**UNIVERSITY AMERICAN COLLEGE SKOPJE**

**SCHOOL OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY**

# Deep Learning for Time Series Forecasting of Historical Stock Prices: A Comparative Study of LSTM, CNN, and Transformer Models

**Graduation Thesis**

**Author:** ***Damjan Dimitriev***

**Supervisor: Prof.** ***Viktor Denkovski, PhD***

***2023***

**UNIVERSITY AMERICAN COLLEGE SKOPJE**

**SCHOOL OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY**

**Academic Year 2022/23**

**GRADUATE of Computer Science and Information Technology**

**Graduation Thesis**

# Deep Learning for Time Series Forecasting of Historical Stock Prices: A Comparative Study of LSTM, CNN, and Transformer Models

**Author:** ***Damjan Dimitriev***

**Supervisor: Prof.** ***Viktor Denkovski, PhD***

**Graduation Thesis Committee:**

1. **Viktor Denkovski, PhD, supervisor**
2. **Marija Stankova Medarovska, PhD, member**

**Date of Graduation Exam: _____**

**Grade suggestion**

- ✓ Pass
- ✓ Pass with credit
- ✓ Pass with merit
- ✓ Pass with distinction

**Awarded _____**

## Acknowledgments

# Table of Contents

## List of Figures

# Abstract

This extensive examination explores time series estimating models, completely surveying their capacity to anticipate past stock costs. It broadly looks at three significant profound learning designs: Long Short-Term Memory (LSTM), Convolutional Neural Network (CNN), and the Transformer model. Through point-by-point code showings and inside and out investigations, this study uncovers the perplexing exhibition of these models when applied to authentic stock cost expectation. It features their assets and limits, giving significant experiences to future examination and progressions in the domain of stock cost gauging. These discoveries offer a clearer comprehension of which models are generally reasonable for explicit datasets and, subsequently, their possible effect on venture systems and monetary navigation.

**Keywords:** Time Series Forecasting, LSTM, Convolutional Neural Network, Transformer Model, Stock Price Prediction.

# Introduction

## *Problem statement*

Stock cost forecast has arisen as a point of convergence of contemporary exploration. Stock costs are helpless to a huge number of impacts, including full scale and microeconomic strategies, corporate lead, financial backer feeling, market elements, unexpected occasions, and the sky is the limit from there. Thus, there is a squeezing interest for exact stock cost gauges among a bunch of financial backers. Various researchers have investigated different scientific and prescient approaches. Strikingly, as of late, computational knowledge applied to back has collected significant consideration, both inside scholarly circles and the monetary area. Seeing the critical weakness and flightiness natural in the protections trade, there is a persuading necessity for a significant learning-based deciding model that considers the idiosyncrasies of monetary backers. (Linlu Mao, 2023).

In the present society, speculation abundance the board has turned into a standard of the contemporary period. Venture abundance the board alludes to the utilization of assets by financial backers to organize reserves sensibly, for instance, investment funds, bank financial items, securities, stocks, ware spots, land, gold, workmanship, and numerous others. Abundance the executives' devices oversee and dole out families, people, ventures, and establishments to accomplish the motivation behind expanding and keeping up with worth to speed up resource development (Wu et al, 2021). Among them, in speculation and financial the board, individuals' #1 result of venture frequently stocks, since the financial exchange enjoys extraordinary benefits and appeal, particularly contrasted and other speculation techniques. More and more researchers have created strategies for expectation from different plots for the securities exchange. As indicated by the component of financial time series and the undertaking of cost expectation, this article proposes another system construction to accomplish a more precise forecast of the stock cost, which joins Convolution Neural Network (CNN) and Long–Short-Term Memory Neural Network (LSTM) (Wu et al, 2021). This new strategy is suitably named stock sequence exhibits convolutional LSTM (SACLSTM). It develops a succession cluster of verifiable information and its proactive factors (choices and prospects) and uses the exhibit as the information picture of the CNN system, and

concentrates specific element vectors through the convolutional layer and the layer of pooling, what's more, as the info vector of LSTM, and takes ten stocks in U.S.A and Taiwan as the exploratory information. Contrasted and past techniques, the forecast exhibition of the proposed calculation in this article prompts improved results when thought about straightforwardly (Wu et al, 2021).

The expectation of financial exchange patterns has been a huge area of exploration because of its true capacity for worthwhile returns and its innately turbulent nature. Different AI models, such as Long Short-Term Memory (LSTM), Convolutional Neural Networks (CNN), and Transformers, have been used to predict these examples.

Stock cost information has the qualities of time series. Simultaneously, in view of AI long transient memory (LSTM) which enjoys the benefits of examining connections among time series information through its memory capability, a strategy for stock cost considering CNN-LSTM. In the meantime, it was used MLP, CNN, RNN, LSTM, CNN-RNN, and other gauging models to anticipate the stock cost individually. Besides, the determining aftereffects of these models are dissected and thought about. The information used in this examination concerns the everyday stock costs from July 1, 1991, to August 31, 2020, including 7127 exchanging days. As far as verifiable information, eight elements were picked, including opening cost, greatest cost, most reduced cost, shutting cost, volume, turnover, promising and less promising times, and change. First and foremost, to take on CNN to productively remove highlights from the information, which are the things of the past 10 days. And afterward, the LSTM was took on to anticipate the stock cost with the extricated highlight information. As indicated by the trial results, the CNN-LSTM can furnish a solid stock cost determining with the most noteworthy forecast exactness. This gauging strategy not just gives another examination thought to stock cost estimating yet additionally gives down to earth insight to researchers to concentrate on monetary time series information (Lu et al, 2020).

The objective was to build a robust and accurate predictive framework that contains a suite of deep learning-based regression models. For the purpose of developing and testing these suggested models, historical data on NIFTY 50 index values over a five-and-a-half-year period was used. For these suggested models, it was selected a very practical forecast horizon value of one week. It was predicted that the deep learning models will be able to estimate the future index values with an extremely high degree of

accuracy and will be able to extract a rich feature set from the historical NIFTY 50 index values. To support this idea, it was previously presented a set of four CNN-based regression models. In the present study, they complement our proposal with five distinct regression models based on deep learning. While two of the proposed models were built on CNN, the remaining three models were based on three variants of LSTM network architecture. (Mehtab, 2020; Sen, 2020)

All these models, nevertheless, come with their own unique set of difficulties. For example, while LSTM models work well with sequential data, they have drawbacks when the sequence length rises, such as the vanishing gradient problem and the loss of long-term relationships. CNN models, on the other hand, have been used for sequence prediction but their effectiveness in financial time series prediction is still under exploration.

In current capital market the cost of a stock is frequently viewed as exceptionally unstable and erratic due to different social, monetary, political and other unique elements. With determined and smart speculation, financial exchange can guarantee an attractive benefit with insignificant capital venture, while mistaken expectation can undoubtedly carry disastrous monetary misfortune to the financial backers. This paper presents the use of an as of late presented AI model — the Transformer model, to anticipate the future cost of supplies of Dhaka Stock Trade (DSE), the main stock trade.

in Bangladesh. The transformer model has been broadly utilized for regular language handling also, PC vision undertakings, yet, apparently, has never been utilized for stock cost forecast task at DSE. As of late the acquaintance of time2vec encoding with address the time series highlights have made it conceivable to utilize the transformer model at the stock cost forecast. This paper focuses on the utilization of transformer-based model to foresee the value development of eight explicit stocks recorded in DSE considering their authentic day to day and week by week information. Our tests show promising outcomes and adequate root mean squared blunder on the greater part of the stocks (Muhammad et al, 2023).

Stock pattern investigation has been a compelling time-series expectation point because of its rewarding and innately turbulent nature. Many models looking to precisely anticipate the pattern of stocks have been founded on Repetitive Brain Organizations (RNNs). Be that as it may, because of the limits of RNNs, like inclination evaporate and long-haul conditions being lost as succession length increments, in this paper it was

fostered that a Transformer based model that utilizes specialized stock information and feeling examination to lead precise stock pattern expectation throughout lengthy time windows. This paper likewise presents a novel dataset containing day to day specialized stock information and top news title information traversing just about three years. Stock forecast dependent exclusively upon specialized information can endure from slack made by the powerlessness of stock pointers really calculate making it known. The utilization of opinion investigation on top titles can help represent unanticipated changes in economic situations brought about by news inclusion. It was measured that the execution of our model against RNNs over succession lengths traversing 5 workdays to 30 work days to copy unique length exchanging procedures. This uncovers an improvement in directional precision over RNNs as grouping length is expanded, with the biggest improvement being near 18.63% at 30 business days (Kaeley et al, 2023).

Transformers, a relatively new model in this domain, leverage self-attention mechanisms for sequence prediction. While they show promise in handling the temporal dependencies of financial data1, their performance in stock market prediction is yet to be thoroughly investigated.

Stock developments expectation is an exceptionally difficult review for examination and industry. Utilizing social media for stock developments expectation is a powerful however troublesome undertaking. Nonetheless, the current forecast techniques which depend via online entertainment for the most part don't think about the rich semantics and connection for a specific stock. It prompts trouble in viable encoding. To take care of this issue, it was proposed a CapTE (Capsule network based on Transformer Encoder) model which utilizes the Transformer Encoder to separate the profound semantic highlights of the online entertainment and afterward catches the primary relationship of the texts through a case organization. In this paper, it was assessed our technique with various benchmarks, and the outcomes exhibit that our technique works on the exhibition of stock developments expectation (Liu et al, 2019).

Therefore, this study aims to conduct a comparative analysis of LSTM, CNN, and Transformer models in predicting stock market trends. Finding each model's advantages and disadvantages in order to decide which one makes the most accurate forecasts was the goal.

## *Outline of the thesis*

In the segment of the outline of the advances and apparatuses, and the cycles to make and think about the models were depicted. The main model to be made and analyzed was the LSTM. While beginning with the making of the LSTM it was required first and foremost to stack and preprocess the data for preparing, approving, and testing the models. After that the models are made independently getting going with the LSTM. After the creation, preparing and testing of the main model the subsequent model or the CNN was made, prepared, and tried. Lastly, the Transformer was made prepared and tried. The information handling was no different for every one of the models to make an in any event, battleground with the goal that the outcomes are exact.

In the completion of the suggestion, it was found that the LSTM model played out the best among the three models, with the CNN model arriving in a close second and the Transformer model waiting behind. This could be attributed to the natural characteristics of LSTM in managing progressive data, and its vital limit long stretch circumstances, which might have been particularly helpful for this specific dataset and task. The CNN, while not usually used for continuous data, has shown promising results, exhibiting its conceivable flexibility. The Transformer model, no matter what its encouraging in various customary language taking care of endeavors, didn't continue too for this present circumstance. In any case, it's imperative to observe that these results are dependent upon numerous factors including the possibility of the data, the specific task, and how the models were ready and tuned. For future work it could be researched different designs or assortments of these models, use greater or more varying datasets, apply different preprocessing, or incorporate extraction strategies, or assessment with different arrangement procedures to perhaps additionally foster execution.

# Overview of Technologies and Tools

For the completion of this thesis and the practical portion, the following tools were used for the creation of the project:

Python: Python is one of the most diverse programming languages and is used from task automation to the field of data science. It is an open source programming language created by Guido van Rossum and was first released in 1991 ( https://www.python.org, 2019).

Jupyter Notebook: Jupyter Notebook is an open-source web application used for creating and sharing documents containing live code, equations, visualizations, and narrative text. It is one of the main ways for training models these days because of it's segmented structure which helps you figure out what segment of code is the problem. Jupyter Notebook was originally developed as the IPython Notebook by Fernando Pérez and has since evolved. The Jupyter Project was officially announced in 2014 (jupyter.org, 2019).

Pandas: Pandas is a Python library that gives information designs and information examination instruments. It is utilized for information control and handling so it very well may be utilized in different ways. Pandas was made by Wes McKinney and was first delivered in 2008 (pandas.pydata.org, 2019).

NumPy: NumPy is a central bundle for logical figuring in Python. It offers help for exhibits and lattices, as well as numerical capabilities to work on these information structures. NumPy is a basic library for information control and mathematical tasks in this venture. Travis Olliphant made NumPy, and it was first delivered in 2006 (numpy.org, 2009).

Matplotlib: Matplotlib is a well known Python library for making static, enlivened, and intelligent representations in Python. It is utilized for information perception in this undertaking. Matplotlib was made by John D. Tracker in 2003 (matplotlib.org, 2012).

Seaborn: Seaborn is an information representation library based on top of Matplotlib. It gives a significant level point of interaction to making educational and appealing measurable illustrations. Seaborn is utilized for improving information perceptions in this undertaking (seaborn.pydata.org, 2012).

Scikit-Learn: Scikit-Learn, or sklearn, is a famous AI library for Python. It gives straightforward and effective instruments to information mining and information investigation. Scikit-Learn was created by David Cournapeau in 2007 (scikit-learn.org/stable, 2019).

# Problem solving

This study applies deep learning models, particularly Convolutional Neural Networks (CNN), Transformers, and Long Short-Term Memory (LSTM), to the problem of accurate stock price prediction in order to advance the field of financial forecasting. The dynamic nature of financial markets and the need for reliable forecasting tools were the driving forces behind this study, especially since traditional and machine learning techniques have shown promise but still have room for development. This study aims to equip finance practitioners and researchers with the knowledge they need to make wise decisions in a market that is constantly changing by contrasting the distinctive strengths and weaknesses of these deep learning models.

## *Theoretical approach*

The theoretical approach for LSTM-based stock price prediction starts with data preprocessing, involving loading historical stock price data for specific stocks like MSFT, AMD, or NVDA. This data was then cleaned to handle any missing values, outliers, or inconsistencies, and relevant features, particularly the 'Close' price, were selected from the dataset. The date information was converted into datetime format and set as the index for time-series data.

Following data preprocessing, the data was segmented into overlapping windows of a specified size, often set to 3 days. This windowing approach helps in creating sequences of historical stock prices. Subsequently, the dataset was split into training, validation, and test sets, allowing for model training, hyperparameter tuning, and performance evaluation.

The creation and training of the LSTM (Long Short-Term Memory) model cemes next. The model architecture was defined, typically including an input layer, an LSTM layer with a specified number of units, one or more dense layers with ReLU activation functions, and an output layer. User-configured hyperparameters, such as the number of LSTM units, learning rate, and training epochs, influence the model's behavior. The model was then trained on the training dataset, adjusting its internal parameters to minimize the Mean Squared Error (MSE) loss function and learn patterns in historical stock price data.

For stock cost expectation, the framework loads authentic information for the picked stock, sets it up for expectation through cleaning and element determination, and utilizations the prepared LSTM model to create forecasts at future stock costs in light of verifiable groupings. The expectation yield gives figures to the characterized forecast time frame.

Performance evaluation follows the prediction step, where the model's accuracy was assessed using validation or test datasets. Common evaluation metrics such as Mean Absolute Error (MAE) and Mean Squared Error (MSE) were calculated. To aid interpretation, the predicted and observed stock prices were visualized to assess how well the model's predictions align with actual market data.

Users can then analyze the results, examining the model's predictions and performance metrics, which serve as the basis for making informed decisions related to stock trading or investment in the dynamic financial market.

## *Computer application description*

For the competition of the project mentioned above, several methods were used as described below.

