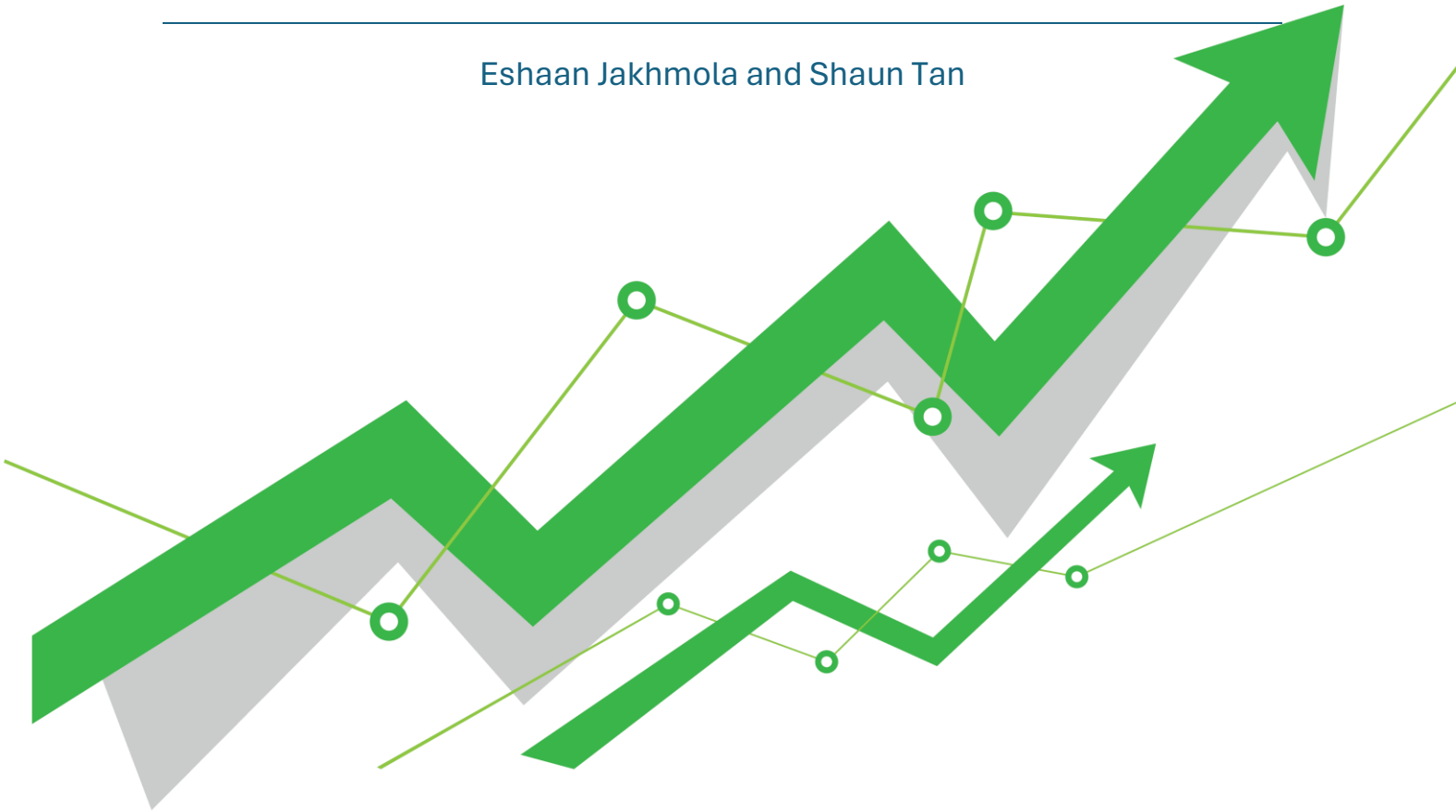
Eshaan 1006196 , Shaun 1006354, Project Group 28

# STOCK PREDICTION USING LSTM AND DRAWBACKS OF RNN

Eshaan Jakhmola and Shaun Tan



STATISTICS AND MACHINE LEARNING

# INDEX

# Motivation

One significant obstacle we faced in our earlier investigation into the use of Recurrent Neural Networks (RNNs) for stock price prediction was the vanishing gradient problem. This problem can cause errors in identifying long-term dependencies in sequential data, even if RNNs are quite successful at doing so. This could lead to inaccurate stock price predictions. We need to reduce the possibility of investment losses given the financial limitations faced by our student body.

Long Short-Term Memory (LSTM) networks are useful in this situation. LSTMs, an RNN variation, solve the vanishing gradient issue and provide better stock price prediction accuracy. We will go into further detail about the difficulties RNNs provide when predicting stocks in our upcoming study. We will also present the idea of LSTM-based prediction and look at its potential uses in financial forecasting. Through acquiring the necessary information and abilities to utilize LSTM networks, our goal is to enable SUTD students to confidently traverse the intricacies of financial markets and make well-informed investment decisions. Together, we can explore the possibilities of LSTM-based stock prediction and clear the path to financial gain.

# Context

We have studied neural networks throughout the semester, looking at both recurrent and convolutional neural networks (RNNs and CNNs). CNNs transformed image processing operations by providing unmatched pattern recognition capabilities. Nevertheless, we found that RNNs had limits when it came to processing sequential data, such time series analysis of stocks. RNNs are excellent at capturing temporal relationships, but they have trouble maintaining long-term dependencies that are necessary for precise stock forecasts and training on long-term data. The solution is provided by Long Short-Term Memory (LSTM) networks, a specialized version of RNNs.

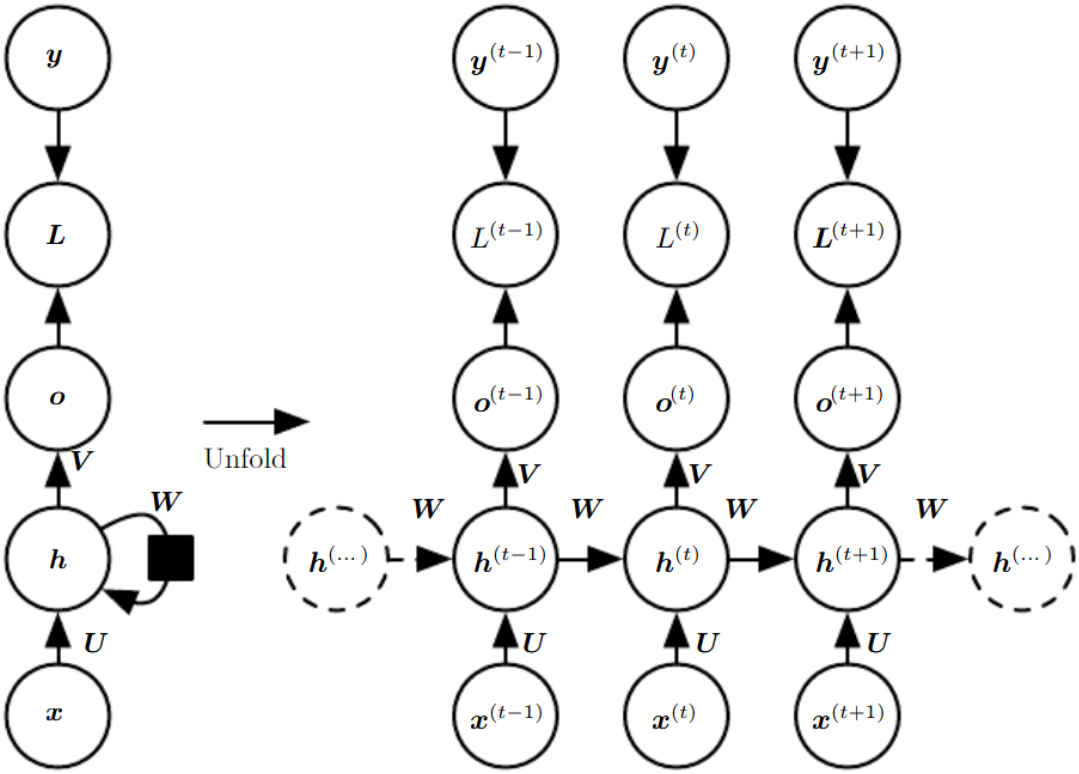
Since memory cells allow for the long-term preservation of significant information, long-term memory banks (LSTMs) are an essential tool for time series research, especially in the field of stock market forecasting. During our investigation into neural networks, we have seen how ANNs and CNNs evolved, and we now understand how important LSTMs are to processing sequential data, which opens the door to more precise forecasts and well-informed financial market decision-making.

# Recurrent Neural Networks (RNNs)

## Introduction

Consider attempting to guess a sentence's following word. Sounds easy, doesn't it? The problem is that words are not isolated entities; rather, they are a series in which one word influences another. Recurrent neural networks, or RNNs, can help with it. Think of them as memory-impaired language detectives who examine the full phrase, not just a single word at a time, while also recording the words that came before. For applications like language processing and speech recognition, RNNs' "memory" enables them to comprehend context. Comparable to computer-generated storytellers, they piece together the storyline word by word, using their prior knowledge to anticipate future events.

Architecture




Recurrent connections are the foundation of the RNN architecture, which allows it to describe short-term dependencies in sequential data. The RNN layer is mostly made up of a network of interconnected RNN cells, each of which represents a distinct time step in the sequence. These cells preserve a concealed state that contains data from earlier stages as they unroll in accordance with the predetermined amount of time steps. As the network changes, this hidden state persists, retaining background knowledge that is essential to comprehending the sequence. The capacity of the RNN's design to handle sequential data by recursively updating

its internal state is essentially what defines it. This allows it to capture temporal dependencies that are necessary for a variety of applications, including time series prediction and natural language processing.

Envision yourself plunging into the realm of TensorFlow, poised to discover the enchantment of Recurrent Neural Networks (RNNs)! Imagine that the RNN quickly snatches up your data as it comes in chunks. It first configures its memory, much like a blank canvas ready for an artistic interpretation. Next, it uses certain weights and biases for each step in the sequence to combine the current input with its prior memories. Consider it like to combining flavors in a smoothie! Following a brief exploration of an activation function (like to incorporating spices into the mixture), the RNN refreshes its memory for the present phase. It can quickly produce the output for that step thanks to the updated memory.

Every step in the cycle repeats this thrilling procedure, which has the rhythm of a dance routine. The big finale, the batch output, is formed when all of the phase outputs join together at last! And just like that, over time, the RNN learns to groove with the data flow.

## Forward Pass/Propagation

Consider yourself a financial analyst who makes predictions about stock values using a Recurrent Neural Network (RNN). It's similar to day by day, step-by-step analysis of previous stock data in the forward pass. The RNN processes each data point in the sequence, updating its internal state (like comprehending market circumstances), much how a stock price review can be used to identify trends. Just like projecting tomorrow's stock prices based on today's patterns, it advances by generating predictions for future stock prices based on this evolving understanding.

In the forward pass for RNN each time you take a step from t = 1 to t = $\tau$, the following update equations are applied:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)}, \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)}, \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)}),
\end{aligned}
$$

## Backward Pass/Propagation

Let's now examine the backward pass using our stock prediction case study as a lens. You evaluate your forecasts' accuracy in light of the actual stock prices after creating them. Should your forecasts prove to be inaccurate, you would reassess your research and modify your comprehension of market trends. Analogously, the RNN assesses its mispredictions against the real stock prices in its backward pass to determine where it went wrong. The RNN

learns from its mistakes and improves its forecasts for future stock values by going back and modifying its internal state (such as comprehending market patterns).

## Computing Gradient

Let's talk about computing gradients in the context of stock price prediction lastly. Assume that you are modifying the parameters of your prediction model, such as the relative importance of the various factors affecting stock prices. In essence, you may determine how sensitive your forecasts are to variations in these factors by computing gradients. This enables you to optimize your model and make it more predictively accurate. It's similar to modifying your prediction method in light of each factor's impact on the result, which aids in the improvement of your forecasts and the production of better investment choices.

Now, let's focus on the RNN's internal state (let's call it "memory") at each step. Say, for example, you're analysing stock prices for the past week. Each day's memory not only depends on that day's data but also on the predictions you made for the next day.

When you iterate back from tomorrow's prediction i.e. from t = $\tau$ (tomorrow's prediction) to t=1(todays actual stock price) then we use the equation

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^{\top} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \tag{10.20}$$

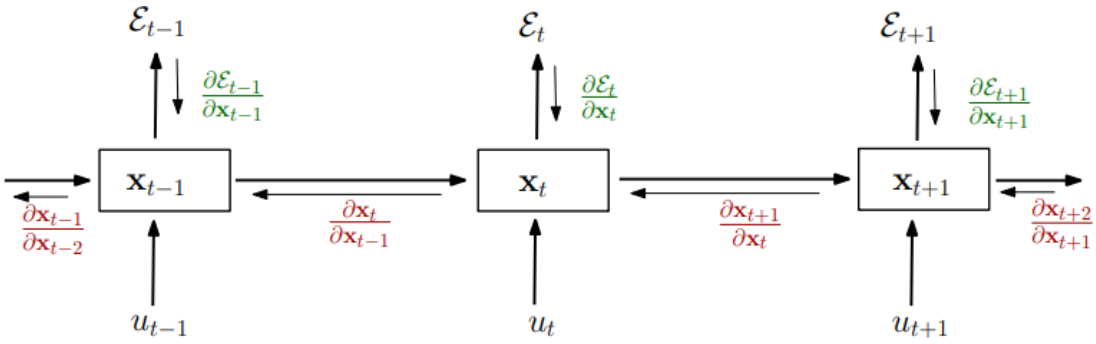
$$= \boldsymbol{W}^{\top} \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t+1)}} L\right) + \boldsymbol{V}^{\top} \left(\nabla_{\boldsymbol{o}^{(t)}} L\right), \tag{10.21}$$

So, as you backtrack from tomorrow's prediction to today's data, you're updating the memory at each step. This helps the RNN learn from its mistakes and improve its predictions for future stock prices.

This sounds great right, now that we finally understand RNN we can go and start predicting stock prices for our favourite` stock and get rich. But there is a small problem which is the vanishing and exploding of this computed gradient.

## Vanishing and Exploding Gradient Problem

The exploding gradients problem arises when there is a considerable increase in the gradient's norm during training, which frequently results in unstable learning dynamics. Long-term components in the gradient can develop exponentially, which can obscure shorter-term components and lead to this problem. The vanishing gradients problem, on the other hand, has the reverse behaviour, with long-term components rapidly decreasing to zero. In this case, the model's inability to properly train is hampered by its inability to grasp correlations between temporally distant events. The training of neural networks is severely hampered by both issues, which also affect the stability and convergence of the networks to optimal solutions.




