Portfolio Optimisation using Reinforcement Learning Algorithms

A Project Report Submitted

in Partial Fulfilment of the Requirements

for the Degree of

BACHELOR OF TECHNOLOGY

in

Mathematics and Computing

by

Neelabh Tiwari & Atharva Amdekar

(Roll No. 160123024 & 160123047)

to the

DEPARTMENT OF MATHEMATICS

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781039, INDIA

May 2020

CERTIFICATE

This is to certify that the work contained in this project report entitled "Portfolio Optimisation using Reinforcement Learning Algorithms" submitted by Neelabh Tiwari (Roll No.: 160123024) and Atharva Amdekar (Roll No.: 160123047) 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 him/her under my supervision.

It is also certified that, along with literature survey, empirical analysis has been done by the students under the project.

Turnitin Similarity: 20%

Guwahati - 781039
May 2020

(Prof. N. Selvaraju)
Project Supervisor

ABSTRACT

Portfolio Optimisation is a fundamental problem in Financial Mathematics. The objective of this project is to explore the applicability of state-of-the-art artificial intelligence techniques, namely Reinforcement Learning algorithms, for trading securities in the Indian stock market. We begin our discussion by glancing over Markov Decision Processes (MDP), the mathematical basis that underpins this RL algorithm. We then examine the Deep-Q-learning algorithm (DQN) to begin our study. We also test the algorithm on two stocks in the Indian market to generate a sizeable profit on a portfolio of a single share, for a discrete set of actions, buy/sell/hold (+1,-1,0).

After setting up an initial model of portfolio management using DQN, we move towards enhancing the capabilities of our model. We explore advance methodologies and algorithms like Actor-Critic and Deep Deterministic Policy Gradient (DDPG). We employ the salient feature of DDPG in allowing our model to take continuous actions making it a more realistic imitation of real world market actions. Essentially, we allow the investor to buy and sell equities in continuous fractional units and quantify the risk assessment through utility of the investment. We compare the trading rewards of our portfolio with the trading in the corresponding market index and analyse the performance of our model on Indian stock market data.

Contents

List of Figures

Chapter 1

Introduction

The idea of learning by interacting 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 so as 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

Reinforcement learning involves learning what actions to take, that is, how to link certain circumstances to corresponding actions to maximize a numerical reward. Essentially, they are control and decision problems since the agent's actions impact its subsequent inputs and rewards. Furthermore, unlike the

case of machine learning algorithms, the agent is not instructed what specific steps or actions to undertake but instead discover and investigate which actions render the optimal reward by either exploiting or exploring. These characteristics: being control problems, not having explicit specifications as to what actions to take, and where the impact of actions and rewards affect future time periods as well, are three unique, distinctive attributes of reinforcement learning.

1.2 Portfolio Optimisation

A portfolio is defined as a group of financial instruments such as bonds, stocks, cash, equities, etc, which is held by investors depending on the individual risk preference and investment objective.

Portfolio optimization can be defined as the process of allocating appropriate weights to the assets in our portfolio so as to maximise a pre-defined objective function. This objective function typically serves as an indicator as to how well the portfolio is performing in the market with respect to factors such as expected return and risk.

Chapter 2

Theory of Reinforcement Learning

2.1 Essential Components of Reinforcement Learning

Broadly speaking, there are five main elements of Reinforcement Learning.

1. The **agent** - The entity which takes an action in the environment.
2. The **environment** - In the context of this project, the environment is the financial environment (Indian Stock Exchange).
3. The **reward** -
   (a) The objective of the reinforcement learning problem is characterized by a reward. The agent's environment sends a numerical reward after every time step.
   (b) The agent's goal is to maximise the total reward that it receives from the environment in the long run.

(c) Thus, what is good and what is bad is solely defined by the signals that the agent receives. Therefore, the favourable and unfavourable events for the agent are decided by the reward signals it receives.

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

where Return G_t = total discounted reward from time step t.
R_i = reward gained at time step i and,
γ = discount factor.

4. The **Policy** - The agent's behavior is dictated by the policy.

   (a) In other words, a policy is a function that maps the agent's states to the actions that can be taken while the agent is present in those states.

   (b) In some circumstances, the policy is a simple function or a lookup table, whereas in others it may require extensive computation.

   (c) The policy is the heart of reinforcement learning algorithms, meaning that it is necessary as well as sufficient to dictate the agent's behavior.

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{2.2}$$

π is the policy which is a distribution over actions given states.
a ϵ A where **A is the action space** and, s ϵ S where **S is a finite set of states**.

5. The **Value** function - It defines what is favourable in the long run.

   (a) In simple terms, the value of a state is the expected cumulative reward an agent can collect in the long term, starting from that state.

   (b) While rewards indicate the immediate value gained by taking a particular action, the value function of a state indicates the long-term profitability of being in that state.

$$V(s) = \mathbb{E}[G_t|S_t = s] \tag{2.3}$$

where V(s) = long term value of being in the state s,
G_t = return gained starting from time step t and,
S_t is the state of the agent at time step t.

The value function can be broken down into two parts:
- Immediate reward R_{t+1}
- Discounted value of the successor state

Hence the value function can be expressed in the form of a well-known formula called the Bellman Equation.

Bellman Equation:

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

Similarly, there exists another similar function which tells the favourability of taking an action a while the agent is in the state s. This function is known as the *action-value function* and is denoted as $q_\pi(s, a)$.

The Bellman Equation corresponding to the action-value function is:

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

2.2 Markov Decision Process

A process satisfies the Markov property if the information stored in the current state sufficiently characterises the future. In the real-world, this property rarely holds true. The state should be formulated keeping in mind that the Markov property should hold as nearly as it can. Under the assumption that Markov Property holds true, the process is called a Markov decision process (MDP).

Definition 2.2.1. A Markov decision process is a 4-tuple (S,A,P_a,R_a), where

1. S is the set containing all possible states
2. A is the set containing all possible actions
3. $P_a(s, s')$ represents the probability of reaching state s' at time t+1 from state s due to the effect of the action a taken at time t

$$P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a) \tag{2.6}$$

4. $R_a(s, s')$ is the reward gained at time t+1 on moving from state s to s' due to effect of action a at time t

The ultimate aim of a Reinforcement Learning algorithm is to optimize the value function (or, action-value function), that is, find an optimal policy. The

solution follows the Bellman equation, and is widely known as the Bellman Optimality Equation as shown below:

Bellman Optimality Equation:

$$V_*(s) = \max_{a \in A} R_s^a + \gamma \sum_{S' \in S} P_{SS'}^a V_*(S') \tag{2.7}$$

$$q_*(s,a) = R_s^a + \gamma \sum_{S' \epsilon S} P_{SS'}^a \max_{a \in A} q_*(s',a) \tag{2.8}$$

The above equation however, is non-linear, and no closed form solution exists. Thus, we employ iterative methods such as Policy Iteration, Value Iteration and Deep Q-learning for solving the same.

2.3 Using Dynamic Programming to Solve The Bellman Optimality Equation

To start with, we investigate how the state-value function is computed under a given policy. Following equation 2.4 written above , we can write:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \tag{2.9}$$

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s] \tag{2.10}$$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P_{ss'}^a [R_s^a + \gamma v_\pi(s')] \tag{2.11}$$

where $\pi(a|s)$ is the probability of taking action a when the agent is in state s and following a given policy π. The subscripted expectations ($\mathbb{E}_\pi$) indicate

that the expectation is taken under the assumption of the policy π being followed. If the termination of an episode is guarenteed under a given policy, and $\gamma < 1$, then it can be proved that v_π exists and is unique.

If we know the transition probability matrix P, that is, the dynamics of the agent's environment are known, then 2.12 is a system of $|S|$ simultaneous linear equations in $|S|$ unknowns. Theoretically, the iterative solution would be the most suitable for our purposes. We consider a series of solutions, namely, v_1, v_2, v_3 and so on until v_k is approximately equal to v_{k+1}. Here, v_i is the mapping from state space to real values after the kth iteration. We start with an initial random approximation v_0 for all the states except the terminal state, which is assigned the value 0. The update rule is as follows:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} p(s', r|s, a)[r + \gamma v_k(s')] \tag{2.12}$$

Since this update is applied $\forall s \in S$ per iteration, this algorithm is called as *policy evaluation.*

Input π which is the policy to be evaluated
Initialize an array V(s) = 0, for all s ∈ S$^+$
Repeat
    Δ ← 0
    For each s ∈ S:
        v ← V (s)
        V (s) ←$\sum_a$ Π(a|s) $\sum_{s',r}$ p(s', r|s, a) [r + γV(s')]
        Δ ← max(Δ, |v − V (s)|)
until Δ < small positive number ε
Output V ≈ v_π

Figure 2.1: Iterative Policy Evaluation Algorithm [5]

Our ultimate aim is to reach an optimal policy π_* and optimal value function v_*. To do so, we extrapolate from policy evaluation, and now improve the arbitrary deterministic policy π. The question we are expected to face is, for some particular state $s \in S$, should we choose an action a such that $a! = \pi(s)$? We know the value of following the current policy π from v_π, but would it be possible to select an action that can create a better policy? So as to solve this dilemma, we select an action a from the action space which has the maximum action-value, and thereafter follow the policy π. By behaving in this way, we have the following value:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \tag{2.13}$$

Intuitively, this can be explained as picking the action a from the state S_t, and then following the same policy π starting from the state S_{t+1}. The action a that we took while in S_t can be chosen greedily as follows:

$$a = \arg\max_{a \in A} q_\pi(S_t, a) = \pi^{'}(s) \tag{2.14}$$

Thus, if the current value of being in state s, $v_\pi(s)$, is less than the action value $q(s, \pi^{'}(s))$, we then update our current policy, meaning that the agent prefers the action $\pi^{'}(s)$ when in state as, as against $\pi(s)$. Thus, if

$$q_\pi(s, \pi^{'}(s)) \geq v_\pi(s) \qquad \forall s \in S \tag{2.15}$$

then we say that the new policy $\pi^{'}$ is better than π, i.e, $\pi^{'} \geq \pi$.

