FORECASTING FINANCIAL TIME SERIES: AN EMPIRICAL ANALYSIS OF LSTM MODEL PERFORMANCE ACROSS MINUTE, HOURLY, AND DAILY TIME FRAMES

A Dissertation
Presented to
The Academic Faculty

By

Gabriel Dutra

In Partial Fulfillment
of the Requirements for the Baldwin Honors Degree
in Computer Science and Statistics
School of Computer Science
Department of Mathematics and Computer Science

Drew University

May 2023

FORECASTING FINANCIAL TIME SERIES: AN EMPIRICAL ANALYSIS OF LSTM MODEL PERFORMANCE ACROSS MINUTE, HOURLY, AND DAILY TIME FRAMES

Thesis committee:

Dr. Steven Kass
Mathematics and Computer Science
Drew University

Dr. Hamed Yousefi
Finance
Drew University

Dr. Yi Lu
Mathematics and Computer Science
Drew University

Date approved: April 24th, 2023

ACKNOWLEDGMENTS

I would like to express my gratitude to my thesis committee members, Dr. Steven Kass, Dr. Yi Lu, and Dr. Hamed Yousefi, for their guidance, support, and valuable feedback throughout my research journey. Your expertise and constructive criticism have greatly contributed to the completion of this thesis.

I am particularly grateful to my main advisor, Dr. Kass, for his dedication, patience, and constant availability to answer my questions and provide valuable insights during the development of this project. Your mentorship and encouragement have been invaluable in shaping my academic career.

I would also like to extend my appreciation to Dr. Minjoon Kouh, my research advisor during the Drew Summer Science Institute (DSSI) program, for introducing me to the exciting world of Machine Learning and LSTM models, and for sparking my passion for science and research.

Finally, I am deeply grateful to my family and friends for their help and encouragement throughout my academic journey. Their unwavering support and understanding have been crucial in helping me overcome challenges and achieve my goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Time series forecasting plays a crucial role in the world of finance. Whether in calculating risk management, or taking investment decisions, the ability to estimate future trends accurately is vital to maximizing profit. In recent years, machine-learning algorithms gained popularity for being able to obtain better results in forecasting than previous statistical models. The success of these algorithms, however, relies heavily on selecting the correct features of the data, as well as fine-tuning multiple parameters in the machine-learning model. Although research in this field is extensive, most published articles use daily historical financial data to perform analysis and draw results, while also relying on large training sets. Experiments that use small training sets, or different time frames, such as hourly and minute data, are still uncommon, which leaves the question of whether results obtained with a single time frame and training set can be generalized. To answer these questions, we developed a framework that creates, trains, and evaluates Long Short Term Memory (LSTM) neural network models for minute, hourly, and daily data of a given time series. After testing the framework with 2022 and 2023 data from the SPDR SP 500 ETF (SPY) and NASDAQ 100 (NQ) quotes, as well as 2020 and 2021 Bitcoin (BTC) data, we found that utilizing hourly data improved the forecasting results of our model when compared to the daily data. We also found that models trained on minute data failed to capture any trend in the data, resulting in higher forecasting errors when compared to the other two time frames.

CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Time Series

A time series can be formally defined as a collection of observations made sequentially in time [1]. In other words, it can be thought of as a "list of numbers, along with some information about what time those numbers were recorded" [2]. Mathematically, we can represent a time series as a sequence composed of multiple values:

$$x = (X_t; t \in T)$$

where X_t is an observation taken at a time t.

A common instance of time series is the prices of stocks or currencies. We can exemplify this by looking at the closing daily prices of Bitcoin (BTC) at the beginning of 2020 (Figure 1.1).

DateTime	Close
2020-01-01	7178.68
2020-01-02	6950.56
2020-01-03	7338.91
2020-01-04	7344.48
2020-01-05	7356.70

Figure 1.1: Daily closing prices of BTC (in US$)

Other common examples of time series are the monthly power consumption of a city, daily calories consumed by a person, the yearly average temperature of a country, etc.

1.2 Forecasting and Modeling

Forecasting is the process of estimating future, unknown data. In the context of time series, forecasting can be defined as the estimation of data points that have not been recorded [2]. When we establish a set of rules used to forecast data, we have a model.

Let us exemplify the idea of forecasting by looking at the closing daily price of BTC in January 2020 (Figure 1.2):

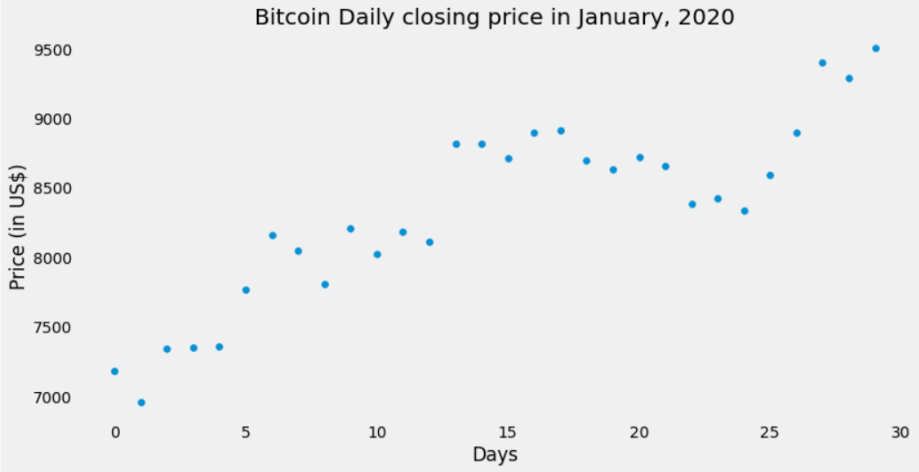

Figure 1.2: Daily closing prices of BTC (in US$)

Imagine that we want to forecast the closing price of BTC for the first two weeks of February by using this data. One way of doing it is assuming that the prices for the next 15 days are equal to the last recorded price. In other words, we assume that the price will not vary. We can visualize this in Figure 1.3.

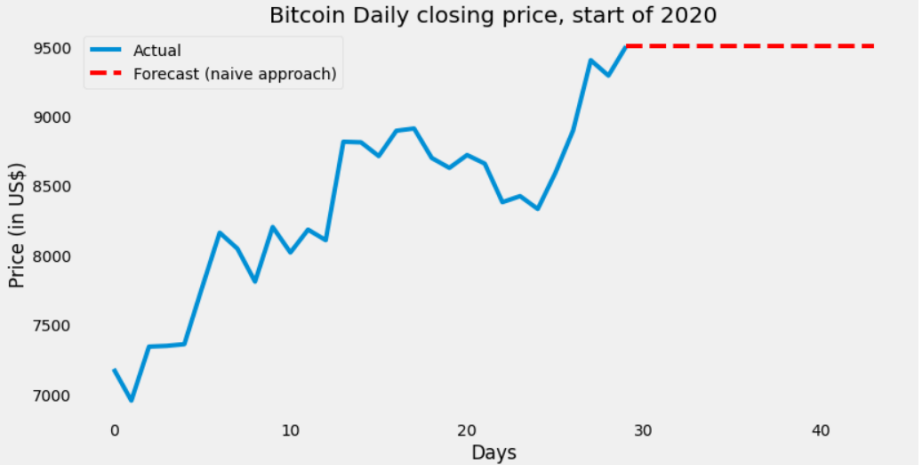

Figure 1.3: Naive BTC price forecast

Due to its simplicity, we can call this form of forecasting the naive model. While this model might not have real world applications, we will use it as a baseline to compare the forecasting performance of other models.

Another, slightly more complex way to make predictions would be to perform Linear Regression. This modeling approach consists of finding a line that best fits all the available data points, and then extending it to predict future ones. We obtain the line of best fit by minimizing the average square difference between a line and all the data points available [1]. We can visualize it in Figure 1.4.

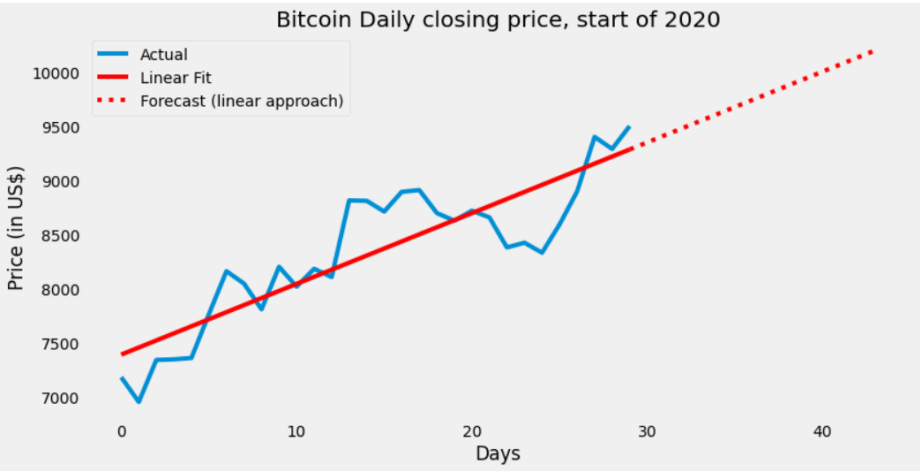

Figure 1.4: Linear BTC price forecast

Both models are valid ways of making estimates, and they might see different uses depending on what the goal of the estimate is, or how the data is distributed.

To compare different models on a particular forecasting task, we need to choose metrics that will quantify the performance of said models. Being able to define what is the 'best' model is essential when assessing different models

1.3 Accuracy

A common concept used in time series forecasting is that of error, which can be defined as the difference between the actual and predicted values. The lower the error, the closer the data point is to the actual value. There are multiple ways of calculating the overall error of a set of predicted values. Three of the most common ones are the Mean Squared Error (MSE), Symmetric Mean Percent Average Error (sMAPE), and Mean Absolute Error (MAE) [3]. Each formula presents a unique way of assessing the accuracy of a model. We present each formula's definition and properties below.

1.3.1 Mean Squared Error (MSE)

The MSE is calculated by taking the sum of the squared differences between the predicted and actual values and dividing by the number of observations. Its formula is:

$$\text{MSE} = \sum_{t=1}^{n} \frac{(F_t - A_t)^2}{n}$$

where A_t stands for the actual value at time t, F_t is the forecast value at the same time and n is the number of observations [3].

By squaring the error differences, the MSE value becomes more sensitive to outliers, meaning that models that fail to predict outliers will present a very high MSE value. Furthermore, the lower bound of MSE values is 0, and the upper bound is directly dependent on the values of the data and the predictions.

1.3.2 Mean Absolute Error

MAE measures the average of the absolute differences between predicted and actual values, and it is defined as follows:

$$\text{MAE} = \sum_{t=1}^{n} \frac{|F_t - A_t|}{n}$$

MAE is very similar to MSE, presenting only two key differences: It is less sensitive to outliers, and its result is presented on the same unit as the data, making it more intuitive to interpret. Similarly to the MSE, the MAE's lower bound is zero, with its upper bound being dependent on the data.

1.3.3 Symmetric Mean Percent Average Error (sMAPE)

sMAPE is calculated by taking the average of the absolute difference between the predicted and actual values, divided by half the sum of the predicted and actual values. Its formula is:

$$\text{sMAPE} = \frac{1}{n} \sum_{t=1}^{n} \frac{|F_t - A_t|}{(A_t + F_t)/2}$$

The biggest advantage of this formula is its well-defined bounds, as it ranges between 0% and 200%. When the sMAPE is 0%, it indicates a perfect forecast, where the predicted and actual values are the same. When it is 200%, the predicted values are three times the size of the actual values, on average [4].