$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \leq t \leq T} \frac{\partial \mathcal{E}_t}{\partial \theta} \tag{3}$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \leq k \leq t} \left( \frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \tag{4}$$

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \geq i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \geq i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \tag{5}$$

$W_{rec}$ *is the reccurent weight matrix*

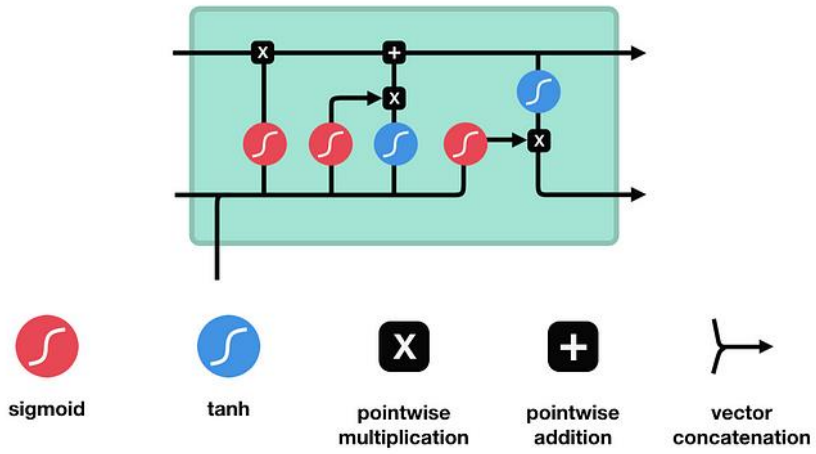
As $t \to \infty$ i.e for long-term components,

- If $W_{rec}$ gets smaller and $\rho < 1$, that is the spectral radius of $W_{rec}$ is lesser than 1 then the gradient vanishes.
- If $W_{rec}$ gets larger and $\rho > 1$, that is the spectral radius of $W_{rec}$ is greater than 1 then the gradient explodes.

The vanishing and exploding gradient difficulties are major obstacles when employing recurrent neural networks (RNNs) to predict stock values. For the model to accurately learn from past data, it is essential to capture subtle patterns and correlations across time, which is typically the case with stock price prediction. But the vanishing gradient issue may make it more difficult for the RNN to identify long-term relationships in the data, which makes it harder to forecast future prices with precision based on historical patterns. On the other hand, the exploding gradient problem has the potential to upset the training procedure, leading to unpredictable changes in the model's parameters and impeding the convergence of the solution to the best one. When employing RNNs, it is crucial to handle the vanishing and exploding gradient problems because they can lead to poor performance and incorrect predictions.Strategies such as gradient clipping and careful initialization of weights can help mitigate these issues and improve the robustness of the model's predictions.

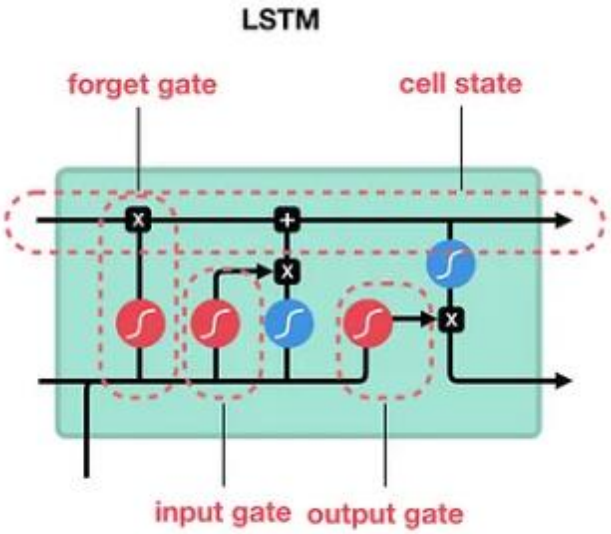
# LSTM

Within the field of recurrent neural networks (RNNs), Long Short-Term Memory (LSTM) networks are a customized architecture designed to overcome the shortcomings of conventional RNNs in terms of acquiring and storing consecutive information for prolonged periods of time. A key component of an LSTM is its unique cell state, which serves as a channel for the transfer of pertinent data throughout sequential processing. Acting as a sort of network "memory," this cell state makes it easier for information to go from earlier time steps to later ones, which helps to address the short-term memory problem.




Consider LSTM networks to be intelligent students with excellent recall. Their ability to retain crucial information over extended data sequences makes them ideal for jobs like stock price prediction. The cell state, which functions as the network's memory, is the central component of an LSTM network. As the network processes new data, its memory enables it to retain pertinent insights from previously processed data points.

Now to underst



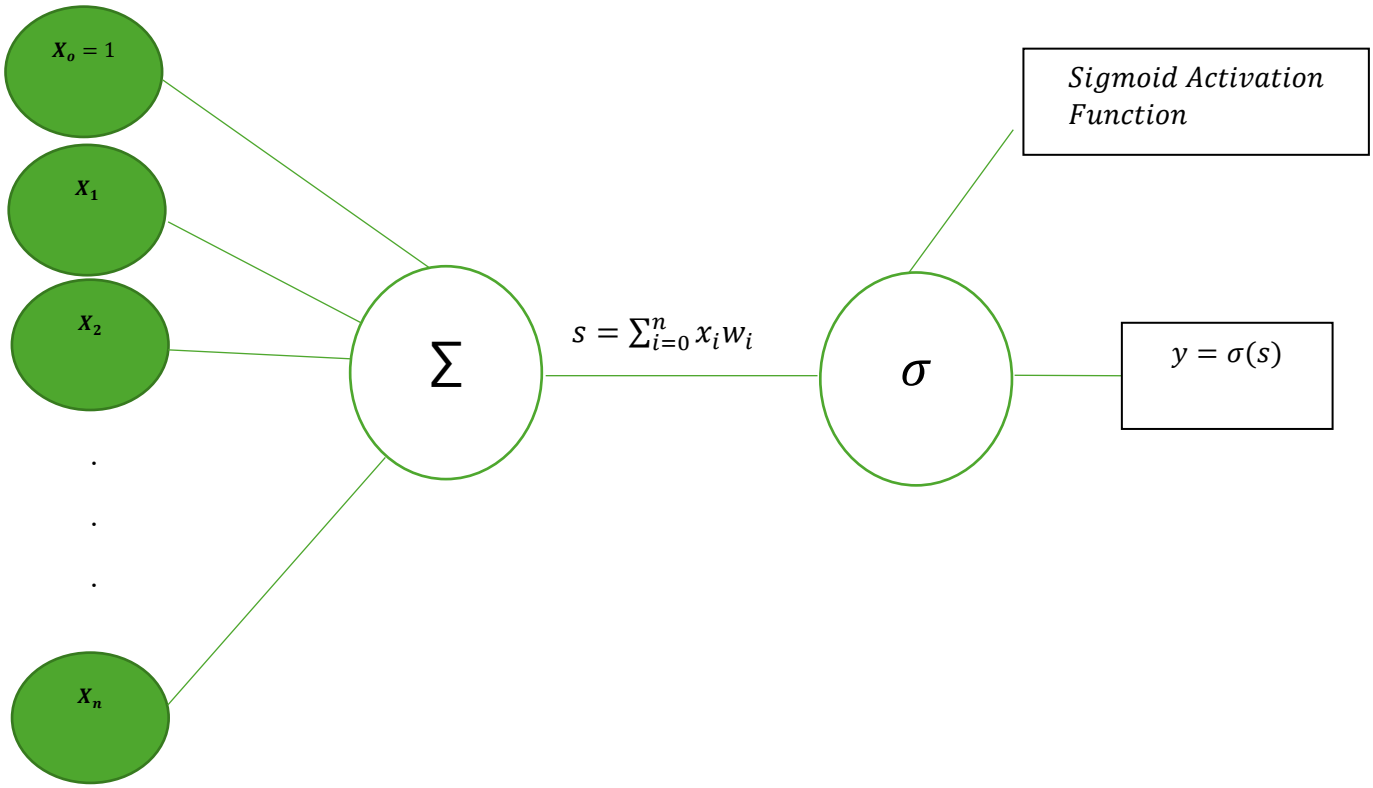

the mathemtical theory that goes behin

**Activation Functions**

There are 2 activation functions used in LSTM operation which are :

- Sigmoid:

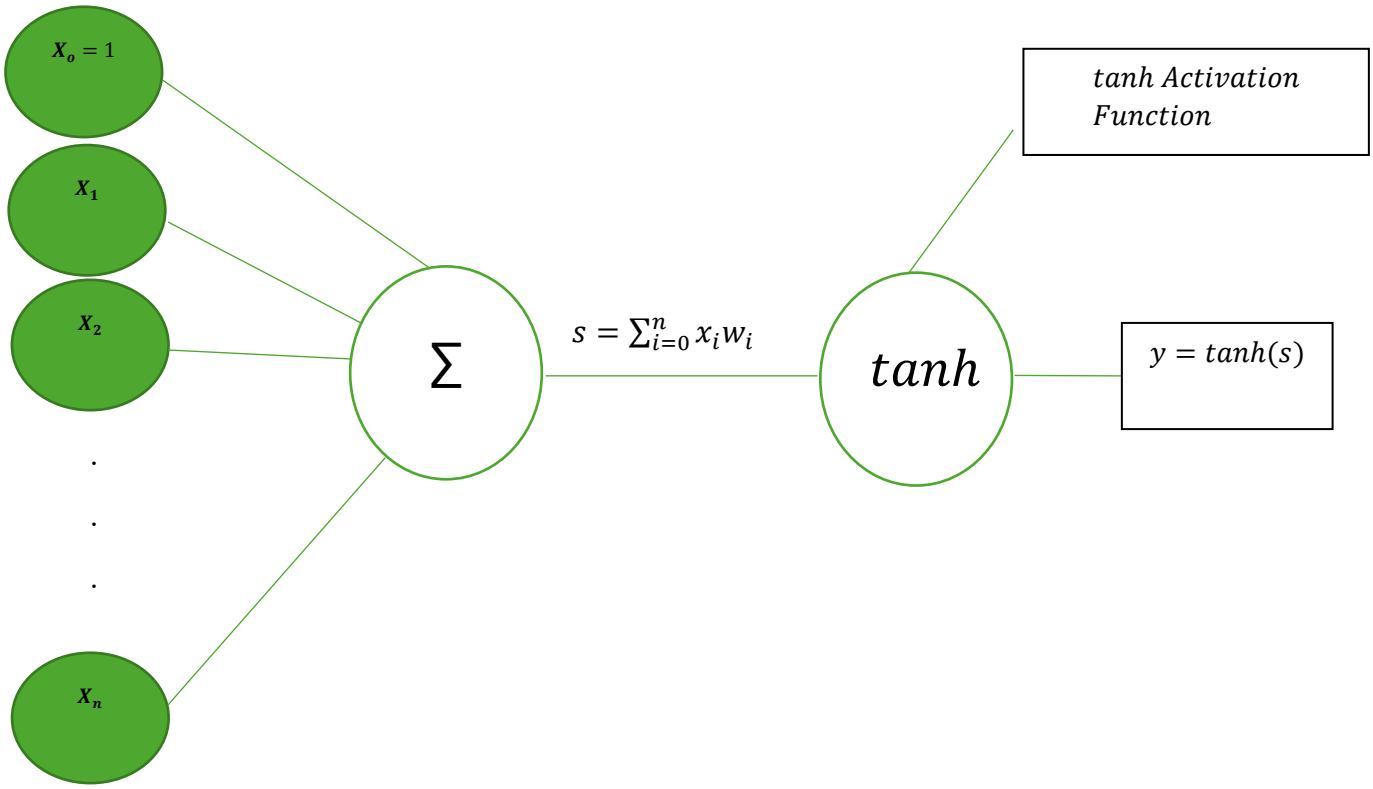
The sigmoid function is like a traffic light for information flow, assigning values between 0 and 1. It's handy for deciding yes or no outcomes, like whether to buy or sell a stock. We also use it in-between calculations to pass a percentage of information to the next layer.




- Tanh:

Think of tanh as a different kind of traffic light, but this time it assigns values between -1 and 1. Applying tanh helps us normalize data, making sure it's centered around zero. Because tanh's range is slightly larger than sigmoid, its gradients are more stable, which is great for learning in deeper layers. That's why we often use tanh for the hidden layers in LSTM networks, helping them learn and remember patterns effectively.

layer.




**Structure**

Within each cell, there are special gates responsible for controlling the flow of information. These gates include the

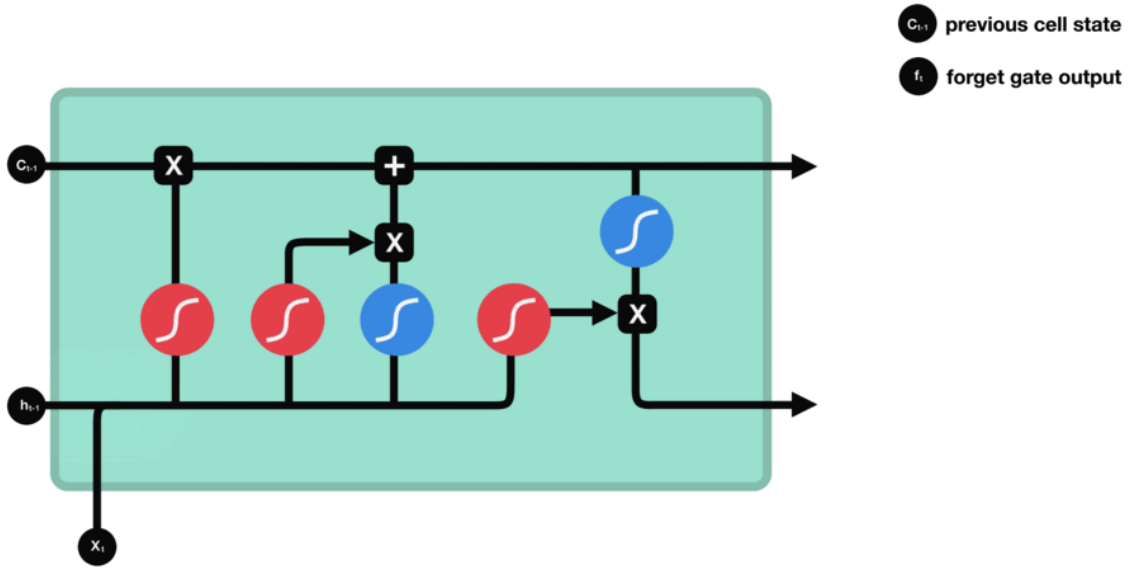
- forget gate,
- input gate,
- output gate

Each gate has its job, facilitated by sigmoid activations, which squish values between 0 and 1. Think of these activations as traffic lights directing the flow of information within the network. The forget gate decides which past information to discard, the input gate determines what new information to add to the memory, and the output gate regulates the information flow to produce the final prediction.