Thus, the crux of the policy improvement step is 2.14. This is also called as the greedy policy. Below we discuss the proof behind why following the greedy policy will eventually lead us to an optimal policy.

Following from 2.15, we can write:

$$\mathrm{v}_\pi(s) \leq q_\pi(s, \pi^{'}(s))$$
$$\mathrm{RHS} = \mathrm{E}_{\pi^{'}}[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

$\mathrm{RHS} \leq \mathbb{E}_{\pi^{'}}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi^{'}(S_{t+1}))|S_t = s]$
$= \mathbb{E}_{\pi^{'}}[R_{t+1} + \gamma \mathbb{E}_{\pi^{'}}[R_{t+2} + \gamma v_\pi(S_{t+2})]|S_t = s]$
$= \mathbb{E}_{\pi^{'}}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2})|S_t = s]$
$\leq \mathbb{E}_{\pi^{'}}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s]$
$= v_{\pi^{'}}(s)$

Thus, $v_\pi(s) \leq v_{\pi^{'}}(s)$

So let us summarise the steps for solving the Bellman Optimality equation following the theory developed so far. First, we arbitrarily assign values to all the states, and then iteratively update these values according to the Bellman Equation 2.4. This step is known as policy evaluation. Thereafter, we improvise our policy according to the greedy policy as discussed above.

The above method is known as policy iteration. Its counterpart, called the *value-based iteration* is just a slight modification, where we iteratively update the value function greedily after every step of evaluation.

However, there is a major bottleneck in the implementation of policy and value-based iteration in real-world scenarios. Consider the equation 2.12.

If we look closely, we observe that the presence of the term $P^a_{ss'}$ implies that we must have a model of the environment, which is not possible in most real-world scenarios, and more so in the context of financial markets which is ridden with uncertainty. Thus, we now turn our attention to *model-free learning* methods, namely *Monte Carlo* learning and *Temporal Difference* learning.

2.4 Model-free Learning Methods

This class of methods rely on what is known as experience sampling of action-state pairs and rewards from the simulated interaction with the agent's environment. Therefore, learning from the agent's experience doesn't explicitly require the model of the environment. In spite of this constraint, we can still attain optimal behaviour from the captured experiences and transitions in those experiences.

In many cases, it is not complicated to generate agent's experience according to the dynamics of the environment, but it is quite difficult to explicitly obtain the transition probability matrix.

2.4.1 Monte Carlo Methods

This class of methods rely of averaging sample returns from the sampled experiences of the agent. We define Monte Carlo methods only for those experiences which are finite in nature, that is, each experience ends in a terminal state. In other words, Monte Carlo methods are defined only for *episodic* tasks. We consider that experience is split up into episodes, and assume that all the episodes terminate irrespective of the actions undertaken

by the agent. Once an episode is completed, we update the value (or action-value) functions as well as our policy.

We will follow the same pattern as we did in the previous section, that is, we will first look at evaluating a given policy, after which we will improve upon the policy. We start by recalling that a state's value function is defined as the expected discounted reward (discounted over the future time periods) starting from that state. A (first) *visit* is defined as the (first) occurrence of a state s in an episode. To estimate v_π using the *first-visit* MC method, we collect the returns and average them out following first visits to s after every episode. In the *every-visit* MC method, we average out the returns following every visit to s. To arrive at the update equation, consider the following operations,

$$\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) \tag{2.16}$$

Thus, the update equation after every episode can be written as follows:

$$N(S_t) = N(S_t) + 1 \tag{2.17}$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)) \tag{2.18}$$

The value of α is set to $1/N(S_t)$ in case of stationary problems, but in almost all financial problems, we will encounter non-stationary problems because of which α, also known as the learning rate, is different. We summarise the MC prediction method in the pseudo code given in figure 2.2.

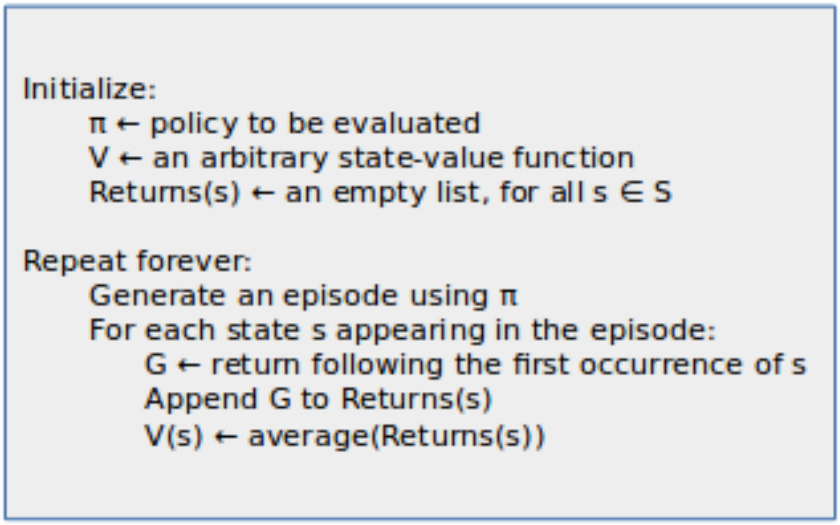

Figure 2.2: MC Prediction for estimating v_π [5]

We now use the same Bellman optimality equations for updating our policy, that is,

$$\pi^{'}(s) = \arg\max_{a \in A} (R_s^a + \sum_{s' \in S} P_{ss'}^a V(s')) \tag{2.19}$$

However, in its raw form the above bellman optimality equation contains the term $P_{ss'}^a$ which contradicts our consideration of a model-free environment. To tackle this problem, we modify our MC method so as to evaluate the action-value pairs instead of the value functions. The update equation for the action-value pair (S_t, A_t) and the policy improvement equation is:

$$\pi^{'}(s) = \arg\max_{a \in A} Q(s, a) \tag{2.20}$$

To summarise, the pseudo-code for Monte Carlo control is as given below:

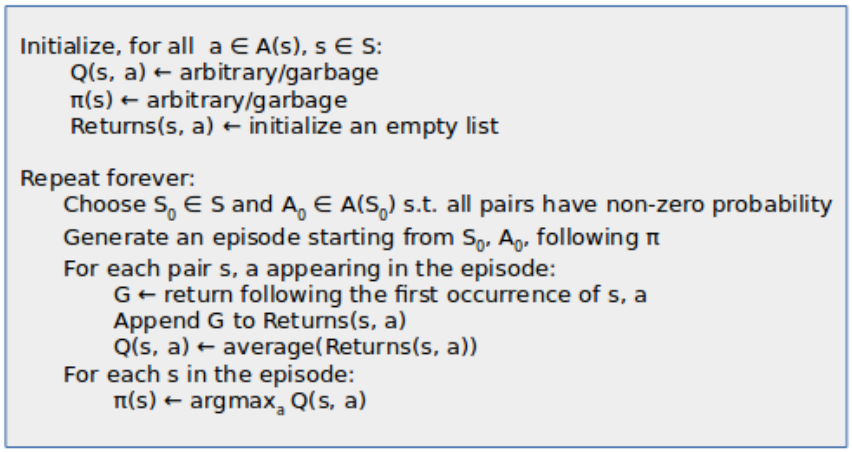

Figure 2.3: Monte Carlo Control [5]

2.4.2 Temporal Difference Learning

The class of methods under this category are at the intersection of MC learning methods as well as the Dynamic Programming methods that we saw in the previous section. Like Monte Carlo methods, TD learning can learn directly without the assumption of the environment's dynamics and like Dynamic Programming, it doesn't need to sample the episode entirely to update the value (or action-value) function. This property is also known as *bootstrapping*.