## LSTM

In Figure 1 the code gave is a Python script that utilizes the Pandas library to peruse and show the items in a CSV record named 'NVDA15.csv.' This CSV record contains authentic stock cost information for NVIDIA Partnership (NVDA) from December 15, 1999, to October 16, 2023. Figuring out down the code and it's result exhaustively.

```
import pandas as pd

df = pd.read_csv(r'data\NVDA15.csv')

df
```

✓ 0.0s

| | Date | Open | High | Low | Close | Adj Close | Volume |
|---|---|---|---|---|---|---|---|
| 0 | 1999-12-15 | 0.769531 | 0.773438 | 0.731771 | 0.750000 | 0.688069 | 20596800 |
| 1 | 1999-12-16 | 0.753906 | 0.765625 | 0.742188 | 0.744792 | 0.683291 | 13257600 |
| 2 | 1999-12-17 | 0.765625 | 0.795573 | 0.761719 | 0.783854 | 0.719128 | 26208000 |
| 3 | 1999-12-20 | 0.785156 | 0.864258 | 0.757813 | 0.848958 | 0.778856 | 40243200 |
| 4 | 1999-12-21 | 0.848958 | 0.886719 | 0.817708 | 0.885417 | 0.812304 | 38836800 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 5992 | 2023-10-10 | 453.100006 | 462.589996 | 450.880005 | 457.980011 | 457.980011 | 36858200 |
| 5993 | 2023-10-11 | 461.959991 | 468.589996 | 460.500000 | 468.059998 | 468.059998 | 37813700 |
| 5994 | 2023-10-12 | 467.769989 | 476.089996 | 463.299988 | 469.450012 | 469.450012 | 48132500 |
| 5995 | 2023-10-13 | 469.600006 | 471.160004 | 452.799988 | 454.609985 | 454.609985 | 47411500 |
| 5996 | 2023-10-16 | 450.630005 | 462.250000 | 449.119995 | 460.950012 | 460.950012 | 37509900 |

5997 rows × 7 columns

Figure 1. Data Importing

This CSV file contains historical stock price data for NVIDIA Corporation (NVDA) from December 15, 1999, to October 16, 2023. Breaking down the code and its output in detail:

1. import pandas as pd: This line imports the Pandas library, which is a popular data manipulation and analysis library in Python. It is often used for handling and processing tabular data, such as CSV files.

2. df = pd.read_csv(r'data\NVDA15.csv'): This line reads the CSV file 'NVDA15.csv' located in a folder named 'data.' The pd.read_csv() function reads the CSV file and stores its contents in a Pandas DataFrame named 'df.' The 'r' before the file path indicates a raw string, which is often used for file paths.

The code essentially loads the CSV file into a DataFrame and assigns it to the variable 'df.' The contents of the 'df' DataFrame are then displayed as the output in your comment. Here's a breakdown of the output:

• The DataFrame contains several columns, each representing different attributes of the stock data. The columns are as follows:

◦ 'Date': This column represents the date when the stock data was recorded.

◦ 'Open': This column represents the opening price of NVDA stock on a given date.

◦ 'High': This column represents the highest price reached by NVDA stock on a given date.

◦ 'Low': This column represents the lowest price reached by NVDA stock on a given date.

◦ 'Close': This column represents the closing price of NVDA stock on a given date.

◦ 'Adj Close': This column represents an adjusted closing price (used for accounting for events like stock splits or dividends).

◦ 'Volume': This column represents the trading volume, indicating the number of shares traded on a given date.

• The DataFrame displays rows of data, with each row corresponding to a specific date in the dataset.

The displayed data includes the stock price information for NVIDIA Corporation over the years, including the opening and closing prices, daily high and low prices, adjusted closing prices, and trading volumes. It is a structured representation of historical stock market data, making it easy to perform various data analysis and manipulation tasks using Pandas.

Here in Figure 2, the code and its resulting yield are represented, which includes activities performed on the Pandas DataFrame 'df,' lodging verifiable stock cost information for NVIDIA Enterprise (NVDA).

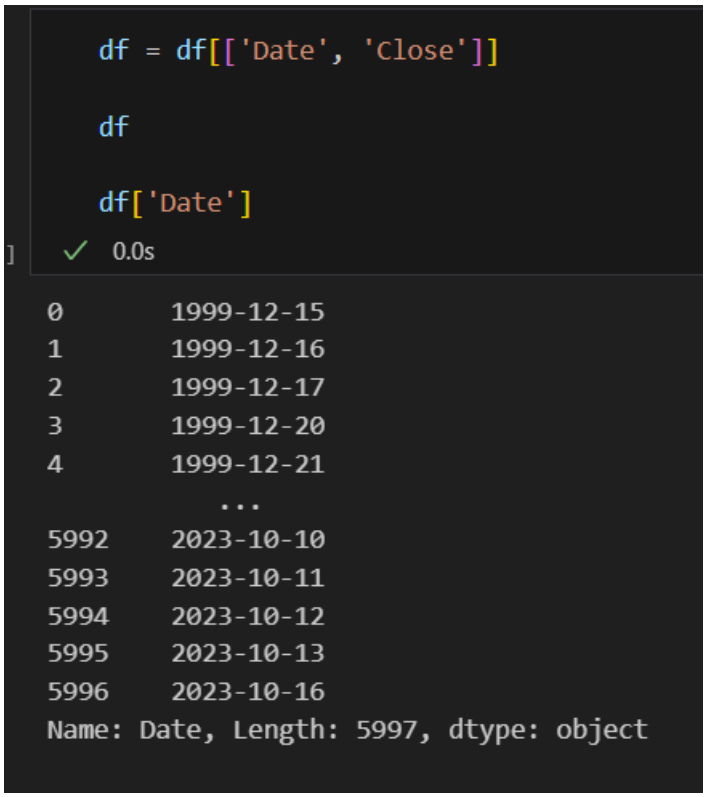



*Figure 2. Formatting Data*

The sequence of actions includes:

df = df[['Date', 'Close']: In this code line, 'df' undergoes a transformation. It is refined to encompass just two columns from the original dataset, specifically 'Date' and 'Close.' 'Date' represents the date information, while 'Close' pertains to the closing price of NVDA stock. The result is a trimmed-down 'df' solely containing these two columns.

df: The subsequent line reveals the adjusted DataFrame, showcasing solely the 'Date' and 'Close' columns. The presented output takes the form of an organized tabular structure with rows and columns.

The ensuing output is reflective of the altered DataFrame, encompassing exclusively the 'Date' and 'Close' columns.

Here, in Figure 2, it is further elaborated on the exhibited output:

The DataFrame now embodies solely two columns: 'Date' and 'Close.'

The 'Date' column is populated with date values, starting from December 15, 1999, and concluding on October 16, 2023. This column fulfills the role of a chronological indicator for each data entry.

The 'Close' column features the closing prices of NVDA stock that correspond to the respective dates. These closing prices are the concluding rates at which NVDA stock was traded on the stock market on each given date.

The provided 'Length' information at the output's conclusion conveys the row count in the DataFrame, signifying a total of 5997 rows.

The refined DataFrame now embraces only the essential date and closing price attributes, rendering it suitable for diverse analytical and visualization tasks concerning NVDA stock pricing. This simplification streamlines the process of working with specific areas of interest within the data, facilitating data processing and analysis tasks.

Figure 3 defines a function called str_to_datetime that converts a date string in the 'YYYY-MM-DD' format into a datetime object.

```
import datetime

def str_to_datetime(s):
  split = s.split('-')
  year, month, day = int(split[0]), int(split[1]), int(split[2])
  return datetime.datetime(year=year, month=month, day=day)

datetime_object = str_to_datetime('1999-12-15')
datetime_object
```

✓ 0.0s

```
datetime.datetime(1999, 12, 15, 0, 0)
```

*Figure 3. DateTime Conversion*

When the function is called with the date string '1999-12-15,' it returns a datetime object representing December 15, 1999, with a time of midnight (00:00:00).

In Figure 4, the provided code segment begins by presenting the DataFrame 'df' in its initial state.

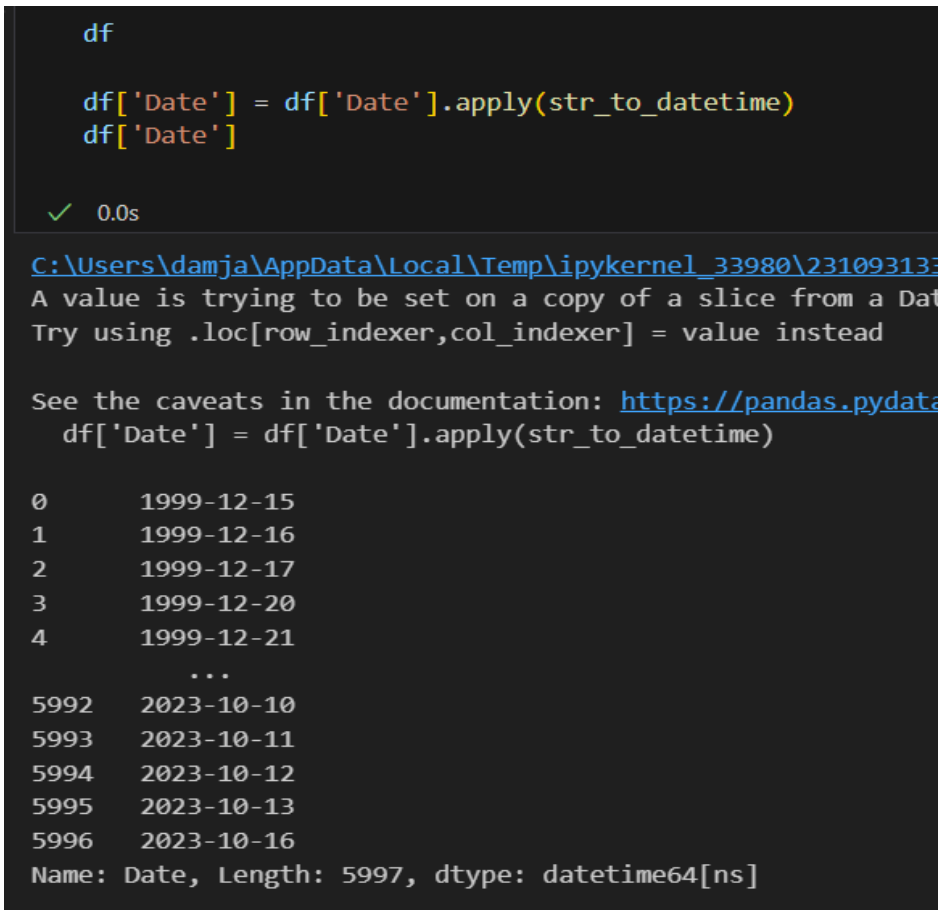



*Figure 4. Data Formatting*

Subsequently, the 'Date' column within this DataFrame undergoes a transformation process utilizing the 'str_to_datetime' function. The result of this transformation is then reassigned to the 'Date' column in the DataFrame. This conversion operation fundamentally alters the data type of the 'Date' column, transitioning it from a string data type to a 'datetime64' data type, representing dates. The outcome of this conversion was then showcased, demonstrating the date values in the 'YYYY-MM-DD' format, with each date now identified as a 'datetime64' data type. Furthermore, it's important to note that during this assignment process, a warning is issued, notifying that the modification of the 'Date' column is performed on a slice of the DataFrame.

Figure 5 presents code where the index of the 'df' DataFrame is modified to align with the values from the 'Date' column, and simultaneously, the 'Date' column is removed from the DataFrame using the 'pop' method.

```
df.index = df.pop('Date'
df
```

✓ 0.0s

| Date | Close |
|---|---|
| 1999-12-15 | 0.750000 |
| 1999-12-16 | 0.744792 |
| 1999-12-17 | 0.783854 |
| 1999-12-20 | 0.848958 |
| 1999-12-21 | 0.885417 |
| ... | ... |
| 2023-10-10 | 457.980011 |
| 2023-10-11 | 468.059998 |
| 2023-10-12 | 469.450012 |
| 2023-10-13 | 454.609985 |
| 2023-10-16 | 460.950012 |

5997 rows × 1 columns

*Figure 5. Date Pop*

Following these operations, the DataFrame is displayed, revealing that the 'Date' column now serves as the new index for the DataFrame. The outcome is a DataFrame where the 'Close' column is intricately associated with the 'Date' index, forming a tabular structure comprising 5997 rows and a single column. In this arrangement, the 'Close' column signifies the closing prices of the stock, while the 'Date' index encapsulates the corresponding dates. This setup establishes a time series structure for the data, facilitating time-based analysis and interpretation of stock close prices over the specified date range.

In Figure 6, this code section incorporates the 'matplotlib' library for data visualization purposes. It leverages the 'plot' function to create a line plot that graphically represents data from the 'df' DataFrame.

```
import matplotlib.pyplot as plt

plt.plot(df.index, df['Close'])
```

✓ 0.2s

```
[<matplotlib.lines.Line2D at 0x1fdc6a89540>]
```

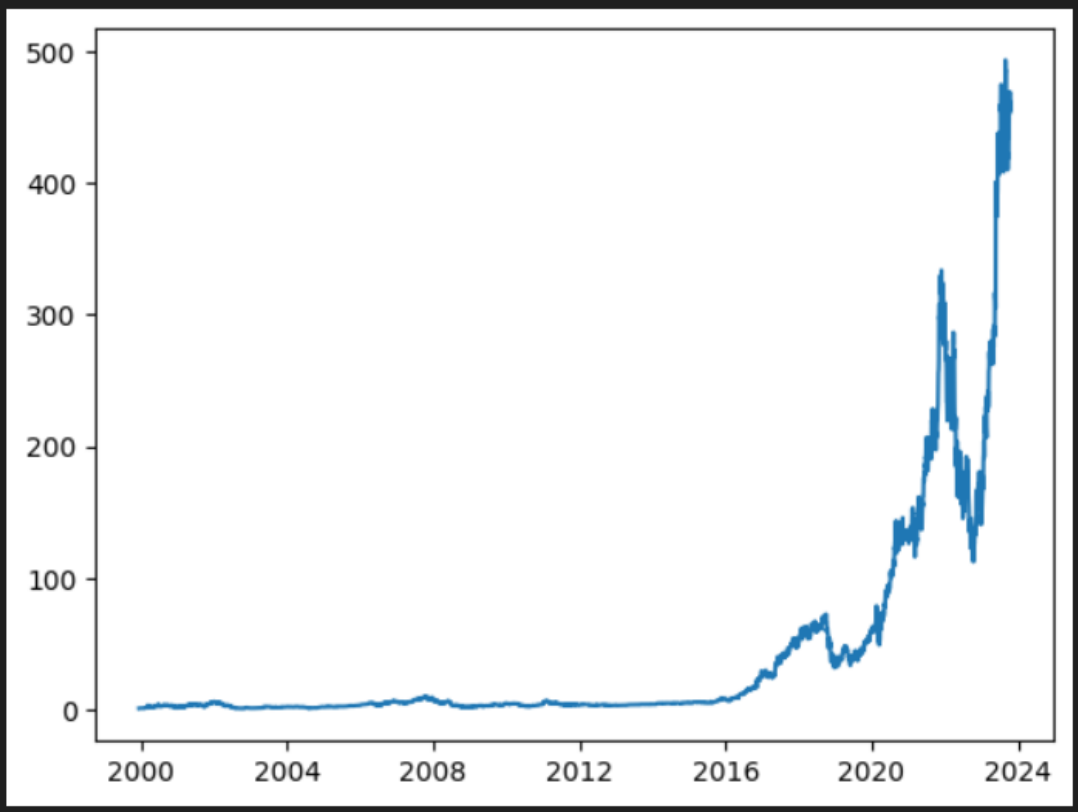



*Figure 6. Plotting*

The plot conveys the relationship between the 'Close' values, which are displayed on the y-axis, and the 'Date' index, presented on the x-axis. The end result is the generation of a line chart, illustrating the progression of the stock's closing prices over time. The x-axis showcases the associated dates, while the y-axis displays the corresponding closing prices, offering a clear and intuitive visualization of the stock's historical performance.

