FinSearch Final Report

Nikhil Biradar
Kushal C. Gajbe
Sanket Kothawade
Sabyasachi Samantaray

July 2023

Indian Institute of Technology, Bombay

Contents

1 RoadMap 1 **2**
1.1 Stock Trading 2
1.2 Traditional Methods 2
1.2.1 ARIMA MODEL:- 2
1.3 RL based Methods 5
1.3.1 Traditional RL Methods 6
1.3.2 Deep RL Methods 8
1.4 Comparison 8

2 RoadMap 2 **10**

3 Roadmap 3 and 4 **11**
3.1 Comparison of LSTM and RL 11
3.2 Our Approach 12
3.3 Some Simplistic Assumptions 12
3.4 Agent Code Summary 14
3.4.1 Custom Loss Function: `huber_loss(y_true, y_pred, clip_delta=1.0)` . . . 14
3.4.2 Agent Class: `Agent` 14
3.4.3 Training Functions 14
3.5 Method Code Summary 15
3.5.1 Function 1: `train_model(agent, episode, data, ep_count=100, batch_size=32, window_size=10)` 15
3.5.2 Function 2: `evaluate_model(agent, data, window_size, debug)` 15
3.6 Results 16

Chapter 1

RoadMap 1

1.1 Stock Trading

The practice of trying to determine the future value of a business stock or other financial instrument that is traded on an exchange is known as a stock market prediction. A stock trader who can accurately forecast the price of the stock in the future may realize big profits.
The purpose of stock market investment is to obtain more profits. In recent years, an increasing number of researchers have tried to implement stock trading based on machine learning. Facing the complex stock market, how to obtain effective information from multisource data and implement dynamic trading strategies is difficult. To solve these problems, this study proposes a new reinforcement learning model to implement stock trading, analyzes the stock market through stock data, technical indicators and candlestick charts, and learns dynamic trading strategies. Fusing the features of different data sources, the agent in reinforcement learning makes trading decisions on this basis.

1.2 Traditional Methods

There are various Traditional algorithms/techniques for stock/option trading but one of the most famous ones is ARIMA. We will be talking about the ARIMA model in the traditional part of our research.

1.2.1 ARIMA MODEL:-

The ARIMA (AutoRegressive Integrated Moving Average) model is a widely used time series forecasting technique that is effective in capturing complex patterns and trends within sequential data.
Time series data often exhibit patterns like trends, seasonality, and autocorrelation, which makes them challenging to model using traditional regression techniques. ARIMA models are specifically designed to handle such complexities and provide reliable forecasts for a wide range of applications, including economics, finance, weather forecasting, and sales forecasting. It is especially effective when the data exhibits temporal patterns, trends, and seasonality.
The ARIMA (AutoRegressive Integrated Moving Average) model requires the time series data to be stationary for accurate forecasting. Stationarity implies that the statistical properties of the data, such as the mean, variance, and autocorrelation, do not change over time. To achieve stationarity in the data, various techniques can be utilized, including rolling statistics and the Dickey-Fuller test.
Rolling Statistics:-
In the rolling statistics approach, a moving window is applied to the original time series data, and the mean (or other statistical measures) is computed within that window. By plotting the moving average alongside the original data, we can visually inspect if the mean remains constant over time.

If the mean stays consistent, it indicates that the data may be stationary.

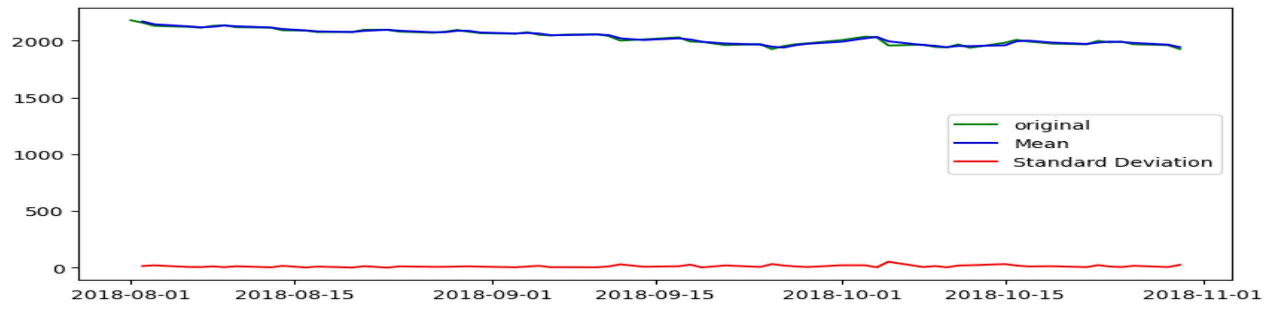


Figure 1.1: Rolling Statistics

Also, taking the log of the data can also make it stable and/or stationary. However, one must note that he/she should take exponential of the predictions to get original results.

