PORTFOLIO OPTIMIZATION USING DEEP LEARNING TECHNIQUES

A Project Report Submitted

in Partial Fulfilment of the Requirements

for the Degree of

BACHELOR OF TECHNOLOGY

in

Mathematics and Computing

by

Sakshi Sharma

(Roll No. 170123044)

Harit Gupta

(Roll No. 170123020)

to the

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

April 2021

CERTIFICATE

This is to certify that the work contained in this project report entitled **Portfolio Optimization using Deep Learning Techniques** submitted by **Sakshi Sharma (Roll No.: 170123044)** and **Harit Gupta (Roll No.: 170123020)** to the Department of Mathematics, Indian Institute of Technology Guwahati towards partial requirement of Bachelor of Technology in Mathematics and Computing has been carried out by them under my supervision.

It is also certified that this report is a survey work based on the references in the bibliography.

Turnitin Similarity: 20 %

Guwahati - 781 039
April 2020

(Dr. N. Selvaraju)
Project Supervisor

ABSTRACT

Portfolio Optimisation is a fundamental problem in Financial Mathematics. It involves choosing the best portfolio from all the available options, in accordance with some objective, generally maximizing returns and minimizing risk. This project aims to explore the applicability of state-of-the-art artificial intelligence techniques, namely Reinforcement Learning (RL) algorithms, for trading securities. We begin our discussion by glancing over Markov Decision Processes (MDP), which is the mathematical concept that helps us solve RL problems. We then examine the Deep-Q learning algorithm that uses traditional Neural Networks, to begin our study. To overcome the shortcomings of our first model, we perform analysis using Recurrent Neural Networks with Long Short Term Memory. We also test these algorithms on two stocks each in the Indian and the American markets, to generate a sizeable profit on a portfolio of a single share, for a discrete set of actions, buy/sell/hold (+1,-1,0).

Contents

List of Figures **vii**

1 Introduction **1**

1.1 Reinforcement Learning 1

1.2 Recurrent Neural Networks 2

1.3 Portfolio Optimization 3

2 Theory of Reinforcement Learning **4**

2.1 Characteristics of Reinforcement Learning 4

2.2 Markov Decision Process 7

2.2.1 History and State 7

2.3 Bellman Optimality Equation 8

2.3.1 Bellman Expectation Equation 9

2.3.2 Optimal Value Function and Policy 10

2.3.3 Solving Bellman Optimality Equation 12

2.4 Model Free Prediction and Control 14

2.4.1 Q-Learning 15

2.5 Motivation for using RNN with LSTM 17

3 Theory of Recurrent Neural Networks **18**

3.1 Concepts of Recurrent Neural Networks 18

3.2 The Vanishing Gradient Problem 20
3.3 Implications of RNN 22
3.4 Solutions to Vanishing and Exploding Gradient Problem . . . 22
3.5 LSTM 23

4 Experimental Results on Single Stock for Discrete-Time Step Model 25

5 Continuous Time Step on a portfolio of Stocks 35

6 Policy Based Learning 37
6.1 Policy Gradient 38
6.1.1 Optimisation of Policy Objective Functions 39
6.1.2 Reinforce Algorithm 40
6.2 Variance Reduction using Actor-Critic 40
6.2.1 Critic Reducing Variance 41

7 Deep Deterministic Policy Gradient 42
7.1 Background 42
7.2 Learning 43
7.2.1 Q-Learning 43
7.2.2 Policy Learning 44
7.3 Network Architecture 45
7.3.1 Target Network 45
7.4 DDPG Pseudo-Code 46

8 Experimental Results of Updated Training Model 47
8.1 Analysis of graphs 50

9 Conclusion and Potential Enhancements 51

Bibliography **53**

List of Figures

1.1 Representation of RNN 2

2.1 Diagram of RL scheme 5
2.2 Deep-Q Learning 16

3.1 Simple Neural Network 18
3.2 (RNN) Recurrent Neural Network 19
3.3 Time variants of RNN [5] 20
3.4 Vanishing Gradient Problem [5] 21
3.5 Inside RNN [4] 23
3.6 Inside LSTM [4] 24

4.1 Results for the Amazon price using DeepQ, profit earned = $ 162.73. 27
4.2 Results for the Amazon price using LSTM, profit earned = $ 193.45. 28
4.3 Results for the Facebook price using DeepQ, profit earned = $ 90.57. 29
4.4 Results for the Facebook price using LSTM, profit earned = $ 122.59. 30

4.5 Results for the Reliance price using DeepQ, profit earned = Rs. 254.25 31
4.6 Results for the Reliance price using LSTM, profit earned = Rs. 370.14 32
4.7 Results for the SBI price using DeepQ, profit earned = Rs. 332.64 33
4.8 Results for the SBI price using LSTM, profit earned = Rs. 354.25 34

6.1 Learning Models 38
6.2 Reinforce Algorithm 40
6.3 Actor Critic 41

7.1 Deep Deterministic Policy Gradient Algorithm 46

8.1 Results for portfolio using window size of 3 48
8.2 Results for portfolio using window size of 7 49
8.3 Results for portfolio using window size of 14 49
8.4 Combined result of different window sizes 50

Chapter 1

Introduction

The idea of learning by experience is deep-rooted in our environment, for instance, when an infant learns to walk or learns to speak. In both of these cases, the infant does not have any explicit teacher training him/her through the steps to speak or walk, but it does have an involuntary sensory connection to the environment. It is this sensorimotor interaction that produces tons of information about activities and rewards, that is, the outcomes of our actions and what steps to follow to accomplish our objectives. We have a subconscious connection to the environment and its reactions to our moves, in this case, the outcome of our activity, irrespective of whether we are playing an instrument or a sport.

1.1 Reinforcement Learning

Machine learning has three basic paradigms, namely supervised learning, unsupervised learning, and reinforcement learning. In reinforcement learning, the agent tries to reach the maximum value of a cumulative reward in an uncertain, potentially complex environment. It is a trial and error method

where the agent either gets rewards or penalties for the actions it performs and learns from that feedback. One of the biggest advantages is that the agent learns without a training dataset, only from its experience. Hence essentially, they are control and decision problems since the agent's actions impact its subsequent inputs and rewards.

1.2 Recurrent Neural Networks

Recurrent Neural Network (RNN) is a specific type of Neural Network which utilizes the output of the previous layer to compute the output of the current layer. RNN dominates simple neural networks in cases where it is required to remember the previous outputs. For instance: In a sentence, previous words play an important role in predicting the next word. Thus the presence of a memory-like structure is important.

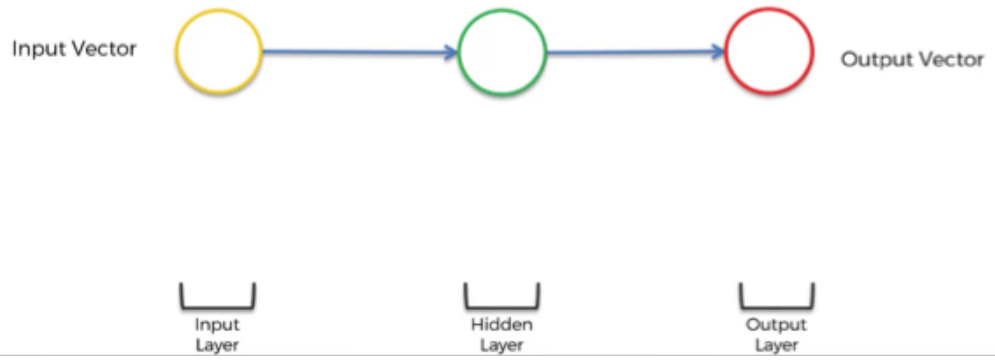

Figure 1.1: Representation of RNN

1.3 Portfolio Optimization

A portfolio is defined as a basket of financial assets and investment tools that are held by an individual, a financial institution, or an investment firm. Portfolio optimization refers to the process of choosing the best portfolio that maximizes expected return and minimizes financial risk.

Chapter 2

Theory of Reinforcement Learning

2.1 Characteristics of Reinforcement Learning

The following are the key elements of Reinforcement Learning:
i) It just has a *reward* signal, no explicit supervisor
ii) Feedback is not instantaneous, it might be delayed up to some steps or till the end
iii) Time plays an important part (data is sequential, not independent and identically distributed)
iv) Agent's current actions affect the subsequent inputs

The main elements of Reinforcement Learning are **agent, environment, action, state** and **reward**. The agent is the entity that takes actions in the environment and the environment is the one that selects observations and

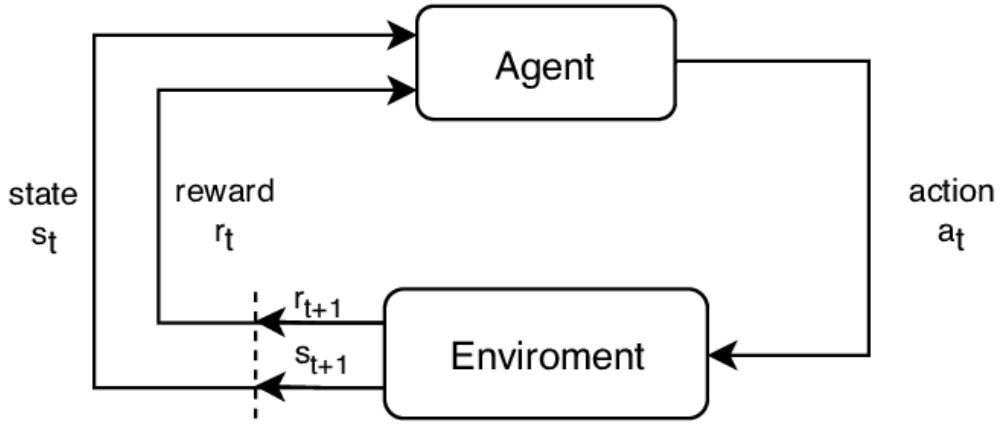

Figure 2.1: Diagram of RL scheme

rewards. In the context of our project, the agent is the Neural Network that we are training and the environment is Indian and American Stock Exchange. The elements interact as given in the diagram. At every step t,
The agent :
i) Carries out action a_t
ii) Gets an observed state s_t
iii) Also gets a scalar reward r_t
The environment :
i) Gets an action a_t
ii) Sends a state s_{t+1}
iii) Sends a scalar reward r_{t+1}

A Reinforcement Learning agent consists of one or more of these components:

1. Policy: It is the behaviour function of the agent which dictates its

actions.