In Figure 7, this code segment defines a function named 'df_to_windowed_df' tailored for transforming a given DataFrame for time series prediction.

```python
import numpy as np

def df_to_windowed_df(dataframe, first_date_str, last_date_str, n=3):
  first_date = str_to_datetime(first_date_str)
  last_date  = str_to_datetime(last_date_str)

  target_date = first_date

  dates = []
  X, Y = [], []

  last_time = False
  while True:
    df_subset = dataframe.loc[:target_date].tail(n+1)

    if len(df_subset) != n+1:
      print(f'Error: Window of size {n} is too large for date {target_date}')
      return

    values = df_subset['Close'].to_numpy()
    x, y = values[:-1], values[-1]

    dates.append(target_date)
    X.append(x)
    Y.append(y)

    next_week = dataframe.loc[target_date:target_date+datetime.timedelta(days=7)]
    next_datetime_str = str(next_week.head(2).tail(1).index.values[0])
    next_date_str = next_datetime_str.split('T')[0]
    year_month_day = next_date_str.split('-')
    year, month, day = year_month_day
    next_date = datetime.datetime(day=int(day), month=int(month), year=int(year))

    if last_time:
      break

    target_date = next_date

    if target_date == last_date:
      last_time = True

  ret_df = pd.DataFrame({})
  ret_df['Target Date'] = dates

  X = np.array(X)
  for i in range(0, n):
    X[:, i]
    ret_df[f'Target-{n-i}'] = X[:, i]

  ret_df['Target'] = Y

  return ret_df
```

*Figure 7. Transforming Data*

The function takes three main parameters: 'dataframe,' 'first_date_str,' and 'last_date_str,' along with an optional 'n' parameter that defaults to 3.

Here's how the code operates:

- It calculates 'first_date' and 'last_date' based on the provided date strings.

- A loop is initiated, with 'target_date' initially set to 'first_date.'
- Three lists, 'dates,' 'X,' and 'Y,' are created to store the target date, input sequences, and corresponding target values, respectively.
- A 'last_time' flag is set to 'False' to control the loop.
- Within the loop:
    - It extracts a subset of the input DataFrame, including rows up to and including the 'target_date,' retaining the last 'n+1' rows. This subset represents a sliding window of data for the current 'target_date.'
    - If the length of this subset isn't equal to 'n+1,' it issues an error message, indicating that the window size is too large for the current date, and exits the function.
    - The 'values' variable is created by converting the 'Close' column of the subset into a NumPy array.
    - 'x' represents the input sequence, composed of the first 'n' values of 'values,' and 'y' represents the target value, which is the last value of 'values.'
    - 'dates,' 'X,' and 'Y' are updated with the current 'target_date,' 'x,' and 'y,' respectively.
    - The function calculates the date for the next week and continues the loop until it reaches the 'last_date.'
    - After the loop, a new DataFrame 'ret_df' is formed to hold the processed data.
    - The 'Target Date' column in 'ret_df' is populated with the 'dates' list.
    - The input sequences in 'X' are converted into a NumPy array, and a loop is utilized to construct columns in 'ret_df,' each representing an input value.
    - The 'Target' column in 'ret_df' is filled with the 'Y' values, representing the target values.
    - The final 'ret_df' is returned, housing the prepared data for time series prediction, including input sequences and their associated target values.

This code segment in Figure 8 first calls the 'df_to_windowed_df' function with specific arguments to create a 'windowed_df' DataFrame, which is structured for time series prediction.

```
# Start day second time around: '2022-10-17'
windowed_df = df_to_windowed_df(df,
                                '2022-10-17',
                                '2023-10-16',
                                n=3)
windowed_df
def windowed_df_to_date_X_y(windowed_dataframe):
  df_as_np = windowed_dataframe.to_numpy()

  dates = df_as_np[:, 0]

  middle_matrix = df_as_np[:, 1:-1]
  X = middle_matrix.reshape((len(dates), middle_matrix.shape[1], 1))

  Y = df_as_np[:, -1]

  return dates, X.astype(np.float32), Y.astype(np.float32)

✓ 0.0s
```

*Figure 8. Training/Validation/Testing Window*

The provided date range for this windowed DataFrame is from '2022-10-17' to '2023-10-16,' and a window size of 3 is specified.

After obtaining the 'windowed_df,' the code defines another function named 'windowed_df_to_date_X_y.' This function takes the 'windowed_dataframe' as input and processes it further:

It converts the 'windowed_dataframe' into a NumPy array named 'df_as_np.'

The dates are extracted from the first column of the 'df_as_np' array and stored in the 'dates' variable.

A middle matrix is created from 'df_as_np' by excluding the first and last columns. This middle matrix represents the input sequences for the time series data.

'X' is generated by reshaping the 'middle_matrix' to have three dimensions, where the first dimension corresponds to the number of dates, the second dimension represents the input sequence length, and the third dimension is set to 1.

'Y' is extracted from 'df_as_np' and represents the target values from the last column of the 'windowed_df' DataFrame.

The function returns 'dates,' 'X' as a NumPy array with a float32 data type, and 'Y' as a NumPy array with a float32 data type, providing the processed data suitable for training a time series prediction model.

This code segment in Figure 9 begins by calling the 'windowed_df_to_date_X_y' function with the 'windowed_df' DataFrame as input.

```
# Start day second time around: '2022-10-17'
windowed_df = df_to_windowed_df(df,
                                '2022-10-17',
                                '2023-10-16',
                                n=3)
windowed_df
def windowed_df_to_date_X_y(windowed_dataframe):
  df_as_np = windowed_dataframe.to_numpy()

  dates = df_as_np[:, 0]

  middle_matrix = df_as_np[:, 1:-1]
  X = middle_matrix.reshape((len(dates), middle_matrix.shape[1], 1))

  Y = df_as_np[:, -1]

  return dates, X.astype(np.float32), Y.astype(np.float32)

✓ 0.0s
```

*Figure 9. X & Y Pre-processing*

This function processes the 'windowed_df' and returns three arrays:

'dates': An array containing dates extracted from the 'windowed_df' to represent the timeline of the dataset.

'X': A three-dimensional array that holds the input data for time series prediction. The first dimension corresponds to the number of dates (251 in this case), the second dimension represents the input sequence length (3 in this case), and the third dimension is set to 1, indicating that the input data is univariate.

'y': An array containing target values for the time series prediction.

The code then further examines the shapes of these arrays using the 'shape' attribute and displays their dimensions. The output indicates that 'dates' has a shape of (251,), 'X' has a shape of (251, 3, 1), and 'y' has a shape of (251). These shapes provide insights into the structure of the processed data, which is essential for building and training a time series prediction model.

In the code segment in Figure 10, the data was split into training, validation, and test sets.

```
q_80 = int(len(dates) * .8)
q_90 = int(len(dates) * .9)

dates_train, X_train, y_train = dates[:q_80], X[:q_80], y[:q_80]

dates_val, X_val, y_val = dates[q_80:q_90], X[q_80:q_90], y[q_80:q_90]
dates_test, X_test, y_test = dates[q_90:], X[q_90:], y[q_90:]
```
✓ 0.0s

*Figure 10. Segmenting Data*

'q_80' is calculated as 80% of the total number of dates, and 'q_90' is calculated as 90% of the total number of dates, effectively determining where to split the data.

The training set is created by slicing the 'dates,' 'X,' and 'y' arrays from the beginning up to 'q_80.' This represents the first 80% of the data, which will be used to train the model.

The validation set is created by slicing the data from 'q_80' to 'q_90.' This portion, comprising 10% of the data, will be used to validate the model during training.

The test set is created by slicing the data from 'q_90' to the end. This portion, also 10% of the data, will be used to evaluate the model's performance after training.

These splits were essential for evaluating the model's performance and ensuring that it generalizes well to unseen data.

This code in Figure 11 generates a plot to visualize the stock price data for the training, validation, and test sets.

```
plt.plot(dates_train, y_train)
plt.plot(dates_val, y_val)
plt.plot(dates_test, y_test)

plt.legend(['Train', 'Validation', 'Test'])
```

✓ 0.2s

```
<matplotlib.legend.Legend at 0x1fdc6a899c0>
```

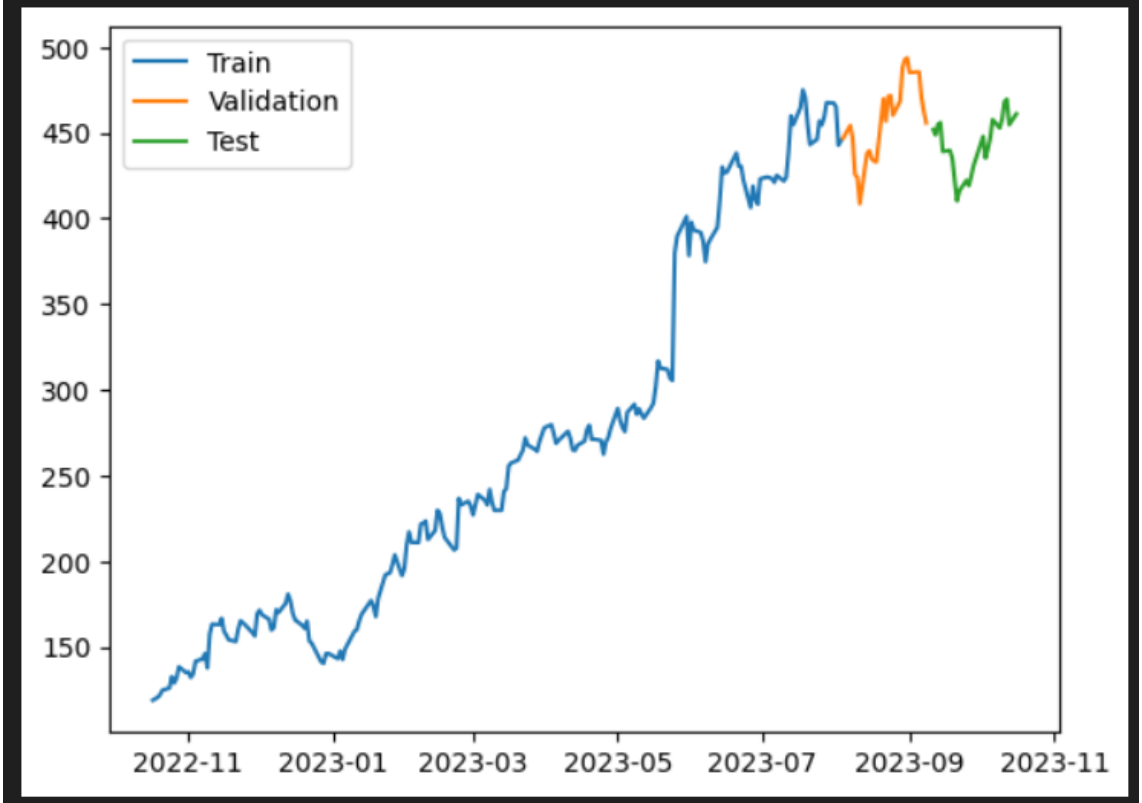



*Figure 11. Visualization of Segments*

plt.plot(dates_train, y_train): This line plots the stock prices (y-values) from the training set against their corresponding dates (x-values). It shows how the stock prices change over time in the training data.

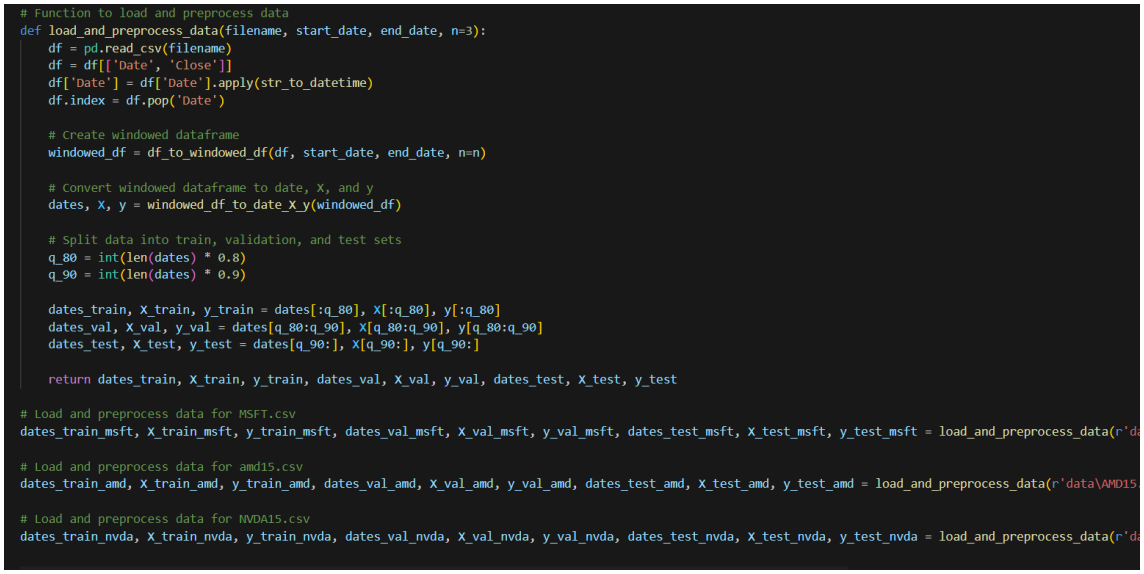
plt.plot(dates_val, y_val): This line plots the stock prices from the validation set in a similar manner, illustrating how they vary over the validation period.

plt.plot(dates_test, y_test): Here, the stock prices from the test set are plotted, displaying their fluctuations during the test period.

plt.legend(['Train', 'Validation', 'Test']): A legend is added to the plot to label the three lines. It indicates that the first line represents the training set, the second is for the

validation set, and the third represents the test set. This makes it clear which dataset each line corresponds to in the plot.

This code in Figure 12 defines a function and then uses it to load and preprocess stock price data from different CSV files for three different companies (Microsoft, AMD, and NVIDIA).



```
# Function to load and preprocess data
def load_and_preprocess_data(filename, start_date, end_date, n=3):
    df = pd.read_csv(filename)
    df = df[['Date', 'Close']]
    df['Date'] = df['Date'].apply(str_to_datetime)
    df.index = df.pop('Date')

    # Create windowed dataframe
    windowed_df = df_to_windowed_df(df, start_date, end_date, n=n)

    # Convert windowed dataframe to date, X, and y
    dates, X, y = windowed_df_to_date_X_y(windowed_df)

    # Split data into train, validation, and test sets
    q_80 = int(len(dates) * 0.8)
    q_90 = int(len(dates) * 0.9)

    dates_train, X_train, y_train = dates[:q_80], X[:q_80], y[:q_80]
    dates_val, X_val, y_val = dates[q_80:q_90], X[q_80:q_90], y[q_80:q_90]
    dates_test, X_test, y_test = dates[q_90:], X[q_90:], y[q_90:]

    return dates_train, X_train, y_train, dates_val, X_val, y_val, dates_test, X_test, y_test

# Load and preprocess data for MSFT.csv
dates_train_msft, X_train_msft, y_train_msft, dates_val_msft, X_val_msft, y_val_msft, dates_test_msft, X_test_msft, y_test_msft = load_and_preprocess_data(r'da

# Load and preprocess data for amd15.csv
dates_train_amd, X_train_amd, y_train_amd, dates_val_amd, X_val_amd, y_val_amd, dates_test_amd, X_test_amd, y_test_amd = load_and_preprocess_data(r'data\AMD15

# Load and preprocess data for NVDA15.csv
dates_train_nvda, X_train_nvda, y_train_nvda, dates_val_nvda, X_val_nvda, y_val_nvda, dates_test_nvda, X_test_nvda, y_test_nvda = load_and_preprocess_data(r'da
```

*Figure 12. Load and Preprocess*

def load_and_preprocess_data(filename, start_date, end_date, n=3): This function takes a CSV file's filename, start date, end date, and an optional window size 'n' as input. It performs the following steps:

Reads the CSV file into a Pandas DataFrame.

Selects and keeps only the 'Date' and 'Close' columns.

Converts the 'Date' column to a datetime format using the str_to_datetime function.

Sets the 'Date' column as the DataFrame's index.

Calls the df_to_windowed_df function to create a windowed DataFrame within the specified date range.

Calls the windowed_df_to_date_X_y function to convert the windowed DataFrame into date, X (input features), and y (output target) arrays.

