse, but it does perform the best in the single-layer architecture. Also, it has been observed that increasing the number of units in hidden layers increases the performance of the model.

When the number of units(neurons) is kept uniform, and the experiments are performed by varying the number of layers in a sequential and parallel manner, then it is observed that the best results are obtained for the Bidirectional GRU in 2-Layer architecture with MAPE value 5.55. The worst performance is observed for a 3-Layer architecture Bidirectional LSTM with a MAPE value of 12.01. In general, it is noticed that when we move to a 2-Layer architecture from the 1-Layer, the results show better performance. However, the further addition of another layer, i.e. 3-Layer architecture, performs the worst compared to the other two architectures. So, the choice of a particular model is based on the project requirements; however, a higher number of the units (neurons) in the hidden layers and multilayer architecture like two layers architecture should be considered instead of just using a single layer for building a model since from our investigations it is observed this architecture configuration produce better results.

8. Future work

The results obtained in the above experiments using the different architecture with bidirectional GRUs and LSTMs can be further analyzed by optimizing the model parameters. These Hyperparameter tuning can derive better results in terms of performance. Following are some of the future work ideas which can be implemented along with the current strategy.

1. Changing the TIME_STEPS value to a lower number might result in improved prediction values as more data will be available for analysis as compared to a higher value of timestep.
2. Adding or removing the hidden layers.
3. Changing the optimiser used during model compilation.
4. Additional Hyperparameter tuning.

9. References

1. Ji, S., Kim, J. and Im, H., 2019. A comparative study of bitcoin price prediction using deep learning. *Mathematics*, *7*(10), p.898.
2. Velankar, S., Valecha, S. and Maji, S., 2018, February. Bitcoin price prediction using machine learning. In *2018 20th International Conference on Advanced Communication Technology (ICACT)* (pp. 144-147). IEEE.
3. Shankhdhar, A., Singh, A.K., Naugraiya, S. and Saini, P.K., 2021, April. Bitcoin Price Alert and Prediction System using various Models. In *IOP Conference Series: Materials Science and Engineering* (Vol. 1131, No. 1, p. 012009). IOP Publishing.
4. Rizwan, M., Narejo, S. and Javed, M., 2019, December. Bitcoin price prediction using Deep Learning Algorithm. In *2019 13th International Conference on Mathematics, Actuarial Science, Computer Science and Statistics (MACS)* (pp. 1-7). IEEE.
5. Ly, B., Timaul, D., Lukanan, A., Lau, J. and Steinmetz, E., 2018. Applying deep learning to better predict cryptocurrency trends. In *Midwest Instruction and Computing Symposium*.
6. Khare, K., Darekar, O., Gupta, P. and Attar, V.Z., 2017, May. Short term stock price prediction using deep learning. In *2017 2nd IEEE international conference on recent trends in electronics, information & communication technology (RTEICT)* (pp. 482-486). IEEE.
7. Lahmiri, S. and Bekiros, S., 2019. Cryptocurrency forecasting with deep learning chaotic neural networks. *Chaos, Solitons & Fractals*, *118*, pp.35-40.
8. Yogeshwaran, S., Kaur, M.J. and Maheshwari, P., 2019, April. Project based learning: Predicting bitcoin prices using deep learning. In *2019 IEEE Global Engineering Education Conference (EDUCON)* (pp. 1449-1454). IEEE.
9. Politis, A., Doka, K. and Koziris, N., 2021, May. Ether Price Prediction Using Advanced Deep Learning Models. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (pp. 1-3). IEEE.
10. Patel, M.M., Tanwar, S., Gupta, R. and Kumar, N., 2020. A deep learning-based cryptocurrency price prediction scheme for financial institutions. *Journal of Information Security and Applications*, *55*, p.102583.
11. Phaladisailoed, T. and Numnonda, T., 2018, July. Machine learning models comparison for bitcoin price prediction. In *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)* (pp. 506-511). IEEE.
