Hochschule Stralsund
Faculty of Electrical Engineering and Computer Science

Master's Thesis

Applications of Deep Reinforcement Learning on Control System Environments

Author: Rituraj Singh
Matr.Num - 19539

Supervisor: Prof. Dr. André Grüning
Mathematik und Künstliche Intelligenz

Co-Supervisor: Prof. Dr. Christian Bunse
Elektrotechnik und Informatik

March 6, 2022

Abstract

In past years Reinforcement Learning is finding applications in different branches of science and technology, whether it is health,finance, autonomous vehicles or complex control systems. Such complex problem can be solved by combining the rudimentary Reinforcement Learning methods with powerful Deep Neural Networks giving rise to Deep Reinforcement Learning solutions. Substantial amount of research has been and is still being conducted in this area to exploit the potentials of the these learning methods.Due to which the researchers encounter challenges in determining which methods to employ in different areas of application. This Thesis will apply two different DRL methods on control system environments and evaluate the performance of these DRL methods. We implement two different agent based on two DRL algorithms namely Deep-Q learning and Reinforce Policy Gradient methods to solve two different control system environments (Cartpole-v1 and LunarLander-v2) that require sequential decision making. We evaluate the performance and the behaviour of the methods through these implemented agents and characterize the suitability for these control system problems. We expect that the finding from our research prove worthy in making comparison across these methods, offering particulars needed about these DRL methods.

Acknowledgements

First of all, I am thankful to my Supervisor *Prof. Dr. André Grüning* for providing me this topic and for motivating and guiding me througout my research. I am also grateful to my Co-Supervisor *Prof. Dr. Christian Bunse* who helped me in taking care of the presentation aspects for the topic. It was exciting to learn and work in area of Deep Reinforcement Learning.

Finally, I also express my gratitude to my friends and my family, who supported me financially and mentally throughout my studies and in writing my thesis.

Declaration

I hereby declare that this thesis is solely my own work and I have cited all external sources used.

Stralsund, March 6, 2022

Rituraj Singh

CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

AI Artificial intelligence. 4, 5, 7, 35

DNN Deep Neural Network. I, 14–19, 27–29, 47, 59, 65, 72, 76, 77

DQN Deep Q-Network. 28, 46, 47, 49–51, 63

MDP Markov Decision Process. 19, 21–27

ML Machine Learning. 5, 7

RL Reinforcement Learning. I, 6–10, 19, 28, 29, 33, 77

w.r.t with respect to. 29

CHAPTER 1

INTRODUCTION

This chapter will give an introduction to the research conducted and will present the context of the work in Section 1.1, followed by the problem statement (Section 1.3) which instantiates the idea behind the Thesis. Section 1.4 describes the research-method followed. The goals of the research are clarified in the Section 1.6, and the Chapter winds-up with Section 1.7 where we outline the structure of the thesis.

> Code snippets explained in the thesis use Python Programming language.So intermediate level of cognizance in Python programming language is a pre-requisite

1.1 Context of the work

In this section we lay down the context for the thesis by giving an idea about the history, the overview of functional differences and the relevance of Machine and Reinforcement Learning in the current world.

1.1.1 Historical Inspiration

From the time of the *Industrial Revolution(1750-1850)* [18], there has been a demand to automate manual solutions which led to the *Machine Revolution(1870-1940)* [18] replacing some of the work done physically by Machines. Due to the Machine Revolution masses of productivity increase were noticed worldwide and this fed to a new stage called *Information Age(1950-now)* [18]. Information Age was all about automating mental solutions a classical example would be performing difficult mental calculations with help of computers, which can automate the process of calculation in a way which is very fast and very precise and hence replace the slower version of mental calculation performed by Humans. As a matter of fact, Information Age also led to a lot of productivity increase. But these phases had something in common for which we had to invent a new solution every time in order to either automate physical or the mental problems. Contrary to these phases there is another domain that is already ongoing and that allows machines to find solutions by themselves also known by the name of **Artificial intelligence**. Artificial Intelligence has huge potential upside cause if machines can determine solutions themselves then it takes away the responsibilities of humans to find solutions for a specific problem. And humans would just have to specify a problem with the goals for the problem and let the machine figure out how to solve that problem by itself. In order for this to happen the Machine has to be ARTIFICIALLY made extensively intelligent so that it can autonomously learn to make decisions for the particular problem by itself. Nonetheless, having a huge potential upside and currently being an in-demand field of research, Artificial Intelligence has been on top of the investigation for many decades now. One of the first papers which put some light on whether a machine can attain decision-making capabilities and purposed to answer the question *Can machines think?* was published by Alan Turing and was titled **COMPUTING MACHINERY AND INTELLIGENCE** .

Which quotes -
In the process of trying to imitate an adult human mind we are bound to think a good deal about the process which has brought it to the state that it is in. We may notice three components,

- *The initial state of the mind, say at birth,*
- *The education to which it has been subjected,*
- *Other experience, not to be described as education, to which it has been subjected.*

Instead of trying to produce a program to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child-brain is something like a note-book as one buys it from the stationers. Rather little mechanism, and lots of blank sheets. (Mechanism and writing are from our point of view almost synonymous.) Our hope is that there is so little mechanism in the child-brain that something like it can be easily programmed. The amount of work in the education we can assume, as a first approximation, to be much the same as for the human child. – **Alan Turing, 1950**

The quote [31] essentially talks about the process of learning and elaborates that it could be difficult to actually write a program that can replicate an adult mind rather Turing was conjecturing that it could be easy to write a program that could learn by itself as a child does by interacting with the world and gaining experience from it.

1.1.2 Artificial Intelligence and Machine Learning

Definition of AI Numerous definitions were quoted over decades of research by different scientists in Artificial Intelligence, but the most popular definitions says :

Intelligence is something which is demonstrated by machines, as opposed to natural intelligence displayed by animals including humans

The above definition describes an Intelligent Agent [27] that can perceive an environment and take actions on it with the motive of maximizing the agent's possibilities of achieving its target. And hence AI is just a science that enables a machine or a program to learn, think and act like humans.

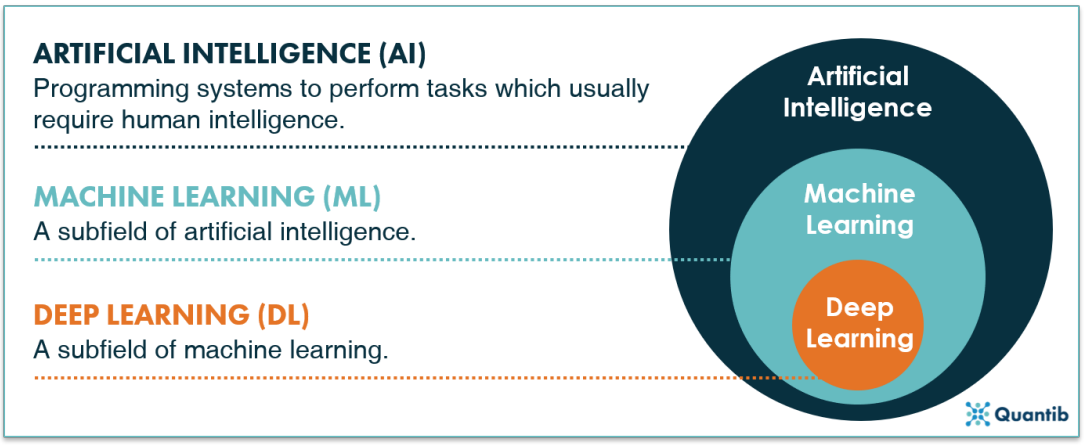

Figure 1.1: Hierarchy of Artificial Intelligence

Machine Learning AI is a vast and diverse domain containing a number of sub-classes, one of its main subset is *Machine Learning*(Fig 1.1)[1] which inherits ideas from AI . The term was coined by American scientist Arthur Samuel in 1959 [5] and the area focuses on synthesizing useful experiences from the data or otherwise referred to as a LEARNING PROBLEM where we build machines which can perform specific tasks WITHOUT EXPLICIT PROGRAMMING. This involves feeding the machine learning algorithms also termed as *models* with structured data — usually in the form of rows and columns — and then **learning** from it to act as precisely as possible on the unknown versions of the data. Hence, the name MACHINE LEARNING.

Deep learning In order to learn from the structured data and make predictions on the unknown data, a number models were established. But, Deep Neural Networks — a mechanism which mimics human brain functionality — provide an edge over other mechanisms. *Deep learning* [11] is a subset of Machine learning (Fig 1.1) which takes the advantage of Deep Neural Networks to memories — just like a human brain — plethora of important information about the data.

1.1.3 Types of Machine Learning

As far as the LEARNING PROBLEMS are concerned ML consists of three prime categories - Supervised, Unsupervised, and Reinforcement learning(Fig 1.2)[2]

Supervised Learning This ML technique makes use of *labeled* data-sets to train machine learning algorithms in order to reliably classify or predict for unseen data. As more data is introduced into the model, the parameters of the model or

[1]Image source : quantib.com
[2]Image taken from amazonaws.com

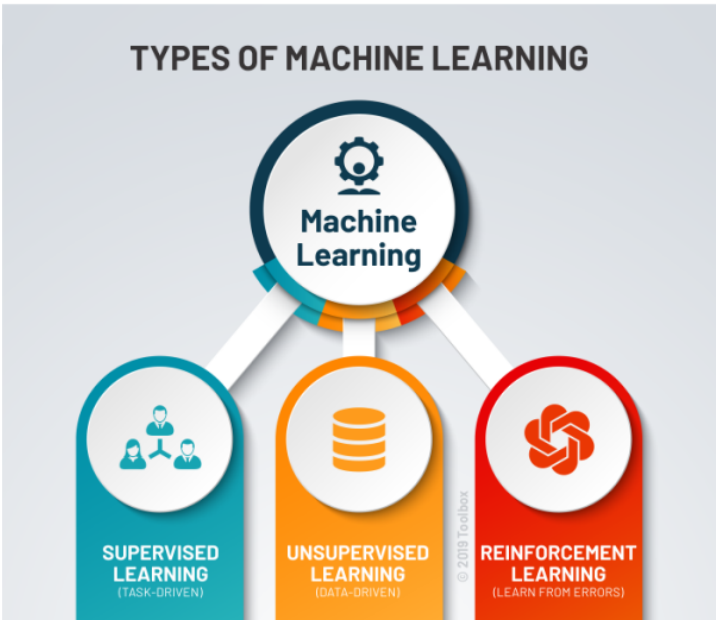

Figure 1.2: Types of Machine Learning

the algorithm are adjusted to determine correct labels for the unseen data. Neural networks, linear regression, logistic regression, and random forest are examples [3] of supervised learning techniques which are intensively used by organizations to tackle a wide range of real-world problems like email-spam detection, speech and object recognition.

Unsupervised learning As compared to labelled data-sets *unlabelled* data-sets exist too. Unsupervised Learning, uses a process of clustering unlabeled data and analyses these clusters by uncovering hidden patterns within the them. The algorithms in Unsupervised learning have the ability to find similarities and contrasts in the data making them a data driven LEARNING PROBLEM. Neural networks, k-means clustering, probabilistic clustering are some of the approaches [3] employed for Unsupervised Learning which prove worthy in applications like exploratory data analysis, consumer segmentation, picture and pattern recognition.

Reinforcement Learning The third category is a behavioral machine learning paradigm that is comparable to Supervised Learning but does not have sample labelled data to train the algorithms.Instead, RL algortihms use *trial and error* mechanism to generate the sample data and learn from it. While creating the sample data ,the algorithm trys a bunch of stuff and *reinforce* a series of successful outputs back into the model , eventually improving the models accuracy as it goes.Thus, it is called *Reinforcement Learning*.

1.1.4 Relevance of Reinforcement Learning in the Industry

Reinforcement Learning has a wide range of applications in different areas like games, robotics, computer vision,finance and healthcare.It is one of the most exciting fields of Machine Learning today. With a rough market size of 8.81 Billion USD in 2022 [2], Artificial Intelligence is an in-demand field which is growing size-ably and RL has a lot of potential to grow with it.