(a) It is a mapping from state to action that gives actions that the agent can take while it is present in a certain state.

(b) It can be of two types :

i. Deterministic policy: $a = \pi(s)$

ii. Stochastic policy: $\pi(a|s) = P[A_t = a \mid S_t = s]$

π is the policy. It is a distribution over actions, given the states. S is the finite set of states, $s \in S$, A is the action space, $a \in A$.

2. Value function: It is the expected value of future reward. It is used to evaluate how good or bad a state is and hence make a choice between actions.

(a) The value of a state is the expected cumulative reward an agent can collect in the long term, beginning from that state.

(b) Hence it is given by the expression :
$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s]$
where $v_\pi(s)$ is the value of the state s, R_i is the reward at time i, π is the policy and γ is the discounting factor.

(c) Rewards indicate the immediate value gained by taking a particular action being in a particular state while the value function of a state indicates the long-term profitability of being in that state.

3. Model: A model predicts the future movements of the environment. We can say, it is the agent's view of the environment.

2.2 Markov Decision Process

Markov decision processes can be used to formally describe a reinforcement learning environment that is completely observable, which means that the current state can characterize the complete process.

2.2.1 History and State

History can be regarded as a sequence of all observable variables up to time t (observations, actions and rewards)

$$H_t = O_1, R_1, A_1, ..., A_{t-1}, O_t, R_t$$

They depict the sensorimotor flow of an agent. History decides what will happen next, the actions chosen by the agent, and the observations and rewards chosen by the environment. All the information is stored in a state. In formal terms, a state can be described as a function of history:

$$S_t = f(H_t)$$

But, a **Markov state** contains all the useful information about the history.

Definition 2.2.1. A state S_t is said to be Markov if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$$

This means that if the Markov property is satisfied, then given the present, the future is not dependent on the past.
$H_{1:t} -> S_t -> H_{t+1:\infty}$
The state is a sufficient statistic of the future. It means that after knowing the state, the history is not needed and can be discarded.

Definition 2.2.2. A Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where

1. $\mathcal{S}$ is the set containing all possible states

2. $\mathcal{A}$ is the set containing all possible actions

3. $\mathcal{P}$ is a transition probability matrix. It defines transition probabilities from a Markov state s to all its successor states $s^{'}$, given an action a
   $\mathcal{P}^{a}_{ss^{'}} = \mathbb{P}[S_{t+1} = s^{'} \mid S_t = s, A_t = a]$

4. $\mathcal{R}$ is a reward function.
   $\mathcal{R}^{a}_{s} = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$

5. $\gamma \in [0,1]$ is called the discount factor.

2.3 Bellman Optimality Equation

Definition 2.3.1. The return G_t is defined as the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

$\gamma \in [0,1]$ is called the discount factor that gives estimate of the current value of future rewards.

Definition 2.3.2. For a Markov Decision Process, the state-value function $v_\pi(s)$ is defined as the expected return beginning from state s, and then following policy π

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{2.2}$$

Definition 2.3.3. For a Markov Decision Process, the action-value function $q_\pi(s, a)$ is defined as the expected return starting from state s, choosing

action a and then following policy π

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{2.3}$$

2.3.1 Bellman Expectation Equation

We observe that we can decompose the state-value function defined above into two parts as follows:

1. R_{t+1} : immediate reward
2. $\gamma v_\pi(S_{t+1})$: discounted value of successor state's value function

From equation (2.2),

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \end{aligned}$$

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \tag{2.4}$$

Similarly, we can decompose the action-value function in equation (2.3) to obtain,

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \tag{2.5}$$

From equation (2.4),

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a) \tag{2.6}$$

From equation (2.5),

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \tag{2.7}$$

Put (2.7) in (2.6), to get **Bellman Expectation equation for** v_π

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s)(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')) \tag{2.8}$$

Using (2.8) and (2.6), to get **Bellman Expectation equation for** q_π

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s')\ q_\pi(s', a') \tag{2.9}$$

2.3.2 Optimal Value Function and Policy

Solving a Markov Decision Process means finding the optimal value function. This function gives the best possible performance of the MDP.

Definition 2.3.4. The maximum value function that we get considering all the policies, is called optimal state-value function $v_*(s)$.

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.10}$$

Definition 2.3.5. The maximum action-value function that we get considering all the policies, is called the optimal action-value function $q_*(s,a)$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.11}$$

A partial ordering can be defined over policies as

$$\pi \geq \pi^{'} if v_{\pi}(s) \geq v_{\pi^{'}}(s) \forall s \in S \tag{2.12}$$

Since this is only a partial ordering, there is a possibility that two policies, π_a and π_2 are not comparable. It means that there exist subsets of the state space, S_1 and S_2 such that:

$$v_{\pi}(s) \geq v_{\pi'}(s) \forall s \in S_1$$
$$v_{\pi'}(s) \geq v_{\pi}(s) \forall s \in S_2$$

Here, one policy can't be said to be better than the other. But such a case will never occur because we are dealing with finite Markov Decision Processes with bounded value functions. Hence we can state the following theorem:

Theorem 2.3.6. *For any Markov Decision Process:*

1. *There will always be a policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$. However, it is possible to have multiple optimal policies.*

2. *The optimal value function is achieved by all optimal policies, $v_{\pi *}(s) = v_*(s)$.*

3. *The optimal action-value function is achieved by all optimal policies, $q_{\pi *}(s, a) = q_*(s, a)$.*

By maximizing over $q_*(s, a)$, we can find an optimal policy

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \text{argmax}_{a \in \mathcal{A}}\, q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

The optimal value function is recursively related to optimal action-value function as follows,

$$v_*(s) = \max_a q_*(s, a) \tag{2.13}$$

Using equation (2.7),

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \tag{2.14}$$

From equation (2.13) and (2.14), we obtain **the Bellman Optimality equation for** v_*

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \tag{2.15}$$

and **the Bellman Optimality equation for** q_*,

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a q_*(s', a) \tag{2.16}$$

We can see clearly that the Bellman Optimality equation is not linear and in general, it does not have a closed-form solution. We discuss various methods to solve this equation in the next section.

2.3.3 Solving Bellman Optimality Equation

The following are some of the methods used for solving the Bellman Optimality equation:

1. Policy Iteration : It consists of the following steps:

   (a) We consider a deterministic policy, $a = \pi(s)$

   (b) We act greedily and improve the policy, $\pi'(s) = \text{argmax}_{a \in \mathcal{A}}\, q_\pi(s,a)$

   (c) Hence, at each step, the value improves
   $q_\pi(s, \pi'(a)) = \max_{a \in \mathcal{A}} q_\pi(s,a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$

(d) The improvement in value improves the value function also.
From (c),

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi^{'}(s)) = \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi^{'}(S_{t+1})) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma q_\pi(S_{t+2}, \pi^{'}(S_{t+2})) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + ... \mid S_t = s] = v_{\pi'}(s) \end{aligned}$$

$$Hence, v_{\pi'}(s) \geq v_\pi(s) \tag{2.17}$$

2. Value iteration : This method involves applying Bellman optimality backup iteratively.

   (a) At each $(k+1)th$ iteration, for all states $s \in S$, we update $v_{k+1}(s)$ from $v_k(s)$ using synchronous backups.

   (b) The iterations $v_1 -> v_2 -> ... -> v_*$ may contain value functions in the sequence, that may not map to any policy.

   (c) There is no notion of explicit policy in this case unlike policy iteration.

Till now, we solved a known Markov Decision Process. It means that we are assuming that we have with us $\mathcal{P}^a_{ss'}$ that gives the transition probabilities from a Markov state s to all its successor states $s^{'}$, given action a. This is not the case in real world scenarios, specially for stock market which is extremely complex and uncertain. Hence, we need to explore **model-free prediction and control** in which we **estimate and optimize the value function of an unknown Markov Decision Process**.

2.4 Model Free Prediction and Control

We will use model free prediction and control when either the MDP model is not known but we can sample the experience or the MDP model is known but it is too big to use and can only be used by sampling.
So far we have observed that, improving over v(s) using greedy policy requires model of Markov Decision Process due to presence of $\mathcal{P}^{a}_{ss'}$,

$$\pi^{'}(s) = \text{argmax}_{a\in\mathcal{A}}\, \mathcal{R}^{a}_{s} + \mathcal{P}^{a}_{ss'} v(s^{'})$$

While improving over q(s,a) using greedy policy is model-free,

$$\pi^{'}(s) = \text{argmax}_{a\in\mathcal{A}}\, q(s,a)$$

We discussed earlier that being in a certain state, the agent picks up the action that gives the maximum cumulative reward. However, this might lead to a situation where the complete action space is not explored. Hence, we may not find the optimal policy.
Let m be the number of possible actions from a state. We use the idea of ϵ **- Greedy exploration** for continual exploration. With probability ϵ, we choose an action at random and with probability $1-\epsilon$, we choose the greedy action.

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \text{if } a^{*} = \text{argmax}_{a\in\mathcal{A}}\, q(s,a) \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases}$$

It can be shown that for an ϵ - greedy policy π, the ϵ - greedy policy $\pi^{'}$ with respect to q_π is an improvement ie. $v_{\pi^{'}}(s) \geq v_\pi(s)$.

$$\begin{aligned} q_\pi(s, \pi^{'}(s)) &= \sum_{A \in \mathcal{A}} \pi^{'}(a|s) q_\pi(s,a) \\ &= \epsilon/m \sum_{A \in \mathcal{A}} q_\pi(s,a) + (1-\epsilon) \max_{A \in \mathcal{A}} q_\pi(s,a) \\ &\geq \epsilon/m \sum_{A \in \mathcal{A}} q_\pi(s,a) + (1-\epsilon) \sum_{a \in \mathcal{A}} \frac{\pi(a|s) - \frac{\epsilon}{m}}{1-\epsilon} q_\pi(s,a) \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a) = v_\pi(s) \end{aligned}$$

Hence, from equation (2.17), we get $v_{\pi^{'}}(s) \geq v_\pi(s)$.