1.3.4 Shortcomings of Error Metrics

One shortcoming of those metrics, however, is that they do not accurately measure how well the predictions were able to capture the trend on a given time series. We can interpret the trend as being the ups and downs of the sequence

To exemplify this issue, let us get back to the BTC forecasting example. Imagine we are given the predictions of a model m, and we want to see whether this model can outperform

the naive and linear model when forecasting the daily price of BTC in the first two weeks of February 2020, using only the January 2020 data. To answer this question, we can start by plotting the two forecasts made in Figure 1.3 and Figure 1.4, the forecasts of Model *m*, and the actual BTC price at that time period. Doing so will result in the following (Figure 1.5).

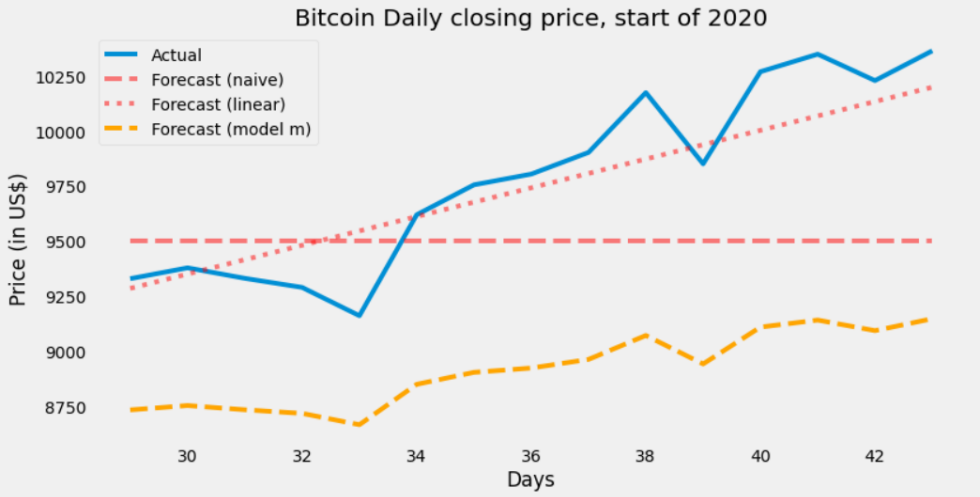

Figure 1.5: Forecast comparison

In the figure, we can see that the naive and linear forecasts are a better approximation of the actual BTC price, while Model *m*'s forecast seems to be off by a couple thousand dollars. Does that mean that the other two models are better? We can calculate the MSE and sMAPE of all three forecasts to answer this. Remember that, when talking about error metrics, the lower the metric the closer the forecast data is to the actual data, implying a better forecast performance.

	Naive	Linear	Model *m*
MSE	498.88	182.56	905.59
sMAPE (%)	4.32	1.478	9.25

Based on these metrics, we can see that the linear model has the best performance.

However, this is a case where these accuracy metrics are misleading, as they only take data point values into account, and not the overall trend of the data.

To understand why this is a problem, let us scale the BTC price and the Model *m*'s prediction between 0 and 1. We do this by dividing each data point by the list's highest value. This will allow us to overlap the two plots and compare their trends (see Figure 1.6).

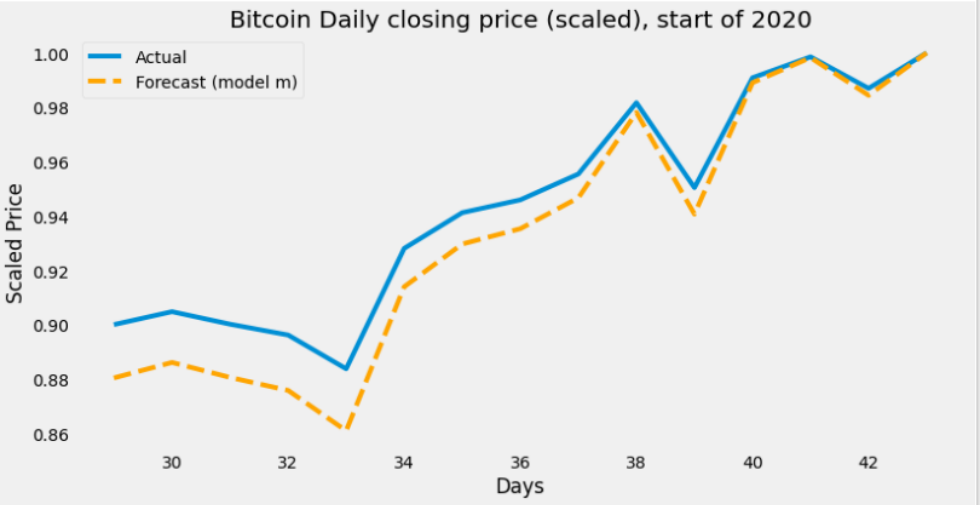

Figure 1.6: Scaled down plots

Now we can clearly see that Model *m*'s predictions are extremely close to the actual BTC price trend. If we were to blindly evaluate each model's performance by solely comparing their error metrics, we would miss out on a model that is great at estimating price trends. This example proves the point that we can't rely solely on graphs or the presented metrics to evaluate the performance of a model.

To evaluate the trend similarity between two sequences, we will introduce a new metric.

1.3.5 Trend Similarity metric

The trend similarity metric works by looking at whether a data point at time t will increase, decrease, or stay constant at time t+1. For each data point in a sequence up to, but not including the last data point, we classify it as either 1, 0, or -1, where 1 indicates that the following data point is higher, 0 indicates that it is constant, and -1 indicates that it is lower. This allows us to convert this regression problem into a classification one.

For example, we can look at the model m's forecast and plot 1, -1, or 0 for each data

point (Figure 1.7).

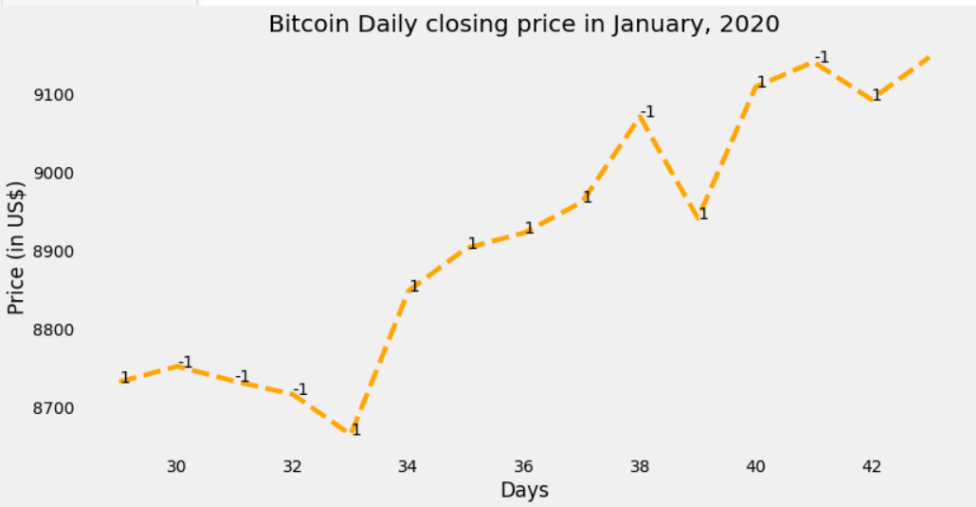

Figure 1.7: Forecast similarity

We can extract each value to obtain the following list: [1, -1, -1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1, 1].

We perform the same process for the actual price sequence of BTC. Once we have both lists, we can obtain the trend similarity of the sequences by using the following formula:

$$\text{Trend similarity} = \frac{1}{n}\sum_{t=1}^{n} I(\hat{y}_i = y_i) \times 100$$

Where n is the total number of data points being analyzed minus 1, and $I(\hat{y}_i = y_i)$ is a function that returns 1 if the current item $\hat{y}_i$ is equal to the true value y_i.

The algorithm for the Trend similarity calculation can be divided into two functions, where the first one converts a sequence to its classification form, and the second takes the actual and forecast sequence as arguments, and uses the first function to calculate the trend similarity of the sequences:

```
def getClassSequence(pred):


  signals = []

  i, j = 0, 1
```

```
    while j < len(pred):
      if pred[j] > pred[i]:
        signals.append(1)
      elif pred[j] < pred[i]:
        signals.append(-1)
      else:
        signals.append(0)
      j += 1
      i += 1

    return signals


def getTrendSimilarity(actual, forecast):

    actual_signals = getClassSequence(actual)
    forecast_signals = getClassSequence(forecast)

    n_correct_classfied = 0
    for i in range(len(actual_signals)):
      if actual_signals[i] == forecast_signals[i]:
        n_correct_classfied += 1

    trend_similarity = (n_correct_classfied/len(actual_signals)) *
       100
    return trend_similarity
```

The trend similarity (TS) is a metric between 0 and 100, where 100 means that the two sequences have the same up, down, or constant trends, and 0 means they have completely

different trends.

We can now compare our 3 models using this new metric to obtain the following results:

	Naive	Linear	Model m
Trend Similarity (%)	0.0	64.28	100.0

With this metric, we can see that the forecast of Model m matches the actual trend perfectly.

While the sMAPE and MSE metrics are useful for telling how close the forecast values are to the actual ones, the trend similarity metric allows us to also see how well the trends match.

1.4 Machine Learning for time series forecasting

Machine Learning (ML) can be defined as a sub-field of computer science that focuses on developing algorithms that have the ability to *learn* without being explicitly programmed [5].

The ability to learn can be defined as follows:

Given a task T and a performance metric P, a computer program is said to learn from experience E if its performance on T, as measured by P, improves with experience E [6].

ML algorithms are extremely useful when we are dealing with data sets that are updated frequently, as they can be developed once and will still be able to remain relevant as the data changes. An example of such datasets is time series, like the daily closing price of a stock.

If we want to predict the overall trend of prices, a simple forecasting model might be able to do so for some short period of time, but as the trends in price change the algorithm's performance drops, as it becomes outdated. An ideal ML model, however, would update itself as new data is obtained, which, in theory, would result in better performance.

While there exist multiple ML algorithms, each one focused on different tasks, for our task of forecasting values in time series data, we can rely on artificial neural networks, one

of the most widely used ML algorithms.

1.4.1 Artificial Neural Networks (ANNs)

In order to properly explain ANNs, we need to understand the structure of an artificial neuron.

Firstly proposed by Warren McCulloh and Walter Pitts in 1947, the artificial neuron is an attempt to mathematically model the structure of a biological neuron. The idea is to represent a neuron as an equation that takes in multiple inputs and outputs a single value.[7] Hence, it can be illustrated as follows (see Figure 1.8).

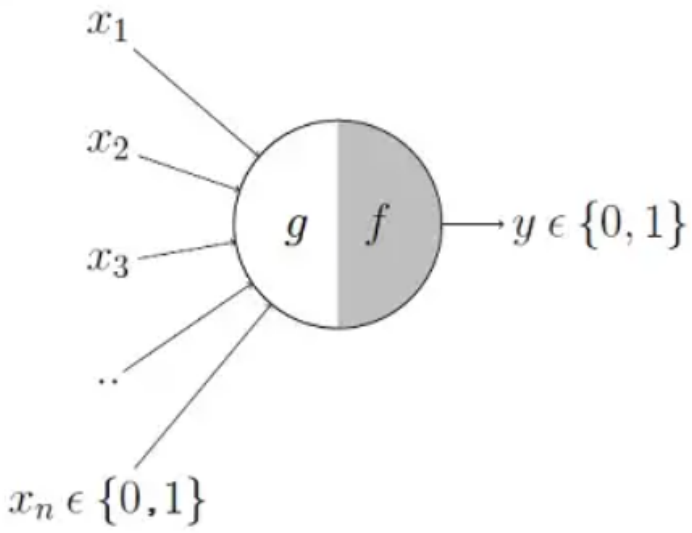

Figure 1.8: Artificial Representation of a neuron [7]

Here, the neuron is represented with an aggregation function *g*, that performs an operation on a set of inputs X and feeds it to function *f*, which finally outputs a value. Although simplistic at first, neurons become more complex as we alter the functions g and f.