12. Aggarwal, A., Gupta, I., Garg, N. and Goel, A., 2019, August. Deep learning approach to determine the impact of socio economic factors on bitcoin price prediction. In *2019 Twelfth International Conference on Contemporary Computing (IC3)* (pp. 1-5). IEEE.
13. Dutta, A., Kumar, S. and Basu, M., 2020. A gated recurrent unit approach to bitcoin price prediction. *Journal of Risk and Financial Management*, *13*(2), p.23.
14. Althelaya, K.A., El-Alfy, E.S.M. and Mohammed, S., 2018, April. Evaluation of bidirectional LSTM for short-and long-term stock market prediction. In *2018 9th international conference on information and communication systems (ICICS)* (pp. 151-156). IEEE.
15. Burke, R., 2021. *Bitcoin Bonanza!*. [online] Medium. Available at: <https://towardsdatascience.com/bitcoin-bonanza-2cb208026bbd> [Accessed 20 August 2021].
16. InfluxData. 2021. *What is time series data? | Definition and Discussion | InfluxData*. [online] Available at: <https://www.influxdata.com/what-is-time-series-data/> [Accessed 9 August 2021].
17. Beginners, I. and Stock?, W., 2021. *What Is the Significance of a Closing Price on a Stock?*. [online] Finance - Zacks. Available at: <https://finance.zacks.com/significance-closing-price-stock-3007.html> [Accessed 10 August 2021].
18. Darwinrecruitment.com. 2021. *4 Issues Facing Cryptocurrency Today*. [online] Available at: <https://www.darwinrecruitment.com/blog/2018/02/big-issues-cryptocurrency-today> [Accessed 10 August 2021].
19. Kadir, S., 2020. *7COM1033-0206-2019 - Neural Networks and Machine Learning*. [online] Herts.instructure.com. Available at: <https://herts.instructure.com/courses/60015/files/1568578?module_item_id=930375> [Accessed 28 June 2021].

20. Medium. 2021. *Illustrated Guide to LSTM's and GRU's: A step by step explanation*. [online] Available at: <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21> [Accessed 15 August 2021].
21. Medium. 2021. *Understanding Bidirectional RNN in PyTorch*. [online] Available at: <https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66> [Accessed 16 August 2021].
22. Medium. 2021. *Activation Functions in Neural Networks*. [online] Available at: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6> [Accessed 18 August 2021].
23. Seaborn.pydata.org. 2021. *seaborn: statistical data visualization — seaborn 0.11.2 documentation*. [online] Available at: <https://seaborn.pydata.org/> [Accessed 26 August 2021].
24. Burke, R., 2021. *ryancburke - Overview*. [online] GitHub. Available at: <https://github.com/ryancburke> [Accessed 26 August 2021].
25. Medium. 2021. *Predictive Analytics: LSTM, GRU and Bidirectional LSTM in TensorFlow*. [online] Available at: <https://towardsdatascience.com/predictive-analysis-rnn-lstm-and-gru-to-predict-water-consumption-e6bb3c2b4b02> [Accessed 26 August 2021].
26. Investopedia. 2021. *Bitcoin Definition*. [online] Available at: <https://www.investopedia.com/terms/b/bitcoin.asp> [Accessed 26 August 2021].

10. Appendix

10.1. Single Layer Model Architecture

```
***IMPORTING LIBRARIES***
"""

# Importing Data Analysis Libraries
import pandas as pd
import numpy as np

# Normalize Dataset
from sklearn.preprocessing import MinMaxScaler

# Splitting Data into train set and Test set
from sklearn.model_selection import train_test_split

# Importing Data Visualization Libraries
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.dates import DateFormatter
sns.set_style("darkgrid")

# Python drawing
from IPython.core.pylabtools import figsize

# Importing Tensorflow and Keras libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential, layers, callbacks
from tensorflow.keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional, Input
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import CSVLogger # Generating Model logs
from tensorflow.keras.utils import plot_model


# Model Evaluation
from sklearn.metrics import r2_score

# Convert Object to Datetime and display in matplotlib without casting
```

```
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()

"""***DEFINING GLOBAL VARIABLES AND FUNCTIONS FOR CODE REUSE***"""