RL has been in lame light since 1950s and has produced many interesting applications, particularly in gaming Industry and Optimal control. TD-Gammon is an example of a computer program — to play the two player-backgammon game — developed in 1992 by Gerald Tesauro at IBM's Thomas J. Watson Research Center. TD-Gammon made use of Temporal-difference (a type of RL approach) [30] hence the name TD-Gammon.

A British startup in 2013 named DeepMind, demonstrated using a RL algorithm called *Q-Learning* that a system could learn to play Atari games [20] just using raw pixel values and eventually outperformed humans.This was one of the first big achievements accomplished in the field of Reinforcement Learning. Another big achievement was the development of `AlphaGo` computer program which was trained to play the board game "Go". The program defeated *Ke Jie* (the number one human Go player) at Future Go Summit in 2017. This was a miracle because no computer program had ever beaten a master of this game before.

As of today , the area of RL is an in-demand field of machine learning with an extensive range of applications. Since ML is being utilized across various branches for its real-world applications, it becomes obvious that RL will also play a vital role in constructing general purpose AI systems.

1.1.5 Use of RL in Control Systems

Apart from being frequently employed in gaming Industry. Reinforcement Learning is nowadays also being used by a number of Fortune 500 companies for Control system problems. These are problems which are quite complex and require a lot of human intensive calibration and optimization , some examples include Autonomous Cars , Vertical takeoff or landing systems for Rockets and Automated Industrial Manufacturing. Dynamic systems like these , have control mechanism [8] that can manage and regulate the behavior of other devices in the system. One important thing which is common in all these complex systems is the role of an *Objective function* , which can be visualized as a target function for the system and needs to be optimized.*Optimal control theory* [4] a branch of mathematical optimization that deals with finding a control for such dynamic systems over a period of time helps in *optimizing* these Objective functions.

Since , these Control System problems are very complex and involve thousands of variables — through which the dynamics of the system are controlled — , they become suitable for RL paradigm because such high-dimensional problems involving thousands of variables can be solved using the rigorously trained models or Deep Neural Networks through the *trail and error* approach.And the prime benefit of using Reinforcement Learning for Control System Problems is, it certainly saves the time needed for EXPLICIT PROGRAMMING and avoids human intervention.

1.1.6 Overview and Illustration of RL

This section gives a short overview of a Reinforcement learning problem through an illustration so that it becomes easy to understand details of other approaches in the upcoming chapters.

Ingredients of RL A Reinforcement Learning problem consists of two prime entities ,an agent and an environment [28].The agent is an *Intelligent Agent* as discussed in Section 1.1.2 . Environment is the world of this Agent in which it lives and interacts.The environment can be in different states at different points in time. During the training phase of a RL problem, an agent executes the following sequence of operations on the environment at any time step t:

1. Perceive the **State** S_t of the Environment
2. Take an **Action** A_t on the Environment based on the perceived state
3. Receive **Reward** R_{t+1} from the Environment based the action taken

Environment makes a transition to a new state based on the agent's action.

These sequence of operations are perceived as the basic setting of a Reinforcement Learning problem and are termed as the *Loop of Interaction* [Fig 1.3] [3].And one iteration of such loop is known as a *Step* taken by the agent.Hence, *Step* can be represented as a tuple of three :

$$Step_t = (S_t, A_t, R_{t+1}) \tag{1.1}$$

Rewards are the points that the agent receives from the environment when it takes an action on it. The rewards can either be positive or negative designating if the action taken is good or bad respectively. The magnitude of the reward point represents *how much* good or bad the chosen action was.

[3]Image source : freecodecamp.com

The prime *goal* of the agent during the training phase, is to maximize the cumulative rewards through **iterative** execution of the Loop of Interaction.

Figure 1.3: The Loop of Interaction

Illustration Figure 1.3 illustrates the game of Super-Mario where the environment is represented by a 2D space that could be in different states at different times. The agent is the player in the game which can move through the environment taking actions like (JUMP/LEFT/RIGHT). So, given the state of the environment, the agent has the goal of maximizing the cumulative rewards by repeatedly interacting with environment following the *trial and error* approach.

1.1.7 The two RL Approaches

Since agent is the decision making entity in the RL problem, so making correct decision by taking correct actions is eventually the prime motive of the agent.

For choosing correct actions two of the popular decision making approaches are Value Based and Policy Based [Sec 2.3]. In *Value Based* approach the *quality* of the taken action known as *action-value* is being taken into consideration while making correct decision. Whereas, in *Policy Based* approach the Policy being used to choose specific action is refined to improve its probability of choosing correct actions as per the state received from the environment.

1.2 Motivation

Over the years a number of Deep Reinforcement learning algorithms have been developed and two of the most popular types follow the *Value Based* and the *Policy Based* approaches . However, there still exists lack of prior guidelines for choosing one of these methods and how one can apply these algorithms on

different control system environments. At the same time, it is also important to analyze which algorithm works better in a specific environment and also Judge Agents behavior. The behaviour of agent usually differs based on mathematical implementation of the algorithms. So, getting the mathematical insights of the algorithms ,helps us in contrasting the agents behaviour.

1.3 Problem Definition

The Problem consists of implementing agents using two algorithms namely *Deep Q learning with Experience Replay* and *Reinforce Policy Gradient* which are based Value Based and Policy Based approaches respectively. Both the algorithms are different in characteristics and behavior.Hence ,these algorithms can help us answer the questions as described in Section 1.2, which include -

- How to apply these algorithms in constrained Environment conditions?
- Analyze which algorithm is better in a particular environment?
- Difference in behavior sighted by the Agents?

These unaddressed consequences laid the foundation of the research, and the research will provide insights and solutions of these questions in Chapter 5 once experiments have been conducted for both the algorithms.And we eventually setup guidelines for choosing one algorithm over the other.

1.4 Research Method

This section will explains the experimental approach used in carrying out the research.

1.4.1 Experimental Approach

For conducting the experiments we used environments offered under a RL toolkit named Gym — a benchmark framework for testing RL algorithms. We use two different environments available in Gym framework namely the `CartPole-v1` and the `LunarLander-v2` for our experimentation.The environments were chosen based on there problem definition and complexity. We implemented the two agents (based on the two RL algorithms) as described in the Figure 1.4 :

- **DQN_Agent** : Based on Deep Q-learning with Experience Replay algorithm

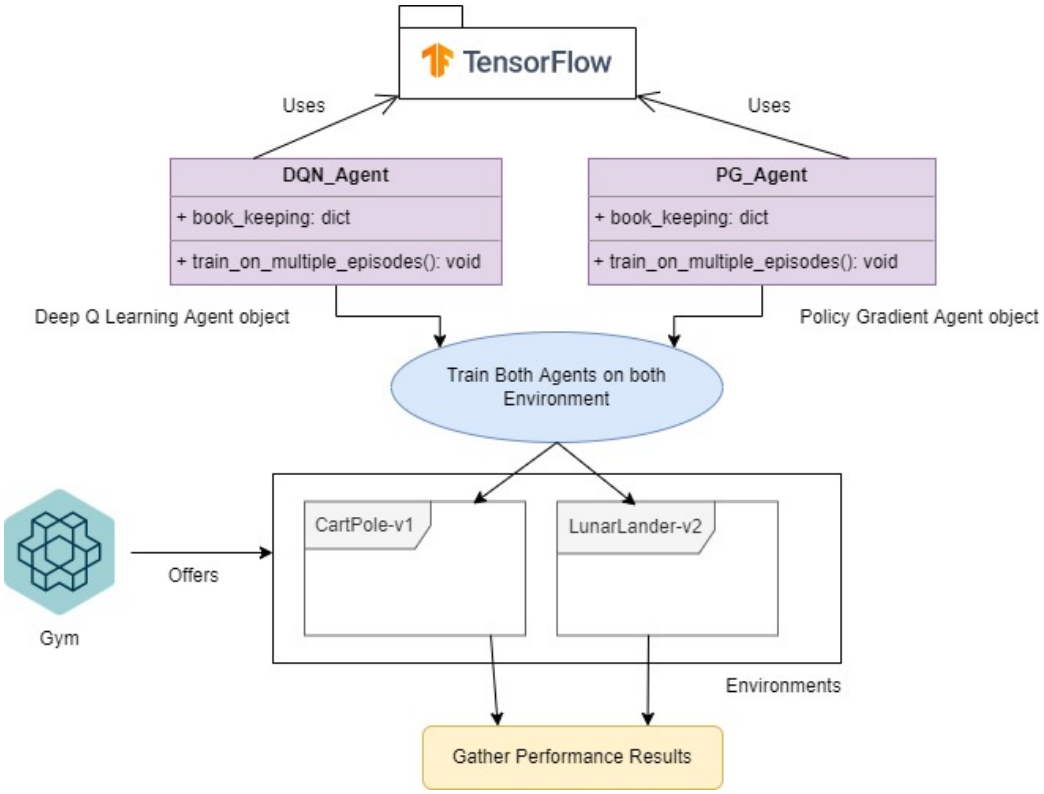

Figure 1.4: Experimental approach flowchart

- **PG_Agent** : Based on Reinforce Policy Gradient algorithm

Each agent internally used TensorFlow as an open-source package for training the deep learning models. Both the agents were trained on both the environments to gather the corresponding performance results.

1.5 Related Work

A number of papers have been written in past years which have relevance to our work. Andrew Zhang [1] worked on comparing two *Policy based* algorithms namely PPO and Vanilla Policy Gradient(VPG) or Reinforce Policy Gradient for Cartpole-v0 (rather than Cartpole-v1). On the other hand Journal [13] found results when working on LunarLander-v2 for the *Value based* Q-learning mechanism. But a `comparison` of the two different approaches —*Policy based* and *Value based* — on the both the environments — Cartpole-v1 and LunarLander-v2 — was inevitable which this paper tries to coup up with.

1.6 Aim of the Research

The research would try to answer the questions quoted in Section 1.3 by analyzing the performance of two agents namely the `DQN_Agent` and the `PG_Agent` based

on the evaluation criterion mentioned below :

- **Number of Episodes** required for sufficient training of agents
- **Average Steps** to solve the reinforcement learning problem
- Agent's Goodness to achieve the target which will try to capture agent's behavior by executing the agent on **multiple test instances** of the same environment.

Conclusively, the research will setup guidelines for the choice of algorithms once we have found the performance of the agents as per the above-mentioned evaluation criterion.

1.7 Thesis Outline

Thesis are organised into following Chapters:

- **Chapter 2** : Describes the background of the work elaborating the concepts related to Deep Reinforcement Learning and also giving mathematical insights of the Value Based and the Policy Based approaches
- **Chapter 3** : Defines experimental setup , where the environment specific configurations for the both the environments namely `CartPole-v1` and `LunarLander-v2` are elaborated.
- **Chapter 4** : Deals with the implementation of both the agents namely `DQN_Agent` and `PG_Agent` based on the Deep Q-learning with Experience Replay and Reinforce Policy Gradient algorithms respectively.
- **Chapter 5**: Deals with the results found during the experimentation and analyzes these results to setup guidelines for choosing the algorithms.
- **Chapter 6**: This is our final concluding chapter which sums up our findings and elucidates about the possible future works.

CHAPTER 2

BACKGROUND

This chapter has three main sections which will give a theoretical overview about the background of the research. The Sections include:

- **Deep learning** : This Section will elaborate how Deep Neural networks are structured and how they function. Understanding the concepts of Deep Learning is important because Deep Reinforcement Learning internally uses Deep Learning models
- **Reinforcement Learning** : This Section will dive deep into the components of Reinforcement Learning and will clarify how they work together
- **Value and Policy based Iterations Methods** : Here we will understand the two types of mentioned RL approaches in detail. And we will put some light on the ideas behind the Q-Learning and Policy-Gradient algorithms which are based on these two approaches respectively.

2.1 Deep Learning

In Section 1.1.2 we mentioned that Deep learning uses DNN which try to simulate the behavior of a human brain. A Deep Neural Network uses *Single Layer perceptron* as a building block to replicate the human brain behaviour.