A good way to visualize the power of this structure is to understand the architecture of a Perceptron.

Created by Frank Rosenblatt in 1957, the Perceptron is a type of artificial neuron ca-

pable of performing binary classification [8]. It achieves that by associating weights to every input, and by applying a threshold function to the sum of said inputs, as illustrated in Figure 1.10.

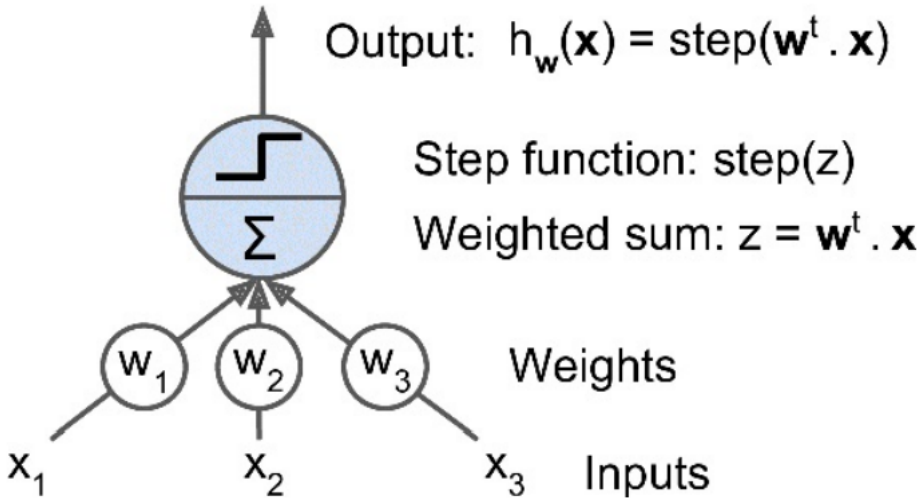

Figure 1.9: Sample representation of a Perceptron [6]

We can see that the aggregation function g is defined as $\sum_{i=1}^{n} x_i w_i$, where n is the number of inputs x_i that the Perceptron takes, and w_i is the weight associated with each input x_i, usually a value between -1 and 1.

Finally, the f is defined as follows:

$$f = 0 \text{ if } z < 0, f = 1 \text{ if } z \geq 0$$

A Perceptron can be trained, for example, to identify if there exists an object on a given picture. If we feed each pixel value from that image as an input x to the Perceptron, it will update its weights until it correctly identifies if a given object is on the picture. If we feed it pixel values of multiple images, its weight values would, theoretically be well adjusted to identify the presence of a said object in any picture.

We can also connect multiple Perceptrons to have a neural network. The main benefit from this comes from the fact that, typically, the more weights a network can tune, and the more operations we perform with the initial set of inputs, the better it can perform at more complex tasks. This, of course, comes at the cost of computer power.

1.4.2 Training a Machine Learning Model

The idea behind training an ML model, like a network of Perceptrons, is to adjust its weights so that it can perform well at a said task. While many algorithms dictate how the weights can be updated, the most widely used, and one of the most effective algorithms to do so is backpropagation [9].

Backpropagation works as follows: For each training instance, the algorithm makes a prediction using the network, measures its error, then goes through each set of neurons in reverse to measure how much each neuron contributed to the error. Finally, it uses this information to adjust the weights associated with each neuron. We measure the error between the predicted and the actual output by using a loss function. The goal of training is to minimize this error, and the more a model gets trained, the lower the error becomes until it converges to a minimum value.

Overfitting

This way of training, however, comes with a nuance: The model being trained may reach the minimum value possible for the loss function, but that does not mean that it will perform well when being used with new data. This happens because the model starts to learn the noise of the data that it was trained on, and this noise is typically intrinsic to the data set, meaning that new data will not have the same noise [6, 5]. When that happens, we have a case of overfitting. In short, whenever our model has better performance on the training data than on other data sets, we can say that the model overfits.

1.4.3 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks are special types of neural networks that can store information when trained. This allows for a more precise estimation of time series, once the network is capable of storing t_{n-j} data points to make predictions after t_n. [10]

This is possible due to a different neuron structure than that of a Perceptron. We can see its structure in Figure 1.10.

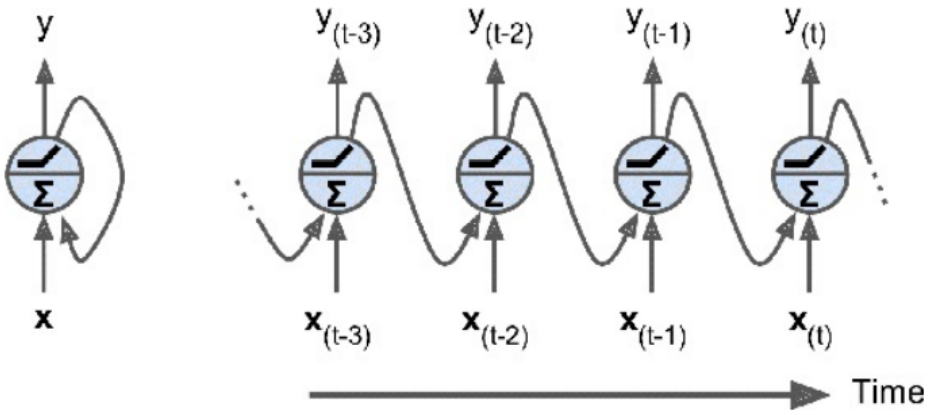

Figure 1.10: Illustration of a Recurrent Neuron [6]

In this figure, we can see that the recurrent neuron (RN) connects to itself through time. This means that an RN at time t receives information from itself at time $t-1$, which is connected to itself at time $t-2$, and so on. Hence, the output of an RN at time t is a function of all inputs from previous time steps.

This multiplicity of connections that happen through an RNN is what gives it the capability of 'memorizing' past information, as we use this information as inputs for the network.

RNNs have a lingering problem, though. As we add more and more time steps to be trained, each neuron becomes a function with too many terms. Most of these terms, however, are not very relevant to getting the desired output. For example, in training an RNN to forecast daily stock prices, the price at $t-100$ is probably not as relevant as the price at $t-1$. One way to solve that is to implement a forget function, that removes connections as time progresses. The problem with this solution, though, is that, if $t-100$ is relevant, the network would not know.

Long Short Term Memory (LSTM) Neural Networks

LSTM Networks are a sub-type of RNNs that can store even more information. They were created by Sepp Hochreiter in 1997 as an attempt to fix the problem with RNNs described above. They do so by altering the structure of a recurrent neuron. Instead of the typical time-connected neurons, an LSTM cell stores information in memory through a memory vector [11]. Its cell is illustrated in Figure 1.11.

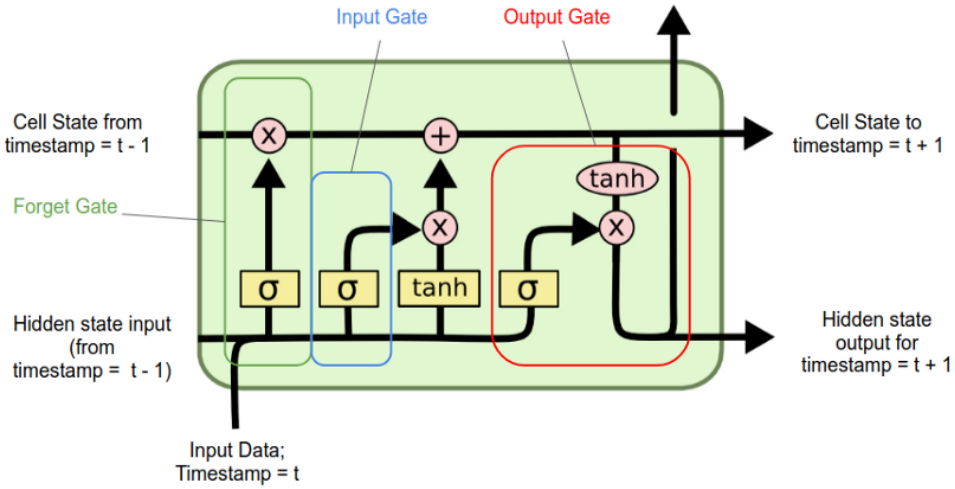

Figure 1.11: Visual representation of a LSTM cell [12]

The cell takes three vectors as arguments:

- The input data at the current timestamp. This is simply the current data point being analyzed by the model.

- A hidden state input from the previous timestamp. The hidden state is also called the output of the LSTM cell, and it holds information about the current state of the LSTM. This can be thought of as the 'short-term memory' part of the model.

- The cell state of the previous timestamp. This state holds information about the previous states of the LSTM. It can be thought of as the 'long-term memory' part of the model.

Inside the cell, these inputs go through multiple operations before being fed into other cells. These operations can be divided into 3 different sections of the cell:

- Forget gate: This section is responsible for determining how much information from the previous states should be considered. It does so by applying the following operation:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

and f_t is the forget gate output at time step t, W_f is the forget gate weight matrix, h_{t-1} is the previous time step's hidden state, x_t is the input at time step t, and b_f is the forget gate bias vector.

By applying a sigmoid to the input and the hidden state, it creates a value between 0 and 1 that will determine the importance of the previous state. The lower the value of the forget term, the more the cell 'forgets'.

- Input gate: Similar to the forget gate, the input gate controls how much information from the input and the previous hidden state should be considered. It also uses the sigmoid function to obtain the value:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

where

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

and i_t is the input gate output at time step t, $\tilde{C}_t$ is the current cell state, W_i is the input gate weight matrix, W_c is the current weight matrix, b_i is the input gate bias vector, and b_c is the current bias vector.

- Output gate: It takes as input the previous output of the memory cell and the current input, and decides what information to output from the cell. It computes the hidden state of the network and decides which of its components should be output. It does so with the following equations:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \cdot \tanh(C_t)$$

where o_t is the output gate output at time step t, h_t is the hidden state output at time step t, C_t is the cell state at time step t, W_o is the output gate weight matrix, and b_o is the output gate bias vector.

1.5 Time frame selection problem

As discussed, machine learning models such as Linear Regression and RNNs rely heavily on data to perform forecasts. Hence, it is important that we only feed relevant data to our model and avoid feeding noisy, non-related data. That way, the model will be better suited to learning useful information about the data.

When we think about forecasting daily values, such as the daily closing price of BTC, it is worth to think whether the intra-day data can be used to improve our overall forecasting accuracy. On one hand, adding more data to the model might help it better pick the trend and seasonality of the data. On the other, adding too much intra-day data might add too much noise, making the model worse at forecasting. Moreover, even if the addition of intra-day data results in an improvement in our accuracy metrics, does that improvement justify the extra computational power and time necessary to process the extra data? This project seeks to explore these issues.