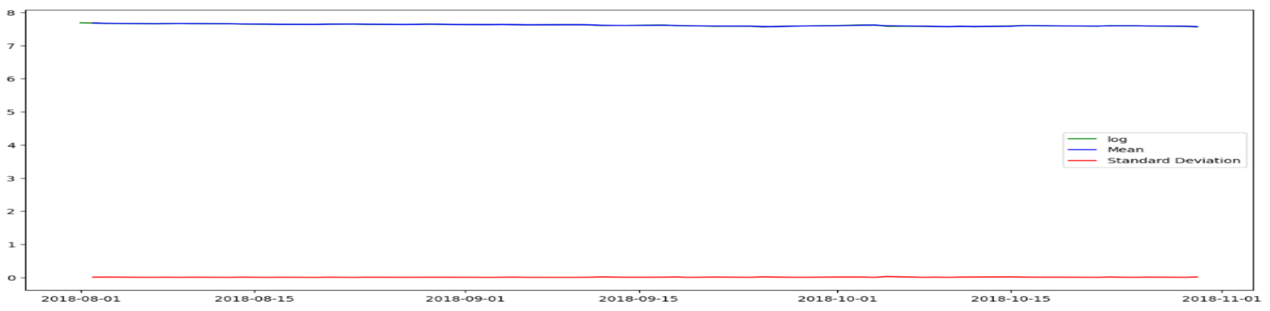


Figure 1.2: After taking Log(DATA)

Differencing:- In differencing, the time series data is transformed by taking the difference between consecutive observations. This process removes the underlying trend and can make the data stationary. By examining the differenced series, we can observe if the data no longer exhibit any clear trends or seasonality.

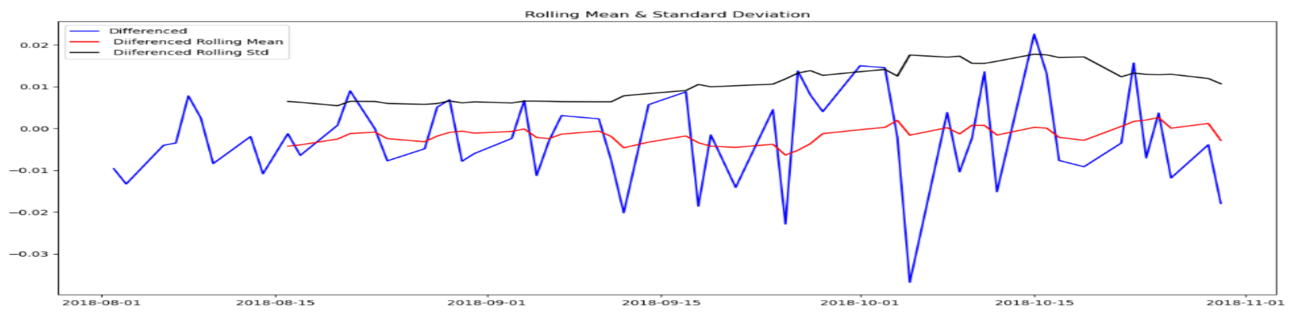


Figure 1.3: Differencing

The ARIMA (AutoRegressive Integrated Moving Average) model is a combination of three components: AutoRegressive (AR), Integrated (I), and Moving Average (MA). The notation for the ARIMA

model is ARIMA(p, d, q), where:
AutoRegressive (AR) component: The AutoRegressive component models the relationship between the current value of the time series and its past values. The AR component is expressed mathematically as follows:

$$\hat{y}_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \epsilon_t$$

Figure 1.4: AR

Intrgrated (I) component: The Integrated component handles the differencing of the time series to make it stationary. Differencing involves taking the difference between consecutive observations. The differencing step is performed on the original time series, and it is expressed mathematically as follows:

$$y'_t = (1 - B)^d y_t$$

Figure 1.5: I

Moving Average (MA) component: The Moving Average component models the relationship between the current value of the time series and the past forecast errors (residuals). It captures the short-term patterns and noise in the data.

$$\hat{y}_t = c + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q}$$

Figure 1.6: MA

Overall Equation for ARIMA:-

$$\hat{y}'_t = c + \phi_1 y'_{t-1} + \phi_2 y'_{t-2} + \ldots + \phi_p y'_{t-p} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} + \epsilon_t$$

Figure 1.7: ARIMA

where:

- Y(t) represents the value of the time series at time ”t.”
- c is a constant term (intercept) in the model.
- ϕi denotes the coefficients of the AR terms for lags ”i = 1, 2, ..., p.”
- θi denotes the coefficients of the MA terms for lags ”i = 1, 2, ..., q.”
- ϵ(t) is the white noise error term at time ”t.”