Splits the data into training, validation, and test sets based on the specified date ranges and percentage splits.

Returns the date, X, and y arrays for the training, validation, and test sets.

The function is then used to load and preprocess data for three different companies: Microsoft (MSFT), AMD, and NVIDIA (NVDA). Each dataset is loaded from a specific CSV file ('MSFT15.csv', 'AMD15.csv', 'NVDA15.csv') within the date range from '2022-10-17' to '2023-10-16', with a window size of 3.

For each company, the resulting arrays (dates_train, X_train, y_train, dates_val, X_val, y_val, dates_test, X_test, y_test) are assigned to separate variables, effectively providing preprocessed data for further analysis or modeling. This allows for consistency in handling the data from multiple sources.

This code represented in Figure 13 defines a function to create and train a neural network model using the Keras library for deep learning:

```
from keras.models import Sequential
from keras.optimizers import Adam
from keras import layers

# Function to create and train the model
def create_and_train_model(X_train, y_train, X_val, y_val, epochs=200):
    model = Sequential([
        layers.Input((3, 1)),
        layers.LSTM(64),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(1)
    ])

    model.compile(loss='mse', optimizer=Adam(learning_rate=0.001), metrics=['mean_absolute_error'])

    model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=epochs)

    return model
✓ 0.0s
```

*Figure 13. LSTM Model*

from keras.models import Sequential: It imports the Sequential model from Keras, which is a linear stack of layers.

from keras.optimizers import Adam: It imports the Adam optimizer, a popular optimization algorithm used in training neural networks.

from keras import layers: It imports the layers module from Keras, which includes various types of layers for building neural networks.

def create_and_train_model(X_train, y_train, X_val, y_val, epochs=200): This function takes input features (X_train and X_val), target values (y_train and y_val), and an optional number of training epochs (default is 200).

Inside the function:

A neural network model is created using the Sequential model. It consists of several layers:

An input layer with a shape of (3, 1), which indicates that the input data has a shape of (3, 1).

An LSTM layer with 64 units, which is a type of recurrent layer used for sequence data.

Two fully connected (Dense) layers with 32 units each, both using the ReLU activation function.

A final output layer with 1 unit, which is a regression task with a single output.

The model is compiled with the mean squared error (MSE) loss function, the Adam optimizer with a learning rate of 0.001, and the mean absolute error (MAE) as a metric.

The model is trained using the fit method on the training data (X_train, y_train) with validation data (X_val, y_val) for a specified number of training epochs.

Once training is complete, the trained model is returned.

This function encapsulates the process of creating a specific neural network architecture and training it with the provided data. It is designed for regression tasks where the goal is to predict a continuous output.

In Figure 14, three machine learning models are created and trained on different datasets: MSFT.csv, amd15.csv, and NVDA15.csv.

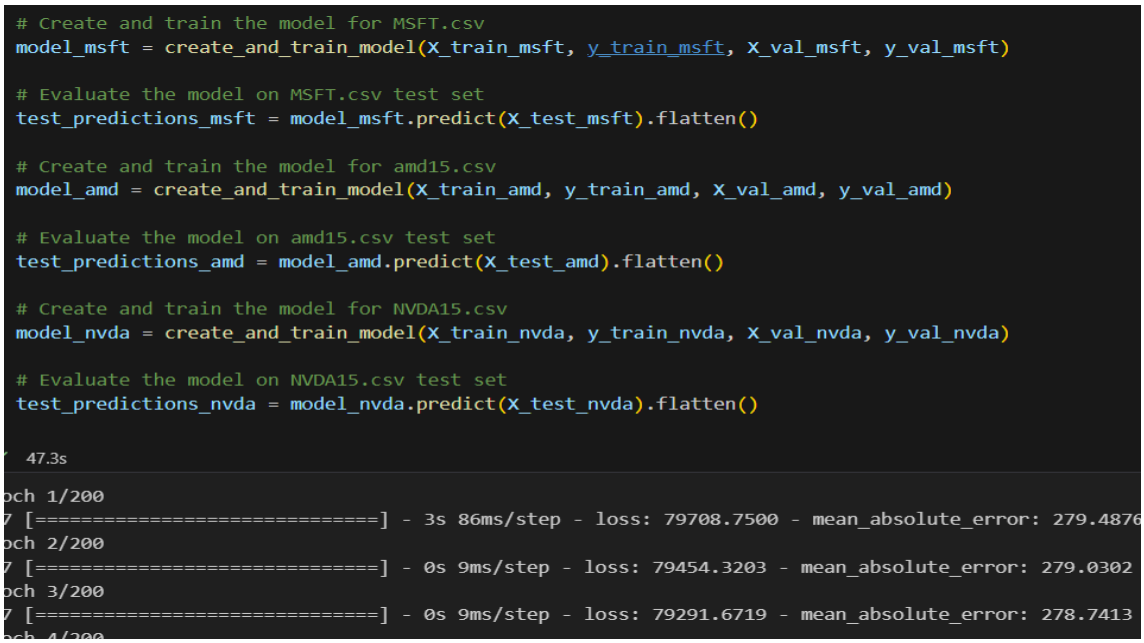



*Figure 14. Training for every Stock*

These models are trained using training data (X_train and y_train) and validated using validation data (X_val and y_val). After training, the models are used to make predictions on a test set (X_test) specific to each dataset, and the predictions are flattened to obtain a one-dimensional array. The process is repeated for each dataset, resulting in test predictions for Microsoft (MSFT), AMD, and NVIDIA (NVDA) stocks.

In the code in Figures 15, 16 & 17, three sets of results were being plotted for different datasets: MSFT.csv, amd15.csv, and NVDA15.csv.

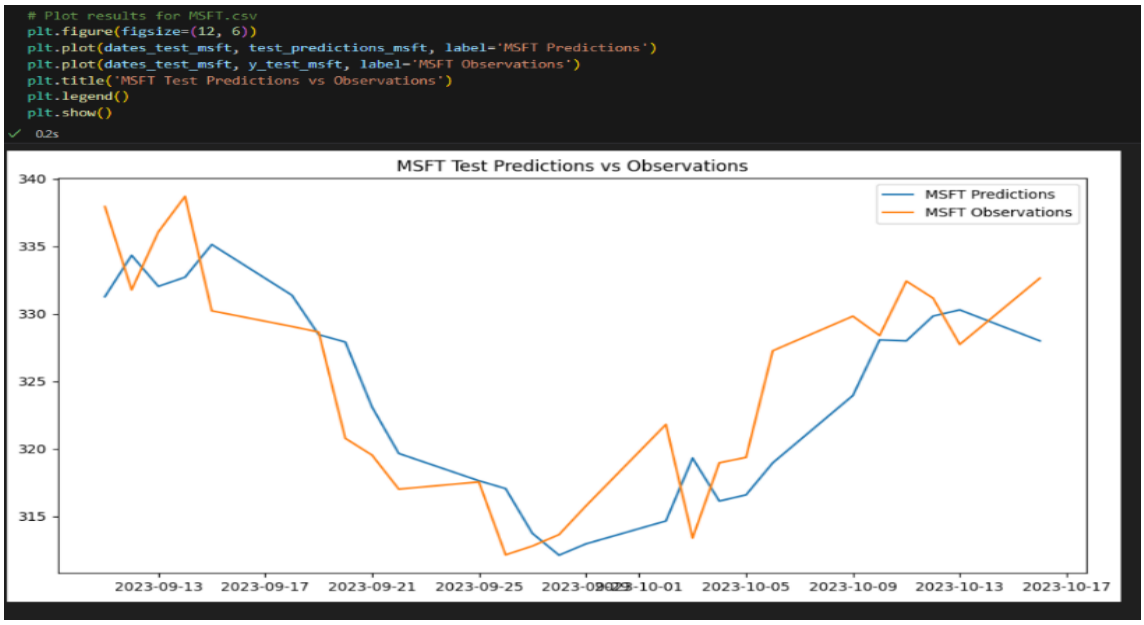



*Figure 15. MSFT Test*

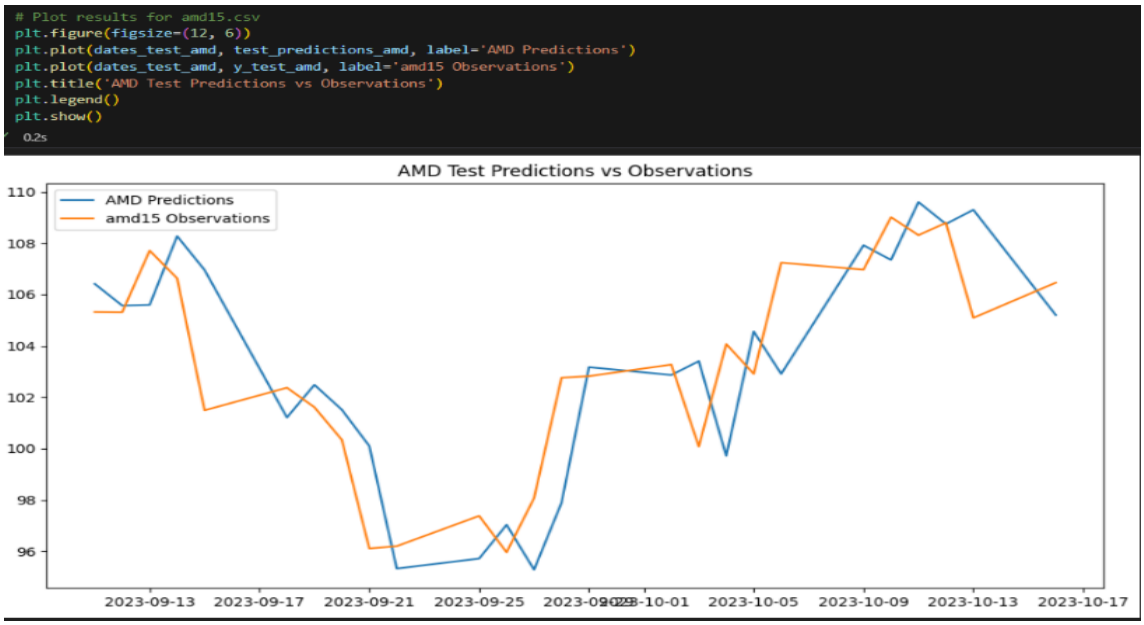



*Figure 16. AMD Test*

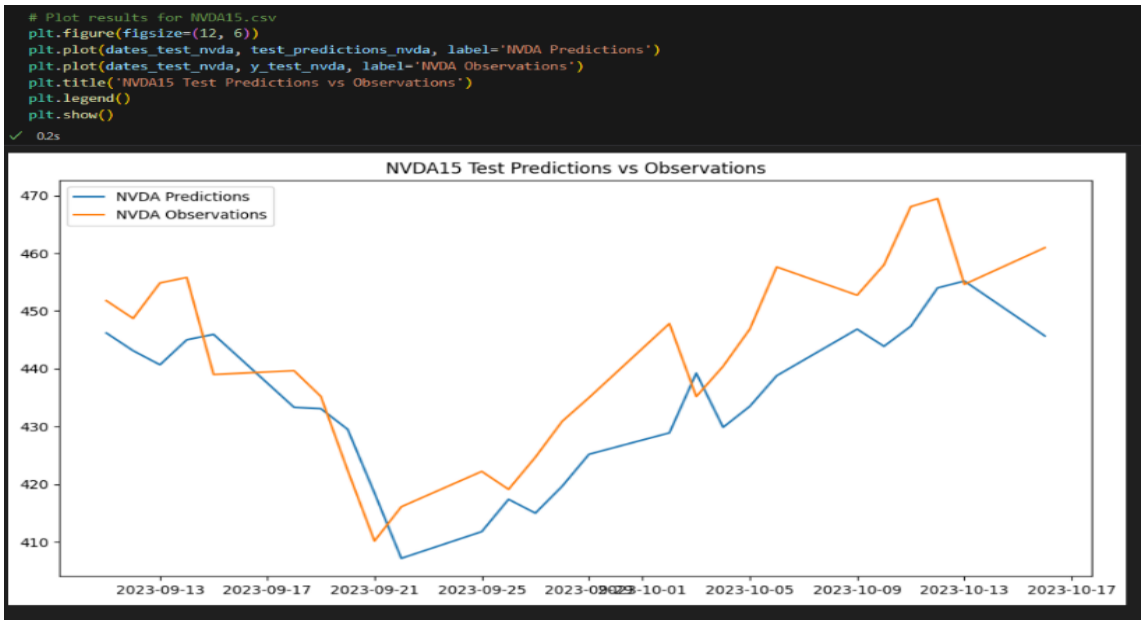



*Figure 17. NVDA Test*

Each set of results was displayed in a separate figure with a specified figure size. For each dataset, two lines were plotted in the figure: one representing the predictions made by the model (test_predictions_msft, test_predictions_amd, test_predictions_nvda), and the other representing the actual observations (y_test_msft, y_test_amd, y_test_nvda). The plots are labeled to indicate whether they represent predictions or observations, and each figure has a title indicating which dataset and what was being compared. Finally, a legend was added to distinguish between predictions and observations, and the figures are displayed to visualize and compare the model's predictions with the actual data for each dataset.

## CNN

The data processing for the Convolutional Neural Network (CNN) follows the same procedures as the previously explained LSTM model. The initial steps involve loading and preprocessing the time series data, creating windowed datasets, and splitting them into training, validation, and test sets. These processes are consistent across both models, ensuring that the input features (historical closing prices) are prepared in a similar manner. The CNN model, like the LSTM model, utilizes sliding windows of historical data for making predictions, with the choice of hyperparameters, training, and evaluation following the same general approach. Therefore, the primary distinction between the models lies in the architectural differences between CNN and LSTM, while the data processing and evaluation aspects remain consistent.

In this code in Figure 18, a Convolutional Neural Network (CNN) model was defined using the Keras library.

```
from keras.models import Sequential
from keras.optimizers import Adam
from keras import layers

model = Sequential([
    layers.Input((3, 1)),
    layers.Conv1D(64, kernel_size=2, activation='relu'),  # CNN layer
    layers.Flatten(),  # Flatten the output before feeding it to dense layers
    layers.Dense(32, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(1)
])

model.compile(loss='mse',
              optimizer=Adam(learning_rate=0.001),
              metrics=['mean_absolute_error'])
✓ 5.9s
```

*Figure 18. CNN Model*

The model was structured as a sequential set of layers. It starts with an input layer, which expects input data of shape (3, 1). The CNN layer follows, with 64 filters and a kernel size of 2, using the ReLU activation function. The next layer was a Flatten layer, which reshapes the output from the previous layer into a one-dimensional vector to prepare it for dense layers. Two dense layers with 32 units and ReLU activation functions were added, followed by a final dense layer with a single unit. The model was compiled with a mean squared error (MSE) loss function, the Adam optimizer with a learning rate of 0.001, and it computes the mean absolute error (MAE) as a metric during training. This code defines the architecture of a CNN model for regression tasks, where the goal was to predict a continuous target variable.