2.4.1 Q-Learning

Q-Learning is a type of off-policy learning technique for model-free control. It's considered off-policy because it does not require a policy. We learn from actions that are not a part of the current policy, for example taking random actions. 'Q' refers to quality. In this case, quality depicts the usefulness of a given action in gaining some future reward.

While the **on-policy** learning comprises "learning on the job" and learning about the policy from the experience sampled by itself, **off-policy** learning means "looking over somebody else's shoulder".

In off-policy learning, we do the following:

1. Compute $v_\pi(s)$ and $q_\pi(s,a)$ by evaluating target policy $\pi(a|s)$. We follow behaviour policy $\mu(a|s)$.
   $\{S_1, A_1, R_2, ..., S_T\} \sim \mu$

2. Here we are learning about the optimal policy while following the exploratory policy. Therefore while following one policy, we might learn

about multiple policies.

3. It involves re-using the experience gained in older policies $\pi_1, \pi_2 ... \pi_{t-1}$.

Q-Learning involves improvement of both behavior and target policies.
The target policy π is *greedy* w.r.t $Q(s, a)$.

$$\pi(S_{t+1}) = \text{argmax}_a \, Q(S_{t+1}, a))$$

The behaviour policy μ is $\epsilon - greedy$ w.r.t $Q(s, a)$.
The next action is chosen using $A_{t+1} \sim \mu(.|S_t)$ which is the behaviour policy but we consider $A^{'} \sim \pi(.|S_t)$ which is the alternative successor action for updating $Q(S_t, A_t)$. We update $Q(S_t, A_t)$ towards it.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A^{'}) - Q(S_t, A_t))$$

α is called as the learning rate. The term $R_{t+1} + \gamma Q(S_{t+1}, A^{'})$ is the Q-learning target. It can be simplified as follows:

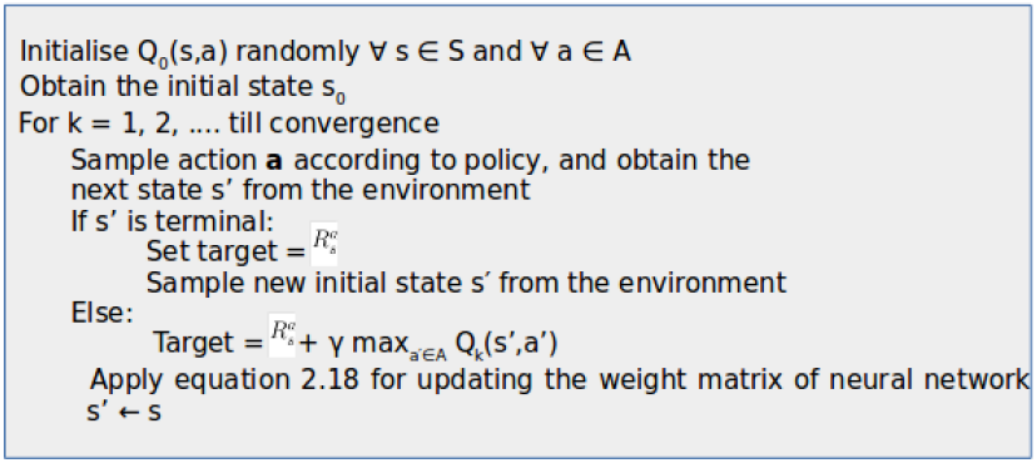

Figure 2.2: Deep-Q Learning

2.5 Motivation for using RNN with LSTM

One of the major shortcomings of traditional Neural Networks is that they lack persistence. Persistence can be explained by taking an example. While reading an essay we don't start thinking from scratch at each instance. We understand new words based on the understanding we developed from the older words. Hence, there is persistence in our thoughts.

Keeping in mind the extreme complexity and volatility of the stock market, it is known that the history of stock prices can play a major role in making accurate and effective predictions. Therefore, we chose to include the class of neural networks that use earlier stages of observations to learn and forecast future trends. This class is called **Recurrent Neural Network (RNN)**, as described by the term 'recurrent'. But RNN can't store long-term memory, therefore we introduce LSTM to enable the model to learn long-term dependencies. LSTM cells differentiate from traditional artificial in their architecture of the hidden layer. LSTM effectively utilizes information from the previous events and hence is better suited to work on the dynamic structure of stock movement and predict with accuracy.

Chapter 3

Theory of Recurrent Neural Networks

3.1 Concepts of Recurrent Neural Networks

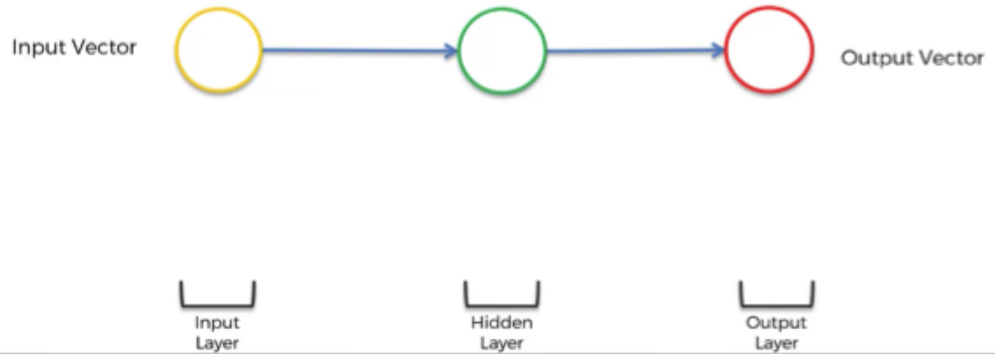

Figure 3.1: Simple Neural Network

Recurrent Neural Networks are an up-gradation to the simple Neural Networks because they use memory to forecast time series, which gives them an edge. In figure 3.1, we have a simple Neural Network. It comprises three components: an input vector, a hidden layer, and an output layer with no memory.

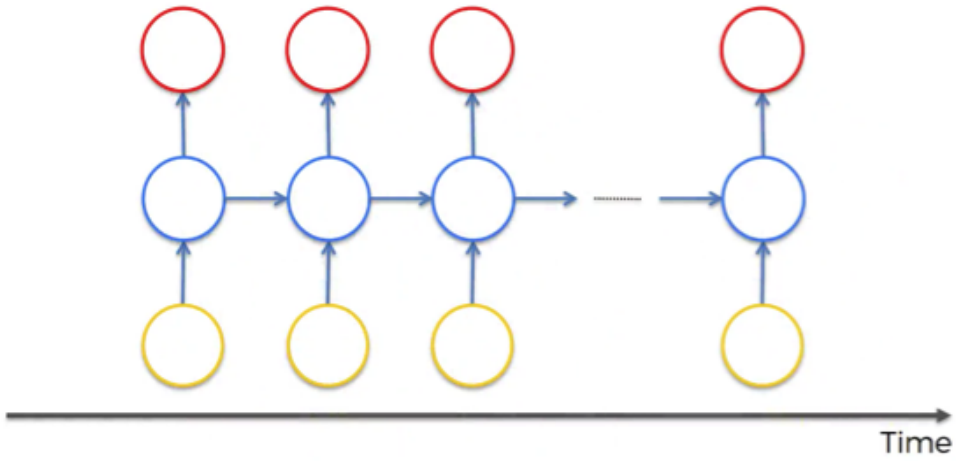

Figure 3.2: (RNN) Recurrent Neural Network

As shown in figure 3.2, RNN has an additional line in the structure that connects neighboring neurons. This hidden layer now also provides output to the next layer, thus forming a memory.
The error is computed after the information has been passed through the neural networks. The network backpropagates this error to adjust the weights. RNN possesses a memory-like structure, differentiating them from the simple neural network. This memory, like structure, enables RNN to remember the previous information and make better decisions.

3.2 The Vanishing Gradient Problem

In simple neural networks, information travels from the direction of input neurons to the output neurons. The error term is computed and is back-propagated along the neural network, which adjusts the weights accordingly to reduce the cost function.

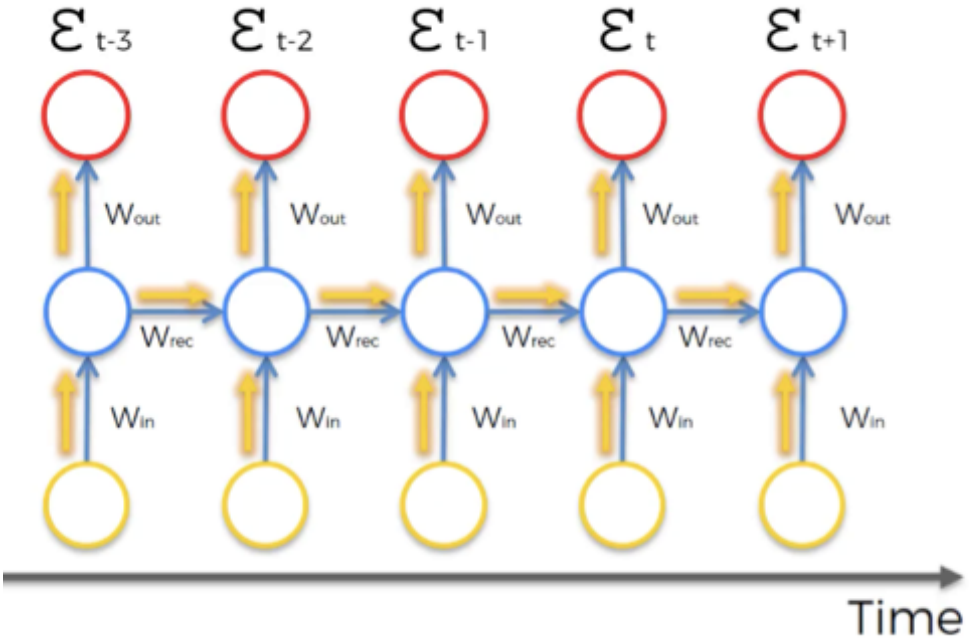

Figure 3.3: Time variants of RNN [5]

Gradient Descent algorithm is used to minimize the cost function by reaching the global minimum. This is essential in the training of neural networks. In RNN, there's an additional component of complexity. First, there is a time-series in RNNs, implying input of a time point t requires the output of previous time points. Secondly, the value of the cost function is calculated for every time point. There are computed values of error for each output layer (red circle).