# Creating directory for saving log and model.
!mkdir -p saved_data

# CSVLogger variable for generating model logs
csv_logger = CSVLogger('saved_data/training.log', ',', append=True)

# Future close variable
next_close = 'FUTURE_CLOSE'

# Hyperparameter for data transformation
TIME_STEPS = 10

# Number of neurons used
total_units = 64

# FUNCTION TO PLOT A FIGURE OBJECT
legend_loc= 'upper left'
def plot_figure(xlabel, ylabel, title, legend_loc):
    plt.figure(figsize=(10, 6))
    plt.rcParams['figure.dpi'] = 360
    plt.title(title)
    plt.legend(loc=legend_loc)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    return plt

"""***IMPORTING DATA***"""

# Fetching the csv dataset from GitHub repository
crypto_dataframe =
pd.read_csv("https://raw.githubusercontent.com/247pankaj/crypto-dataset/main/Bitcoin_
price.csv")

# Printing the last 5 records in dataset
crypto_dataframe.tail()
```

```
# Feature DataTypes
crypto_dataframe.dtypes

"""***DATA PREPROCESSING***"""

# Converting Date Object into Datetime
crypto_dataframe['Date'] = crypto_dataframe['Date'].astype('datetime64[ns]')

# Plotting the dataset
plot_fig = plot_figure("Date", "Close Value(US$)", "Historical Data", legend_loc)
#plot_figure(xlabel, ylabel, title, legend_loc)
sns.set_context("paper", font_scale=2.1, rc={"lines.linewidth": 3.2})
sns.lineplot(data=crypto_dataframe, x="Date", y="Close")
horizontalalignment='center'
sns.despine()

crypto_dataframe[next_close] = crypto_dataframe['Close'].shift(-1,fill_value=0)

crypto_dataframe.drop(crypto_dataframe.tail(1).index,inplace=True)

crypto_dataframe = crypto_dataframe.drop(columns=['Close'])

crypto_dataframe = crypto_dataframe.set_index('Date')

crypto_dataframe.tail()

# Train-Test Split(80%-20%)
training_size = int( len (crypto_dataframe) * 0.80)
training_dataset, testing_dataset =
crypto_dataframe.iloc[:training_size],crypto_dataframe.iloc[training_size:]

plt_fig = plot_figure("Date", "Close Value(US$)", "Train & Test plot", legend_loc)
#plot_figure(xlabel, ylabel, title, legend_loc)

plt_fig.plot(training_dataset.FUTURE_CLOSE)
plt_fig.plot(testing_dataset.FUTURE_CLOSE)
plt_fig.legend(['Training dataset', 'Testesting dataset'], loc='upper left')

# Splitting training_dataset into Predictors and outcomes
```

```
X_train = training_dataset.drop(next_close, axis = 1)
y_train = training_dataset.loc[:,[next_close]]

# Splitting testing_dataset into Predictors and outcomes
X_test = testing_dataset.drop(next_close, axis = 1)
y_test = testing_dataset.loc[:,[next_close]]

min_max_scaler = MinMaxScaler(feature_range = (-1,1))

# Fitting MinMaxScaler to Trainset(X_train,y_train)
predictor_scaler = min_max_scaler.fit(X_train)
outcome_scaler = min_max_scaler.fit(y_train)

# Scaler in trainset
y_norm_train = outcome_scaler.transform(y_train)
x_norm_train = predictor_scaler.transform(X_train)

# Scaler in testset
y_norm_test = outcome_scaler.transform(y_test)
x_norm_test = predictor_scaler.transform(X_test)

"""***DATA TRANSFORMATION***"""

def transform_dataset (X_array, y_array, time_steps = 1):
    X_transform, y_transform = [], []
    for i in range(len(X_array) - time_steps):
        output = X_array[i:i+time_steps, :]