In the code in Figure 19, the machine learning model is being trained. It uses the training data, X_train and y_train, to optimize its parameters for making predictions.

```
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=100)
✓ 6.5s
Epoch 1/100
7/7 [==============================] - 1s 45ms/step - loss: 79292.0469 - mean_absolute_error: 260.7567 - val_loss: 149068.2656 - val_mean_absolute_error: 385.576
Epoch 2/100
7/7 [==============================] - 0s 9ms/step - loss: 47378.2031 - mean_absolute_error: 201.4758 - val_loss: 85954.3203 - val_mean_absolute_error: 292.6616
Epoch 3/100
7/7 [==============================] - 0s 8ms/step - loss: 25477.1348 - mean_absolute_error: 147.4915 - val_loss: 37417.0508 - val_mean_absolute_error: 192.8083
Epoch 4/100
7/7 [==============================] - 0s 8ms/step - loss: 9753.7217 - mean_absolute_error: 90.2302 - val_loss: 7954.3701 - val_mean_absolute_error: 87.9937
Epoch 5/100
7/7 [==============================] - 0s 8ms/step - loss: 1315.1034 - mean_absolute_error: 29.3098 - val_loss: 503.3594 - val_mean_absolute_error: 18.8340
Epoch 6/100
7/7 [==============================] - 0s 9ms/step - loss: 643.1213 - mean_absolute_error: 21.0545 - val_loss: 4242.5137 - val_mean_absolute_error: 63.0614
Epoch 7/100
7/7 [==============================] - 0s 9ms/step - loss: 1324.8883 - mean_absolute_error: 32.1187 - val_loss: 2280.4111 - val_mean_absolute_error: 44.9868
Epoch 8/100
7/7 [==============================] - 0s 9ms/step - loss: 444.1693 - mean_absolute_error: 17.1681 - val_loss: 321.8033 - val_mean_absolute_error: 14.9322
Epoch 9/100
7/7 [==============================] - 0s 7ms/step - loss: 165.2545 - mean_absolute_error: 8.7991 - val_loss: 396.0145 - val_mean_absolute_error: 16.2687
Epoch 10/100
```

*Figure 19. CNN Train*

Additionally, it evaluates the model's performance during training using the validation data, which consists of X_val and y_val.

The training process occurs over a specified number of iterations known as epochs, which is set to 100 in this case. During each epoch, the model updates its internal parameters based on the training data and calculates its performance on the validation data to monitor progress and prevent overfitting. This line of code essentially initiates the training process for the model.

```
train_predictions = model.predict(X_train).flatten()

plt.plot(dates_train, train_predictions)
plt.plot(dates_train, y_train)
plt.legend(['Training Predictions', 'Training Observations'])

✓ 0.3s
7/7 [==============================] - 0s 1ms/step
<matplotlib.legend.Legend at 0x1e25521e650>
```

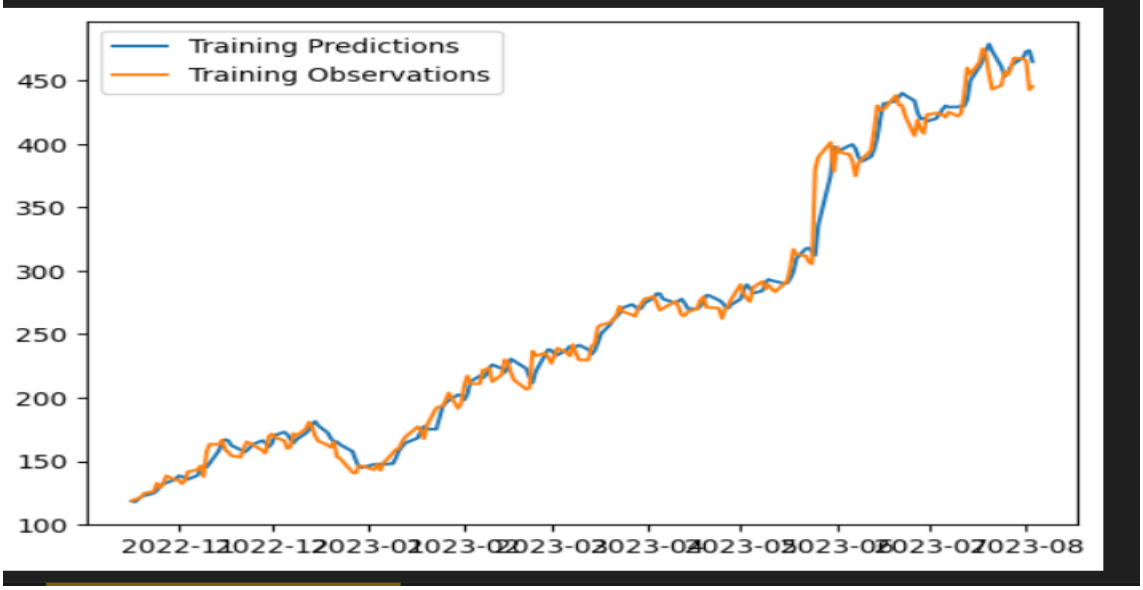



*Figure 20. Train Predictions NVDA*

In the code above in Figure 20, the machine learning model, referred to as model, was used to make predictions on the training data X_train. The predicted values were flattened into a one-dimensional array and stored in the variable train_predictions. Subsequently, the code creates a plot using the matplotlib library. It displays two lines on the plot: one line represents the training predictions made by the model, and the other line represents the actual training observations stored in y_train. The plt.plot function was used to visualize these two sets of data over the training dates, and plt.legend adds a legend to the plot to distinguish between training predictions and training observations for the MSFT dataset. This code allows for a visual comparison of the model's predictions and the actual training data to assess its performance.

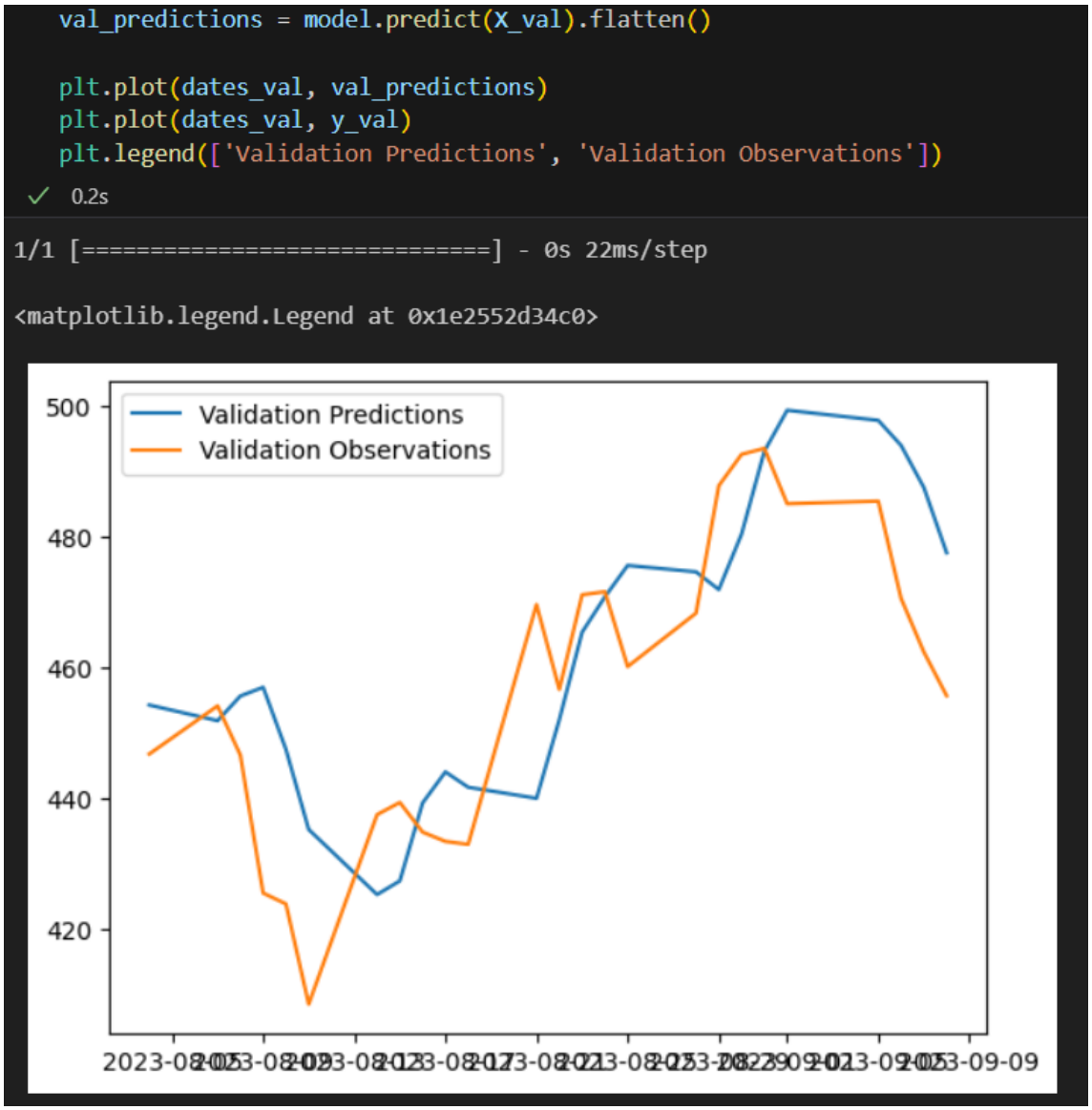



*Figure 21. Validation Predictions NVDA*

In the code above in Figure 21, the machine learning model (model) was used to make predictions on the validation data, which is contained in X_val. The predicted values were then flattened into a one-dimensional array and stored in the variable val_predictions. The code proceeds to create a plot using the matplotlib library, showing two lines on the plot: one line represents the validation predictions generated by the model, and the other line represents the actual validation observations stored in y_val. The plt.plot function was utilized to visualize these two sets of data over the validation dates, and plt.legend is employed to add a legend to the plot, differentiating between validation predictions and validation observations for the MSFT dataset. This code allows for a visual comparison of the model's predictions and the actual validation data to evaluate its performance on the validation set.

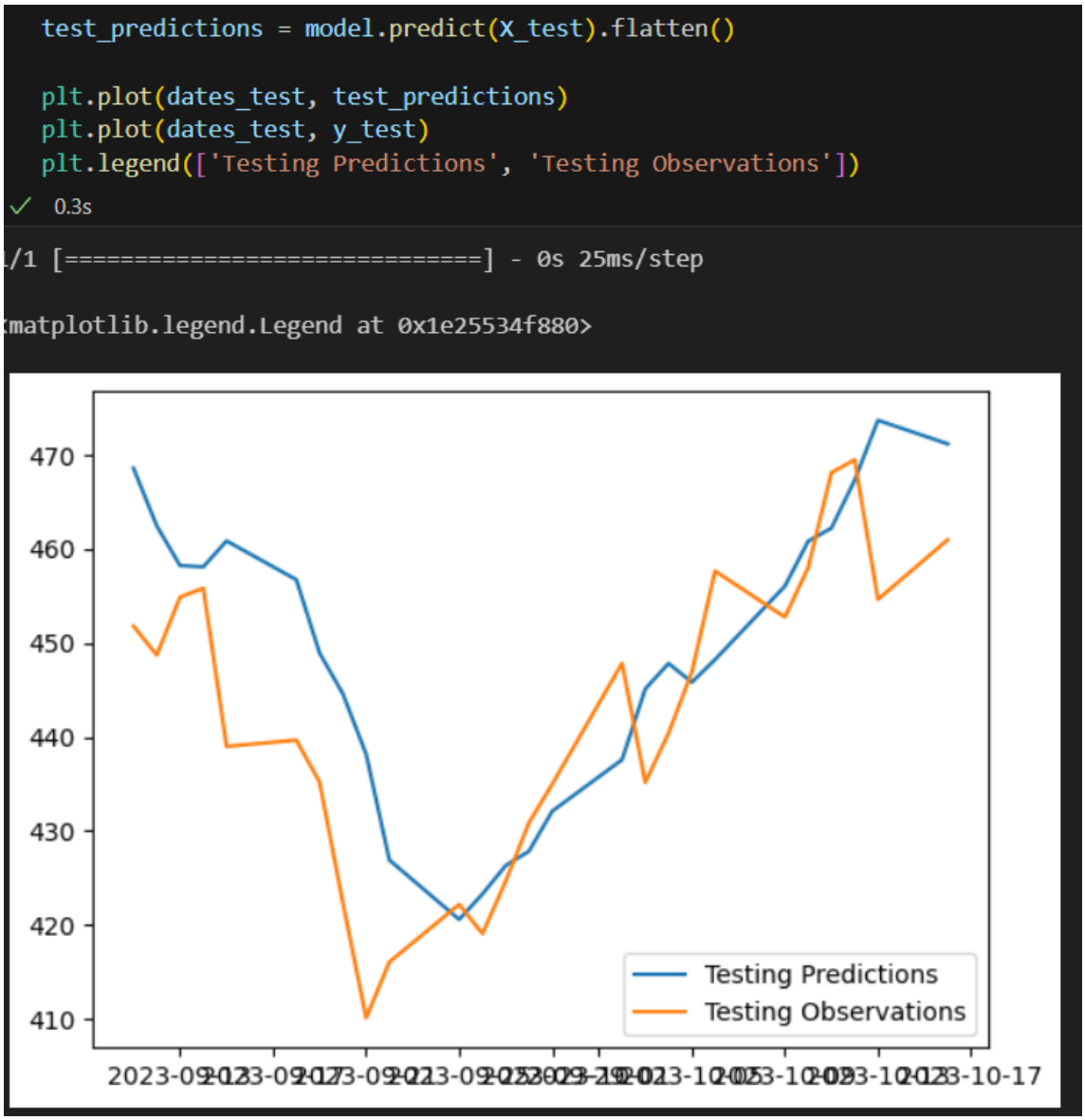



*Figure 22. Test Predictions NVDA*

In this code, the machine learning model, referred to as model, was used to make predictions on the test data, which was contained in X_test. The predicted values were then flattened into a one-dimensional array and stored in the variable test_predictions. The code proceeds to create a plot using the matplotlib library, displaying two lines on the plot: one line represents the testing predictions generated by the model, and the other line represents the actual testing observations stored in y_test. The plt.plot function was employed to visualize these two sets of data over the test dates, and plt.legend was added to the plot to distinguish between testing predictions and testing observations for the MSFT dataset. This code enables a visual comparison of the model's predictions and the actual testing data, providing insight into the model's performance on the test set.

The following series of plots presented in Figures 23-25 represent the training, validation, and testing results for the AMD (Advanced Micro Devices) stock.