Let us understand mathematically the difference between Temporal Difference learning and Monte Carlo learning through the equations written below:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t)) \tag{2.21}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.22}$$

Following from the above equations we observe that the return G_t in the MC update equation 2.21 is the reward gained from the beginning to the

termination of an episode whereas the TD method update equation 2.22 uses an already existing estimate $\alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ for the updation. In other words, the *target* for MC is G_t whereas it is $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ for TD learning. Thus, the only major differences between the Monte Carlo and Temporal Difference learning methods are the update equations in the two algorithms and the timing of the update. The update occurs periodically through every episode in case of Temporal Difference learning whereas it occurs only at the end of an episode in MC learning.

Previously we discussed the greedy policy wherein the agent greedily picks up the action which gives the maximum cumulative return. However, one major drawback in following this policy is that it is a very real possibility that the agent might not discover the entire action space which is why the attainment of an optimal solution is not guaranteed. Thus, to explore the entirety of our action space, we now follow a modified policy known as the **ϵ-greedy policy** which is defined as follows:

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon, & \text{if } a^* = arg\max_{a \epsilon A} Q(s,a) \\ \frac{\epsilon}{m}, & \text{otherwise} \end{cases}$$

In simple words, this policy generates a uniform random number between $[0, 1]$, compares its value to ϵ, and if greater than ϵ, acts greedily otherwise picks an action randomly from the action space.

Using the theory developed so far, we can now put forth the first major algorithm of Reinforcement Learning, namely, the **Q-learning** algorithm.

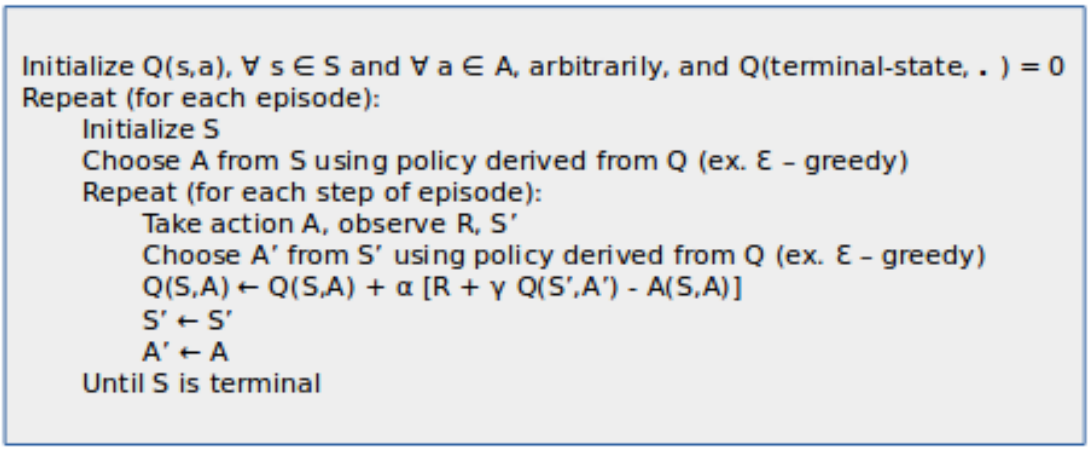

Figure 2.4: Q-learning Algorithm [5]

2.5 Value Function Approximation

In our policy evaluation step, we updated the action-value function for all ordered-pairs (S_t, A_t). In the financial environment, the states generally contain information such as the current portfolio value, the return of the last few days' trades, or some properties of the spot price of the stock it is representing. Similarly, the action space of the agent consists of specifying the number of shares of a stock the agent is intending to buy/sell. Thus, we can observe that this would lead to a huge state-action space. Updating the Q-values for every ordered pair (S_t, A_t) would be a computationally infeasible task, which is why we now turn to **Value Function Approximation**.

The idea behind the approximation of the agent's value function is quite simple. State-action pairs numerically close to each other must have their action-value functions close to each other. Thus, we require a black-box which inputs a state-action ordered pair and outputs the approximate action value function corresponding to that pair. Some common examples of this black-box function approximator are linear combination of weights, decision

trees or neural networks.

Thus, our main objective in this section is to minimise the error, or, the *mean-squared error* (MSE) between the approximated values $v_w^{'}, q_w^{'}$ and the actual values. Here, the subscript w indicates the weights of the black-box approximator that we will improvise so as to get the least MSE. Let $J(w)$ be the loss function that measures the favourability of our weights w.

$$J(w) = \mathbb{E}_\pi[(v_\pi(s) - v^{'}(s, w))^2] \tag{2.23}$$

We minimise the MSE using *Stochastic Gradient Descent* (SGD), one of the most commonly used optimisation algorithms. The update equation for our weight matrix in SGD is:

$$\Delta_w = -\alpha/2 * \nabla_w(J(w)) \tag{2.24}$$

Here, $v_\pi(s)$ is called the supervisor which acts a feedback to our approximator. But how do we get hold of this supervisor? Just like in the case of TD-learning, we will use bootstrapping in order to generate the supervisor for our approximation. Consider an episode as follows: $< S_1, R_2 + \gamma v^{'}(S_2, w) >$ $, < S_2, R_3 + \gamma v^{'}(S_3, w) >$ and so on. Thus, similar to TD-learning, but after the completion of the episode, we update our weight matrix by the following update equation:

$$\Delta w = \alpha(R_{t+1} + \gamma q^{'}(S_{t+1}, A_{t+1}, w) - q^{'}(S_t, A_t, w))\nabla_w(q^{'}(S_t, A_t, w)) \tag{2.25}$$

We will utilise a neural network model to approximate our value (or action-value) function as of now. The neural network architecture will be such that

we will only input the state s and it will output a set of n values, each corresponding to an action-value for every action in our discrete action space A. In case the action space is too large or a continuous one, we sample discrete values from the space uniformly. The agent will then take the action based on its policy, either ϵ-greedy or greedy. In the below figure, input 1 to input n are the features of a state s and output 1 to output n are the action-values for each action.

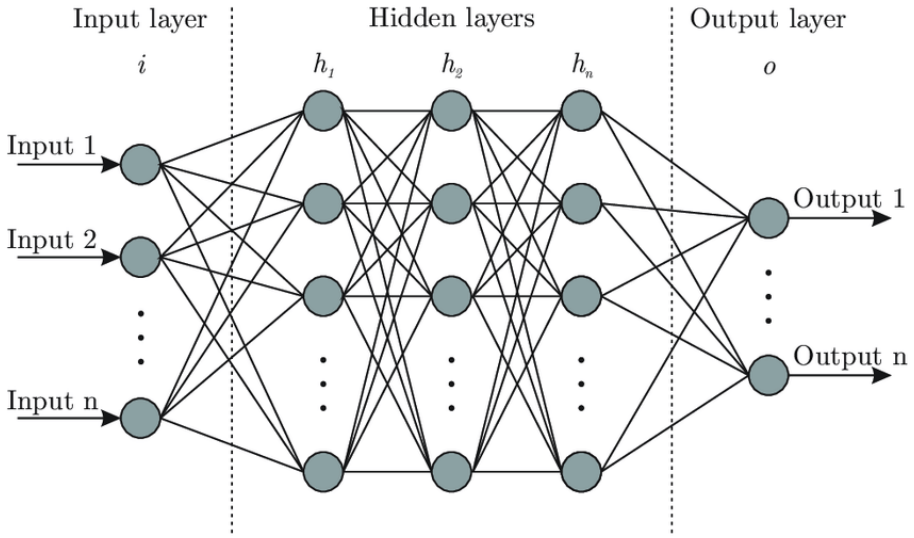

Figure 2.5: Neural Network Architecture [1]

2.6 Deep Q-Learning Algorithm

All the theory developed so far ultimately boils down to this algorithm, which was one of the major breakthroughs in the field of Reinforcement Learning. This algorithm was patented by Google DeepMind. It is an application of Reinforcement Learning to Deep Learning, titled as "Deep Reinforcement Learning" or "Deep Q-learning", and is able to play Atari at expert human levels.

All the steps of the algorithm have been explained in parts in the above sections. We briefly outline the steps below:

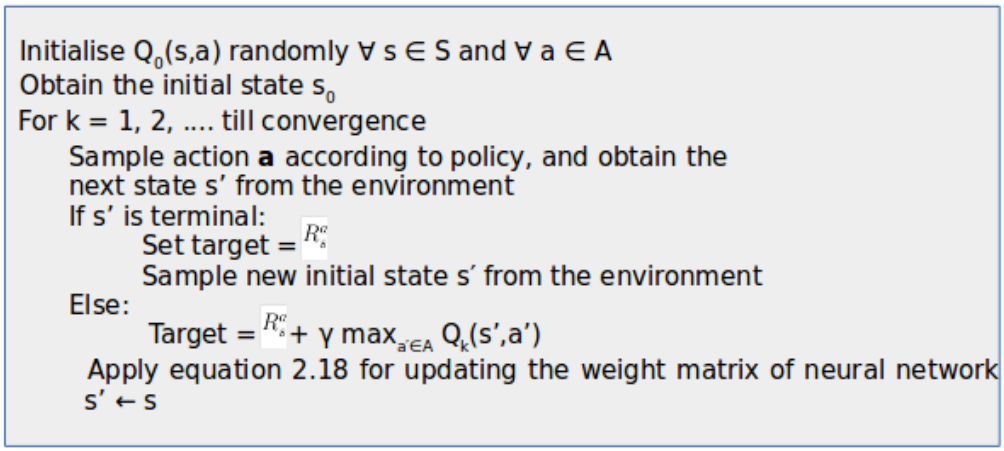

Figure 2.6: Deep Q-Learning Algorithm [5]

Chapter 3

Experimental Results on Single Stock discrete action Trading Model

We implement the Deep Q-learning algorithm on real-world datasets, namely the stocks SBI and Reliance Industries Private Limited. 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 state space of the agent is a tuple containing the information of last seven days spot price of the considered stock and also the current value of the agent's portfolio, and the rewards are simply the profits gained after closing a long position.