The ”p” in ARIMA(p, d, q) represents the order of the AutoRegressive component, denoting the number of lagged terms included in the model. The ”d” in ARIMA(p, d, q) represents the order of differencing required to achieve stationarity. The "q" in ARIMA(p, d, q) represents the order of the Moving Average component, indicating the number of past forecast errors included in the model.

Challenges with traditional RL-based stock trading algorithms:

- **Sample Efficiency**: Learning efficient trading methods involves a large quantity of data, which may be expensive and time-consuming. Stock market data can be quite noisy.
- **Highly Dynamic**: Stock markets are dynamic and non-stationary, making it difficult for models trained on historical data to generalize to new market conditions.
- **Overfitting**: Lacking appropriate regularisation methods might cause RL algorithms to overfit to historical data, which can result in subpar performance in actual trading scenarios
- **Transaction Costs**: RL algorithms may struggle to account for transaction costs, which are significant in real trading scenarios.

While RL-based stock trading algorithms have been explored, it's worth noting that traditional RL methods have certain limitations when applied directly to financial markets. Many successful trading strategies today utilize a combination of RL techniques with other methodologies like supervised learning, time series analysis, and expert knowledge to create more robust and effective trading systems. Additionally, advancements in deep learning and reinforcement learning, such as Proximal Policy Optimization (PPO) and Actor-Critic algorithms, have shown promise in addressing some of the challenges faced by traditional RL approaches in stock trading

1.3 RL based Methods

Recent works focus more on Machine Learning based methods for stock trading owing to the rapid growth of the ML development community and support. Traditional methods cannot cope up with the non-linear fluctuations of the stock market.

The supervised and unsupervised machine learning methods aim to give a good judgment at the immediate next step, but not at a cumulative measure. Most supervised learning algorithms provide action recommendations on particular stocks whereas using RL can lead us directly to the decision-making step, i.e., deciding how to buy, hold or sell any stock.
Any RL problem can be viewed as a **Markov Decision Process** i.e., a process of sequential decision making. MDPs form the bedrock of RL formalization. We have an agent(the decision maker) who interacts with the environment and performs an action and gets a reward as a consequence. **Agent's aim is to maximize the cumulative reward**. Mathematically, at some time t, let the environment be at state $S_t \in \mathbb{S}$, and the agent after analyzing the environment decides to choose action $A_t \in \mathbb{A}$. The consequence of this state-action pair (S_t, A_t) is reward $R(S_t, A_t) = R_{t+1}$. Figure 1.8 summarises this MDP.
Assuming reward space $\mathbb{R}$ and state space $\mathbb{S}$ are finite sets, R_t, S_t have well defined **probability distributions**, depending on the previous state action pair (S_{t-1}, A_{t-1}).

$$P(s', r|(s, a)) = Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$$

Expected return formalizes the idea of discounted cumulative reward. We define expected return G at time t as

$$G = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$$

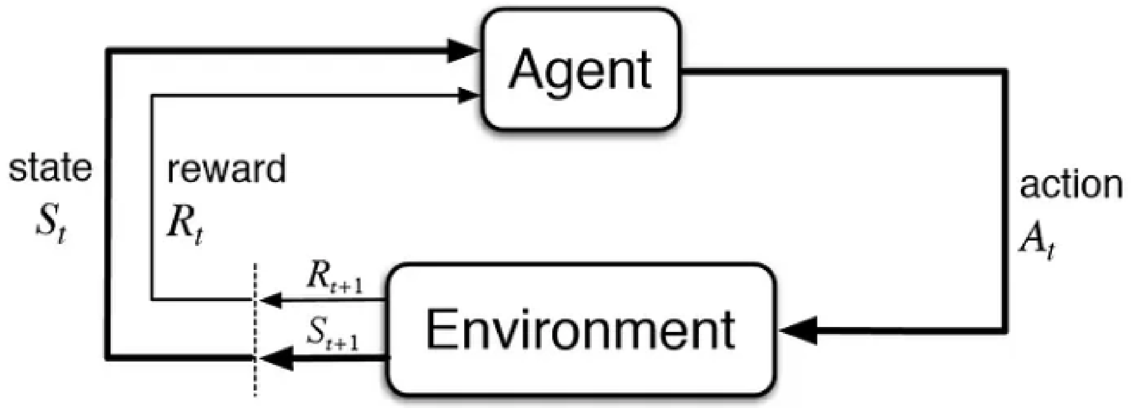


Figure 1.8: Action-Reward feedback loop of a generic RL model

γ is the **decay/discount rate**, the rate for which we discount the future reward. "**One dollar today is better than one dollar tomorrow**". However, in high-frequency trading and a short period of time, we can set γ close to 1.
An agent tries to learn an **optimal polic**y(a probability function to select actions based on state) $\pi(s|a)$. **Value Functions** measure how good the specific action/state is. These functions drive the learning process for optimizing policy. **State value functions** measure how good is a given state under policy π. **Action value(Q-value)** functions measure how good an action at a given state is under policy π.