After the cost function ε_t has been computed. Backpropagation of the error takes place to adjust the network's weights. Every neuron responsible for calculating the output associated with this cost function would be affected. Their weights are updated to minimize that error.

And that's where the complexity arises. In RNNs, for any time point t, all the neurons with previous time points contributed to the computation of ε_t. So, all neurons in this time frame need to be updated.

The issue is associated with adjusting the value of W_{rec} (weight recurring) – that is the weight between the nodes of RNN.

For example, to get from X_{t-3} to X_{t-2} we multiply X_{t-3} by W_{rec}. Then, to get from X_{t-2} to X_{t-1} we again multiply X_{t-2} by W_{rec}. So, W_{rec} is getting multiplied several times, and this leads to a problem: A number after getting multiplied by a small number multiple times, its value diminishes very quickly.

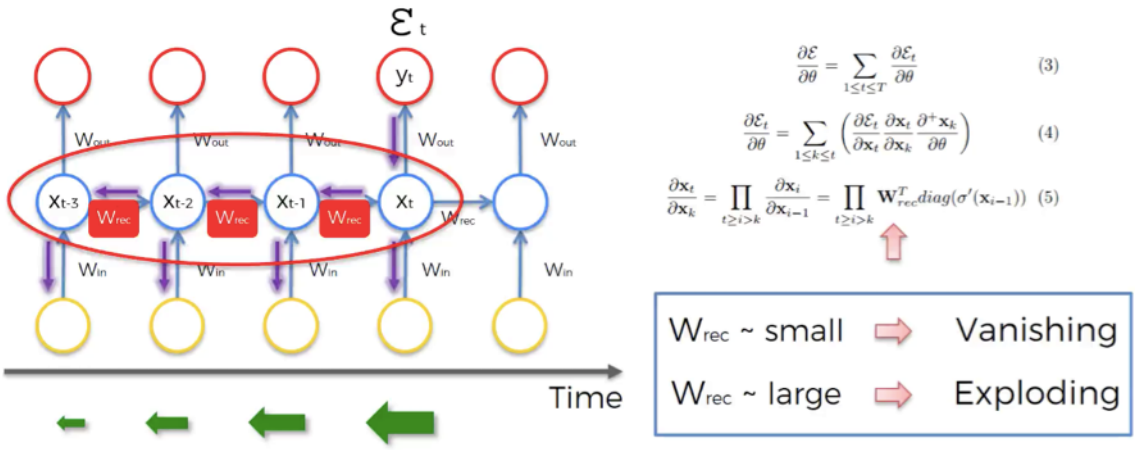

Figure 3.4: Vanishing Gradient Problem [5]

3.3 Implications of RNN

The smaller the gradient value, the more difficult it becomes to adjust the network's weights, and this process takes even more time to reach the final output.

For instance, 500 epochs can be enough to train a time point t, but it might lack when it comes to training previous time points such as t-3 as the value of gradient is small at time t-3.

However, it is not the full problem. Input for a layer depends on the output of the previous layers. This implies that training at some point t is based on the previous inputs, which are not trained themselves. This creates a domino effect resulting in poor training of the complete neural network.

To conclude: a small value of W_{rec} leads to vanishing gradient problem, and a large value of W_{rec} leads to exploding gradient problem.

3.4 Solutions to Vanishing and Exploding Gradient Problem

For exploding gradient:

- Stopping backpropagating after some time point. It is generally not the best approach as all the weights associated with the network don't get updated.
- Can manually reduce gradient or penalize to bound the value
- Can put a threshold maximum limit on the value of gradient.

For vanishing gradient :

- Weights can be initialized in such a way to reduce the potential of vanishing gradient.
- Introduce skip connections which join two different hidden layers that are far in time
- Can have gated networks - Long Short-Term Memory Networks (LSTMs).

3.5 LSTM

LSTM or Long Short Term Memory networks are a specific type of RNN that is capable of having long-term dependencies, by taking $W_{rec} = 1$. Figure 3.5 demonstrates how a standard RNN looks like from the inside.
The hidden layer in the central block receives input X_t from the input layer and also from itself in time point t-1, then it generates output h_t and also another input for itself but in time point t+1. Figure 3.5 shows the standard

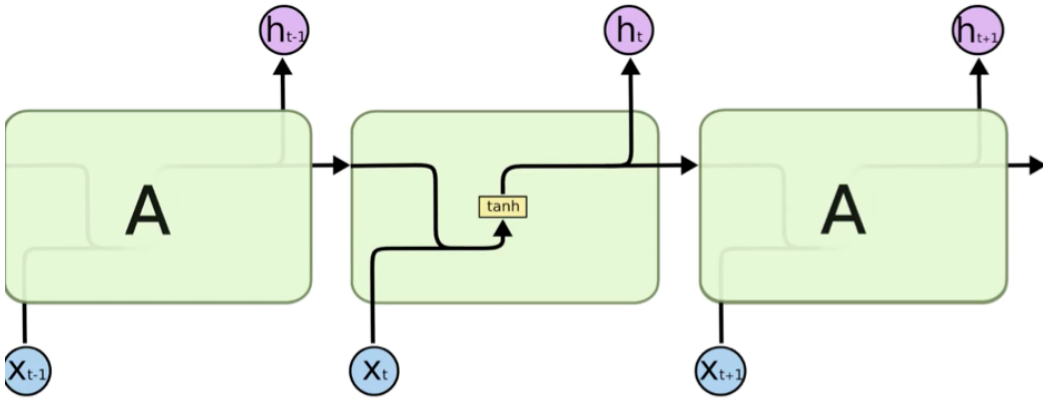

Figure 3.5: Inside RNN [4]

architecture of RNN that doesn't solve the vanishing gradient problem.
Figure 3.6 shows how LSTMs look like. One of the main differences between RNN and LSTM is their architecture. Unlike RNNs, the hidden layer of

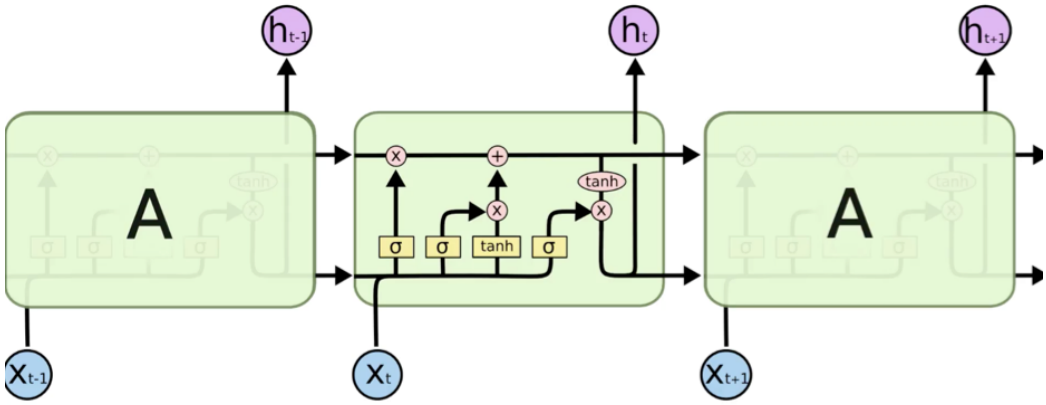

Figure 3.6: Inside LSTM [4]

LSTM consists of a gated unit. RNNs neural net layer consists of a single tanh component. Whereas there are three logistic sigmoid gates in addition to the one tanh layer in LSTM. The purpose of gates is to put a restriction on the movement of information through the cell.

Gates are responsible for deciding which information is to be passed to the next cell and which information is to be discarded. The range of the output is between 0 ('reject all') and 1 ('include all').

As stated previously in LSTMs , $W_{rec} = 1$. This feature is reflected as a straight pipeline on the top of the scheme and is usually referenced as a memory cell.

It can very freely flow through time. Though sometimes it might be removed, erased, or can be updated. Otherwise, it freely flows through time, and the problem of the vanishing gradient is removed when backpropagating through these LSTMs.

Chapter 4

Experimental Results on Single Stock for Discrete-Time Step Model

Following Experimental Results are based on the Deep Q model which uses discrete time steps on a single stock.

We implemented the Deep Q-learning algorithm and RNN with LSTM on real-world datasets, namely the stocks Amazon and Facebook in the American Market and SBI and Reliance in the Indian Market. There are a few characteristics that need to be kept in mind which we mention below:

- The interaction of the trading agent with the financial market (the environment) is at discrete time steps.
- The legal set of actions for the agent include buying, selling, or do nothing.
- The stock exchange presents new and unpredictable information to the

agent after every time step which enables the agent to make trading decisions. However, the model of the stock exchange is completely unknown.

- We only take positions +1, 0, or -1, that is, buy/sell nothing, or buy/sell 1 unit of stock as a trial run for implementation. This ensures that our orders to the exchange will not affect the spot price.
- The rewards are simply the profits gained after closing a long position.

We present below our results obtained on the four stocks mentioned above. We used training data of daily closing prices from 1st Jan 2014 to 31st June 2014 and tested our agent on data from 1st July 2014 to 1st July 2015. The experimental values match with the theoretical- LSTM provides better results as compared to DeepQ Learning.