We present below our results obtained on the two stocks mentioned above. We used training data of daily closing prices from 5th October 2016 to 5th October 2018, and tested our agent on data from 6th October 2018 to 6th October 2019. The profit earned in case of Reliance Industries was Rs. 668.79 and that for SBI was Rs. 1104.23.

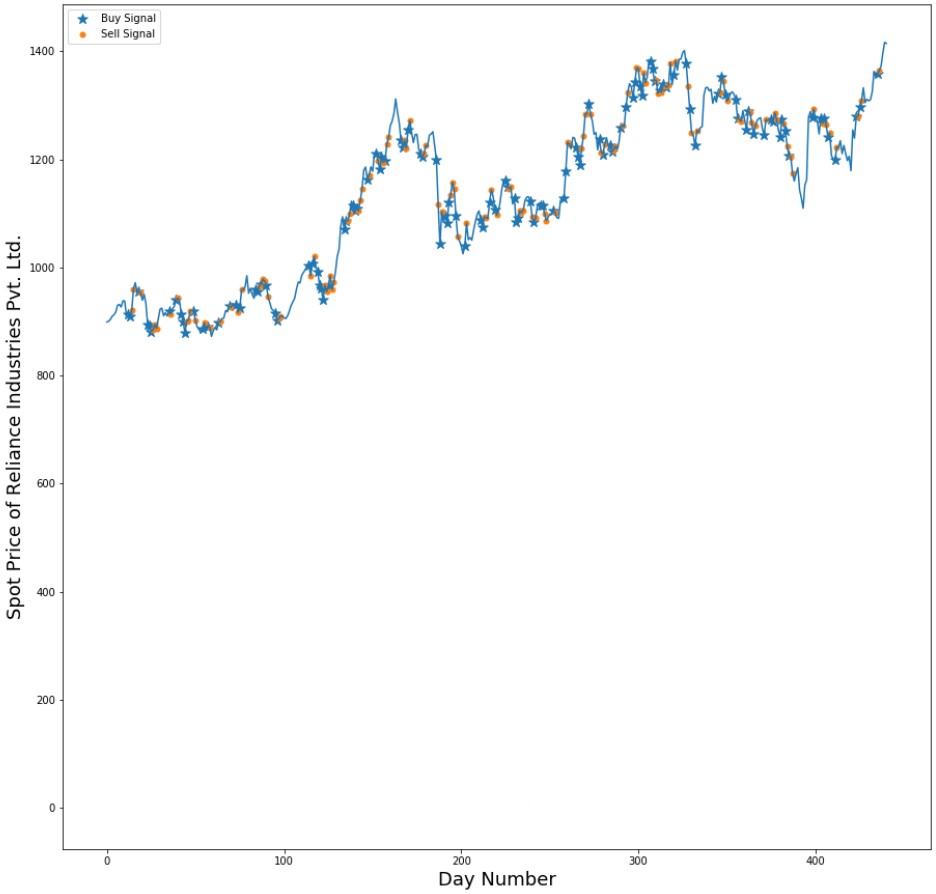

Figure 3.1: Results for the Reliance spot price, profit earned = Rs. 1104.23

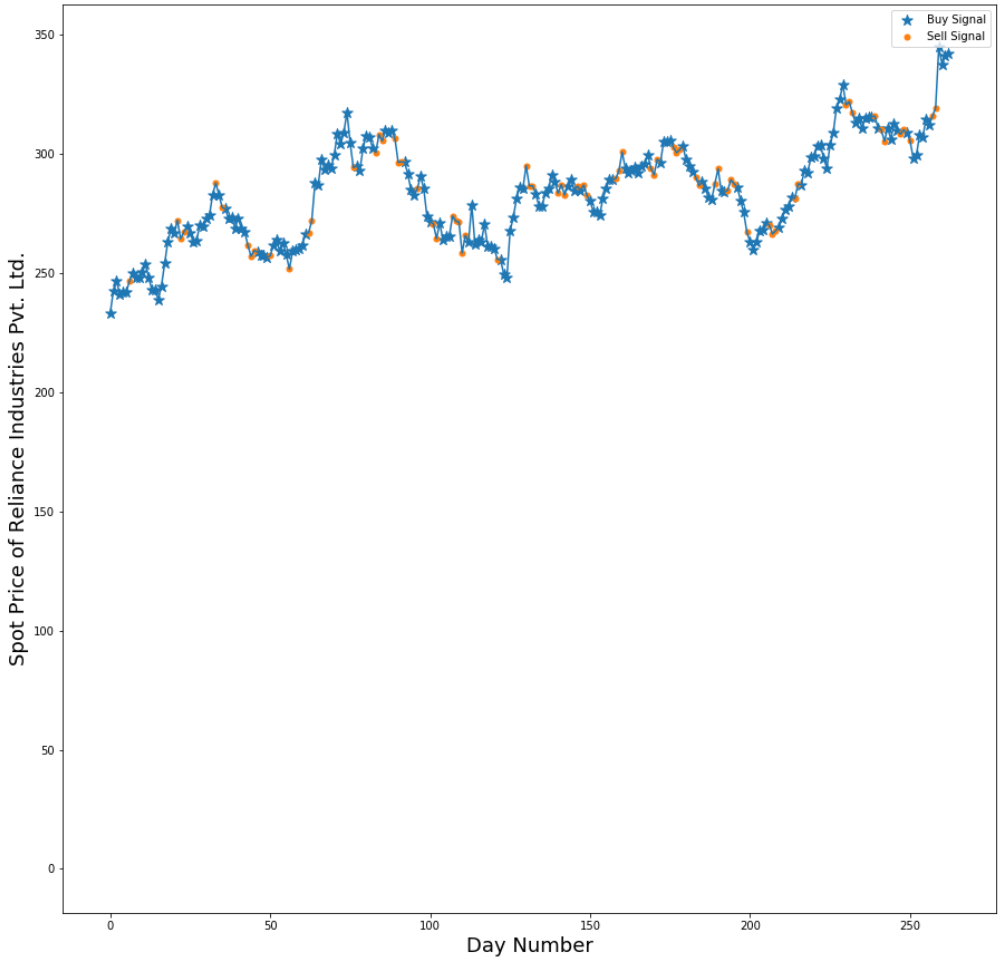

Figure 3.2: Results for the SBI spot price, profit earned = Rs. 668.79

Chapter 4

Increasing Complexity of the Trading Model

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

Further, we described the main elements of Reinforcement Learning, viz., Agent, Environment, Reward, Policy and Value. We stated the mathematical concepts that underpins the working of Reinforcement Learning, viz., 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 optimisation. The real world trading environment does not have access to deterministic policies, hence we explored model-free learning methods like Monte Carlo Prediction and Monte-Carlo Control. We also explored methods like Temporal Difference Learning to enhance the stability of Monte Carlo methods. 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 and Reliance Industries Private Limited stocks. 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.

1. Our goal will be to extend the model to take continuous actions and optimise a multi-stock portfolio using some quantification of risk associated (or utility) with the investment.

2. we explore the textbook methods for continuous control in Reinforcement Learning like Actor-Critic methodology and Deep Deterministic Policy Gradient (DDPG).

3. Finally, we implement the DDPG model to take continuous actions on a multi-stock portfolio. Additionally, we also incorporate the utility of wealth in our model using it as a feature of the state space.

4. We test the model on Indian stock market data on portfolios consisting of several combinations like industry specific and diversified. We

measure the relative reward of portfolio with respect to related market index.

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

Chapter 5

Policy Based Learning

The models explored so far, viz., Q-Learning, Deep Q-Networks (DQN) take upon the task of reward optimisation by approximating the value function or action-value function. The optimised policy is formulated using ϵ-greedy selection over the action-value function. This methodology is known as **Value-Based Learning**.

We will now explore methods that directly optimize the parameterized policy. These methods does not involve learning the value functions. As opposed to learning an implicit policy (via ϵ-greedy selection), **Policy-Based Learning** learns the exact deterministic policy. The advantage of these methods above value-based learning is the efficient space use which is useful while working with continuous action spaces and high-dimensional state spaces.