**Forget Gate**

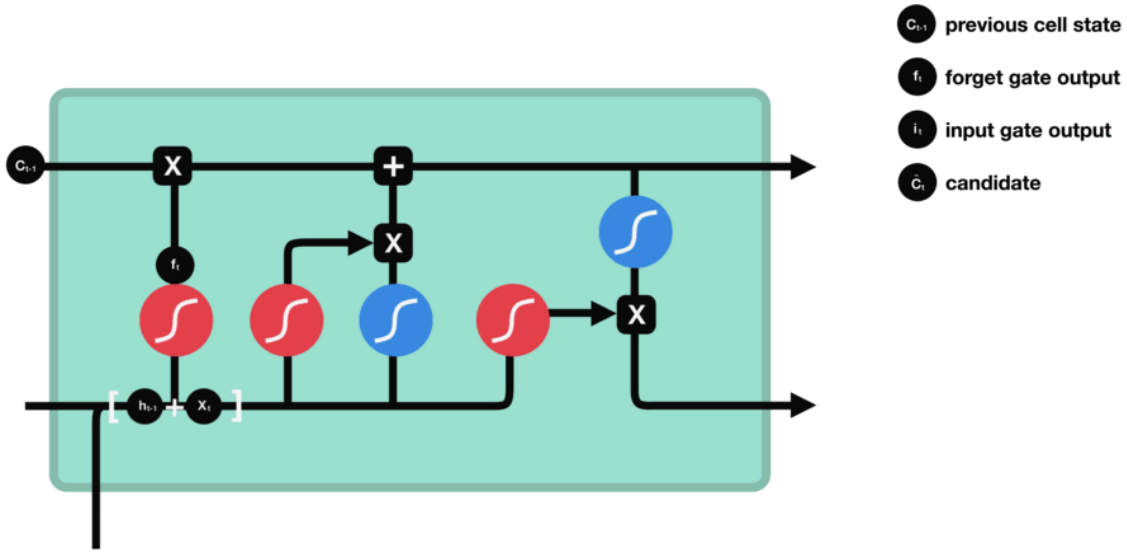
In LSTM networks, the forget gate determines which historical data, depending on the current input and historical data, should be remembered or ignored. A value closer to 0 indicates forgetting, and a value closer to 1 indicates remembering.

For instance, the gate allocates values closer to 1 to retain information if previous stock prices have a significant impact on future patterns. However, numbers closer to 0 are allocated to ignore historical prices if they are no longer relevant because of changes in the market. In order to make precise predictions, the forget gate aids the LSTM in concentrating on significant prior data.



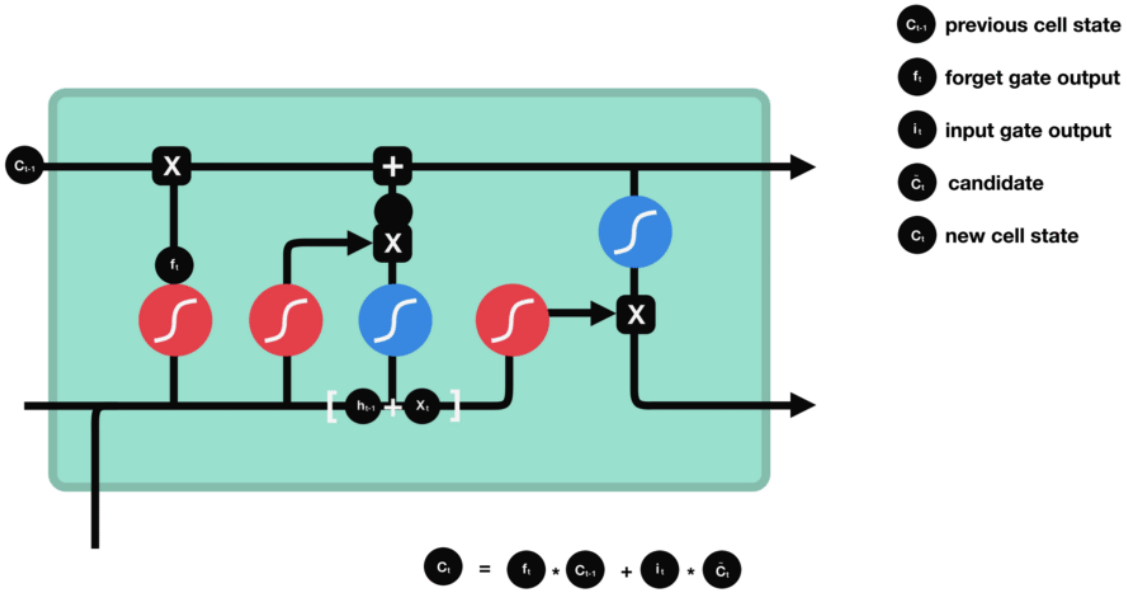
### Input Gate

The input gate determines which data from the current input and prior hidden state is essential to keep, which is how it updates the cell state. First, a sigmoid function is applied to the current input and the previous hidden state, allocating significance values ranging from 0 to 1. Concurrently, the tanh function compresses values between -1 and 1, which aids in network regulation. The outputs of tanh and sigmoid are then multiplied. To help update the cell state, the sigmoid output determines whether data from the tanh output is meaningful. In essence, the input gate lets the LSTM network add new data just when necessary while keeping pertinent insights for precise predictions.



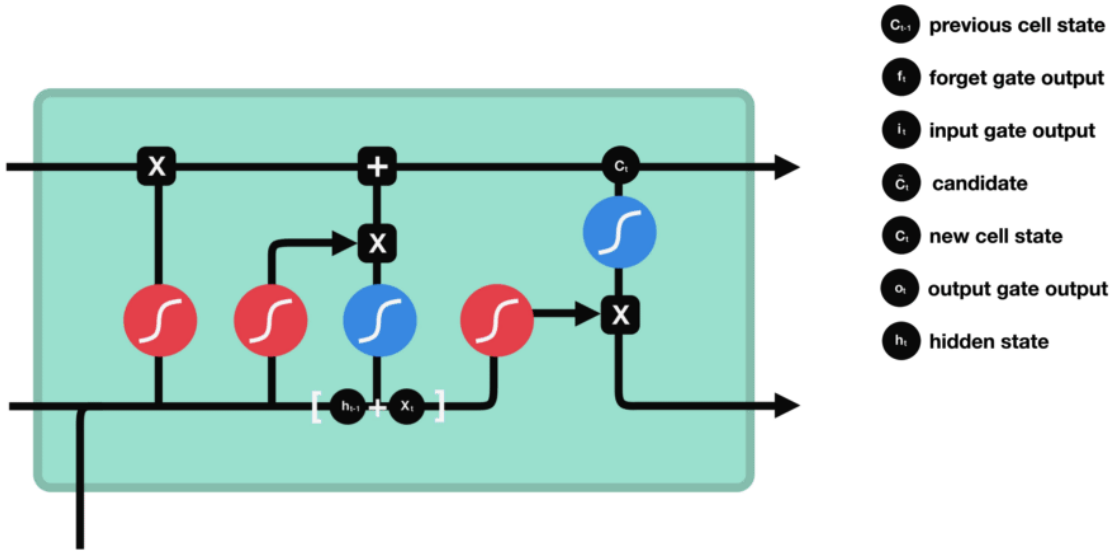

### Cell State

Consider the cell state as a valuable information conveyor belt that transports data across the network's processing. We use the forget gate to determine which information should be kept or discarded before updating this conveyor belt. A forget gate indicates that some information has been forgotten if it assigns values around 0. Next, we use the input gate to add fresh, pertinent data. By changing the cell state with values that the network deems significant, this gate contributes new information to it. To put it simply, the cell state changes with time, keeping important information while eliminating less important information. It functions as an ever-updating memory bank to assist the LSTM network in making defensible predictions based on inputs from the past and present.



**Ouput Gate**

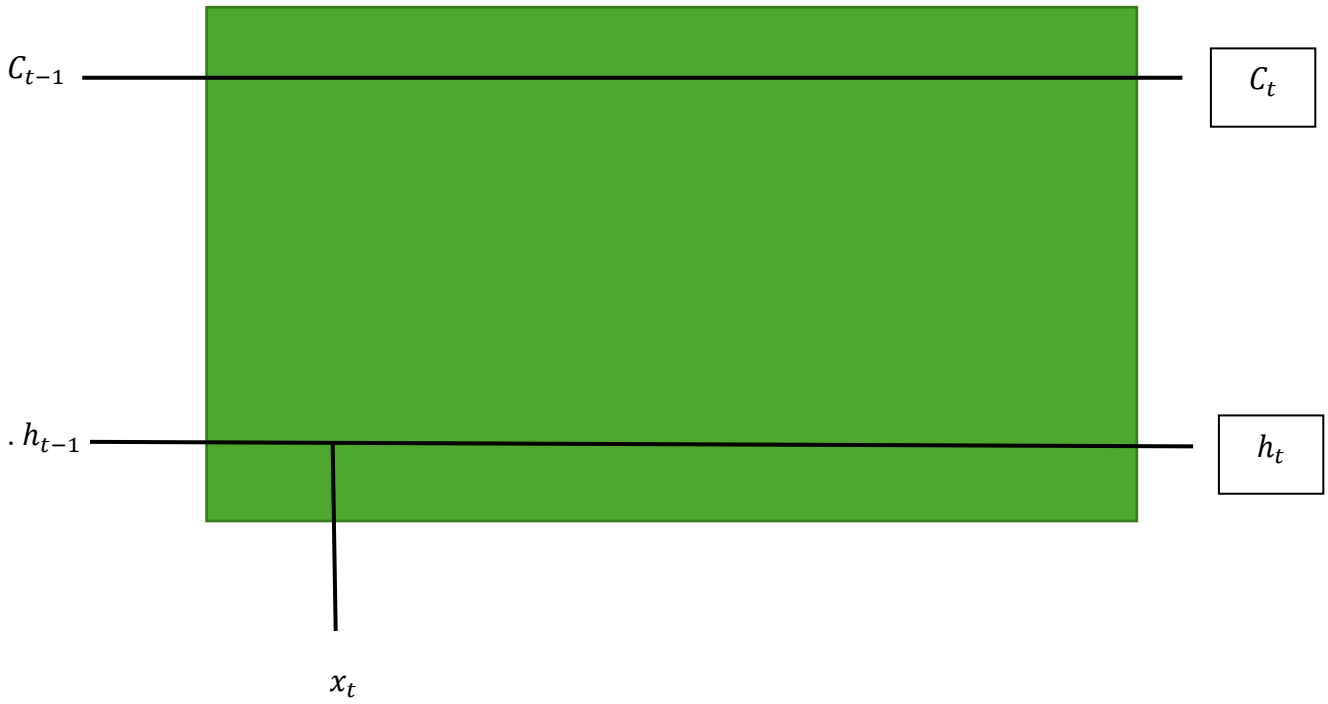
Think of it as the planner who determines which information moves on to the following phase. Predictions rely heavily on the hidden state, which is fundamentally composed of knowledge from earlier inputs. The output gate uses sigmoid activation to assess the relevance of both past and present information before processing any data. After that, the tanh function transforms the cell state. What information the hidden state should maintain is determined by multiplying the sigmoid output by the tanh output. In the end, the concealed state summarizes relevant information for upcoming forecasts. For LSTM-based tasks such as stock price prediction, this improved hidden state and the updated cell state serve as the basis for the analysis in the subsequent time step, guaranteeing consistency and precision.




**How does it operate?**

Consider the past stock prices as a series of data points, each of which corresponds to a distinct time period. The LSTM network utilizes its memory—the cell state—to hold onto important historical information while it processes this sequence.

The LSTM network gains the ability to identify patterns and trends by learning from the past data during training. The forget gate removes outdated, unnecessary data to aid the network in reducing noise. The input gate, meanwhile, enables the network to add fresh, pertinent data to its memory. This procedure of selection guarantees that the final prediction is influenced solely by significant insights. Ultimately, in order to generate a precise forecast of future stock values, the output gate regulates the information flow.




$C_{t-1} = Memory\ Cell\ state: Used\ for\ retaining\ information\ into\ its\ memory\ (long\ term\ info)$

$h_{t-1} = Hidden\ State: Responsible$

So in a simple run the network works as follows:

**Propose a new cell state:**

$\tilde{C}_t = \tanh(W_c * [h_{t-1}, x_t] + b_c)$

Where, $W_c = Matrix\ containing\ past\ information\ of\ c$

$b_c = Bias$

**Decide what to forget from the last cell state:**

$$The\ equation\ for\ the\ forget\ gate: f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f)$$

**Decide what to forget from new candidate cell state:**

$$The\ equation\ for\ the\ input\ gate: i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i$$