Deep Neural Networks can approximate complex non-linear functions from training examples and can solve difficult machine learning problems some of which were mentioned in Section 1.1.4. The crux of this complex decision making capability is delivered through *Artificial Neuron* which is an artificial fabrication of their biological counterparts — Biological Neuron — present in the Human brain.

2.1.1 Biological Neuron

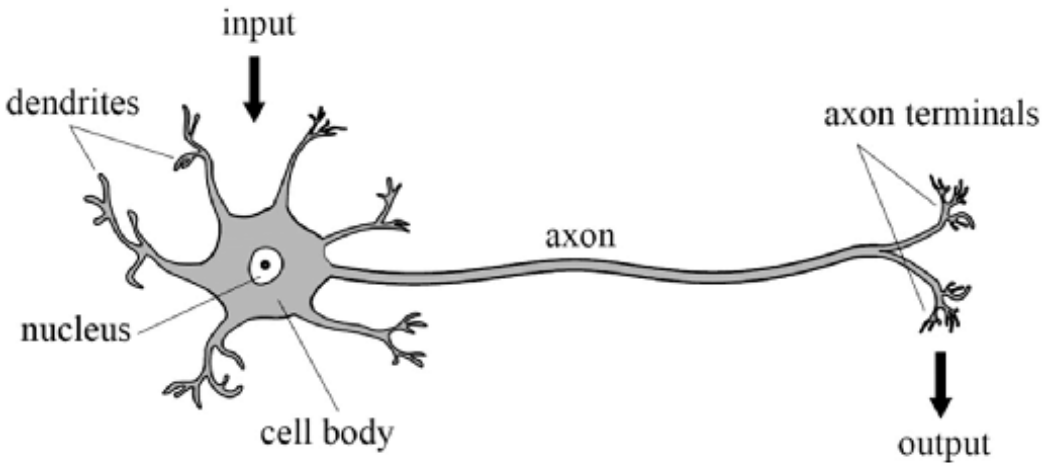

Figure 2.1: Structure of a Biological Neuron

A Biological Neuron [Fig 2.1] [1] is made up of cell body inside of which there is a nucleus. There exist many branched extensions in the nerve cell, along which impulses are received from other cells these extensions are called *dendrites* and there is a long extension known as *axon*. Near to the end of axon are *axon terminals* which are connected to dendrites of other neurons. Through the axon terminals a biological neuron receives electrical signals or *impulses* from other neurons. If a neuron receives sufficient number of signals or impulses then it fires other neurons in its network. These neuron are present with billions of other neurons, which form a vast network of connected neurons termed as *Human Cortex* [Fig 2.2][2]. A Human cortex can be imagined as a DNN which has the ability to perform highly complex computations.

[1] Image source : researchgate.net
[2] Image source : oreilly.com

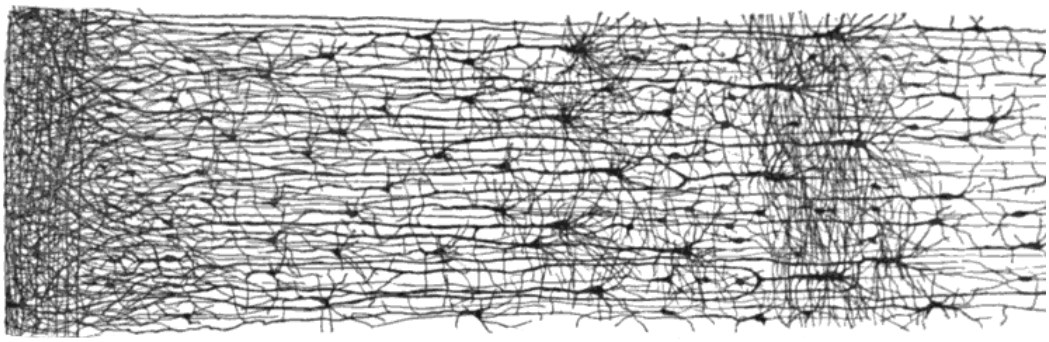

Figure 2.2: Part of Human Cortex

2.1.2 Single Layer perceptron

An artificial neuron — an elementary unit of a DNN — is mathematical function based on a model of biological neurons. In an artificial neuron [Fig 2.3] [3] each of the inputs $x_1 \dots x_n$ in the neuron are weighted separately by weights $w_1 \dots w_n$ to produce a sum z [Eq 2.1]. The sum is then passed through a non-linear function known as an *activation function* σ, which replicates *firing* of biological neuron.

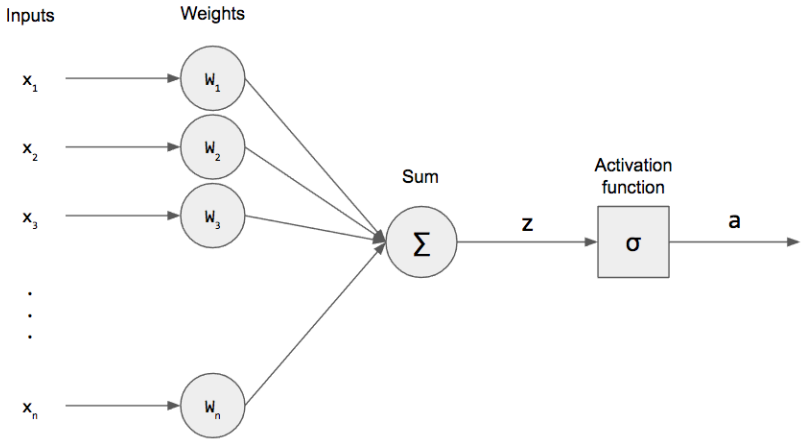

Figure 2.3: Single Layer Perceptron

Following equation represents the weighted sum of input features $x_1 \dots x_n$:

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n \tag{2.1}$$

A better way to represent this weighted sum z is in a vectorized manner through an equivalent matrix notation as follow :

[3] Image Source : pythonmachinelearning.pro

$$z = \theta^\top X \qquad where \qquad \theta = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad and \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.2}$$

And after passing through the activation function σ the output from the artificial neuron will be written as :

$$a(x) = \sigma(\theta^\top X) \tag{2.3}$$

Figure 2.3 is also known as a *Single Layer Perceptron* [10] because it consists of a Single Layer of input vector $x_1 \dots x_n$ and a single layer of output vector a

ReLu Activation function There are a number of activation functions present in the literature some of which include sigmoid, hyperbolic tangent and rectified linear unit (ReLu). However, ReLu is the most popular amongst all because it greatly accelerates the convergence [19] of gradient descent — an optimization algorithm used to minimize the objective function — compared to the sigmoid or hyperbolic tangent functions and often delivers better performance.

ReLU is a function [Eq 2.4] wherein If, the input is positive than it will output the input as it is. Otherwise, it will output zero.

$$\sigma(x) = \begin{Bmatrix} x & x > 0 \\ 0 & x <= 0 \end{Bmatrix} \tag{2.4}$$

2.1.3 Deep Neural Networks

A Deep neural network (DNN) is a collection of numerous simulated neurons. Each neuron can have multiple input and output connections and each neuron is connected to other neurons through *links* or connections that are equivalent to biological axon and dendrites connections in biological Neuron. Every link in a DNN connects the output of one neuron to the input of another neuron. In Figure 2.4[4] each circle represents an artificial neuron and the arrows designate the links. Neurons in a deep network [9] is an enhanced version of Single Layer perceptron and is organized into multiple layers, where the neurons from one layer connect to neurons in the preceding and the following layers. A DNN usually has three types of layers — the input layer, hidden layers and output layer. The **input layer** receives the external data.The **hidden layers** help in improving the accuracy of predictions. A DNN contains zero or more hidden layers and

[4]Image Source : `1.cms.s81c.com`

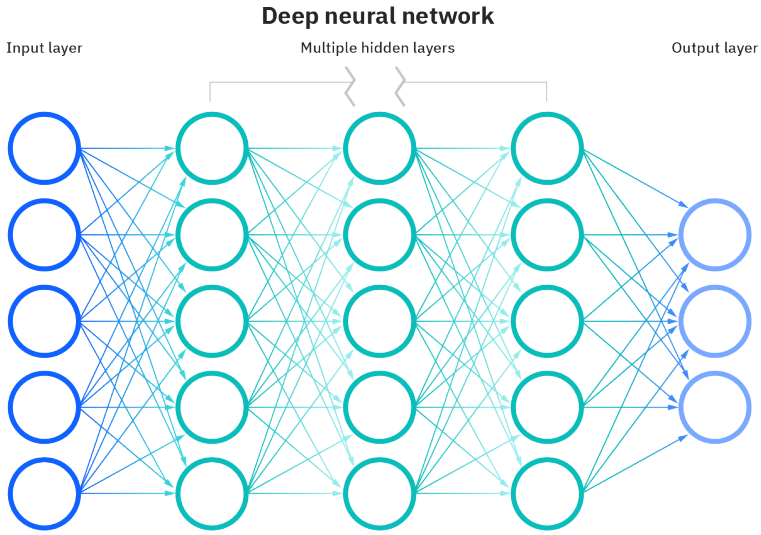

Figure 2.4: Deep Neural Network

at last the output layer gives the ultimate results.The **output layer** gives the predictions. Additionally, in a DNN *every* neuron in one layer is connected to *every* neuron in the next layer making it a *DEEP* connection hence the name Deep Neural Network.

2.1.4 Training of DNN

Training a DNN relies on an optimization algorithm called *Gradient Descent* [5] which determines a good solution for the above quoted neural network with function $a(x)$ [Eq 2.3]. In *Gradient Descent* the `Gradient` is a vector which is partial derivative of the loss function or the *objective function* $J(\theta)$ with respect to each of weights of the DNN denoted by ∇J

$$\nabla J(\theta) = \left(\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_n} \right) \tag{2.5}$$

And the loss function or the *objective function* $J(\theta)$ is a function which is to be *optimized* by reducing the loss or error between the predicted values from the function $a(x)$ and the actual or labelled values [5]. Since, the loss of the network has to be *reduced* the update rule at every time step for the weights of the network is as follows :

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t) \qquad where \;\; \alpha = learning\ rate \tag{2.6}$$

We can repeatedly apply the update rule with the hope that θ — parameter of weights — converges to the value that minimizes $J(\theta)$ [Fig 2.5]. By the use of this

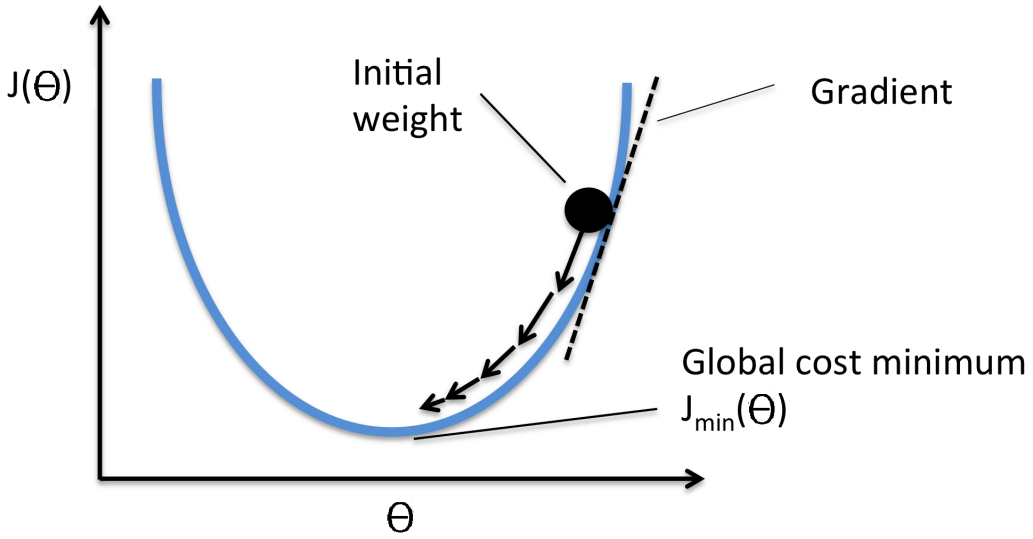

Figure 2.5: Gradient Descent Algorithm

update rule we simply *propagate* the error back into the the network adjusting its weights , so that DNN makes better predictions as it learns.