The methodology known as **Actor-Critic** involves both learning of value functions and policy. These methods form the intersection of both Value-Based and Policy-Based Learning.

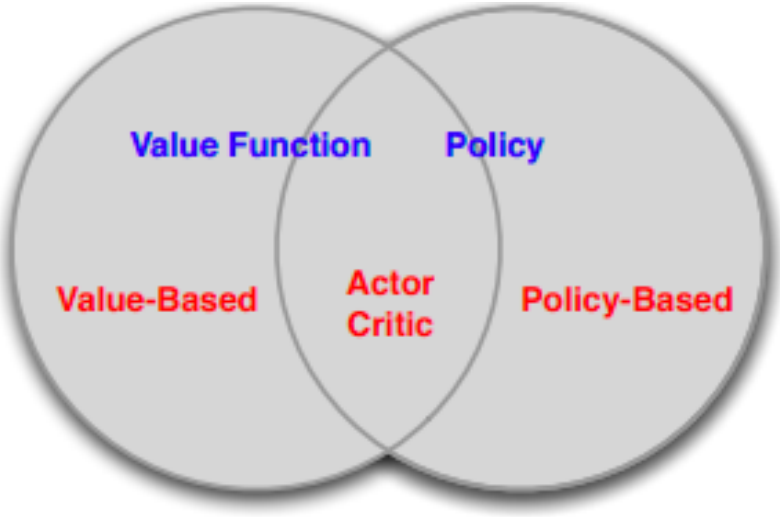

Figure 5.1: Learning Methods [4]

5.1 Policy Gradient

Value Based Learning parametrize the value function using a parameter θ and approximate the optimized values by Q-Learning methods. Similarly Policy Based Learning parametrize the policy and our goal is to formulate an objective function for policy optimisation. Given the parameterized policy, our goal is to find best θ. We will start with an arbitrary policy like Softmax Policy and employ methods that converge to the optimized policy.

$$V_\theta(s) = V^\pi(s) \tag{5.1}$$

$$Q_\theta(s,a) = Q^\pi(s,a) \tag{5.2}$$

$$\Pi_\theta = \mathbf{P}[\mathbf{a} \mid \mathbf{s}, \theta] \tag{5.3}$$

Parameterized Policy

5.1.1 Optimisation of Policy Objective Functions

The reward optimization function for Policy Based Learning is given by -

$$J_{avR}(\theta) = \sum d^{\pi_\theta} \sum \pi_\theta(s,a) R_s^a \tag{5.4}$$

here, d^{π_θ} is the **stationary distribution** for π_θ [4]

Policy Gradient algorithm will converge to maximum in J(θ) by changing θ in direction of gradient. Here change in θ is given by -

$$\Delta\theta = \alpha \nabla_\theta J(\theta) \tag{5.5}$$

here, α is the learning rate

Using Likelihood Ratios, we form an analytic function in order to obtain the policy gradient.[4]

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \tag{5.6}$$

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \tag{5.7}$$

here, ∇_θ[log(π_θ(s,a))] is the score function

The optimisation function[4] J(θ) can be restated as -

$$J(\theta) = \sum_{s \epsilon S} d(s) \sum_{a \varepsilon A} \pi_\theta(s,a) R_s^a \tag{5.8}$$

$$\nabla_\theta J(\theta) = \sum_{s \epsilon S} d(s) \sum_{a \varepsilon A} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) R_s^a \tag{5.9}$$

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{5.10}$$

5.1.2 REINFORCE Algorithm

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

Initialize θ arbitrarily
For each episode $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ 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 5.2: REINFORCE Algorithm [4]

5.2 Variance Reduction using Actor-Critic

Monte Carlo Policy Gradient (REINFORCE) 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.

5.2.1 Critic Reducing Variance

Actor-Critic method keeps two sets of parameters. We also make use of value function approximations like Temporal-Difference Leaning and TD(λ) to estimate the action-value. The two elements of this methodology are -

1. **Critic** - Computes action-value function via value function approximation and updates parameter w

2. **Actor** - Takes cue from critic and changes the policy parameter θ accordingly

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

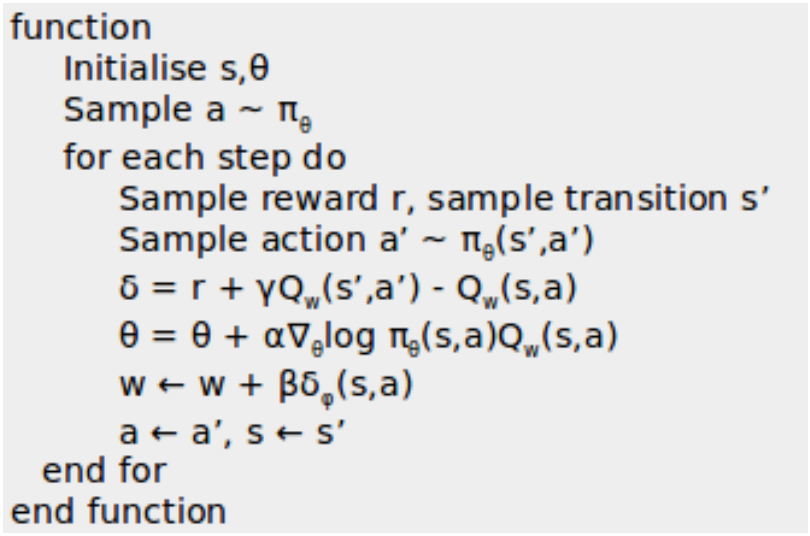

Figure 5.3: Action-Value Policy Gradient Algorithm [4]

There is an implicit bias in the above algorithm because we the sampled trajectories amounting to zero reward will not update the policy parameter towards choosing good actions. To handle this bias, we subtract a baseline function from the action-value function which gives an **Advantage Function** to use in place of Action-Value Function. An example of such an advantage function is -

$$A(s,a) = Q_w(s,a) - V(s) \tag{5.11}$$

Chapter 6

Deep Deterministic Policy Gradient

6.1 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. -

$$a^*(s) = \arg\max_{a \in A} Q^*(s, a) \tag{6.1}$$

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

6.2 Learning

6.2.1 Q-Learning

The Bellman Equation for optimising the action-value function is [3] -

$$Q^*(s,a) = \mathbb{E}[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{6.2}$$

We introduce a performance measure for the Q-Learning model called **Mean-Squared Bellman Error (MSBE)** [3]

$$L(\phi, D) = \mathbb{E}_{(s,a,r,s')\epsilon D}[Q_\phi(s,a) - Q^*(s,a)] \tag{6.3}$$

$$L(\phi, D) = \mathbb{E}_{(s,a,r,s')\epsilon D}[Q_\phi(s,a) - \mathbb{E}[r(s,a) + \gamma \max_{a'} Q^*(s',a')]] \tag{6.4}$$

here, Q_ϕ(s,a) is a target optimised network.
(more details on target networks in next subsection)

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 large number of experiences, then algorithm becomes computationally inefficient. On the other hand, if the set D stores only recent experiences, then there is risk of over-fitting. Hence practical trade-offs are implemented while applying the algorithm depending upon the problem and efficiency of system. It is useful to observe that since DDPG learns policy also, the set of experiences are obtained from outdated policy but Q-Learning part of the algorithm does not care about the origin of transition tuples.

6.2.2 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 optimise the action-value function Q_ϕ(s,a). The objective function can be formulated as -

$$J(\theta) = \mathbb{E}_{s\epsilon D}[Q_\phi(s, \mu_\theta(s))] \tag{6.5}$$

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_ϕ(s,a) is differentiable with respect to action. This allows us to use gradient descent method to optimize the objective function J(θ). The gradient of objective function is formulated as -

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{s\epsilon D}[Q_\phi(s, \mu_\theta(s))] \tag{6.6}$$

As explained earlier, DDPG employs replay buffers to sample experiences for Q-Learning, 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 euations are given by -

$$\nabla_\theta J(\theta) = \nabla_\theta \frac{1}{|B|} \sum_{s\epsilon B} Q_\phi(s, \mu_\theta(s)) \tag{6.7}$$

$$\theta_{\text{new}} = \theta_{\text{old}} + \nabla_\theta \frac{1}{|B|} \sum_{s\epsilon B} Q_\phi(s, \mu_\theta(s)) \tag{6.8}$$

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.

6.3 Network Architecture

DDPG is an off-policy algorithm. As shown in Figure 6.1, DDPG learns both Value and Policy. This requires introduction of additional learning networks over and above the networks used in Policy Gradient.

