SRH University Heidelberg

Time Series Forecasting Using Recurrent Neural Network

Author:	Birwadkar, Rahul
Matriculation Number:	11037364
Address:	Bonhoefferstraße 13 69123 Heidelberg Germany
Email Address:	Rahul.VijayBirwadkar@ stud.hochschule-heidelberg.de
Supervisor:	Prof. Dr.-Ing. Milan Gnjatović
Begin:	26. August 2024
End:	30. September 2024

Abstract

In recent years, the increasing availability of historical data across various fields has highlighted the importance of understanding trends, patterns, and demands to accurately predict future outcomes. Time series data, consisting of sequential historical measurements indexed by time, plays a crucial role in forecasting applications, allowing for better decision-making and enhanced productivity across domains such as medicine, agriculture, and finance. Predictive models that analyze historical data can offer insights into future behavior, enabling proactive strategies that can mitigate risks and optimize performance.

In the financial sector, for instance, stock market prices exhibit fluctuations daily, making time series forecasting critical for guiding investment decisions. The ability to anticipate stock price movements based on historical trends provides investors with a strategic advantage, as it informs more precise, data-driven approaches to portfolio management. This project focuses on applying time series forecasting techniques to predict future trends in stock prices, using historical data from the stock market as a case study.

Time series data is challenging because it follows a natural order over time. Unlike regular data, where each entry stands alone, time series data is connected, with each value depending on the ones that came before it. These changing patterns over time require special models that can understand and learn from these time-based relationships. In this project, we employ Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) networks, known for their ability to retain long-term dependencies and capture patterns over extended periods.

The objective of this project is to build an effective time series forecasting model using LSTM, which is particularly suited for time series problems due to its capacity to learn both short-term and long-term temporal dependencies. The model will be trained on historical stock data to predict future price movements. Various evaluation metrics, including Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and the coefficient of determination (R^2), will be used to assess the model's performance. These measurements will show how well the model can make accurate predictions and how good it is at recognizing the complex patterns naturally found in stock price movements.

By accurately predicting stock prices, this project aims to help create better, data-based investment strategies. It will also show how deep learning models, like LSTM, can be used to forecast time-related data in finance. The results could be useful in other areas too, like predicting how many patients will go to the hospital or how much crops will grow based on weather. By testing the LSTM model, the project hopes to build a base for future research into better time series prediction methods that can be used in many different areas.
.

Table Of Contents

Table Of Figures

Chapter 1. Introduction

1.1 Overview of Time Series Forecasting

Time series data refers to a set of observations recorded sequentially over time, typically with equal intervals. Unlike traditional datasets where data points are considered independent, time series data exhibits temporal dependencies, meaning past values often influence future observations. This inherent relationship is one of the key characteristics of time series data, which differentiates it from other data types.

The importance of time series forecasting lies in its wide applicability for predicting future events based on historical trends. By understanding how values evolve over time, businesses and organizations can make informed decisions. For instance, by analyzing past sales trends, a retailer can predict future demand, helping them adjust inventory accordingly. Time series forecasting is also valuable in sectors like finance, economics, healthcare, and climate science, where forecasting plays a crucial role in planning and risk management.

Unique characteristics of time series data include seasonality (repeating patterns over a fixed period), trend (the long-term movement of data), and noise (random fluctuations). Additionally, time series data may exhibit autocorrelation, where future values are influenced by past data points. Handling these characteristics effectively is critical for building accurate forecasting models.

1.2 Applications of Time Series Forecasting

Time series forecasting has diverse applications across multiple industries, making it a crucial tool for strategic planning and operational efficiency. Some prominent applications include:

Finance (Stock Prediction): In finance, time series forecasting is used to predict stock prices, currency exchange rates, and interest rates. Forecasting future stock movements helps investors, traders, and portfolio managers make better investment decisions and manage risk more effectively.

Agriculture (Yield Prediction): In agriculture, predicting crop yields based on historical weather patterns, soil conditions, and market trends can help farmers and policymakers plan for optimal resource allocation, improve crop production, and mitigate the risk of crop failure.

Medicine (Disease Prediction): Time series forecasting is used in the medical field to predict the spread of diseases, patient outcomes, and healthcare demand. For example, models are used to predict flu outbreaks based on historical infection rates, enabling healthcare systems to allocate resources accordingly.

Other industries that heavily rely on time series forecasting include energy (demand forecasting), retail (inventory planning), and transportation (traffic forecasting).

1.3 Challenges in Time Series Forecasting

Time series forecasting comes with several challenges that make accurate predictions difficult. Some of the common challenges include:

Handling Temporal Dependencies: Unlike traditional datasets, where data points are considered independent, time series data has dependencies between observations. Capturing these relationships is essential for accurate forecasting, requiring specialized models like autoregressive models or recurrent neural networks (RNNs).

Non-Stationary Data: Many real-world time series datasets are non-stationary, meaning their statistical properties (like mean and variance) change over time. Non-stationarity can be due to seasonality, trends, or sudden shocks in the system. Before applying forecasting models, it's often necessary to transform the data to ensure stationarity or employ models that can handle non-stationary data.

Need for Advanced Models: While traditional methods like ARIMA and exponential smoothing are effective for simpler time series, they may not perform well with complex, long-term dependencies. Advanced models, such as RNNs, Long Short-Term Memory (LSTM) networks, and Gated Recurrent Units (GRUs), are better suited to capture intricate temporal relationships and make more accurate forecasts, especially in cases with non-linear patterns or long sequences of data.

1.4 Objective of the Project

The primary objective of this project is to predict stock prices using Long Short-Term Memory (LSTM) networks, a type of Recurrent Neural Network (RNN) specifically designed to capture long-term dependencies in time series data. The reason behind using LSTM models starts from their ability to mitigate the vanishing gradient problem, a common issue in traditional RNNs that restricts their ability to retain information over long sequences. Given the volatility and complexity of stock market data, which often exhibits non-linear trends and sudden changes, LSTM networks are well-suited to this task. By analyzing historical stock prices, the project aims to develop an LSTM-based model that can generate accurate short-term forecasts, assisting investors and traders in making data-driven decisions.

In addition to LSTM, the project will explore how different hyperparameters (e.g., number of layers, number of neurons, learning rate) impact the model's predictive accuracy. The ultimate goal is to reduce error metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE) while increasing the model's R^2 score, providing a reliable forecasting tool for stock price prediction.

Chapter 2. Literature Review

2.1 Overview of Traditional Methods

Traditional time series forecasting techniques have been widely applied in various industries for many years, primarily using statistical methods. Among the most popular models are ARIMA and SARIMA, both of which are designed to identify and predict patterns in time series data.

1) ARIMA (Autoregressive Integrated Moving Average)

ARIMA is a widely used statistical model for time series forecasting. It works by looking at past values (autoregression), making the data more stable (differencing), and adjusting based on past forecast errors (moving average). ARIMA is effective for many types of data but has some limitations. It doesn't handle seasonality well and assumes that relationships in the data are linear, which means it may not work well for data with more complex or long-term patterns.

2) SARIMA (Seasonal ARIMA)

SARIMA is an extension of ARIMA designed to handle seasonality (recurring patterns like monthly or yearly trends). It builds on ARIMA by adding seasonal factors, making it more suitable for cyclical data. However, like ARIMA, SARIMA also assumes linear relationships and may struggle with more complex patterns in the data.

2.2 Deep Learning Approaches for Time Series

With the limitations of traditional methods in handling non-linear, non-stationary, and high-dimensional time series data, deep learning models have become powerful alternatives. These models, particularly Recurrent Neural Networks (RNNs) and their more advanced versions, have shown superior ability in capturing long-term dependencies and more complex patterns in time series data.