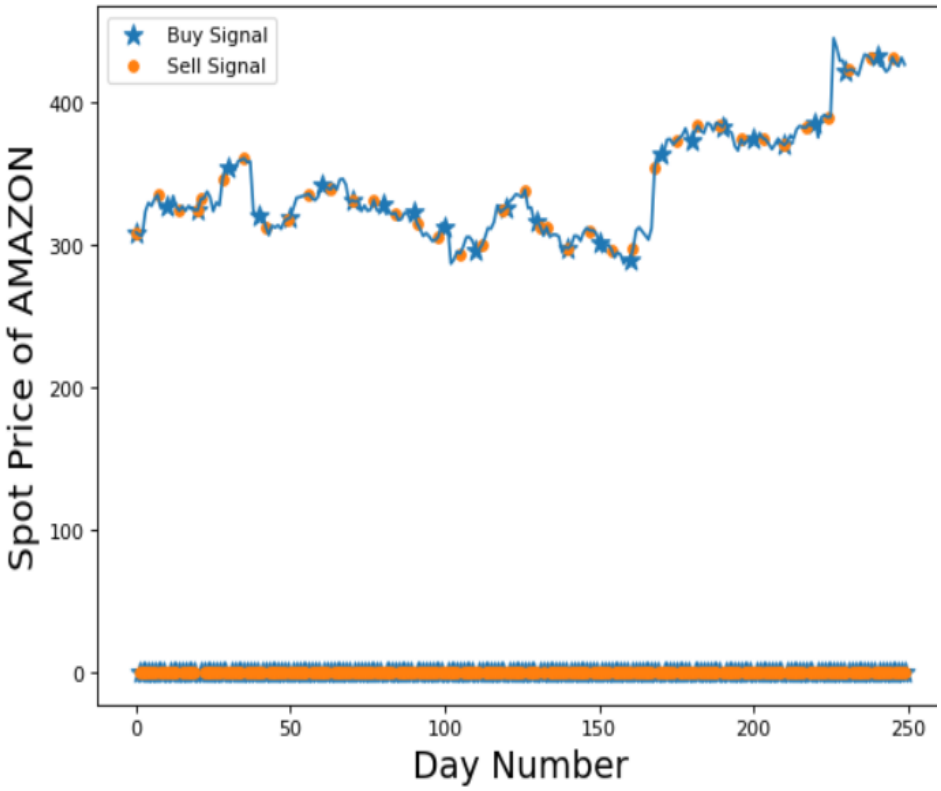

Figure 4.1: Results for the Amazon price using DeepQ, profit earned = $ 162.73.

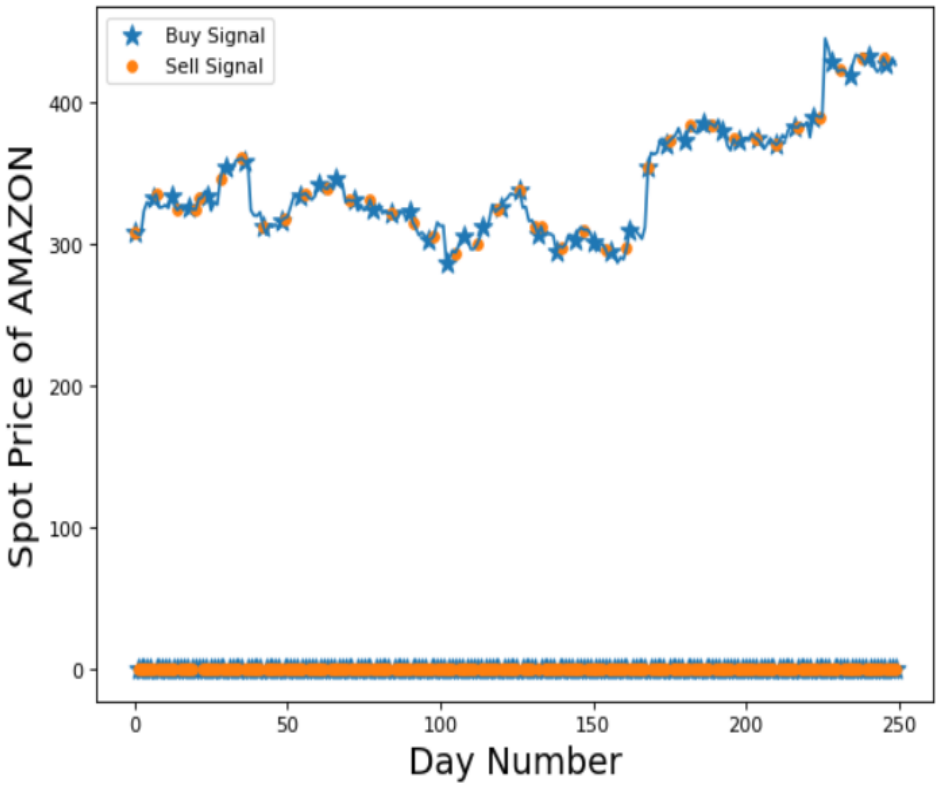

Figure 4.2: Results for the Amazon price using LSTM, profit earned = $ 193.45.

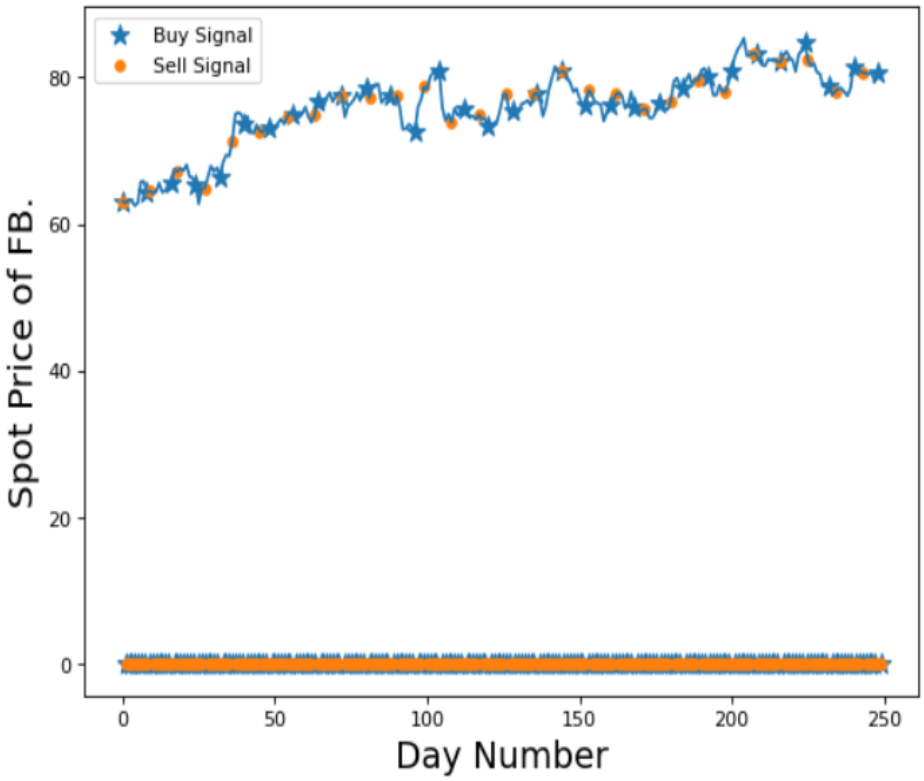

Figure 4.3: Results for the Facebook price using DeepQ, profit earned = $ 90.57.

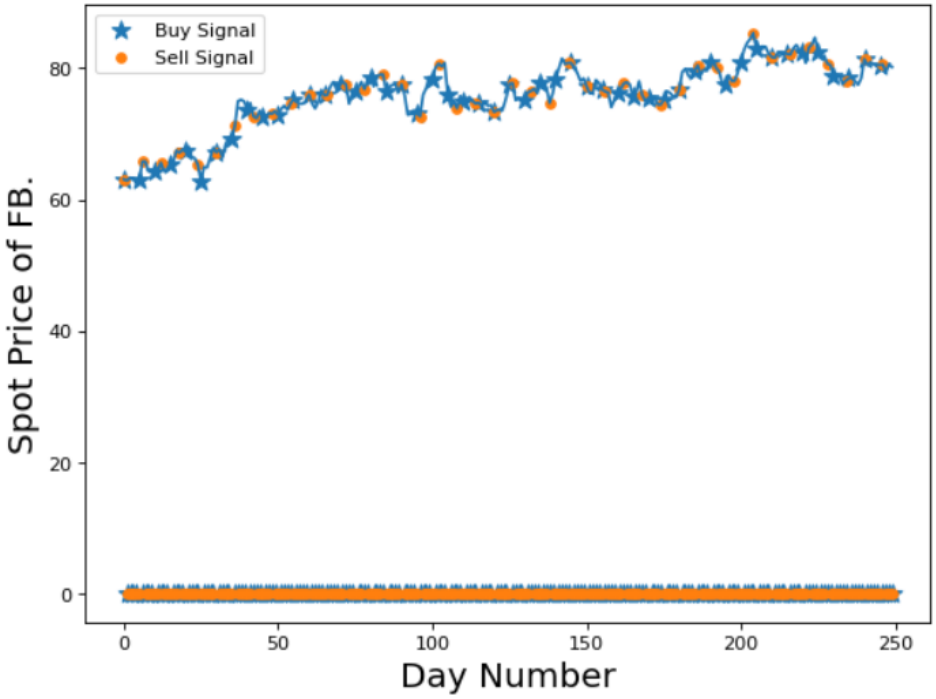

Figure 4.4: Results for the Facebook price using LSTM, profit earned = $ 122.59.