Gradient ascent Gradient descent optimization algorithm is closely related to its counterpart the `Gradient ascent`. The only difference is that gradient descent tries to *minimizes* the function $J(\theta)$ by taking steps in the direction of the -ve gradient, whereas `Gradient ascent` tries to *maximize* the function by taking steps in the direction of the +ve gradient and hence the update rule becomes as follows:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \qquad where \;\; \alpha = learning\ rate \tag{2.7}$$

2.1.5 Hyperparameters

Set of parameters which can improve the learning process of the DNN's are known as Hyperparameters. These parameters mainly include : number of hidden layers,number of neurons per layer,learning rate or batch-size.

Number of Hidden Layers Data in its physical form is usually structured in a hierarchical manner. Say, lines belong to a low-level , circle or square belong to intermediate-level and objects belong to high-level in the hierarchy [16]. DNNs can exploit this hierarchical composition where input layers replicate the low-level , hidden-layers recreate the intermediate-level and output layers mimic the high-level in this composition.The composition helps DNNs determine good solution quickly and also extrapolate to new datasets.Hence, the number of layer in a

DNN depend on this composition. Usually, one or two hidden layers are sufficient, though one can increase the number of layers for more complex problems.

Neurons per Layer The number of neurons in the input and output layers are evident as per definition of the RL problem. For example, in LunarLander-v2 environment [Section 3.3.2] the input layer consists of 8 neurons and the output layer consists of 4 neurons based on the size of input and output vectors respectively. In case of hidden layers, the usual convention is to have more number of neurons in the preceding layers and fewer in upcoming layers [16], because features of preceding layers get fused into fewer features in upcoming layers.

Learning Rate A small positive value denoted by the symbol α in the gradient descent update rule [Eq 2.6] is known as Learning Rate.It is usually in range 0.0 and 1.0. Learning Rate controls how swiftly the weights of the DNN will get updated. If α is small the convergence of the network is slow and fast otherwise.If α is quite big then weights of the model or the network could fluctuate, never converging to a sub-optimal solution. So, the Learning Rate is regarded as the vital hyperparameter for the DNN [17].

Batch Size The batch size makes the task of training the network easy by dividing the training dataset into a number of samples [12]. Say, the training dataset is of size 1,000 and our batch-size is 10 ,then the we would have 100 batches of size 10 each. And during the training process the algorithm would pick one batch after the other to train the network. Batch size is also considered another important hyperparameter [16] for the model and is usually initialized with a value lower than 32

2.2 Reinforcement Learning

In Section 1.1.6 we gave an overview of RL through an illustration and explained some of its ingredients. In addition to this there are three main sub-components of a RL system which include the *policy* , the *value function* [28] and Markov Decision Process(MDP).Where MDP is a classical formalization of sequential decision making in any RL paradigm.

Policy A policy [28] usually denoted by π defines the behavior of an agent at a given time. The policy is normally stochastic in nature carrying a *mapping* from the states of the environment to probabilities of choosing each feasible action. Then, we already mentioned that *Reward* is a number received by the agent once

it takes an action on the environment. It indicates the goodness of an action in an immediate sense and in long run the agent's motive is to maximize the cumulative rewards. The policy is dependent on reward in a way that; if an action being chosen by the policy results in low rewards, then it makes sense for the policy to discourage that action in the future and if an action selected by the policy results in high rewards, then the policy should encourage that action in the future.

Policy π_θ is usually a DNN parameterized by θ — weights of the DNN.

$$\pi(a|s,\theta) = \mathbb{P}(A_t = a|S_t = s, \theta_t = \theta) \tag{2.8}$$

So, π_θ [Eq 2.8] represents the probability of taking action a , when the environment is in state s at time t and is parameterized by θ.

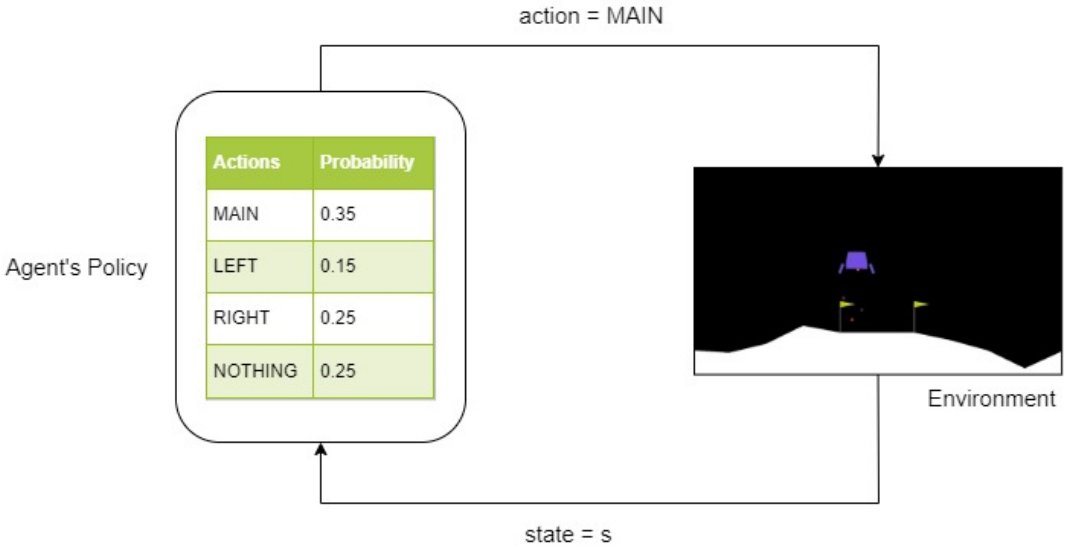

Figure 2.6: Agent's Policy in LunarLander-v2

If we consider the example of LunarLander-v2 [Section 3.3.2] environment which has 4 actions to choose from.The agent's policy at any specific time step t would output an action-probability distribution.And according to this probability distribution,the agent would pick an action to execute on the environment. Here, probability distribution indicates the probability of an action to get selected, as per [Fig 2.6] MAIN, LEFT , RIGHT and NOTHING had 35 , 15 , 25 ,25 percent chance of being selected respectively.

Consequently, a *trained* agent would have mappings in its policy π_θ , which would specify the BEST action — in terms of probability — to be chosen by the agent

Value function If policy describes the behaviour of the agent a value function — a function of states or a function (state–action) pairs — [28] defines the

goodness of a state in the *long run*. It means "how good" it is for the agent to be in a specific state and this goodness of a state is evaluated in terms of the *future expected* rewards. In a sense this means the agent would try to choose actions that brings it to a state with the *highest value* and not the highest reward, because agent would obviously have to think into the future and not in the immediate sense. Value Function depends on MDP for sequential decision making and evaluation of state-values.

2.2.1 Markov Decision Process

A Russian mathematician named Andre Markov published a paper [15] in the year 1906 on a topic *Markov Chains*. Markov Chains is a process which has no memory and has fixed number of states. At any timestamp, the process can be in any of the possible state and can make transition to a new state based on the transition probabilities distribution.For instance, as per Figure 2.7 [5] the probability of making a transition from state S_0 to S_1 is 0.2 . State S_3 represents a **terminal state** because when the process reaches this state it always remains in the same state with 100 percent probability and S_0 the **starting state**.The transition is independent of the past states of the process and hence the system has no memory.

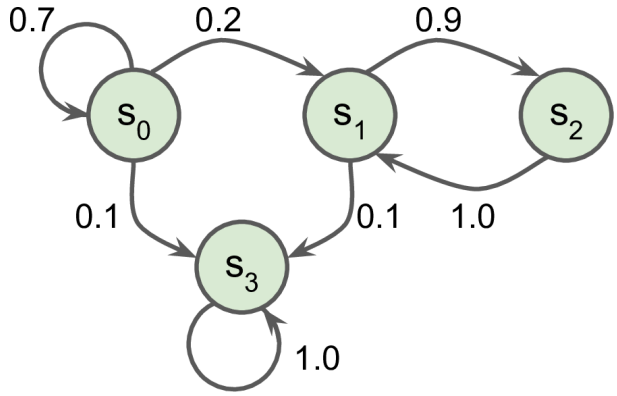

Figure 2.7: Markov Chain with four states

Markov decision process [7] which uses Markov Chains and is based on Dynamic Programming was first introduced in the year 1957 by mathematician Richard Bellman. MDP clubbed the states with respective actions — that could be chosen from those states — and every *state-action* pair had an attached transition probability and a reward. For instance, assuming at any time step the agent is in state S_0 [Fig 2.8][6] there exists three possible state-action pairs from S_0, namely (S_0,a_0) , (S_0,a_1) and (S_0,a_2). Being in S_0, if the agent chooses action

[5]Image Source : `1.cms.s81c.com`
[6]Image Source : `learning.oreilly.com`

a_0 then the process has 70% chance to make transition to state S_0 and receive +10 as reward points , otherwise its has 30% chance to land on the state S_1 receiving no reward point.

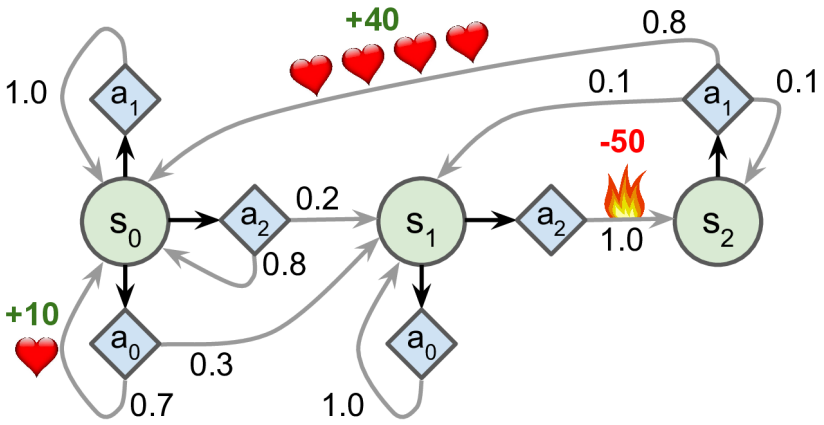

Figure 2.8: An example of Markov Decision Process

For the future reference,the following notations were used in context to MDP:

- $T(S, A, S')$ is the probability of making a transition from the state S to a new state S' if agent has chosen action A. For example, $T(S_2, a_1, S_0) = 0.8$
- $R(S, A, S')$ is the Reward point, which is received by the agent if it makes a transition from state S to state S' when the agent chooses action A. For example, $R(S_2, a_1, S_0) = +40$

The prime goal of the agent would be to determine an *optimal* MDP, which does sequential decision making and makes transition to new states in turn maximizing the cumulative rewards over long run.

2.2.2 Agent–Environment Interface

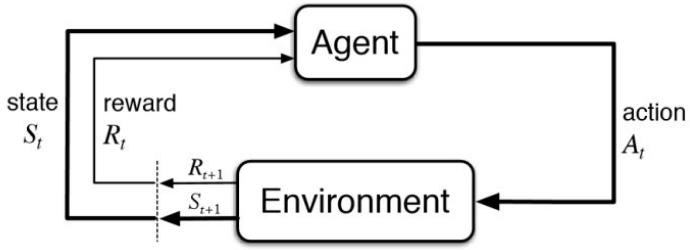

Figure 2.9: Agent–Environment Interface

As already mentioned, a MDP simplifies the sequential decision making process by the use of state-action pairs.The agent makes use of the MDP to interact with the environment [Fig 2.9] and in turn recreate the “Loop of Interaction”.

At each time step $t = 0, 1, 2, 3 \ldots$, the agent executes a *Step* [Sec 1.1.6] as per the "Loop of Interaction". And sequential executions of these *Step* creates a trajectory τ of operations through the MDP which could be represented as follows:

$$\tau = (S_0, A_0, R_1), (S_1, A_1, R_2), \ldots \tag{2.9}$$

where, S_0 is one of the starting states of the MDP. The MDP makes a transition from the state S_0 to state S_1 when action A_0 was chosen by the agent and it received a reward R_1 for this chosen action. In a similar manner the trajectory is generated.