$$v_\pi(s) = E[G_t|S_t = s]$$

$$q_\pi(s, a) = E[G_t|s_t = s, A_t = a]$$

The Q Value function satisfies the **Bellman optimality equation** given as

$$q_\pi(s, a) = E[R_{t+1} + \gamma max_{a'} q_\pi(s', a')]$$

1.3.1 Traditional RL Methods

This refers to a class of algorithms that were widely used before the advent of Deep Learning based RL methods. These traditional RL methods typically involve explicit representation of the environment, state space, action space, and transition dynamics.
Here are some examples of traditional RL algorithms:

Temporal Difference Learning

Temporal Difference (TD) Learning is a class of reinforcement learning (RL) algorithms that enable agents to learn from incomplete sequences of experience without requiring a model of the environment. The fundamental concept behind TD learning is the use of **bootstrapping**, where the agent updates its value function estimates based on its own predictions. Instead of waiting until the end of an episode to compute the true return, as Monte Carlo methods do, or performing a full backup based on a model, as dynamic programming methods do, TD learning updates the value function estimate after experiencing each time step. Temporal Difference learning has proven to be a powerful and widely used approach in RL due to its ability to learn efficiently from incomplete experiences, making it suitable for **online and real-world applications**. Common TD learning algorithms include **SARSA** (an on-policy method) and **Q-Learning** (an off-policy method), both of which use **TD updates** to learn action-value functions.

Q Learning

Q-Learning is a **model-free, off-policy** RL algorithm that aims to learn the **optimal action-value function (Q-function)**. It iteratively updates the Q-values based on the Bellman equation until convergence. It has an exploration vs exploitation tradeoff, which is decided using an ϵ-greedy strategy. Initially the exploration rate $\epsilon = 1$ reduces exponentially with exploration. Learning rate α is used during the update of the q-value. Let during a time-step, the model learns the value $q_{learned}$ given by

$$q_{learned}(s, a) = R_{t+1} + \gamma max_{a'} q(s', a')$$

then the updated q-value is as follows

$$q_{new}(s, a) = (1 - \alpha)q_{old}(s, a) + \alpha q_{learned}(s, a)$$

The performance of Q-Learning is poor when in complex and/or sophisticated environments.

SARSA

SARSA(State-Action-Reward-State-Action) algorithm follows an "on-policy" Temporal Difference (TD) learning algorithm because it uses the same policy to select actions $(a\_t + 1)$ during learning and execution. This means that the agent explores the environment while following its current policy, which can lead to better convergence if the policy is continuously updated during learning. The SARSA Update rule is as follows

$$q_{new}(s, a) = (1 - \alpha)q_{old}(s, a) + \alpha[R_{t+1} + \gamma q(s', a')]$$

Monte Carlo Methods

Monte Carlo methods are a class of algorithms used to estimate value functions or policies by sampling complete episodes of interactions with the environment. These methods do not require a model of the environment and directly learn from experience, making them model-free RL techniques. Here are two fundamental Monte Carlo methods used in traditional RL:

Monte Carlo Policy Evaluation: This method is used to estimate the state-value function (V(s)) for a given policy. It involves running multiple episodes of the agent interacting with the environment and recording the returns received for each state visited in each episode. Afterward, the state-value function is updated by averaging the observed returns for each state. The estimate becomes more accurate as more episodes are sampled.
Monte Carlo Control: This method is used to estimate the action-value function (Q(s, a)) for an optimal policy. Similar to Monte Carlo policy evaluation, Monte Carlo control involves running multiple episodes, but this time the agent takes actions following a particular exploration policy (e.g., ϵ-greedy). The Q-values are updated based on the observed returns and actions taken in each episode, aiming to find the optimal action-value function (Q*) for the optimal policy.

Monte Carlo methods do not require knowledge of the environment's dynamics or transition probabilities. They can learn directly from real experiences and handle stochastic environments effectively. Monte Carlo methods are typically simple to implement and understand.