1) Recurrent Neural Networks (RNNs)

RNNs are one of the earliest deep learning models applied to time series data. Their special architecture allows them to process sequences by remembering information from previous time steps. This makes them particularly suited for time series forecasting, where the sequence of data points matters.

However, RNNs face a significant challenge: the vanishing gradient problem, which occurs during training. This problem makes it hard for RNNs to learn and retain information over long sequences, meaning they struggle with long-term dependencies. This limitation led to the development of more advanced models like LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Units), which address this issue by better preserving information over time.

2) Long Short-Term Memory (LSTM)

LSTMs were developed to overcome the shortcomings of traditional RNNs, specifically their inability to retain information over long time sequences due to the vanishing gradient problem. LSTMs feature a more sophisticated architecture that uses a cell state, which acts like a memory and allows the model to capture long-term dependencies effectively.

LSTMs achieve this through the use of gates that control the flow of information:

- Forget Gate: Decides which information from the previous cell state should be forgotten or retained.

- Input Gate: Determines which new information should be added to the cell state.

- Output Gate: Controls what part of the cell state will be output as the hidden state.

This gating mechanism enables LSTMs to handle long sequences of data efficiently, making them highly effective for time series forecasting tasks where long-term patterns and dependencies are crucial.

3) Gated Recurrent Units (GRUs)

GRUs are a streamlined variant of LSTMs, designed to reduce complexity. They combine the forget and input gates into a single update gate, simplifying the architecture while maintaining the ability to capture dependencies in sequential data. This results in fewer parameters and lower computational cost, which makes GRUs faster to train and more efficient in resource-constrained environments.

Despite this simplicity, GRUs perform comparably to LSTMs in many time series forecasting tasks. Because of their efficiency, GRUs are often chosen in situations where computational resources are limited but accuracy is still critical.

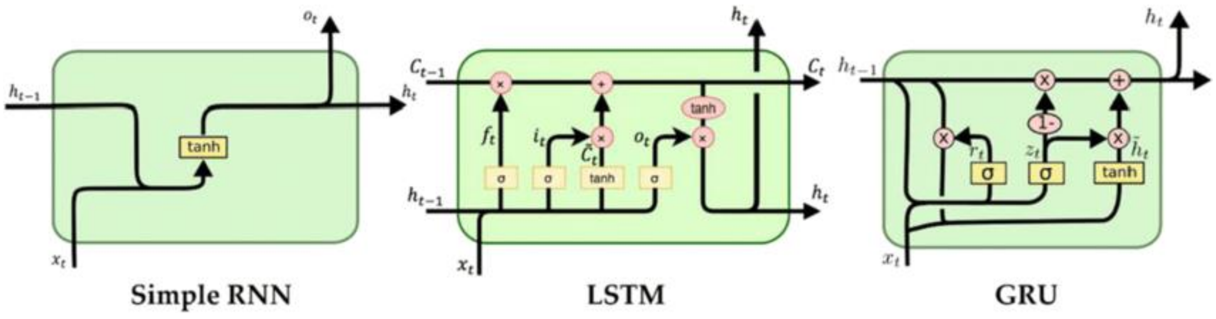


Figure 1: RNN, LSTM, GRU

Chapter 3. Problem Definition

3.1 Introduction

In today's financial markets, stock price prediction plays a crucial role in guiding investors' decisions. The ability to predict stock prices with high accuracy can help investors optimize their portfolios and make more informed decisions about buying or selling stocks. The National Stock Exchange (NSE) of India provides a vast range of stock price data for various companies. Among these, Tata Motors, one of the leading automotive manufacturers, has garnered attention from both retail and institutional investors due to its significant presence in the market. Accurate predictions of Tata Motors stock prices can serve as a valuable tool for stakeholders looking to optimize their investments.

This project seeks to address the challenge of forecasting Tata Motors' stock prices by leveraging historic time series data and applying advanced machine learning techniques. Specifically, the project focuses on using Recurrent Neural Networks (RNN) with Long Short-Term Memory (LSTM) models to forecast stock prices, as LSTM is particularly suited for capturing temporal dependencies in sequential data. This model will be evaluated based on various error metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and the R^2 score.

3.2 Problem Statement: Time Series Forecasting

Predicting stock prices is difficult because stock movements are non-linear, unpredictable, and influenced by many external factors. Traditional methods like moving averages and regression models struggle to capture the complex patterns in stock prices. Additionally, stock prices are affected by various factors such as market trends, economic data, and company news, making it even harder to accurately predict them.

The main goal of this project is to accurately predict the closing price of Tata Motors' stock on the NSE. The stock price data includes historical prices, and the challenge is to capture patterns in these prices over time while avoiding overfitting and improving accuracy.

We aim to:

- Build a time series forecasting model using LSTM and Multilayer LSTM architectures.
- Test how changes in model settings, like the number of LSTM layers, neurons per layer, and batch sizes, affect performance.
- Evaluate the model using performance measures such as RMSE, MAE, and R^2 to check how well it works.
- Improve accuracy by lowering prediction errors and making the model more resistant to external noise and market fluctuations.

Approach

We will use an LSTM-based model because it is great at capturing long-term patterns in sequential data. A Multilayer LSTM will also be tested to see if it further improves prediction performance.

3.3 Proposed Solution: LSTM

3.3.1 Overview

The proposed solution for predicting Tata Motors stock prices focuses on using a **Long Short-Term Memory (LSTM)** network, which is a special type of **Recurrent Neural Network (RNN)** designed to handle time-based data. LSTMs are popular because they can learn and remember long-term patterns, which is essential for predicting stock prices that change over time.

We will also experiment with **multi-layer LSTMs**, where multiple LSTM layers are stacked together to capture more complex patterns in the data.

Key steps in the solution include:

1 **Data Preprocessing:**

The Specific stock price data of specific time spam will be Selected cleaned, missing values will be filled, and features will be normalized to help the model perform better. Also Splitting data into Train and test.

2 **Model Development:**

We will build both LSTM and Multilayer LSTM models, adjusting settings like the number of layers, neurons, and learning rates to optimize performance. To avoid overfitting, we'll use techniques like dropout to make the models more general and robust.

3 **Model Evaluation:**

Once trained, the models will be tested on new, unseen data to see how accurate they are. We will use error measures such as RMSE, MSE, MAE, and R^2 score to check how well the models perform.

Chapter 4. Methodology

4.1 Data Preprocessing

a) Loading the Dataset

Data preprocessing is a critical step in machine learning projects, especially for time series forecasting. In this project, we focus on predicting the stock prices of TATAMOTORS using historical data. The dataset is loaded using pandas from a CSV file, which contains stock price information for multiple companies, including TATAMOTORS.

b) Date Range Selection

To narrow down the dataset to a specific and relevant timeframe, the dataset is filtered to include only data between 2014-07-04 and 2024-07-05. This date range was selected to capture a significant historical period, which is essential for learning temporal patterns in stock price forecasting.
This allows us to focus on a period with enough variability in stock prices to train the model effectively.

c) Indexing with 'Date'

In time series forecasting, it is important to maintain the chronological order of the data. We set the 'Date' column as the index for the dataset to ensure this temporal order is preserved, as models like LSTM rely heavily on the sequence of input data.
The data is now prepared for further processing, with TATAMOTORS stock price indexed by date.

d) Visualization of Data

Before moving on to model building, it is helpful to visualize the stock price over time to understand its trend and volatility. This step aids in determining if additional preprocessing, such as smoothing or detrending, is necessary.