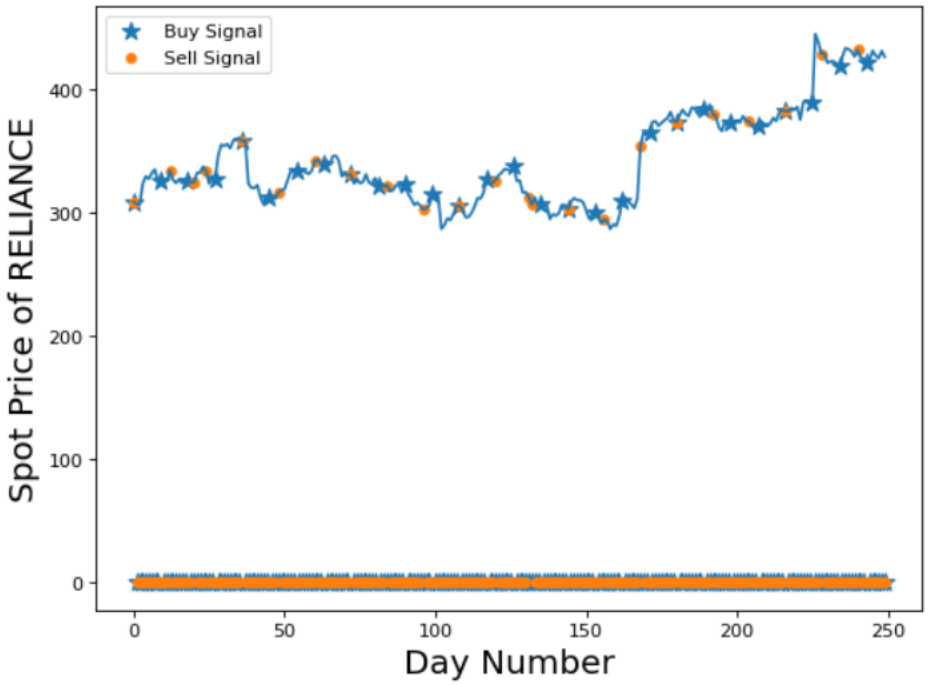

Figure 4.5: Results for the Reliance price using DeepQ, profit earned = Rs. 254.25

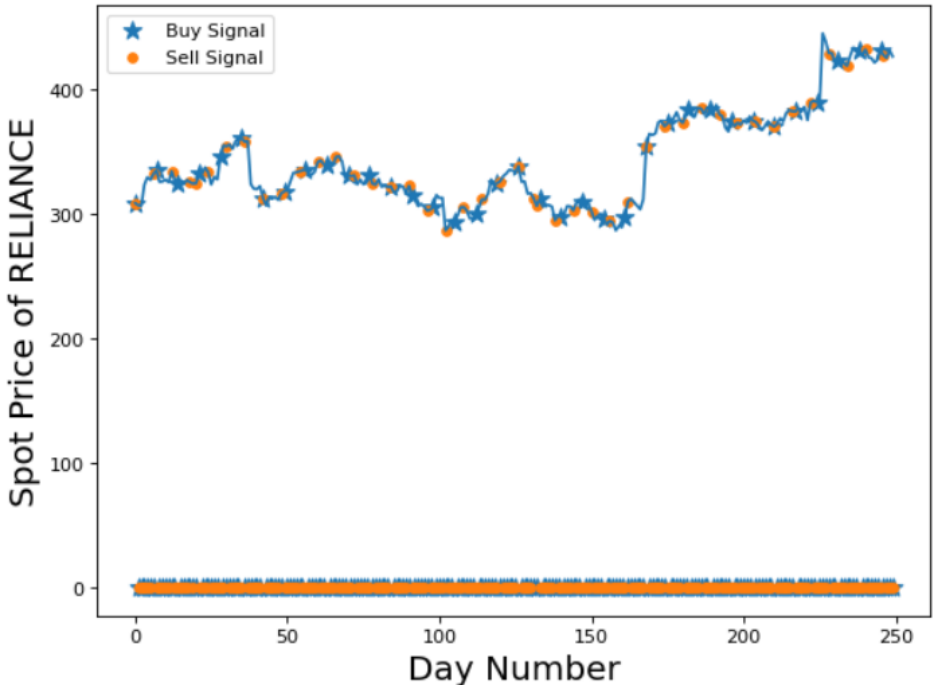

Figure 4.6: Results for the Reliance price using LSTM, profit earned = Rs. 370.14

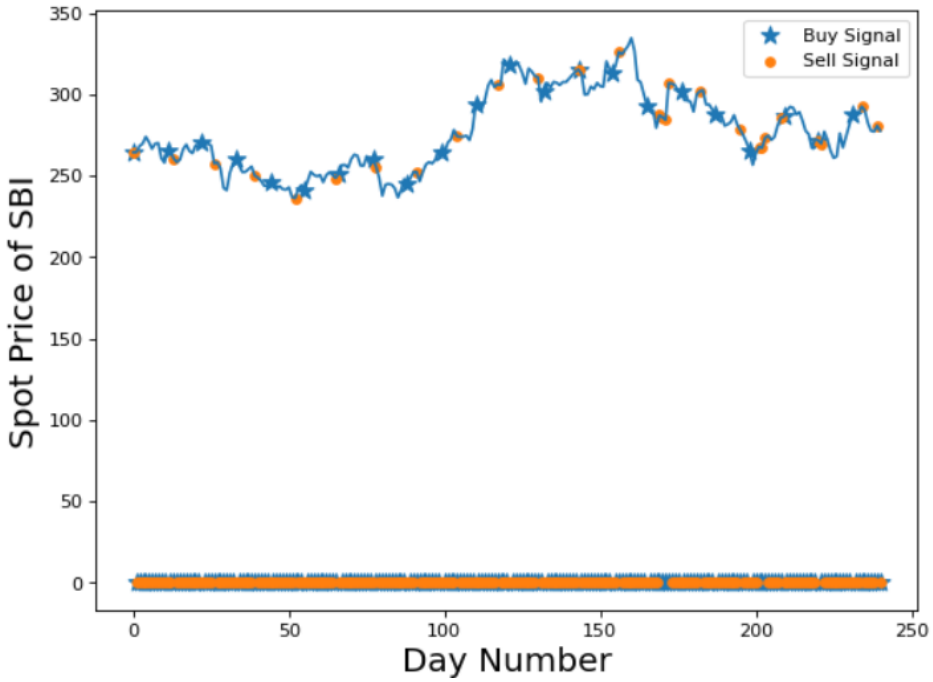

Figure 4.7: Results for the SBI price using DeepQ, profit earned = Rs. 332.64

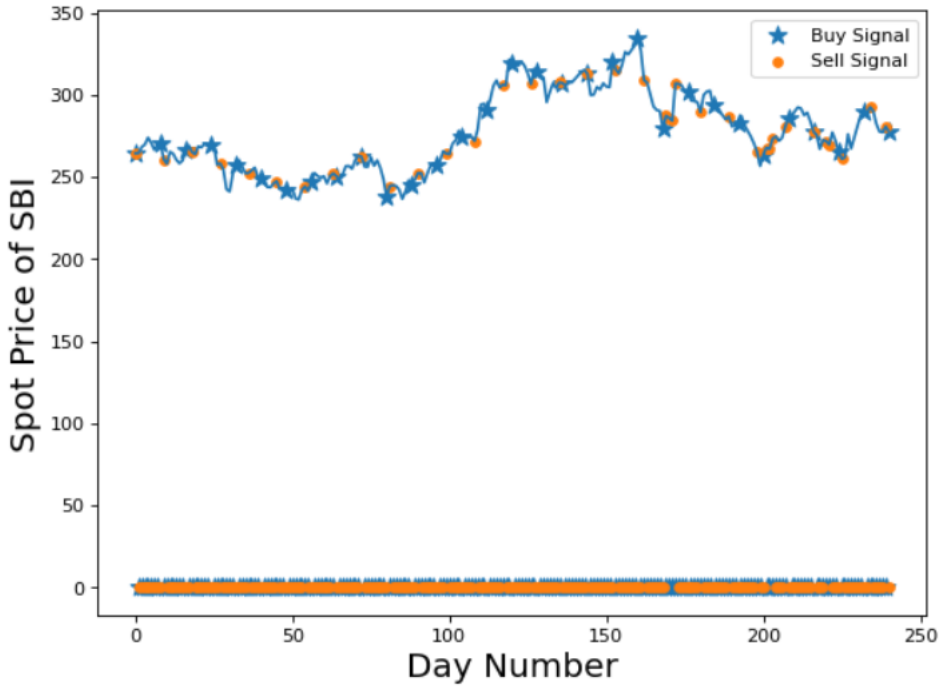

Figure 4.8: Results for the SBI price using LSTM, profit earned = Rs. 354.25

Chapter 5

Continuous Time Step on a portfolio of Stocks

We began our discussion with a basic theoretical background in Reinforcement Learning and Portfolio Optimisation. We described portfolio optimization as a continuous mathematical optimization problem with objective function serving as an indicator of how an investor should take trading actions in the market to maximize the objective function. We made an intuitive relation between the trading profits of the portfolio and rewards generated in a control problem using Reinforcement Learning.

Further, we described the important elements of RL such as Agent, Environment, Policy, Reward, and Value. We stated the mathematical concepts that underpin the working of Reinforcement Learning such as the Markov Decision Process and Bellman Optimality Equation. The equation being non-linear required us to explore methods like Dynamic Programming using Policy Iteration, Value Iteration, and Q-Learning.

We finally shifted our focus to the problem at hand i.e. portfolio optimization. Finally, we state the last building block of Deep Q-Leaning (DQN) namely Value Function Approximation using Neural Networks.

We implemented the DQN algorithm in the data set of SBI, Reliance, Facebook, and Amazon. Our model transformed the market data into quantifiable features and took three integral trading actions, viz., buy (+1), sell (-1), and hold (0) in discrete steps. We focus on improving our model to develop a more real-world imitation of the trading environment. We enumerate the overall walk-through of our model enhancements goals.

- Our goal will be to extend the model to take continuous actions and optimize a multi-stock portfolio.
- We explore the textbook methods for continuous control in Reinforcement Learning like Actor-Critic methodology and Deep Deterministic Policy Gradient (DDPG).
- Finally, we implement the DDPG model to take continuous actions on a multi-stock portfolio.

We first explore the theory of Policy Based Learning Methods. We incrementally develop the most basic REINFORCE algorithm by applying the policy gradient method on Policy Objective Function. We branch out our discussion to include Actor-Critic methods which allow us to further optimize REINFORCE algorithm by reducing variance. Finally, we explore the theory of our crux algorithm DDPG to develop the improved trading model.

Chapter 6

Policy Based Learning

In the models that we explored till now ie. Q-Learning, Deep Q-Networks (DQN), we attempted to optimize the reward by approximating the value function or action-value function. We use ϵ - greedy selection over the action-value function to formulate an optimized policy. This methodology is called as **Value-Based Learning**.

We'll now look at methods for optimizing the parameterized policy directly. These methods do not necessitate the study of value functions. **Policy-Based Learning**, as opposed to learning an implicit policy (via ϵ - greedy selection), learns the exact deterministic policy. These methods have an advantage over value-based learning in that they make efficient use of space, which is beneficial when working with continuous action spaces and high-dimensional state spaces.

The **Actor-Critic methodology** entails learning both value functions and policy. These techniques combine Value-Based and Policy-Based Learning techniques.