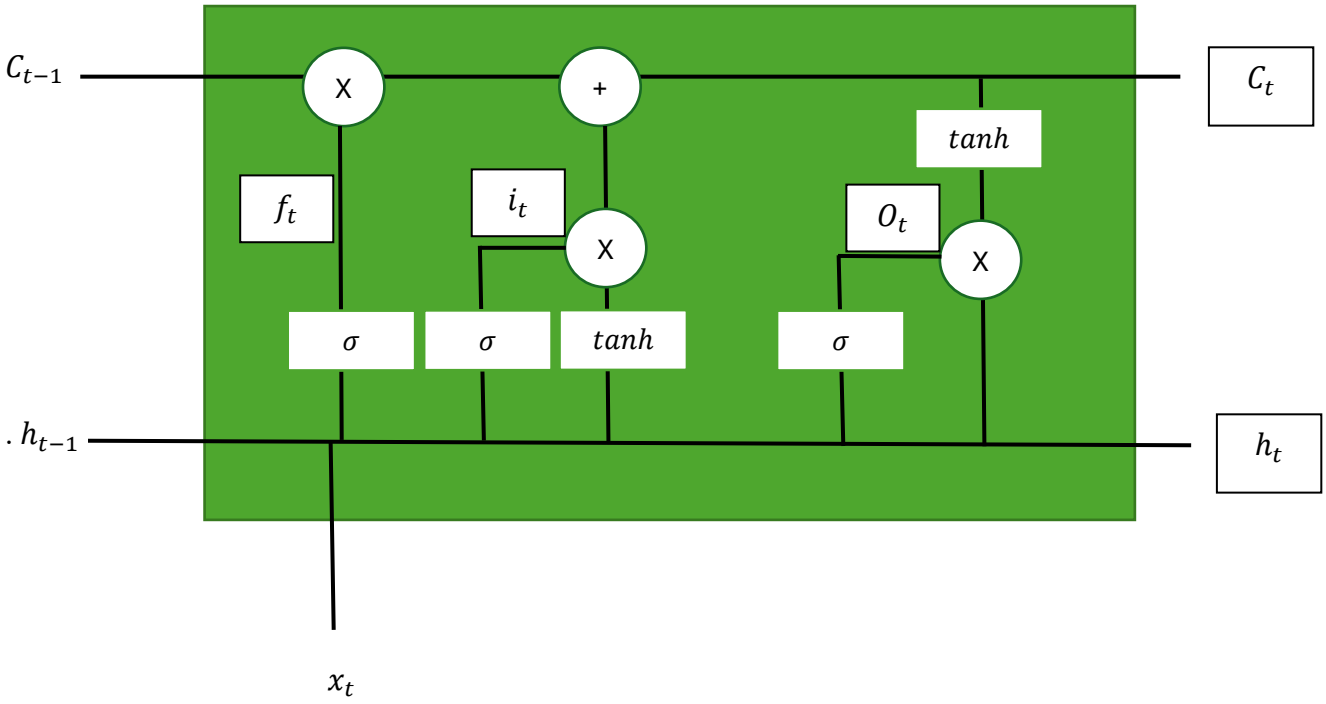
**Create New cell state:**

$$New\ Cell\ state: C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

**Decide what to use from $C_t$ for output:**

$$Equation\ of\ Ouput\ Gate:\ O_t = \sigma(W_o * [h_{t-1}, x_t] + b_o$$

So now our network looks more like this,




Then our final ouput looks something like

**Final Ouput:**

$$h_t = O_t * \tanh(C_t)$$

The process of updating or pruning the cell state is accomplished through the carefully controlled interaction of gates, which are made up of specialized neural networks that use sigmoid activations. These activations limit values to either 0 or 1, which allows for selective information to be retained or ignored. This allows the network to determine the relative importance of data points throughout the sequence. To put it simply, LSTM networks represent an advanced solution inside the RNN architecture that effectively handles temporal dependencies and improves the model's ability to understand long-range connections in consecutive data.

# Citations


Goodfellow, I., Bengio, Y., & Courville, A. (2017). *Deep learning CHAPTER 10. SEQUENCE MODELING: RECURRENT AND RECURSIVE NETS*. The MIT Press.

Phi, M. (2020, June 28). Illustrated Guide to LSTM's and GRU's: A step by step explanation. *Medium*. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

Bora, L. (2021, December 14). Simplified Math behind Complex LSTM equations - Leena Bora - Medium. *Medium*. https://medium.com/@leenabora1/simplified-math-behind-complex-lstm-equations-66cff0d52d78

Rastogi, M. (2021, December 14). Tutorial on LSTMs: A Computational Perspective - towards Data Science. *Medium*. https://towardsdatascience.com/tutorial-on-lstm-a-computational-perspective-f3417442c2cd

Jaiswal, A. (2024, February 26). *Tutorial on RNN | LSTM: with Implementation*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2022/01/tutorial-on-rnn-lstm-gru-with-implementation/

## 5 Application: LSTM Stock Price Prediction

```
#Below are all packages required for the implementation:

import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import numpy as np
from keras.models import Sequential
from keras.layers import LSTM, Dropout, Dense
import math
from sklearn.metrics import mean_squared_error
import yfinance as yf
from scipy.stats import linregress
import mplfinance as mpf
```

## 5.2 An Overview of the Dataset

Getting the data

The dataset was sourced from the Bloomberg terminal, known for its extensive financial market data crucial for identifying lucrative investment opportunities. Accessing this data was facilitated through the Data Analytics Lab's Bloomberg terminal, offering a streamlined method for downloading market data. The dataset, along with all requisite functions, is saved in a CSV file within the same directory as this document. Market data only from Advanced Micro Devices (AMD) was used.

```
import pandas as pd
df = pd.read_csv("AMD 20 YEAR PRICE DATA_1.csv", index_col='Date',
parse_dates=["Date"])

print(df.head())
print(df.tail())
             Open   High    Low   Last  Close    Volume   Turnover
Date
2004-01-04  15.10  15.11  14.77  14.86  14.90   8225700  122954900
2004-01-05  15.05  15.27  15.01  15.20  14.86   9157300  139011700
2004-01-06  15.21  15.82  15.05  15.61  15.20  14592400  226088900
2004-01-07  15.78  15.99  15.49  15.66  15.61  15331000  240806400
2004-01-08  15.95  16.00  15.59  15.93  15.66  12062900  190947700
               Open      High       Low    Last   Close    Volume
Turnover
Date
2024-04-05  168.100  172.6900  165.5800  170.42  165.83  66080261
11254820000
2024-04-08  168.200  171.6599  166.8200  169.90  170.42  43997546
7460862000
2024-04-09  170.210  171.6000  167.2900  170.78  169.90  42927985
7271022000
2024-04-10  166.710  169.7752  164.0000  167.14  170.78  59599958
9911220000
2024-04-11  167.545  170.9499  166.5492  170.50  167.14  48994531
8292987000
```

Content of dataset:

As shown above, the dataset contains 8 columns:

Date: the trading date, covering AMD's stock prices from 2004 to 2024.

Open: opening price at which the stock is traded during the regular trading date

High: the highest price at which the stock is traded during the regular trading date

Low: the lowest price at which the stock is traded during the regular trading date

Last: the final price at which a stock is traded during the standard trading session

Close: the last price at which the stock is traded at the end of the regular trading date

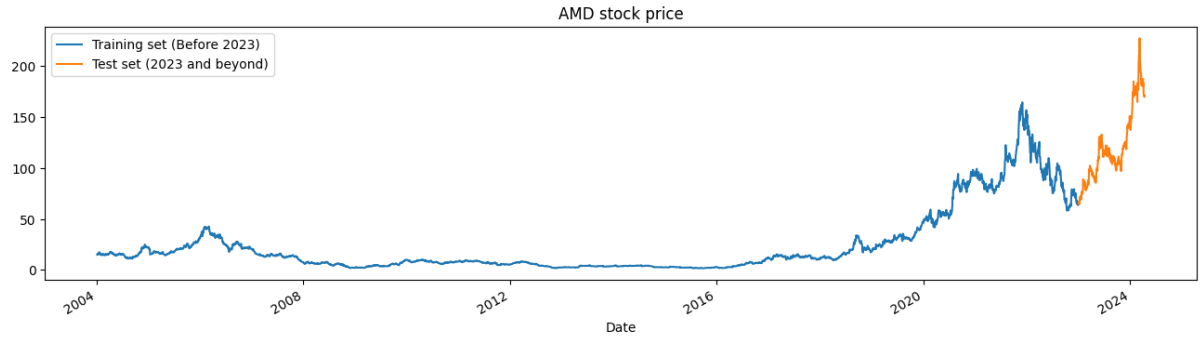
Total Trade Quantity / Volume: total number of shares traded for a particular security during regular trading date

Turnover: total value of securities traded during a specific period, often calculated by multiplying the average price of the security by the total number of shares traded. It provides a measure of the market activity and liquidity for a security or market as a whole.

Data Visualization

Let us visualize our dataset to understand the stock price trends. We'll plot the training set (before 2023) and the test set (2023 and beyond) separately.

```
import matplotlib.pyplot as plt
# Plot the training set
df["High"][:'2022'].plot(figsize=(16, 4), legend=True)
# Plot the test set
df["High"]['2023':].plot(figsize=(16, 4), legend=True)
plt.legend(['Training set (Before 2023)', 'Test set (2023 and beyond)'])
plt.title('AMD stock price')
plt.show()
```




Analysis: The graph indicates a stable trend in Highest prices for AMD from 2004 to 2018. In contrast, the highest prices from 2020 to 2022 are generally on the rise, potentially related to the demand of technology products caused bye COVID-19. The declining trend in the highest prices in 2023 may be associated with the dying down of the virus. The upward trend in 2024 can be associated with the demand for AI Chips.

## 5.3 Preprocessing of the dataset

Data Preprocessing

To ensure we have sufficient data for the time series prediction, we obtained a large dataset containing the price information of AMD, starting from 2004 till now. (~20 year)

Before feeding the data into our LSTM model, we need to preprocess it. We use Min-Max scaling to scale the stock prices to a range between 0 and 1.

```
from sklearn.preprocessing import MinMaxScaler

# here we are seperating the data
training_set = df[:'2022'].iloc[:,1:2].values
test_set = df['2023':].iloc[:,1:2].values

sc = MinMaxScaler(feature_range=(0, 1))
training_set_scaled = sc.fit_transform(training_set)
```

Creating the Training Data

We create our training data by creating sequences of stock prices and their corresponding labels. Each sequence will contain the stock prices of the previous 60 days.

```
X_train = []
y_train = []

for i in range(60, len(training_set_scaled)):
    X_train.append(training_set_scaled[i - 60:i, 0])
    y_train.append(training_set_scaled[i, 0])


X_train, y_train = np.array(X_train), np.array(y_train)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
```

## 5.4 Building the LSTM Models

time to build our LSTM model. We'll create a sequential model with multiple LSTM layers and dropout for regularization.

```
from keras.models import Sequential
from keras.layers import LSTM, Dropout, Dense

regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=100, return_sequences=True,
input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.3))

regressor.add(LSTM(units=80, return_sequences=True))
regressor.add(Dropout(0.1))

regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
```

```
regressor.add(LSTM(units=30))
regressor.add(Dropout(0.3))

regressor.add(Dense(units=1))


regressor.compile(optimizer='adam',loss='mean_squared_error')
WARNING:tensorflow:From
C:\Users\shaun\AppData\Local\Programs\Python\Python311\Lib\site-
packages\keras\src\backend.py:873: The name tf.get_default_graph is
deprecated. Please use tf.compat.v1.get_default_graph instead.

WARNING:tensorflow:From
C:\Users\shaun\AppData\Local\Programs\Python\Python311\Lib\site-
packages\keras\src\optimizers\__init__.py:309: The name tf.train.Optimizer
is deprecated. Please use tf.compat.v1.train.Optimizer instead.
```

## Training the Model

With our model architecture in place, train it on our prepared training data.

```
regressor.fit(X_train, y_train, epochs=50, batch_size=32)
Epoch 1/50
WARNING:tensorflow:From
C:\Users\shaun\AppData\Local\Programs\Python\Python311\Lib\site-
packages\keras\src\utils\tf_utils.py:492: The name
tf.ragged.RaggedTensorValue is deprecated. Please use
tf.compat.v1.ragged.RaggedTensorValue instead.

148/148 [==============================] - 35s 148ms/step - loss: 0.0040
Epoch 2/50
148/148 [==============================] - 20s 133ms/step - loss: 0.0018
Epoch 3/50
148/148 [==============================] - 20s 135ms/step - loss: 0.0019
Epoch 4/50
148/148 [==============================] - 19s 125ms/step - loss: 0.0014
Epoch 5/50
148/148 [==============================] - 17s 112ms/step - loss: 0.0016
Epoch 6/50
148/148 [==============================] - 15s 104ms/step - loss: 0.0013
Epoch 7/50
148/148 [==============================] - 20s 134ms/step - loss: 0.0011
Epoch 8/50
148/148 [==============================] - 20s 135ms/step - loss: 0.0012
Epoch 9/50
148/148 [==============================] - 18s 120ms/step - loss: 0.0012
Epoch 10/50
148/148 [==============================] - 21s 139ms/step - loss: 0.0011
Epoch 11/50
148/148 [==============================] - 20s 135ms/step - loss: 0.0010
Epoch 12/50
148/148 [==============================] - 21s 140ms/step - loss: 0.0011
Epoch 13/50
148/148 [==============================] - 22s 146ms/step - loss: 9.5758e-
04
Epoch 14/50
148/148 [==============================] - 20s 136ms/step - loss: 0.0011
Epoch 15/50
148/148 [==============================] - 21s 141ms/step - loss: 0.0013
```

```
Epoch 16/50
148/148 [==============================] - 21s 142ms/step - loss: 9.5338e-
04
Epoch 17/50
148/148 [==============================] - 19s 126ms/step - loss: 8.2401e-
04
Epoch 18/50
148/148 [==============================] - 20s 138ms/step - loss: 9.3514e-
04
Epoch 19/50
148/148 [==============================] - 20s 138ms/step - loss: 0.0010
Epoch 20/50
148/148 [==============================] - 20s 134ms/step - loss: 0.0010
Epoch 21/50
148/148 [==============================] - 17s 118ms/step - loss: 8.4090e-
04
Epoch 22/50
148/148 [==============================] - 19s 128ms/step - loss: 9.1249e-
04
Epoch 23/50
148/148 [==============================] - 17s 116ms/step - loss: 9.6958e-
04
Epoch 24/50
148/148 [==============================] - 18s 122ms/step - loss: 8.4530e-
04
Epoch 25/50
148/148 [==============================] - 18s 121ms/step - loss: 7.9801e-
04
Epoch 26/50
148/148 [==============================] - 17s 114ms/step - loss: 8.6500e-
04
Epoch 27/50
148/148 [==============================] - 18s 121ms/step - loss: 9.0519e-
04
Epoch 28/50
148/148 [==============================] - 17s 115ms/step - loss: 0.0010
Epoch 29/50
148/148 [==============================] - 17s 116ms/step - loss: 0.0011
Epoch 30/50
148/148 [==============================] - 18s 122ms/step - loss: 9.9999e-
04
Epoch 31/50
148/148 [==============================] - 15s 104ms/step - loss: 8.1431e-
04
Epoch 32/50
148/148 [==============================] - 17s 118ms/step - loss: 8.9097e-
04
Epoch 33/50
148/148 [==============================] - 16s 108ms/step - loss: 8.9673e-
04
Epoch 34/50
148/148 [==============================] - 19s 129ms/step - loss: 8.9259e-
04
Epoch 35/50
148/148 [==============================] - 17s 117ms/step - loss: 9.1549e-
04
Epoch 36/50
148/148 [==============================] - 16s 108ms/step - loss: 8.9117e-
04
Epoch 37/50
```

```
148/148 [==============================] - 16s 109ms/step - loss: 7.7988e-
04
Epoch 38/50
148/148 [==============================] - 16s 108ms/step - loss: 9.1197e-
04
Epoch 39/50
148/148 [==============================] - 16s 108ms/step - loss: 8.6057e-
04
Epoch 40/50
148/148 [==============================] - 17s 112ms/step - loss: 8.5666e-
04
Epoch 41/50
148/148 [==============================] - 17s 113ms/step - loss: 8.1934e-
04
Epoch 42/50
148/148 [==============================] - 17s 113ms/step - loss: 9.5968e-
04
Epoch 43/50
148/148 [==============================] - 16s 111ms/step - loss: 8.6064e-
04
Epoch 44/50
148/148 [==============================] - 16s 109ms/step - loss: 8.1230e-
04
Epoch 45/50
148/148 [==============================] - 16s 108ms/step - loss: 8.4280e-
04
Epoch 46/50
148/148 [==============================] - 16s 108ms/step - loss: 9.1365e-
04
Epoch 47/50
148/148 [==============================] - 16s 109ms/step - loss: 8.3821e-
04
Epoch 48/50
148/148 [==============================] - 16s 110ms/step - loss: 8.0131e-
04
Epoch 49/50
148/148 [==============================] - 16s 111ms/step - loss: 8.1997e-
04
Epoch 50/50
148/148 [==============================] - 16s 110ms/step - loss: 7.9607e-
04
<keras.src.callbacks.History at 0x2026830fe10>
```