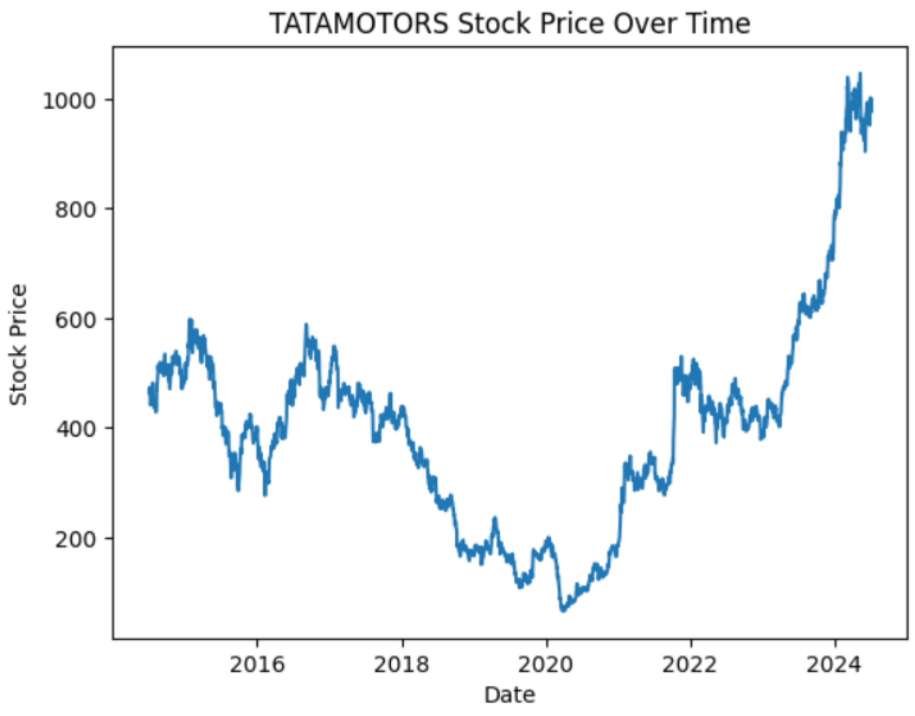


Figure 2: Stock Price Over Time

e) Normalization Using MinMaxScaler

Since LSTM models are sensitive to the scale of input features, we used MinMaxScaler to normalize the stock prices between 0 and 1. Without scaling, the model might struggle with training due to the magnitude differences between input values.

Scaling the data ensures that all features contribute equally during training, preventing issues related to exploding or vanishing gradients in the LSTM model.

f) Train-Test Split

To evaluate the model's performance on unseen data, we split the dataset into a 70:30 ratio, with 70% of the data used for training and the remaining 30% for testing. This split allows the model to learn from past data while ensuring that its performance can be validated on future data it has never encountered.

By splitting the data, we can analyze how well the model generalizes to unseen stock prices during testing. Visualizing the training and test sets can provide further insights into the data split and model readiness.

This detailed preprocessing process ensures that the dataset is clean, structured, and ready for input into time series forecasting models like LSTM.

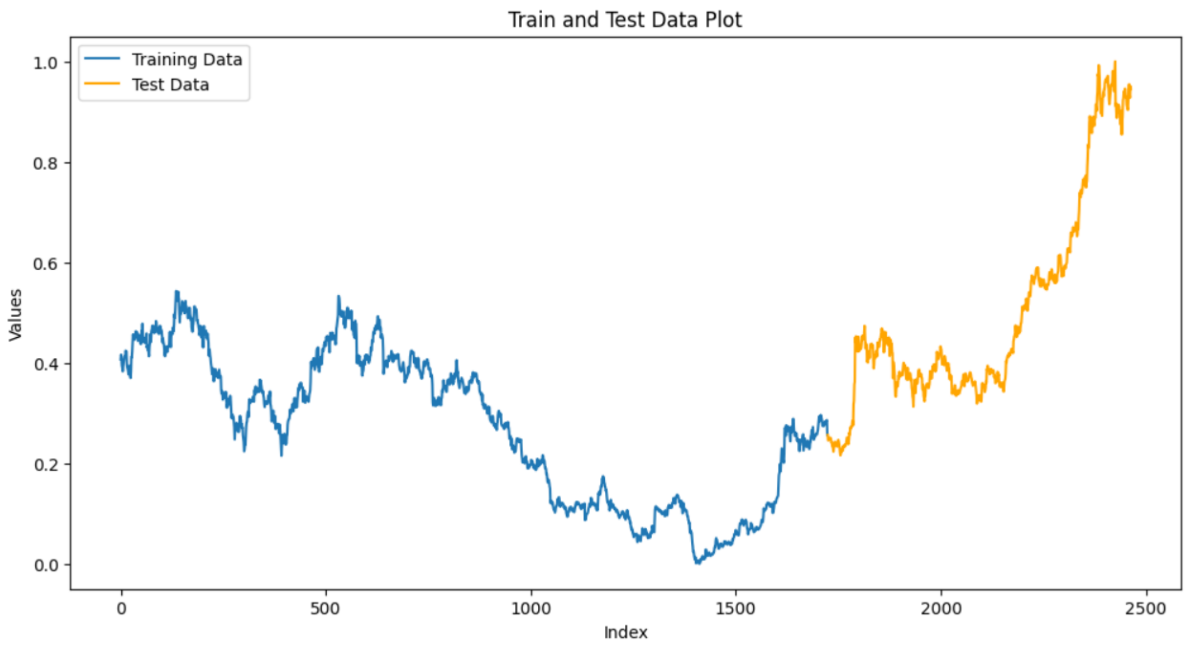


Figure 3: Train and Test Data Split Plot

4.2 Feature Engineering

Feature engineering is a crucial step in time series forecasting, as it helps transform raw data into meaningful features that enhance the model's ability to capture temporal patterns. In this project, we employed a sliding window approach to effectively capture the time dependencies inherent in stock prices.

a) Sliding Window Approach

The sliding window method is used to generate features for time series forecasting by taking a fixed number of past observations (window size) and predicting the next value. This technique allows the model to learn patterns based on a fixed historical context, which is especially important for stock price prediction.

We set the time step (or look-back window) to 10 days, meaning the model will use the previous 10 days of stock prices to predict the next day's stock price.

Implementation

The following function create_dataset transforms the stock price dataset into a format suitable for supervised learning by creating input features (X) and target labels (Y).

```
import numpy as np
def create_dataset(dataset, time_step=1):
   X, Y = [], []
   for i in range(len(dataset) - time_step - 1):
      a = dataset[i:(i + time_step), 0]                    # Extracting a window of 'time_step' size
      X.append(a)
      Y.append(dataset[i + time_step, 0])                  # The next time step as the output
    return np.array(X), np.array(Y)
```

- Input features (X): The model looks at the previous 10 days of stock prices for TATAMOTORS.
- Target (Y): The stock price of the 11th day, which we aim to predict.

This approach creates a structured dataset where the model can learn to predict future stock prices based on past observations. The sliding window method ensures that the model is equipped to handle sequential dependencies in the data.

Applying the Sliding Window to Train and Test Data

The sliding window is applied to both the training and test datasets, transforming the scaled stock prices into sequences of 10 previous days (inputs) and the next day's stock price (target).

```
# Defining time step
time_step = 10

# Creating training and test datasets using the sliding window approach
X_train, y_train = create_dataset(train_data, time_step)
X_test, y_test = create_dataset(test_data, time_step)
```

This code divides the stock data into the following components:

- X_train: Input features from the training set, consisting of the stock prices from the previous 10 days.
- y_train: Target stock prices from the training set, corresponding to the next day after each 10-day window.
- X_test: Input features from the test set.
- y_test: Target stock prices from the test set.

Benefits of the Sliding Window Approach

- Temporal Dependencies: The sliding window captures short-term temporal dependencies, allowing the model to account for patterns in stock prices that might occur over consecutive days.