        X_transform.append(output)
        y_transform.append(y_array[i+time_steps])

    return np.array(X_transform), np.array(y_transform)


X_test, y_test = transform_dataset(x_norm_test, y_norm_test, TIME_STEPS)
X_train, y_train = transform_dataset(x_norm_train, y_norm_train, TIME_STEPS)

print('X_train: ', X_train)

"""***DATA MODELING***
```

```
*   Model Creation
*   Model Training
*   Hyperparameter Tune
*   Sequential and parallel architecture implementation
"""

# BiRNN Model using Functional API of Keras (Parallel)

def build_model_birnn_func(units, mt):
    input1 = Input(shape=(X_train.shape[1], X_train.shape[2]))
    hidden1 = Bidirectional(mt(units = units) )(input1)

    hidden2 = Bidirectional(mt(units = units))(input1)

    hidden3 = keras.layers.concatenate([hidden1, hidden2]) # Concat
    output = Dense(1)(hidden3)

    parallelModel = Model(inputs=input1, outputs=output)

    parallelModel.compile(optimizer='adam',loss='mse')
    return parallelModel

# Building the Sequential Bidirectional model
def build_model_birnn(units, mt):
    sequentialModel = Sequential()
    sequentialModel.add(Bidirectional(mt(units), input_shape=(X_train.shape[1],
X_train.shape[2])))
    sequentialModel.add(Dense(1))

    sequentialModel.compile(optimizer='adam',loss='mse')
    return sequentialModel


# Building LSTM or GRU Sequential model
def build_model_sequential(units, mt):
    sequentialModel = Sequential()
    sequentialModel.add(mt(units,  input_shape=(X_train.shape[1], X_train.shape[2])))
    sequentialModel.add(Dense(1))
    sequentialModel.compile(optimizer='adam',loss='mse')
```

```
    return sequentialModel

# Functional API BiRNN
func_biGruModel = build_model_birnn_func(total_units, GRU)
func_biLstmModel = build_model_birnn_func(total_units, LSTM)


# BiRNN Sequential GRU and LSTM
biGruModel = build_model_birnn(total_units, GRU)
biLstmModel = build_model_birnn(total_units, LSTM)


# GRU and LSTM Sequential
gruModel = build_model_sequential(total_units, GRU)
lstmModel = build_model_sequential(total_units, LSTM)


# Model summary and plot
func_biGruModel.summary()
plot_model(func_biGruModel, to_file='saved_data/func_biGruModel.png',
show_shapes=True, show_layer_names=True)

func_biLstmModel.summary()
plot_model(func_biLstmModel, to_file='saved_data/func_biLstmModel.png',
show_shapes=True, show_layer_names=True)

biGruModel.summary()
plot_model(biGruModel, to_file='saved_data/biGruModel.png', show_shapes=True,
show_layer_names=True)

biLstmModel.summary()
plot_model(biLstmModel, to_file='saved_data/biLstmModel.png', show_shapes=True,
show_layer_names=True)

gruModel.summary()
plot_model(gruModel, to_file='saved_data/gruModel.png', show_shapes=True,
show_layer_names=True)

lstmModel.summary()
```

```
plot_model(lstmModel, to_file='saved_data/lstmModel.png', show_shapes=True,
show_layer_names=True)

# Existing models Fit function
def fit_mType(mType):
    early_stopping = keras.callbacks.EarlyStopping(monitor = 'val_loss',
                                        patience = 10)

    # shuffle = False because the order of the data matters
    modelHistory = mType.fit(X_train, y_train, epochs = 10, validation_split = 0.2,