They can be computationally inefficient because they need to sample complete episodes before any updates. In tasks with long episodes, the variance in return estimates can be high, leading to slow convergence. They might not be well-suited for online, continuous, or partially observable tasks. Despite their limitations, Monte Carlo methods were some of the earliest RL techniques and played a crucial role in laying the foundation for more advanced approaches, such as Temporal Difference learning and Deep Reinforcement Learning.

1.3.2 Deep RL Methods

DRL has been a significant breakthrough in the field of artificial intelligence and has enabled the training of agents to achieve superhuman performance in various tasks. Some of the notable DRL methods are described below.

Deep Q-Networks(DQN)

Policy Gradient Methods

Q-Value Actor Critic(QAC)

Advantage Actor Critic(A2C)

Deep Deterministic Policy Gradient(DDPG)

One problem which is widely discussed in the case of Deep Q Learning Networks and Vanilla Policy Gradient Methods is the high volatility which may be a result of insufficient data to learn accurate value estimates and make optimal decisions or unstable learning and divergence due to random exploration at the starting phases.

1.4 Comparison

Traditional algorithms offer determinism, simplicity, and broad applicability but require manual feature engineering and lack adaptability. On the other hand, RL algorithms provide adaptability, autonomous feature extraction, and scalability, but can be sample inefficient, require careful exploration-exploitation balance, and may need more interpretability. Understanding these trade-offs helps select the appropriate algorithmic paradigm for different problem domains.

Learning Approach:
Traditional Algorithms: Traditional algorithms follow a fixed set of rules or instructions to solve a problem. They are designed by experts and require manual intervention for updating or modifying the rules when necessary.
RL-Based Algorithms: Reinforcement Learning (RL) algorithms learn by interacting with an environment and receiving feedback in the form of rewards or penalties. RL agents aim to maximize cumulative rewards by discovering optimal strategies over time through trial and error.
Adaptability:
Traditional Algorithms: Traditional algorithms are static and do not adapt to changes in the environment or task requirements without manual adjustments.
RL-Based Algorithms: RL algorithms can adapt to dynamic environments as they continuously learn and improve their strategies based on feedback and exploration.

Model Representation:
Traditional Algorithms: Most traditional algorithms use explicit mathematical models to represent the problem and its solutions. RL-Based Algorithms: RL

Domain Expertise:

Traditional Algorithms: Traditional algorithms often require domain-specific knowledge and expertise to design effective rule-based solutions.
RL-Based Algorithms: RL algorithms can learn effective strategies without extensive domain knowledge, making them applicable in a wider range of domains.
Data Dependency:
Traditional Algorithms: Traditional algorithms may not require vast amounts of data to function effectively, especially in cases where rules can be explicitly defined.
RL-Based Algorithms: RL algorithms typically require substantial amounts of data and experience to learn meaningful policies, which can be a limitation in scenarios with sparse or expensive data.
Training Time:
Traditional Algorithms: Traditional algorithms are usually faster to train since they do not involve learning from scratch but rely on predetermined rules.
RL-Based Algorithms: RL algorithms often require longer training times, especially in complex environments, as they learn from experiences and optimize policies.

Chapter 2

RoadMap 2

1. NIFTY50 stock data from Jan 2010 to Jun 2019 collected from Yahoo Finance python module We use the closing prices for the Stock Trading problem.
2. 1000 states have been defined based on portfolio value and return to capture market conditions.
3. At each state, the agent can choose from two actions - Buy or Sell.
4. The reward is the profit/loss incurred after every action.
5. We used the Vanilla Q-Learning algorithm to train on the historical data
6. Hyperparameters were tuned, learning_rate = 0.1, discount_factor = 0.9, num_episodes = 500, epsilon = 0.3 .
7. Results: Investment returns after 1000 episodes: $35000

Chapter 3

Roadmap 3 and 4

3.1 Comparison of LSTM and RL

Long Short-Term Memory (LSTM) and Reinforcement Learning (RL) models are integral components in the realm of stock market prediction and trading. LSTM, a type of recurrent neural network, excels at capturing complex temporal patterns in historical stock data. It is employed for time series forecasting, providing predictions that aid in buy and sell decisions. On the other hand, RL models leverage a dynamic decision-making framework, optimizing actions to maximize returns. These models consider not only historical data but also the impact of trading decisions on a portfolio's performance, making them suitable for adaptive trading strategies. Both LSTM and RL models contribute to enhancing stock market prediction accuracy and crafting robust trading strategies in the ever-evolving financial landscape
We have embarked on a venture to create a stock value prediction model specifically tailored to Reliance's data spanning from January 1, 2010, to December 31, 2018. In our quest for accuracy, we've chosen to employ a Long Short-Term Memory Recurrent Neural Network (LSTM-RNN) as the cornerstone of our predictive framework. Our focus, for the sake of simplicity, is solely on the closing prices of Reliance stock.