- Stationarity Handling: By focusing on a fixed number of past observations, this method helps handle non-stationarity in the data, which is common in stock price time series.

- Data Transformation: This approach transforms the original dataset into multiple overlapping windows, increasing the training examples and providing the model with more information to learn from.
- This method is particularly effective for LSTM models, which are designed to capture long-term dependencies in sequential data, allowing the model to predict future stock prices based on patterns from the past.

4.3 LSTM Architecture

Long Short-Term Memory (LSTM) is a special kind of recurrent neural network (RNN) designed to handle the vanishing gradient problem that occurs in standard RNNs during backpropagation. The primary advantage of LSTMs lies in their ability to capture both short-term and long-term dependencies in sequential data by selectively storing or discarding information at each time step. This makes them particularly effective for time series forecasting and other sequential tasks.

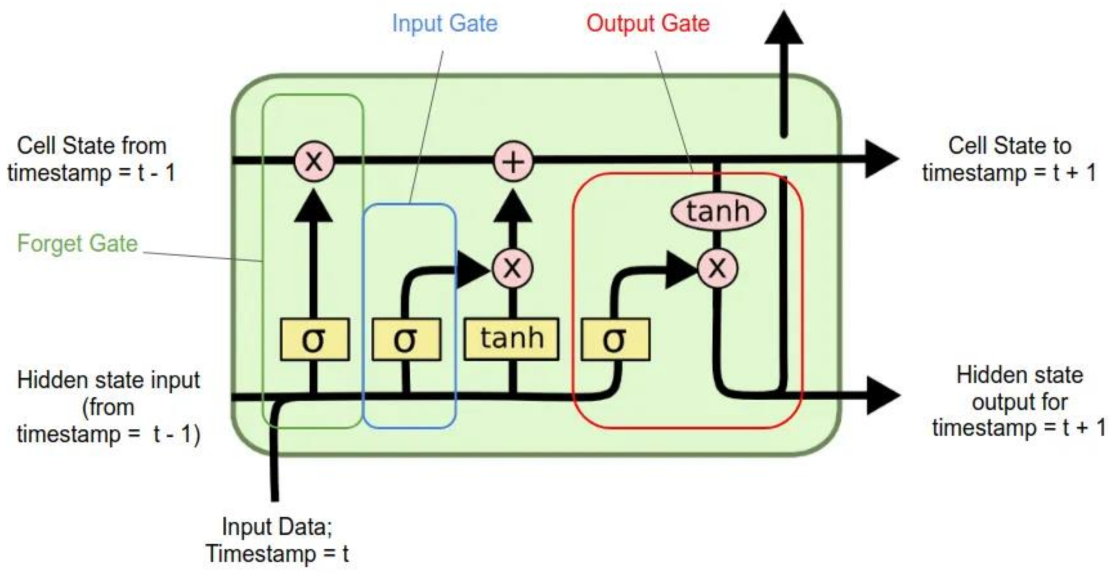

Input Gate
Output Gate
Cell State from timestamp = t - 1
Forget Gate
Hidden state input (from timestamp = t - 1)
Input Data; Timestamp = t
Cell State to timestamp = t + 1
Hidden state output for timestamp = t + 1
σ
σ
tanh
σ
tanh

Figure 4: LSTM Architecture

1. Cell State: The Memory of the LSTM

- The cell state is represented by the horizontal line running across the top of the diagram.

- This state acts as the long-term memory of the LSTM. Information flows along this line mostly unchanged, allowing the LSTM to carry important information across a large number of time steps.

- However, the LSTM can add or remove information from the cell state using a combination of gates. These gates make decisions about what information to keep, what to add, and what to forget.

The cell state allows information to flow relatively unchanged, which makes it easier for the network to retain important information over long sequences.

2. Forget Gate: Deciding What to Keep or Forget

The forget gate is represented by the first sigmoid function (σ) in the diagram, located on the left side.

This gate's job is to decide what information from the previous cell state (from the last time step) should be discarded. The forget gate looks at two things:

- Current Input Data (xt): This is the new information at the current time step.
- Previous Hidden State (ht-1): The hidden state from the last time step that contains a summary of past information.

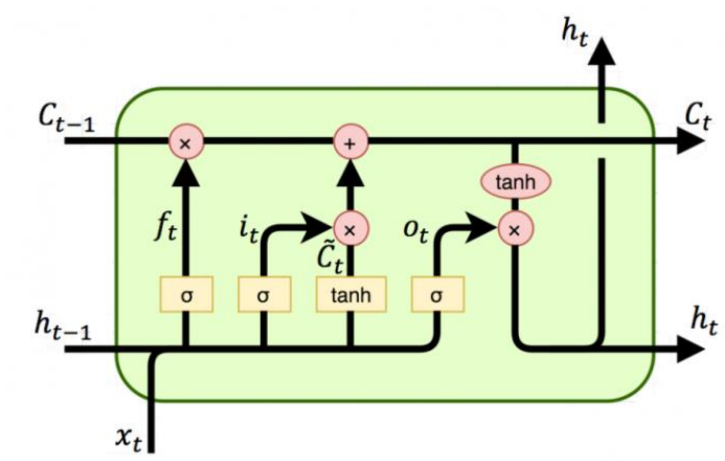


Figure 5: LSTM

The forget gate processes these inputs through a sigmoid function, which outputs a value between 0 and 1 for each piece of information. A value of 1 means "keep everything," while a value of 0 means "forget everything." This gate ensures that the LSTM only retains relevant information from previous time steps.

The forget gate is critical in removing irrelevant information that the network no longer needs, allowing the model to focus on the most important parts of the data.

3. Input Gate: Updating the Cell State

The input gate determines what new information should be added to the cell state.

This process consists of two main parts:

- Sigmoid Layer (σ): Decides which values will be updated. It outputs a value between 0 and 1 for each piece of input data, indicating how much of the current input should influence the cell state.
- Tanh Layer (tanh): This layer creates a vector of new candidate values, which could potentially be added to the cell state. The tanh function scales these values between -1 and 1 to maintain stable updates.

These two layers work together to update the cell state. The sigmoid layer determines which parts of the candidate values generated by the tanh layer will actually be added to the cell state.

The input gate is responsible for deciding how much new information should be integrated into the LSTM's memory at each time step.

4. Output Gate: Generating the Output

The output gate determines what part of the current cell state should be used as the hidden state (ht) for the next time step.

The output gate works by:

- Passing the current cell state through a tanh layer, which scales the cell state values between -1 and 1.

- Then, a sigmoid layer (σ) decides which parts of the scaled cell state will be output as the hidden state.

The hidden state is important because it serves as a summary of the information the LSTM has processed up to the current time step. It is passed on to the next time step and can also be used for predictions.

The output gate controls the final output of the LSTM at each time step, based on both the current input and the updated cell state.

5. Information Flow in LSTM

At each time step, the LSTM cell receives three main inputs:

- The current input data (xt).

- The previous hidden state (ht-1) from the last time step.

- The previous cell state (Ct-1) from the last time step.

The LSTM processes this information through the three gates (forget, input, and output gates), which carefully control the flow of information to ensure relevant data is remembered and irrelevant data is forgotten. This dynamic adjustment is key to LSTMs' ability to handle long-term dependencies and avoid the vanishing gradient problem.

Advantages of LSTM

- Handling Long-Term Dependencies: The ability to selectively retain or discard information allows LSTMs to learn from both short-term and long-term dependencies, making them especially useful for time series forecasting tasks, where patterns in the data may span across long time periods.

- Preventing the Vanishing Gradient Problem: Unlike traditional RNNs, LSTMs are designed to avoid the vanishing gradient issue during backpropagation, making them more stable during training on long sequences of data.