                batch_size = 32, shuffle = False, callbacks = [csv_logger,early_stopping])
    return modelHistory

#Fitting all 6 models created above
parallel_biGru_history = fit_mType(func_biGruModel)
parallel_biLstm_history = fit_mType(func_biLstmModel)

sequential_biGru_history = fit_mType(biGruModel)
sequential_biLstm_history = fit_mType(biLstmModel)

sequential_gru_history = fit_mType(gruModel)
sequential_lstm_history = fit_mType(lstmModel)

y_test = min_max_scaler.inverse_transform(y_test)
y_train = min_max_scaler.inverse_transform(y_train)

pred = gruModel.predict(X_test)
pred.shape

# Prediction
def perform_prediction(mType):
    predict_values = mType.predict(X_test)
    predict_values = min_max_scaler.inverse_transform(predict_values)
    return predict_values

predicted_parallel_biGru = perform_prediction(func_biGruModel)
predicted_parallel_biLstm = perform_prediction(func_biLstmModel)

predicted_sequential_biGru = perform_prediction(biGruModel)
predicted_sequential_biLstm = perform_prediction(biLstmModel)
```

```
predicted_sequential_Gru = perform_prediction(gruModel)
predicted_sequential_Lstm = perform_prediction(lstmModel)

def plot_prediction(prediction, model_type, y_test):

    plt.figure(figsize=(10, 6))
    plt.rcParams['figure.dpi'] = 360
    range_future = len(prediction)


    plt.plot(np.arange(range_future), np.array(y_test), label='True Future')
    plt.plot(np.arange(range_future), np.array(prediction),label='Prediction')

    plt.title('Actual vs prediction for ' + model_type)
    plt.legend(loc='upper left')
    plt.xlabel('Data')
    plt.ylabel('Close value (US $)')


plot_prediction(predicted_parallel_biGru, 'Functional API BiGRU', y_test)
plot_prediction(predicted_parallel_biLstm, 'Functional API BiLSTM', y_test)

plot_prediction(predicted_sequential_biGru, 'BiGRU', y_test)
plot_prediction(predicted_sequential_biLstm, 'BiLSTM', y_test)

plot_prediction(predicted_sequential_Gru, 'GRU', y_test)
plot_prediction(predicted_sequential_Lstm, 'LSTM', y_test)

def prediction_perfomance(predicted_values, actual_values, model_type):
    errors = predicted_values - actual_values
    mse = np.square(errors).mean()

    rmse = np.sqrt(mse)
    mae = np.abs(errors).mean()
    mape = np.mean(np.abs((actual_values - predicted_values) / actual_values)) * 100

    print(model_type + ':')
    print('Mean Absolute Error: {:.2f}'.format(mae))
    print('Root Mean Square Error: {:.2f}'.format(rmse))
```

```
    print('Mean Percentage Error: {:.2f}'.format(mape))
    print("\n")



prediction_perfomance(predicted_parallel_biGru, y_test, 'Func Bidirectional GRU')
prediction_perfomance(predicted_parallel_biLstm, y_test, 'Func Bidirectional LSTM')

prediction_perfomance(predicted_sequential_biGru, y_test, 'Bidirectional GRU')
prediction_perfomance(predicted_sequential_biLstm, y_test, 'Bidirectional LSTM')

prediction_perfomance(predicted_sequential_Gru, y_test, 'GRU')
prediction_perfomance(predicted_sequential_Lstm, y_test, 'LSTM')

"""
corr_matrix = np.corrcoef(y_test, prediction_func_bilstm)
corr = corr_matrix[0,1]
R_sq = corr**2

print(R_sq)
"""

# Calling `save('my_model')` creates a SavedModel folder `my_model`.

gruModel.save("saved_data/gruModel")