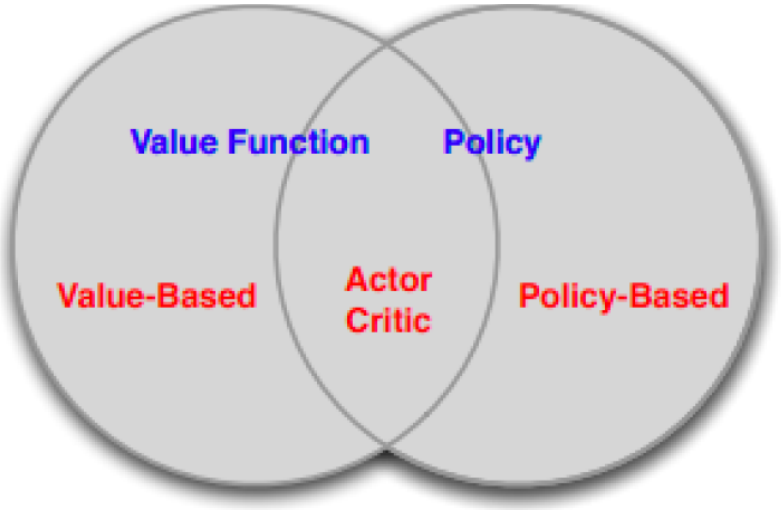

Figure 6.1: Learning Models

6.1 Policy Gradient

Value-Based Learning parametrizes the value function using a parameter θ and approximates the optimized values by Q-Learning methods. Likewise, our goal is to formulate an algorithm using Policy-Based Learning to parametrize the policy. For policy optimization, there is an objective function.

Parameterized Policy

$$V_\theta(s) = V^\pi(s) \tag{6.1}$$

$$Q_\theta(s, a) = Q^\pi(s, a) \tag{6.2}$$

$$\Pi_\theta = P[a \mid s,\ \theta] \tag{6.3}$$

6.1.1 Optimisation of Policy Objective Functions

The reward optimization function for a single step MDP Policy-Based Learning is given by -

$$J_{avR}(\theta) = \sum_{s} d^{\pi_\theta}(s) \sum_{a} \pi_\theta(s,a) R_s^a \tag{6.4}$$

here, d^{π_θ} is the **stationary distribution** for the states of the policy.

By changing the gradient's direction, the Policy Gradient algorithm will converge to a maximum in $J(\theta)$. In this case, the change in θ is given by -

$$\triangle\theta = \alpha \nabla_\theta J(\theta) \tag{6.5}$$

here, α is the learning rate.
Using Likelihood Ratios, we form an analytic function in order to obtain the policy gradient.

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \tag{6.6}$$

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \nabla_\theta [\log(\pi_\theta(s,a))] \tag{6.7}$$

here $\pi_\theta(s,a)\nabla_\theta[\log(\pi_\theta(s,a))]$ is the score function.

The optimisation function $J(\theta)$ can be restated as -

$$J(\theta) = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) R_a^s \tag{6.8}$$

$$\nabla_\theta J(\theta) = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a)\ \nabla_\theta [\log(\pi_\theta(s,a))]\ R_a^s \tag{6.9}$$

For a general MDP, we can replace the instantaneous reward with the action-value function.

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta[\log(\pi_\theta(s,a))]Q^{\pi_\theta}(s,a)] \tag{6.10}$$

6.1.2 Reinforce Algorithm

REINFORCE algorithm is the most basic algorithm that optimizes the policy by an unbiased sampling of state-action episodes. It uses Monte Carlo sampling to generate sample episodes. The pseudo-code of REINFORCE algorithm is -

function REINFORCE
Initialise θ arbitrarily
for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
for $t = 1$ to $T - 1$ **do**
$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
end for
end for
return θ
end function

Figure 6.2: Reinforce Algorithm

6.2 Variance Reduction using Actor-Critic

REINFORCE Algorithm or Monte Carlo Policy Gradient has high variance and bias problems. This is because of noise and randomness in Monte Carlo episode generation which may generate very deviating trajectories. To improve the stability of REINFORCE algorithm, we introduce actor-critic methodology.

6.2.1 Critic Reducing Variance

Actor-Critic method keeps two sets of parameters.The two elements of this methodology are -

1. **Critic** - Computes action-value function and updates parameter w
2. **Actor** - Takes cue from critic and changes the policy parameter θ in direction suggested by critic accordingly

The pseudo-code of actor-critic (on Policy Gradient) algorithm is -

Algorithm 1 Q Actor Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \dots T$: **do**
  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$
  Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \log \pi_\theta(a|s)$; Compute
  the correction (TD error) for action-value at time t:
    $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
  and use it to update the parameters of Q function:
    $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
  Move to a $\leftarrow a'$ and s $\leftarrow s'$
end for

Figure 6.3: Actor Critic

Chapter 7

Deep Deterministic Policy Gradient

7.1 Background

While maximising rewards, algorithms like Deep-Q Network and Policy Gradient choose actions by finding the action for which the optimised action-value function has the highest value. For discrete action spaces, it simply chooses the action a*(s) s.t. -

$$a^*(s) = \underset{a \in A}{\operatorname{argmax}}\, Q^*(s, a) \tag{7.1}$$

For continuous action spaces, the usual optimization method will calculate Q*(s, a) computationally inefficient. So, DDPG algorithm in- stead of learning action-value function, **provides a direct approximate map to optimized action for each state transition**

7.2 Learning

7.2.1 Q-Learning

The Bellman Equation for optimizing the action-value function is

$$Q^*(s,a) = \mathbb{E}[r(s,a) + \gamma \max_{a'} Q^*(s^{'}, a^{'})] \tag{7.2}$$

This Bellman equation is the starting point for learning an approximator to Q*(s, a). Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set D of transitions (s, a,r,s'). We can set up a mean-squared Bellman error (MSBE) function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi, D) = \mathbb{E}_{(s,a,r,s')\in D}[Q_\phi(s,a) - Q^*(s,a)] \tag{7.3}$$

$$L(\phi, D) = \mathbb{E}_{(s,a,r,s')\in D}[Q_\phi(s,a) - \mathbb{E}[r(s,a) + \gamma \max_{a'} Q^*(s^{'}, a^{'})]] \tag{7.4}$$

where $Q_\phi(s,a)$ is a target optimized network. We have to minimize this loss function.

The key element of DDPG Q-Learning is Replay Buffers. The algorithm maintains the set of experienced transitions and rewards in a set D. These set of experiences are sampled to learn the Q-Value and decrease the MSBE. Some practical issues with using replay buffers are associated with the size of the set D. If the set D stores a large number of experiences, then the algorithm becomes computationally inefficient. On the other hand, if set D stores only recent experiences, then there is a risk of over-fitting. Hence practical trade-offs are implemented while applying the algorithm depending upon the

problem and efficiency of the system.

7.2.2 Policy Learning

The goal of the algorithm is to learn a deterministic policy for each state transition. We need to arrive at a policy μ_θ(s) which will optimize the action-value function $Q_\phi(s, a)$. The objective function can be formulated as -

$$J(\theta) = \mathbb{E}_{s \in D}[Q_\phi(s, \mu_\theta(s))] \tag{7.5}$$

We need to find the parameter θ which maximizes J(θ). Since we are dealing with continuous action spaces, we can assume that the action-value function $Q_\phi(s, a)$ is differentiable with respect to action. This allows us to use the gradient descent method to optimize the objective function J(θ). The gradient of the objective function is formulated as -

$$\nabla_\theta J(\theta) = \nabla_\theta \, \mathbb{E}_{s \in D}[Q_\phi(s, \mu_\theta(s))] \tag{7.6}$$

As explained earlier, DDPG employs replay buffers to sample experiences for Q-Learning, the same process is applied for Q-Learning as well. Suppose the method samples a batch of transitions B from experience set D, then the gradient for J(θ) and parameter update equations are given by -

$$\nabla_\theta J(\theta) = \nabla_\theta \frac{1}{|B|} \sum\nolimits_{s \in B} Q_\phi(s, \mu_\theta(s)) \tag{7.7}$$

$$\theta_{new} = \theta_{old} + \nabla_\theta \frac{1}{|B|} \sum\nolimits_{s \in B} Q_\phi(s, \mu_\theta(s)) \tag{7.8}$$

In each iteration, once the algorithm updates Q-value and Policy itself, target networks are updated changing them in the direction of optimized learned

networks.

7.3 Network Architecture

DDPG is an off-policy algorithm. As shown in Figure 7.1, DDPG learns both Value and Policy. This requires the introduction of additional learning networks over and above the networks used in Policy Gradient.

7.3.1 Target Network

DDPG uses four neural networks. Two are implicit networks for Value and Policy Learning. Two are target networks for optimized value and policy.

1. Θ^{Q} - Q Network
2. Θ^{μ} - Deterministic Policy Function
3. $\Theta^{Q^{'}}$ - Target Q Network
4. $\Theta^{\mu^{'}}$ - Target Policy Network

The goal of DDPG is to develop a network $Q_*(s, a)$ with the target of minimizing the MSBE. The deciding term which needs to be minimized is -

$$r(s, a) + \gamma \max_{a^{'}} Q^*(s^{'}, a^{'}) \tag{7.9}$$

But Since the parameter * which we want to optimize itself appears in target, the problem becomes unstable. Hence we choose a nearby parameter ϕ which optimizes the MSBE minimization and this parameter ϕ defines the target network $Q_\phi(s, a)$. After each iteration of Q-Learning and Policy Learning the target network is updated using Polyak averaging

7.4 DDPG Pseudo-Code

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

Figure 7.1: Deep Deterministic Policy Gradient Algorithm

Chapter 8

Experimental Results of Updated Training Model

Following Experimental Results are based on the DDPG model which uses continuous time steps on a portfolio of stocks.

Our model rather than trading on a single stock, trades on our portfolio which consists of stocks from NASDAQ100 that we feel are representative of different sectors in the index fund. They include “AAPL”, “ATVI”, “CMCSA”, “COST”, “CSX”, “DISH”, “EA”, “EBAY”, “FB”, “GOOGL”, “HAS”, “ILMN”, “INTC”, “MAR”, “REGN” and “SBUX”.

The price on each day contains open, high, low, and close. We use data from 2012-08-13 to 2015-08-12 as training data and data from 2015-08-13 to 2017-08-11 as testing data.