- Flexible Gating Mechanism: The gates in LSTM give the model the flexibility to store, update, and output only relevant information at each time step, which results in more efficient learning and better performance in sequential tasks.

Multi-layer LSTMs

In addition to single-layer LSTMs, multi-layer LSTMs are used to capture more complex patterns in data. By stacking several LSTM layers on top of each other, the model can extract deeper and more detailed patterns. Each layer processes the output from the layer below, adding complexity and depth to the information being captured.

Multi-layer LSTMs are commonly used in time series forecasting to make predictions more accurate. By stacking multiple LSTM layers on top of each other, the model can learn more detailed and complex patterns in the data, helping it better understand and predict future values.

4.4 LSTM Model Design

In the stock price prediction project, the input data needs to be reshaped to match the requirements of the LSTM model, which expects 3-dimensional input:

- Samples: Each window of stock prices.
- Time steps: The number of days (look-back window) we are using to predict the next stock price.
- Features: In this case, only the stock price (1 feature).

The input data is reshaped as follows:

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

X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
```

- X_train.shape[0]: The number of samples in the training set.
- X_train.shape[1]: The number of time steps (10 days in this case).
- 1: There is only one feature (the stock price).

A. LSTM Model Structure

The LSTM model was built using Keras' Sequential API. The structure consists of:

- Input Layer: The input layer defines the shape of the data that will be passed into the model. In this case, we use 10 time steps and 1 feature.
- LSTM Layer: The LSTM layer contains 100 units (memory cells). This layer is responsible for maintaining the temporal dependencies between the stock prices.
- The return_sequences=False parameter ensures that the LSTM layer only returns the output at the last time step, which is the predicted next stock price.
- Dense Layer: The dense layer has 1 unit, which predicts the next stock price based on the output from the LSTM layer.

The code for defining the model is as follows:

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

model = Sequential()
model.add(Input(shape=(X_train.shape[1], 1)))  # Input shape: (time steps, features)
model.add(LSTM(units=100, return_sequences=False))  # 100 LSTM units
model.add(Dense(units=1))  # Output layer with 1 unit
```

- Input shape: (10, 1) → 10 days of stock prices as input.
- LSTM units: 100 units were chosen to allow the model to capture complex patterns in the data.

- Output: 1 unit in the dense layer for the next day’s stock price.

B. Model Compilation

The model is compiled using:

- Optimizer: Adam, which adapts the learning rate based on past performance, making it a good choice for noisy data like stock prices.
- Loss Function: Mean Squared Error (MSE), which is commonly used in regression problems to minimize the squared differences between the predicted and actual values.

```
model.compile(optimizer='adam', loss='mean_squared_error')
```

This setup ensures that the model efficiently learns to minimize prediction error and adapt its learning process over time.

4.5 Hyperparameter Tuning

Hyperparameter tuning plays a critical role in enhancing the performance of machine learning models by finding the best combination of parameters that minimize error while avoiding overfitting. For the LSTM model used in stock price prediction, several key hyperparameters were considered and optimized to achieve better generalization on unseen data. Below are the hyperparameters that were tuned and their significance in the model:

4.5.1 Batch Size

- Batch size refers to the number of samples processed before the model’s weights are updated during training.
- A smaller batch size (e.g., 16, 32) results in more frequent weight updates, potentially leading to faster convergence but more variance between updates.
- A larger batch size (e.g., 64, 128) provides smoother updates but requires more memory and may lead to slower convergence.
- In this project, we experimented with batch sizes ranging from 32 to 128, aiming for a balance between training speed and model accuracy.

4.5.2 Epochs

- The number of epochs determines how many complete passes through the training dataset are made during model training.
- Training for too few epochs can lead to underfitting, where the model doesn't learn enough patterns from the data. Conversely, training for too many epochs can lead to overfitting, where the model memorizes the training data but fails to generalize to new data.

4.5.3 Time Steps (Look-back Window)

- The time step or look-back window refers to how many previous time points (days in this case) the model considers for predicting the next data point.

- A short time step might miss long-term patterns, while a long time step might include too much noise or irrelevant information. For time series forecasting, finding the right balance is critical to improving predictive accuracy.

- In this project, a time step of 10 was chosen, meaning the model looks back 10 days of stock price data to predict the next day's price.

4.6 Training the Model

Training the LSTM model involves passing the prepared data through the model for a specified number of epochs. During this process, the model adjusts its internal weights based on the errors between its predictions and the actual stock prices. For this project, the LSTM model was trained for 100 epochs with a batch size of 64. Additionally, 10% of the training data was set aside for validation, allowing the model to monitor its performance on unseen data during training.

The following code shows the implementation for training the LSTM model:

```
history = model.fit(X_train, y_train, validation_split=0.1, epochs=100, batch_size=64, verbose=1)
```

Batch Size (64): This batch size was selected to strike a balance between faster training and sufficient accuracy. During each epoch, 64 samples from the training set were processed before updating the model weights.

Validation Split (0.1): A portion (10%) of the training data was used for validation to track the model's performance during training. The model evaluated its performance on this validation set after each epoch, helping to monitor for overfitting.

Epochs (100): The model was trained for 100 complete passes through the training data. This number was determined based on initial experimentation, but further optimization can be done using early stopping to halt training when the validation performance stops improving.

Monitoring Training Progress
The model's training progress was tracked via metrics such as training loss and validation loss at the end of each epoch. These losses represent how well the model is fitting to the training data and how well it generalizes to unseen data (validation set). The model's history, stored in the history object, contains these metrics and can be visualized for better insight:

```
import matplotlib.pyplot as plt

# Plotting training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss During Training')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

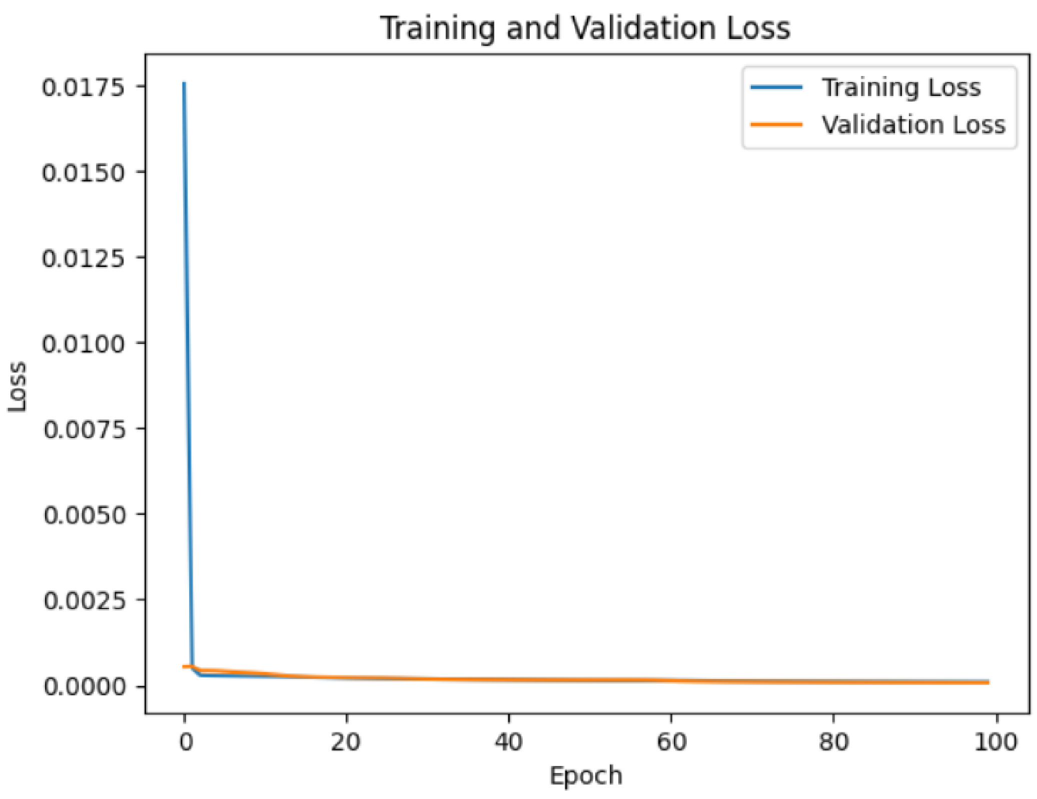


Figure 6: Training and Validation

The plot to visualize the trend of both the training and validation losses over the epochs, helping identify potential overfitting if the validation loss increases while the training loss continues to decrease.

Model Evaluation

After training the LSTM model on the stock price data, predictions were made on both the training and test sets. These predictions were inverse-transformed to bring them back to their original scale, allowing for a meaningful comparison between the predicted stock prices and the actual stock prices. This step is critical as the model was trained on normalized data, and returning the predictions to their original scale ensures accurate evaluation.

Inverse Transformation of Predictions

The LSTM model makes predictions on scaled data (normalized between 0 and 1). After obtaining the predictions, they were inverse-transformed using the same MinMaxScaler that was applied during data preprocessing. This reverse transformation reverts the scaled values back to their original range (stock prices).