To further understand intra-day data, consider the following images (Figure 1.12, Figure 1.13, Figure 1.14):

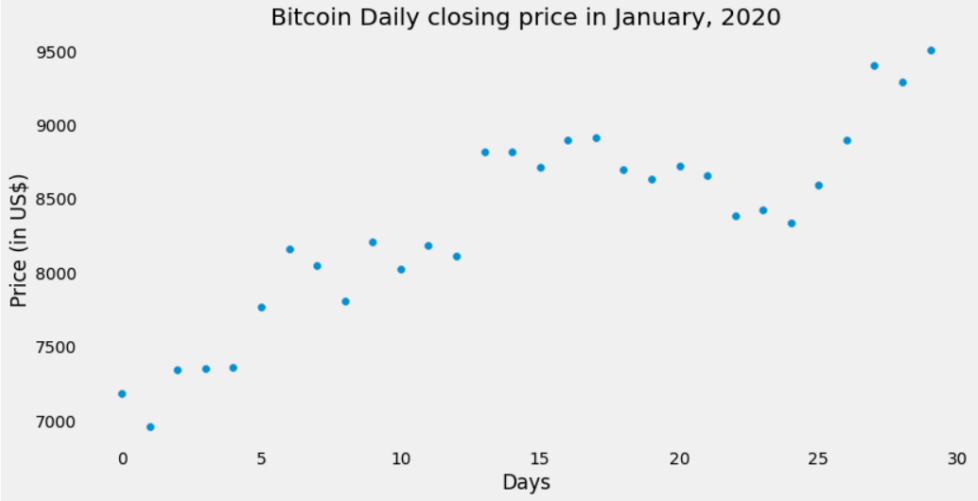

Figure 1.12: BTC daily closing price

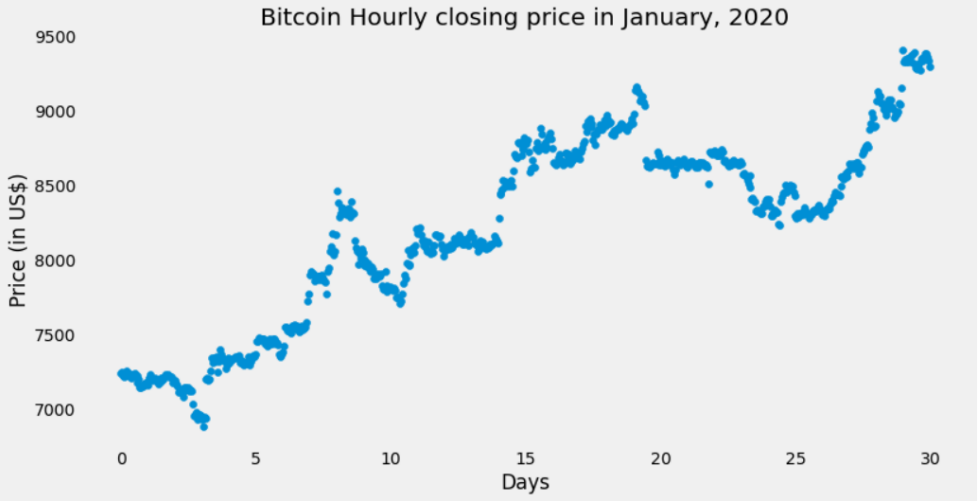

Figure 1.13: BTC hourly closing price

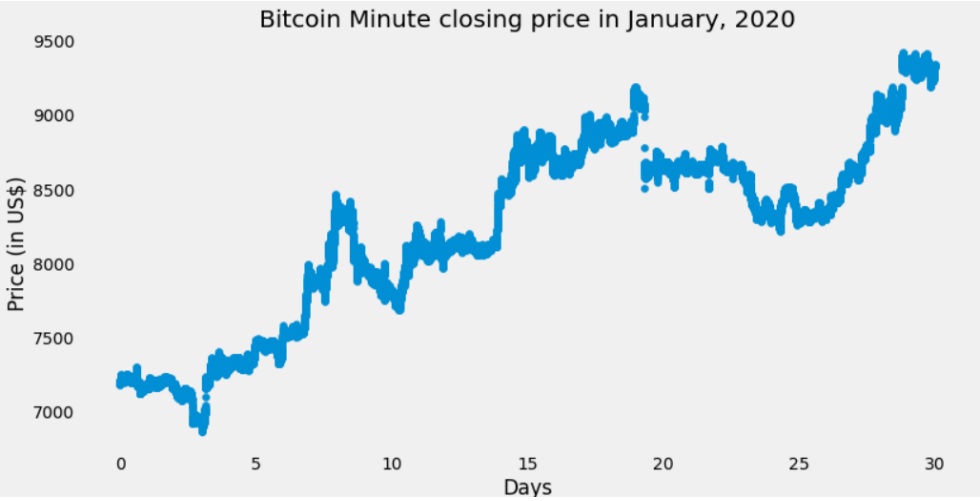

Figure 1.14: BTC minute closing price

As we can see, the lower our time frame, the higher the resolution of our plot. The question that we hope to answer is whether this extra resolution is useful or not when training models for forecasting tasks.

There are two main problems that we may arise when answering this question:

1. The answer might depend heavily on the time series that we are analyzing, hence it will not be an answer that can be generalized.

2. The answer might also be model dependent, meaning that different architectures would produce different results.

Answering this question can be useful for the following reasons:

1. Knowing what timeframe is more efficient can help ease feature selection, as well as improve model accuracy for forecasting.

2. Even if the answer can not be generalized, the final work of this thesis will result in a framework that can be used to determine the optimal timeframe for a given scenario.

1.6 Related Work

Related work has shown that Autoregressive Integrated Moving Average (ARIMA) models have been considered the standard and baseline for forecasting. However, more recent

studies have shown that Long Short-Term Memory (LSTM) models outperform regression-based models like ARIMA [13, 14, 15]. LSTM models have been shown to have robustness to data transformations, as they do not require previous assumptions about the data, such as constant variance, to be generalized to data outside of the training set [15, 16].

Unlike ARIMA, LSTM models do not require the data to be detrended, as they seem to obtain good performance regardless of detrending techniques [17, 18]. This makes LSTM a more robust model than ARIMA.

Previous research has been done to compare monthly, weekly, and daily time frames, which concluded that for low-frequency time frames, the information carried by the high-frequency frame does not provide much help in improving the model's forecasting accuracy. [19, 20] However, as far as we have researched, there has been no previous work comparing minute, hourly, and daily time frames for forecasting.

Furthermore, it is common to utilize LSTM models alongside sentiment analysis, due to their capability of handling natural language processing tasks. Sentiment analysis can provide additional insights into the market's behavior, which can be used to enhance forecasting accuracy.

CHAPTER 2

METHODOLOGY

The main question being investigated is: Does intra-day data help daily price forecasting of time series values for LSTM models? If yes, what is the best time frame to be used: minute or hourly data?

To answer these questions, we will have the following pipeline:

1 - Collect minute-price data of multiple time series.

2 - Perform any necessary cleaning that the data might need.

3 - Format the minute data into hourly and daily data.

4 - Split the daily data into training and validation sets.

5 - Train an LSTM model for each timeframe.

6 - Utilize the models to make forecasts on the validation set.

7 - Answer the question by drawing conclusions based on each model's performance on established accuracy metrics.

2.1 Data Collection

We will be utilizing three main time series data for our experiments:

- Bitcoin minute closing price data ranging between January 1st, 2020, and September 21st, 2021. [21]

- SPY and NQ minute closing price data between January 1st, 2022, and March 9th, 2023. [22]

We opted to restrict our analysis to financial data only due to the difficulties in obtaining minute data from other time series. Thankfully, minute financial data is publicly available on Kaggle.

2.2 Data Wrangling

There were three main tasks needed to get the data into the correct format for the experiment:

1 - Deal with missing information for some minutes (i.e. the data skipped from minute t to minute t + 5.)

2 - Convert the data from minute to hourly and daily data.

3 - Split the data between training and testing sets.

For the first task, we performed padding by adding the missing values using the last known value. While this method can become an issue when there are major gaps in data, it worked well for our datasets, since the gaps present are no larger than 3 minutes. Moreover, the missing data accounts for less than 1% of the whole dataset. Hence, it is safe to say that this method did not introduce any significant bias toward the results.

In the second task, we assume the price value of an hour to be the price value of the last minute at that hour. In other words, we selected all the price values corresponding to minute 59 and made that our hourly dataset. Similarly, for the daily dataset, we assumed the price value for a day to be that of the last hour of that day, so we took apart all the price data corresponding to hour 23 for our daily dataset.

By taking a single minute data point and adding that to our hourly and daily datasets, we assure that we draw the least amount of information possible from the minute data frame. That way we don't introduce any bias when training the models.

Finally, for the third task, we opted for retaining 80% of the data for training and 20% for validating and obtaining results, as multiple articles presented splits ranging between 70% and 30% to 90% and 10% [23, 24, 13, 14], and it allows us to have enough data for the models to learn, while still having a large enough validation set to conclude from. We can better visualize the training test split by having our BTC data as an example:

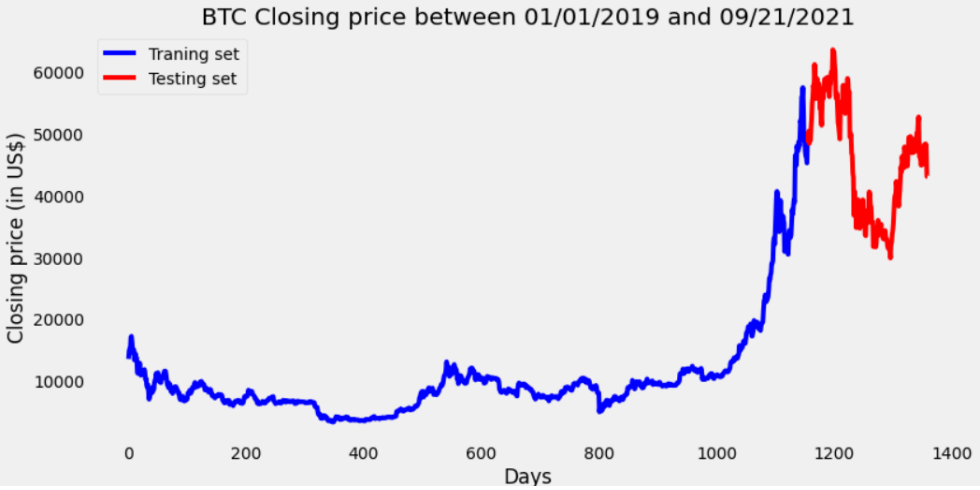

Figure 2.1: BTC training and testing sets

2.3 Data preparation for the LSTM Model

Compared to other models, the LSTM requires unique steps in data preparation to ensure optimal performance.

Sliding Window technique and training frames

The LSTM model works by being trained to predict y-many values from x-many values. In a daily price forecasting context, this means feeding x past days to the network and having it predict the price for the next y days. The values of x and y are parameters to be tuned depending on the specific task at hand. For this project, we decided to set x to 15 and y to 5, which means that the model will take the price of the last 15 days as input and output a forecast of the next 5 days.

Because the model works with an (x,y) training pair, we have to restructure our data into x and y pairs. A simple way of doing that would be to loop through the data, grouping 15 data points and pairing them with the next 5 (Figure 2.2). This figure shows three x,y pairs, where the next pair was created right after the previous one.

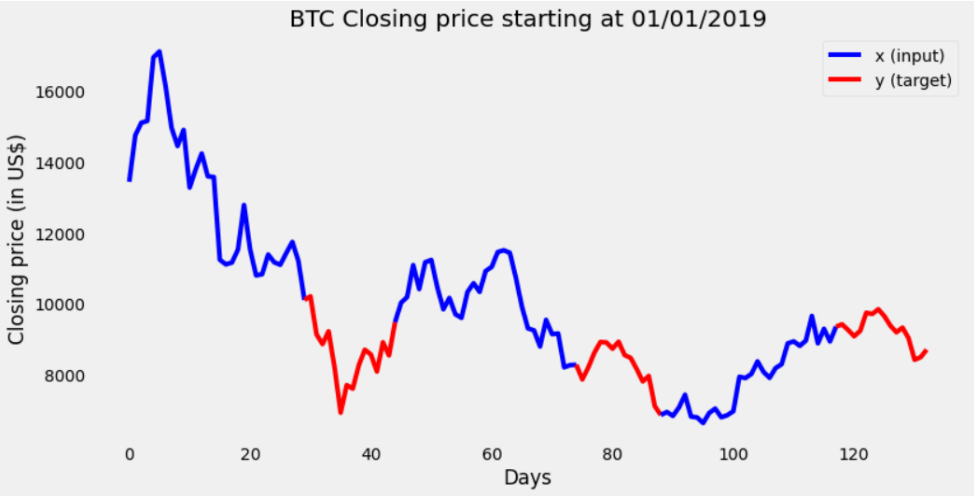

Figure 2.2: x and y split

If we simply split our data like this, however, we will not be utilizing the full information of the data set to train our model, as it will never take the y values as input, nor the x values as targets. To solve this, we need to apply the sliding window technique.