6.3.1 Target Network

DDPG uses four neural networks. Two are implicit networks for Value and Policy Learning. Two are target networks for optimised value and policy. [10]

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 -

$$r(s,a) + \gamma \max_{a'} Q^*(s',a') \tag{6.9}$$

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 [3].

$$\phi_{\text{target}} \leftarrow \rho\phi_{\text{target}} + (1-\rho)\phi \tag{6.10}$$

6.3.2 Noise Induced Exploration

DDPG learns the policy in off-policy manner. Since the action space is continuous and the learnt policies are deterministic, the agent may not explore the actions properly while learning. Lillicrap et al.[8] propose to introduce **Ornstein–Uhlenbeck (OU)** noise to the action selection step of the algorithm. This provides a balance between exploitation of known policy and unexplored actions. The selection step is modified as -

While transitioning, select action a s.t.

$$a = \mu_\theta(s) + \epsilon \tag{6.11}$$

here, ϵ is sampled from Ornstein–Uhlenbeck (OU) process.

6.4 DDPG Pseudo-Code [3]

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

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

      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 6.1: Deep Deterministic Policy Gradient Algorithm [3]

Chapter 7

Experimental Results of the updated Trading Model

To simulate the environment of a trader investing in the stock market, we created a Python *class trade_env()* where the agent (investor) can buy or sell stocks to create his/her own portfolio. Simulating the entire stock market having the presence of transaction costs and non-zero rates of interest was difficult, hence we downloaded historical data of over 10 stocks from various sectors from *Nifty* as well as *Sensex* market indices. The stocks taken from the *Nifty* market index were Asian Paints, Godrej, Hero MotoCorp, IndusInd Bank, Infosys, Reliance, SBI, ITC and Tata Steel, whereas from the *Sensex* market index we took HDFC, Hindustan Unilever, ITC, L&T, M&M, ONGC, Reliance, SBI and Tata Steel. The portfolio from the *Nifty* index were trained from 2007 to 2016, and tested on 2017-2020, whereas the portfolio from *Sensex* was trained from a 1997-2016 and tested on 2016-2020. We experimented with the portfolio by taking all 10 stocks at once in it, or taking sector wise stocks in our portfolio our a combination of one or more stocks from multiple sectors.

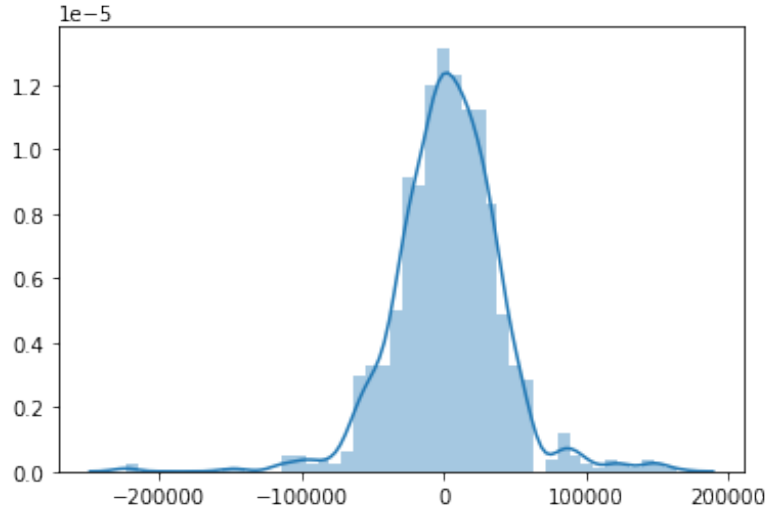

Figure 7.1: Probability Distribution of Instantaneous Relative Rewards for a portfolio containing all Nifty Stocks

For gauging the performance of our algorithm, we execute the optimal trading policy trained on the training set by testing it on the test data set, and comparing its relative performance by investing all our proceeds into the market index (either *Nifty* or *Sensex*). Following are the graphs depicting the distribution of rewards at each time step of our algorithm vs the episode number, and also the simulated probability density function of the relative rewards at each time step.

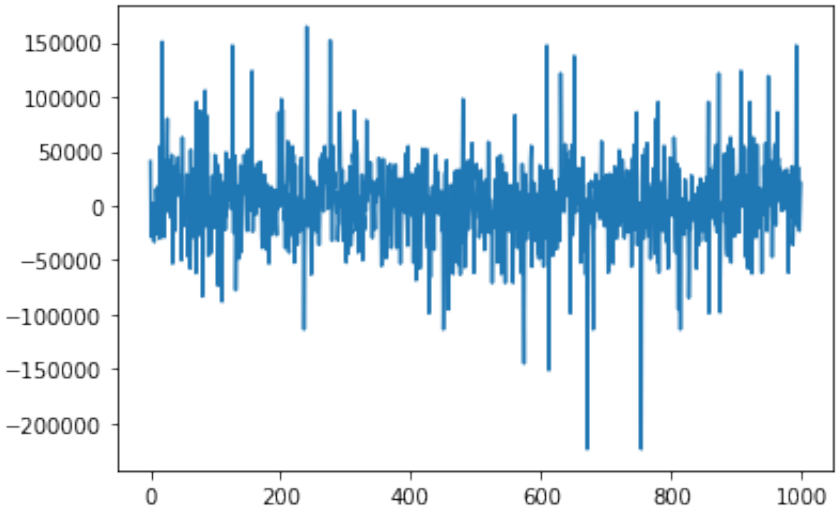

Figure 7.2: Plot of Instantaneous Relative Rewards vs Episode Number for a portfolio containing all Nifty Stocks