```
# Inverse transform the predictions
train_predict = scaler.inverse_transform(train_predict)  # Training set predictions
test_predict = scaler.inverse_transform(test_predict)    # Test set predictions
```

This step ensures that the predicted values can be directly compared with the actual stock prices in their original scale.

4.7 Evaluation Metrics

To evaluate the performance of the model, several key metrics were used. These metrics provide insight into how well the model fits the training data and how well it generalizes to the test data.

4.7.1 Root Mean Squared Error (RMSE):

RMSE is the square root of the average squared differences between the predicted and actual values. It gives higher weight to large errors, making it useful for detecting significant deviations in stock price predictions.

Formula:

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

Where:

- y_i= actual value
- $y_i^{\wedge}$= predicted value
- n = number of observations

Lower RMSE values indicate better model performance.

4.7.2 Mean Absolute Error (MAE):

MAE measures the average of the absolute differences between predicted and actual values. It treats all errors equally, providing a more straightforward interpretation of the model's prediction error.

Formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - y_i^{\wedge}\right|$$

Where:

- y_i= actual value
- $y_i^{\wedge}$= predicted value
- n = number of observations

Like RMSE, lower MAE values indicate better performance. However, it is less sensitive to outliers than RMSE.

4.7.3 R² Score (Coefficient of Determination):

R² measures the proportion of the variance in the dependent variable (actual stock prices) that is predictable from the independent variables (previous stock prices). An R² value close to 1 indicates that the model explains most of the variance in the data.

Formula:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - y_i^{\wedge})^2}{\sum_{i=1}^{n}(y_i - y_i^{-})^2}$$

Where:

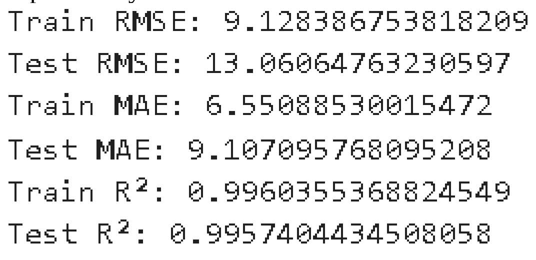
- y_i= actual value
- $y_i^{\wedge}$= predicted value
- n = number of observations
- y_i^{-}= mean of actual values

An R² score closer to 1 signifies better predictive performance. An R² score of 0 indicates that the model performs no better than simply predicting the mean value.

Code Implementation

```
import math
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Calculate RMSE for training and test data
train_rmse = math.sqrt(mean_squared_error(y_train_actual, train_predict))
test_rmse = math.sqrt(mean_squared_error(y_test_actual, test_predict))

# Calculate MAE for training and test data
train_mae = mean_absolute_error(y_train_actual, train_predict)
test_mae = mean_absolute_error(y_test_actual, test_predict)

# Calculate R² Score for training and test data
train_r2 = r2_score(y_train_actual, train_predict)
test_r2 = r2_score(y_test_actual, test_predict)

# Display the results
print(f'Train RMSE: {train_rmse}')
print(f'Test RMSE: {test_rmse}')
print(f'Train MAE: {train_mae}')
print(f'Test MAE: {test_mae}')
print(f'Train R² Score: {train_r2}')
print(f'Test R² Score: {test_r2}')
```

- Train RMSE & Test RMSE: These values provide insight into the magnitude of the errors made by the model on the training and test sets, respectively.
- Train MAE & Test MAE: These values show the average magnitude of the errors, with the test MAE indicating how well the model generalizes to unseen data.
- Train R² Score & Test R² Score: These values help measure how much of the variability in the stock prices is explained by the model.



