Prediction of Stock Prices using Deep Learning Techniques

By: -

Diya Patidar
B.Tech in CSE (AI/ML), Sikkim Manipal Institute of Technology

Manaswi
B.Tech in CSE (AI/ML), Sikkim Manipal Institute of Technology

Aditya Baidya
B.Tech in CSE, Heritage Institute of Technology

Ritesh Ghosh
B.Tech in CSE, Heritage Institute of Technology

Soham Ganguly
B.Tech in CSE, Heritage Institute of Technology

Anubhab Mondal
B.Tech in CSE, Heritage Institute of Technology

Mentor Name: **Prof. Indranil Basu**
Period of Internship: 19th May 2025 - 15th July 2025

Report submitted to: IDEAS – Institute of Data Engineering, Analytics and Science Foundation, ISI Kolkata

Table of Content

Sl No.	Title	Page No.
1.	Abstract	3
2.	Introduction	4 - 6
3.	Project Objective	7
4.	Methodology	8 - 12
5.	Data Analysis	13 - 40
6.	Results	41 - 72
7.	Observation	73 - 74
8.	Conclusion	75
9.	References	76

ABSTRACT

This project focuses on forecasting the closing prices of INFOSYS stock using deep learning techniques. The goal is to build predictive models that can learn temporal dependencies from historical data and accurately estimate future stock prices.

Four models were developed and compared: a base Simple Recurrent Neural Network (Simple RNN), an RNN fine-tuned using Reinforcement Learning (RL), a Long Short-Term Memory (LSTM) network, and an LSTM integrated with RL.

Stock data from 2020 to mid-2025 was collected using the Yahoo Finance API and preprocessed using Min-Max Scaling. Sequences of 60 past closing prices were used as inputs for each model.

The models were trained and evaluated based on MSE, MAE, MAPE, and prediction accuracy. The inclusion of RL aimed to optimize the prediction performance by fine-tuning model weights based on reward feedback.

Results show that models with LSTM architecture generally outperformed base RNNs, and RL-based fine-tuning improved accuracy in specific scenarios. This comparative study provides insights into the efficacy of combining deep learning with reinforcement learning in stock market forecasting.

The proposed framework can serve as a baseline for further enhancements using attention mechanisms or hybrid ensemble models.

INTRODUCTION

Stock market forecasting has always been a domain of keen interest due to its direct impact on financial decisions. The rapid advancements in Artificial Intelligence and the availability of vast historical financial data have enabled researchers and practitioners to explore deep learning methods for predicting stock prices. Traditional statistical models often fail to capture the non-linearity and time dependencies inherent in financial time series. In contrast, Recurrent Neural Networks (RNNs) and their variants like Long Short-Term Memory (LSTM) networks have proven effective in modeling sequential data due to their memory capabilities. As discussed, deep learning models like RNNs and LSTMs are particularly suited for sequential prediction tasks due to their internal state retention and time-aware computations.

This project aims to develop a stock price prediction system for INFOSYS Ltd. (NSE: INFY) by building and comparing four models:

1. **Simple RNN (Recurrent Neural Network)**
2. **RNN fine-tuned with Reinforcement Learning**
3. **LSTM (Long - Short Term Memory)**
4. **LSTM fine-tuned with Reinforcement Learning**

We collected historical stock data of INFOSYS from 1st June 2020 to 2nd July 2025 using the Yahoo Finance API and used it to train deep learning models. A sliding window approach was used to generate sequences of 60 previous days' closing prices as input. The models were evaluated using standard regression metrics including Mean Squared Error (MSE), Mean Absolute Error (MAE), and Mean Absolute Percentage Error (MAPE). Reinforcement Learning (RL) was applied as a fine-tuning method to optimize predictions based on reward signals linked to prediction accuracy. The core idea behind RL — learning from feedback through reward signals — is widely studied in modern AI systems.

The purpose of this project is to study the comparative effectiveness of sequence learning models in financial forecasting and to explore the integration of deep learning and reinforcement learning for improved accuracy in time-series prediction tasks. The results of this project may provide actionable insights for retail investors, financial analysts, and algorithmic traders.

Technologies and Tools Used

- Programming Language: Python
- Libraries: TensorFlow/Keras, NumPy, Pandas, scikit-learn, Matplotlib
- Data Source: Yahoo Finance (via yfinance)
- Models: Simple RNN, LSTM, Reinforcement Learning
- Hardware: Google Colab / Jupyter Notebook

Training Topics Covered During First Two Weeks of Internship

During the initial phase of the internship, Received training and hands-on practice in the following areas:

1. **Data Science Project:**

A data science project involves collecting, analyzing, and interpreting large volumes of data to extract meaningful insights. It typically follows a structured workflow: problem definition, data collection, data preprocessing, exploratory data analysis (EDA), model building, evaluation, and deployment. The goal is to apply statistical and machine learning techniques to support decision-making or solve business problems.

2. **How to conduct a Research Project:**

   Conducting a research project begins with identifying a clear research problem or question. This is followed by a literature review, hypothesis formulation, data collection (primary or secondary), and selecting appropriate methodologies for analysis. The final steps include drawing conclusions, validating results, and presenting findings in a formal report or publication.

3. **Power BI:**

   Power BI is a Microsoft business analytics tool used to visualize data and share insights across organizations. It enables users to connect to various data sources, build interactive dashboards, and perform real-time reporting. Its user-friendly drag-and-drop interface makes it ideal for both technical and non-technical users to generate business intelligence insights.

4. **Streamlit:**

   Streamlit is an open-source Python library that allows data scientists and developers to build and deploy interactive web applications for machine learning and data visualization with minimal effort. It supports easy integration with Pandas, NumPy, Matplotlib, and TensorFlow, making it highly suitable for creating ML model dashboards and rapid prototyping.

5. **Machine Learning:**

   Machine learning (ML) is a subset of artificial intelligence that enables systems to learn patterns from data and make predictions without explicit programming. ML models are trained on datasets to perform tasks such as classification, regression, and clustering. Supervised, unsupervised, and reinforcement learning are the primary paradigms used in ML.

6. **Deep Learning:**

   Deep learning is a specialized branch of machine learning that uses multi-layered neural networks to model complex patterns in large datasets. It is particularly effective for tasks like image recognition,

   natural language processing, and speech synthesis. Techniques like Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks are commonly used in deep learning applications.

7. **Prompt Engineering:**

   Prompt engineering is the practice of crafting effective input queries (prompts) for large language models (LLMs) like GPT to generate accurate and relevant outputs. It involves understanding the behaviour of LLMs, using structured prompts, examples, or instructions to guide the model's response. Prompt engineering plays a crucial role in applications like code generation, summarization, and chatbot development.

8. **Advanced Deep Learning and Pytorch:**

   Advanced deep learning explores concepts beyond standard neural networks, including attention mechanisms, transformers, generative models, and optimization techniques. PyTorch, an open-source deep learning framework developed by Meta AI, is widely used for building and training these models due to its flexibility, dynamic computation graph, and user-friendly API.

9. **Generative AI:**

   Generative AI enhances Business Intelligence by enabling natural language querying, automated report generation, and predictive insight synthesis. By integrating LLMs with BI tools, users can ask complex questions about their data and receive narrative answers or visualizations. This reduces reliance on technical users and democratizes data access across organizations.

PROJECT OBJEECTIVE

The primary objective of this project is to develop and compare various deep learning models to accurately predict the future stock prices of INFOSYS Ltd. using historical closing price data. The specific goals of the project are:

1. To design and implement four predictive models: Simple RNN, RNN fine-tuned with Reinforcement Learning (RL), LSTM, and LSTM fine-tuned with RL.

2. To evaluate the models performance using standard regression metrics such as Mean Squared Error (MSE), Mean Absolute Error (MAE), Mean Absolute Percentage Error (MAPE), and prediction accuracy.

3. To analyze whether Reinforcement Learning can significantly enhance time-series model performance when used for fine-tuning deep learning models.

4. To illustrate the practical application of deep learning models in financial forecasting by predicting real stock prices during a defined test period.

5. To determine the most suitable model for forecasting based on a balance of predictive performance and architectural complexity.

METHODOLOGY

The methodology adopted for this project involves a systematic pipeline comprising data acquisition, preprocessing, model development, training, prediction, evaluation, and visualization. Below are the key steps:

1. **User Input and Validation**

   The system prompts the user to enter a date (in YYYY-MM-DD format) for which they want the INFOSYS stock closing price prediction.
   Input is validated to ensure:

   - The date falls on a weekday (Monday to Friday).
   - It is not a trading holiday as per the official NSE/BSE holiday calendar.

- The date is within the valid range (June 1st, 2025 – July 2nd, 2025).
- Invalid inputs are rejected, and the user is asked to re-enter until a valid date is provided.

2. Data Acquisition

- Historical daily stock data for INFOSYS is collected using APIs such as **nsepy**, **NSEPython**, **Yahoo Finance** or **pandas_datareader**.
- Only closing prices are extracted and used for modeling.

3. Data Preprocessing

- The data is cleaned to handle any missing or anomalous entries.
- The closing prices are scaled between 0 and 1 using **MinMaxScaler** for better neural network convergence.
- A sliding window approach is used to generate sequences of the previous 60 days as input and the next day's closing price as the label.

4. Model Development: Base RNN

A simple Recurrent Neural Network (RNN) is constructed using the **SimpleRNN** layer from tensorflow.keras.layers.The model includes:

- Input layer with shape (60, 1)
- A SimpleRNN layer
- A Dense output layer with 1 neuron for price prediction
- The model is compiled with **Mean Squared Error (MSE)** loss function and as Adam the optimizer.

5. Model Training

- The model is trained on the preprocessed data from 1st June 2020 to 31st May 2025.
- A portion of the data is used for validation to monitor overfitting.
- Training is done for a defined number of epochs with a batch size tuned through experimentation.

6. Prediction Process

- For the user-specified date, the last 60 trading days before that date are used as input to the trained RNN model.
- The model predicts the normalized closing price, which is then inverse-transformed to get the actual price.

7. Performance Comparison

- A Simple RNN **model** is trained on the same data to provide a baseline for comparison.
- Additionally, an **LSTM model** is developed and fine-tuned using R**einforcement learning**, where the model is optimized based on reward signals linked to prediction accuracy.

8. Visualization

- The actual vs predicted closing prices are plotted using **Matplotlib**.
- Historical trends, prediction errors, and model performance curves are visualized to support evaluation.

9. Output and Evaluation

- The system displays the predicted price for the chosen date.
- If the actual price is available, it is shown for comparison.
- Evaluation metrics like MSE, MAE, and MAPE are computed for model assessment.

This structured approach ensures reproducibility, robustness, and interpretability of the prediction model, while also allowing further extension through advanced learning techniques like LSTM and reinforcement learning.

Flow Chart:-

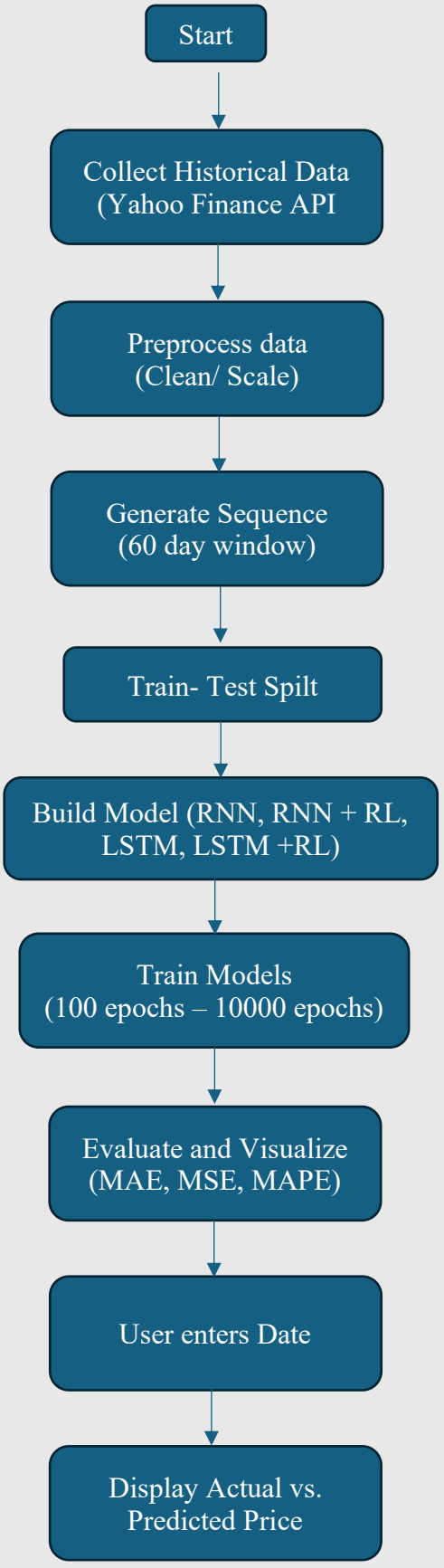


Fig 1: Workflow of INFOSYS Stock Price Prediction Project

Tools and Technologies Used

Component	Tool / Library
Language	Python
Data Source	Yahoo Finance
Libraries	Yfinance, Numpy, Pandas, Matplotlib, scikit-learn, TensorFlow/ Keras
Model Types	Simple RNN, LSTM
Optimization	Adam Optimizer, Man Squared Error
Platform	Google Colab

Model Selection & Validation

- Model Selection was based on time-series nature of the data: RNN and LSTM are well-suited for sequential modeling.
- Validation Strategy: Date-based split; no random shuffling.
- Hyperparameters: Number of RNN/LSTM units = 50, Activation = "tanh", Loss = MSE, Optimizer = Adam.

🔗 GitHub Repository

GitHub Link: INFOSYS_STOCK_PRICE_PREDICTION

DATA ANALYSIS

1. Data Overview:

The dataset used comprises historical daily stock prices of INFOSYS (NSE: INFY) from 1st June 2020, to 2nd July 2025 Key attributes include Date, Open, High, Low, Close, and Volume. For this project, only the Closing Price is considered for forecasting purposes.

- **Total Records**: ~1240 days
- **Missing Values**: Handled by forward filling and interpolation.
- **Outliers**: No significant anomalies were observed in the closing price trend.
- **Date Range Validation**: Data was filtered to exclude weekends and official NSE trading holidays using published holiday lists from 2020 to 2025.

Price	Close	High	Low	Open	Volume
Ticker	INFY.NS	INFY.NS	INFY.NS	INFY.NS	INFY.NS
Date					
2020-06-01	630.987183	637.391346	623.726173	630.265602	12276438
2020-06-02	638.699219	641.901328	629.183233	631.844115	7059304
2020-06-03	632.791199	642.126830	627.785164	641.044431	11378568
2020-06-04	638.383545	639.916939	629.138146	633.197101	10782424
2020-06-05	634.595215	641.946457	631.934333	637.707092	7718679
...	...	...	...	...	...
2025-05-26	1558.570679	1561.824940	1541.017604	1544.863516	3170102
2025-05-27	1548.315063	1560.937513	1534.805197	1557.979138	7775714
2025-05-28	1549.991455	1565.670748	1548.216382	1559.063758	3623370
2025-05-29	1563.599976	1586.576739	1554.724849	1566.952825	8609500
2025-05-30	1562.699951	1572.000000	1555.099976	1571.000000	13148092

1240 rows × 5 columns

Fig 2 : Infosys data set from 1st June 2020 to 2nd July 2025

2. Data Preprocessing:

- The closing prices were scaled to the [0, 1] range using MinMaxScaler to improve training stability.
- A sliding window of 60 previous trading days was used to create input sequences and corresponding labels.

The final dataset was split into training (80%) and validation (20%) sets.

```python
# Get Indian trading holidays
def get_nse_holidays():
    in_holidays = holidays.India(years=range(2020, 2026))
    return set(in_holidays.keys())
```

Fig 3 : Filtering out Trading Holidays

```python
def get_stock_data():
    print("Fetching data from Yahoo Finance...")
    full_df = yf.download(SYMBOL, start=TRAIN_START_DATE, end=TEST_END_DATE)
    full_df = full_df[['Close']]
    full_df.dropna(inplace=True)
    full_df.index = pd.to_datetime(full_df.index)
    display(full_df.head())
    train_df = full_df.loc[TRAIN_START_DATE:TRAIN_END_DATE]
    test_df = full_df.loc[TEST_START_DATE:TEST_END_DATE]
    return train_df, test_df, full_df
```

Fig 4: Preprocessing of original dataframe

3. Model Training and Evaluation Base RNN (SimpleRNN)

A **Recurrent Neural Network (RNN)** is a type of artificial neural network designed for processing sequential data such as text, time series, and speech. Unlike traditional feedforward neural networks, RNNs maintain a hidden state that carries information from previous time steps, enabling the model to learn temporal patterns and dependencies. At each time step, the RNN takes an input vector and the previous hidden state to compute a new hidden state using the formula ht=tanh(W·ht−1+U·xt), and then generates an output y^t=V·ht, where U, W, and V are learned weight matrices. This structure allows RNNs to share weights across time steps, making them efficient for tasks involving sequences. However, standard RNNs struggle with long-term dependencies due to vanishing or exploding gradients, which is why more advanced versions like LSTM and GRU are often used in practice.