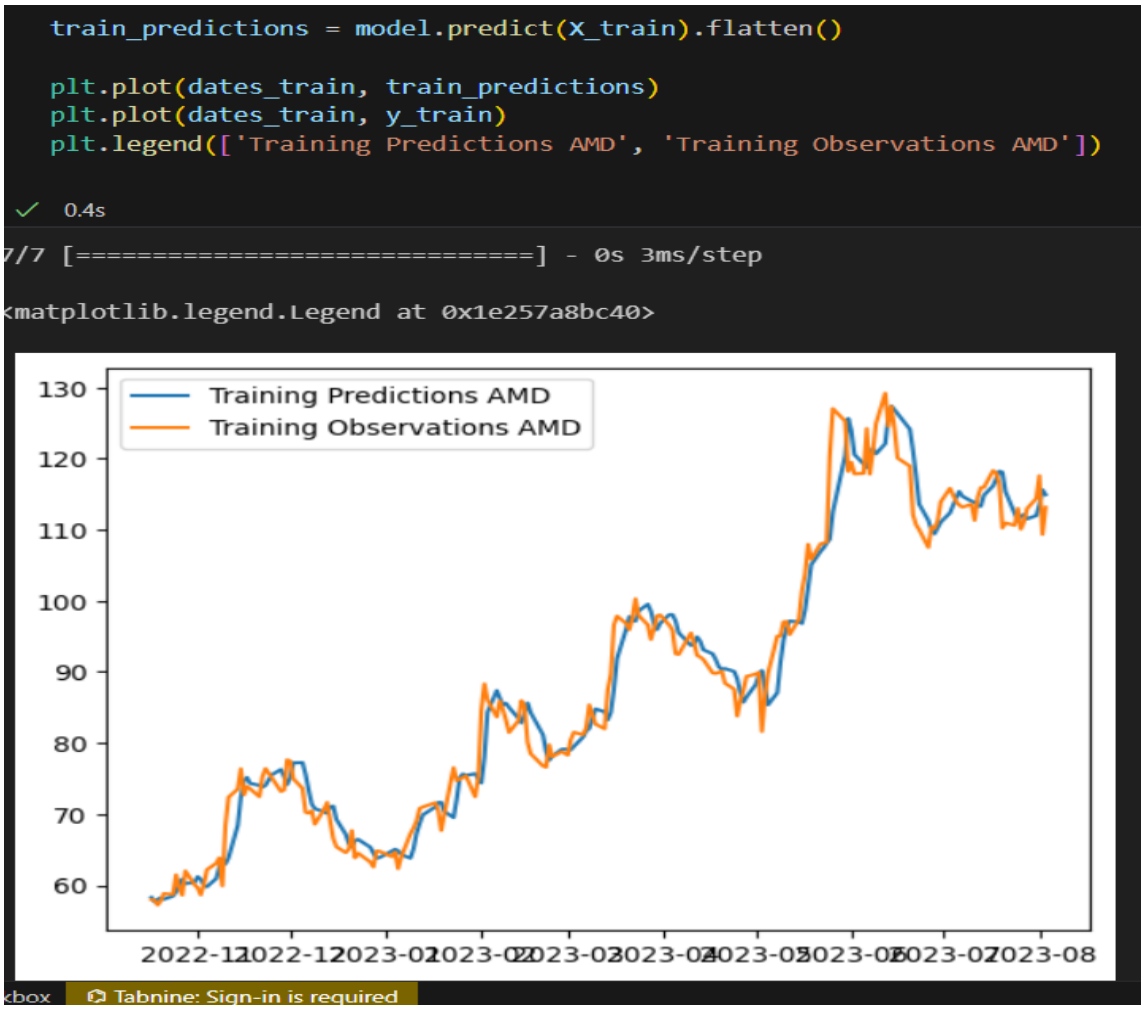



*Figure 23. Train Prediction AMD*


*Figure 24. Validation Prediction AMD*


*Figure 25. Test Prediction AMD*

Each plot showcases the performance of a machine learning model applied to AMD stock data. In the first plot, the training results were displayed, illustrating how well the model's predictions align with the actual training observations over a specific period of time. The second plot shows the validation results, offering insights into the model's performance on previously unseen validation data. Lastly, the third plot visualizes the testing results, which assess the model's effectiveness in making predictions on a separate testing dataset. These plots collectively provide a comprehensive view of how the machine learning model performs on AMD stock data across various stages of training and evaluation.

The series of plots showcased depicted in Figures 26-28 are the training, validation, and testing results specifically tailored to the MSFT (Microsoft Corporation) stock.

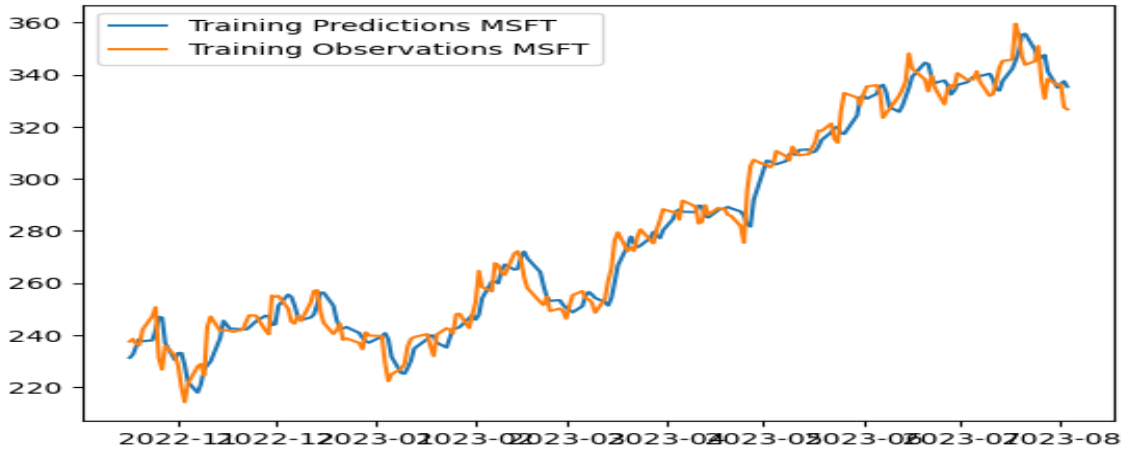



*Figure 26. Train Prediction MSFT*

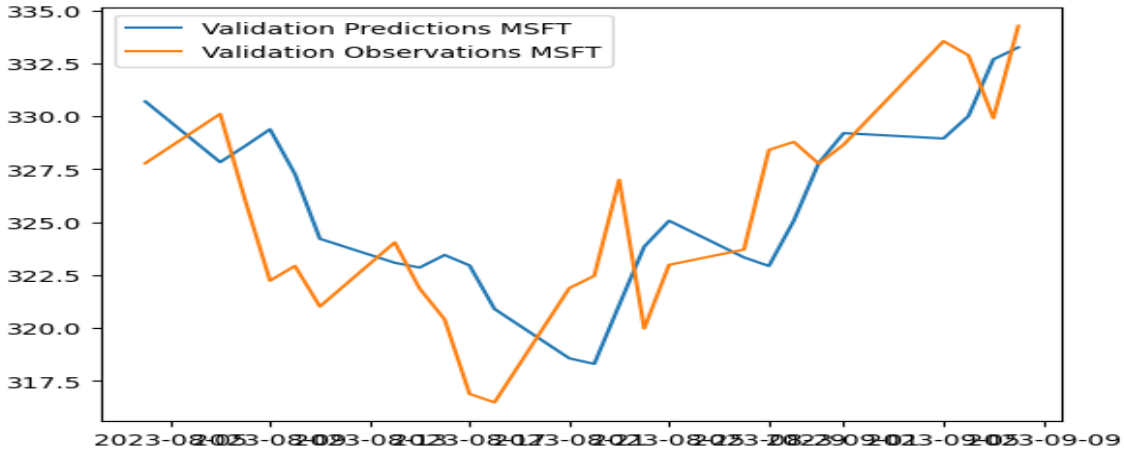



*Figure 27. Validation Prediction MSFT*

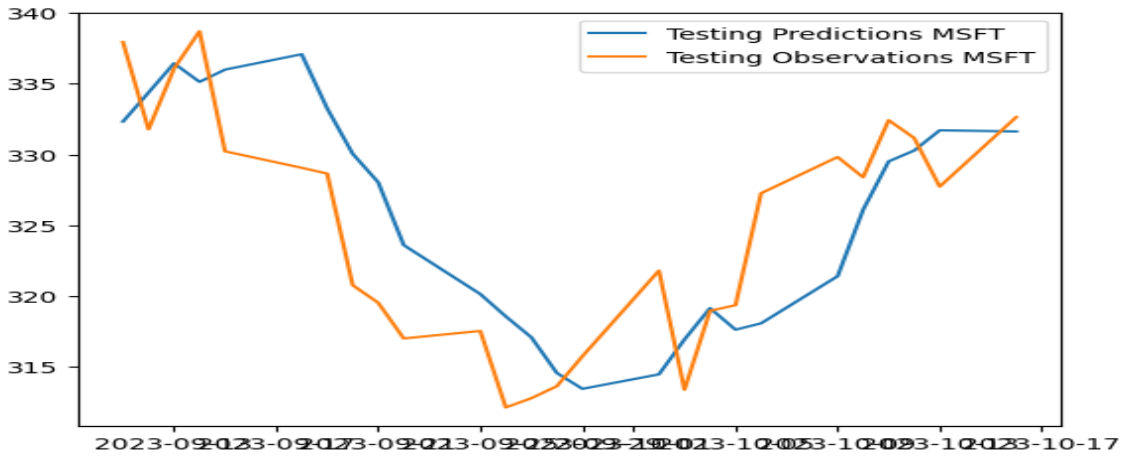



*Figure 28. Test Prediction MSFT*

Each plot offers a distinct perspective on the performance of a machine learning model applied to MSFT stock data. The first plot represents the training results, showcasing the alignment between the model's predictions and the actual training observations

throughout a defined time frame. The second plot was dedicated to the validation results, providing insights into how well the model generalizes to new data not seen during training. Finally, the third plot visualizes the testing results, reflecting the model's predictive capabilities on a distinct testing dataset. Together, these plots provide a comprehensive assessment of the model's efficacy in handling MSFT stock data across various stages of training and evaluation.

## Transformer

The Transformer model, like the previously discussed models, employs the same data processing procedures. It begins with loading and preprocessing time series data, constructing windowed datasets, and splitting them into training, validation, and test sets. These fundamental steps ensure that the input features, which are historical closing prices, are prepared consistently. The Transformer model, however, distinguishes itself in its architecture and mechanism for handling sequential data. Despite the architectural differences, the underlying data processing remains consistent, allowing for a seamless transition from one model to another while ensuring a consistent approach to data preparation and evaluation.

In the code below in Figure 29, a Transformer-based time series model was defined using the TensorFlow and Keras libraries.

```python
from keras.models import Sequential
from keras.optimizers import Adam
from keras import layers
import tensorflow as tf

# Define the model
class TransformerTimeSeries(tf.keras.Model):
    def __init__(self, seq_length, d_model, num_heads, dff, num_layers, dropout_rate=0.1):
        super(TransformerTimeSeries, self).__init__()

        self.embedding = layers.Dense(d_model)
        self.transformer = layers.MultiHeadAttention(num_heads=num_heads, key_dim=d_model, value_dim=d_model)
        self.linear = layers.Dense(d_model, activation='relu')  # Add this line
        self.dropout = layers.Dropout(dropout_rate)
        self.layer_norm = layers.LayerNormalization(epsilon=1e-6)
        self.global_average_pooling = layers.GlobalAveragePooling1D()
        self.dense_output = layers.Dense(1)  # Regression task, predicting one value

    def call(self, inputs, training=True):
        x = self.embedding(inputs)
        attn_output = self.transformer(x, x)
        attn_output = self.linear(attn_output)  # Add this line
        x = self.dropout(attn_output, training=training)
        x = self.layer_norm(x + attn_output)
        x = self.global_average_pooling(x)
        return self.dense_output(x)


# Instantiate the model
seq_length = X.shape[1]  # Set the sequence length based on the input shape
model = TransformerTimeSeries(seq_length, d_model=32, num_heads=3, dff=64, num_layers=4)

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001), loss="mse")
```

*Figure 29. Transformer Model Definition*

The model architecture was encapsulated within a custom class called TransformerTimeSeries. This class comprises several layers, including an initial embedding layer that converts input data into a higher-dimensional representation, a multi-head self-attention layer, a linear layer with ReLU activation, a dropout layer for regularization, a layer normalization step, a global average pooling layer, and a final dense output layer for regression tasks. The call method defines the forward pass of the model, processing the input data through these layers. The model was instantiated with specific hyperparameters such as the sequence length, model dimensions, number of attention heads, and layer depth. It was compiled with the Adam optimizer and the mean squared error (MSE) loss function, indicating that the model was designed for regression tasks. This code sets up a Transformer-based model for time series forecasting, with the capacity to make predictions based on the provided input data.

In the code in Figure 30, the Transformer-based time series model was trained on the provided time series data.

```
# Train the model
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=800)
```

✓ 46.8s

```
Epoch 1/800
7/7 [==============================] - 2s 38ms/step - loss: 77721.9766 - val_loss: 204773.9844
Epoch 2/800
7/7 [==============================] - 0s 9ms/step - loss: 77122.5547 - val_loss: 204409.7812
Epoch 3/800
7/7 [==============================] - 0s 9ms/step - loss: 76943.8516 - val_loss: 204118.2812
Epoch 4/800
7/7 [==============================] - 0s 8ms/step - loss: 76787.2812 - val_loss: 203836.8750
Epoch 5/800
7/7 [==============================] - 0s 8ms/step - loss: 76626.5781 - val_loss: 203540.9375
Epoch 6/800
7/7 [==============================] - 0s 9ms/step - loss: 76463.4609 - val_loss: 203254.1250
Epoch 7/800
7/7 [==============================] - 0s 10ms/step - loss: 76298.1250 - val_loss: 202952.1562
Epoch 8/800
7/7 [==============================] - 0s 9ms/step - loss: 76132.6719 - val_loss: 202649.9062
Epoch 9/800
7/7 [==============================] - 0s 10ms/step - loss: 75955.0312 - val_loss: 202340.5625
Epoch 10/800
7/7 [==============================] - 0s 9ms/step - loss: 75789.3672 - val_loss: 202026.2188
Epoch 11/800
7/7 [==============================] - 0s 9ms/step - loss: 75610.8984 - val_loss: 201700.3438
Epoch 12/800
7/7 [==============================] - 0s 10ms/step - loss: 75427.3828 - val_loss: 201369.5469
Epoch 13/800
7/7 [==============================] - 0s 9ms/step - loss: 75238.3203 - val_loss: 201033.1562
Epoch 14/800
7/7 [==============================] - 0s 9ms/step - loss: 75056.9141 - val_loss: 200688.6875
Epoch 15/800
7/7 [==============================] - 0s 8ms/step - loss: 74861.8516 - val_loss: 200338.3125
Epoch 16/800
```

*Figure 30. Training the Transformer*

The training process entails using the training data (X_train and y_train) to optimize the model's parameters. Additionally, the model's performance was assessed using the validation data (X_val and y_val) to ensure its ability to generalize to unseen data. Notably, the model was trained over a relatively high number of epochs, specifically 800. This extended training duration can be attributed to the model's increased complexity compared to the other models. As a more intricate architecture, the Transformer model initially exhibits a higher mean squared error (MSE) due to its complexity, and it requires a greater number of training iterations to reach a desirable level of accuracy. This code showcases the extensive training process needed to fine-tune the Transformer model and improve its predictive performance for time series data.