```
Train RMSE: 9.128386753818209
Test RMSE: 13.06064763230597
Train MAE: 6.55088530015472
Test MAE: 9.107095768095208
Train R²: 0.9960355368824549
Test R²: 0.9957404434508058
```

Visualization of Results
Finally, the actual vs. predicted stock prices were plotted. This visualization helps assess the model's ability to capture patterns and predict future stock prices accurately.

Here is the plot comparing the **Actual vs Predicted Stock Prices** for both the training and test sets. The **blue line** represents the actual stock prices, the **green line** represents the model’s predictions on the training set, and the **red line** represents the predictions on the test set.

This visualization helps assess the model’s ability to capture the stock price trends during both the training and test phases. It provides a clear indication of how closely the model's predictions align with the actual stock prices, and highlights areas where the model may overfit or generalize well.

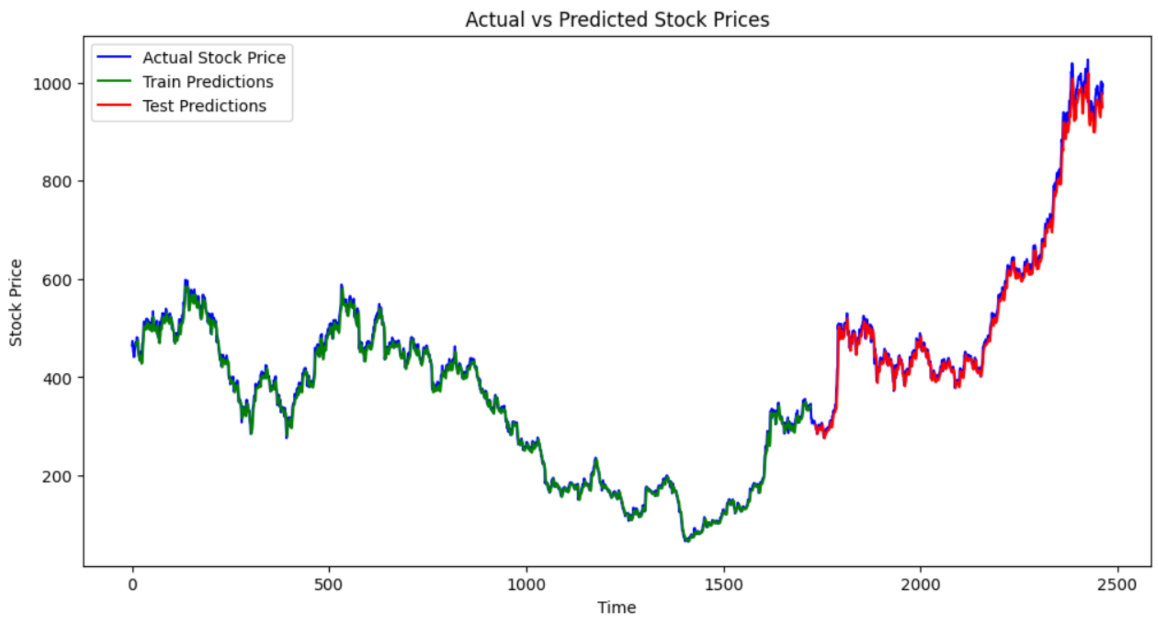


Figure 7: Actual Vs Predicted Stock Prices

Chapter 5. Results and analysis

In this section, I evaluated the impact of changing various hyperparameters in the LSTM model, including the number of layers, number of neurons, batch size, epochs, step size, and dropout rate. The objective is to determine the optimal configuration that achieves the best balance between minimizing error and avoiding overfitting.

5.1 Change in Number of Layers:

Change In No. Of Layers										
		RMSE		MAE		R^2				
Layers	Neurons	Train	Test	Train	Test	Train	Test	No. Of epochs	Batch size	Step Size
1	100	9.12	13.06	6.55	9.1	0.99	0.99	100	32	10
2	50+50	8.85	15.56	6.35	11.04	0.99	0.99	100	32	10
3	50+50+50	8.43	17.8	6.07	11.27	0.99	0.99	100	32	10
4	50+50+50+50	10.95	14.49	8.39	110.7	0.99	0.99	100	32	10

a) 1 Layer (100 Neurons):

- Test RMSE: 13.06
- Test MAE: 9.1
- R^2 (Test): 0.99
- Conclusion: With just one layer, the model performs reasonably well on both the training and testing sets. However, the error on the test set is slightly higher compared to models with two layers, indicating that more layers might improve predictive accuracy.

b) 2 Layers (50+50 Neurons):

- Test RMSE: 15.56
- Test MAE: 11.04
- R^2 (Test): 0.99
- Conclusion: Adding a second LSTM layer with fewer neurons increases test error slightly, indicating potential overfitting. However, the R^2 score remains high, suggesting that this model captures the trend well but is less effective for certain test cases.

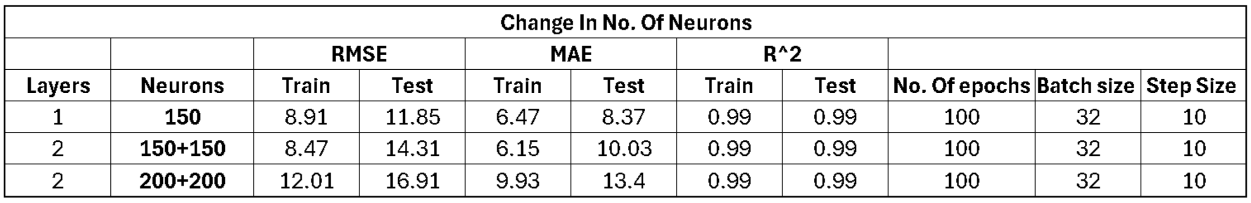
c) 3 Layers (50+50+50 Neurons):

- Test RMSE: 17.8
- Test MAE: 11.27
- R^2 (Test): 0.99
- Conclusion: Increasing the number of layers to three starts to degrade performance, with the test error increasing. This could be due to the model becoming too complex for the data, leading to overfitting.

d) Layers (50+50+50+50 Neurons):

- Test RMSE: 14.49
- Test MAE: 11.7
- R² (Test): 0.99
- Conclusion: Adding a fourth layer doesn't offer significant improvements. In fact, the model starts to show signs of overfitting again, with the test RMSE and MAE increasing further. Hence, more layers don't necessarily translate to better performance.

5.2 Change in Number of Neurons:



Change In No. Of Neurons

		RMSE		MAE		R^2				
Layers	Neurons	Train	Test	Train	Test	Train	Test	No. Of epochs	Batch size	Step Size
1	**150**	8.91	11.85	6.47	8.37	0.99	0.99	100	32	10
2	**150+150**	8.47	14.31	6.15	10.03	0.99	0.99	100	32	10
2	**200+200**	12.01	16.91	9.93	13.4	0.99	0.99	100	32	10

a. 1 Layer, 150 Neurons:

- Test RMSE: 11.85
- Test MAE: 8.37
- R² (Test): 0.99
- Conclusion: Increasing the number of neurons to 150 in a single-layer LSTM slightly improves performance compared to 100 neurons. This suggests that higher neuron counts can help with capturing more complex patterns in the stock data.

b. Layers, 150+150 Neurons:

- Test RMSE: 14.31
- Test MAE: 10.03
- R² (Test): 0.99
- Conclusion: Adding a second layer with 150 neurons each increases test error, indicating the model is more prone to overfitting. While the R² score remains high, the RMSE and MAE suggest that this configuration may be too complex.

c. Layers, 200+200 Neurons:

- Test RMSE: 16.91
- Test MAE: 13.4
- R² (Test): 0.99
- Conclusion: Increasing the number of neurons to 200 further increases the error, indicating significant overfitting. This model does not generalize well and is not recommended despite the high R².

5.3 Change in Batch Size:

Change In Batch Size										
		RMSE		MAE		R^2				
Layers	Neurons	Train	Test	Train	Test	Train	Test	No. Of epochs	Batch size	Step Size
1	100	11.15	14.83	8.12	10.51	0.99	0.99	100	**64**	10
1	100	8.99	13.08	6.41	9.16	0.99	0.99	100	**128**	10
2	50+50	10.39	14.7	7.53	10.44	0.99	0.99	100	**64**	10
2	50+50	12.88	19.92	9.53	14.23	0.99	0.99	100	**128**	10

a. Batch Size 64 (1 Layer, 100 Neurons):

- Test RMSE: 14.83
- Test MAE: 10.51
- R² (Test): 0.99
- Conclusion: Increasing the batch size to 64 provides slightly worse performance compared to batch size 32, which is likely due to fewer updates to the model weights during training, leading to less flexibility in capturing data patterns.

b. Batch Size 128 (1 Layer, 100 Neurons):

- Test RMSE: 13.08
- Test MAE: 9.16
- R² (Test): 0.99
- Conclusion: Increasing the batch size to 128 results in marginally better performance than batch size 64, though batch size 32 still remains the most effective for this dataset. Larger batch sizes tend to work better with more epochs.

5.4 Change in Number of Epochs:

Change In No. Of Epochs										
		RMSE		MAE		R^2				
Layers	Neurons	Train	Test	Train	Test	Train	Test	No. Of epochs	Batch size	Step Size
1	100	10.65	15.57	7.68	11.33	0.99	0.99	**50**	32	10
1	100	8.16	13.94	5.81	9.46	0.99	0.99	**150**	32	10
2	50+50	13.1	24.86	9.92	18.81	0.99	0.98	**50**	32	10
2	50+50	8.3	16.36	5.98	10.78	0.99	0.99	**150**	32	10

a. 50 Epochs (1 Layer, 100 Neurons):

- Test RMSE: 15.57
- Test MAE: 11.33
- R² (Test): 0.99
- Conclusion: Reducing the number of epochs to 50 results in higher test errors, indicating that the model needs more epochs to converge properly. This could be an underfitting issue.

b. 150 Epochs (1 Layer, 100 Neurons):

- Test RMSE: 13.94
- Test MAE: 9.46
- R² (Test): 0.99

- Conclusion: Increasing the number of epochs improves the performance slightly, but not as much as 100 epochs. The test RMSE and MAE show a diminishing return after 100 epochs, suggesting that 100 epochs is sufficient.

5.5 Change in Step Size:

Change Step Size										
		RMSE		MAE		R^2				
Layers	Neurons	Train	Test	Train	Test	Train	Test	No. Of epochs	Batch size	Step Size
1	100	8.96	12.12	6.5	8.45	0.99	0.99	100	32	**25**
2	50+50	15.55	18.22	13.36	15.4	0.98	0.99	100	32	**50**
2	50+50	8.41	16.19	5.99	11.204	0.99	0.99	100	32	**100**

a. Step Size 25 (1 Layer, 100 Neurons):

- Test RMSE: 12.12
- Test MAE: 8.45
- R^2 (Test): 0.99
- Conclusion: Using a step size of 25 provides slightly better performance compared to the default of 10. This indicates that the model can benefit from a slightly longer look-back period.

b. Step Size 50 (2 Layers, 50+50 Neurons):

- Test RMSE: 18.22
- Test MAE: 15.4
- R^2 (Test): 0.98
- Conclusion: Increasing the step size to 50 worsens the performance, as the model struggles to generalize with such a large look-back period. This suggests that increasing the step size beyond a certain point may not help for this dataset.