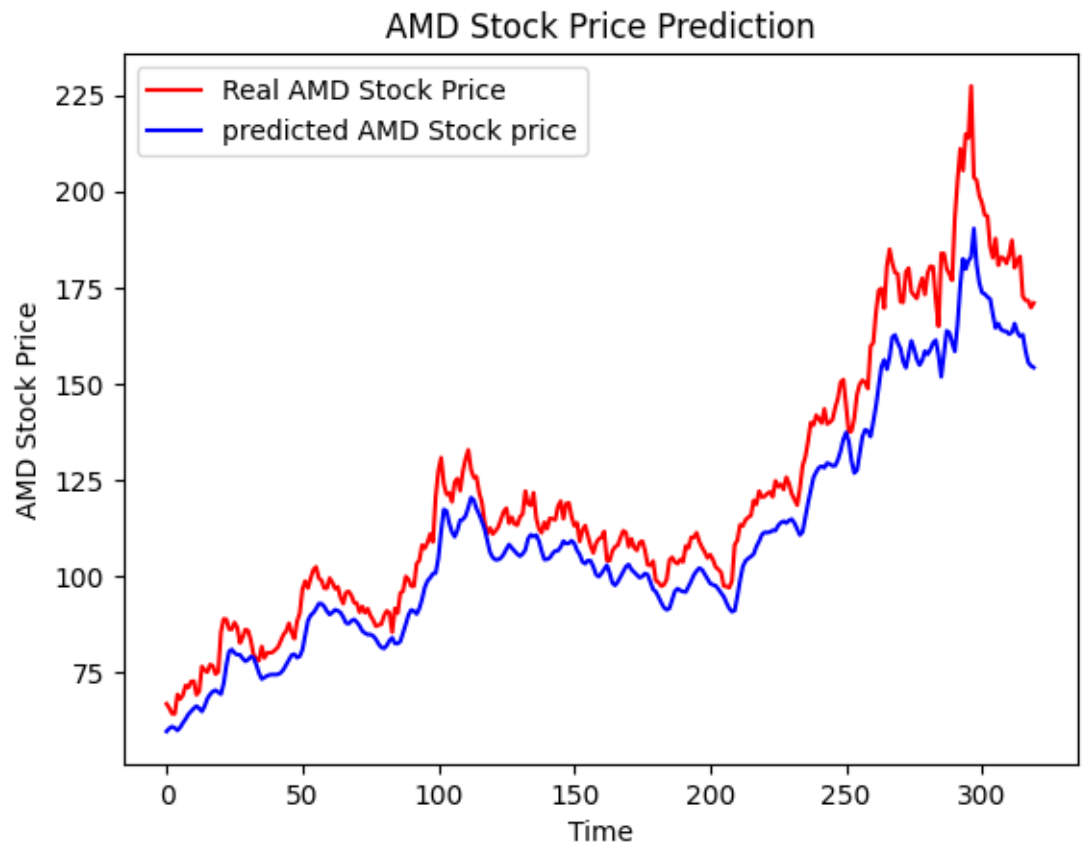
Now, we can make predictions on our test data.

```
# pre-processing the data
dataset_total = pd.concat((df["High"][:'2022'],df["High"]['2023':]),axis=0)
inputs = dataset_total[len(dataset_total)-len(test_set) - 60:].values
inputs = inputs.reshape(-1,1)
inputs  = sc.transform(inputs)

# making the test data
X_test = []
for i in range(60,len(inputs)):
  X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
10/10 [==============================] - 5s 55ms/step
```

## 5.5 Visualisation of the Predicted Results

visualize our model's predictions against the actual test data.

```
def plot_prediction(test,prediction):
    plt.plot(test,color='red',label="Real AMD Stock Price")
    plt.plot(prediction, color="blue",label="predicted AMD Stock price")
    plt.title("AMD Stock Price Prediction")
    plt.xlabel("Time")
    plt.ylabel("AMD Stock Price")
    plt.legend()
    plt.show()
# now we'll use this function to visualize our test and predicted data

plot_prediction(test_set,predicted_stock_price)
```



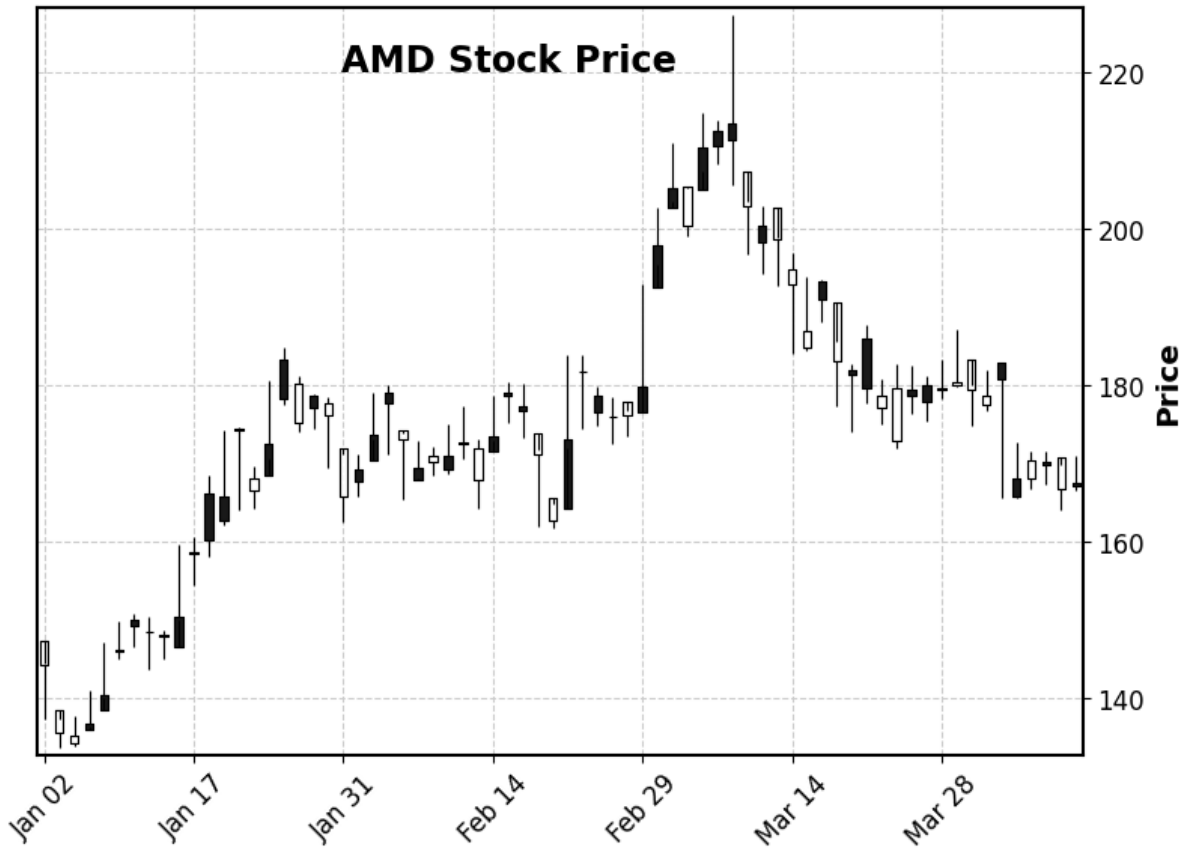

## 5.5 Evaluation

Evaluate our model's performance using the Root Mean Squared Error (RMSE).

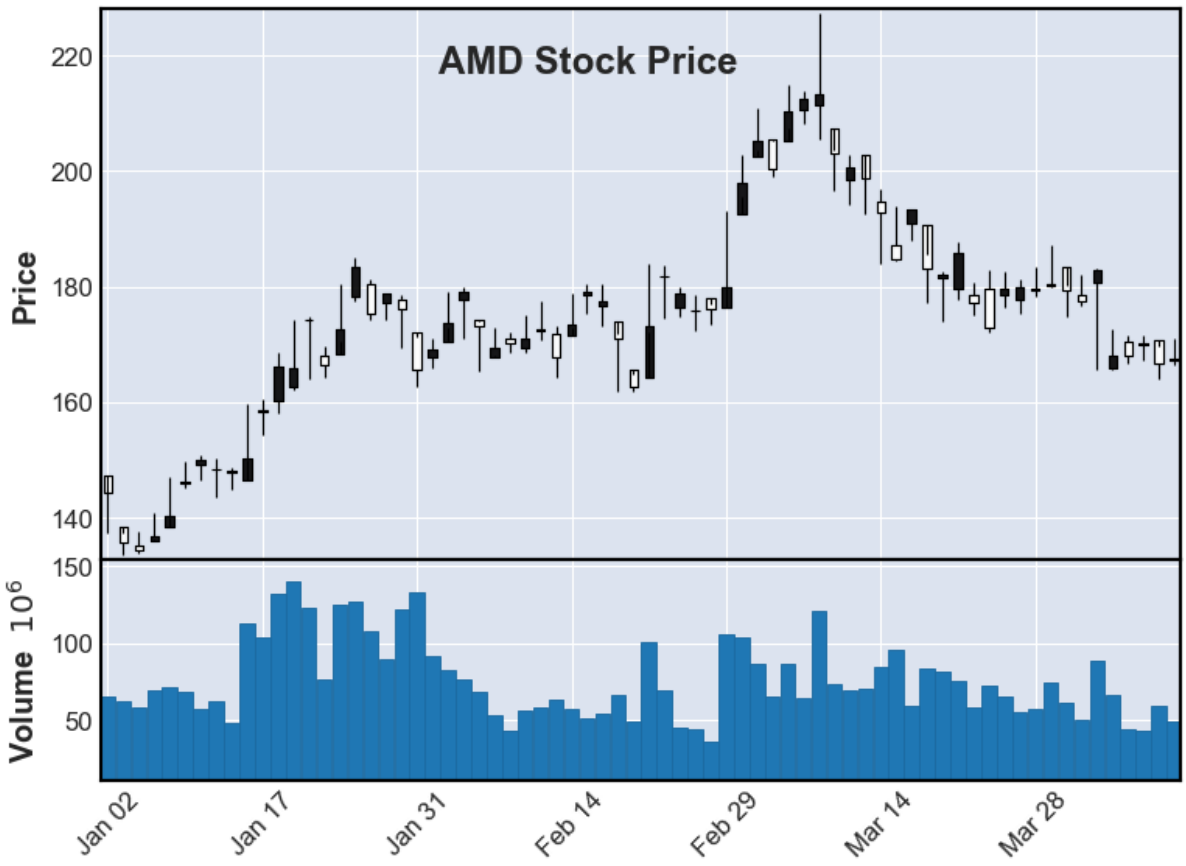
```
import math
from sklearn.metrics import mean_squared_error

def return_rmse(test, predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}.".format(rmse))

return_rmse(test_set, predicted_stock_price)
The root mean squared error is 12.703288787014554.
```

The RMSE of 7 to 13 provides context for evaluating the accuracy of the LSTM stock prediction model. While there isn't a universal benchmark for what constitutes a "good" RMSE, lower values generally indicate better predictive performance, suggesting that the model's predictions closely align with the actual stock values. In this case, an RMSE of 7.83 suggests a moderate level of prediction accuracy.

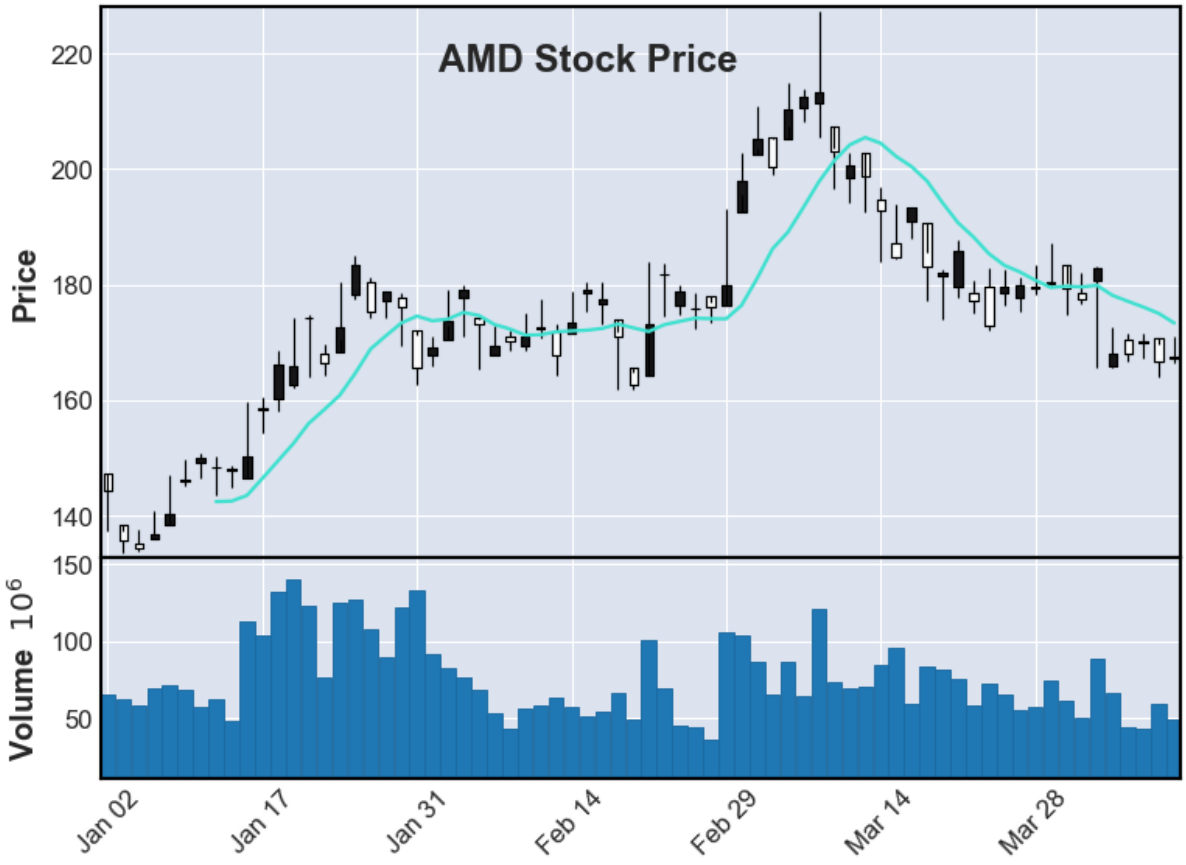
```
import mplfinance as mpf

import yfinance as yf
# candlestick

mpf.plot(df['2024-01-01':], type='candle', style = 'classic', title = 'AMD
Stock Price',tight_layout = True )
```




```
#Adding Volume Bars and Moving Averages

mpf.plot(df['2024-01-01':], type='candle', title = 'AMD Stock
Price',tight_layout = True, volume = True)
```




```
mpf.plot(df['2024-01-01':], type='candle', title = 'AMD Stock
Price',tight_layout = True, volume = True, mav = (8))
```



The provided code utilizes the mpf.plot() function to generate a candlestick chart depicting the stock price of AMD (Advanced Micro Devices) from January 1, 2024, onwards. Let's break down the parameters used:

df['2024-01-01':]: This specifies the data range to be plotted, starting from January 1, 2024, and extending to the end of the available data in the DataFrame df.

type='candle': Indicates that the chart type to be plotted is a candlestick chart, which provides visual representation of open, high, low, and close prices of the stock over the specified time period.

title='AMD Stock Price': Sets the title of the chart as "AMD Stock Price".