To use this approach, instead of creating the next x, y pair after the previous, we only move one day to create the next pair. We can visualize this in Figure 2.3.

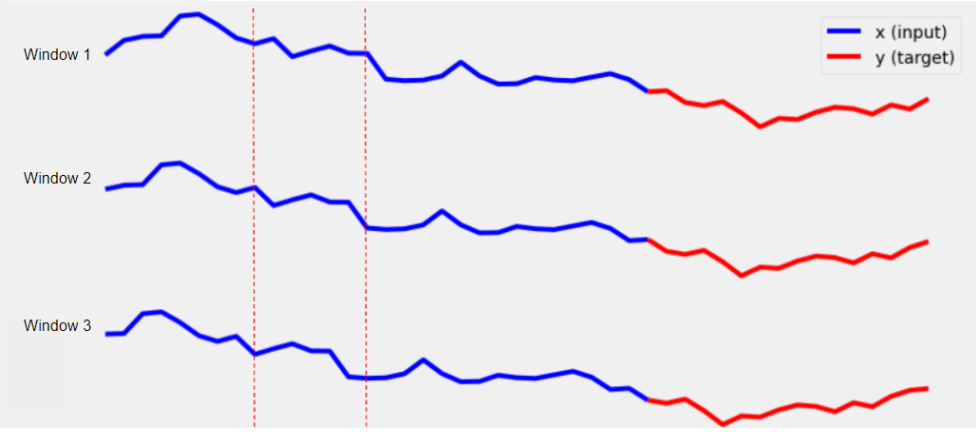

Figure 2.3: Sliding Window demonstration

Here, we can see that on every window, the data moves by one additional data point. Although this approach is significantly more expensive when it comes to memory, it ensures that we utilize all the data points of the data both as targets and as input.

This technique has been widely used by many papers that research LSTM models for forecasting tasks [14, 23, 6, 13]

For our project, because we want to investigate each time frame's performance in predicting future daily data, our y values will be the same regardless of the time frame that the model is trained on.

Scaling the data

Scaling refers to a transformation applied to the data to make it range between 2 values. It can be done so with the following formula:

$$x_{scaled} = (b - a)\frac{x - min(X)}{max(X) - min(X)} + a$$

where x is a data point, a and b are the desired lower and upper range of the scaled data, and min(X), max(X) are the minimum and maximum values of the data set X, respectively. Typically, we scale the data between 0 and 1, or -1 and 1.

There are two main benefits of scaling the data before feeding it to an LSTM model:

1 - We prevent exploding and vanishing gradients [11]. Since the weights of each neuron are derived from the values of the data, having a data set with very large values can result in some neurons having extremely large weights, which hinders convergence into an optimal solution, as the model starts to rely only on these neurons. Similarly, if the data set has very small values, neurons may obtain weights that are close to zero during training, which effectively lowers the size of the network, resulting in slow convergence. These scenarios are called exploding and vanishing gradients, respectively.

2 - We improve the generalization of the model. Scaling the data reduces the effects that outliers may have on training. This helps the model to converge into a more generalizable solution for the problem [6].

We first split the data between training and testing, and then scale the sets independently. This is done to avoid look-ahead bias in the model.

Typically, data is scaled between -1 and 1, or 0 and 1. We tested both scaling ranges and found no significant difference in the model's output. We will utilize the ranges of 0 and 1 for convenience. Since we will be training the model on data scaled between 0 and 1, its outputs will also be scaled. To get the original value prediction, we simply reverse the formula above to obtain the original value x:

$$x = x_{scaled} \times (max(X) - min(X)) + min(X)$$

2.4 Model architecture and parameters

The architecture of a neural network model refers to the specific layout or structure of the network, including the number of layers, the number of neurons in each layer, and how the neurons are interconnected.

Our LSTM model follows a single-layer, univariate, stateful, and multi-step DIRMO architecture, where the number of LSTM neurons is equal to the number of values being predicted. The model is compiled with MSE as the loss function and Adam as the optimizer.

The details of each characteristic, as well as the justification for its choice, are explained below:

2.4.1 Single layer architecture

A single-layer architecture refers to a neural network model with only one layer of neurons between the input and output layers. In this architecture, the input data is fed into a layer of neurons, which process the data and produce an output. Siami-Namini et al. [15] showed that single-layer LSTM architectures were sufficient in obtaining better performance than regression-based models such as ARIMA when performing single-step forecasting. Furthermore, Hum Nath Bhandari et al. [25] conducted experiments that proved that single-layer LSTM models achieve lower RMSE and MAPE values when compared to multi-layer models.

2.4.2 Univariate architecture

A univariate architecture is a neural network architecture designed to work with a single input variable. In other words, the model is designed to process and learn patterns from a single feature or dimension of the input data. In the context of this project, this means that the model will forecast the next values of a time series based solely on its previous values. In contrast to univariate architectures, multivariate architectures are designed to work with multiple input variables or features. These models are commonly used in forecasting tasks where multiple features of the input data can provide valuable information for predicting future values. For instance, a multivariate model can use both the closing prices and the volume traded of a stock to predict its future prices, whereas a univariate architecture would only consider the previous prices of the stock.

Univariate architectures are less memory-consuming than multivariate ones, while also requiring less feature selection and data wrangling. As for its performance, Miller and Kim conducted an extensive experiment comparing the performance of the two architectures in multiple cryptocurrencies. Their finds show that the architectures achieve similar RMSE values when forecasting values of influential cryptocurrencies such as Bitcoin and Ethereum [23]

2.4.3 Stateful LSTM

In an LSTM network, the hidden state and cell state of the network are updated and passed along from one time step to the next. By default, the hidden state and cell state are reset to zero at the start of each input sequence.

Stateful LSTMs are a type of LSTM that allow the hidden state and cell state to be carried over between batches. In other words, the final hidden state and cell state of a batch is used as the initial hidden state and cell state of the next batch. This allows the LSTM to capture longer-term dependencies across multiple sequences in a time series.

Katrompas and Metsis have shown that the stateful LSTMs are better at retaining long-

term information from input data, which results in a better performance in time series forecasting [26].

2.4.4 Multi-step forecasting

A multi-step architecture implies that the model will be used for predicting multiple data points. In the context of time series, this can be exemplified as predicting the closing price of the next five days of a stock.

We will be utilizing the direct multi-step output (DIRMO) strategy [27, 28]

This strategy involves predicting multiple future time steps of a time series directly, without using the predicted values as inputs for subsequent predictions. In other words, the model is trained to directly output a sequence of future values based on the current input, without feeding its predictions back into the model as input for subsequent predictions.

To implement the DIRMO strategy, the neural network model is designed to have multiple output neurons, with each neuron corresponding to a specific time step in the future. The input to the model is typically a window of past observations, and the goal is to predict a sequence of future values for each time step in the output window.

Based on the works of Taieb [28], we concluded that DIRMO appears to offer a good compromise between accuracy and computing efficiency. Strategies with better performance, such as the Multi-Input Multi-Output (MIMO), require one model to be trained for each step being forecast. On the other hand, less expensive strategies such as DirRec [28], utilize the model's previous predictions to output future ones, which causes the accumulation of errors, resulting in worse performance.

2.4.5 MSE as the Loss function

As previously mentioned, the loss function is the function being used to measure the model's performance. The goal of a neural network is to minimize the value of the loss function over the training data, by adjusting the weights and biases of the model during

the training process. We chose the MSE function as it is widely used as a loss function for time series forecasting tasks [15, 17, 6, 26, 14]. Furthermore, the MSE function has an important property called convexity, which means that the model is guaranteed to converge during training.

2.4.6 Adam as the optimizer

An optimizer is an algorithm that is used to update the weights and biases of the model during the training process, with the goal of minimizing the loss function. The optimizer is responsible for computing the gradients of the loss function with respect to the model parameters, and then using these gradients to update the parameters to reduce the value of the loss function.

The Adaptive Moment Estimation (Adam) Algorithm was introduced in 2015 by Kingma and Ba [29] and it has been widely used in multiple neural network models due to it being computationally inexpensive, while also producing faster convergence when compared to other optimization algorithms [30].

2.5 Software and Hardware

2.5.1 Sample code

We built our architecture utilizing the TensorFlow-Keras framework [31] in the Python Language. This allows us to build our models as follows:

```
from keras.layers.core import Dense, LSTM
model = Sequential()
model.add(LSTM(y_range, return_sequences=False,
           stateful=True))
model.compile(optimizer='adam', loss='mse')
```

For training, we utilized the fit function present in the Sequential class:

```
model.fit(x_train, y_train, epochs=5, batch_size=1, shuffle=False)
```

Here, x_train and y_train correspond to the x and y data we train the model on. Epochs correspond to how many times the model is trained on the data. We chose 5 as we observed that the training loss values of all models took at most 5 epochs to stabilize.

Batch size corresponds to how many batches the data will be split on during training. We chose to keep the batch size of one as our architecture is stateful, and a small batch size allows our model to update its parameters after each training example, which can help the model capture long-term dependencies of the data [15]. Finally, we set shuffle to False to ensure that our data is processed sequentially.

All the code developed for this project is available on Github [32].

2.5.2 Hardware

We performed the data wrangling and model training on Google Colab Free GPU environment, with the following specifications:

- Intel(R) Xeon CPU - 2.20GHz.
- 12.7 GB of RAM.
- Tesla T4 GPU, with 12GB of memory and CUDA 12.0.

2.6 Evaluating the performance of different time frames

For a given time series data, we trained three models, one for each time frame (minute, hourly, daily). While each of these models was trained on different data, they were all trained to predict the same daily values.

After training the models, we assess their performances based on the four accuracy metrics presented in Section 1.3: Mean Square Error (MSE), Mean Absolute Error (MAE), symmetric Mean Percent Absolute Error (sMAPE), and Trend Similarity (TS). We then compare the performance of each time frame based on said metrics.

CHAPTER 3

RESULTS

We ran our pipeline for three time series: Bitcoin (BTC), Nasdaq 100 (NQ), and S&P500 (SPY) closing prices. For each time series, we trained one model with the daily, hourly, and minute data respectively. We then utilized the models to make forecasts in our test data set. Because all the models were trained to forecast daily prices, their test set is the same for a given series. Finally, we plotted the forecast and actual values and calculated the MAE, MSE, sMAPE, and Trend Similarity metrics of those values. All the metrics were calculated with the data values scaled between 0 and 1, while the plots have their values scaled back to original market prices.

3.1 SPY forecast results

Model	**MAE**	**MSE**	**sMAPE**	**Trend Similarity**
Daily	0.27	0.31	69.77%	55.17%
Hourly	**0.17**	**0.20**	**33.07%**	**67.79%**
Minute	0.33	0.39	54.59%	56.14%

Table 3.1: SPY metrics

For the SPY index, we can see that the hourly model outperforms the daily and minute models in all the metrics, with MSE and MAE twice lower than the minute model, and about a third lower than the daily model. Furthermore, the hourly model produced a forecast with higher trend similarity than the other two models.

When comparing the daily and minute models, we see a significant difference in the MSE and MAE metrics in favor of the daily model. For the sMAPE, however, the minute

model has a significantly lower value. We believe that this disagreement in the error metrics happens because all the forecasts of the minute model have high values. Since the sMAPE formula has an Actual + Forecast value in the denominator, higher forecast values can result in a lower sMAPE, while the MAE and MSE metrics only take the difference between the values into account.