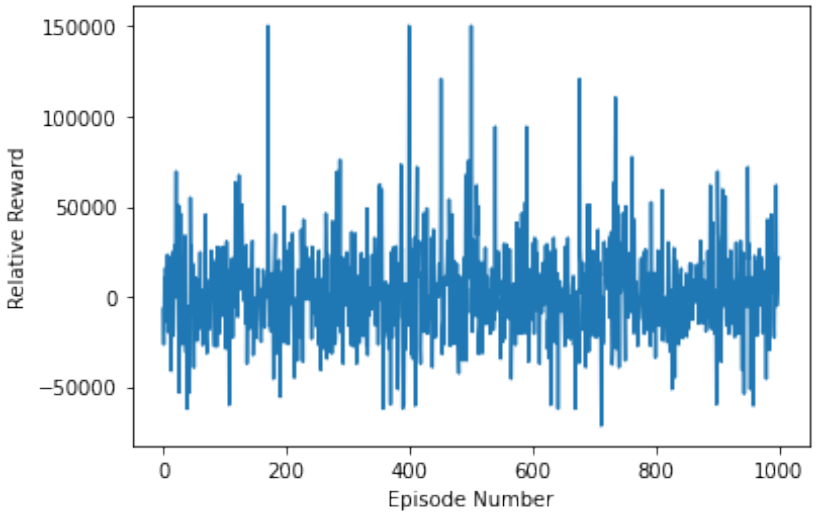

Figure 7.3: Portfolio of stocks Asian Paints + Hero MotoCorp + Tata Steel

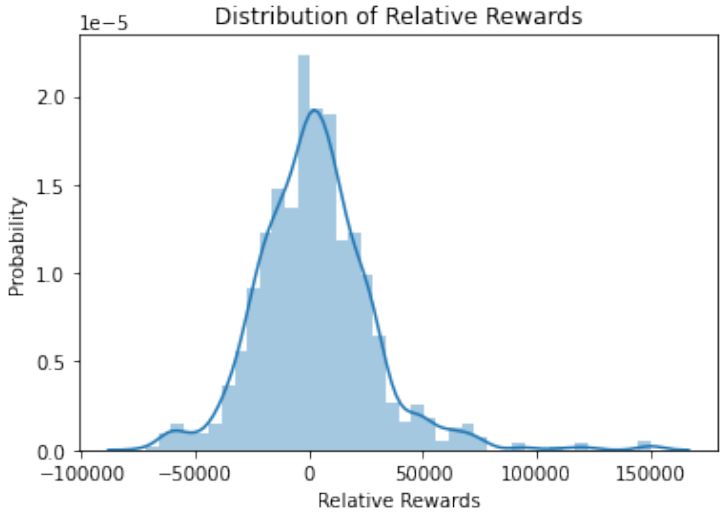

Figure 7.4: Portfolio of stocks Asian Paints, Hero MotoCorp, Tata Steel

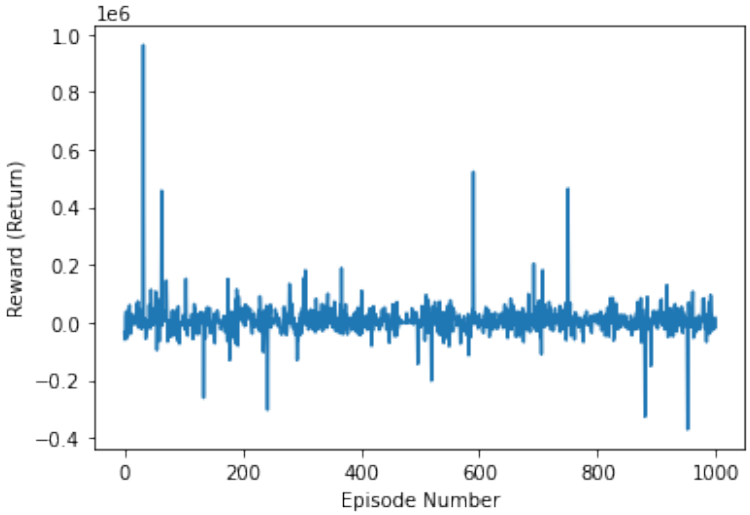

Figure 7.5: Portfolio of stocks from ITC, Godrej and Infosys

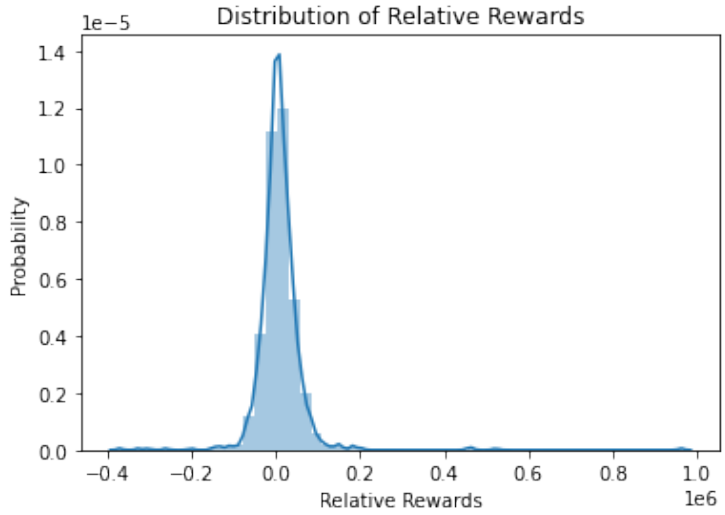

Figure 7.6: Portfolio of stocks from ITC, Godrej and Infosys

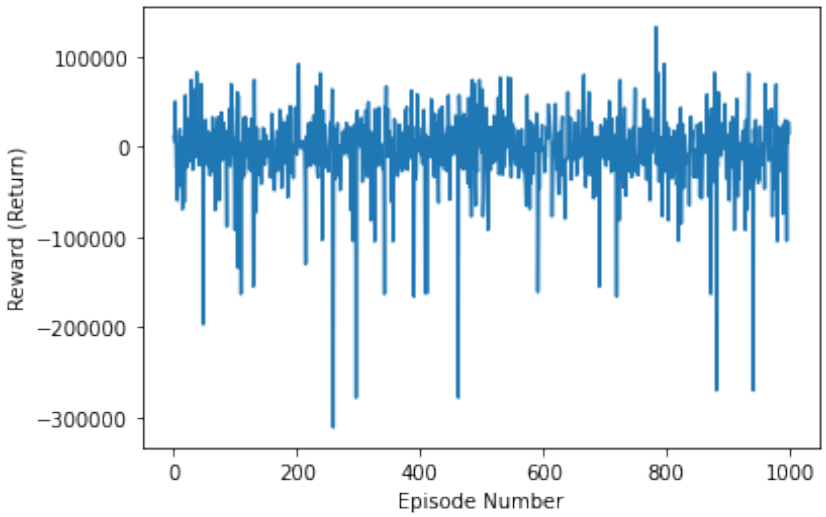

Figure 7.7: Portfolio of stocks from IndusInd Bank and SBI

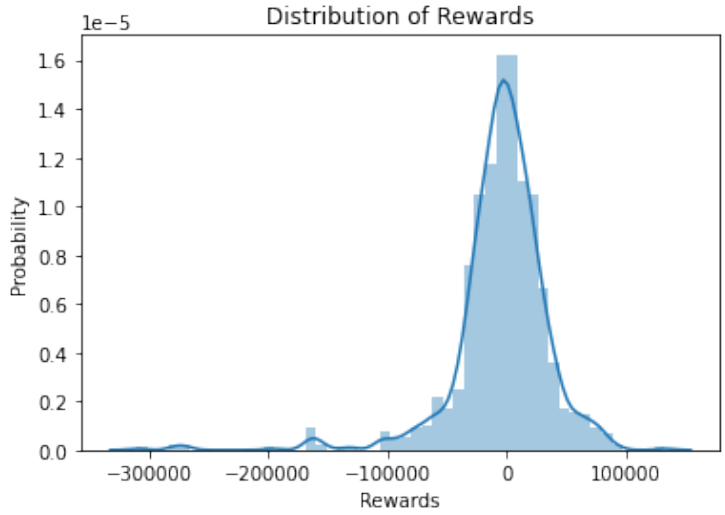

Figure 7.8: Portfolio of stocks from IndusInd Bank and SBI

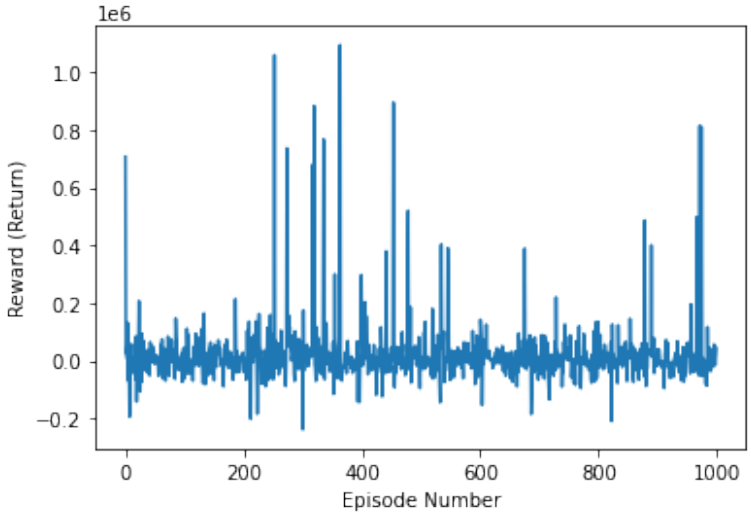

Figure 7.9: Portfolio of all stocks from Sensex

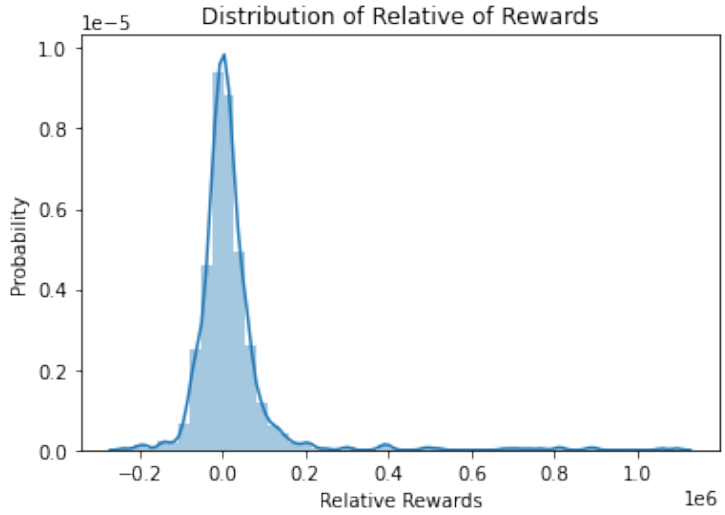

Figure 7.10: Portfolio of all stocks from Sensex

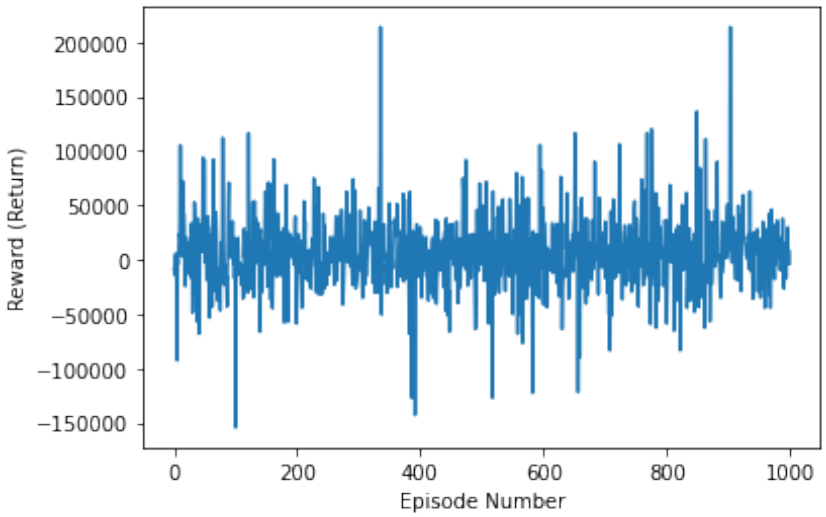

Figure 7.11: Sensex portfolio of stocks L&T, M&M, ONGC and Tata Steel

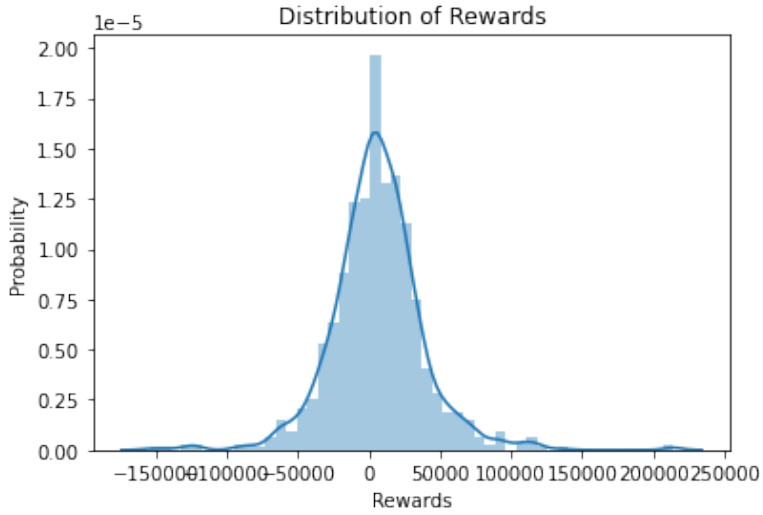

Figure 7.12: Sensex portfolio of stocks L&T, M&M, ONGC and Tata Steel

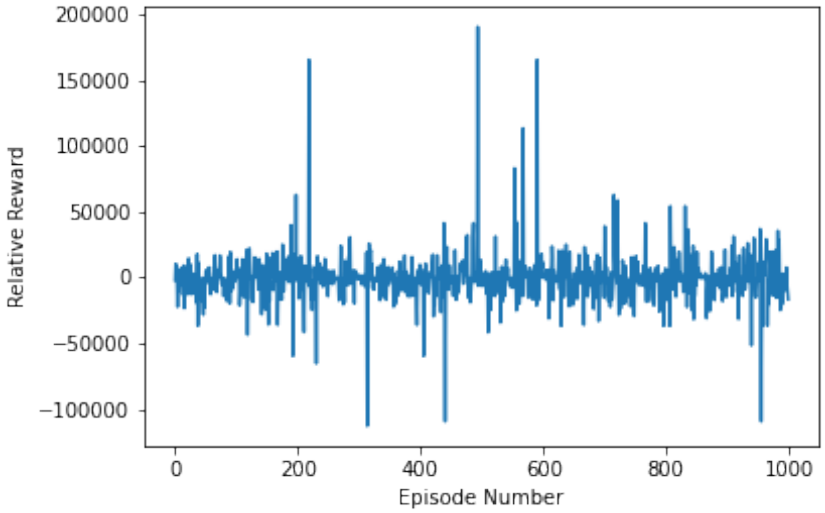

Figure 7.13: Sensex portfolio of stocks Hindustan Unilever and ITC

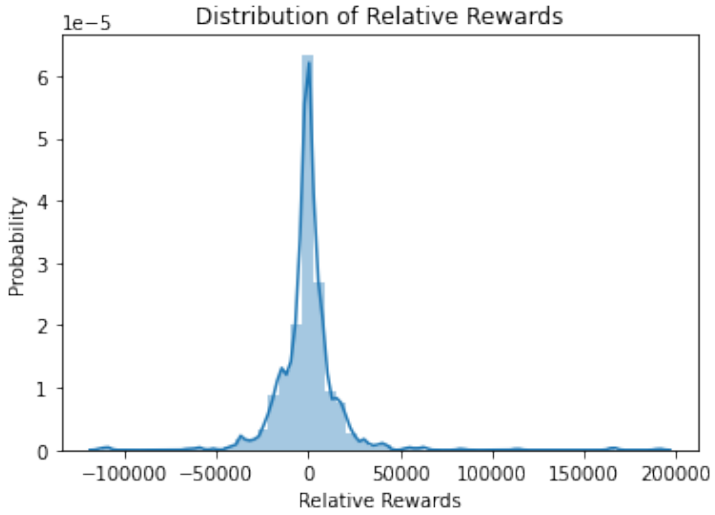

Figure 7.14: Sensex portfolio of stocks Hindustan Unilever and ITC

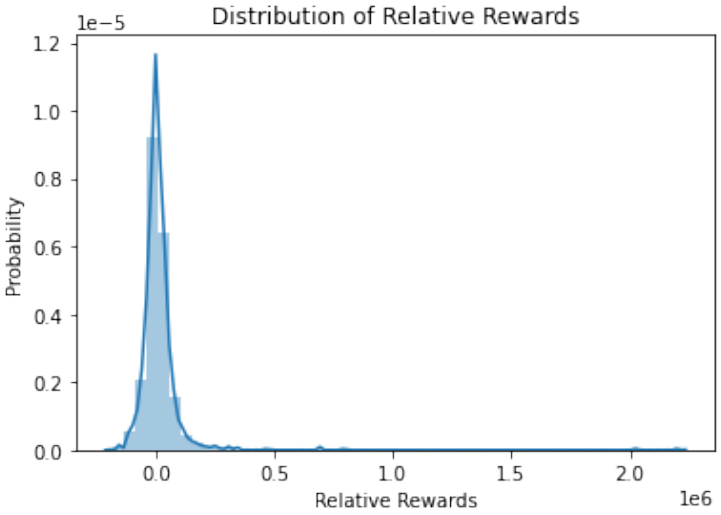

Figure 7.15: Sensex portfolio of stocks SBI and HDFC

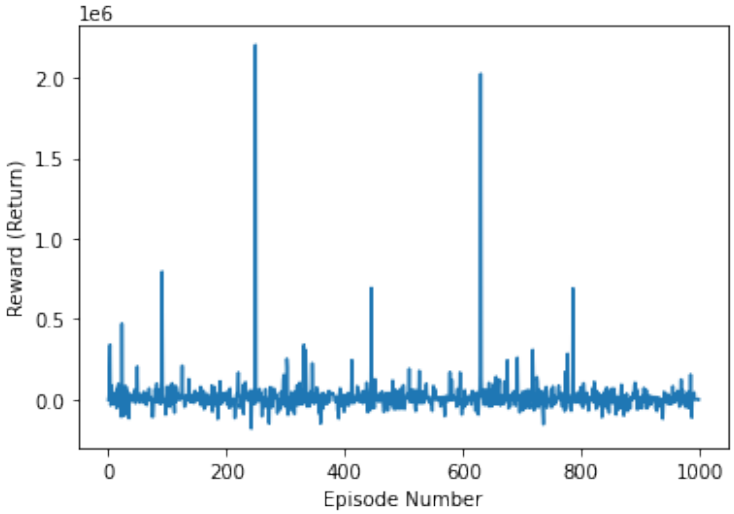

Figure 7.16: Sensex portfolio of stocks SBI and HDFC

Portfolio	Index	Median	Mean
Nifty All	Nifty	3142.829949	3122.18035
Asian Paints + Hero + Tatasteel	Nifty	2048.004897	2746.107104
SBI + IndusInd Bank	Nifty	-2434.412406	-4450.067109
ITC + Godrej + Infosys	Nifty	4686.145282	8378.635466
Sensex All	Sensex	4044.268488	19143.86109
(L&T + M&M + ONGC + Reliance + Tata Steel	Sensex	5985.812378	5022.033708
SBI + HDFC	Sensex	15293.67423	3441.494371