To facilitate the training and testing of our model, we've judiciously partitioned the dataset into a 90-10 split, allocating 90% of the data for training and reserving the remaining 10% for testing and evaluation. This segregation ensures that we have ample data for training while also maintaining a separate set for assessing the model's predictive capabilities.

The first step in our data preprocessing involves employing a Min-Max scaler. This essential step normalizes the data by transforming it into a uniform scale, making it conducive for our LSTM model to learn and generalize effectively. The Min-Max scaler is applied to the training data, allowing us to scale the values appropriately. For the training phase, we implement a sequence-based approach. Our input sequences are defined with a window size parameter, denoted as 'w.' This parameter is crucial as it determines how many past days' closing prices our model will consider when making predictions for the next day. Notably, we constrain 'w' to a range between 0 and 30, allowing us to experiment with different historical data perspectives.

The essence of our LSTM-based model lies in its ability to learn temporal dependencies and patterns in the data. As each training sequence moves through the network, the model's internal state is updated and refined, enabling it to predict the next day's stock value with increasing accuracy.For training we give input sequence of window size = w and train the LSTM model to predict what is the next

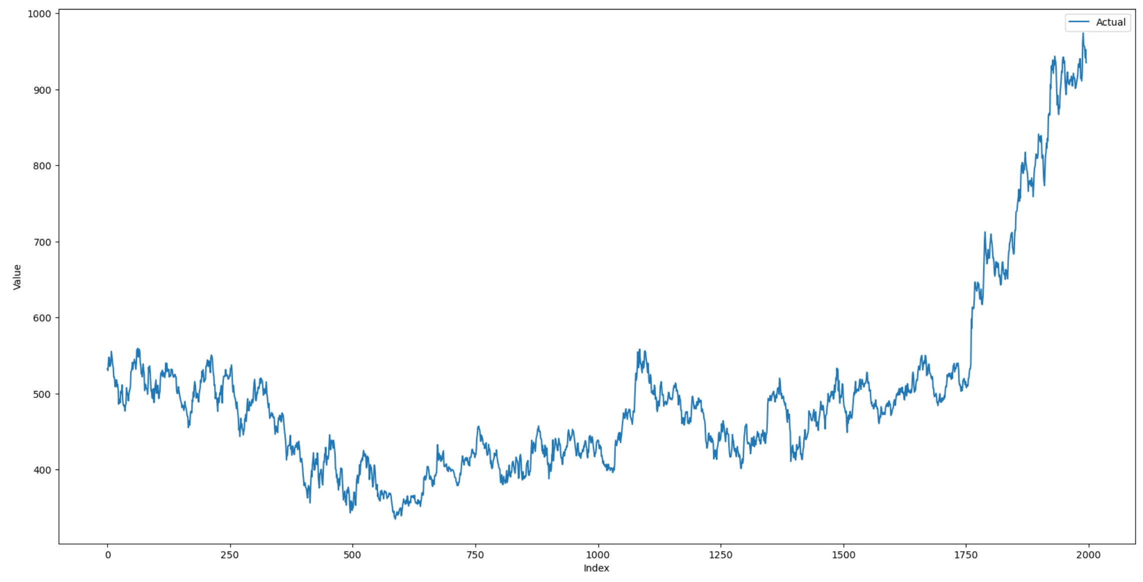


Figure 3.1: Training data from 2010-2018 Reliance, NIFTY50

day stock value. Our LSTM based model looks as follows, refer image 3.2 We train our model for 30 epochs, we fix window size to be w = 12. And then take stock trading decision based on a simple rule, if the predicted value for next day is greater than moving average, then buy a stock and otherwise, sell. We include short-selling to allow the user to sell stocks even when holdings are zero, so that the user can make future profits by later buying them at lower costs. The Sharpe ratio obtained is 9.1127. With Initial Investment of 91000.64, this model was able to build to a Final Portfolio value of 102413.92, which accounts to ROI of 10.62%.