When looking at the line plots for the models' forecasts, we can observe that the daily model (Figure 3.1) produces predictions with relatively low price variance, ranging from values between $380 and $400. Furthermore, the model does not seem to do well in capturing the trend in the data. On the other hand, in Figure 3.2, we see how the hourly model does a better job at capturing the trend and the price values of the data. Finally, the minute model's predictions (Figure 3.3) flattened out around the 410$ price, suggesting that the model was not able to learn any features of the data.

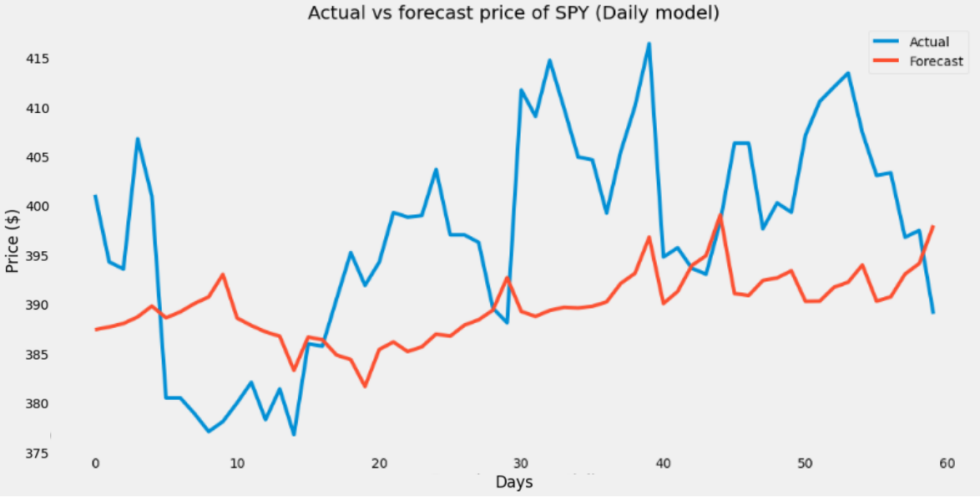

Figure 3.1: SPY daily model forecast

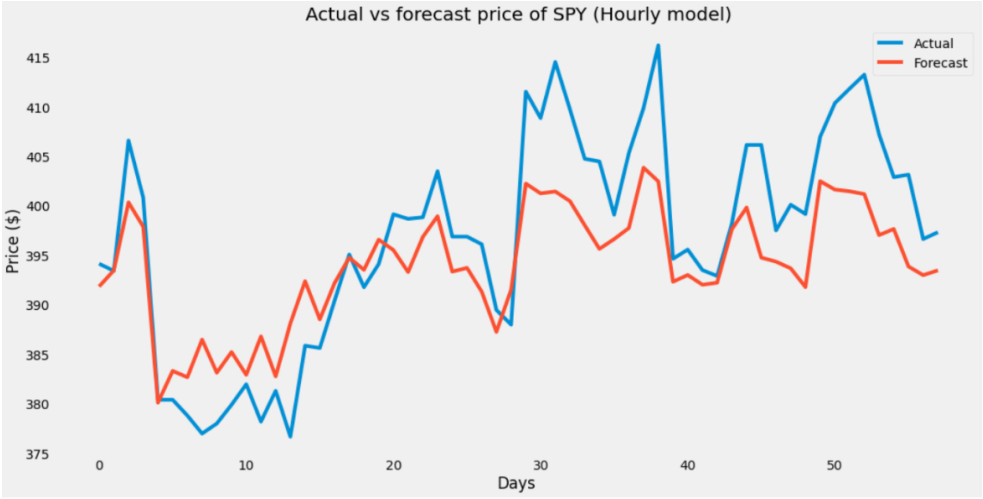

Figure 3.2: SPY hourly model forecast

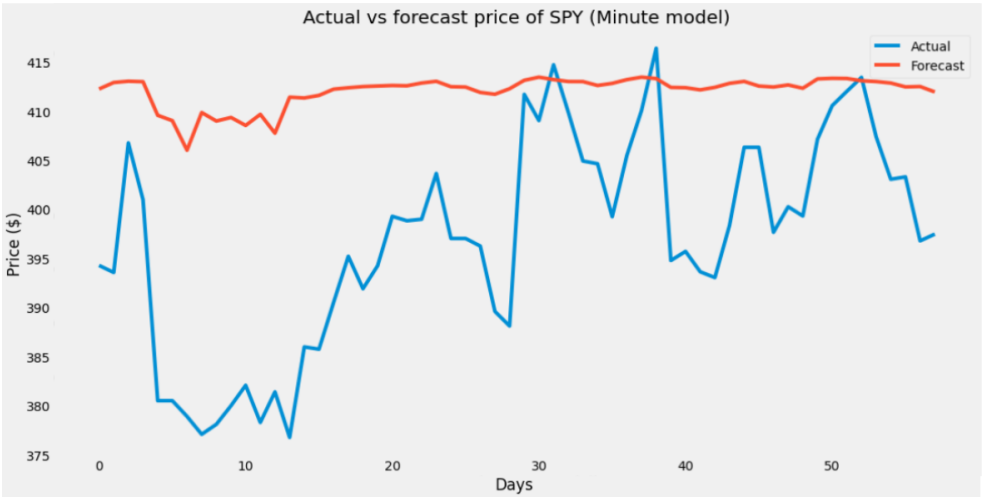

Figure 3.3: SPY minute model forecast

3.2 NQ forecast results

Model	MAE	MSE	sMAPE	Trend Similarity
Daily	0.22	0.26	54.19%	56.89%
Hourly	**0.19**	**0.24**	**42.26%**	**65.55%**
Minute	0.28	0.36	57.31%	48.28%

Table 3.2: NQ metrics

Similarly to what we observed in the previous section, the hourly model for the NQ data outperformed the minute and daily models trained on the same data. Here, however, the MAE and MSE of the daily model are significantly closer to the hourly one. In addition, the daily model has better performance in all metrics compared to the minute one.

We can further examine the difference in performance by looking at their line plots. When comparing the daily and hourly models (Figure 3.4 and Figure 3.5, respectively), we can observe that the daily captures the overall uptrend that started around day 25, but it fails to capture local price movements. Meanwhile, the forecasts of the hourly model look more accurate in the first 25 days, dropping in accuracy afterward. Finally, we can see that the minute model also flattens here (Figure 3.6), similarly to what happened with the SPY minute model.

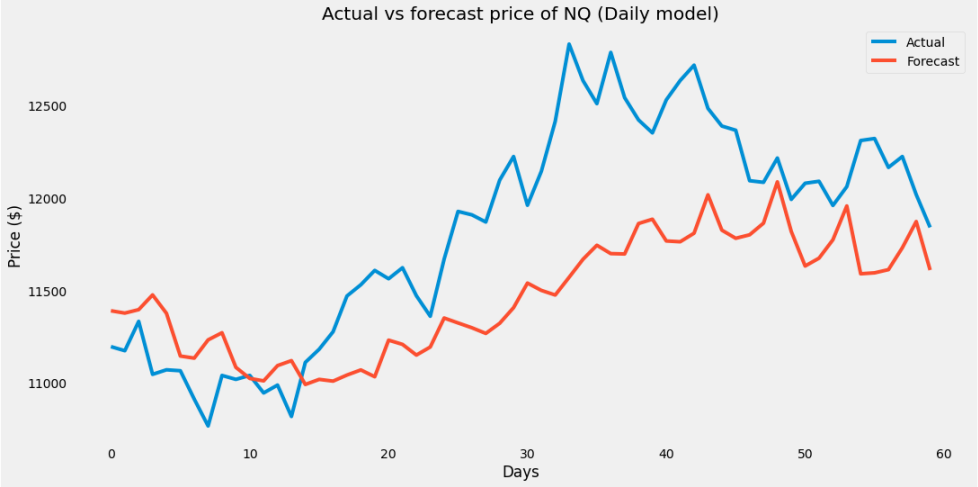

Figure 3.4: NQ daily model forecast

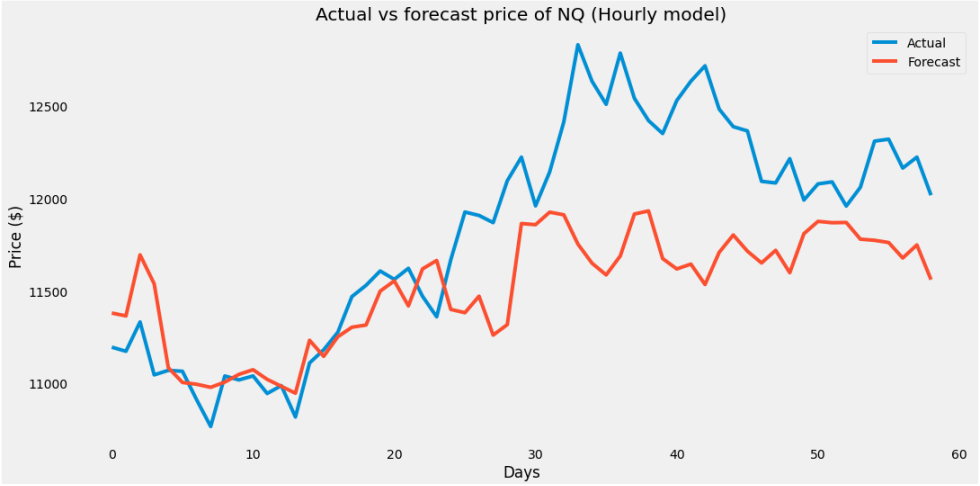

Figure 3.5: NQ hourly model forecast

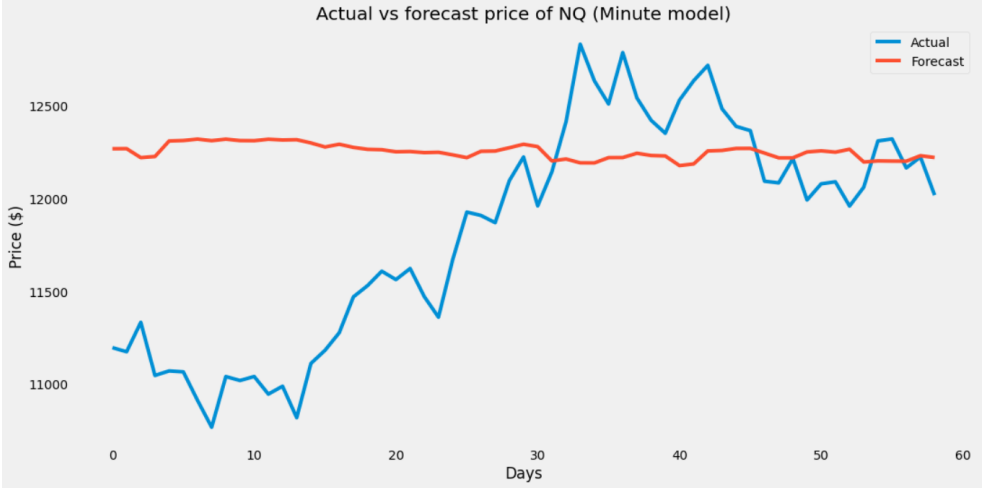

Figure 3.6: NQ minute model forecast

3.3 BTC forecast results

Model	MAE	MSE	sMAPE	Trend Similarity
Daily	**0.18**	**0.21**	**50.30%**	51.66%
Hourly	**0.18**	0.23	56.74%	**72.50%**
Minute	0.32	0.38	53.61%	53.33%

Table 3.3: BTC metrics

By analyzing the MAE and MSE values, we can see a clear similarity in performance between the daily and hourly models, while the minute model has a significantly worse performance in those metrics, producing almost twice as much error.