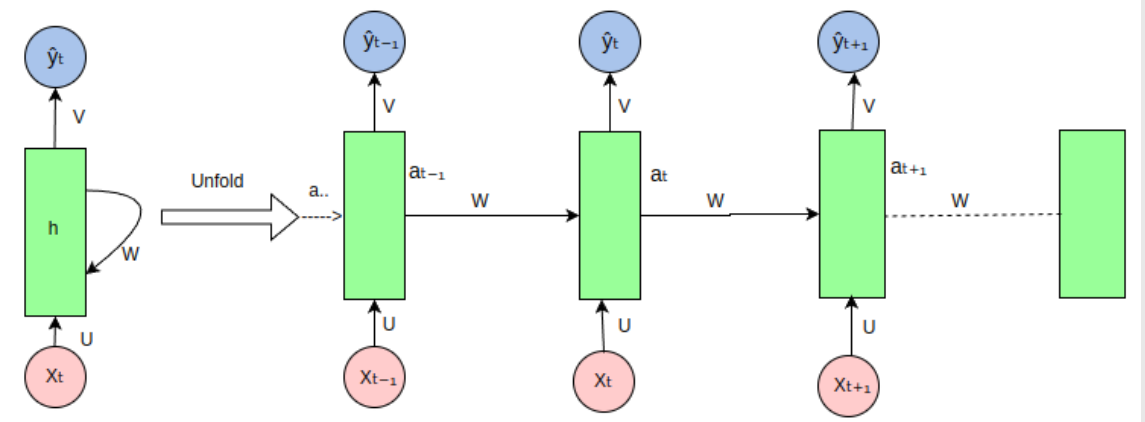


Fig 5: SimpleRNN Architecture

Snapshot of Simple RNN Code :-

Step 1: Importing All the necessary Library

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
from datetime import date
import yfinance as yf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN
from sklearn.preprocessing import MinMaxScaler
```

Step 2: Downloading Data

```
np.random.seed(42)

print("Downloading INFOSYS historical data...")
df = yf.download("INFY.NS", start="2020-01-01", end="2025-07-02")

if df.empty:
    print("ERROR: No data returned.")
    exit()
```

Step 3: Using Only Closing Price

```
df = df[["Close"]]
df.reset_index(inplace=True)
df["Date"] = pd.to_datetime(df["Date"]).dt.date
df.sort_values("Date", inplace=True)
```

Step 4: Splitting Data into Training and Testing Data

```
train_df = df[df["Date"] <= date(2025, 5, 31)]
test_df = df[(df["Date"] >= date(2025, 6, 1)) & (df["Date"] <= date(2025, 7, 1))]
```

Step 5: Scaling the Data

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_close = scaler.fit_transform(df["Close"].values.reshape(-1, 1))
```

Step 6: Creating Sequences

```
sequence_length = 60
X, y = [], []
for i in range(sequence_length, len(scaled_close)):
    X.append(scaled_close[i-sequence_length:i, 0])
    y.append(scaled_close[i, 0])
X, y = np.array(X), np.array(y)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))
```

Step 7: Determining the Split index

```
model = Sequential()
model.add(SimpleRNN(50, activation="tanh", input_shape=(X.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer="adam", loss="mean_squared_error")
```

Step 8: Building Simple RNN Model

```
model = Sequential()
model.add(SimpleRNN(50, activation="tanh", input_shape=(X.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer="adam", loss="mean_squared_error")
```

Step 9: Train model with validation on test data

```
print("Training model for 100 epochs...")
history = model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_data=(X_test, y_test),
    verbose=1
)
```

Step 10: Predicting on test data

```
predicted_scaled = model.predict(X_test)
predicted = scaler.inverse_transform(predicted_scaled)
actual = scaler.inverse_transform(y_test.reshape(-1, 1))
```

Step 11: Training Set Metrics

```
train_pred_scaled = model.predict(X_train)
train_pred = scaler.inverse_transform(train_pred_scaled)
train_actual = scaler.inverse_transform(y_train.reshape(-1, 1))

mse_train = np.mean((train_pred - train_actual)**2)
mae_train = np.mean(abs(train_pred - train_actual))
mape_train = np.mean(abs(train_pred - train_actual) / train_actual) * 100
accuracy_train = 100 - mape_train
avg_train_price = np.mean(train_actual)
mse_train_percent = (mse_train / avg_train_price) * 100
mae_train_percent = (mae_train / avg_train_price) * 100
```

Step 12: Test Set Metrics

```
mse = np.mean((predicted - actual) ** 2)
mae = np.mean(abs(predicted - actual))
avg_price = np.mean(actual)
mse_percent = (mse / avg_price) * 100
mae_percent = (mae / avg_price) * 100
mape = np.mean(abs(predicted - actual) / actual) * 100
accuracy = 100 - mape
```

Step 13: Print Metrics

```
print("\n========== Training Set Metrics ==========")
print(f"MSE: {mse_train:.4f} ({mse_train_percent:.4f}% of avg price)")
print(f"MAE: {mae_train:.4f} ({mae_train_percent:.4f}% of avg price)")
print(f"Accuracy: {accuracy_train:.2f}%")

print("\n========== Test Set Metrics ==========")
print(f"MSE: {mse:.4f} ({mse_percent:.4f}% of avg price)")
print(f"MAE: {mae:.4f} ({mae_percent:.4f}% of avg price)")
print(f"Accuracy: {accuracy:.2f}%")
print("=======================================\n")
```

Step 14: Prompt user for date

```
while True:
    user_input = input("Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): ").strip()
    try:
        user_date = datetime.datetime.strptime(user_input, "%Y-%m-%d").date()
    except ValueError:
        print("Invalid format. Try again.")
        continue
    if not (date(2025, 6, 1) <= user_date <= date(2025, 7, 1)):
        print("Date out of range.")
        continue
    if user_date not in test_dates:
        print("Date not in test data.")
        continue
    break
```

Step 15: Finding Index

```
idx = np.where(test_dates == user_date)[0][0]
pred_price = predicted[idx][0]
act_price = actual[idx][0]
error_pct = abs(pred_price - act_price) / act_price * 100
```

Step 16: Displaying the predictions

```
print(f"\nFor {user_date}:")
print(f"Predicted closing price: ₹{pred_price:.2f}")
print(f"Actual closing price: ₹{act_price:.2f}")
print(f"Absolute error: {error_pct:.2f}%")
```

Step 17: Plotting the Actual Vs Predicted Closing Prices

```
plt.figure(figsize=(14, 6))
plt.plot(test_dates, actual, label="Actual Closing Price")
plt.plot(test_dates, predicted, label="Predicted Closing Price")
plt.axvline(x=np.datetime64(user_date), color="red", linestyle="--", label="Selected Date")
plt.xlabel("Date")
plt.ylabel("Price (₹)")
plt.title("INFOSYS - Actual vs Predicted Closing Prices (Test Period)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Step 18: Combined Loss Plot

```
plt.figure(figsize=(12, 6))
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Testing Loss")
plt.title("Training and Testing Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

2. Simple RNN Fine Tuning with Reinforcement Learning

A Simple Recurrent Neural Network (RNN) is a type of neural network designed to process sequential data by maintaining a memory of previous inputs through hidden states, making it effective for time-series and sequence prediction tasks. When combined with Reinforcement Learning (RL), the RNN learns to make decisions over time based on rewards from the environment. In this setup, the RNN processes sequences of states or observations and outputs actions, which are evaluated by a reward signal. Over multiple episodes, the model is trained using RL algorithms (such as Policy Gradient or Q-Learning) to maximize the cumulative reward. This combination allows the RNN to not only understand temporal dependencies but also learn optimal strategies in dynamic environments, making it useful in applications like game playing, robotics, and stock trading.

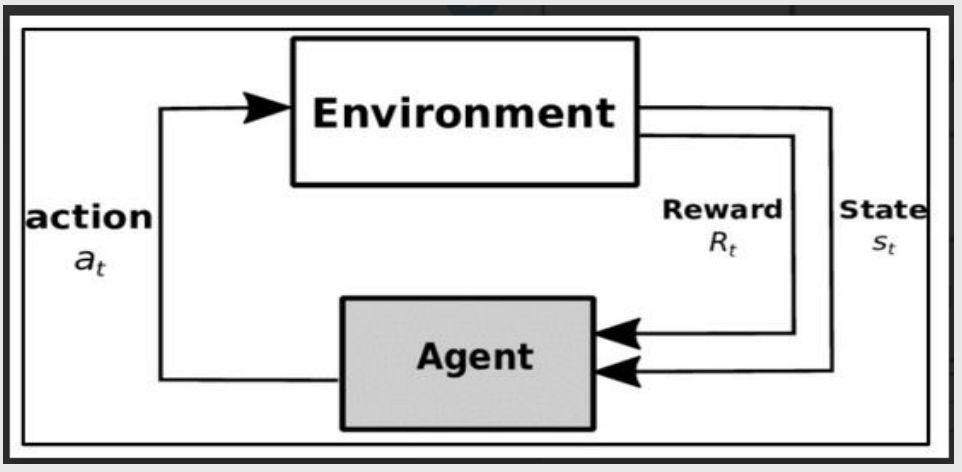


Fig 6: Process of Reinforcement Learning

The diagram illustrates the fundamental architecture of Reinforcement Learning (RL), which involves an interaction between an **Agent** and an **Environment**. At each time step t, the agent observes the current **state of** the environment and takes an **action based** on a policy. The environment then responds by transitioning to a new state and providing a **reward** Rt that evaluates the effectiveness of the agent's action. This reward serves as feedback, guiding the agent to learn an optimal strategy that maximizes the cumulative reward over time. The loop continues, allowing the agent to refine its policy through trial and error, which is central to learning in dynamic and uncertain environments.

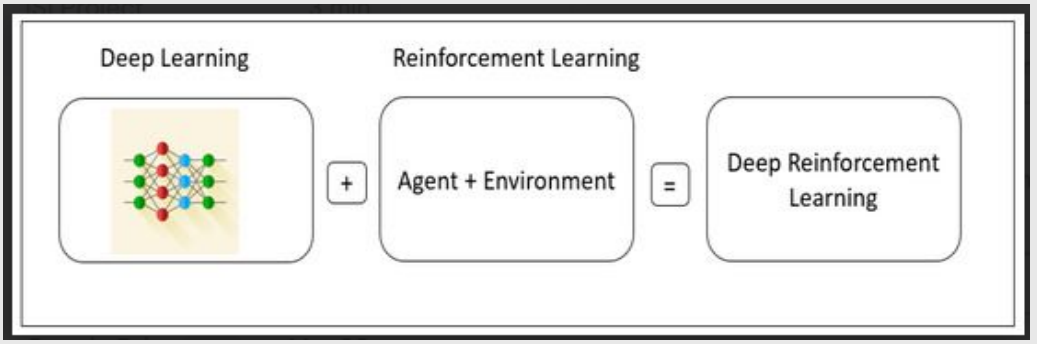


Fig 7: SimpleRNN fine tuned with Reinforcement learning

Snapshot of Simple RNN fine Tuned with RL

Step 1: Importing all the Required Libraries

```
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import LSTM, SimpleRNN, Dense, Input
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, mean_absolute_percentage_error
from matplotlib.dates import DateFormatter
import tensorflow as tf
import numpy as np
import pandas as pd
import datetime
import holidays
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import warnings

warnings.filterwarnings('ignore')
```

Step 2: Set Global Variables and Seed

```
np.random.seed(42)
tf.random.set_seed(42)

SYMBOL = "INFY.NS"
SEQUENCE_LENGTH = 60

TRAIN_START_DATE = "2020-06-01"
TRAIN_END_DATE = "2025-05-31"
TEST_START_DATE = "2025-06-01"
TEST_END_DATE = "2025-07-01"
```

Step 3: Define trading Holidays and Input Validator

```
def get_nse_holidays():
    in_holidays = holidays.India(years=range(2020, 2026))
    return set(in_holidays.keys())

TRADING_HOLIDAYS = get_nse_holidays()
```

```python
def is_valid_date(date_str):
    try:
        date = datetime.datetime.strptime(date_str, "%Y-%m-%d").date()
        if not (datetime.date(2025, 6, 1) <= date <= datetime.date(2025, 7, 1)):
            print("Date must be between 2025-06-01 and 2025-07-01.")
            return False
        if date.weekday() >= 5:
            print("Weekends are not trading days.")
            return False
        if date in TRADING_HOLIDAYS:
            print("Entered date is a trading holiday.")
            return False
        return True
    except ValueError:
        print("Invalid date format. Use YYYY-MM-DD.")
        return False
```

Step 4: Data Fetching And User Date Input

```python
def get_stock_data():
    print("Fetching data from Yahoo Finance...")
    full_df = yf.download(SYMBOL, start=TRAIN_START_DATE, end=TEST_END_DATE)
    full_df = full_df[['Close']]
    full_df.dropna(inplace=True)
    full_df.index = pd.to_datetime(full_df.index)

    train_df = full_df.loc[TRAIN_START_DATE:TRAIN_END_DATE]
    test_df = full_df.loc[TEST_START_DATE:TEST_END_DATE]
    return train_df, test_df, full_df
```

```python
def get_user_date():
    while True:
        date_input = input("Enter a date (YYYY-MM-DD): ")
        if is_valid_date(date_input):
            return datetime.datetime.strptime(date_input, "%Y-%m-%d").date()
```

Step 5: Build Base RNN Model

```python
def build_model(input_shape, learning_rate=0.001): # Added learning_rate parameter
    model = Sequential()
    model.add(SimpleRNN(units=50, activation='tanh', input_shape=input_shape))
    model.add(Dense(1))
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate) # Use custom learning rate
    model.compile(optimizer=optimizer, loss='mean_squared_error')
    return model
```

Step 6: Reinforcement Learning Agent and Fine – Tuning Logic

```python
class RLAgent:
    """
    A simple RL Agent to fine-tune the RNN model.
    The "environment" is the time series data, and the "action" is the prediction.
    The reward is based on prediction accuracy.
    """
    def __init__(self, model, learning_rate=0.001):
        self.model = model
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate) # Pass learning_rate to RLAgent

    @tf.function
    def train_step(self, state, target):
        """
        Performs a single training step to fine-tune the model weights.
        """
        with tf.GradientTape() as tape:
            prediction = self.model(state, training=True)
            # Use Mean Squared Error directly as the loss to minimize
            loss = tf.reduce_mean(tf.square(tf.cast(target, tf.float32) - prediction))

        grads = tape.gradient(loss, self.model.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
        return loss # Return the mean loss for the batch
```

```python
def fine_tune_with_rl(model, X_train, y_train, epochs=5, learning_rate=0.001, batch_size=32):
    """
    Fine- tunes the pre-trained RNN model using a simple reinforcement learning loop with batched data.
    """
    print("\n--- Starting Reinforcement Learning Fine-Tuning (Batched) ---")
    agent = RLAgent(model, learning_rate=learning_rate)

    # Create a TensorFlow Dataset and batch it
    dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
    dataset = dataset.batch(batch_size)

    num_batches = tf.data.experimental.cardinality(dataset).numpy()

    for epoch in range(epochs):
        total_loss = 0
        batch_count = 0
        for batch_x, batch_y in dataset:
            # Ensure batch_x has the correct shape (batch_size, sequence_length, 1)
            batch_x_reshaped = tf.reshape(batch_x, (tf.shape(batch_x)[0], tf.shape(batch_x)[1], 1))
            loss = agent.train_step(batch_x_reshaped, batch_y)
            total_loss += loss.numpy() # Sum up the mean loss for each batch
            batch_count += 1

        # Calculate average loss per epoch
        average_epoch_loss = total_loss / num_batches if num_batches > 0 else 0
        print(f"RL Epoch {epoch + 1}/{epochs}, Average Loss per Batch: {average_epoch_loss:.6f}") # Print average loss

    print("--- RL Fine-Tuning Complete ---")
    return model
```

Step 7: Sequence Generation and Prediction for a Date

```python
def create_test_sequences(train_df, test_df, scaler):
    combined = pd.concat([train_df, test_df])
    scaled = scaler.transform(combined[['Close']])
    X_test = []
    test_dates = test_df.index.date

    for i in range(len(train_df), len(combined)):
        if i - SEQUENCE_LENGTH < 0:
            continue
        seq = scaled[i - SEQUENCE_LENGTH:i]
        X_test.append(seq)

    X_test = np.array(X_test)
    return X_test, test_dates
```

```python
def predict_price_on_test(model, full_df, scaler, user_date):
    all_dates = full_df.index.date
    try:
        idx = list(all_dates).index(user_date)
    except ValueError:
        print("User date not found in data.")
        return None, None, user_date

    if idx < SEQUENCE_LENGTH:
        print("Not enough data to predict for this date.")
        return None, None, user_date

    input_seq = scaler.transform(full_df[['Close']].values)[idx - SEQUENCE_LENGTH:idx]
    input_seq = np.reshape(input_seq, (1, SEQUENCE_LENGTH, 1))

    predicted_scaled = model.predict(input_seq, verbose=0)
    predicted_price = scaler.inverse_transform(predicted_scaled)[0][0]

    actual_price = float(full_df.iloc[idx]['Close']) if idx < len(full_df) else None
    return predicted_price, actual_price, full_df.index[idx].date()
```

Step 8: Plotting Functions (Loss and Prediction)

```python
# Plot the result
def plot_results(df, predicted_date, predicted_price, actual_price):
    plt.figure(figsize=(12, 6))
    plt.plot(df.index, df['Close'], label='Historical Close Price',zorder=1)
    #plt.axvline(pd.to_datetime(predicted_date), color='red', linestyle='--', label='Predicted Date')
    plt.scatter(pd.to_datetime(predicted_date), predicted_price, color='orange', label='Predicted Price',zorder=2)
    if actual_price is not None:
        plt.scatter(pd.to_datetime(predicted_date), actual_price, color='green', label='Actual Price',zorder=3, s=100)
    plt.title("INFOSYS Stock Price with Prediction (yfinance)")
    plt.xlabel("Date")
    plt.ylabel("Closing Price (₹)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