# It can be used to reconstruct the model identically.
#reconstructed_model = keras.models.load_model("biLstmModel_total_units")

!ls saved_data

new_model = tf.keras.models.load_model('saved_data/gruModel')

# Check its architecture
new_model.summary()

'''
from google.colab.patches import cv2_imshow # for image display

cv2_imshow('saved_data/func_biGruModel.png')
```

```
cv2_imshow('saved_data/func_biLstmModel.png')
cv2_imshow('saved_data/biGruModel.png')
cv2_imshow('saved_data/biLstmModel.png')
cv2_imshow('saved_data/gruModel.png')
cv2_imshow('saved_data/lstmModel.png')
'''

from tabulate import tabulate

d = [ ["Parallel Bidirectional - GRU", 5.97,         5.69,   7.77 ],
    ["Parallel Bidirectional - LSTM", 8.59, 6.96,   8.93],
    ["Bidirectional - GRU", 6.49,     5.55,   8.48],
    ["Bidirectional - LSTM", 8.12,   6.99,   12.01 ],
    ["GRU", 5.78,      7.58,   7.52],
    ["LSTM", 7.83,    8.55,   9.35]]

print(tabulate(d, headers=["Model", "MAPE(1-layer)", "MAPE(2-layer)",
"MAPE(3-layer)"]))

performance_data = {'architecture': ['1 - Layer', '2 - Layer', '3 - Layer', '1 - Layer', '2 -
Layer', '3 - Layer', '1 - Layer', '2 - Layer', '3 - Layer','1 - Layer', '2 - Layer', '3 - Layer', '1 -
Layer', '2 - Layer', '3 - Layer', '1 - Layer', '2 - Layer', '3 - Layer'],
    'model': ["Parallel Bidirectional - GRU","Parallel Bidirectional - GRU","Parallel
Bidirectional - GRU", "Parallel Bidirectional - LSTM", "Parallel Bidirectional -
LSTM","Parallel Bidirectional - LSTM","Bidirectional - GRU","Bidirectional -
GRU","Bidirectional - GRU","Bidirectional - LSTM","Bidirectional - LSTM","Bidirectional -
LSTM", "GRU","GRU","GRU", "LSTM", "LSTM", "LSTM"],
    'mean_absolute_percent_error':[5.97,5.69,   7.77,
8.59,   6.96,   8.93,
6.49,   5.55,   8.48,
8.12,   6.99,   12.01,
5.78,   7.58,   7.52,
7.83,   8.55,   9.35]}

dfp = pd.DataFrame(data=performance_data)
data_sns = dfp.set_index('architecture')

layer_1 = data_sns.loc[ '1 - Layer' ]
layer_2 = data_sns.loc['2 - Layer']
layer_3 = data_sns.loc['3 - Layer']
```

```
fig, axes = plt.subplots(1, 3, figsize=(15, 5), sharey=True)
fig.suptitle('Model Performance Evaluation ')


# Layer_1
sns.barplot(ax=axes[0], x=layer_1.model, y=layer_1.mean_absolute_percent_error)
axes[0].set_title('1 - Layer Architecture')
axes[0].tick_params(axis='x', labelrotation=90)


# Layer_2
sns.barplot(ax=axes[1], x=layer_2.model, y=layer_2.mean_absolute_percent_error)
axes[1].set_title('2 - Layer Architecture')
axes[1].tick_params(axis='x', labelrotation=90)


# Layer_3
sns.barplot(ax=axes[2], x=layer_3.model, y=layer_3.mean_absolute_percent_error)
axes[2].set_title('3 - Layer Architecture')
axes[2].tick_params(axis='x', labelrotation=90)

plt.subplots_adjust(top=0.85)
```

10.2. Two Layer Model Architecture

```
# BiRNN Model using Functional API of Keras (Parallel)
def build_model_birnn_func(units, mt):
  input1 = Input(shape=(X_train.shape[1], X_train.shape[2]))
  hidden1 = Bidirectional(mt(units = 2*units, return_sequences=True))(input1)
  hidden2 = Bidirectional(mt(units = 2*units))(hidden1)

  hidden3 = Bidirectional(mt(units = units, return_sequences=True))(input1)
  hidden4 = Bidirectional(mt(units = units))(hidden3)

  hidden5 = keras.layers.concatenate([hidden2, hidden4])
  output = Dense(1)(hidden5)

  parallelModel = Model(inputs=input1, outputs=output)

  parallelModel.compile(optimizer='adam', loss='mse')
  return parallelModel


# Building the Sequential Bidirectional model
def build_model_birnn(units, mt):
  sequentialModel = Sequential()