3.2 Our Approach

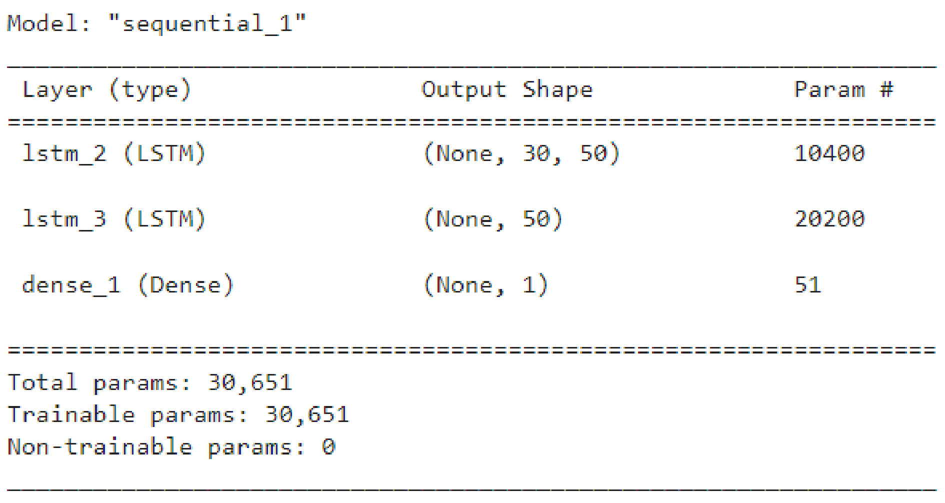
Deep Q-Learning, a neural form of Q-Learning, is a Model-free Reinforcement Learning approach used in this research. An agent maintains its current state (n-day window stock price representation) at any given time (episode), chooses and executes an action (buy/sell/hold), observes a subsequent state, receives some reward signal (difference in portfolio position), and finally modifies its parameters based on the gradient of the loss computed.
There have been several improvements to the Q-learning algorithm over the years, and a few have been implemented in this project[1]:

- Vanilla DQN
- DQN with fixed target distribution
- Double DQN

3.3 Some Simplistic Assumptions

- The agent can only choose to purchase or sell one stock at a time in any particular condition. Since choosing how much stock to buy or sell is a decision related to portfolio redistribution, it