```python
#Plotting error with no. of epochs
def plot_training_loss(history):
    plt.figure(figsize=(12, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    if 'val_loss' in history.history:
        plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title("Training and Validation Loss Over Epochs (RNN)")
    plt.xlabel("Epoch no.")
    plt.ylabel("Loss (MSE)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Step 9: Main Logic (Model Training, RL Fine-Tuning, Metrics)

```python
def main():
    print("=== INFOSYS LSTM Stock Price Predictor (Test Phase June-July 2025) ===")
    user_date = get_user_date()

    # Step 1: Get stock data
    train_df, test_df, full_df = get_stock_data()

    # Step 2: Preprocess
    scaler = MinMaxScaler()
    scaled_train = scaler.fit_transform(train_df[['Close']])
    X, y = [], []
    for i in range(SEQUENCE_LENGTH, len(scaled_train)):
        X.append(scaled_train[i - SEQUENCE_LENGTH:i])
        y.append(scaled_train[i])
    X = np.array(X)
    y = np.array(y)

    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, shuffle=False)
    #early_stop = EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)
    # Step 3: Build and train model
    model = build_model((X_train.shape[1], 1), learning_rate=0.0005) # Reduced Adam learning rate
    print("Training RNN model...")
    history = model.fit(X_train, y_train, epochs=10000, batch_size=64, validation_data=(X_val, y_val), verbose = 1) #callbacks = [early_stop], verbose=1)

    #print(f"\nBest weights restored from epoch: {early_stop.best_epoch + 1}")

    # Step 4: Plot training loss
    plot_training_loss(history)

    # Step 5: Fine-tune with Reinforcement Learning
    rl_tuned_model = fine_tune_with_rl(model, X_train, y_train, epochs=200, learning_rate=0.0001, batch_size=32)
```

```python
# Predict on training data to evaluate performance
y_train_pred_scaled = rl_tuned_model.predict(X_train, verbose=0)
y_train_pred = scaler.inverse_transform(y_train_pred_scaled)
y_true_train = scaler.inverse_transform(y_train.reshape(-1, 1))

# Calculate and print training metrics
mae_train = mean_absolute_error(y_true_train, y_train_pred)
mse_train = mean_squared_error(y_true_train, y_train_pred)
rmse_train = np.sqrt(mse_train)
r2_train = r2_score(y_true_train, y_train_pred)
mape_train = mean_absolute_percentage_error(y_true_train, y_train_pred) * 100
accuracy_train = 100 - mape_train
mse_percent_train = (mse_train / np.mean(y_true_train)) * 100 if np.mean(y_true_train) != 0 else np.nan
mae_percent_train = (mae_train / np.mean(y_true_train)) * 100 if np.mean(y_true_train) != 0 else np.nan
print("\n Training Set Performance Metrics (RNN):")
print(f" Mean Absolute Error (MAE) : ₹{mae_train:.2f}")
print(f" Mean Squared Error (MSE)  : ₹{mse_train:.2f}")
print(f" Root Squared Error (RMSE) : ₹{rmse_train:.2f}")
print(f" R² Score                  : {r2_train:.4f}")
print(f" Mean Absolute Percentage Error (MAPE) : {mape_train:.2f}%")
print(f" Accuracy (100 - MAPE) : {accuracy_train:.2f}%")
print(f" MSE Percentage of Average Price : {mse_percent_train:.2f}%")
print(f" MAE Percentage of Average Price : {mae_percent_train:.2f}%")
```

```python
# Predict on validation set
y_val_pred_scaled = rl_tuned_model.predict(X_val, verbose=0)
y_val_pred = scaler.inverse_transform(y_val_pred_scaled)
y_val_true = scaler.inverse_transform(y_val.reshape(-1, 1))

# Calculate and print validation metrics
mae_val = mean_absolute_error(y_val_true, y_val_pred)
mse_val = mean_squared_error(y_val_true, y_val_pred)
rmse_val = np.sqrt(mse_val)
r2_val = r2_score(y_val_true, y_val_pred)
mape_val = mean_absolute_percentage_error(y_val_true, y_val_pred) * 100
accuracy_val = 100 - mape_val
mse_percent_val = (mse_val / np.mean(y_val_true)) * 100 if np.mean(y_val_true) != 0 else np.nan
mae_percent_val = (mae_val / np.mean(y_val_true)) * 100 if np.mean(y_val_true) != 0 else np.nan
print("\n Validation Set Performance Metrics (RNN):")
print(f" Mean Absolute Error (MAE) : ₹{mae_val:.2f}")
print(f" Mean Squared Error (MSE)  : ₹{mse_val:.2f}")
print(f" Root Squared Error (RMSE) : ₹{rmse_val:.2f}")
print(f" R² Score                  : {r2_val:.4f}")
print(f" Mean Absolute Percentage Error (MAPE) : {mape_val:.2f}%")
print(f" Accuracy (100 - MAPE) : {accuracy_val:.2f}%")
print(f" MSE Percentage of Average Price: {mse_percent_val:.2f}%")
print(f" MAE Percentage of Average Price: {mae_percent_val:.2f}%")
```

```python
# Step 6: Predict for user-input test date
predicted_price, actual_price, predicted_date = predict_price_on_test(rl_tuned_model, full_df, scaler, user_date)
if predicted_price is not None:
    print(f"\n Prediction for {predicted_date}:")
    print(f" Predicted Closing Price: ₹{predicted_price:.2f}")
    if actual_price is not None:
        print(f" Actual Closing Price   : ₹{actual_price}")
    else:
        print("Actual price not available.")

    plot_results(full_df, predicted_date, predicted_price, actual_price)
else:
    print("❌ Unable to make prediction.")
```

Step 10: Test Set Predictions and Final Comparison Plot

```python
# Reuse combined full_df
combined_df = pd.concat([train_df, test_df])
scaled_combined = scaler.transform(combined_df[['Close']])

X_test_all = []
test_date_indexes = []

for i in range(len(train_df), len(combined_df)):
    if i - SEQUENCE_LENGTH < 0:
        continue
    seq = scaled_combined[i - SEQUENCE_LENGTH:i]
    X_test_all.append(seq)
    test_date_indexes.append(combined_df.index[i])

X_test_all = np.array(X_test_all)

# Predict on test sequences
y_test_pred_scaled = rl_tuned_model.predict(X_test_all, verbose=0)
y_test_pred = scaler.inverse_transform(y_test_pred_scaled)

# Actual prices
y_test_true = test_df.loc[test_date_indexes, 'Close'].values
```

```python
# Calculate and print test metrics
mae_test = mean_absolute_error(y_test_true, y_test_pred)
mse_test = mean_squared_error(y_test_true, y_test_pred)
rmse_test = np.sqrt(mse_test)
r2_test = r2_score(y_test_true, y_test_pred)
mape_test = mean_absolute_percentage_error(y_test_true, y_test_pred) * 100
accuracy_test = 100 - mape_test
mse_percent_test = (mse_test / np.mean(y_test_true)) * 100 if np.mean(y_test_true) != 0 else np.nan
mae_percent_test = (mae_test / np.mean(y_test_true)) * 100 if np.mean(y_test_true) != 0 else np.nan
print("\n Test Set Performance Metrics (RNN):")
print(f" Mean Absolute Error (MAE) : ₹{mae_test:.2f}")
print(f" Mean Squared Error (MSE)  : ₹{mse_test:.2f}")
print(f" Root Squared Error (RMSE) : ₹{rmse_test:.2f}")
print(f" R² Score                  : {r2_test:.4f}")
print(f" Mean Absolute Percentage Error (MAPE) : {mape_test:.2f}%")
print(f" Accuracy (100 - MAPE) : {accuracy_test:.2f}%")
print(f" MSE Percentage of Average Price: {mse_percent_test:.2f}%")
print(f" MAE Percentage of Average Price: {mae_percent_test:.2f}%")
```

```
# Plot actual vs predicted
plt.figure(figsize=(12, 6))
plt.plot(test_date_indexes, y_test_true, label='Actual Price')
plt.plot(test_date_indexes, y_test_pred, label='Predicted Price')
plt.title("Actual vs Predicted INFY Closing Prices (01-Jun to 01-Jul 2025)")
plt.xlabel("Date")
plt.ylabel("Closing Price (₹)")
plt.xticks(rotation=45)
plt.gca().xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'))
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Step 11: Run the Main Function

```
if __name__ == "__main__":
    main()
```

3. LSTM (Long Short Term Memory)

A Long Short-Term Memory (LSTM) network is a special kind of Recurrent Neural Network (RNN) designed to overcome the limitations of standard RNNs, especially the vanishing gradient problem when learning long-term dependencies. The LSTM architecture introduces a memory cell that can maintain information over long periods and uses three gates to regulate the flow of information: the forget gate, input gate, and output gate. The forget gate decides which information from the previous cell state Ct−1 should be discarded, the input gate determines which new information should be added to the cell state, and the output gate controls what information from the cell state is output to the next hidden state ht. Each gate uses a sigmoid activation function to output values between 0 and 1 (interpreted as how much information to let through), while the cell state is updated using element-wise multiplication and addition. The tanh activation is used for candidate values and final outputs, enabling LSTMs to learn both short- and long-term dependencies effectively, making them ideal for complex sequential tasks like language modeling, time series forecasting, and speech recognition.

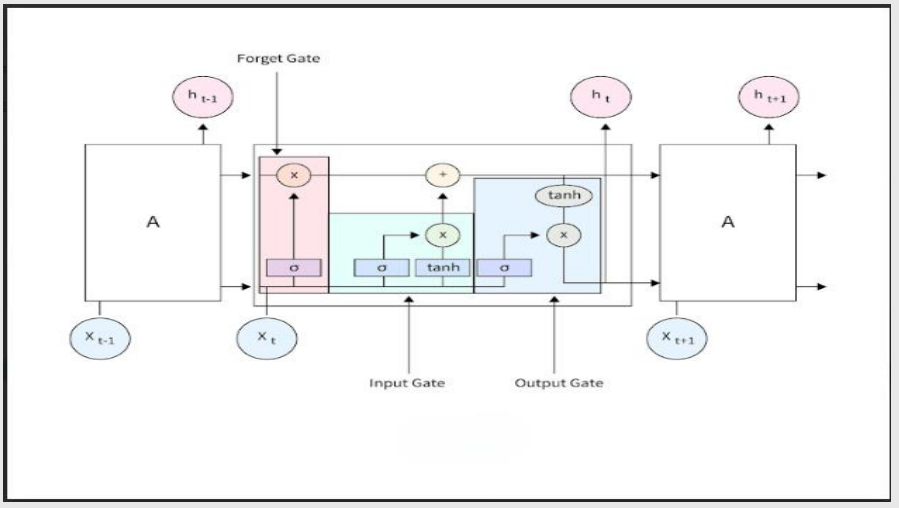


Fig 8: LSTM (Long Short Term Memory)

Snapshot of LSTM Code :-

Step 1: Importing All the necessary Library

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime
from datetime import date
import yfinance as yf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from sklearn.preprocessing import MinMaxScaler
```

Step 2: Downloading data

```
print("Downloading INFOSYS historical data...")
df = yf.download("INFY.NS", start="2020-01-01", end="2025-07-02")

if df.empty:
    print("ERROR: No data returned.")
    exit()
```

Step 3: Using only closing prices

```
df = df[["Close"]]
df.reset_index(inplace=True)
df["Date"] = pd.to_datetime(df["Date"]).dt.date
df.sort_values("Date", inplace=True)
```

Step 4: Split Tain/Test

```
train_df = df[df["Date"] <= date(2025, 5, 31)]
test_df = df[(df["Date"] >= date(2025, 6, 1)) & (df["Date"] <= date(2025, 7, 1))]
```

Step 5: Scaling the data

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_close = scaler.fit_transform(df["Close"].values.reshape(-1, 1))
```

Step 6: Creating Sequences

```
sequence_length = 60
X, y = [], []
for i in range(sequence_length, len(scaled_close)):
    X.append(scaled_close[i-sequence_length:i, 0])
    y.append(scaled_close[i, 0])
X, y = np.array(X), np.array(y)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))
```

Step 7: Building LSTM Model

```
model = Sequential()
model.add(LSTM(50, activation="tanh", input_shape=(X.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer="adam", loss="mean_squared_error")
```

Step 8: Train model with validation on test data

```
print("Training model for 7500 epochs...")
history = model.fit(
    X_train, y_train,
    epochs=7500,
    batch_size=32,
    validation_data=(X_test, y_test),
    verbose=1
)
```

Step 9: Predicting on test data

```
predicted_scaled = model.predict(X_test)
predicted = scaler.inverse_transform(predicted_scaled)
actual = scaler.inverse_transform(y_test.reshape(-1, 1))
```

Step 10: Training Set Metrics

```
train_pred_scaled = model.predict(X_train)
train_pred = scaler.inverse_transform(train_pred_scaled)
train_actual = scaler.inverse_transform(y_train.reshape(-1, 1))

mse_train = np.mean((train_pred - train_actual)**2)
mae_train = np.mean(abs(train_pred - train_actual))
mape_train = np.mean(abs(train_pred - train_actual) / train_actual) * 100
accuracy_train = 100 - mape_train
avg_train_price = np.mean(train_actual)
mse_train_percent = (mse_train / avg_train_price) * 100
mae_train_percent = (mae_train / avg_train_price) * 100
```

Step 11: Test Set Metrics

```
mse = np.mean((predicted - actual) ** 2)
mae = np.mean(abs(predicted - actual))
avg_price = np.mean(actual)
mse_percent = (mse / avg_price) * 100
mae_percent = (mae / avg_price) * 100
mape = np.mean(abs(predicted - actual) / actual) * 100
accuracy = 100 - mape
```

Step 12: Printing the metrics

```
print("\n========== Training Set Metrics ==========")
print(f"MSE: {mse_train:.4f} ({mse_train_percent:.4f}% of avg price)")
print(f"MAE: {mae_train:.4f} ({mae_train_percent:.4f}% of avg price)")
print(f"Accuracy: {accuracy_train:.2f}%")

print("\n========== Test Set Metrics ==========")
print(f"MSE: {mse:.4f} ({mse_percent:.4f}% of avg price)")
print(f"MAE: {mae:.4f} ({mae_percent:.4f}% of avg price)")
print(f"Accuracy: {accuracy:.2f}%")
print("========================================\n")
```

Step 13: Prompt user for data

```
while True:
    user_input = input("Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): ").strip()
    try:
        user_date = datetime.datetime.strptime(user_input, "%Y-%m-%d").date()
    except ValueError:
        print("Invalid format. Try again.")
        continue
    if not (date(2025, 6, 1) <= user_date <= date(2025, 7, 1)):
        print("Date out of range.")
        continue
    if user_date not in test_dates:
        print("Date not in test data. Please choose a date from the available test dates.")
        print(f"Available test dates: {test_dates[0]} to {test_dates[-1]}")
        continue
    break
```

Step 14: Finding Index

```
idx = np.where(test_dates == user_date)[0][0]
pred_price = predicted[idx][0]
act_price = actual[idx][0]
error_pct = abs(pred_price - act_price) / act_price * 100
```

Step 15: Displaying the prediction

```
print(f"\nFor {user_date}:")
print(f"Predicted closing price: ₹{pred_price:.2f}")
print(f"Actual closing price: ₹{act_price:.2f}")
print(f"Absolute error: {error_pct:.2f}%")
```

Step 16: Plotting the Actual Vs Predicted Closing Prices