Episode In the trajectory τ , when the MDP ends up being in one of the `terminal states` say S_T.Then, for time steps $t = 0, 1, 2, 3 \ldots T$, where

$$\tau = (S_0, A_0, R_1), (S_1, A_1, R_2), \ldots S_T \tag{2.10}$$

Such a trajectory is called an *Episode* through the MDP

2.2.3 Return

We mentioned in Section 2.2 for value function that the goal of the agent is to maximize the expected rewards in the long run. Assuming at timestamp t, the MDP is in state S_t and the Episode ends at timestamp $T(T > t)$, now being in the state S_t we want to estimate "how good" it is for the agent to be in this state, in terms of the `expected` rewards. So, *from* the timestamp t , we denote the rewards thereafter by $R_{t+1}, R_{t+2}, R_{t+3}, R_{t+4}, \ldots$. Using this notation we aim to maximize the *expected future Return* [28] denoted by G_t. G_t could be any function which is based on these future rewards from timestamp t such as :

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.11}$$

Future Discounting of Return In G_t the future rewards are usually discounted by an arbitrary factor γ called the discount factor [28]. γ can have a value anywhere in between 0 and 1 , but it mostly ranges from 0.9 to 0.99. When γ is close to 0, then immediate rewards — after timestamp t in G_t — are given priority to the ones far away in the future and if γ is close to 1, then it is the other way around. Equation 2.12 represents future discounted Return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.12}$$

We can also rewrite the Equation 2.12 in a *recursive* manner based Return at timestamp $t+1$ as in Equation :

$$\begin{aligned} G_t &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.13}$$

2.2.4 Exploration–Exploitation dilemma

During the training of the agent, one of challenge is to tackle the trade-off between `exploration` and `exploitation`. To maximize the total rewards the agent should prioritize or *exploit* states that were effective in generating more rewards. However, at the same time the agent also has to try or *explore* new states that were not selected before. This creates a *dilemma* for the agent to deal with , wherein neither exploration nor exploitation can be pursued at the same time.Hence, the agent has to try a bunch of actions and select the ones that prove worthy. This is the same *trail and error* approach we discussed in section 1.1.5

ϵ - greedy method Since it becomes important for the agent to explore the MDP thoroughly and using a purely random exploration policy would eventually visit every state and every transition in the MDP a number of times.However, a better alternative is to balance the exploration–exploitation factors , and this balancing method is termed as ϵ-greedy method : wherein at each step the agent either acts randomly by selecting a random action with probability ϵ or otherwise the agent acts greedily by selecting an action which is having the highest `State-Value` with the probability 1- ϵ. Using ϵ-greedy method of action selection is beneficial cause the agent would then start by exploring more at the beginning of the training and will be eventually exploiting more — the information it has learned in the exploration phase — during the end of the training phase.

2.3 Value Based and Policy Based Methods

Value based methods [Fig 2.10] [7] are used to learn value functions and these methods usually have an implicit mechanism of action selection like ϵ-greedy approach , and use Values of States or State-action pairs to decide which action has to be selected at specific timestamp.

On the other hand *Policy based* methods try to determine the best policy from among the set of feasible policies. For determining the best policy the methods tweak the policy itself through its parameters.

[7] Image Source : RL Course by David Silver - Lecture 7

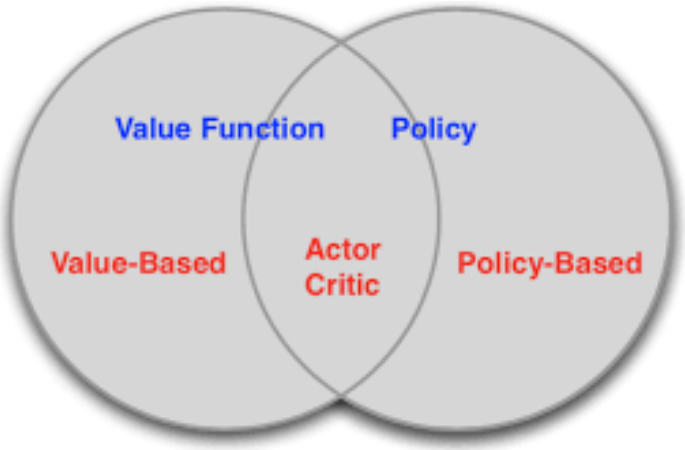

Figure 2.10: Summary of approaches in Reinforcement Learning

Apart from these methods there another kind of method called *Actor-Critic* which try to take the benefits of both the worlds wherein these methods try to club both the ideas by not only estimating the value functions but also trying to determine the best policy.However, our problem statement just focuses on the first two approaches.

2.3.1 Value Based Method

In the introductory part of this section, we clarified how the value functions helps us find the goodness of actions by estimating the State-values , and hence these value functions are called `State-Value` [28] functions, however sometimes we could also be estimating the values of state-action pairs terming these value functions as `Action-value` [28] functions. For determining the *expected* values of states ,value iteration methods can be used and Q-learning is one such method that determines the optimal values of state-action pairs.

State-Value Functions The `State-Value` function of a state S in an MDP is denoted by $V_\pi(S)$. The function is the Return *expected* by the agent if starts from S and follows the policy π. Here $\mathbb{E}$ denotes the expected value. Refer Equation 2.13 for G_t

$$V_\pi(S) = \mathbb{E}_\pi[\, G_t \mid S_t = S] \tag{2.14}$$

Action-Value Functions In a similar way we can also define the value or the goodness of taking an action A when in state S ,as state-action pairs denoted by $Q_\pi(S, A)$ — "Q" indicating the *Quality* or the goodness of the action chosen.

Here $Q_\pi(S, A)$ would now represent the Return expected by the agent starting

from the State S given it picked action A and followed π as its policy.

$$Q_\pi(S, A) = \mathbb{E}_\pi[\ G_t \mid S_t = S\ ,\ A_t = A]$$

Bellman Optimality Equation

In the year 1957 mathematician Richard Bellman introduced an iterative equation to determine the optimal value for a state , this equation was named after him as the *Bellman Optimality Equation*.The equation could be derived from the State-value function $V_\pi(S)$.

Derivation In the Section above we already computed $V_\pi(S)$ .Now we can use Equation 2.13 to change the equation as follows:

$$\begin{aligned} V_\pi(S) &= \mathbb{E}_\pi\left[G_t \mid S_t = S\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = S\right] \end{aligned}$$

Since , the agent follows the policy π from the state S. $\pi(A|S)$ would give the probability-distribution of choosing different possible actions from the state S and after taking those actions the MDP would end up in the new state S'. But, the MDP would make the transition to this new state S' from state S under the transition probability $T(S, A, S')$ and would then receive the Reward $R(S, A, S')$. And since we would have made the transition to the new State S' , we would then evaluate the expected return from S' in the next timestamp t+1. And this expected return can be written in terms of the State-Value function for the next state S' as $V_\pi(S')$. So we may rewrite above equation as follows:

$$\begin{aligned} V_\pi(S) &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = S\right] \\ &= \sum_A \pi(A|S) \sum_{S'} T(S, A, S') \left[R(S, A, S') + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = S'\right]\right] \\ &= \sum_A \pi(A|S) \sum_{S'} T(S, A, S') \left[R(S, A, S') + \gamma V_\pi(S')\right] \end{aligned}$$

Now the **optimal** State-value for a state S would be the maximum-value from amongst the different policies possible from state S , stated as :

$$V_*(S) = \max_A \sum_{S'} T(S, A, S') \left[R(S, A, S') + \gamma V_*(S')\right]\ \forall\ S \tag{2.15}$$

This Recursive Equation 2.15 is the *Bellman Optimality Equation* and it manifests that once the agent is in a particular state S it has a set of actions “A” to

choose from then if the agent acts optimally then $V_*(S)$ — the optimal value of the current state S — is equal to the maximum of sum of discounted future reward the agent is going to receive on average after it takes an optimal action, plus the expected optimal value of all possible next states S'. This equation helps in deriving the `Q-learning` algorithm which can precisely estimate the optimal action-value of every possible state in the MDP.

The recursive version of Equation 2.15 can be written as an iterative version as mentioned below and is known as the `Value Iteration` Equation [16] :

$$V_{k+1}(S) = V_k(S) + \max_A \sum_{S'} T(S, A, S')[R(S, A, S') + \gamma.V_k(S')] \ \ \forall \ S \qquad (2.16)$$

In this equation, $V_k(S)$ denotes the estimated value of the state S at k^{th} iteration of the algorithm. Conclusively, when iterated a number of times it is guranteed to converge to optimal State-Value for the state.

Q-Learning

When the agent starts with the training process, it has no information about the transition probabilities $T(S, A, S')$ and rewards $R(S, A, S')$. Therefore, the agent has to explore the states of the MDP — many times — to estimate the transition probabilities.In this concern Q-learning algorithm, which is a Value-based algorithm uses Value Iteration equation for its derivation.

Initially, when the Q-learning agent only has some information — usually in terms of random weights and biases of the DNN — about the MDP. The agent would use an exploration policy like the ϵ-greedy strategy [Sec 2.2.4] and start exploring new states. When the agent explores a state of the environment the algorithm will update the Q-value of the state based on the observed rewards and the current transition probabilities. This step in then repeated during the training phase by the agent until all the Q-values for all state-action pairs converge .For this, `Q-value Iteration` [16] equation which is similar to Value iteration equation proves worthy :

$$Q_{k+1}(S, A) = Q_k(S, A) + \sum_{S'} T(S, A, S') \underbrace{\left[R(S, A, S') + \gamma. \max_{A'} Q_k(S', A')\right]}_{\text{target Q-value}} \ \ \forall \ (S, A) \qquad (2.17)$$

The only difference here as compared to Value Iteration equation is that once the agent is in state S, it should choose the action with the *highest Q-Value* for that state from the next State S' which is done using *max* . The algorithm keeps track of running average reward and the sum of future discounted rewards —

calculated by taking the highest or the maximum of the Q-value for the next state S' — this information or "knowledge" of every state-action pair could be usually stored in form of a table or a dictionary.

Importance of Deep RL

The agent usually gathers the "knowledge" in a rudimentary way in form of a table containing Q-values for the state-action pairs. However, when the environment gets complex, the knowledge space can become really huge and it is no longer feasible to store all state-action pairs.This phenomenon is known as the "Curse of Dimensionality" which was again coined by Richard Bellman in 1957 [6]. Curse of Dimensionality is a situation where the algorithms are hard to design in high dimensional spaces and such algorithms often have a running time which exponential in nature. To overcome this drawback the concept of *Deep Reinforcement Learning* was introduced which takes the advantage of DNNs , which can memorize outputs even for high-dimensional spaces. Effectively, a DNN could be used to predict the Q-values based on the input state-action pairs. This is a more tractable way than storing every possible value as a tabular representation. The end effect is, deep neural network allows reinforcement learning to be applied to larger problems and exhibit impressive results for such high dimensional complex problems. Therefore, this process of clubbing DNNs with RL is termed as Deep Reinforcement Learning.

Training the Q-Network

So, as far as complex real-world problems are concerned , for gathering the "knowledge" DNN's are useful. And the DNN which uses Q-learning Algorithm for its training is called a `Deep Q-Network`.To train the DQN one needs to calculate the target Q-Values [Refer Eq 2.17] for every state-action pairs.

$$Q_{target}(S, A) = R(S, A, S') + \gamma . \max_{A'} Q_\theta(S', A') \tag{2.18}$$

Target Q-value is calculated by summing reward R that is observed after taking action A in state S, plus the discounted value by acting optimally from there on and to determine the sum of future discounted rewards, the DQN is executed on the next state S' and for all possible actions A' and choosing the optimal value with highest Q-value denoted as Q_θ. Once the target Q-value is calculated we just need to run a training step of the Gradient descent [Sec 2.1.4] thereby reducing the error between the Q-values $Q(S, A)$ — predicted by the DQN , when state-action pair is passed into the DQN — and the target Q-value $Q_{target}(S, A)$ using regression loss function like mean squared errors.