When we analyze the sMAPE metric, however, we see that the daily model is the best performer, while the hourly model is the worst, and the minute model is the second best. The difference in performance, however, is minimal between the three models.

As for the trend similarity, the hourly model clearly surpasses the other two models, with its forecast trend being 72.5% similar to the original one.

We can further examine each model's performance with a line plot of their respective forecasts.

In Figure 3.7, we can see how the forecast values followed a relatively smooth curve and were able to capture the overall downtrend that started at day 40, as well as the uptrend that followed after day 60. The forecast values, however, are generally higher than the actual ones.

Meanwhile, in Figure 3.8, the model seems to better capture the daily price trend, frequently matching the ups and downs of the actual values. However, the model starts by predicting prices lower than expected, with the difference being even worse after day 60, when the price of BTC quickly rises.

Finally, in Figure 3.9, we can see that the model is not able to learn from the data, as its predictions default to a 'noisy' straight line with no clear correlation with the data.

It is important to note that, for BTC, we exposed our model to significantly more data than with SPY or NQ, as not only our data set was larger for BTC (covering 32 months instead of 16), but the BTC price is also registered continuously, while the prices of SPY and NQ were only registered during market hours (9:30 AM to 4:30 PM, weekdays only).

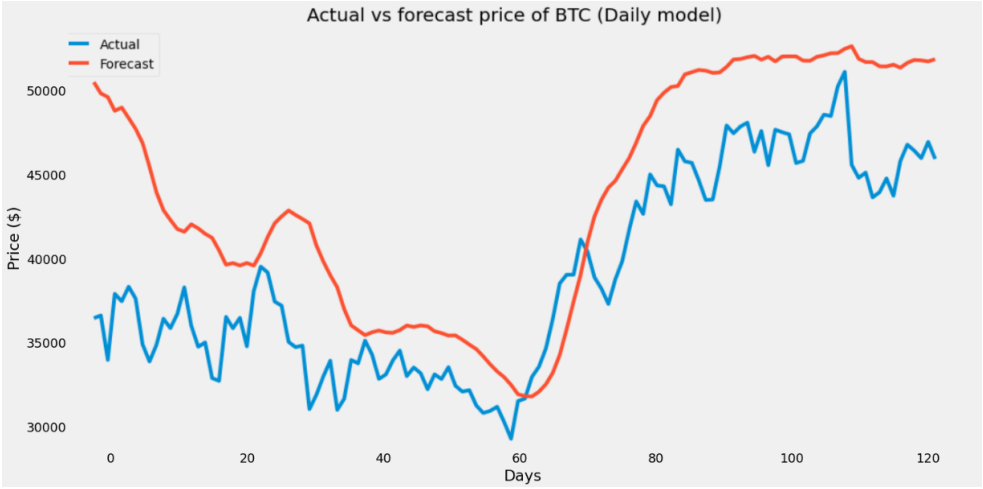

Figure 3.7: BTC daily model forecast

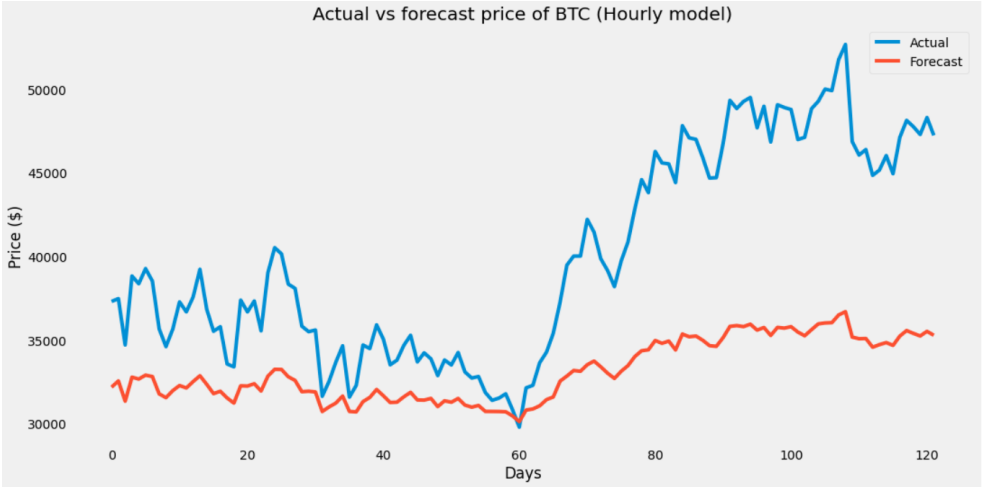

Figure 3.8: BTC hourly model forecast

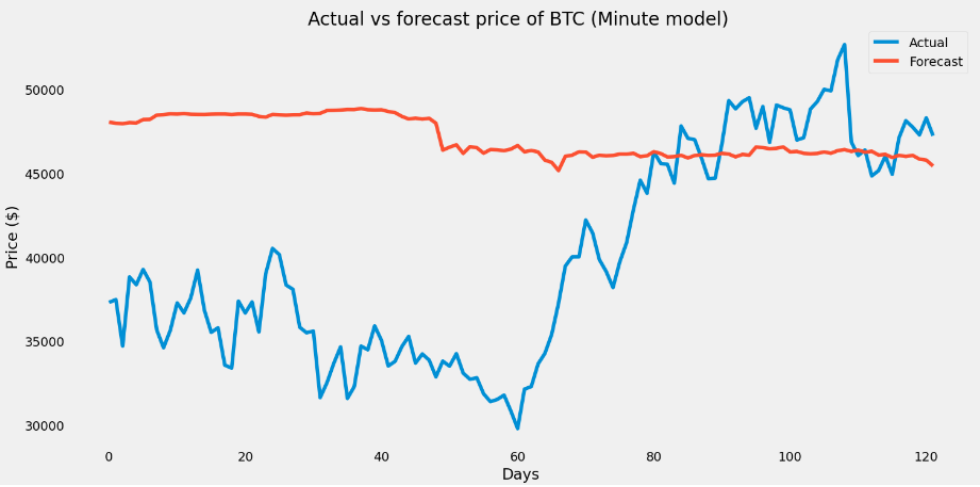

Figure 3.9: BTC minute model forecast

3.4 Feedforward Neural Network Results

To further test the robustness of our framework, we decided to test it with another architecture. For this, we chose to use a simple Feedforward Neural Network (FNN), which consists of stacked perceptrons, introduced in Section 1.4.1. This new architecture will consist of two layers, where the first layer has the number of neurons equal to the input size of the data, and the second layer the has number of neurons equal to the desired number of outputs. For example, when training this architecture on our daily data, we feed it 15 data points to predict the next 5 data points, so the model will have a layer of 15 neurons and another with 5. For our hourly data, we feed it 120 data points (15 days * 8 hours) and have it predict the next 5 data points, so the model has two layers with 120 and 5 neurons, respectively.

Contrary to the LSTM models, these models do not have long or short-term memory capabilities. Running this model on our framework will not only help us test its robustness against different models, but will also help us visualize the impact that the LSTM neuron has on time series forecasting.

The following are the results of this architecture when trained on NQ data:

Model (FNN)	MAE	MSE	sMAPE	Trend Similarity
Daily	**0.23**	**0.29**	**61.23%**	49.15%
Hourly	0.29	0.37	77.32%	**55.93%**
Minute	1.30	1.76	160.42%	50.84%

Table 3.4: NQ - FNN metrics

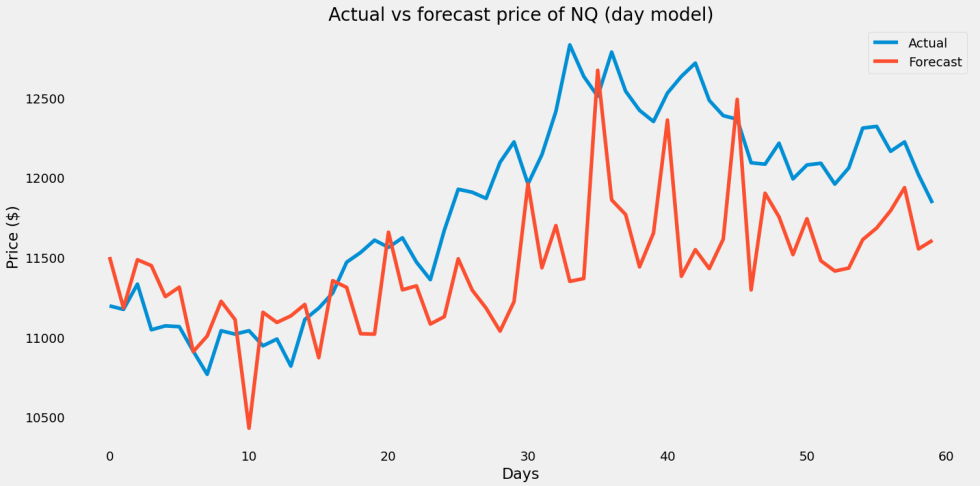

Figure 3.10: NQ daily FNN model forecast

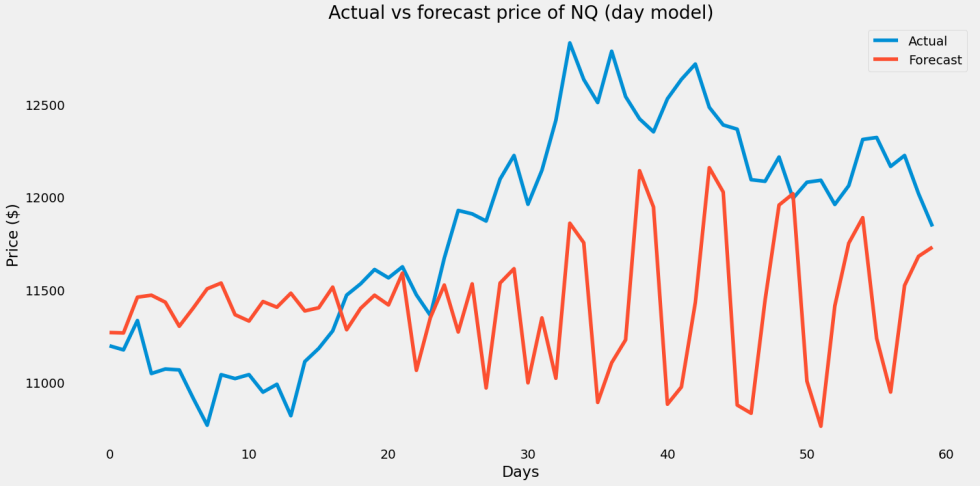

Figure 3.11: NQ hourly FNN model forecast

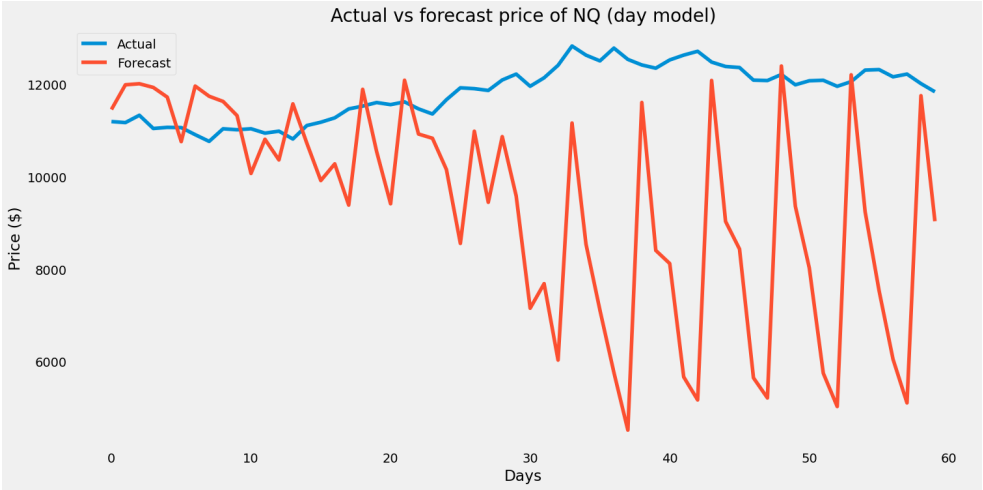

Figure 3.12: NQ minute FNN model forecast

We can clearly see that the performance of the FNN model is significantly worse than that of the LSTM model on all the metrics evaluated. When looking at the forecast figures, it is possible to observe an oscillatory trend every five days, showing that the main shortcoming of this architecture is the inability to successfully perform multi-step forecasting. Furthermore, we can also note a performance drop as we go from the daily data to the hourly and minute data, which suggests that the memory capabilities of the LSTM cell are what allow the architecture to learn underlying trends on intraday data. Nonetheless, this also shows that our framework is able to support other neural network models.