```
plt.figure(figsize=(14, 6))
plt.plot(test_dates, actual, label="Actual Closing Price")
plt.plot(test_dates, predicted, label="Predicted Closing Price")
plt.axvline(x=np.datetime64(user_date), color="red", linestyle="--", label="Selected Date")
plt.xlabel("Date")
plt.ylabel("Price (₹)")
plt.title("INFOSYS - Actual vs Predicted Closing Prices (Test Period)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Step 17: Combining the loss plot

```
plt.figure(figsize=(12, 6))
plt.plot(history.history["loss"], label="Training Loss")
plt.plot(history.history["val_loss"], label="Testing Loss")
plt.title("Training and Testing Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

4. LSTM (Long Short Term Memory) Fine Tuned with Reinforcement Learning

Long Short-Term Memory (LSTM) networks, when integrated with Reinforcement Learning (RL), provide a powerful mechanism for decision-making in environments with sequential and time-dependent data. LSTM networks are a type of Recurrent Neural Network (RNN) designed to retain long-term dependencies, making them ideal for learning patterns across time. In an RL setup, the LSTM acts as part of the agent's architecture, helping it to remember and process past states, actions, and rewards, which is critical when the environment's dynamics depend on past events. The agent, guided by the LSTM's memory capabilities, selects actions based not only on the current observation but also on historical context, which improves performance in partially observable or delayed reward scenarios. This combination is particularly useful in applications like dialogue systems, financial prediction, or game playing, where understanding sequences and long-term outcomes is essential.

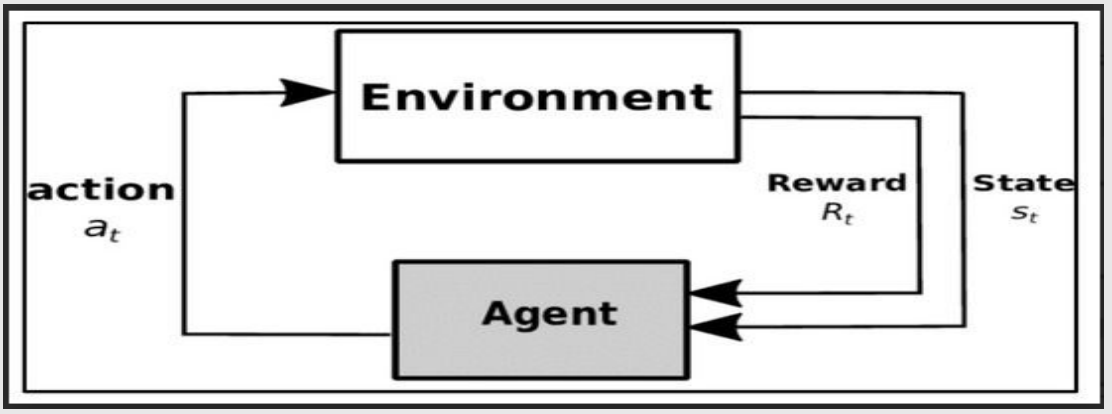


Fig 9: Process of Reinforcement Learning

The diagram illustrates the fundamental architecture of Reinforcement Learning (RL), which involves an interaction between an **Agent** and an **Environment**. At each time step t, the agent observes the current **state of** the environment and takes an **action based** on a policy. The environment then responds by transitioning to a new state and providing a **reward** Rt that evaluates the effectiveness of the agent's action. This reward serves as feedback, guiding the agent to learn an optimal strategy that maximizes the cumulative reward over time. The loop continues, allowing the agent to refine its policy through trial and error, which is central to learning in dynamic and uncertain environments.

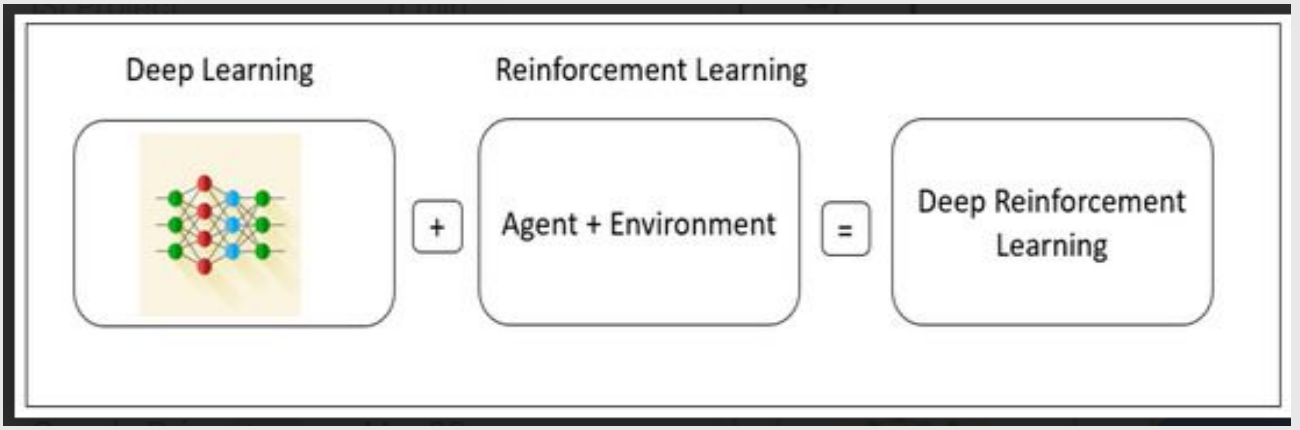


Fig 10: LSTM fine tuned with Reinforcement Learning

Snapshot of LSTM Fine Tuned With RL Code :-

Step 1: Importing All the necessary Libraries

```
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import SimpleRNN, LSTM
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score, mean_absolute_percentage_error
from matplotlib.dates import DateFormatter
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input
import numpy as np
import pandas as pd
import datetime
import holidays
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import warnings
```

Step 2: Define constants: symbol, date range, sequence length

```
np.random.seed(42)
tf.random.set_seed(42)

SYMBOL = "INFY.NS"
SEQUENCE_LENGTH = 60

TRAIN_START_DATE = "2020-06-01"
TRAIN_END_DATE = "2025-05-31"
TEST_START_DATE = "2025-06-01"
TEST_END_DATE = "2025-07-01"
```

Step 3: Filtering out trading holidays

```python
# Get Indian trading holidays
def get_nse_holidays():
    in_holidays = holidays.India(years=range(2020, 2026))
    return set(in_holidays.keys())

TRADING_HOLIDAYS = get_nse_holidays()
```

Step 4: Fetching stock data from yfinance and only including closing prices from dataframe

```python
def get_stock_data():
    print("Fetching data from Yahoo Finance...")
    full_df = yf.download(SYMBOL, start=TRAIN_START_DATE, end=TEST_END_DATE)
    full_df = full_df[['Close']]
    full_df.dropna(inplace=True)
    full_df.index = pd.to_datetime(full_df.index)

    train_df = full_df.loc[TRAIN_START_DATE:TRAIN_END_DATE]
    test_df = full_df.loc[TEST_START_DATE:TEST_END_DATE]
    return train_df, test_df, full_df
```

Step 5: Defining the LSTM model structure

```python
def build_model(input_shape, learning_rate=0.001): # Added learning_rate parameter
    model = Sequential()
    model.add(LSTM(units=50, activation='tanh', input_shape=input_shape))
    model.add(Dense(1))
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='mean_squared_error')
    return model
```

Step 6: Defining RL agent and training steps

```python
def __init__(self, model, learning_rate=0.001):
    self.model = model
    self.optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)

@tf.function
def train_step(self, state, target):
    """
    Performs a single training step to fine-tune the model weights.
    """
    with tf.GradientTape() as tape:
        prediction = self.model(state, training=True)
        # Use Mean Squared Error directly as the loss to minimize
        loss = tf.reduce_mean(tf.square(tf.cast(target, tf.float32) - prediction))

    grads = tape.gradient(loss, self.model.trainable_variables)
    self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))
    return loss # Return the mean loss for the batch
```

Step 7: Fine tuning function using RL agent

```python
print("\n--- Starting Reinforcement Learning Fine-Tuning (Batched) ---")
agent = RLAgent(model, learning_rate=learning_rate)

# Create a TensorFlow Dataset and batch it
dataset = tf.data.Dataset.from_tensor_slices((X_train, y_train))
dataset = dataset.batch(batch_size)

num_batches = tf.data.experimental.cardinality(dataset).numpy()

for epoch in range(epochs):
    total_loss = 0
    batch_count = 0
    for batch_x, batch_y in dataset:
        # Ensure batch_x has the correct shape (batch_size, sequence_length, 1)
        batch_x_reshaped = tf.reshape(batch_x, (tf.shape(batch_x)[0], tf.shape(batch_x)[1], 1))
        loss = agent.train_step(batch_x_reshaped, batch_y)
        total_loss += loss.numpy() # Sum up the mean loss for each batch
        batch_count += 1

    # Calculate average loss per epoch
    average_epoch_loss = total_loss / num_batches if num_batches > 0 else 0
    print(f"RL Epoch {epoch + 1}/{epochs}, Average Loss per Batch: {average_epoch_loss:.6f}") # Print average loss

print("--- RL Fine-Tuning Complete ---")
return model
```

Step 8: Creating test sequences for the model

```python
def create_test_sequences(train_df, test_df, scaler):
    combined = pd.concat([train_df, test_df])
    scaled = scaler.transform(combined[['Close']])
    X_test = []
    test_dates = test_df.index.date

    for i in range(len(train_df), len(combined)):
        if i - SEQUENCE_LENGTH < 0:
            continue
        seq = scaled[i - SEQUENCE_LENGTH:i]
        X_test.append(seq)

    X_test = np.array(X_test)
    return X_test, test_dates
```

Step 9: Predicting price on user input date, within test range

```python
input_seq = scaler.transform(full_df[['Close']].values)[idx - SEQUENCE_LENGTH:idx]
input_seq = np.reshape(input_seq, (1, SEQUENCE_LENGTH, 1))

predicted_scaled = model.predict(input_seq, verbose=0)
predicted_price = scaler.inverse_transform(predicted_scaled)[0][0]

actual_price = float(full_df.iloc[idx]['Close']) if idx < len(full_df) else None
return predicted_price, actual_price, full_df.index[idx].date()
```

Step 10: Plotting MSE loss over epochs

```python
def plot_training_loss(history):
    plt.figure(figsize=(12, 6))
    plt.plot(history.history['loss'], label='Training Loss')
    if 'val_loss' in history.history:
        plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title("Training and Validation Loss Over Epochs (LSTM)") # Changed title
    plt.xlabel("Epoch no.")
    plt.ylabel("Loss (MSE)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Step 12: Fine-tune with RL agent for 100 episodes

```python
# Step 5: Fine-tune with Reinforcement Learning
rl_tuned_model = fine_tune_with_rl(model, X_train, y_train, epochs=100, learning_rate=0.0001, batch_size=32)
```

Step 13: Predict price for user input date

```
predicted_price, actual_price, predicted_date = predict_price_on_test(rl_tuned_model, full_df, scaler, user_date)
if predicted_price is not None:
    print(f"\n Prediction for {predicted_date}:")
    print(f" Predicted Closing Price: ₹{predicted_price:.2f}")
    if actual_price is not None:
        print(f" Actual Closing Price   : ₹{actual_price}")
    else:
        print("Actual price not available.")

    plot_results(full_df, predicted_date, predicted_price, actual_price)
else:
    print("❌ Unable to make prediction.")
```

Step 14: Predict and plot prices for all dates within test range

```
# Reuse combined full_df
combined_df = pd.concat([train_df, test_df])
scaled_combined = scaler.transform(combined_df[['Close']])

X_test_all = []
test_date_indexes = []

for i in range(len(train_df), len(combined_df)):
    if i - SEQUENCE_LENGTH < 0:
        continue
    seq = scaled_combined[i - SEQUENCE_LENGTH:i]
    X_test_all.append(seq)
    test_date_indexes.append(combined_df.index[i])

X_test_all = np.array(X_test_all)

# Predict on test sequences
y_test_pred_scaled = rl_tuned_model.predict(X_test_all, verbose=0)
y_test_pred = scaler.inverse_transform(y_test_pred_scaled)

# Actual prices
y_test_true = test_df.loc[test_date_indexes, 'Close'].values
```

Step 15: Calculating and printing test set performance metrics

```
# Calculate and print test metrics
mae_test = mean_absolute_error(y_test_true, y_test_pred)
mse_test = mean_squared_error(y_test_true, y_test_pred)
rmse_test = np.sqrt(mse_test)
r2_test = r2_score(y_test_true, y_test_pred)
mape_test = mean_absolute_percentage_error(y_test_true, y_test_pred) * 100
accuracy_test = 100 - mape_test
mse_percent_test = (mse_test / np.mean(y_test_true)) * 100 if np.mean(y_test_true) != 0 else np.nan
mae_percent_test = (mae_test / np.mean(y_test_true)) * 100 if np.mean(y_test_true) != 0 else np.nan
print("\n Test Set Performance Metrics (LSTM):")
print(f" Mean Absolute Error (MAE) : ₹{mae_test:.2f}")
print(f" Mean Squared Error (MSE)  : ₹{mse_test:.2f}")
print(f" Root Squared Error (RMSE) : ₹{rmse_test:.2f}")
print(f" R² Score                  : {r2_test:.4f}")
print(f" Mean Absolute Percentage Error (MAPE) : {mape_test:.2f}%")
print(f" Accuracy (100 - MAPE) : {accuracy_test:.2f}%")
print(f" MSE Percentage of Average Price: {mse_percent_test:.2f}%")
print(f" MAE Percentage of Average Price: {mae_percent_test:.2f}%")
```

RESULTS

The following table summarizes the key performance metrics for each model:

1. **SimpleRNN** :

The results obtained from the SimpleRNN model are as followed :

Train Set Performance Metrics :

Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	39.3155%	1.2507%	98.73%	1632.90	1627.43
500	34.9755%	1.1503%	98.82%	1632.90	1642.50
1000	35.6484%	1.1730%	98.80%	1632.90	1636.28
2000	36.7051%	1.1998%	98.77%	1632.90	1638.53
5000	24.7128%	1.0163%	98.94%	1632.90	1650.30
7500	36.1829%	1.1809%	98.79%	1632.90	1645.30
10000	29.2033%	1.1054%	98.85%	1632.90	1640.05

Test Set Performance Metrics:

Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	26.0151%	0.9640%	99.04%	1632.90	1627.43
500	17.1933%	0.8198%	99.18%	1632.90	1642.50
1000	19.5524%	0.8404%	99.16%	1632.90	1636.28
2000	20.3558%	0.8409%	99.16%	1632.90	1638.90
5000	25.7490%	1.0363%	98.96%	1632.90	1650.30
7500	21.4106%	0.9091%	99.09%	1632.90	1645.30
10000	23.9973%	0.9411%	99.06%	1632.90	1640.05

- For 100 Epochs :-

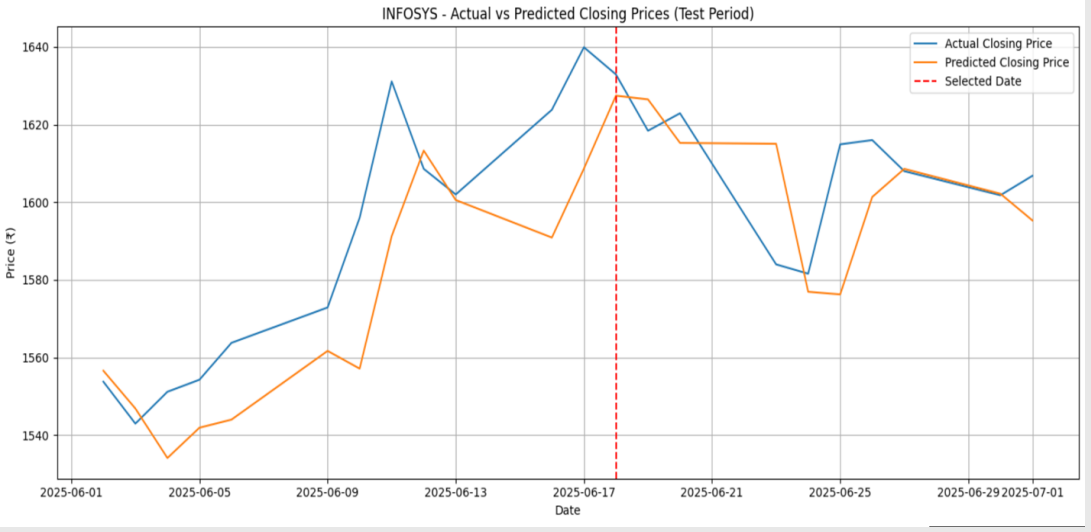
i) Performance metrics for 100 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 544.1213 (39.3155% of avg price)
MAE: 17.3089 (1.2507% of avg price)
Accuracy: 98.73%

========== Test Set Metrics ==========
MSE: 415.3867 (26.0151% of avg price)
MAE: 15.3924 (0.9640% of avg price)
Accuracy: 99.04%
=======================================
```

ii) Predicted vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1627.43
Actual closing price: ₹1632.90
Absolute error: 0.33%
```

iii) Actual Vs Predicted Graph Plot :



- **For 500 Epochs :-**

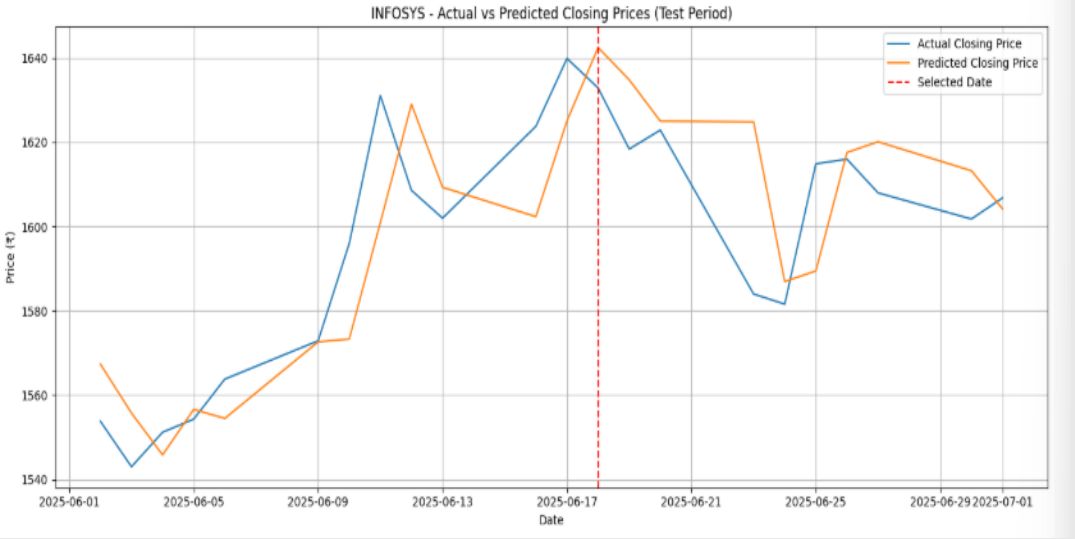
  i) Performance metrics for 500 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 544.1213 (39.3155% of avg price)
MAE: 17.3089 (1.2507% of avg price)
Accuracy: 98.73%

========== Test Set Metrics ==========
MSE: 415.3867 (26.0151% of avg price)
MAE: 15.3924 (0.9640% of avg price)
Accuracy: 99.04%
=======================================
```

  ii) Predicted Vs Actual Closing Price:

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1627.43
Actual closing price: ₹1632.90
Absolute error: 0.33%
```

  iii) Actual Vs Predicted Graph Plot :