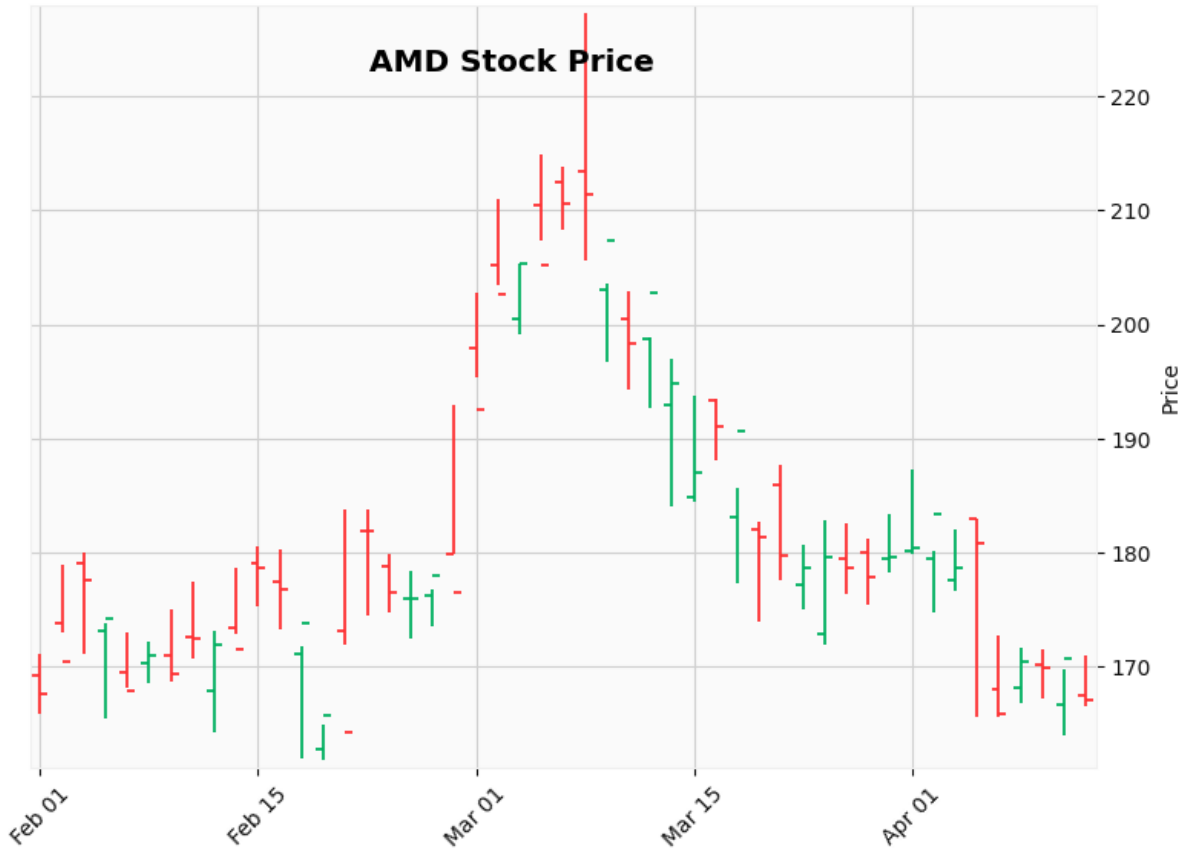
tight_layout=True: Ensures that the layout of the chart is optimized to prevent overlapping elements.

volume=True: Includes volume bars on the chart, depicting the trading volume of AMD stock over the specified time period.

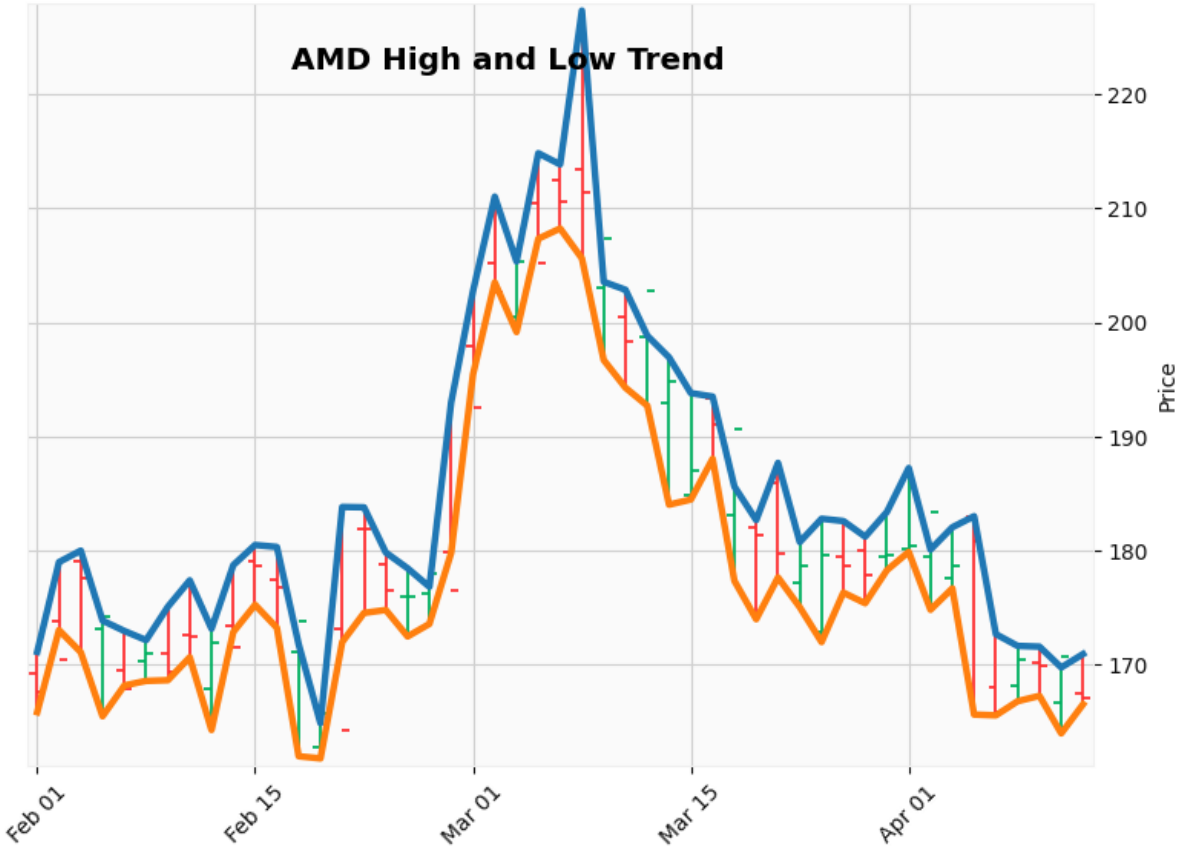
mav=(8): Adds a moving average line with a window of 8 periods to the chart, offering additional insight into the stock's price trend.

Description: The code generates a comprehensive visualization of AMD's stock price dynamics from January 1, 2024, onwards. It incorporates candlestick patterns to represent the open, high, low, and close prices of the stock for each trading day, enabling users to assess price movements and patterns. Additionally, the inclusion of volume bars provides information about the level of trading activity accompanying price changes, aiding in understanding market sentiment. The moving average line offers further analysis by smoothing out short-term fluctuations and highlighting long-term price trends.

```
#OHLC Chart

mpf.plot(df['2024-02-01':], style = 'yahoo', title='AMD Stock Price',
tight_layout = True )
```




```
high_low_lines = mpf.make_addplot(df.loc['2024-02-01':,['High','Low']])
mpf.plot(df["2024-02-01":], addplot=high_low_lines, title = 'AMD High and
Low Trend', style = 'yahoo', tight_layout = True)
```



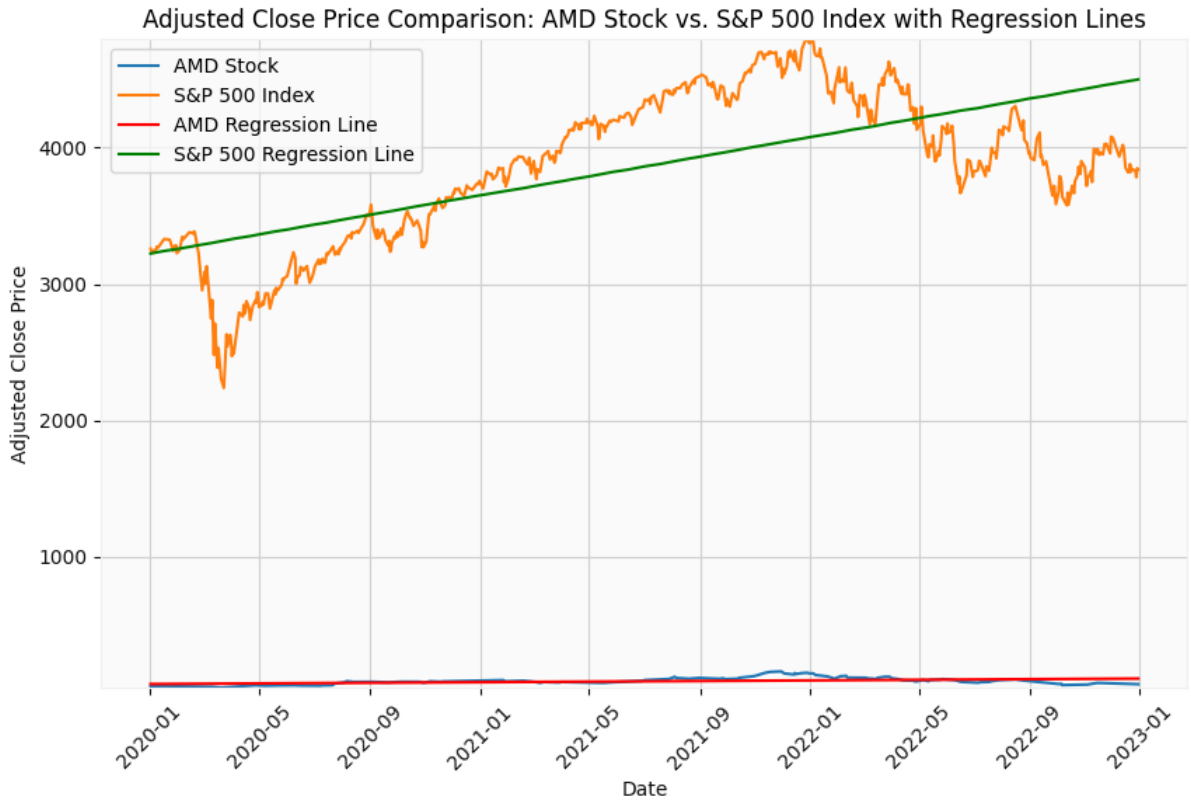

The provided code utilizes the mpf.make_addplot() function to create an additional plot showing the high and low prices of AMD stock from February 1, 2024, onwards. Let's delve into the description and conclusion:

Description: The code generates a candlestick chart representing AMD's stock price movement from February 1, 2024, onwards, utilizing the mpf.plot() function. Additionally, it includes an additional plot, created using mpf.make_addplot(), which specifically displays the high and low prices of the stock over the same time period. This supplementary plot provides a focused view of the price range within each trading day, offering insights into intraday volatility and price fluctuations. The style='yahoo' parameter ensures the chart adopts the Yahoo Finance style, enhancing readability and visual appeal. The tight_layout=True parameter optimizes the layout to prevent overlapping elements.

```
# Download historical data for AMD stock and S&P 500 index
amd_data = yf.download('AMD', start='2020-01-01', end='2023-01-01',
progress=False)
sp500_data = yf.download('^GSPC', start='2020-01-01', end='2023-01-01',
progress=False)

print(amd_data.head())
print(sp500_data.head())

# Extract adjusted close prices
amd_adj_close = amd_data['Adj Close']
sp500_adj_close = sp500_data['Adj Close']

# Perform linear regression
```

```
amd_slope, amd_intercept, _, _, _ = linregress(range(len(amd_adj_close)),
amd_adj_close)
sp500_slope, sp500_intercept, _, _, _ =
linregress(range(len(sp500_adj_close)), sp500_adj_close)

# Plot adjusted close prices with regression lines
plt.figure(figsize=(10, 6))

# Plot AMD Stock
plt.plot(amd_adj_close.index, amd_adj_close, label='AMD Stock')

# Plot S&P 500 Index
plt.plot(sp500_adj_close.index, sp500_adj_close, label='S&P 500 Index')

# Plot AMD Regression Line
plt.plot(amd_adj_close.index, amd_intercept +
amd_slope*np.arange(len(amd_adj_close)), color='red', label='AMD Regression
Line')

# Plot S&P 500 Regression Line
plt.plot(sp500_adj_close.index, sp500_intercept +
sp500_slope*np.arange(len(sp500_adj_close)), color='green', label='S&P 500
Regression Line')

# Set y-axis limits
min_price = min(amd_adj_close.min(), sp500_adj_close.min())
max_price = max(amd_adj_close.max(), sp500_adj_close.max())
plt.ylim(min_price, max_price)

plt.title('Adjusted Close Price Comparison: AMD Stock vs. S&P 500 Index
with Regression Lines')
plt.xlabel('Date')
plt.ylabel('Adjusted Close Price')
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.show()

print("AMD Regression Equation: AMD Price = {:.2f} + {:.2f} *
Time".format(amd_intercept, amd_slope))
print("S&P 500 Regression Equation: S&P 500 Price = {:.2f} + {:.2f} *
Time".format(sp500_intercept, sp500_slope))
                Open       High        Low      Close  Adj Close    Volume
Date
2020-01-02  46.860001  49.250000  46.630001  49.099998  49.099998  80331100
2020-01-03  48.029999  49.389999  47.540001  48.599998  48.599998  73127400
2020-01-06  48.020000  48.860001  47.860001  48.389999  48.389999  47934900
2020-01-07  49.349998  49.389999  48.040001  48.250000  48.250000  58061400
2020-01-08  47.849998  48.299999  47.139999  47.830002  47.830002  53767000
                  Open         High          Low        Close    Adj Close
\
Date
2020-01-02  3244.669922  3258.139893  3235.530029  3257.850098  3257.850098
2020-01-03  3226.360107  3246.149902  3222.340088  3234.850098  3234.850098
2020-01-06  3217.550049  3246.840088  3214.639893  3246.280029  3246.280029
2020-01-07  3241.860107  3244.909912  3232.429932  3237.179932  3237.179932
2020-01-08  3238.590088  3267.070068  3236.669922  3253.050049  3253.050049

                Volume
Date
2020-01-02  3459930000
```

```
2020-01-03  3484700000
2020-01-06  3702460000
2020-01-07  3435910000
2020-01-08  3726840000
```




```
AMD Regression Equation: AMD Price = 65.90 + 0.05 * Time
S&P 500 Regression Equation: S&P 500 Price = 3223.51 + 1.69 * Time
```

The provided Python code conducts a comparative analysis between the adjusted close prices of AMD stock and the S&P 500 index from January 1, 2020, to January 1, 2023. Let's delve into the description and conclusion:

Description:

Data Retrieval: The code utilizes the Yahoo Finance API (yfinance) to download historical data for AMD stock (AMD) and the S&P 500 index (^GSPC) from January 1, 2020, to January 1, 2023.

Linear Regression: After extracting the adjusted close prices for both AMD and the S&P 500, the code performs linear regression on each dataset. This calculates the slope and intercept of the regression lines, representing the overall trends in stock prices over the specified period.

Visualization: The code generates a plot using Matplotlib to visualize the adjusted close prices of AMD stock and the S&P 500 index, alongside their respective regression lines. This allows for a comparative analysis of the two assets' price movements and trends over time.

Regression Equations: Lastly, the code prints the regression equations for both AMD stock and the S&P 500 index, providing insight into the relationship between time and price for each asset.

In finance, the terms "alpha" and "beta" are commonly used to describe the performance of a stock relative to a market index, such as the S&P 500. Let's break down their meanings and implications based on the provided regression equations:

Alpha: Alpha represents the excess return of a stock or portfolio relative to the return of the overall market, adjusted for the risk taken. It measures the stock's performance after accounting for the market's movements. In the context of the regression equation for AMD stock, the intercept term (65.90) can be interpreted as the alpha.

Interpretation: An alpha value greater than zero implies that AMD stock has outperformed the market (S&P 500) on average during the analyzed period, after adjusting for market risk. Conversely, a negative alpha suggests underperformance compared to the market. Beta: Beta measures the sensitivity of a stock's returns to changes in the market index. It quantifies the systematic risk or volatility of a stock relative to the market. In the context of the regression equation for AMD stock, the coefficient of the time variable (0.05) represents the beta.

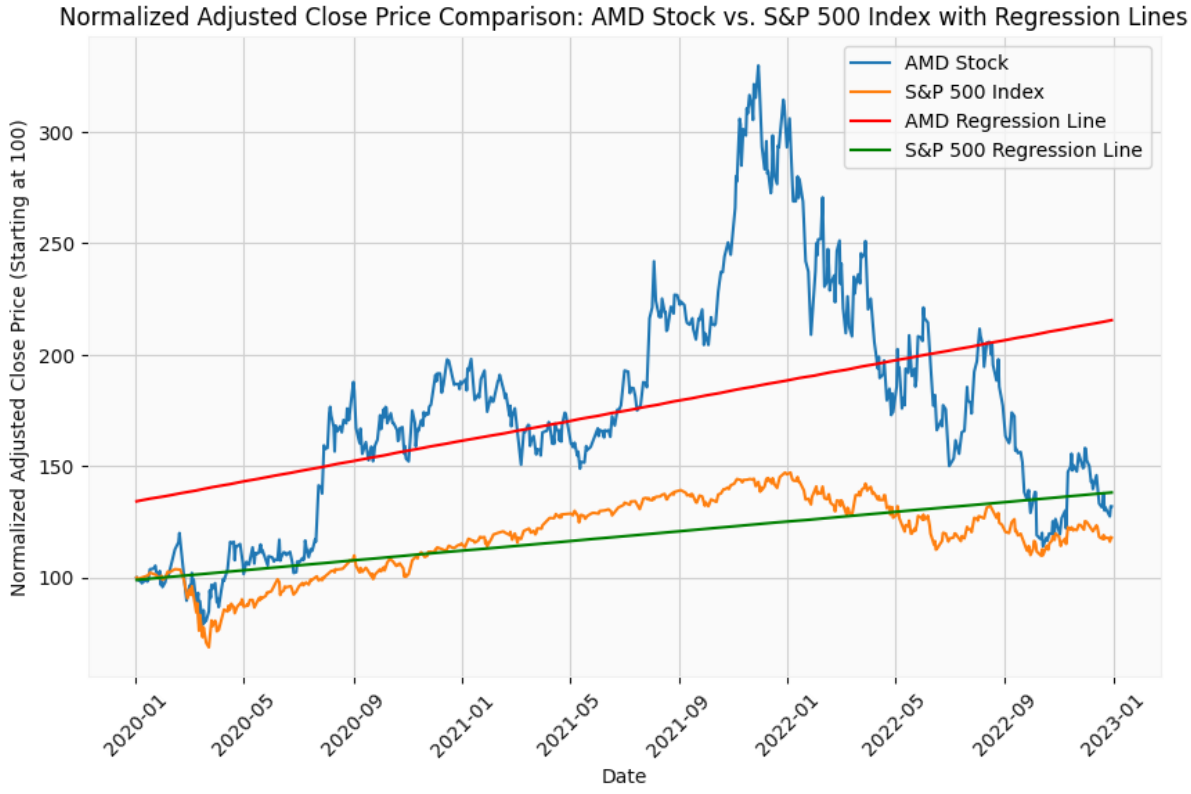
Interpretation: A beta greater than 1 indicates that AMD stock is more volatile than the market (S&P 500), meaning its price tends to move more than the market in the same direction. A beta less than 1 suggests lower volatility compared to the market, while a beta close to 1 implies similar volatility to the market.

Conclusion:

Alpha: With an alpha of 65.90, AMD stock has shown an excess return compared to the market (S&P 500) during the analyzed period, after adjusting for market risk. This suggests that AMD stock has outperformed the broader market, on average. Beta: With a beta of 0.05, AMD stock's price movements are less sensitive to changes in the market (S&P 500) compared to the average stock. This implies that AMD stock exhibits lower volatility or systematic risk relative to the overall market.

```
# Download historical data for AMD stock and S&P 500 index
amd_data = yf.download('AMD', start='2020-01-01', end='2023-01-01',
progress=False)
sp500_data = yf.download('^GSPC', start='2020-01-01', end='2023-01-01',
progress=False)

# Extract adjusted close prices
amd_adj_close = amd_data['Adj Close']
sp500_adj_close = sp500_data['Adj Close']

# Normalize prices
amd_adj_close_normalized = (amd_adj_close / amd_adj_close.iloc[0]) * 100
sp500_adj_close_normalized = (sp500_adj_close / sp500_adj_close.iloc[0]) *
100

# Perform linear regression on normalized prices
amd_slope, amd_intercept, _, _, _ =
linregress(range(len(amd_adj_close_normalized)), amd_adj_close_normalized)
sp500_slope, sp500_intercept, _, _, _ =
linregress(range(len(sp500_adj_close_normalized)),
sp500_adj_close_normalized)

# Plot adjusted close prices with regression lines
plt.figure(figsize=(10, 6))
```

```
# Plot normalized AMD Stock
plt.plot(amd_adj_close_normalized.index, amd_adj_close_normalized,
label='AMD Stock')

# Plot normalized S&P 500 Index
plt.plot(sp500_adj_close_normalized.index, sp500_adj_close_normalized,
label='S&P 500 Index')

# Plot AMD Regression Line
plt.plot(amd_adj_close_normalized.index, amd_intercept +
amd_slope*np.arange(len(amd_adj_close_normalized)), color='red', label='AMD
Regression Line')

# Plot S&P 500 Regression Line
plt.plot(sp500_adj_close_normalized.index, sp500_intercept +
sp500_slope*np.arange(len(sp500_adj_close_normalized)), color='green',
label='S&P 500 Regression Line')

plt.title('Normalized Adjusted Close Price Comparison: AMD Stock vs. S&P
500 Index with Regression Lines')
plt.xlabel('Date')
plt.ylabel('Normalized Adjusted Close Price (Starting at 100)')
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.show()

print("Normalized AMD Regression Equation: AMD Price = {:.2f} + {:.2f} *
Time".format(amd_intercept, amd_slope))
print("Normalized S&P 500 Regression Equation: S&P 500 Price = {:.2f} +
{:.2f} * Time".format(sp500_intercept, sp500_slope))
```




```
Normalized AMD Regression Equation: AMD Price = 134.21 + 0.11 * Time
Normalized S&P 500 Regression Equation: S&P 500 Price = 98.95 + 0.05 * Time
```

To normalize the prices of AMD stock and the S&P 500 index so that they are on the same scale, you can divide each price series by its respective initial value and then multiply by 100. This will make both series start at 100, allowing for a direct comparison of their relative performance over time.

Normalized AMD Regression Equation:

Alpha (α) = 134.21: Alpha represents the excess return of the asset (AMD) relative to the risk-free rate and the benchmark return (S&P 500), adjusted for its risk (beta). In this case, an alpha of 134.21 suggests that when the time variable is zero, the expected normalized AMD price is 134.21.

Interpretation: Even when the market and time factor are accounted for, AMD is expected to have an excess return compared to the benchmark and risk-free rate.

Beta (β) = 0.11: Beta measures the sensitivity of the asset's returns to changes in the returns of the benchmark (S&P 500). A beta greater than 1 indicates higher volatility compared to the benchmark, while a beta less than 1 indicates lower volatility. Interpretation: With a beta of 0.11, AMD is less volatile than the benchmark (S&P 500). For every unit increase in time, the normalized AMD price is expected to increase by 0.11 units. This suggests that AMD's price movements are positively correlated with the market but to a lesser extent.

Normalized S&P 500 Regression Equation:

Alpha (α) = 98.95: An alpha of 98.95 implies that when the time variable is zero, the expected normalized S&P 500 price is 98.95.

Interpretation: This suggests that, on average, the S&P 500 index is expected to provide a return of 98.95 when adjusted for the time factor.

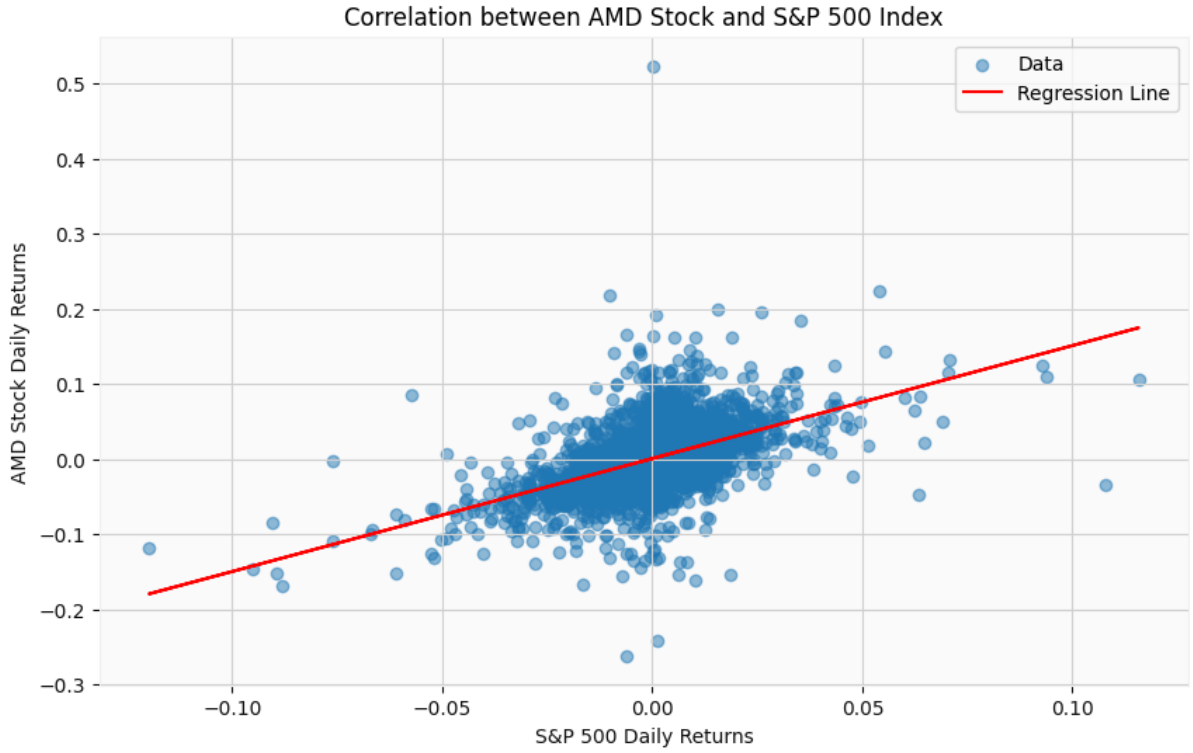
Beta (β) = 0.05: With a beta of 0.05, the S&P 500 index is less volatile compared to itself. For every unit increase in time, the normalized S&P 500 price is expected to increase by 0.05 units.