HINDUNILVR + ITC	Sensex	1526.345719	1800.764506

Figure 7.17: Summary of Instantaneous Relative Rewards (in Rs.)

Chapter 8

Conclusion and Potential Enhancements

Following our earlier endeavours 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 actually DQN with continuous action spaces. By parameterising the policy as well along with value function approximation, the algorithm is able to 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 modify the reward function from the absolute returns to the risk-adjusted returns which may drastically improve the standard deviation (risk) of our returns. However, due to time constraints, that has been left as future scope.

Bibliography

[1] Facundo Bre, Juan Gimenez, and Víctor Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158, 11 2017.

[2] Chien-Yi Huang. Financial trading as a game: A deep reinforcement learning approach. page arXiv:1807.02787v1, 2018.

[3] OpenAI. *Deep Deterministic Policy Gradient*. `https://spinningup.openai.com/en/latest/algorithms/ddpg.html`.

[4] David Silver. *UCL Course on Reinforcement Learning*. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html` `https://bit.ly/2WHS8pH`.

[5] Richard S.Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second Edition. The MIT Press, 2014.

[6] Wikipedia The Free Encyclopedia. *Markov Decision Process*. `https://en.wikipedia.org/wiki/Markov_decision_process`.

[7] Wikipedia The Free Encyclopedia. *Q-learning Algorithm*. `https://en.wikipedia.org/wiki/Q-learning`.

[8] Alexander Pritzel Nicolas Heess Tom Erez Yuval Tassa David Silver Timothy P. Lillicrap, Jonathan J. Hunt and Daan Wierstra. Continuous control with deep reinforcement learning. page arXiv:1509.02971v6, 2019.

[9] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. `https://bit.ly/2r3vz35`.

[10] Chris Yoon. *Deep Deterministic Policy Gradients Explained*. `https://bit.ly/2yni7v5`.