- **For 1000 Epochs :-**

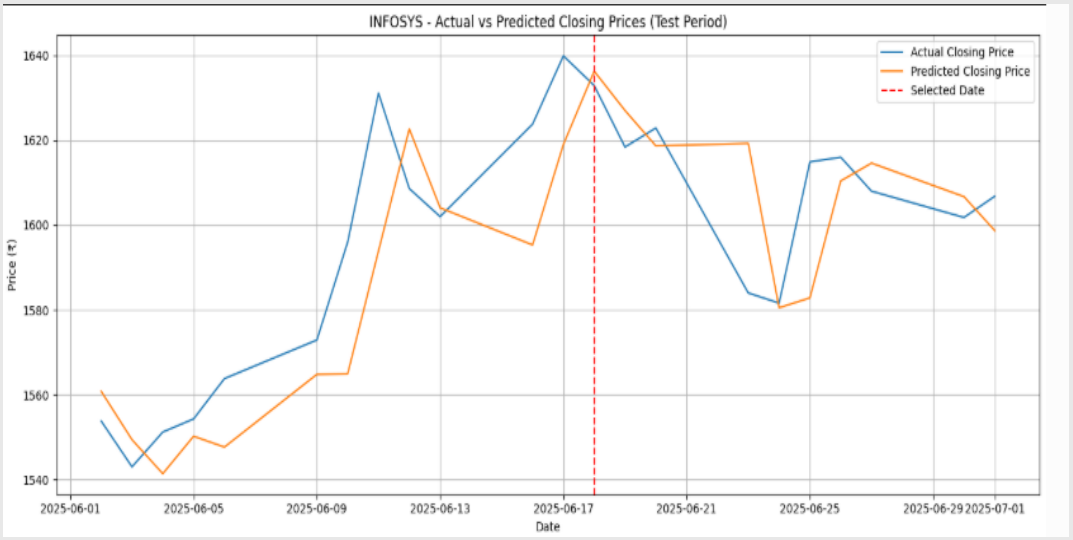
  i) Performance Metrics for 1000 epochs:

```
========== Training Set Metrics ==========
MSE: 493.3683 (35.6484% of avg price)
MAE: 16.2340 (1.1730% of avg price)
Accuracy: 98.80%

========== Test Set Metrics ==========
MSE: 312.1957 (19.5524% of avg price)
MAE: 13.4182 (0.8404% of avg price)
Accuracy: 99.16%
=======================================
```

ii) Predicted Vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1636.28
Actual closing price: ₹1632.90
Absolute error: 0.21%
```

iii) Actual Vs Predicted Graph Plot :



- **For 2000 Epochs :-**

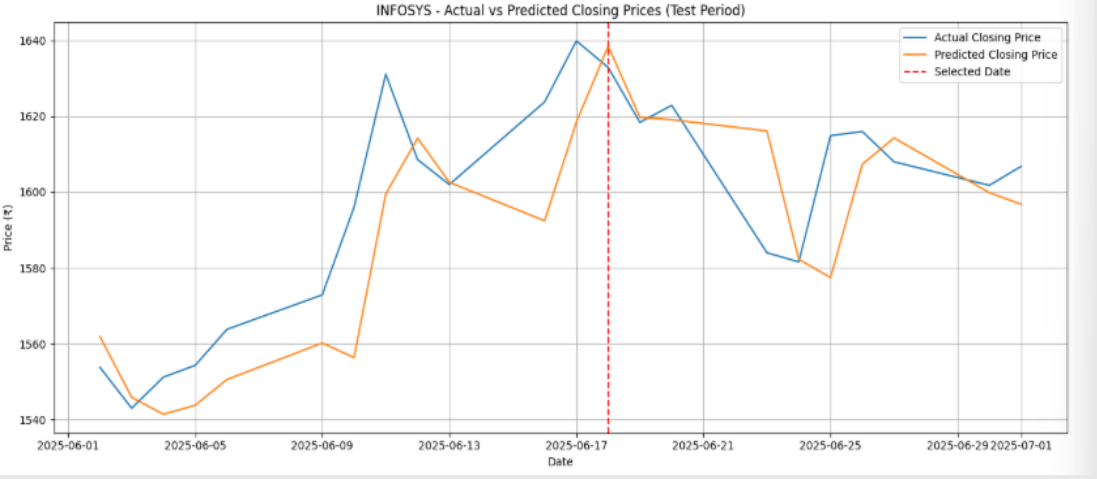
i) Performance Metrics for 2000 Epochs :

```
========== Training Set Metrics ==========
MSE: 507.9934 (36.7051% of avg price)
MAE: 16.6054 (1.1998% of avg price)
Accuracy: 98.77%

========== Test Set Metrics ==========
MSE: 334.6042 (20.9558% of avg price)
MAE: 13.4275 (0.8409% of avg price)
Accuracy: 99.16%
=========================================
```

ii) Predicted Vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1638.53
Actual closing price: ₹1632.90
Absolute error: 0.35%
```

iii) Actual Vs Predicted Graph Plot :



- **For 5000 Epochs :-**

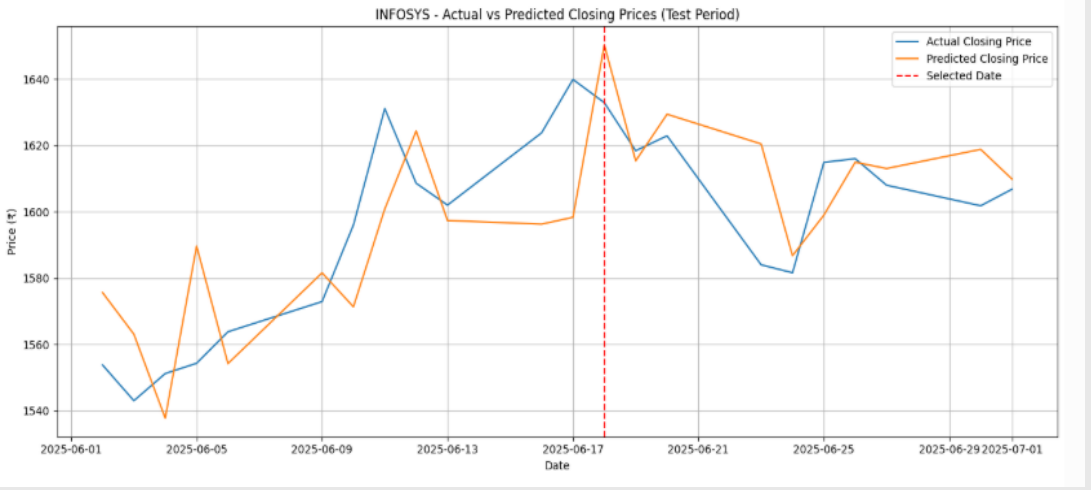
i) Performance Metrics for 5000 Epochs :

```
========== Training Set Metrics ==========
MSE: 342.0216 (24.7128% of avg price)
MAE: 14.0657 (1.0163% of avg price)
Accuracy: 98.94%

========== Test Set Metrics ==========
MSE: 411.1371 (25.7490% of avg price)
MAE: 16.5462 (1.0363% of avg price)
Accuracy: 98.96%
========================================
```

ii) Predicted Vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1650.30
Actual closing price: ₹1632.90
Absolute error: 1.07%
```

iii) Actual Vs Predicted Graph Plot :



- **For 7500 Epochs :-**

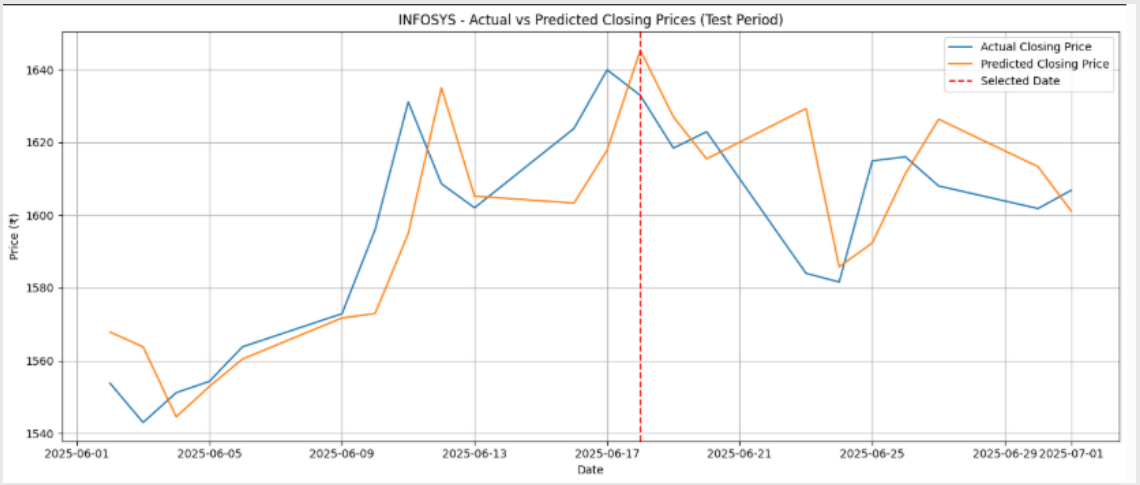
i) Performance Metrics for 7500 Epochs :-

```
========== Training Set Metrics ==========
MSE: 500.7663 (36.1829% of avg price)
MAE: 16.3430 (1.1809% of avg price)
Accuracy: 98.79%

========== Test Set Metrics ==========
MSE: 341.8655 (21.4106% of avg price)
MAE: 14.5152 (0.9091% of avg price)
Accuracy: 99.09%
=======================================
```

ii) Predicted Vs Actual Closing Price :-

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1645.30
Actual closing price: ₹1632.90
Absolute error: 0.76%
```

iii) Actual Vs Predicted Graph Plot :-



- **For 10000 Epochs :-**

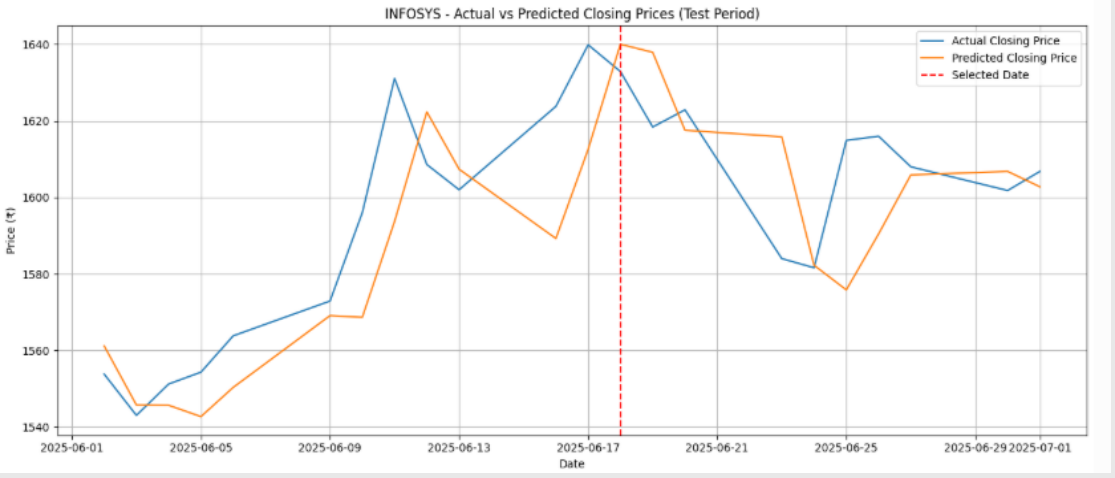
i) Performance Metrics for 10000 Epochs :-

```
========== Training Set Metrics ==========
MSE: 404.1694 (29.2033% of avg price)
MAE: 15.2986 (1.1054% of avg price)
Accuracy: 98.85%

========== Test Set Metrics ==========
MSE: 383.1675 (23.9973% of avg price)
MAE: 15.0274 (0.9411% of avg price)
Accuracy: 99.06%
========================================
```

ii) Predicted Vs Actual Closing Price :-

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1640.05
Actual closing price: ₹1632.90
Absolute error: 0.44%
```

iii) Actual Vs Predicted Graph Plot :-



2. **RNN fine-tuned with Reinforcement Learning**:

The results obtained from the RNN + RL model are as follows :

Train Set Performance Metrics :

Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	36.48%	1.19%	98.80%	1632.90	1644.12
500	32.24%	1.11%	98.89%	1632.90	1641.39
1000	22.85%	0.95%	99.04%	1632.90	1630.35
2000	9.73%	0.64%	99.34%	1632.90	1632.87
5000	1.46%	0.25%	99.74%	1632.90	1611.64
7000	0.96%	0.21%	99.79%	1632.90	1653.48
10000	0.67%	0.18%	99.82%	1632.90	1636.99

Test Set Performance Metrics:

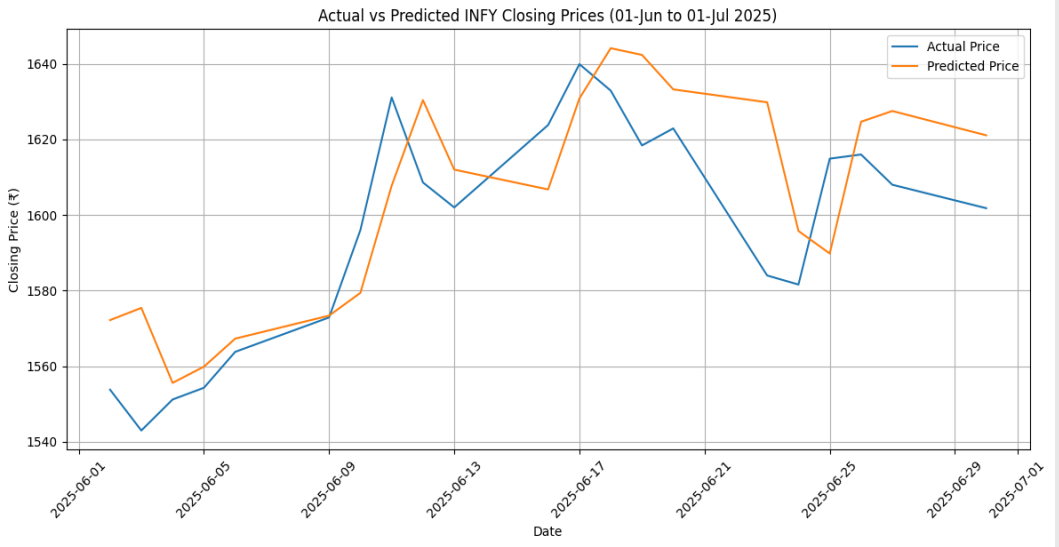
Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	23.22%	1.02%	98.98%	1632.90	1644.12
500	53.38%	1.28%	98.69%	1632.90	1641.39
1000	39.93%	1.25%	98.75%	1632.90	1630.35
2000	51.94%	1.50%	98.49%	1632.90	1632.87
5000	58.04%	1.54%	98.47%	1632.90	1611.64
7000	42.59%	1.35%	98.65%	1632.90	1653.48
10000	49.55%	1.30%	98.70%	1632.90	1636.99

For 100 Epochs :-

i) Performance metrics for 100 epochs are as follows:

```
Test Set Performance Metrics (RNN):
Mean Absolute Error (MAE) : ₹16.22
Mean Squared Error (MSE)  : ₹370.60
Root Squared Error (RMSE) : ₹19.25
R² Score                  : 0.5653
Mean Absolute Percentage Error (MAPE) : 1.02%
Accuracy (100 - MAPE) : 98.98%
Mean Squared Error as percentage of average price(%) : 23.22%
Mean Absolute Error as percentage of average price(%) : 1.02%
```

ii) The predicted closing price plot is as follows:

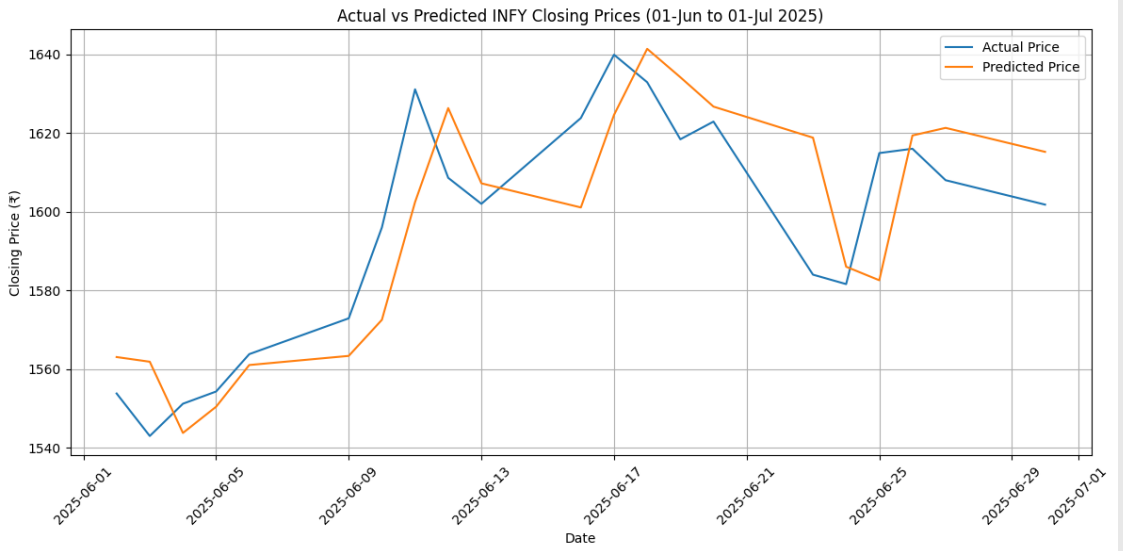


For 500 Epochs :-

i) Performance metrics for 500 epochs are as follows:

```
Test Set Performance Metrics (RNN):
Mean Absolute Error (MAE) : ₹14.03
Mean Squared Error (MSE)  : ₹288.31
Root Squared Error (RMSE) : ₹16.98
R² Score                  : 0.6619
Mean Absolute Percentage Error (MAPE) : 0.88%
Accuracy (100 - MAPE) : 99.12%
Mean Squared Error as percentage of average price(%) : 18.06%
Mean Absolute Error as percentage of average price(%) : 0.88%
```

ii) The predicted closing price plot is as follows:

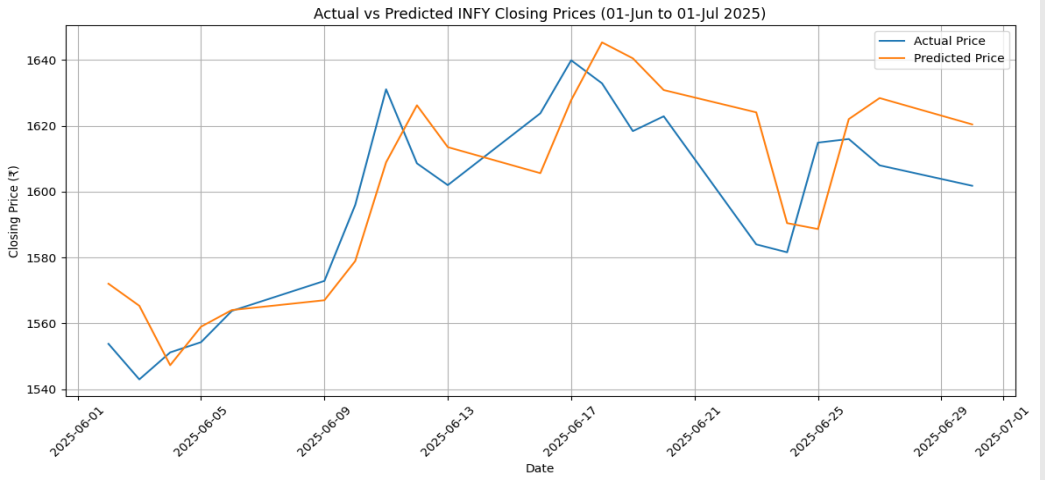


For 1000 Epochs :-

i) Performance metrics for 1000 epochs are as follows:

```
Test Set Performance Metrics (RNN):
Mean Absolute Error (MAE) : ₹15.09
Mean Squared Error (MSE)  : ₹309.10
Root Squared Error (RMSE) : ₹17.58
R² Score                  : 0.6375
Mean Absolute Percentage Error (MAPE) : 0.94%
Accuracy (100 - MAPE) : 99.06%
Mean Squared Error as percentage of average price(%) : 19.36%
Mean Absolute Error as percentage of average price(%) : 0.95%
```

ii) The predicted closing price plot is as follows:

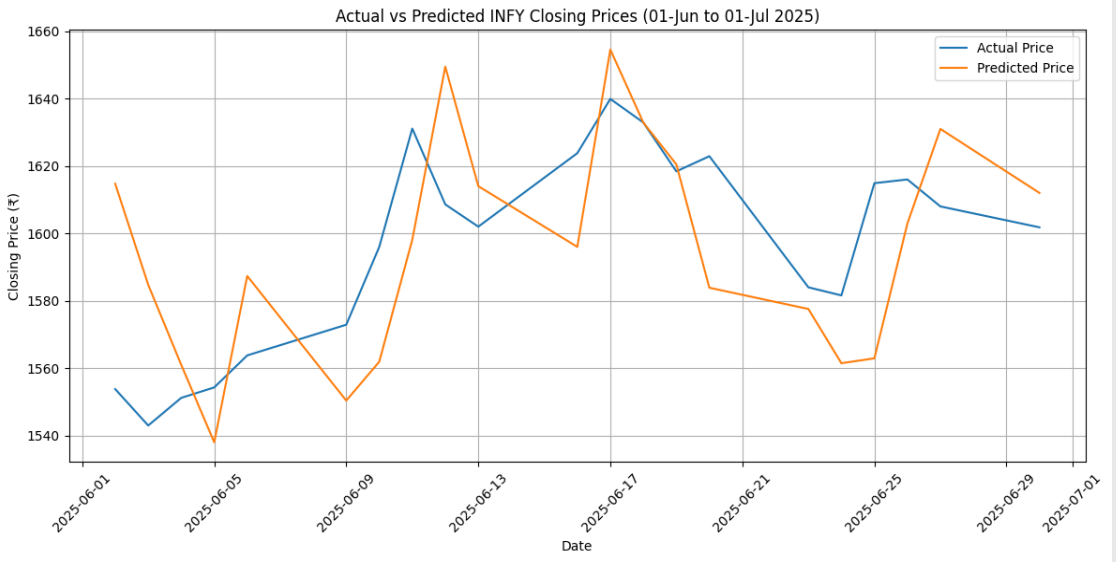


For 2500 Epochs :-

i) Performance metrics for 2000 epochs are as follows:

```
Test Set Performance Metrics (RNN):
Mean Absolute Error (MAE) : ₹23.96
Mean Squared Error (MSE)  : ₹829.03
Root Squared Error (RMSE) : ₹28.79
R² Score                  : 0.0277
Mean Absolute Percentage Error (MAPE) : 1.51%
Accuracy (100 - MAPE) : 98.49%
MSE Percentage of Average Price: 51.94%
MAE Percentage of Average Price: 1.50%
```

ii) The predicted closing price plot is as follows:

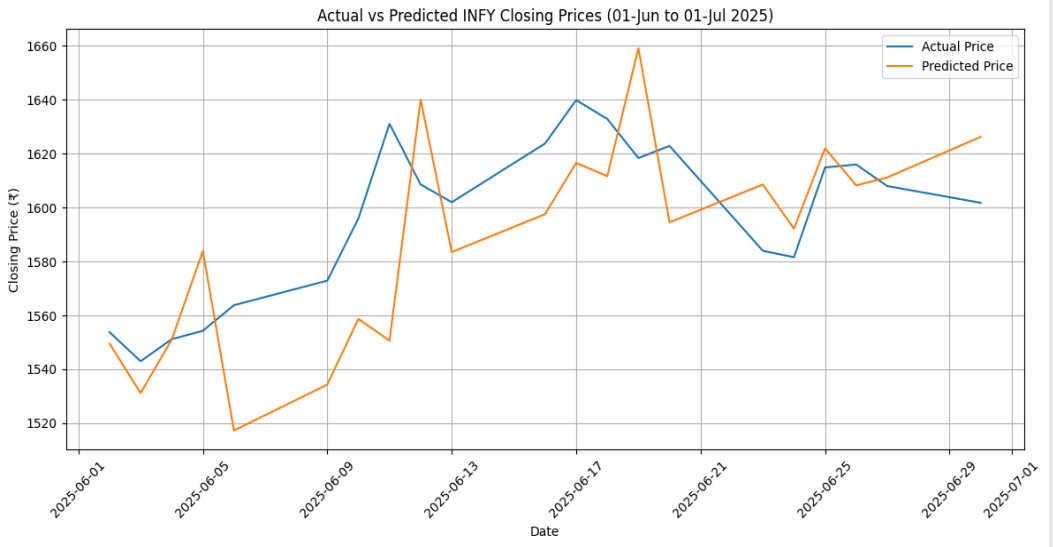


For 5000 Epochs :-

i) Performance metrics for 5000 epochs are as follows:

```
Test Set Performance Metrics (RNN):
Mean Absolute Error (MAE) : ₹24.58
Mean Squared Error (MSE)  : ₹926.53
Root Squared Error (RMSE) : ₹30.44
R² Score                  : -0.0867
Mean Absolute Percentage Error (MAPE) : 1.53%
Accuracy (100 - MAPE) : 98.47%
MSE Percentage of Average Price: 58.04%
MAE Percentage of Average Price: 1.54%
```

ii) The predicted closing price plot is as follows:

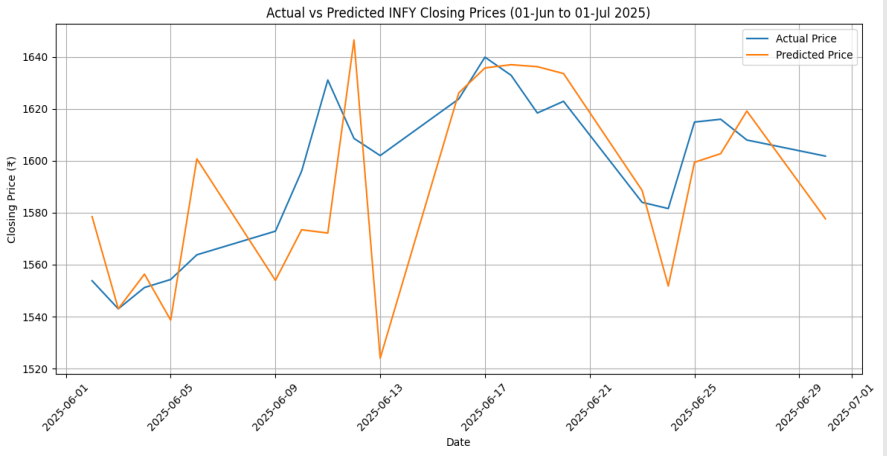


For 7500 Epochs :-

i) Performance metrics for 7000 epochs are as follows:

```
Test Set Performance Metrics (RNN):
Mean Absolute Error (MAE) : ₹20.78
Mean Squared Error (MSE)  : ₹790.90
Root Squared Error (RMSE) : ₹28.12
R² Score                  : 0.0724
Mean Absolute Percentage Error (MAPE) : 1.30%
Accuracy (100 - MAPE) : 98.70%
MSE Percentage of Average Price: 49.55%
MAE Percentage of Average Price: 1.30%
```

ii) The predicted closing price plot is as follows:

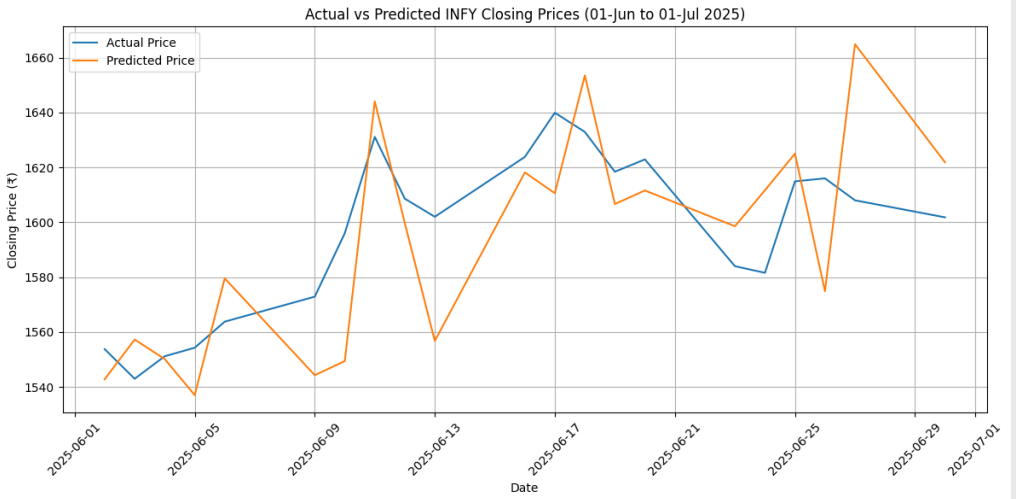


For 10000 Epochs :-

i) Performance metrics for 10000 epochs are as follows:

```
Test Set Performance Metrics (RNN):
Mean Absolute Error (MAE) : ₹21.57
Mean Squared Error (MSE)  : ₹679.76
Root Squared Error (RMSE) : ₹26.07
R² Score                  : 0.2027
Mean Absolute Percentage Error (MAPE) : 1.35%
Accuracy (100 - MAPE) : 98.65%
MSE Percentage of Average Price: 42.59%
MAE Percentage of Average Price: 1.35%
```

ii) The predicted closing price plot is as follows:



3. LSTM (Long Short Term Memory) :

The results obtained from the LSTM model are as followed :

Train Set Performance Metrics :

Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	70.5963 %	1.7712%	98.23%	1632.90	1607.71
500	35.2937 %	1.1642 %	98.80 %	1632.90	1637.99
1000	37.5304 %	1.2097 %	98.75 %	1632.90	1646.56
2000	26.8360 %	1.0287 %	98.94 %	1632.90	1634.50
5000	4.1505 %	0.4092 %	99.58 %	1632.90	1585.71
7500	0.8148 %	0.1851%	99.81%	1632.90	1659.23
10000	0.2228 %	0.0982 %	99.90%	1632.90	1627.44

Test Set Performance Metrics:

Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	57.7874 %	1.5929 %	98.41%	1632.90	1607.71
500	18.3871 %	0.8199 %	99.18%	1632.90	1637.99
1000	20.3010 %	0.9118%	99.09%	1632.90	1646.56
2000	35.0464 %	1.1924%	98.81%	1632.90	1634.50
5000	109.9753 %	2.2273 %	97.78 %	1632.90	1585.71
7500	106.6815%	2.1327 %	97.88 %	1632.90	1659.23
10000	36.6550 %	1.1546 %	98.85 %	1632.90	1627.44

For 100 Epochs :-

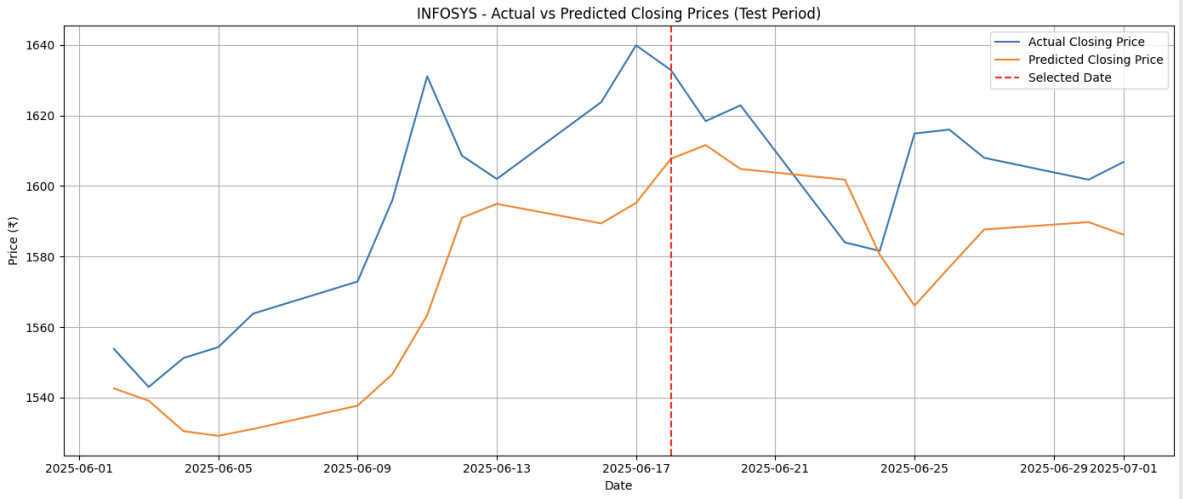
i) Performance metrics for 100 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 977.0417 (70.5963% of avg price)
MAE: 24.5133 (1.7712% of avg price)
Accuracy: 98.23%

========== Test Set Metrics ==========
MSE: 922.6989 (57.7874% of avg price)
MAE: 25.4344 (1.5929% of avg price)
Accuracy: 98.41%
=======================================
```

ii) Predicted vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1607.71
Actual closing price: ₹1632.90
Absolute error: 1.54%
```

iii) Actual Vs Predicted Graph Plot :



For 500 Epochs :-

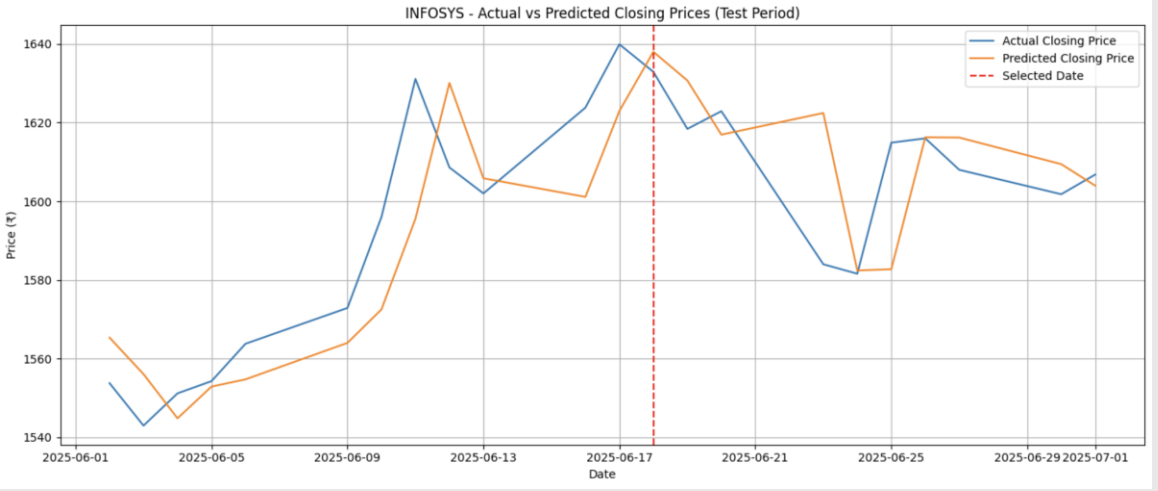
i) Performance metrics for 500 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 488.4592 (35.2937% of avg price)
MAE: 16.1128 (1.1642% of avg price)
Accuracy: 98.80%

========== Test Set Metrics ==========
MSE: 293.5897 (18.3871% of avg price)
MAE: 13.0912 (0.8199% of avg price)
Accuracy: 99.18%
=========================================
```