Interpretation: The S&P 500 index is relatively stable, and its price movements are less influenced by changes in time compared to AMD.

Conclusion: The alpha values indicate the expected excess returns (or underperformance) of the assets compared to the benchmark when the time variable is zero. The beta values represent the sensitivity of the assets' returns to changes in the benchmark returns. A beta greater than 1 suggests higher volatility, while a beta less than 1 indicates lower volatility. Based on the provided regression equations, AMD is expected to outperform the benchmark (S&P 500) when adjusted for risk (alpha > 0), but it is less volatile (beta < 1) compared to the S&P 500. Conversely, the S&P 500 index is relatively stable with lower expected returns and lower volatility.

```
# Download historical data for AMD stock and S&P 500 index
amd_data = yf.download('AMD', start='2004-01-01', end='2024-01-01',
progress=False)
sp500_data = yf.download('^GSPC', start='2004-01-01', end='2024-01-01',
progress=False)
```

```
# Extract adjusted close prices
amd_adj_close = amd_data['Adj Close']
sp500_adj_close = sp500_data['Adj Close']

# Calculate daily returns
amd_returns = amd_adj_close.pct_change().dropna()
sp500_returns = sp500_adj_close.pct_change().dropna()

# Compute correlation coefficient
correlation = amd_returns.corr(sp500_returns)

# Perform linear regression
slope, intercept, r_value, p_value, std_err = linregress(sp500_returns,
amd_returns)

# Plot scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(sp500_returns, amd_returns, alpha=0.5, label='Data')
plt.plot(sp500_returns, intercept + slope*sp500_returns, color='red',
label='Regression Line')
plt.title('Correlation between AMD Stock and S&P 500 Index')
plt.xlabel('S&P 500 Daily Returns')
plt.ylabel('AMD Stock Daily Returns')
plt.grid(True)
plt.legend()
plt.show()

print("Correlation coefficient between AMD Stock and S&P 500 Index:",
correlation)
print("Regression Equation: AMD Returns = {:.2f} + {:.2f} * S&P 500
Returns".format(intercept, slope))
```




```
Correlation coefficient between AMD Stock and S&P 500 Index:
0.5013001779165915
Regression Equation: AMD Returns = 0.00 + 1.50 * S&P 500 Returns
```

The correlation coefficient between AMD stock returns and the S&P 500 index returns is approximately 0.50, indicating a moderately positive correlation between the two. This suggests that there is a tendency for AMD stock returns to move in the same direction as the overall market represented by the S&P 500 index, but the relationship is not perfect.

The regression equation derived from the linear regression analysis suggests that for every 1% increase in the S&P 500 index returns, AMD stock returns increase by approximately 1.50%. This indicates that AMD stock tends to exhibit slightly higher volatility compared to the broader market, as reflected in the coefficient of 1.50.

Overall, based on the correlation coefficient and regression analysis:

There exists a moderate positive correlation between AMD stock returns and the S&P 500 index returns. AMD stock returns tend to move somewhat in line with the broader market, but they also demonstrate some independent variability. Investors in AMD stock may consider the performance of the S&P 500 index as a reference point, but they should also be aware of AMD's unique market dynamics and factors affecting its stock returns.

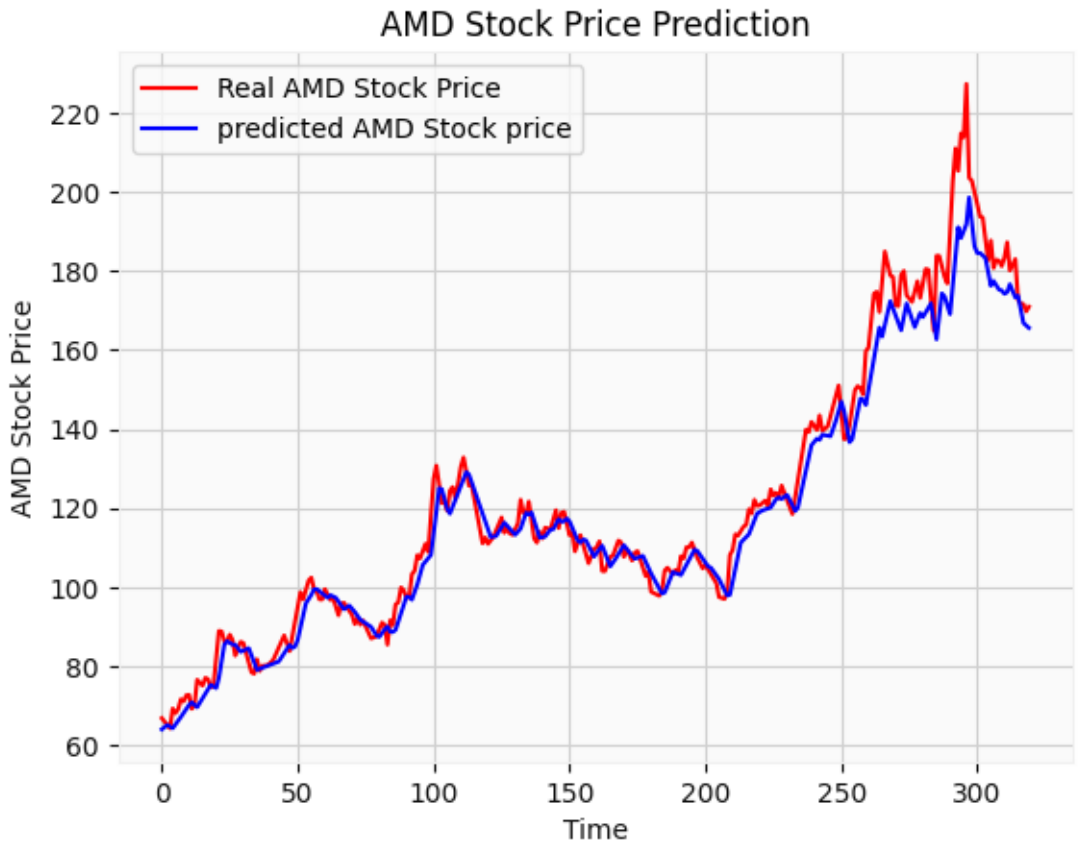
## Repeat Steps 5.1 to 5.5 to include correlation

```
# Assume you have calculated the correlation value
correlation_value = 0.5013001779165915 # Example correlation value

# Preprocessing the data
training_set = df[:'2022'].iloc[:, 1:2].values
test_set = df['2023':].iloc[:, 1:2].values

sc = MinMaxScaler(feature_range=(0, 1))
training_set_scaled = sc.fit_transform(training_set)

# Concatenate the correlation value with the training set
correlation_array = np.full((len(training_set_scaled), 1),
correlation_value)
training_set_scaled_with_corr = np.concatenate((training_set_scaled,
correlation_array), axis=1)

# Prepare the training data
X_train = []
y_train = []

for i in range(60, len(training_set_scaled_with_corr)):
    X_train.append(training_set_scaled_with_corr[i - 60:i, :])
    y_train.append(training_set_scaled_with_corr[i, 0])

X_train, y_train = np.array(X_train), np.array(y_train)

# Define the LSTM model
regressor = Sequential()
regressor.add(LSTM(units=100, return_sequences=True,
input_shape=(X_train.shape[1], X_train.shape[2])))
regressor.add(Dropout(0.3))
regressor.add(LSTM(units=80, return_sequences=True))
regressor.add(Dropout(0.1))
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
regressor.add(LSTM(units=30))
regressor.add(Dropout(0.3))
```

```
regressor.add(Dense(units=1))

# Compile the model
regressor.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
regressor.fit(X_train, y_train, epochs=50, batch_size=32)

# Pre-processing the test data
dataset_total = pd.concat((df["High"][:'2022'], df["High"]['2023':]),
axis=0)
inputs = dataset_total[len(dataset_total) - len(test_set) - 60:].values
inputs = inputs.reshape(-1, 1)
inputs = sc.transform(inputs)

# Prepare the test data
X_test = []
for i in range(60, len(inputs)):
    # Ensure both arrays have the same number of dimensions
    input_sequence = np.expand_dims(inputs[i - 60:i, 0], axis=1)  # Expand
dimensions to make it 2D
    correlation_array = np.full((60, 1), correlation_value)
    # Append the correlation value to each input
    combined_input = np.hstack((input_sequence, correlation_array))
    X_test.append(combined_input)
X_test = np.array(X_test)

# No need to reshape X_test since it already has the correct shape
# X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

# Make predictions
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price_corr = sc.inverse_transform(predicted_stock_price)
Epoch 1/50
148/148 [==============================] - 33s 109ms/step - loss: 0.0043
Epoch 2/50
148/148 [==============================] - 16s 109ms/step - loss: 0.0020
Epoch 3/50
148/148 [==============================] - 16s 107ms/step - loss: 0.0017
Epoch 4/50
148/148 [==============================] - 16s 110ms/step - loss: 0.0018
Epoch 5/50
148/148 [==============================] - 15s 102ms/step - loss: 0.0014
Epoch 6/50
148/148 [==============================] - 16s 108ms/step - loss: 0.0012
Epoch 7/50
148/148 [==============================] - 16s 109ms/step - loss: 0.0013
Epoch 8/50
148/148 [==============================] - 17s 114ms/step - loss: 0.0012
Epoch 9/50
148/148 [==============================] - 17s 115ms/step - loss: 0.0012
Epoch 10/50
148/148 [==============================] - 17s 113ms/step - loss: 0.0011
Epoch 11/50
148/148 [==============================] - 15s 102ms/step - loss: 0.0013
Epoch 12/50
148/148 [==============================] - 16s 107ms/step - loss: 0.0011
Epoch 13/50
148/148 [==============================] - 16s 107ms/step - loss: 0.0011
Epoch 14/50
148/148 [==============================] - 16s 108ms/step - loss: 0.0011
```

```
Epoch 15/50
148/148 [==============================] - 16s 107ms/step - loss: 0.0011
Epoch 16/50
148/148 [==============================] - 16s 106ms/step - loss: 0.0010
Epoch 17/50
148/148 [==============================] - 16s 107ms/step - loss: 0.0010
Epoch 18/50
148/148 [==============================] - 16s 106ms/step - loss: 0.0011
Epoch 19/50
148/148 [==============================] - 16s 106ms/step - loss: 9.7074e-
04
Epoch 20/50
148/148 [==============================] - 16s 106ms/step - loss: 8.8841e-
04
Epoch 21/50
148/148 [==============================] - 16s 107ms/step - loss: 0.0011
Epoch 22/50
148/148 [==============================] - 16s 109ms/step - loss: 9.8063e-
04
Epoch 23/50
148/148 [==============================] - 16s 109ms/step - loss: 8.2750e-
04
Epoch 24/50
148/148 [==============================] - 16s 108ms/step - loss: 9.3630e-
04
Epoch 25/50
148/148 [==============================] - 16s 108ms/step - loss: 9.7695e-
04
Epoch 26/50
148/148 [==============================] - 16s 108ms/step - loss: 9.3384e-
04
Epoch 27/50
148/148 [==============================] - 16s 108ms/step - loss: 8.5453e-
04
Epoch 28/50
148/148 [==============================] - 16s 107ms/step - loss: 0.0010
Epoch 29/50
148/148 [==============================] - 16s 108ms/step - loss: 9.7220e-
04
Epoch 30/50
148/148 [==============================] - 16s 108ms/step - loss: 0.0010
Epoch 31/50
148/148 [==============================] - 16s 110ms/step - loss: 8.9177e-
04
Epoch 32/50
148/148 [==============================] - 16s 111ms/step - loss: 8.9063e-
04
Epoch 33/50
148/148 [==============================] - 16s 110ms/step - loss: 8.7289e-
04
Epoch 34/50
148/148 [==============================] - 17s 112ms/step - loss: 9.1318e-
04
Epoch 35/50
148/148 [==============================] - 17s 118ms/step - loss: 9.2305e-
04
Epoch 36/50
148/148 [==============================] - 17s 113ms/step - loss: 8.1803e-
04
Epoch 37/50
```

```
148/148 [==============================] - 16s 110ms/step - loss: 9.0229e-
04
Epoch 38/50
148/148 [==============================] - 16s 111ms/step - loss: 8.6794e-
04
Epoch 39/50
148/148 [==============================] - 16s 111ms/step - loss: 8.4833e-
04
Epoch 40/50
148/148 [==============================] - 17s 114ms/step - loss: 8.9167e-
04
Epoch 41/50
148/148 [==============================] - 17s 112ms/step - loss: 8.7696e-
04
Epoch 42/50
148/148 [==============================] - 16s 110ms/step - loss: 8.7595e-
04
Epoch 43/50
148/148 [==============================] - 16s 109ms/step - loss: 9.0236e-
04
Epoch 44/50
148/148 [==============================] - 16s 110ms/step - loss: 8.4378e-
04
Epoch 45/50
148/148 [==============================] - 16s 109ms/step - loss: 8.4476e-
04
Epoch 46/50
148/148 [==============================] - 17s 113ms/step - loss: 9.0062e-
04
Epoch 47/50
148/148 [==============================] - 17s 113ms/step - loss: 9.6813e-
04
Epoch 48/50
148/148 [==============================] - 16s 108ms/step - loss: 9.3959e-
04
Epoch 49/50
148/148 [==============================] - 16s 107ms/step - loss: 8.1745e-
04
Epoch 50/50
148/148 [==============================] - 16s 106ms/step - loss: 8.3824e-
04
10/10 [==============================] - 5s 46ms/step
def plot_prediction(test,prediction):
    plt.plot(test,color='red',label="Real AMD Stock Price")
    plt.plot(prediction, color="blue",label="predicted AMD Stock price")
    plt.title("AMD Stock Price Prediction")
    plt.xlabel("Time")
    plt.ylabel("AMD Stock Price")
    plt.legend()
    plt.show()
# now we'll use this function to visualize our test and predicted data

plot_prediction(test_set,predicted_stock_price_corr)
```




```
def return_rmse(test, predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}.".format(rmse))

return_rmse(test_set, predicted_stock_price_corr)
The root mean squared error is 6.5062292918784665.
```

An RMSE of 6 to 10 indicates a moderate level of prediction accuracy. While it suggests that the model captures some of the variability in the actual stock prices, there is still improvement compared to the prediction without correlation to SP500.