Data of 2020-2021 hasn’t been used due to covid pandemic since Reinforcement learning model will not be able to understand a

unique and extraordinary event of such scale.

We have used the DDPG model and have made 2 cases. One with normal CNN neural networks and the other using LSTM. Also, we have taken different window sizes of neural networks to compare the results.

The market value is obtained by equally distributing your investment to all the stocks. We find that LSTM predictor-based models have better performance than the CNN predictor-based models in practice as well.

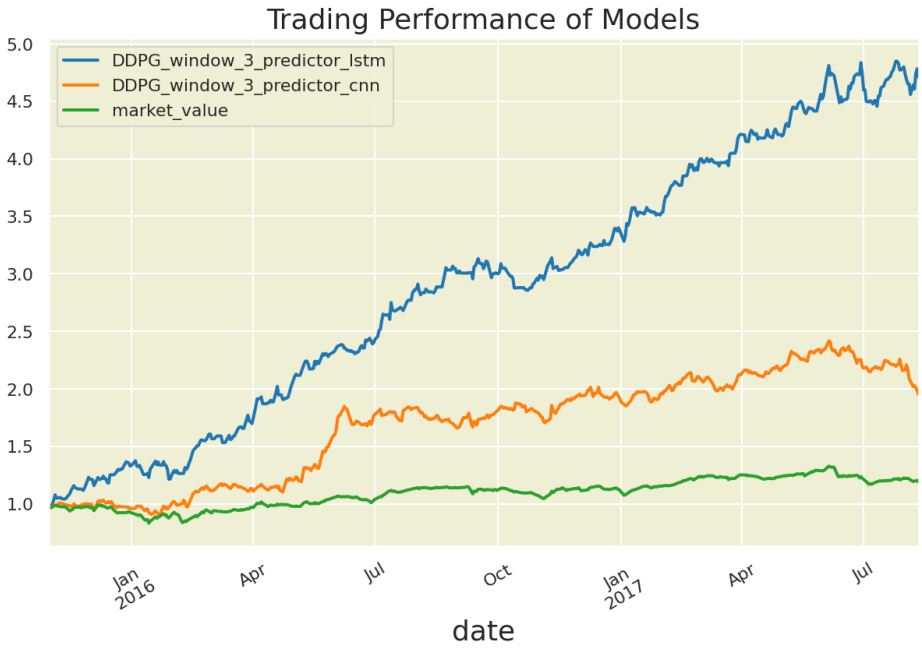

Figure 8.1: Results for portfolio using window size of 3

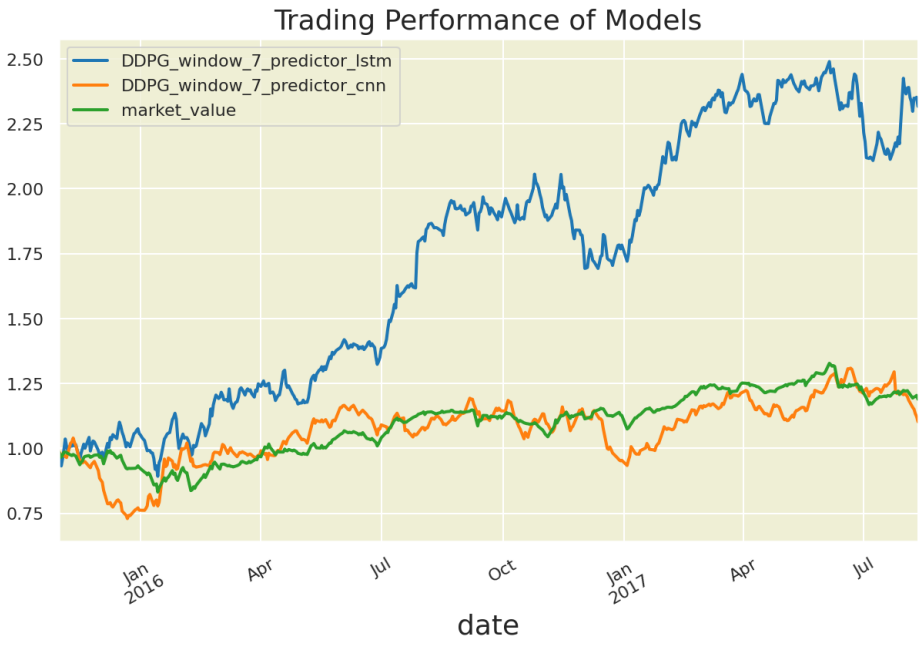

Figure 8.2: Results for portfolio using window size of 7

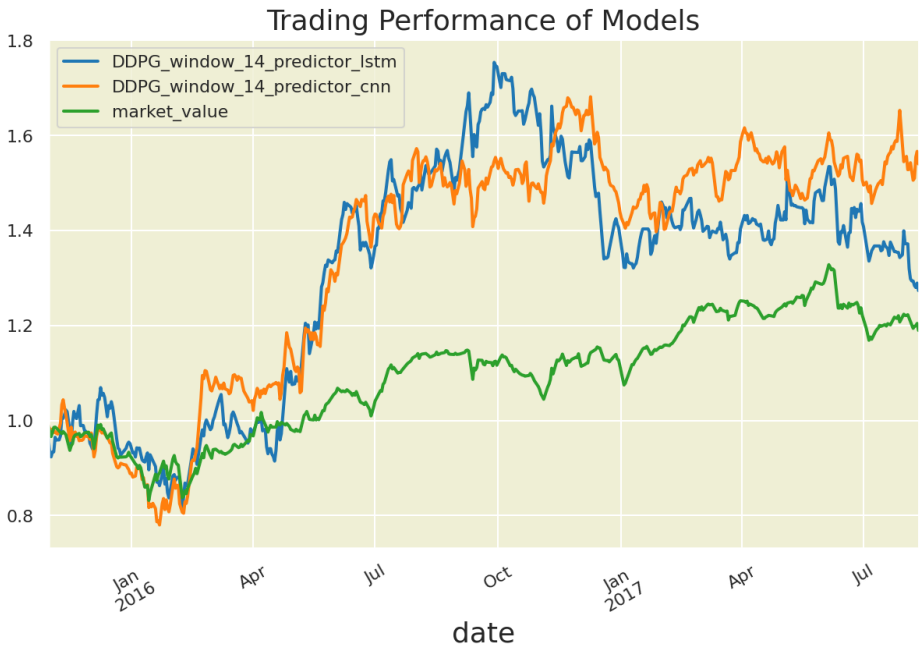

Figure 8.3: Results for portfolio using window size of 14

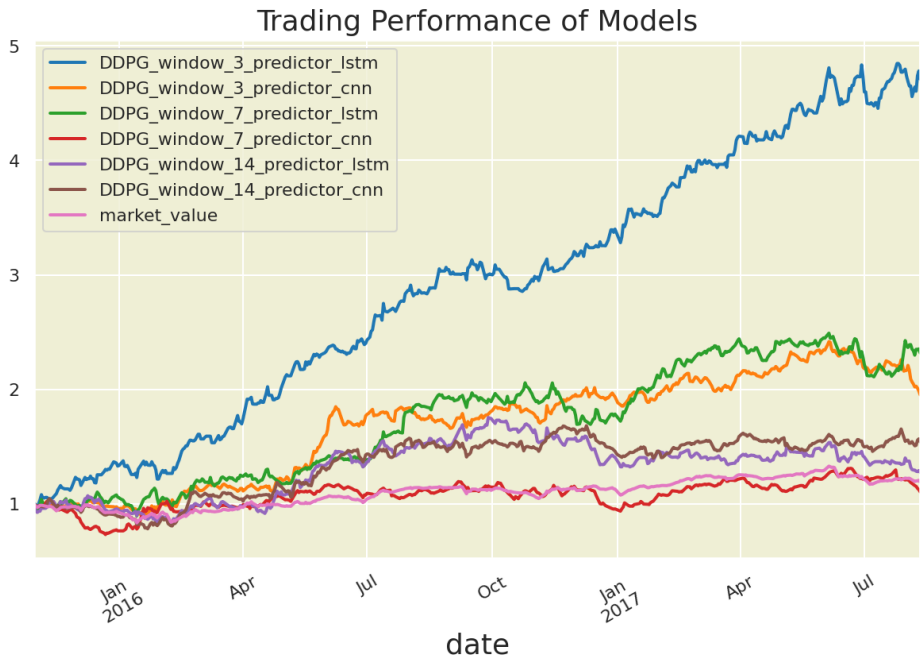

Figure 8.4: Combined result of different window sizes

8.1 Analysis of graphs

We can see smaller the window size, the more the difference between CNN and LSTM. The difference between the two models starts to decrease as window size increases as both the models have more amount of data to train with.

The larger the window size, the more time it takes to train the model. It took 10 hours to train the model with size 14.

Also using LSTM with DDPG produces better performance compared to using DDPG with CNN as LSTM improves the performance of the neural networks and thus the model.

Chapter 9

Conclusion and Potential Enhancements

Following our earlier endeavors wherein we took only ternary positions (buy, sell, hold) in a portfolio containing a single stock, we extended our project by allowing the investor to hold a portfolio containing multiple stocks, as well as hold continuous positions in those stocks. We implemented the Deep Deterministic Policy Gradient algorithm, which in layman terms is DQN with continuous action spaces.

By parameterizing the policy as well along with value function approximation, the algorithm can learn a deterministic policy which aids in making investment decisions. By observing the graphs and the results in the previous section, we can firmly say that our algorithm does perform better than blindly investing our proceeds into the market index.

A potential enhancement to the algorithm is that we can implement other state-of-art continuous reinforcement learning algorithms such as Proximal

Policy Optimization (PPO) and Policy Gradient (PG)in portfolio management for continuous actions. However, due to time constraints, that has been left as future scope.

Bibliography

[1] Chien Yi Huang. Financial trading as a game: A deep reinforcement learning approach, 2018.

[2] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. A deep reinforcement learning framework for the financial portfolio management problem, 2017.

[3] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[4] Christopher Olah. Understanding lstm networks, 2015. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[5] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013.

[6] David Silver. Ucl course on reinforcement learning. https://www.davidsilver.uk/teaching/.

[7] Christopher Yoon. Deep deterministic policy gradients explained. https://bit.ly/2yni7v5.