ii) Predicted vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1637.99
Actual closing price: ₹1632.90
Absolute error: 0.31%
```

iii) Actual Vs Predicted Graph Plot :



For 1000 Epochs :-

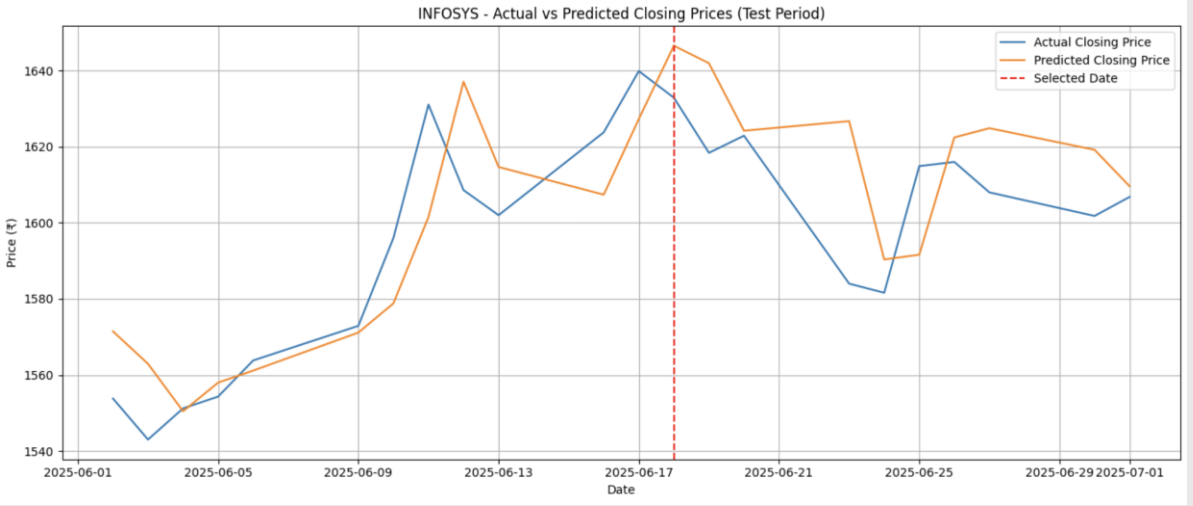
i) Performance metrics for 1000 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 519.4150 (37.5304% of avg price)
MAE: 16.7416 (1.2097% of avg price)
Accuracy: 98.75%

========== Test Set Metrics ==========
MSE: 324.1492 (20.3010% of avg price)
MAE: 14.5587 (0.9118% of avg price)
Accuracy: 99.09%
=========================================
```

ii) Predicted vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1646.56
Actual closing price: ₹1632.90
Absolute error: 0.84%
```

iii) Actual Vs Predicted Graph Plot :



For 2500 Epochs :-

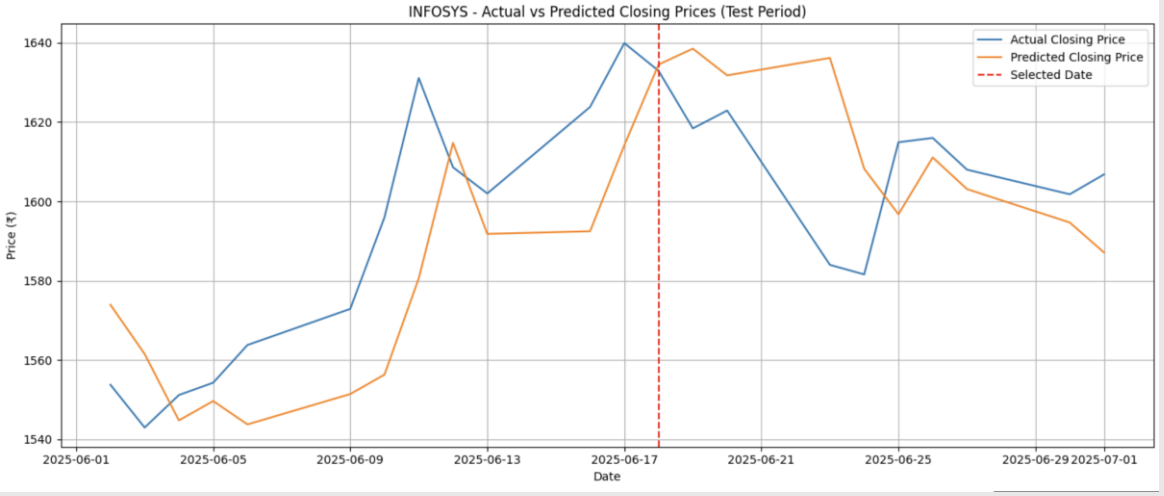
i) Performance metrics for 2500 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 371.4068 (26.8360% of avg price)
MAE: 14.2369 (1.0287% of avg price)
Accuracy: 98.94%

========== Test Set Metrics ==========
MSE: 559.5900 (35.0464% of avg price)
MAE: 19.0387 (1.1924% of avg price)
Accuracy: 98.81%
=======================================
```

ii) Predicted vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1634.50
Actual closing price: ₹1632.90
Absolute error: 0.10%
```

iii) Actual Vs Predicted Graph Plot :



For 5000 Epochs :-

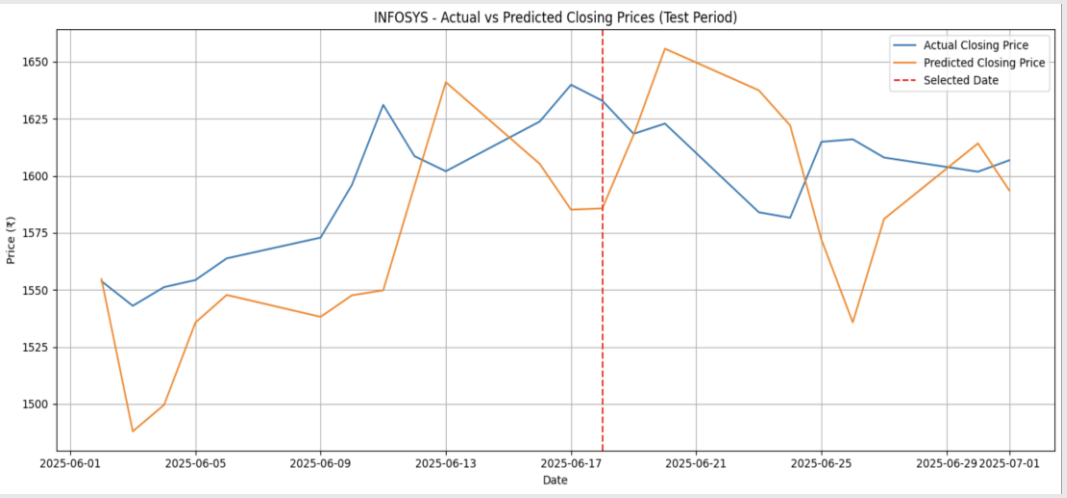
i) Performance metrics for 5000 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 57.4416 (4.1505% of avg price)
MAE: 5.6638 (0.4092% of avg price)
Accuracy: 99.58%

========== Test Set Metrics ==========
MSE: 1755.9907 (109.9753% of avg price)
MAE: 35.5478 (2.2263% of avg price)
Accuracy: 97.78%
=========================================
```

ii) Predicted vs Actual Closing Price:

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1585.71
Actual closing price: ₹1632.90
Absolute error: 2.89%
```

iii) Actual Vs Predicted Graph Plot:



For 7500 Epochs :-

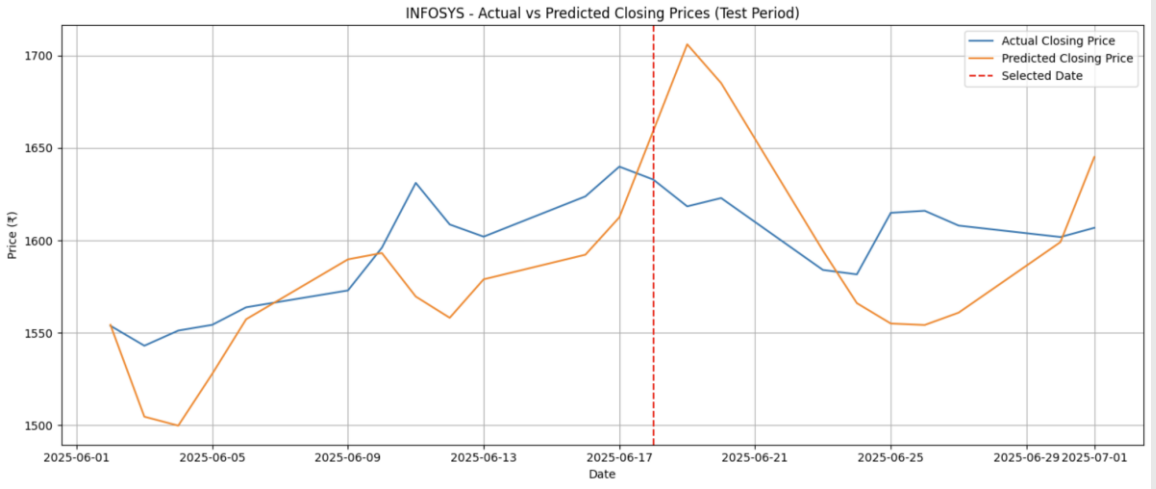
i) Performance metrics for 7500 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 11.2761 (0.8148% of avg price)
MAE: 2.5621 (0.1851% of avg price)
Accuracy: 99.81%

========== Test Set Metrics ==========
MSE: 1703.3982 (106.6815% of avg price)
MAE: 34.0526 (2.1327% of avg price)
Accuracy: 97.88%
=======================================
```

ii) Predicted vs Actual Closing Price :

```
Enter a date between 2025-06-01 and 2025-07-01 (YYYY-MM-DD): 2025-06-18

For 2025-06-18:
Predicted closing price: ₹1659.23
Actual closing price: ₹1632.90
Absolute error: 1.61%
```

iii) Actual Vs Predicted Graph Plot :



For 10000 Epochs :-

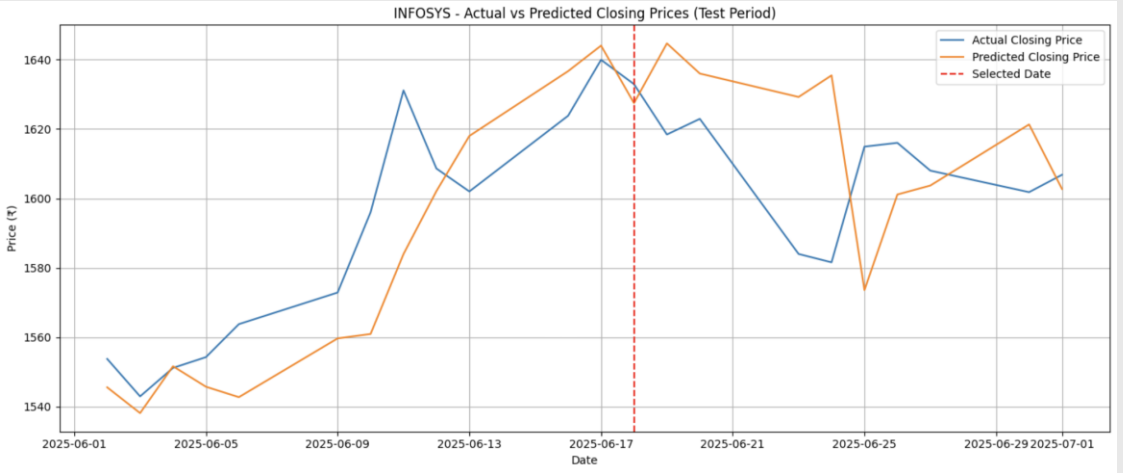
i) Performance metrics for 10000 epochs are as follows:

```
========== Training Set Metrics ==========
MSE: 3.1167 (0.2228% of avg price)
MAE: 1.3737 (0.0982% of avg price)
Accuracy: 99.90%

========== Test Set Metrics ==========
MSE: 585.2760 (36.6550% of avg price)
MAE: 18.4361 (1.1546% of avg price)
Accuracy: 98.85%
=======================================
```

ii) Predicted vs Actual Closing Price :

```
For 2025-06-18:
Predicted closing price: ₹1627.44
Actual closing price: ₹1632.90
Absolute error: 0.33%
```

iii) Actual Vs Predicted Graph Plot :



3. LSTM fine-tuned with Reinforcement Learning :

The results obtained from the LSTM + RL model are as followed :

Train Set Performance Metrics :

Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	51.23%	1.44%	98.55%	1632.90	1635.27
500	31.71%	1.12%	98.88%	1632.90	1633.05
1000	29.13%	1.09%	98.91%	1632.90	1632.54
2000	26.56%	1.04%	98.96%	1632.90	1633.17
5000	7.82%	0.57%	99.42%	1632.90	1605.72
7000	5.33%	0.47%	99.51%	1632.90	1596.71
10000	2.28%	0.31%	99.68%	1632.90	1538.36

Test Set Performance Metrics:

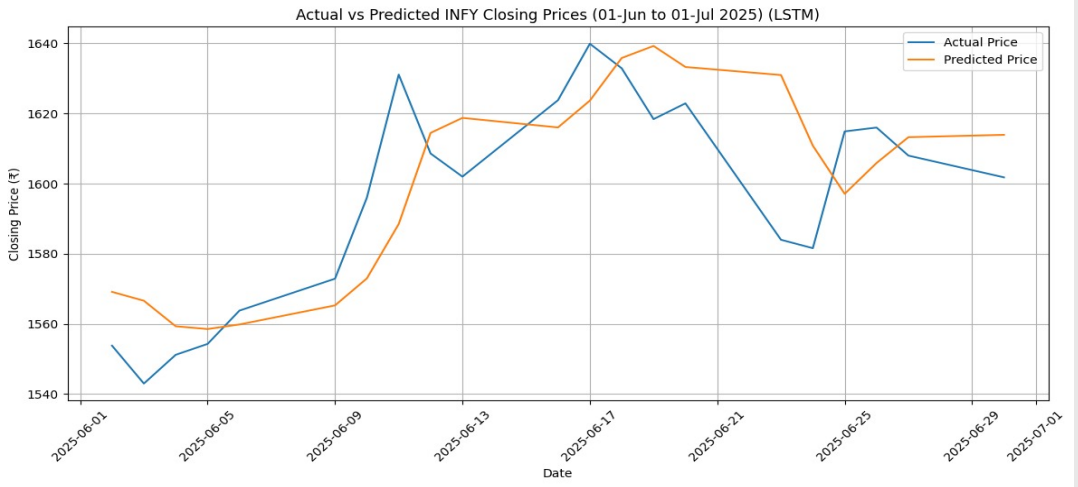
Epochs	MSE	MAE	Accuracy	Actual Price (Rs.)	Predicted Price (Rs.)
100	24.29%	0.99%	99.01%	1632.90	1635.27
500	21.46%	0.92%	99.08%	1632.90	1633.05
1000	22.25%	0.90%	99.11%	1632.90	1632.54
2000	22.74%	0.96%	99.04%	1632.90	1633.17
5000	70.29%	1.78%	98.22%	1632.90	1605.72
7000	81.63%	1.92%	98.09%	1632.90	1596.71
10000	177.83%	2.58%	97.45%	1632.90	1538.36

For 100 Epochs :-

i) Performance metrics for 100 epochs are as follows:

```
Test Set Performance Metrics (LSTM):
Mean Absolute Error (MAE) : ₹15.75
Mean Squared Error (MSE)  : ₹387.67
Root Squared Error (RMSE) : ₹19.69
R² Score                  : 0.5453
Mean Absolute Percentage Error (MAPE) : 0.99%
Accuracy (100 - MAPE) : 99.01%
MSE Percentage of Average Price: 24.29%
MAE Percentage of Average Price: 0.99%
```

ii) The predicted closing price plot is as follows:

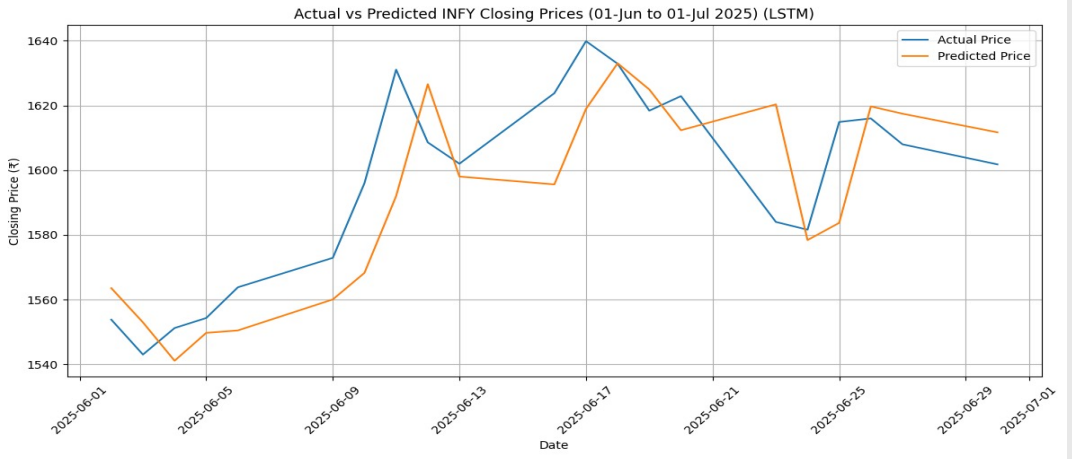


For 500 Epochs :-

i) Performance metrics for 500 epochs are as follows:

```
Test Set Performance Metrics (LSTM):
Mean Absolute Error (MAE) : ₹14.74
Mean Squared Error (MSE)  : ₹342.50
Root Squared Error (RMSE) : ₹18.51
R² Score                  : 0.5983
Mean Absolute Percentage Error (MAPE) : 0.92%
Accuracy (100 - MAPE) : 99.08%
MSE Percentage of Average Price: 21.46%
MAE Percentage of Average Price: 0.92%
```

ii) The predicted closing price plot is as follows:

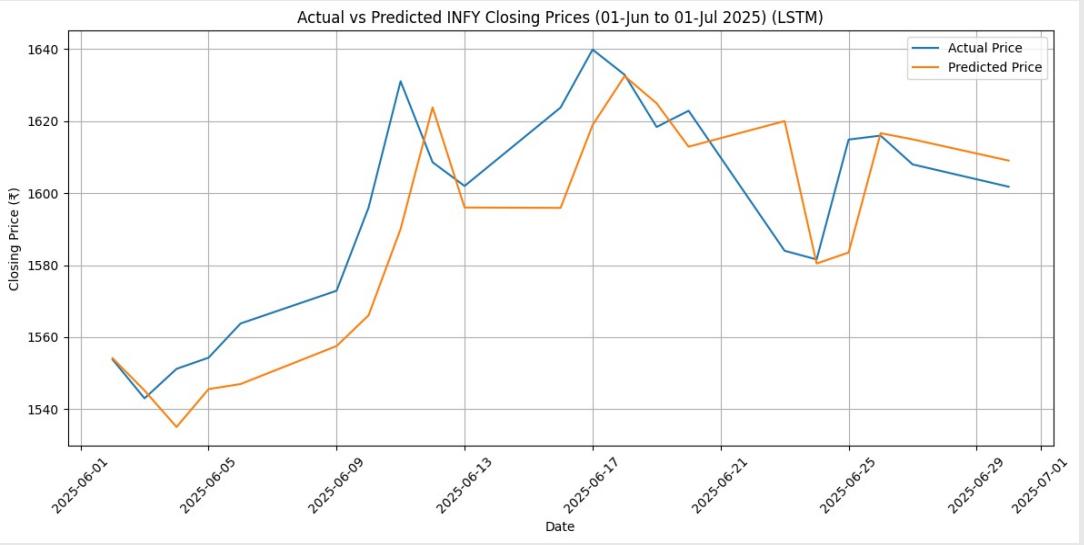


For 1000 Epochs :-

i) Performance metrics for 1000 epochs are as follows:

```
Test Set Performance Metrics (LSTM):
Mean Absolute Error (MAE) : ₹14.33
Mean Squared Error (MSE)  : ₹355.10
Root Squared Error (RMSE) : ₹18.84
R² Score                  : 0.5835
Mean Absolute Percentage Error (MAPE) : 0.89%
Accuracy (100 - MAPE) : 99.11%
MSE Percentage of Average Price: 22.25%
MAE Percentage of Average Price: 0.90%
```

ii) The predicted closing price plot is as follows:

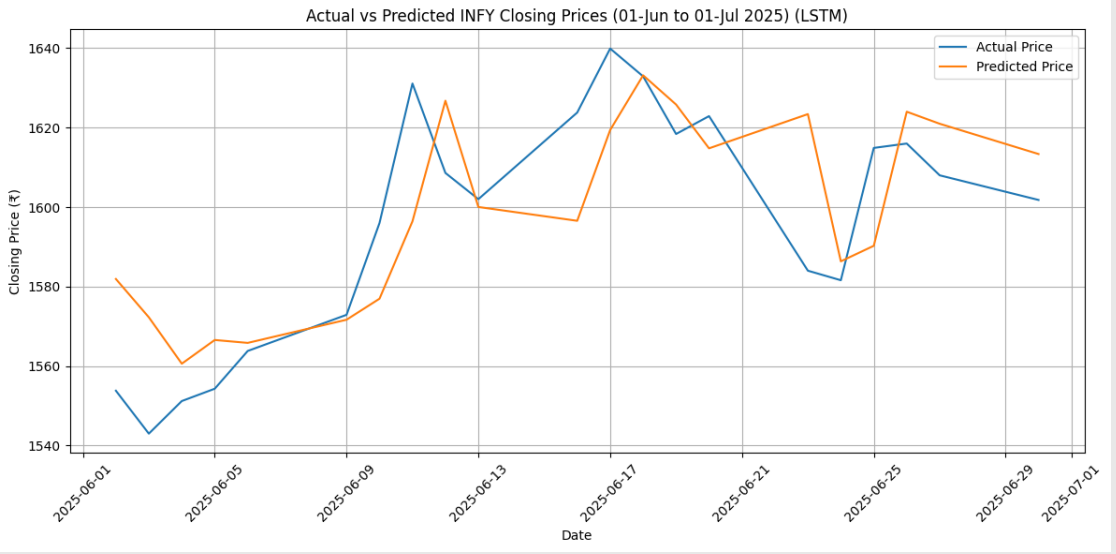


For 2500 Epochs :-

i) Performance metrics for 2000 epochs are as follows:

```
Test Set Performance Metrics (LSTM):
Mean Absolute Error (MAE) : ₹15.29
Mean Squared Error (MSE)  : ₹362.98
Root Squared Error (RMSE) : ₹19.05
R² Score                  : 0.5743
Mean Absolute Percentage Error (MAPE) : 0.96%
Accuracy (100 - MAPE) : 99.04%
MSE Percentage of Average Price: 22.74%
MAE Percentage of Average Price: 0.96%
```

ii) The predicted closing price plot is as follows:

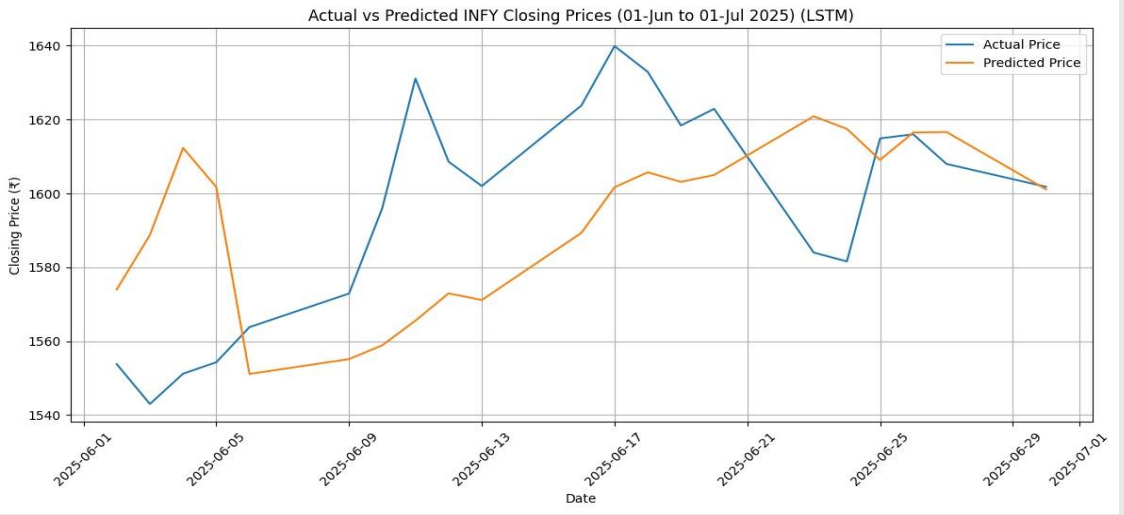


For 5000 Epochs :-

i) Performance metrics for 5000 epochs are as follows:

```
Test Set Performance Metrics (LSTM):
Mean Absolute Error (MAE) : ₹28.36
Mean Squared Error (MSE)  : ₹1122.06
Root Squared Error (RMSE) : ₹33.50
R² Score                  : -0.3160
Mean Absolute Percentage Error (MAPE) : 1.78%
Accuracy (100 - MAPE) : 98.22%
MSE Percentage of Average Price: 70.29%
MAE Percentage of Average Price: 1.78%
```

ii) The predicted closing price plot is as follows:

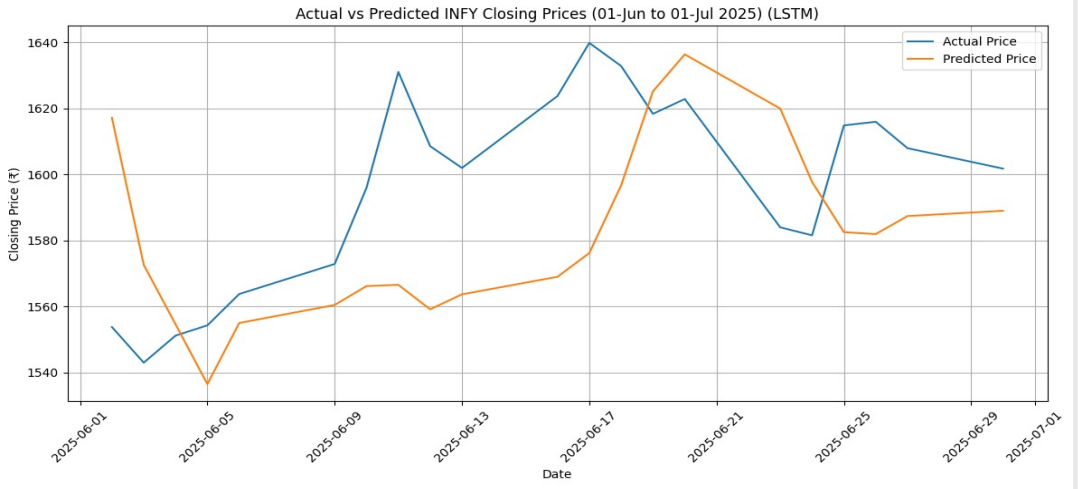


For 7000 Epochs :-

i) Performance metrics for 7000 epochs are as follows:

```
Test Set Performance Metrics (LSTM):
Mean Absolute Error (MAE) : ₹30.69
Mean Squared Error (MSE)  : ₹1303.05
Root Squared Error (RMSE) : ₹36.10
R² Score                  : -0.5283
Mean Absolute Percentage Error (MAPE) : 1.91%
Accuracy (100 - MAPE) : 98.09%
MSE Percentage of Average Price: 81.63%
MAE Percentage of Average Price: 1.92%
```

ii) The predicted closing price plot is as follows:

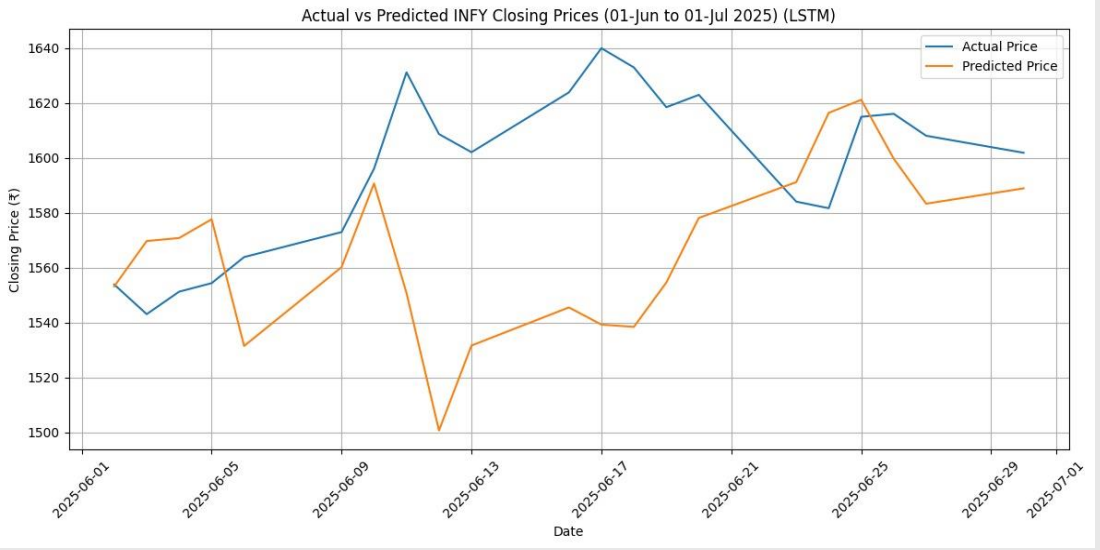


For 10000 Epochs :-

i) Performance metrics for 10000 epochs are as follows:

```
Test Set Performance Metrics (LSTM):
Mean Absolute Error (MAE) : ₹41.14
Mean Squared Error (MSE)  : ₹2838.51
Root Squared Error (RMSE) : ₹53.28
R² Score                  : -2.3291
Mean Absolute Percentage Error (MAPE) : 2.55%
Accuracy (100 - MAPE) : 97.45%
MSE Percentage of Average Price: 177.83%
MAE Percentage of Average Price: 2.58%
```

ii) The predicted closing price plot is as follows:



OBSERVATION

After analyzing and evaluating the performance of all four models—Simple RNN, RNN fine-tuned with Reinforcement Learning, LSTM, and LSTM fine-tuned with Reinforcement Learning, the following key insights and conclusions can be drawn:

1. Model Wise Performance Overview

Simple RNN:

- Peak Accuracy: Achieved a training accuracy of 98.94% at 5000 epochs, with Test accuracy of 99.04% at 100 epochs.
- Stability: While accuracy remained consistently high, MSE and MAE values fluctuated across epochs, indicating occasional overfitting or underfitting.
- General Observation: Performs decently with longer training, but the learning is not as deep or stable when compared to LSTM.

RNN + Reinforcement Learning:

- Best Train Accuracy: 99.82% at 10000 epochs with low MSE (0.67%) and MAE (0.18%).
- Best Test Accuracy: 99.18% at 500 epochs, but degrades sharply with more episodes (overfitting).
- Observation: The RL fine-tuning helps the model learn more accurate weights, but tends to overfit the training data beyond 2000 episodes.

LSTM:

- Best Train Accuracy: 99.90% at 10000 epochs, with lowest MSE (0.2228%) and MAE (0.0982%).
- Test Accuracy: Peaks at 99.18% at 500 epochs, then declines with higher epochs.
- Observation: LSTM handles sequence dependencies better than SimpleRNN, but suffers from overfitting beyond a certain point.

LSTM + Reinforcement Learning:

- Best Train Accuracy: 99.68% at 10000 epochs, but Test accuracy declines drastically after 2000 epochs.
- Overfitting Issues: The gap between train and test performance becomes significant after 5000 epochs, suggesting that RL may be over-optimizing the training set.
- Observation: Although RL improves learning, generalization ability deteriorates with very high training durations.

2. Accuracy Vs Epochs Analysis

- SimpleRNN showed gradual improvements up to 5000 epochs, but beyond that, accuracy didn't improve much.
- LSTM and RNN with RL both showed very fast convergence in the early epochs (2000–5000), after which overfitting kicked in.
- LSTM with RL had impressive train metrics, but poor test performance in later epochs due to over-tuning.

3. Preferred Model

Based on all metrics (MSE, MAE, Accuracy) and especially generalization on test data, the LSTM without RL model at 500 epochs is the most balanced and preferred model.

Because :
It Achieved high test accuracy (99.18%), with low MSE (18.38%) and MAE (0.8199%).
It Avoided overfitting better than the RL versions.
It Maintained good generalization with minimal training time compared to RL-based approaches.

4. Role of Reinforcement Learning (RL)

- RL improves training accuracy significantly.
- However, it comes at the cost of overfitting and reduced test performance after long episodes.
- Works better when the number of episodes is controlled (≤2000).
- RL is best suited for fine-tuning, not for complete model training.

5. Ideal Epochs/Episodes

Model	Ideal Epochs	Reason
Simple RNN	500	Good balance of MSE and accuracy
RNN + RL	2000	Optimal before overfitting
LSTM	500	Best generalization
LSTM + RL	2000	After 2000 epochs, test accuracy drops

CONCLUSION

After thoroughly analyzing all four models, it is evident that Long Short-Term Memory (LSTM) models outperform traditional SimpleRNN in terms of capturing long-term dependencies in stock price sequences. While Reinforcement Learning (RL) fine-tuning significantly improves training accuracy across both RNN

and LSTM models, it also introduces a risk of overfitting, especially when the number of episodes exceeds a certain threshold (typically beyond 2000).

Among all models, the LSTM model without RL at 500 epochs emerged as the most balanced solution. It demonstrated a high test accuracy (99.18%) with relatively low MSE and MAE, indicating strong generalization capability on unseen data. This model was able to effectively learn temporal patterns without overfitting, and it required less training time and tuning compared to RL-based variants.

On the other hand, RL-enhanced models, particularly LSTM + RL, showed exceptional training performance—reaching as high as 99.90% accuracy—but suffered a noticeable drop in test accuracy at higher episode counts, reflecting limited generalizability in real-world, unseen scenarios. This suggests that while RL can be an effective optimization tool, its benefits must be balanced with proper validation and early stopping strategies.

In summary, LSTM at 500 epochs is the preferred model for this stock prediction task due to its robust performance, minimal overfitting, and efficient training time. RL-based enhancements may be further explored in future work with better regularization techniques or adaptive episode control to avoid performance degradation.

References

[1] P. Zhou, Z. Chen, Q. Li, and Y. Huang, “Stock market predictions with a sequence-oriented BiLSTM stacked model,” *Human–Computer Interaction and Blockchain*, vol. 1, no. 1, pp. 50–58, Mar. 2024. doi: 10.1016/j.iswa.2024.200439

[2] C. C. Aggarwal, *Neural Networks and Deep Learning: A Textbook*, 2nd ed. Cham, Switzerland: Springer, 2023.

[3] A. L. Awad, S. M. Elkaffas, Md. W. Fakhr, “Stock Market Prediction Using Deep Reinforcement Learning”, *Appl. Syst. Innov.* 2023, *6*, 106. Available: https://www.mdpi.com/2571-5577/6/6/106. doi: 10.3390/asi6060106

[4] K. Pardeshi, S. S. Gill, and A. M. Abdelmoniem, “Stock Market Price Prediction: A Hybrid LSTM and Sequential Self-Attention based Approach,” *arXiv*, Aug. 7, 2023. [Online]. Available: https://arxiv.org/abs/2308.04419. doi: 10.48550/arXiv.2308.04419

[5] Zhang, Zihao, Stefan Zohren, and Stephen Roberts. "Deep reinforcement learning for trading.", Nov. 22, 2019. Available: https://arxiv.org/abs/1911.10107. doi: 10.48550/arXiv:1911.10107

[6] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 3rd ed. Sebastopol, CA, USA: O’Reilly Media, 2022.

[7] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ, USA: Pearson, 2020.

[8] W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 3rd ed. Sebastopol, CA, USA: O’Reilly Media, 2018.