2.3.2 Policy Based Method

Till now, we discussed about Value-Based methods and understood Q-learning which is a Value-Based method. Now we will concentrate on Policy based methods which is another class of widely used RL methods which learns a parameterized policy π_θ that can select actions without the use of a value function. In a policy method the agent usually looks at an experience and then the agent figures out how to alter the policy in the direction that makes the policy better. So, the main idea is given a state s , which action has to be taken in order to maximize the reward. The way to achieve this objective is to fine tune the parameters of the Policy π_θ denoted by θ — usually the weights of DNN — to improve the policy. The policy denoted by π_θ is stochastic in nature as discussed before in Section 2.2. A DNN termed as the *Policy Network* recreates the policy by taking the state of the environment as input and outputs the action-probability-distribution based on an activation function, where the activation function is usually a *softmax* function which sums the action-probability-distribution to 1.

Since the policy is parameterized by θ and our goal would be to tweak the policy so that we can maximize the expected return by maximizing the gradient of the *objective function* $J(\theta)$ [Refer 2.3.2] w.r.t to the policy parameters θ

Policy Gradient Method Policy based method seeks to maximize the objective function using the *Gradient ascent* [Section 2.1.4] update rule.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{2.19}$$

Here $\nabla J(\theta_t)$ is the gradient of objective function $J(\theta)$ with respect to θ.
The methods which follow this update rule are named the *policy gradient methods.* Following the gradient ascent update rule , Vector θ moves in the direction of the gradient $\nabla_\theta J(\theta)$ after every update and eventually converges to the best θ for our policy π_θ which would give the highest return.

Reinforce Algorithm

Policy Gradient methods try to estimate the optimal policy by tweaking the weights of Policy Network through Gradient ascent update rule, thereby maximizing the expected return for the agent.The update rule 2.19 evaluates the gradient which indicates the direction which could increase the expected return. This process is repeated with the hope of determining the maximum expected return.The principal motive behind policy gradients is `Reinforcing` good actions , wherein we increase the probabilities of the actions that lead to higher expected returns, and decrease the probabilities of actions that lead to a lower expected

returns and repeat this until the agent is able to reach the `optimal policy`. Hence, The policy gradient method amend the weights of policy network to make state-action pairs which result in positive expected returns more likely and the ones with negative expected returns less likely to be chosen by the agent in the future.

One of the basic versions of the policy gradient algorithm called `REINFORCE` algorithm mimics the idea described above and acts as a benchmark for other advanced policy gradient algorithms like `Actor-Critic`. Reinforce algorithm [32] introduced back in 1992 by Ronald Williams relies on the action selection policy or primarily just the policy network to sample `trajectories` (Refer 2.10) and search for optimal policies by estimating gradients for the trajectories.Since for the fact the reinforce algorithm samples trajectories based on the Policy Network it is also known as `Monte Carlo` [28] Policy Gradients method because it uses episodes of experience (trajectories) — for maximizing the expected returns.The calculation of expected returns gets delayed until the episode terminates. Intuitively, in the Monte Carlo approach the gradient update does not happen at every step rather at the end of every episode.

Mathematical formulation of Reinforce Algorithm Now for the case reinforce algorithm tries to maximize the expected returns using sampled trajectories.It evaluates the `Return of trajectories` denoted by R(τ) for a specific trajectory τ.

Say for a particular episode E, our trajectory having T time steps is as follows when policy $\pi(a|s, \theta)$ is followed:

$$\tau = S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T \tag{2.20}$$

And `Return of a trajectory` τ is a tuple of expected returns until that specific time step:

$$R(\tau) = (G_0, G_1, G_2, \ldots, G_{T-1}) \tag{2.21}$$

where parameter G_k is the expected return for the transition k as depicted below

$$(S_k, A_k, R_k + 1) \tag{2.22}$$

Finally, `Return of trajectories` is the rewards that are expected to be collected from time step k till the end of the trajectory, and it is estimated by discounting the sum of rewards by the discount factor γ. For each time step k of the episode ranging from 0 to T-1 we evaluate G_k and build the `Return of trajectories` tuple as follows:

$$G_k \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.23}$$

Expected return in Reinforce Algorithm The goal of this algorithm was to find the weights θ of the Policy network that maximize the expected return denoted by $J(\theta)$.The definition of $J(\theta)$ is derived based on the Return of the Trajectory and the probability of the trajectory $\mathbb{P}(\tau|\theta)$ [26]. The probability of a trajectory τ starting from one of the states S_0 , defined in starting state-distribution ρ_0 and given that actions come from the policy π_θ is :-

$$P(\tau|\theta) = \rho_0(S_0) \prod_{t=0}^{T-1} \pi_\theta(A_t|S_t)\, T(S_t, A_t, S_{t+1}) \tag{2.24}$$

where $T(S_t, A_t, S_{t+1})$ [Refer Sec 2.2.1] is the transition probability in the MDP at timestamp t. And $\prod$ denotes the product of the individual terms.

Now , Based on the Return of the Trajectory $R(\tau)$ and the probability of the trajectory $\mathbb{P}(\tau|\theta)$, the objective function or the expected return $J(\theta)$ is defined as:

$$J(\theta) = \sum_{\tau} P(\tau|\theta) R(\tau) \tag{2.25}$$

We calculate the weighted average, where the weights are given by $P(\tau|\theta)$, the probability of each possible trajectory, of all the possible values that the return $R(\tau)$ can take. It is important to note that probability of trajectory $[P(\tau|\theta)]$ depends on the weights θ of the *Policy Network* π_θ which selects the actions following the policy π.

Gradient estimation in Reinforce Algorithm For the estimation of the Gradient $\nabla J(\theta)$ the following steps are needed -

- According to the simple rule of calculus : "the derivative of $\log x$ with respect to x is 1/x". Therefore,

$$\frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} = \nabla_\theta \log P(\tau|\theta) \tag{2.26}$$

  We can rearrange it the equation as follows:

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta).\nabla_\theta \log P(\tau|\theta) \tag{2.27}$$

- The log of the *probability of a trajectory* $P(\tau|\theta)$ is given by :

$$\log P(\tau|\theta) = \log \rho_0(S_0) + \sum_{t=0}^{T-1} \Big(\log \pi_\theta(A_t|S_t) \; + \log T(S_t, A_t, S_{t+1}) \Big) \quad (2.28)$$

- Now we evaluate the gradient of equation above, as follows:

$$\begin{aligned} \nabla_\theta \log P(\tau|\theta) &= \cancel{\nabla_\theta \log \rho_0(S_0)} + \sum_{t=0}^{T-1} \Big(\cancel{\nabla_\theta \log T(S_t, A_t, S_{t+1})} + \nabla_\theta \log \pi_\theta(A_t|S_t) \Big) \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t|S_t) \end{aligned} \quad (2.29)$$

  We cancel or remove some terms from the equation above, since the gradients of $\rho_0(S_0)$, $T(S_t, A_t, S_{t+1})$, and $R(\tau)$ are zero because they are independent of θ.

- Now ,we actually start evaluating the *gradient* of the objective function $J(\theta)$ as follows -

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_\pi[R(\tau)] \\ &= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) && \text{By Equation 2.25} \\ &= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) && \text{Bring gradient under integral} \\ &= \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) && \text{By Equation 2.27} \\ &= \mathbb{E}_\pi[\nabla_\theta \log P(\tau|\theta) R(\tau)] && \text{Expectation form} \\ \therefore \nabla_\theta J(\theta) &= \mathbb{E}_\pi \Big[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t|S_t).G_t \Big] && \text{By Equation 2.29} \end{aligned}$$

Note in the last equation we replaced $R(\tau)$ with G_t because of summation over the trajectory. Here, $\log \pi_\theta(A_t|S_t)$ is called the **score function** [32] and G_t the Advantage.

Score function indicates whether to increase or decrease the the weights θ of the policy network and the Advantage represents by what amount should this increase or decrease be made. If we consider the last equation from the derivation above, then the Equation 2.30 will do all the updates for each state-action pair

at every timestamp t of the trajectory:

$$\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(A_t|S_t) G_t \tag{2.30}$$

Why to optimize log-probability In Gradient estimation above we optimized the log-probability, because the gradient of log-probability is generally well-scaled. Since the normal probabilities range between 0 and 1, so the range of values to be optimized by the optimizer is limited and small, due to which the optimizer can run into numerical issues while optimizing with small values. Instead, log-probabilities have a larger "dynamic range" which ranges from $(-\infty, 0)$, making the computation with respect to log-probabilities easier for the optimizer to evaluate.

2.3.3 Comparison between methods

Policy based methods are good at convergence and are effective in learning stochastic policies. However, evaluating a policy may be inefficient and slower in case of high-variance in rewards.When using Value based methods the use of *max* is extremely aggressive and in just one step the agent is kind of *immediately* pushed to currently best policy whereas Policy gradient method takes a *small step* in that direction by just sort of smoothing the update in the correct direction which make Policy based methods more stable but also sometimes less efficient. Value-based RL learns a near to deterministic policy like epsilon-greedy wherein the policy just deterministically states that in a particular state this action is suitable to be taken whereas in Policy based methods which are based on stochastic policies gives us the distribution of probabilities for the actions to choose from instead of just acting greedily at every step ending up taking very long time reaching our goal in *some* cases.

2.4 Summary

This chapter illustrated how both the RL based methods namely value and policy methods work and we also understood the mathematical insights of the how the algorithms work. In Chapter 4 we will further illustrate the implementation of two algorithms which are based on these methods namely *Deep Q Network with experience replay* and Monte-Carlo Based *Reinforce Policy Gradient* method.

CHAPTER 3

EXPERIMENTAL SETUP

The agents when implemented using a specific algorithm need an experimental setup and need to know about the configurations of the environment on which the agent has to take certain actions. Simulator like OpenAI's Gym prove useful in these situations since it offers a wide range of environments to conduct experiments on different Reinforcement Learning Problems.As per our problem statement, we will explain the details about two environment namely Cartpole-v1 and LunarLander-v2 in this chapter used as experimental base for the implementation of agents in Chapter 4. We will also elaborate the reason for choosing OpenAI gym as our Simulator in the concluding Section 3.5.

Code snippets in this chapter use Python Programming language.So, intermediate level of cognizance is required in Python is required.

3.1 OpenAI's Gym

Gym [24] is a toolkit developed by OpenAI which is an AI based research laboratory founded by Elon Musk in 2015.The toolkit allows to develop and compare Reinforcement Learning agents.The agents can be implemented independently, using any of the preferrable Reinforcement learning algorithms and can be trained on any of the test problems or environments available in the toolkit.The toolkit is compatible with computational libraries like TensorFlow , Pytorch and Sklearn. The test problems or environments have a shared interface, which allow us to built plug and play agents[1].

Benefits When using such toolkits the cost of setting up physical environment can be avoided.Apart from that the experiments when conducted on in these tool-kits use simulation environments and they eventually don't harm the human in anyway whatsoever hence rigorous amount of testing can also be conducted.

3.2 Overview of Gym Environments

Gym toolkit [21] as described on there website's homepage [25] comes up with different *categories* of the environments which include :

- **Algorithms** : These are environments in which the agent learns to imitate computations. *Reverse-v0* is one such environment where the goal is to reverse a sequence of input symbols.
- **Atari Games** : These are a set of Atari games in which the agent tries to achieve high score by playing the games. These include games like *AirRaid-v0 , Pong-v0*
- **Box2D** : These are a set of continuous control tasks rendered in 2D rectangle or a boxed simulator. Our *LunarLander-v2* environment lies in this category.
- **Classic control** : These are reinforcement learning problems which are based on classic control theory literature. *Cartpole-v1* is one of the examples of of classical control theroy problem.
- **Robotic** : These are environments replicate robots which have a predefined task to accomplish like a Robotic hands which has to reach a given target orientation for the block.