CHAPTER 4
DISCUSSION

In this section, we will discuss the implications of the results and the limitations of this study, as well as potential future work.

4.1 Implications of the results

When looking solely at the metrics, we can see that the models trained on the hourly data have better performance than the other models. This shows that there is value in intra-day data for daily price forecasting of both stocks and cryptocurrencies. The hourly data provides the model with more information on the daily trends of the stock or cryptocurrency, which seems to allow for more accurate predictions. The daily model has the second-best performance, sitting closer to the hourly model than to the minute model. This suggests that using daily data alone can still provide reasonable predictions, but is not as accurate as using intra-day data.

The minute model, however, is not able to capture the trends of the data, resulting in predictions that remain relatively constant through time. We have two hypotheses as to why this is happening. The first one is that the sheer amount of noise introduced by the minute data hinders the model's ability to forecast. This implies that the information present in the minute data is not relevant for daily forecasts.

The second hypothesis is that our architecture is not able to handle the amount of data used. While we have pointed out before that single-layer architecture that employs a low number of neurons seem to be effective in producing good forecast results [15, 25, 24], this has never been investigated for models trained on high-frequency data, such as hourly and minute data. We speculate that a more robust architecture, with a higher number of layers and neurons, could improve the forecast of the models trained on minute data. In either

case, it is clear that the minute model is not suitable for daily price forecasting of stocks and cryptocurrencies when utilizing our proposed LSTM architecture.

Nonetheless, we conclude from our results that LSTM models that follow our architecture, when trained in hourly data, are capable of producing more accurate forecasts than when trained on hourly or minute data. Furthermore, while we investigated only three time series, we believe that this conclusion could be generalized for other cryptocurrencies and stock indexes, as the three time series that were tested (BTC, SPY, NQ) are influential in the trends of most cryptocurrencies and stocks. However, deciding whether we can generalize our results for other time series, such as weather data, would require more testing.

4.2 Limitations of this study

As mentioned before, the results obtained in this are dependent on the LSTM architecture being used. A single-step architecture, for example, might not see much improvement from utilizing intra-day data, as a single-step model are already capable of obtaining extremely low forecasting errors [33, 13]. Furthermore, a multi-layer model could benefit more from the minute data, as architectures that have more neurons could, in theory, find underlying patterns [16, 34] on the minute data, if there are any.

In addition, we trained our models with no more than three years' worth of data, as our BTC dataset was composed of roughly 36 months, while our NQ and SPY data consists of 15 months. Using a larger dataset could be beneficial for our models, as more training can result in more forecasting accuracy, and having more data to train on could result in the models converging to similar accuracy values. We believe that this would be unlikely, however, because similar LSTM architectures have been shown to converge with fewer data points [13, 24].

Finally, our conclusions were based on the results obtained from only three different time series. While we believe that the chosen time series serve as good representations of multiple cryptocurrencies and stock indexes, further testing would need to be done to

confirm this.

4.3 Future Work

Several avenues for future work could build upon the findings of this study. The first one would be to extend the optimal time frame question to other forecasting models, such as ARIMA or Prophet, and see whether the results obtained in our project would translate to regression-based models. It would also be worth investigating whether a different deep learning algorithm, such as convolution neural networks (CNNs), or transformer-based forecasting algorithms, would also benefit from intra-day data as the LSTM does.

We also believe that, to find out why our minute model under-performed, one could utilize a more complex LSTM architecture, and retrain it on the minute data. If the new model is capable of outperforming our current hourly model, it would prove that the issue lies in the LSTM architecture. If, however, the new model is not able to outperform the hourly one, then it would imply that the minute data does not hold valuable information for daily forecasting, and therefore should not be used.

Another avenue worth exploring is that of intra-hourly time frames. While we showed that training on hourly data resulted in better forecasts than training on minute, it can be worth exploring whether a time frame in between, such as a 30-minute time frame, would improve the performance of the model.

Overall, there is still much room for improvement in the field of financial forecasting using deep learning models. Further exploration and experimentation are needed to determine the most effective methods for predicting financial time series data.

CHAPTER 5
CONCLUSION

This study aimed to compare the performance of LSTM models on different time frames, ranging from minutes to hours and days, for predicting the closing price of BTC, NQ, and SPY. To perform this comparison, we built upon previous research in the field to establish a framework for cleaning and restructuring minute data into other time frames, as well as a framework for building, training, and evaluating LSTM models for each time frame.

Our findings suggest that a univariate, multi-step, stateful, single-layer, and LSTM model performs significantly better when trained on hourly data than when trained on minute or daily data. In addition, our results showed that models trained on the minute data were not able to learn patterns of the data, which resulted in poor daily forecasting performance. Whether this is a result of our architecture utilizing a low number of layers and neurons or the possibility that the data present on the minute time frame is simply not relevant for forecasting daily values, is still inconclusive.

In conclusion, this study provides valuable insights into the use of LSTM models in different time frames for forecasting financial time series. The results suggest that the model's performance is highly dependent on the time frame used for prediction. Additionally, the study highlights the importance of choosing the appropriate model architecture. However, further research is needed to explore the model's performance on different types of time series data and the use of multivariate models. Overall, this study provides a foundation for future work in this area and suggests potential directions for further research.

REFERENCES

[1] C. Chatfield, *Time-series forecasting*. Chapman Hall/CRC, 1995.

[2] J. Hyndman Rob, *Forecasting: Principles and practice*. OTEXTS, 2021.

[3] B. G. E. P., G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time Series Analysis: Forecasting and Control*. Wiley, 2016.

[4] J. S. Armstrong, *Long-range forecasting: From Crystal Ball to Computer*. John Wiley Sons, 1985.

[5] T. M. Mitchell, *Machine learning*. McGraw-Hill, 1997.

[6] G. Aurélien, *Hands-on machine learning with scikit-learn, keras and tensorflow: Concepts, tools, and techniques to build Intelligent Systems*. O'Reilly, 2023.

[7] W. S. M. Walter Pitts, "How we know universals: The perception of auditory and visual forms," *Neurocomputing, Volume 1*, pp. 29–42, 1943, doi:10.7551/mitpress/4943.003.0005.

[8] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the theory of Brain Mechanisms*. Spartan Books, 1962.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, doi:10.1038/323533a0.

[10] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990, doi:10.1207/s15516709cog1402_1.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997, doi:10.1162/neco.1997.9.8.1735.

[12] Ryan T. J., *LSTMs explained: A complete, technically accurate, conceptual guide with keras*, https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2, Sep. 2021.

[13] A. Sagheer and M. Kotb, "Time series forecasting of petroleum production using deep lstm recurrent networks," *Neurocomputing*, vol. 323, pp. 203–213, 2019, doi:10.1016/j.neucom.2018.09.082.

[14] P. Verma, S. V. Reddy, L. Ragha, and D. Datta, "Comparison of time-series forecasting models," *2021 International Conference on Intelligent Technologies (CONIT)*, 2021, doi:10.1109/conit51480.2021.9498451.

[15] S. Siami-Namini, N. Tavakoli, and A. Siami Namin, "A comparison of arima and lstm in forecasting time series," *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2018, doi:10.1109/icmla.2018.00227.

[16] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[17] A. Tealab, "Time series forecasting using artificial neural networks methodologies: A systematic review," *Future Computing and Informatics Journal*, vol. 3, no. 2, pp. 334–340, 2018, doi:10.1016/j.fcij.2018.10.003.

[18] A. Ahmed and M. Khalid, "Multi-step ahead wind forecasting using nonlinear autoregressive neural networks," *Energy Procedia*, 2017, doi:10.1016/j.egypro.2017.09.609.

[19] M. T. Owyang and A. B. Galvão, "Forecasting low frequency macroeconomic events with high frequency data," 2020, doi:10.20955/wp.2020.028.

[20] F. Ma, Y. Li, L. Liu, and Y. Zhang, "Are low-frequency data really uninformative? a forecasting combination perspective," *The North American Journal of Economics and Finance*, vol. 44, pp. 92–108, 2018, doi:10.1016/j.najef.2017.11.006.

[21] Zielak, *Bitcoin historical data*, https://www.kaggle.com/datasets/mczielinski/bitcoin-historical-data, Apr. 2021.

[22] D. Barton-Smith, *Intraday market data*, https://www.kaggle.com/datasets/brtnsmth/intraday-market-data, Mar. 2023.

[23] D. Miller and J.-M. Kim, "Univariate and multivariate machine learning forecasting models on the price returns of cryptocurrencies," *Journal of Risk and Financial Management*, vol. 14, no. 10, p. 486, 2021, doi:10.3390/jrfm14100486.

[24] S. S.-N. Siami-Namini, N. Tavakoli, and A. Siami Namin, "The performance of lstm and bilstm in forecasting time series," 2019.

[25] H. N. Bhandari, B. Rimal, N. R. Pokhrel, R. Rimal, K. R. Dahal, and R. K. Khatri, "Predicting stock market index using lstm," *Machine Learning with Applications*, vol. 9, p. 100 320, 2022, doi:https://doi.org/10.1016/j.mlwa.2022.100320.

[26] A. Katrompas and V. Metsis, "Enhancing lstm models with self-attention and stateful training," *Lecture Notes in Networks and Systems*, pp. 217–235, 2021, doi:10.1007/978-3-030-82193-7_14.

[27] S. Ben Taieb, A. Sorjamaa, and G. Bontempi, "Multiple-output modeling for multi-step-ahead time series forecasting," *Neurocomputing*, vol. 73, no. 10-12, pp. 1950–1957, 2010, doi:10.1016/j.neucom.2009.11.030.

[28] S. Ben Taieb, G. Bontempi, A. F. Atiya, and A. Sorjamaa, "A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition," *Expert Systems with Applications*, vol. 39, no. 8, pp. 7067–7083, 2012, doi:10.1016/j.eswa.2012.01.039.

[29] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *[1412.6980v2] Adam: A Method for Stochastic Optimization*, Jan. 2015.

[30] M. Bhandari, P. Parajuli, P. Chapagain, and L. Gaur, "Evaluating performance of adam optimization by proposing energy index," *Communications in Computer and Information Science*, pp. 156–168, 2022, doi:10.1007/978-3-031-07005-1_15.

[31] Keras-Team, *Keras/lstm.py at v2.11.0 · keras-team/keras*, https://github.com/keras-team/keras/blob/v2.11.0/keras/layers/rnn/lstm.py#L382-L893, Sep. 2022.

[32] G. Dutra, *Dutra-apex/timeframe_analysis*, https://github.com/Dutra-Apex/Timeframe_Analysis, 2023.

[33] Y. Hua, "Deep learning with long short-term memory for time series," *arxiv.com*, 2018.

[34] G. Van Houdt, C. Mosquera, and G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5929–5955, 2020, doi:10.1007/s10462-020-09838-1.