[1]Code is uploaded at https://github.com/Sachi-27/FinSearch—Deep-RL-for-Finance/



```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, 30, 50)            10400

 lstm_3 (LSTM)               (None, 50)                20200

 dense_1 (Dense)             (None, 1)                 51

=================================================================
Total params: 30,651
Trainable params: 30,651
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.2: LSTM Model

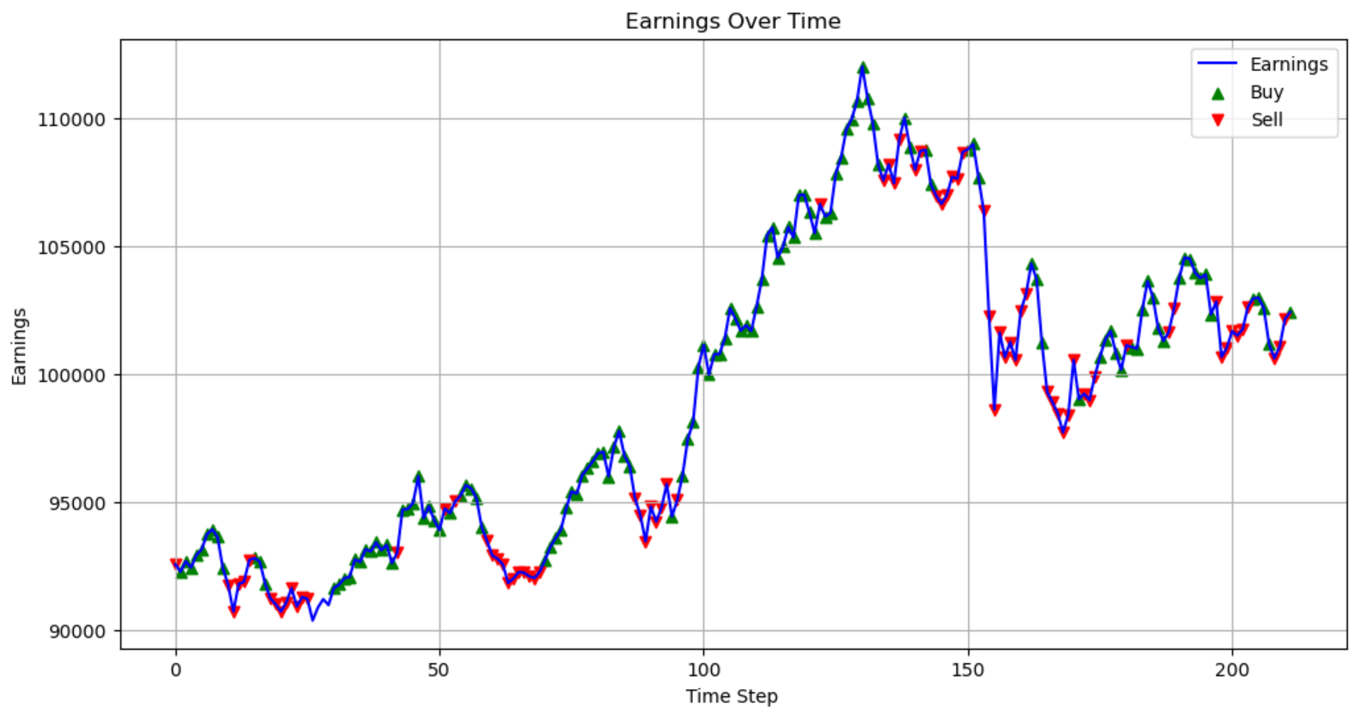


Figure 3.3: Testing model with buy/sell/hold choices

is done in an effort to keep things as straightforward as possible.

- The n-day window feature representation is a vector of consecutive adjustments to the closing price of the stock we're trading, followed by a sigmoid operation to normalise the values to the range [0, 1].
- Due to its sequential nature, training is best performed on a computer. We replay each trading episode once (1 epoch over a tiny minibatch) and update model parameters.

3.4 Agent Code Summary

3.4.1 Custom Loss Function: `huber_loss(y_true, y_pred, clip_delta=1.0)`

This custom loss function, used in Q-learning, computes the Huber loss between predicted values (`y_pred`) and target values (`y_true`). The Huber loss handles both small and large errors by using a squared loss for small errors and a linear loss for larger errors.

3.4.2 Agent Class: `Agent`

The `Agent` class represents a stock trading bot. It includes the following methods and attributes:

- `__init__(self, state_size, strategy="t-dqn", reset_every=1000, pretrained=False, model_name=None)`: Initializes the agent with various configuration parameters, including state size, action size, strategy, and model-related settings. It also creates the neural network model for the agent.
- `_model(self)`: Defines the neural network model architecture for the agent, which consists of multiple dense layers.
- `remember(self, state, action, reward, next_state, done)`: Stores relevant data in the agent's memory (replay buffer) to remember experiences.
- `act(self, state, is_eval=False)`: Determines the action to take based on the current state. During training, it can choose to explore (take random actions) or exploit (choose the best action based on the current model). In evaluation mode (`is_eval=True`), it always exploits.
- `train_experience_replay(self, batch_size)`: Trains the agent using experience replay. It samples a mini-batch of experiences from the memory and updates Q-values based on the chosen strategy (DQN, T-DQN, Double-DQN).
- `save(self, episode)`: Saves the model's weights and configuration to a file, including the episode number.
- `load(self)`: Loads a pre-trained model from a file.

The agent can be configured to use different strategies for training, including DQN, T-DQN, and Double-DQN.

3.4.3 Training Functions

The code includes functions for training and evaluating the agent:

- `train_model(agent, episode, data, ep_count=100, batch_size=32, window_size=10)`: This function trains the agent using historical price data. It iterates through the data, making trading decisions at each time step and updating the agent's memory. The agent is trained using experience replay. It returns episode information, total profit, and average loss during training.
- `evaluate_model(agent, data, window_size, debug)`: This function evaluates the trained agent using historical price data. It iterates through the data, making trading decisions and calculating total profit. It also records the trading history. The function returns total profit and the trading history.

These functions are used to train and evaluate the stock trading agent using deep reinforcement learning.
Explanation of method code is:

3.5 Method Code Summary

3.5.1 Function 1: `train_model(agent, episode, data, ep_count=100, batch_size=32, window_size=10)`

This function is responsible for training the stock trading agent.

- `agent`: The trading agent object.
- `episode`: The current episode number.
- `data`: The historical price data of the stock.
- `ep_count`: The total number of episodes.
- `batch_size`: The batch size for training the agent.
- `window_size`: The size of the state window used for feature extraction.

The function initializes various variables, such as total profit, agent's inventory, and an empty list for average loss. It then iterates through the historical price data, making trading decisions at each time step. The agent's actions include buying, selling, or holding. The function records the reward, updates the agent's memory with the experience, and trains the agent using experience replay. It returns the episode number, total profit, and the average loss during training.

3.5.2 Function 2: `evaluate_model(agent, data, window_size, debug)`

This function is responsible for evaluating the trained stock trading agent.

- `agent`: The trading agent object.
- `data`: The historical price data of the stock.
- `window_size`: The size of the state window used for feature extraction.
- `debug`: A boolean flag indicating whether to log debug information.

The function initializes variables for total profit and a history list to record trading actions. It then iterates through the historical price data, making trading decisions at each time step. The agent's actions include buying, selling, or holding. The function records the actions in the history list and calculates the total profit. It returns the total profit and the trading history.

3.6 Results

Obtained ROI for test data on the DQN implementation is 16%. This shows better results in comparison to LSTM Benchmark(10% ROI)