[1]Agents built with different algorthims may exhibit different behaviour in different environments

All these different categories of the environments exhibit different behaviour because of differences in definition of the problem which include set of possible actions , rewards functions and many other minute details. However, Gym attempts to overcome this challenge of difference in behaviour and has a set of built in and well defined set of functions and objects that manifest the same functionality and allow painless interaction with most of environment.

Create an instance of environment To create an instance of an environment we use *make()* function after importing gym package. We usually store the reference to the environment in a variable such as *env*. In the following code snippet we have created an instance of `LunarLander-v2` environment

```
import gym
env=gym.make('LunarLander-v2')
```

If we want to use any other environments we replace "LunarLander-v2" with something like Cartpole-v1 or Pong-v0.

Rendering an environment To render the steps or the states of an environment at different time steps we use *render()* function. Say we want to render 500 steps of the our LunarLander-v2 environment while taking random actions at every step.

```
import gym
env = gym.make('LunarLander-v2')
# reset the environment to one of the starting states
env.reset()
for step in range(500):
    # render the state on the pop up window
    env.render()
    # sample a random action and execute it on the environment
    env.step(env.action_space.sample())
env.close()
```

The the loop which iterates over certain number of steps till the episode ends is called the `Step-Loop`.One would see a pop up window rendering the LunarLander-v2 problem once we execute the code above.

Observing the environment To observe the state of the environment *step()* [24] function is invoked on the instance of the environment as shown below. It takes the action to be executed on the environment as input and returns four environment-specfic values as a tuple explained in detail below

```
state , reward , done , info= env.step(action)
```

- **state** is an `object` which represents the current observation or the state of the environment. The state could be in the form of raw pixel values from

a game like Pong or in the form of Joint-angles and velocities in case of a robot.

- **reward** is a `floating point value` which represents the amount of reward received by taking an action "action" on the environment.The scale of the reward received is quite dependent on the environment being observed by the agent.

- **done** is a `boolean` value indicating if the episode has terminated or not. Say in case of LunarLander-v2 if the lander crashes or lands successfully on the landing pad then the episode has terminated.After episode terminates its usually the time to reset the environment carried out using *reset()* function

- **info** is `python dictionary` which holds diagnostic information useful for debugging. Officially it does not contain any information which can prove worthy to train the agent though.

Spaces Every environment has a set of feasible actions which can be taken at every time steps and a possibly very big set of observations or states for the environment to be in at a specific time step. These set of feasible actions or states are termed as `Spaces`.Hence every environment has an action space and a state or observation space.And in Gym these spaces are the attributes of every instance of an environment variable (env) and end with `_space`

```
import gym
env = gym.make('LunarLander-v2')
print(env.action_space)
#### Discrete(4)
print(env.observation_space)
#### Box(-inf, inf, (8,), float32)
```

It is visible in the case of LunarLander-v2 environment that it has discrete action space of 4 , discrete means that there are four integral actions [0,1,2,3] that the agent can choose.The Box space constitute an n-dimensionalbox having bounds between $(-\infty, \infty)$ an observation of size 8 represented with `float32` data type.

Loop of Interaction In Chapter 1 we discussed about the theoretical perspective of how the classical `Loop of Interaction` works here comes the time to actually see it in action.The two primary things that notations used in Loop of interaction are the `Episodic-Loop` and `Step-Loop`.Episodic-loop iterates over an episode and Step-Loop iterates over the steps that can executed within an episode. Code snippet below replicates the Loop of Interaction.It runs for five episodes wherein the episode starts by resetting the environment to an initial state of the environment. Then we initialize done to False which is equivalent to

saying that we not in the terminal state yet.We start the Step-Loop where we iterate till the terminal state. And at each `step` we print the state configuration, sample a random action, take that action on the environment.In turn the *step()* function returns a new state and the reward from the environment.

```
env=gym.make("LunarLander-v2")
# Episodic-loop
for i_episode in range(5):
    state = env.reset()
    done=False
    # Keep iterating until the terminal state
    # Step-Loop
    while not done:
        print(state)
        # sample a random action from action_space
        action = env.action_space.sample()
        # execute that action on the environment
        state, reward, done, info = env.step(action)
env.close()
```

3.3 Experiments

In Section 1.4.1 we gave an overview of how experiments were carried out. In the implementation section we implemented the two agents namely `DQN_Agent` and `PG_Agent` based on the two algorithms.For the first part of the experimentation both agents where trained and tested on Cartpole-v1 environment and for second part the same was repeated for LunarLander-v2 environment. Both the environments are offered by gym toolkit.The choice of these environments is based on the complexity and differences in the problem definition of these two environments.However, both the problems are based on *Optimal control theory* [4].

Later for the evaluation of the trained agents the performance metrics mentioned in the Section 1.6 were used.And the results of which are described in the Chapter 5.

3.3.1 Experiment 1

Our first experiment was carried out on the `CartPole-v1` environment.This environment is based on optimal control theory problem and is also described by Barto, Sutton, and Anderson in there popular book [28].As per gym's problem definition [22] there exists a pole which is attached by a joint to a cart. The cart is free to move along a friction-less track [Fig 3.1]. The pole can balanced by applying a force of magnitude 1 on the cart.This force can be applied in any of the directions left or right.In the starting state of the environment, the pole stands vertically-straight and the cart is in the center of the display. Agent has

to prevent the pole from falling.An epsiode of the environment terminates if the the pole inclines more that 15° towards right or left *or* the cart is more than 2.4 units away from the center.The agent receives a reward of +1 at every time step if it is able to successfully balance the pole and remain within range of 2.4 units away from the center.There exist a *threshold* of 500 steps which is the maximum number of steps the agent is allowed to execute. One more thing to notice here is that the since at every step the agents receives +1 point therefore the number of steps and cumulative reward points for the agent would be the same in this environment.

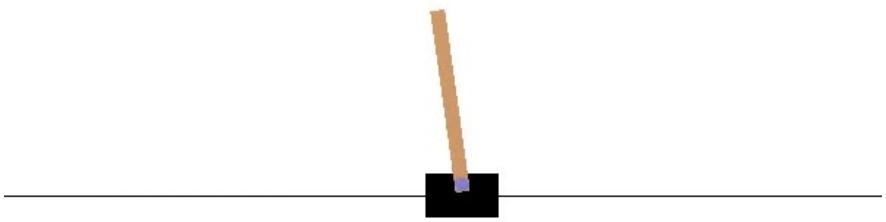

Figure 3.1: CartPole-v1 Environment

Observation Space of the environment is represented as a `Box` object of size 4 which consists of the below described parameters :

Table 3.1: Observation space of CartPole-v1

index	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-15 deg	15 deg
3	Pole Angular Velocity	-Inf	Inf

Action Space of the environment is represented as a `Discrete` object of size 2 which consists of the below described parameters :

Table 3.2: Action space of CartPole-v1

index	Action
0	Push cart to the left
1	Push cart to the right

So according to the action space , a force of magnitude 1 can push the cart either towards the left or towards the right at any time step.And the respective indices indicate that in order to apply a force on the cart from the left side action 0 has to chosen by the agent and 1 otherwise.

Solved For any reinforcement learning problem there is a phase when agent is sufficiently trained and it is able to solve the problem no matter where it starts from.In the case of CartPole-v1 ,the environment is considered to be solved when the agent is able to receive 500 reward points on average over the last 100 episodes.

3.3.2 Experiment 2

Our next experiment was carried out on the `LunarLander-v2` environment.It is also optimal control problem where the vehicle trajectory has to be optimized with optimal control of the vehicle. According to the problem definition [23] a vehicle having three engines (left , right , main) starts from the top of the display and its goal is to land softly and fuel-efficiently on the landing pad at the coordinates (0,0).The coordinates of the vehicle on the display are indicated by the first two values in the observation space.If the vehicle or the agent approaches the landing pad at coordinates (0,0) then it obtains rewards, else it receives negative rewards from the environment.The rewards when the vehicle starts from the top of the screen till it rests successfully on the landing pad ranges from 100 to 140 points.

An episode terminates if the vehicle crashes in which case the agent receives -100 points or it lands successfully on the landing pad receiving +100 points. If the leg of the vehicle makes a contact with the ground then the agent receives additional +10 points.However, firing any of the engines costs -0.3 points per time step because fuel-efficiency is also a concern in this problem though the vehicle has infinite fuel.Landing outside the landing pad is possible in which case agent is again penalized appropriately.

Vehicle's Coordinate System The coordinates [14] of the vehicle are taken relative to the landing pad. Since the landing pad is considered at coordinates

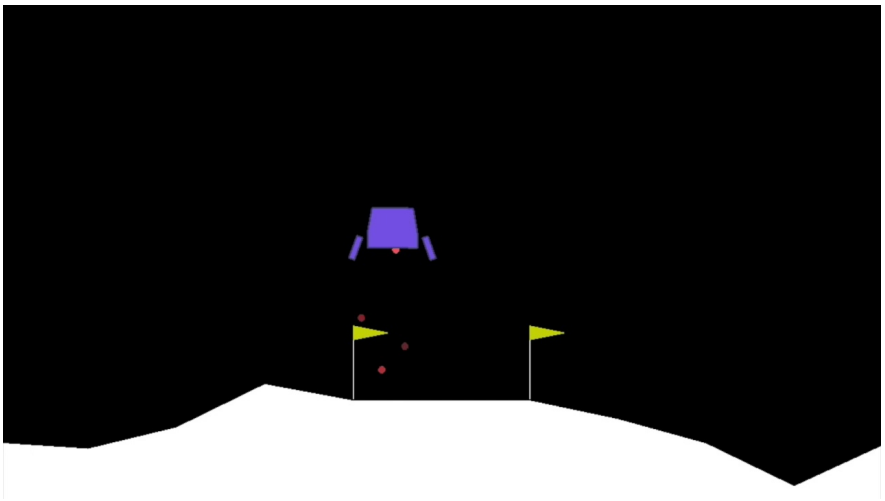

Figure 3.2: LunarLander-v2 Environment

(0,0) , if the vehicle is to the left of the landing pad then its x coordinate is negative and positive otherwise.Similarly if the vehicle is above the landing pad then its y coordinates are positive and negative if the vehicle is below the landing pad.

Observation Space of the environment is represented as a `Box` object of size 8 which consists of the below described attributes [14] :

Table 3.3: Observation space of LunarLander-v2

index	Observation
0	x coordinate of the vehicle
1	y coordinate of the vehicle
2	v_x, Horizontal velocity of the vehicle
3	v_y, Vertical velocity of the vehicle
4	θ, Orientation of the vehicle in space
5	v_θ, Angular velocity of the vehicle
6	Left leg touching the ground (Boolean)
7	Right leg touching the ground (Boolean)

Action Space of the environment is represented as a `Discrete` object of size 4 which consists of the below described parameters :

Table 3.4: Action space of LunarLander-v2

index	Action
0	Do nothing
1	Fire left engine
2	Fire main engine
3	Fire right engine

The vehicle may rotate clockwise or counter-clockwise when left or right engines are triggered respectively making it difficult for the vehicle to stabilize. As per the problem there exist a *threshold* of 1000 steps which is the maximum number of steps the agent is allowed to execute in any episode otherwise the episode would terminates.

Solved The problem is considered to be solved if the agent is able to consecutively receive 200 reward points on average over last 100 episodes.

3.4 Testing on multiple instances

When the agent is trained using an RL algorithm it should be able to generalize well the different instances of the same environment, because an environment could have different formations. These formations could be referred to as instances of the environment. To generate different instances of an environment, we may use *env.seed(value)* method available in the Gym-toolkit, wherein the environment may be initialized with different seed `value` in turn generating an instance of the environment. In diagram below one can see the two instances of `LunarLander-v2` environment having different seed values and neglecting the vehicle in the environment, one may notice that the formation of the environments is different.