  sequentialModel.add(Bidirectional(mt(units = units, return_sequences=True),
input_shape=(X_train.shape[1], X_train.shape[2])))

  sequentialModel.add(Bidirectional(mt(units = units)))
  sequentialModel.add(Dense(1))

  sequentialModel.compile(optimizer='adam',loss='mse')
  return sequentialModel


# Building LSTM or GRU Sequential model
```

```
def build_model_sequential(units, mt):
  sequentialModel = Sequential()

  sequentialModel.add(mt (units = units, return_sequences = True,
            input_shape = [X_train.shape[1], X_train.shape[2]]))

  sequentialModel.add(mt (units = units))
  sequentialModel.add(Dense(units = 1))

  sequentialModel.compile(optimizer='adam',loss='mse')
  return sequentialModel

# Functional API BiRNN
func_biGruModel = build_model_birnn_func(total_units, GRU)
func_biLstmModel = build_model_birnn_func(total_units, LSTM)


# BiRNN Sequential GRU and LSTM
biGruModel = build_model_birnn(total_units, GRU)
biLstmModel = build_model_birnn(total_units, LSTM)


# GRU and LSTM Sequential
gruModel = build_model_sequential(total_units, GRU)
lstmModel = build_model_sequential(total_units, LSTM)

# Model summary and plot
func_biGruModel.summary()
plot_model(func_biGruModel, to_file='saved_data/func_biGruModel.png',
show_shapes=True, show_layer_names=True)
```

10.3. Three Layer Model Architecture

```
# BiRNN Model using Functional API of Keras (Parallel)
def build_model_birnn_func(units, mt):
  input1 = Input(shape=(X_train.shape[1], X_train.shape[2]))
  hidden1 = Bidirectional(mt(units = units) )(input1)

  hidden2 = Bidirectional(mt(units = units) )(input1)

  hidden3 = Bidirectional(mt(units = units) )(input1)

  hidden4 = keras.layers.concatenate([hidden1, hidden2,hidden3])
  output = Dense(1)(hidden4)

  parallelModel = Model(inputs=input1, outputs=output)

  parallelModel.compile(optimizer='adam',loss='mse')
  return parallelModel


# Building the Sequential Bidirectional model
def build_model_birnn(units, mt):
  sequentialModel = Sequential()
  sequentialModel.add(Bidirectional(mt(units = units, return_sequences=True),
input_shape=(X_train.shape[1], X_train.shape[2])))
  sequentialModel.add(Bidirectional(mt(units = units, return_sequences=True),
input_shape=(X_train.shape[1], X_train.shape[2])))

  sequentialModel.add(Bidirectional(mt(units = units)))
  sequentialModel.add(Dense(1))

  sequentialModel.compile(optimizer='adam', loss='mse')
  return sequentialModel
```

```
# Building LSTM or GRU Sequential model
def build_model_sequential(units, mt):
  sequentialModel = Sequential()

  sequentialModel.add(mt(units, return_sequences=True,
input_shape=[X_train.shape[1], X_train.shape[2]]))  # returns a sequence of vectors of
dimension 32
  sequentialModel.add(mt(units, return_sequences=True))  # returns a sequence of
vectors of dimension 32
  sequentialModel.add(mt(units))  # return a single vector of dimension 32

  sequentialModel.add(Dense(1))

  sequentialModel.compile( optimizer='adam',loss='mse')
  return sequentialModel

# Functional API BiRNN
func_biGruModel = build_model_birnn_func(total_units, GRU)
func_biLstmModel = build_model_birnn_func(total_units, LSTM)


# BiRNN Sequential GRU and LSTM
biGruModel = build_model_birnn(total_units, GRU)
biLstmModel = build_model_birnn(total_units, LSTM)


# GRU and LSTM Sequential
gruModel = build_model_sequential(total_units, GRU)
lstmModel = build_model_sequential(total_units, LSTM)

# Model summary and plot
func_biGruModel.summary()
plot_model(func_biGruModel, to_file='saved_data/func_biGruModel.png',
show_shapes=True, show_layer_names=True)
```