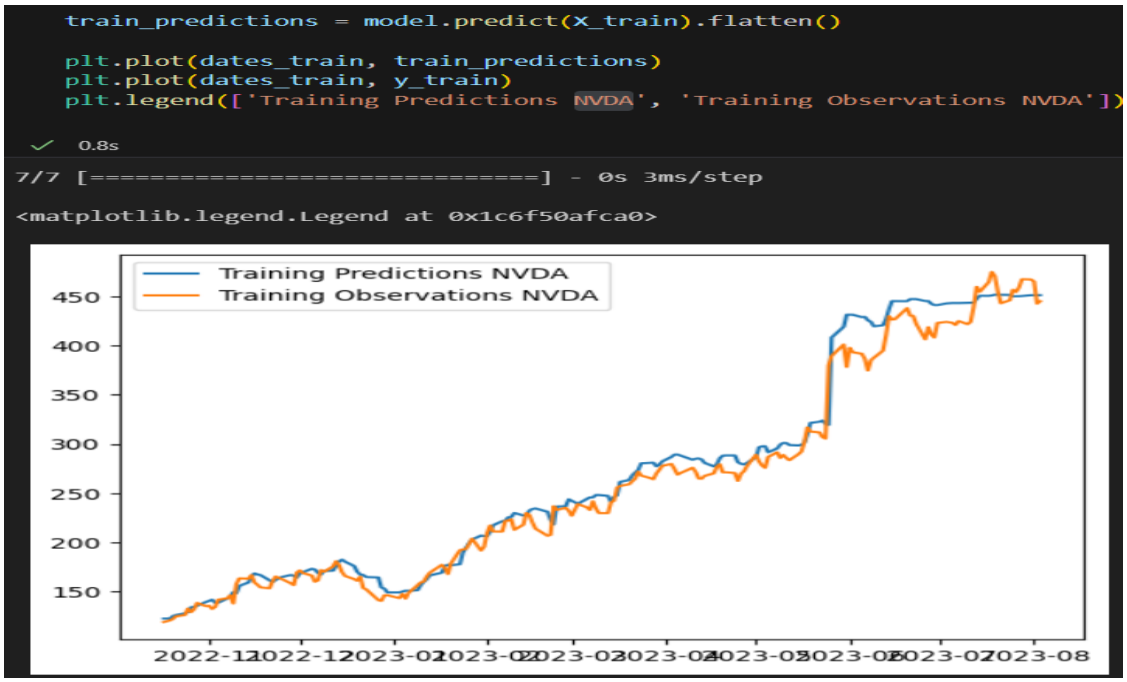



*Figure 31. Train NVDA*

```
val_predictions = model.predict(X_val).flatten()

plt.plot(dates_val, val_predictions)
plt.plot(dates_val, y_val)
plt.legend(['Validation Predictions NVDA', 'Validation Observations NVDA'])
```

0.3s

```
1/1 [==============================] - 0s 33ms/step

<matplotlib.legend.Legend at 0x1c6f514baf0>
```

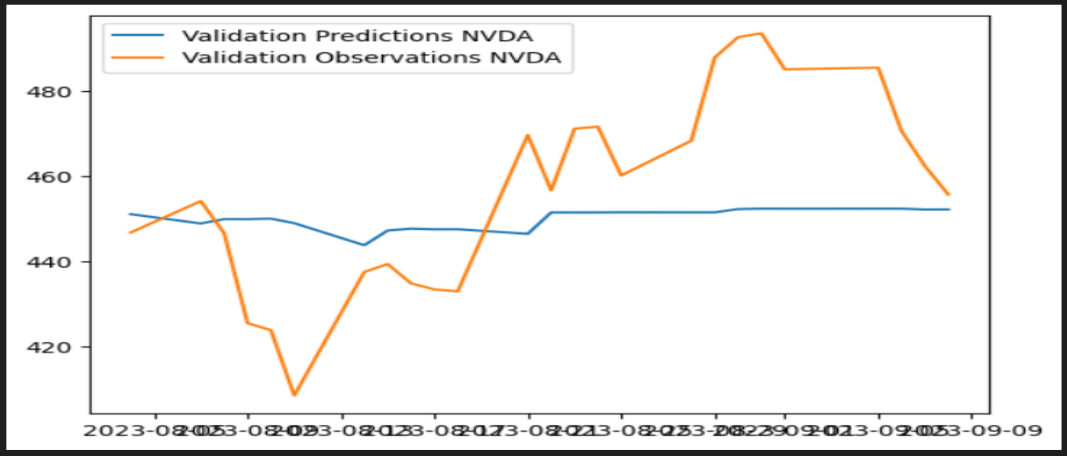



*Figure 32. Validation NVDA*

```
test_predictions = model.predict(X_test).flatten()

plt.plot(dates_test, test_predictions)
plt.plot(dates_test, y_test)
plt.legend(['Testing Predictions NVDA', 'Testing Observations NVDA'])
```

0.3s

```
1/1 [==============================] - 0s 30ms/step

<matplotlib.legend.Legend at 0x1c6f519fa90>
```