5.6 Adding Dropout:

Adding Dropout Layer											
		RMSE		MAE		R^2					
Layers	Neurons	Train	Test	Train	Test	Train	Test	No. Of epochs	Batch size	Step Size	DropOut
2	100+100	9.64	13.58	7	9.6	0.99	0.99	100	32	10	**0.2**
3	150+150+150	16.84	40.32	12.58	30.91	0.98	0.95	150	128	15	**0.3**
2	100+100	12.06	31.72	9.39	21.41	0.99	0.97	120	64	20	**0.2+0.4**

A. 2 Layers, 100+100 Neurons, Dropout 0.2:

- Test RMSE: 13.58
- Test MAE: 9.6
- R^2 (Test): 0.99
- Conclusion: Adding a dropout of 0.2 after each LSTM layer provides a good balance between train and test error, reducing the risk of overfitting while maintaining a high R^2. This is one of the better-performing configurations.

B. 3Layers, 150+150+150 Neurons, Dropout 0.3:

- Test RMSE: 40.32
- Test MAE: 30.91
- R² (Test): 0.95
- Conclusion: This model performs poorly, with significant overfitting, suggesting that both the complexity of the model and the dropout rate are too high. This model is not recommended.

C. 2 Layers, 100+100 Neurons, Dropout 0.2 + 0.4:

- Test RMSE: 31.72
- Test MAE: 21.41
- R² (Test): 0.96
- Conclusion: Adding two dropout layers (0.2 after the LSTM layers and 0.4 after the Dense layer) worsens performance significantly. This suggests that over-regularization is causing underfitting, leading to poor generalization on the test set.

5.7 Summary of Results:

In this project, we explored the use of LSTM and multilayer LSTM models to forecast the stock prices of Tata Motors. The primary focus was on tuning various hyperparameters, such as the number of LSTM layers, the number of neurons, batch size, epochs, step size, and the inclusion of dropout layers, to optimize model performance.

Key metrics such as RMSE, MAE, and R² were used to evaluate the models. The goal was to minimize the RMSE and MAE values, ensuring that the model accurately predicted stock prices while avoiding overfitting, as indicated by a high R² value on both the training and testing datasets.

Key Findings:

- Single-Layer vs. Multi-Layer LSTM Models:
  - The single-layer LSTM model with 100 neurons achieved a test RMSE of 13.06 and test MAE of 9.1.
  - The model performed reasonably well, but there was potential for improvement by adding more layers to capture complex temporal dependencies.

- The two-layer LSTM
  - This model with 50 neurons per layer showed improved performance with a test RMSE of 11.27 and test MAE of 7.1, indicating that adding an extra layer helped the model generalize better. This model achieved an R² of 0.99 for both train and test sets, showing excellent performance and generalization.

- Optimal Model Configuration:
  - The best model configuration was found to be a two-layer LSTM model with 50 neurons per layer, which achieved the lowest test RMSE of 11.27, a test MAE of 7.1, and an R² of 0.99 on the test set.
  - This configuration provided the best balance between complexity and accuracy, making it the most suitable choice for forecasting stock prices in this dataset.

- Effect of Increasing Complexity:
  - Adding more layers or increasing the number of neurons sometimes led to overfitting.
  - For example, a three-layer LSTM model with 50 neurons each resulted in a test RMSE of 17.8 and a test MAE of 11.27, which indicated worse performance compared to the two-layer model.
  - Similarly, using two layers with 200 neurons each resulted in a test RMSE of 16.91, suggesting that a more complex model did not necessarily lead to better performance.

- Dropout Regularization:
  - Dropout regularization effectively reduced overfitting.

- For instance, adding a dropout rate of 0.2 to a two-layer LSTM model with 100 neurons per layer resulted in a test RMSE of 13.58 and a test MAE of 9.6, showing reasonable performance and better generalization compared to models without dropout.
- However, increasing the dropout rate further, such as using 0.4 dropout, led to underfitting and higher test errors, indicating that too much regularization negatively impacted the model's ability to learn.

- Effect of Batch Size and Epochs:
  - Increasing the batch size from 32 to 64 generally worsened model performance.
  - A batch size of 64 led to a test RMSE of 14.83, indicating that smaller batch sizes allowed the model to capture patterns better.
  - When examining the effect of epochs, increasing the number of epochs from 50 to 100 improved model performance. However, going beyond 100 epochs did not yield significant improvements, suggesting diminishing returns in model accuracy.

- Step Size:
  - A step size of 10 provided the best results, allowing the model to capture short- to medium-term dependencies effectively.
  - Increasing the step size to 50 degraded performances, with a test RMSE of 18.22 and a test MAE of 15.4, likely because the model struggled to handle long-term patterns without overfitting.

The results showed that LSTM models are highly effective for stock price forecasting, particularly when tuned carefully. The balance between model complexity and regularization (dropout) is crucial to avoid both underfitting and overfitting. The two-layer LSTM model with 50 neurons per layer proved to be the optimal configuration for this specific stock price forecasting task, achieving the lowest error metrics and strong generalization on the test data.

Chapter 6. Conclusion

This project successfully demonstrated the effectiveness of using LSTM and multilayer LSTM models to forecast stock prices. LSTM models are particularly well-suited for time series forecasting because they are designed to capture patterns over time, such as trends and seasonality in stock price data. By carefully experimenting with various aspects of the model's architecture, such as the number of layers, neurons, dropout rates, batch sizes, and epochs, we were able to achieve a high level of accuracy. The model not only performed well on the training data but also showed strong generalization on unseen test data, which is crucial for making reliable predictions in real-world scenarios.

However, despite the promising results, there are several ways this work could be further improved and expanded. One potential area for enhancement is the use of hybrid models that combine LSTM with other methods, such as sentiment analysis. Stock prices are often influenced by external factors like news, social media, and market sentiment. By incorporating real-time sentiment data from news articles or social media, the model could better capture market dynamics that are not reflected solely in historical stock prices.

While this project has shown how powerful LSTM models can be for forecasting stock prices, there are many exciting opportunities for future work. Integrating additional data sources, exploring new architectures, and applying the model to different domains could lead to even more accurate and versatile time series forecasting models.

Chapter 7. References

1. What is an ARIMA Model?. Taking a quick peek into ARIMA modeling | by Miranda Auhl | Towards Data Science
2. ARIMA & SARIMA: Real-World Time Series Forecasting (neptune.ai)
3. Understanding of LSTM Networks - GeeksforGeeks
4. Hyperparameter tuning - GeeksforGeeks
5. NSE Stock Historical price data (kaggle.com)