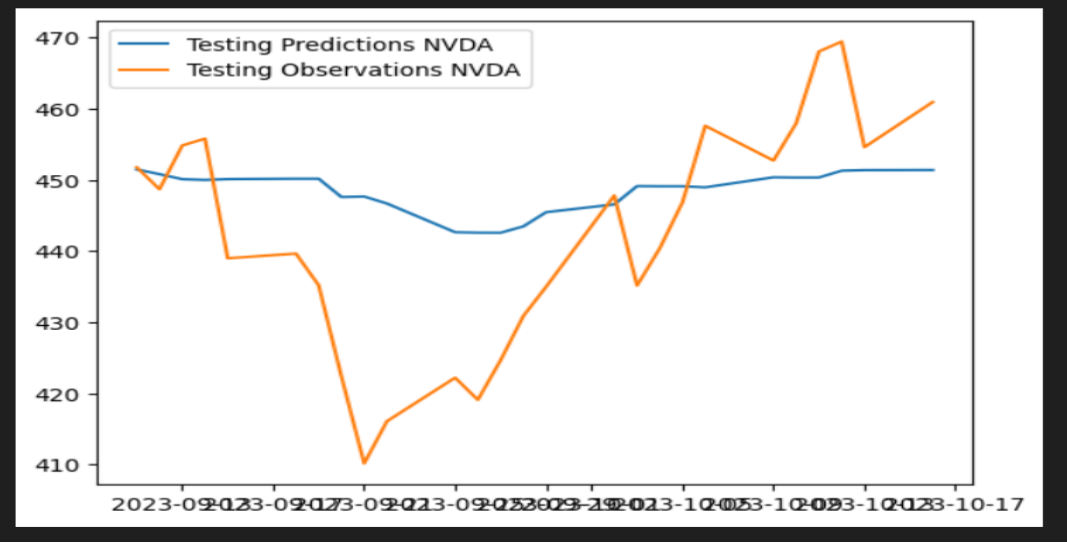



*Figure 33. Test NVDA*

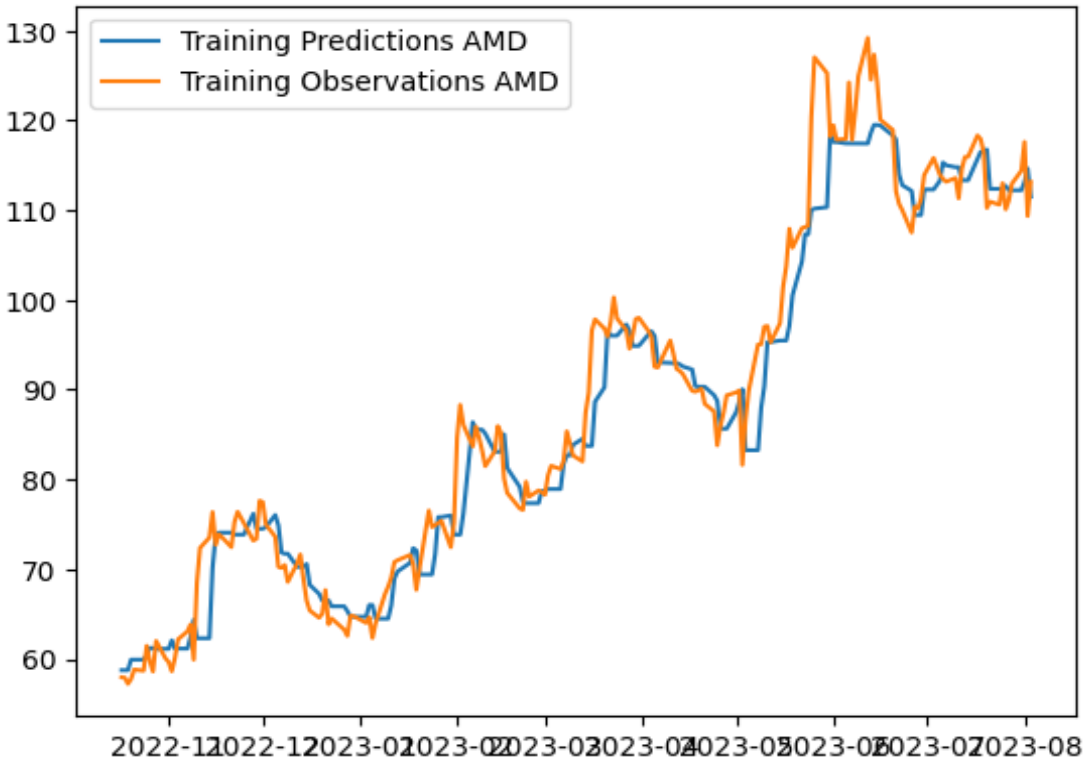



*Figure 34. Train AMD*

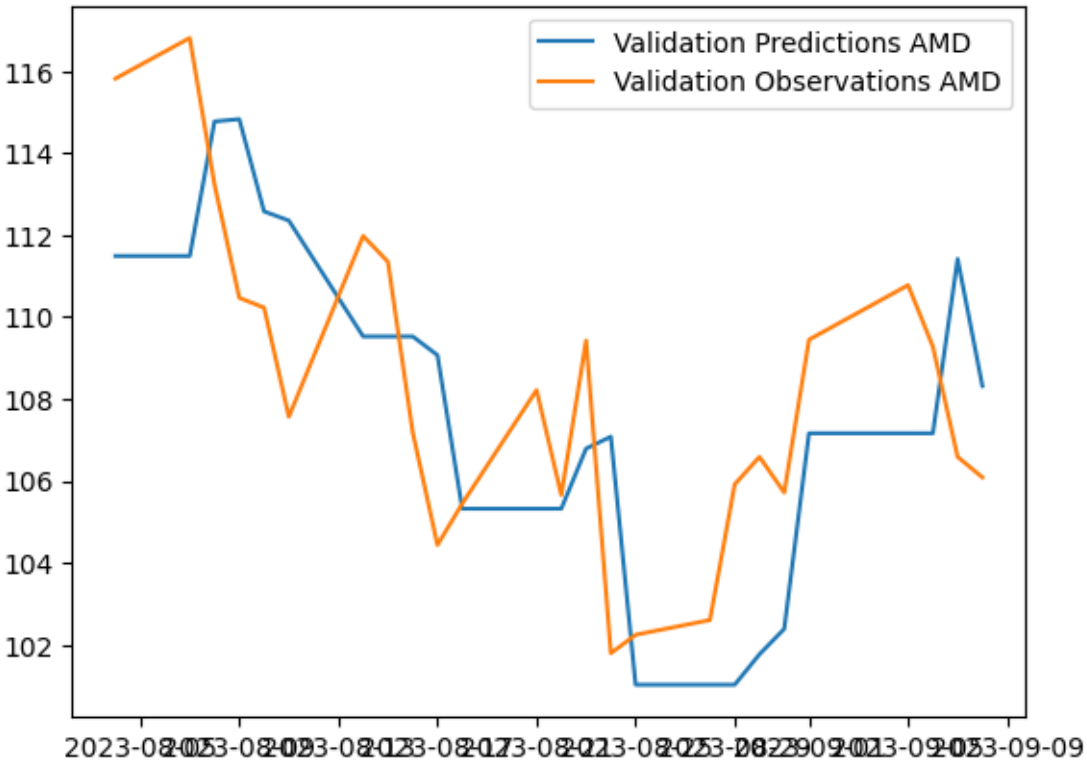



*Figure 35. Validation AMD*

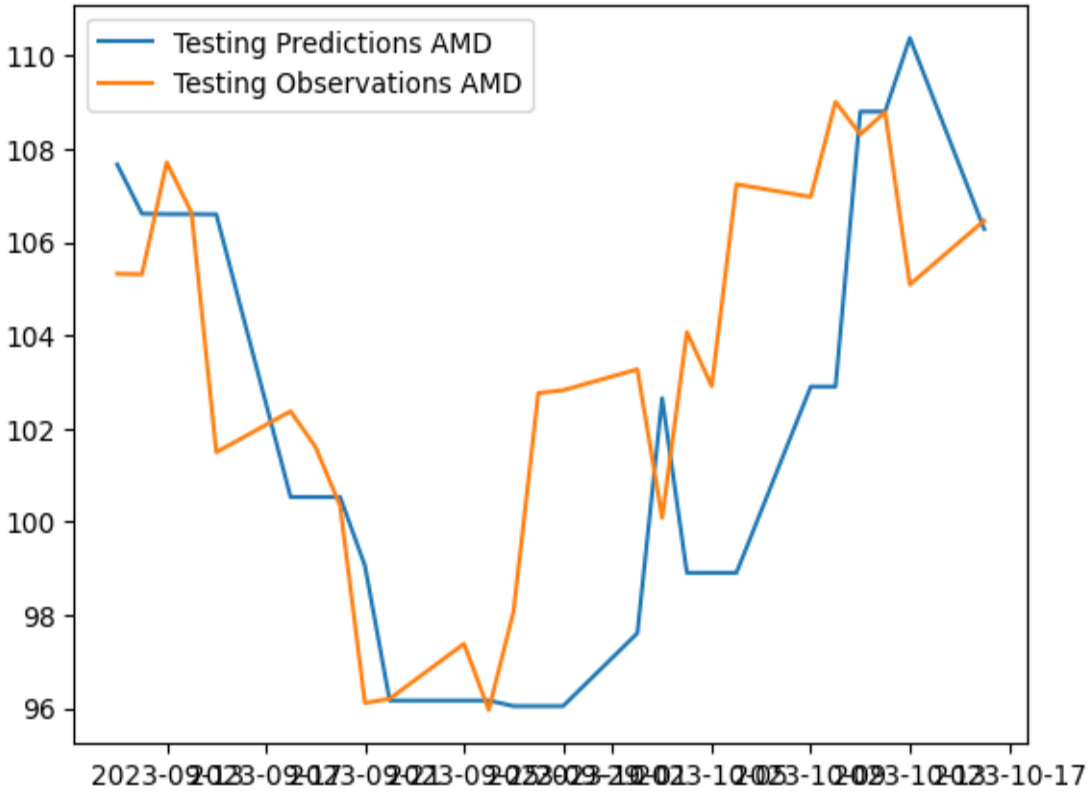



*Figure 36. Test AMD*

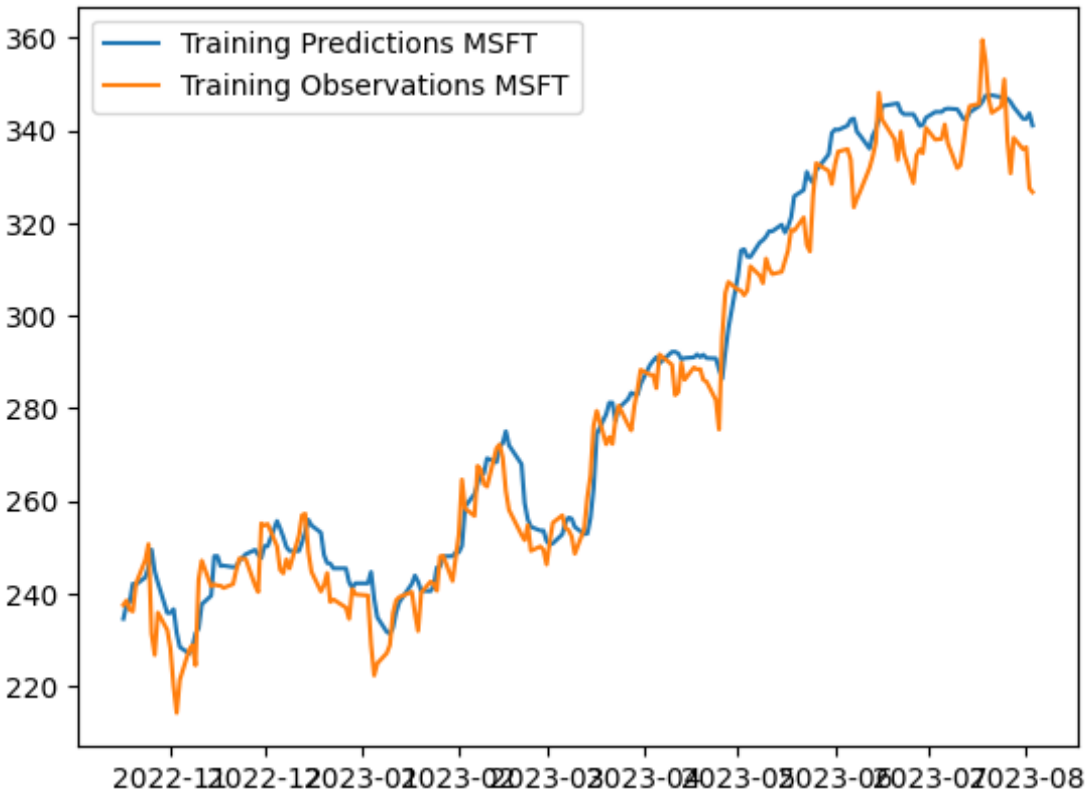



*Figure 37. Train MSFT*

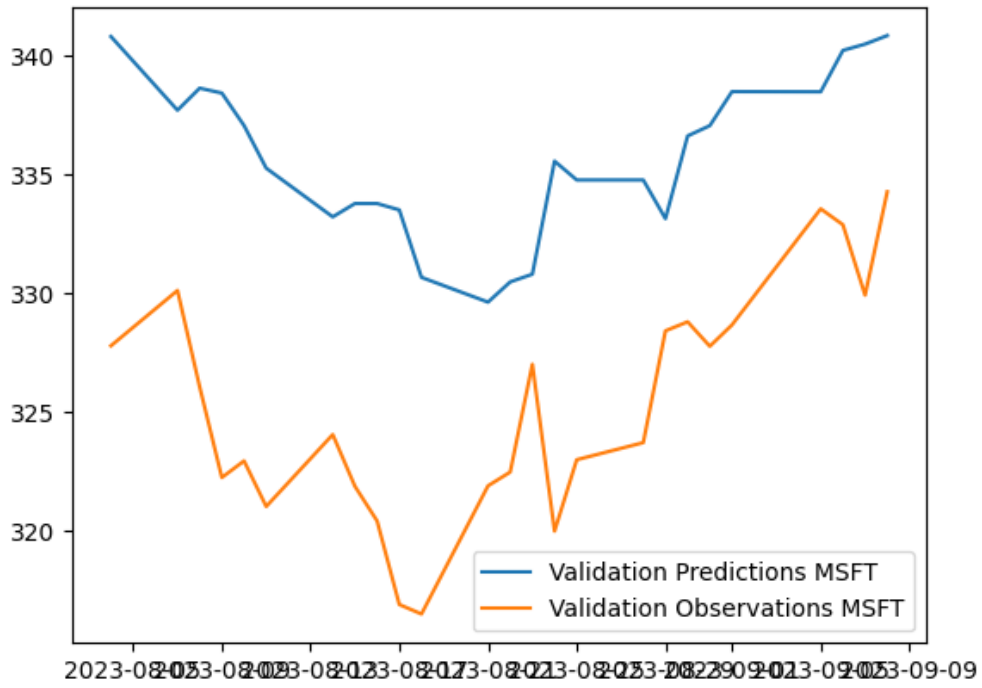



*Figure 38. Validation MSTF*

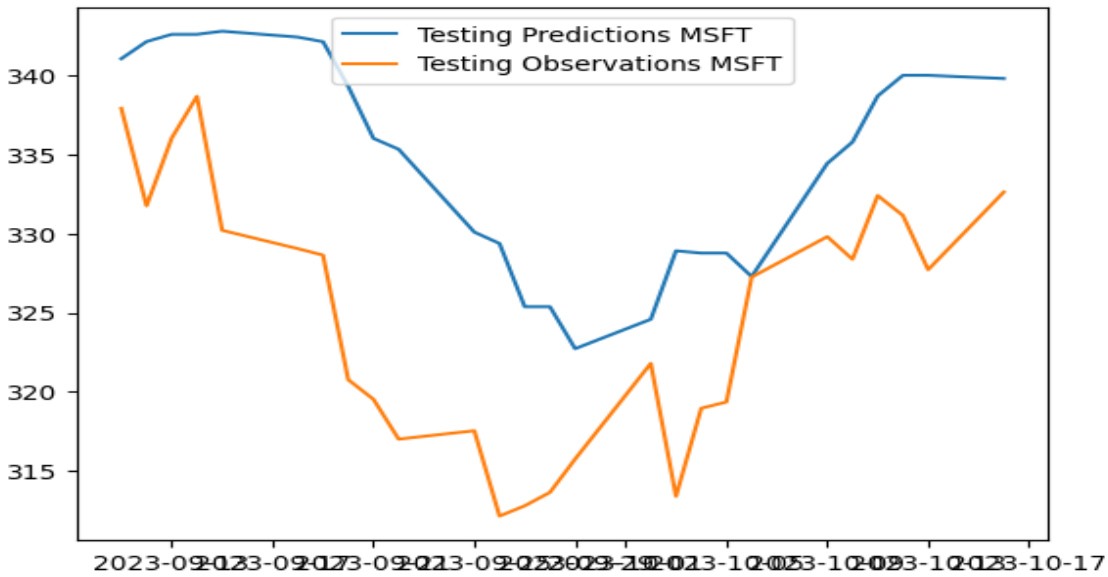



*Figure 39. Testing MSFT*

In the context of the time series forecasting models evaluated, the plots presented above in Figures 31 – 39 showcase the training, validation, and testing results for every stock. Notably, the Transformer model, as seen in these visualizations, exhibited relatively poorer performance when compared to its LSTM or CNN counterparts. However, it's essential to underscore that while the Transformer model didn't perform optimally for NVDA stock, it managed to provide some level of accuracy in its predictions. In

contrast, the other stocks, AMD and MSFT, displayed worse predictive capabilities, even if they were still able to provide some degree of accuracy. These results underscore the intricate nature of time series forecasting and emphasize the importance of selecting the most suitable model architecture for each specific dataset, as the best-performing model may vary depending on the characteristics of the underlying data.

# Conclusion

In this extensive investigation of time series estimating models applied to authentic stock cost information, important experiences were aquired into the presentation of three essential model designs: Long Short-Term Memory (LSTM), Convolutional Neural Network (CNN), and the Transformer model. Through code exhibitions and visual portrayals, it was arrived at a few vital resolutions.

Foremost, the findings reveal that the LSTM model emerged as the top performer in capturing temporal dependencies within the stock price data. Its simplicity, combined with its ability to effectively retain sequential information, made it a reliable choice for stock price prediction. Across all three stocks -Microsoft (MSFT) MSE (Mean Squared Error) of 4.88 and VMSE (Validation Mean Squared Error metric provided by the library keras) of 2.76, Advanced Micro Devices (AMD) MSE of 2.09 and VMSE of 2.42, and NVIDIA Corporation (NVDA)MSE of 7.60 and VMSE of 14.35 - the LSTM consistently demonstrated the highest predictive accuracy as the averege MSE and VMSE were 4.85 and 6.49 respectfully.

The CNN model, with its aptitude for identifying spatial patterns in data, presented promising results. Across all three stocks - -Microsoft (MSFT) MSE of 5.54 and VMSE of 3.33, Advanced Micro Devices (AMD) MSE of 2.5 and VMSE of 2.84, and NVIDIA Corporation (NVDA)MSE of 8.05 and VMSE of 13.88 - the LSTM consistently demonstrated the highest predictive accuracy as the averege MSE and VMSE were 5.36 and 6.68 respectfully.

Conversely, the Transformer model initially exhibited significantly higher Mean Squared Error (MSE) values and necessitated a more extended training period. Across all three stocks - -Microsoft (MSFT) MSE of 6.84 and VMSE of 8.31, Advanced Micro Devices (AMD) MSE of 2.96 and VMSE of 3.59, and NVIDIA Corporation (NVDA)MSE of 13.62 and VMSE of 19.33 - the LSTM consistently demonstrated the highest predictive accuracy as the averege MSE and VMSE were 7.80 and 10.41

respectfully. This initial lower performance can be attributed to the model's complex multi-head attention mechanism. Nevertheless, it's crucial to recognize that with a more extended training duration, the Transformer model ultimately revealed strong predictive capabilities. This observation underscores the Transformer's potential in time series forecasting. It is worth noting that for the Transformer to reach its full potential, more sophisticated encoding and decoding mechanisms may be required.

With that said the models were made as simple as possible to even the playing feild and make the results comparable. With that said the hyperparameters did make a difference when changed but to a small degree so the focuse was on the data processing. Additionally, future research can explore the incorporation of a broader set of features beyond just 'Close,' including 'Volume,' 'High,' 'Low,' and other relevant parameters. Also the models could be made more complex with more extensive hyperparameter tuning and the addition of more dense and RNN layers.

Finally, the LSTM model emerged as the most reliable choice for stock price prediction in our analysis, with the CNN following closely behind. While the Transformer initially lagged in performance, it holds promise for future research with the implementation of more advanced techniques and feature engineering. This thesis offers valuable insights into the strengths and limitations of deep learning models in the context of historical stock price prediction, guiding future explorations and advancements in this domain.

## *Future work*

To upgrade the accuracy of LSTM, CNN, and Transformer models in resulting applications, a few methodologies can be utilized.

For LSTM models, it's pivotal to guarantee that your information is pre-handled suitably. This incorporates:

- Tokenization: This is the method involved with separating the information into more modest parts, or "tokens". With regards to message information, this normally implies dividing sentences into individual words.
- Cushioning: LSTM models require input arrangements to be of a similar length. Cushioning is utilized to guarantee this by adding "filler" values until all successions are a similar length.

- Word Implanting: This includes addressing words as vectors in a high-layered space. The place of each word in this space is picked up during preparing and can catch semantic significance.

Parameter tuning: Trying different things with various boundaries can altogether affect the presentation of a LSTM model. These boundaries include:

- Number of epochs: This alludes to how frequently the learning calculation will manage the whole preparation dataset.
- Learning Rate: This decides the amount to change the model in light of the assessed mistake each time the model loads are refreshed.
- Bunch Size: This is the quantity of preparing models utilized in one cycle.
- Number of Layers: This alludes to the quantity of layers in the brain organization. More layers can catch more complicated designs, yet in addition require more computational assets and chance overfitting.

Regularization: Methods like dropout or L1/L2 regularization can be utilized to forestall overfitting. Dropout includes haphazardly setting a small part of information units to 0 at each update during preparing time, which can forestall complex co-transformations on preparing information. L1/L2 regularization adds a punishment to the misfortune capability to decrease overfitting by putting complex models down.

Convolutional Neural Networks (CNN) Models:

- Transfer Learning: Transfer learning includes utilizing a model that has been pre-prepared on comparable errands as opposed to preparing another model without any preparation. For instance, a CNN model that has been prepared on picture characterization undertakings can be calibrated for a particular picture acknowledgment task. This approach can save critical time and computational assets.
- Data Augmentation: Data augmentation includes making new preparation models by applying changes, for example, trimming, cushioning, or flat turning to your current dataset. This can build the size and variety of your preparation set, prompting worked on model execution and speculation.

- Parameter Tuning: Like LSTM models, tuning parameters like epochs, learning rate, bunch size, and the quantity of layers can work on model execution.

Transformer Models:

- Increasing Model Size: In deep learning, utilizing more register assets frequently prompts higher precision. This could mean expanding model size (the components of the info and result layers and additionally the quantity of layers), dataset size (the quantity of models in your preparation set), or preparing steps (the quantity of emphases over your dataset).
- Early Stopping: Early stopping includes observing the presentation of your model on an approval set during preparing and halting preparation when execution quits getting to the next level. Joined with bigger models, this approach can forestall overfitting and save computational assets.
- Optimization Techniques: Different advancement strategies can be applied to further develop execution and proficiency. These incorporate diagram advancement (which improves on calculations), down projecting (which diminishes accuracy of numbers to save memory), and quantization (which lessens accuracy considerably further for more memory saving and computational speedup), utilizing specific runtimes.

These are general ideas and their adequacy can change contingent upon the particular assignment and information. Continuously approve upgrades with a different approval set or cross-approval.

While these are general ideas and their viability can change contingent upon the particular errand and information, it's consistently critical to approve any upgrades with a different approval set or cross-approval. This guarantees that the model's presentation has truly improved and isn't simply fitting to the commotion in the preparation information.

Presently, making this a stride further, a high-level way to deal with upgrade execution is by consolidating various sorts of brain organizations. This frequently prompts further developed execution by utilizing the qualities of each model kind. In particular, LSTM,

CNN, and Transformer models can be interconnected in different ways to make half and half models that catch the best parts of each.

Consolidating various kinds of neural networks can frequently prompt better execution by utilizing the qualities of each model. This is the way LSTM, CNN, and Transformer models can be interconnected:

- CNN and LSTM: In stock prediction, a CNN can be utilized to distinguish examples or highlights inside a progression of verifiable stock costs, which are treated as a 1D picture. The LSTM can then utilize this data to figure out the fleeting conditions of these examples. For instance, the CNN could distinguish a specific example that frequently goes before an ascent in stock costs, and the LSTM could figure out how to foresee this ascent when it perceives this example.
- CNN and Transformer: A CNN can be utilized to separate spatial elements from different time-series information connected with stocks (e.g., opening cost, closing cost, volume). The Transformer model can then handle these highlights in their consecutive request to anticipate future stock costs. The self-attention system in the Transformer model permits it to gauge the significance of various time steps while making forecasts.
- LSTM and Transformer: A LSTM model can handle the time-series information of stock costs to figure out the fleeting conditions, while a Transformer model can deal with the global dependencies in the information utilizing its self-attention system. For example, a LSTM could catch the momentary patterns in a stock's cost, while a Transformer could recognize longer-term cycles or patterns that range across many time steps.

In this large number of blends, the models are regularly associated by taking care of the result of one model (e.g., CNN) as contribution to the following model (e.g., LSTM or Transformer). This permits the second model to make forecasts in light of the elements recognized by the principal model.

# Bibliography


GitHub (2013). Build software better, together. [online] GitHub. Available at: https://github.com.

Jupyter.org. (2019). Project Jupyter. [online] Available at: https://jupyter.org.

Kaeley, H., Qiao, Y. and Bagherzadeh, N. (n.d.) (2023). Support for Stock Trend Prediction Using Transformers and Sentiment Analysis. [online] Available at: https://arxiv.org/pdf/2305.14368.pdf

Keras (2019). Home - Keras Documentation. [online] Keras.io. Available at: https://keras.io.

Liu, J., Liu, X., Lin, H., Xu, B., Ren, Y., Diao, Y. and Yang, L. (n.d.). Transformer-Based Capsule Network For Stock Movements Prediction. [online] Available at: https://aclanthology.org/W19-5511.pdf.

Lu, W., Li, J., Li, Y., Sun, A. and Wang, J. (2020). A CNN-LSTM-Based Model to Forecast Stock Prices. [online] Complexity. Available at: https://www.hindawi.com/journals/complexity/2020/6622927/.

Mao, P. Linlu (2023). Analysis of Forecasting Stock Prices Using CNN Model. Academic Journal of Computing & Information Science, [online] 6(10). doi:https://doi.org/10.25236/AJCIS.2023.061012.

Matplotlib.org. (2012). Matplotlib: Python plotting — Matplotlib 3.1.1 documentation. [online] Available at: https://matplotlib.org.

Mehtab, S. and Sen, J. (2020). Stock Price Prediction Using CNN and LSTM-Based Deep Learning Models. 2020 International Conference on Decision Aid Sciences and Application (DASA), [online] pp. 447–453. doi:https://doi.org/10.1109/DASA51403.2020.9317207.

MICROSOFT (2016). Visual Studio Code. [online] Visualstudio.com. Available at: https://code.visualstudio.com.

Muhammad, T., Aftab, A.B., Ibrahim, M., Ahsan, Md.M., Muhu, M.M., Khan, S.I. and Alam, M.S. (2023). Transformer-Based Deep Learning Model for Stock Price Prediction: A Case Study on Bangladesh Stock Market. International Journal of Computational Intelligence and Applications. [online] doi:https://doi.org/10.1142/s146902682350013x.

Numpy.org. (2009). NumPy — NumPy. [online] Available at: https://numpy.org.

Pydata.org. (2012). seaborn: statistical data visualization — seaborn 0.9.0 documentation. [online] Available at: https://seabom.pydata.org.

Pydata.org. (2019). Python Data Analysis Library — pandas: Python Data Analysis Library. [online] Available at: https://pandas.pydata.org.

Python Software Foundation (2019). Welcome to Python.org. [online] Python.org. Available at: https://www.python.org.

scikit-learn (2019). scikit-learn: machine learning in Python. [online] Scikit-learn.org. Available at: https://scikit-learn.org/stable.

TensorFlow (2019). TensorFlow. [online] TensorFlow. Available at: https://www.tensorflow.org.

Wu, J.M.-T., Li, Z., Herencsar, N., Vo, B. and Lin, J.C.-W. (2021). A graph-based CNN-LSTM stock price prediction algorithm with leading indicators. Multimedia Systems.