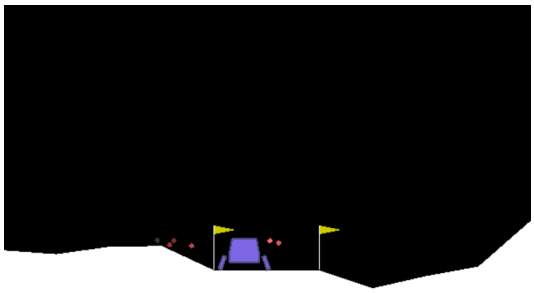

Figure 3.3: Instance: seed value 42

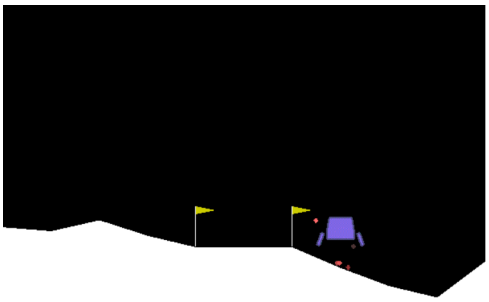

Figure 3.4: Instance: seed value 450

3.5 Gym as Framework

The definition of Reinforcement learning problems can vary not only based on the category of the problem but also based on its practical applications and hence even slight change in the definition of problem, for instance changing the action space from discrete to continuous can significantly effect the complexity of the problem.As a result the environment configurations like observation space and action space differ as well making them hard to setup and use.Gym attempts to

overcome these challenges by providing shareable interfaces , proper well defined set of objects and methods.Hence, it is popular framework which is used as a benchmark to test and compare different Reinforcement learning algorithms.

3.6 Summary

In this Chapter we discussed about the OpenAI gym toolkit , our experimental setups and how Gym can be used for the implementation purposes. We also understood the configuration setting of the two environment and understood in detail about the observation and state space of the environments.In the upcoming Chapter we will see how the agents were implemented using these experimental setups.

CHAPTER 4

IMPLEMENTATION

This section deals with the implementation of agents based on the two algorithms, differences between which we have discussed in Chapter 2

- Section 4.1 describes which tools and packages were used for the implementation.
- Section 4.2 and Section 4.3 guides one with the organisation of the `Github` repository and how one can spin up a docker image to start the `Jupyter` notebooks
- Section 4.4 will elaborate how we implemented Deep Q learning agent using the Deep Q-Learning with Experience-Replay algorithm.
- Section 4.5 will delineate how we implemented Policy Gradient agent using the Reinforce algorithm which is based on the Monte Carlo Method.

4.1 Tools and Packages

The implementation of the agents was carried out using *python* programming language and following libraries with respective use cases were included as part of the agent's implementation :

- *jupyter notebook* : This platform is used for interactive data science and scientific computing. It offers support for code, narrative text and visualizations
- *gym* : This library offers the two simulation environments namely Cartpole-v1 and LunarLander-v2
- *numpy* : Used in mathematical Calculations and other operations.
- *pandas* : Used for Booking-Keeping
- *tensorflow*: Used for Creating and Training Deep Neural Networks
- *matplotlib* : Used for plotting the results accumulated while training and testing of the agents.
- *operating system libraries* : Libraries like swig , opengl are Operating system specific libraries which are used in rendering the output of the agents within the jupyter notebooks.
- *docker* : We implemented a docker image which can be pulled online through docker desktop.The image comes in handy to run the Jupyter notebooks.
- *github* : If one want to have a glance on how the agents were implemented and one may check the source code of the project in our Github repository

4.2 Structure of the Docker Image

Our Docker Image is organized into the following directories :

- **Experiment 1** contains the jupyter notebooks which describe the results of training and testing phases of both the `DQN_Agent` and `PG_Agent` on Cartpole-v1 environment.
- **Experiment 2** contains the jupyter notebooks which describe the results of training and testing phases of both the `DQN_Agent` and `PG_Agent` on LunarLander-v2 environment.

- **agents** contains two .py files which hold the implementation source code of the agents namely the `DQN_Agent` and `PG_Agent` respectively.This directory also contains a `Random_Agent`
- **helpers** contains .py files which are used for plotting the results and rendering the outputs of the agents as animations within the jupyter notebooks.
- **DQN_trained_models** contain the models which have already been trained by the `DQN_Agent`
- **PG_trained_models** contain the models which have already been trained by the `PG_Agent`

4.3 How to start Jupyter Notebooks

Before one can start running the Jupyter notebook Docker desktop has to be installed on the local machine to pull the already implemented docker image.

- You can follow this link to download and install docker desktop if it is not already installed.
- Once installed pull the docker image using the command mentioned below on `Windows Powershell` or `Terminal`(on Linux):

```
docker pull riturajsingh2015/deep_rl:v2.0
```

- After pulling the image spin a docker container using the following command on port 8888

```
docker run -d -p 8888:8888 riturajsingh2015/deep_rl:v2.0
```

- When the container starts, open your browser and type *localhost:8888* to see Jupyter Interface with all the directories as listed in Section 4.2. In case you need token as password to login copy the token from the container's logs.

4.4 Deep Q-learning with Experience Replay

When training a DQN one of the important thing that network requires is sufficient amount of training data.But even if we train the network with a lot of data situations may arise where the network does not converge to the optimal value

function and the weights of the network may oscillate because of correlation — sequential training data — between actions and states.

To reduce this problem of correlation we use a replay buffer of fixed size to store the experiences as tuples

$$(S_t, A_t, R_t, S_{t+1}, done_t) \tag{4.1}$$

where S_t is the current state and A_t is action taken and R_t is the reward received from the environment at time t and S_{t+1} is the next state once action A_t is taken on the environment. $done_t$ will help the agent determine whether it is in a `terminal` or a `non-terminal` state.These experience tuples are gradually added to the replay buffer while the agent is interacting with the Environment. So,we store the experiences in the Replay Memory D instead of training the agent just on the latest experiences. Then the agent randomly samples these experiences from the buffer preventing the action values from oscillating or diverging catastrophically. Hence, the name *Deep Q-learning with Experience Replay* [20]

> In section 2.3.1 we have discussed that the DNN which is to be trained needs to take state-action pair as input and output the estimated Q-values. However, in Practice it is more efficient to use a DNN which takes a state and outputs one Q-value for each of the feasible actions.

The DQN_Agent [Fig 1.4] was implemented based on the Pseudo-Code described in the Algorithm 1

Lets walk through the Pseudo Code

- *Line 1* : D is the replay buffer having a certain capacity C
- *Line 2* : Q is the DQN initialized with random weights
- *Line 3-20* : Is the `Episodic-loop` wherein we train the agent for a specific number of episodes.
- *Line 4* : We reset the environment to a starting state.
- *Line 5-19* : Is the `Step-loop` which iterates over time steps t until the episode terminates.
- *Line 6-7* : Based on the ϵ-greedy method and the value of ϵ we either select a random action(`explore`) or select an action from among the set of feasible actions the action which has the highest Q-value(`exploit`)

Algorithm 1 Deep Q-learning with Experience Replay

1: Initialize Replay Memory D with Capacity C
2: Initialize the action-value function Q as Deep-Q Network
3: **for** $e = 1, 2, \ldots, M$ **do**
4: $S_t \leftarrow S_1$ ▷ Make starting state the current state
5: **for** $t = 1, 2, \ldots, T$ **do**
6: with Probability ϵ select a random action A_t
7: **or** select action $A_t \leftarrow \max_a Q^*(S_t)$
8: Agent receives reward R_t and the next state S_{t+1} for action A_t,
9: Store the experience tuple $(S_t, A_t, R_t, S_{t+1}, done_t)$ in D
10: *****Learning-Phase Starts*****
11: Sample a mini-batch of size B transitions from D
12: **for** *every transition* $(S_j, A_j, R_j, S_{j+1}, done_j)$ in the mini-batch **do**
13: $y_j \leftarrow R_j$ **if** $done_j$ is True
14: $y_j \leftarrow R_j + \gamma . \max_{a \in A} Q(S_{j+1})$ **if** $done_j$ is False
15: Perform Gradient descent step $(y_j - Q(S_j))^2$
16: *where* y_j = target values and $Q(S_j)$ = Predicted values
17: **end for**
18: *****Learning-Phase Ends*****
19: **end for**
20: **end for**

- *Line 8* : Once the action A_t is chosen by the agent. Then,the agent receives a reward R_t and observes the next state S_{t+1} after taking this chosen action.
- *Line 9* : Stores the experience tuple in replay memory D
- *Line 10-18* : `Learning-phase` of the agent
- *Line 11* : The agent samples a mini-batch of size B transitions from the Memory D
- *Line 13-14* : For every transition in this sampled mini-batch if state S_j is a terminal state then the estimated Q-value(y_i) is immediate reward otherwise it is discounted future reward.
- *Line 15-16* : We perform the *Gradient Descent Step* wherein we propagate the mean squared error between the target values y_j and predicted values $Q(S_j)$ back into the action-value network Q to train the network.

4.4.1 Python implementation of DQN_Agent

The DQN_Agent we implemented is present in the `agents` directory in our Github repository.As far as implementation of DQN_Agent file is consisdered it has a `ReplayMemory` and a `DQN_Agent` class. The implementation took Algorithm 1 and its notation as reference.

ReplayMemory class : Deals with the implementation of the Experience Replay Memory D. It is initialized to a maximum capacity of C and consists of a method(`save_transition`) to store an experience in the memory and another method(`sample_mini_batch`) to sample transitions of batch-size B from the Memory D.

DQN_Agent class : Takes care of the implementation of the agent.It is initialized with a number of agent specific hyper parameters like the environment name , learning rate(α), sol_th, discount factor(γ) , batch-size(B), capacity of Replay Memory(C), ϵ specific initialization values consisting of the starting and end values eps_start,eps_end and ϵ-decay rate denoted by eps_dec. sol_th is the threshold limit of cumulative rewards over the last 100 episodes of training which the agent would aim to reach.Our action-value Q-function is a deep Q network implemented using tensorflow's Keras library as follows :

```
self.Q = keras.Sequential([
    keras.layers.Dense(self.layer1_size, activation='relu'),
    keras.layers.Dense(self.layer2_size, activation='relu'),
    keras.layers.Dense(self.n_actions, activation=None)])

    self.Q.compile(optimizer=keras.optimizers.Adam(learning_rate=alpha),
loss='mean_squared_error')
```

Listing 4.1: Deep Q Network

The DQN consists of two hidden layers namely `layer1` and `layer2` with corresponding neurons per layer (layer1_size,layer2_size) and the hidden layers use `Relu` as activation function and as the output layer it has n action-outputs as per the action space of the environment which are just n *Q-values* for each of the n actions that the agent can take given a state. Apart from that the DQN is compiled with *alpha* as the learning rate of the network and with mean squared error as loss function to reduce the gradient loss between the target and predicted Q-values. `DQN_Agent` class consists of five main methods apart from some other complimentary methods.

- `save_experience` : saves the experience tuple in Replay Memory D
- `epsilon_greedy_action` : the action chosen as per ϵ-greedy strategy

```
    def epsilon_greedy_action(self, state):
        if np.random.random() < self.epsilon:
            action = np.random.choice(self.action_space)
        else:
            state__ = np.array([state])
            actions = self.Q.predict(state__)
            action = np.argmax(actions)
        return action
```

Initially the value of `epsilon` is 1.0 and it decreases at a specified decay rate. So when the value of epsilon is high the agent mostly `explores` the environment by taking random actions from among the available set of actions in *lines 2-3*. Later in *lines 5-7* when the value of epsilon decrease the agent `exploits` the information learned from the environment through the Q-Network wherein the network chooses the action with highest Q-value and to choose the action corresponding to the highest Q-value *np.argmax* function is utilized.

- `update_epsilon` : updates the ϵ value as per the ϵ decay rate until it reaches its minimum bound(`eps_min`)

- `learn` : This method replicates the learning phase in Pseudo Code wherein we sample a mini-batch of size B from Replay Memory D using the **sample_mini_batch** method in Replay Memory class.

```
S_js, A_js, R_js, S_nexts, done_js = \
                self.D.sample_mini_batch(self.B)
```

Listing 4.2: Sample mini-batch

Once we sample the transitions from the ReplayMemory D we make use of our DQN to make predictions on the current states S_js and the next states S_nexts

```
        Q_current = self.Q.predict(S_js)
        Q_next = self.Q.predict(S_nexts)
```

Listing 4.3: Make Predictions using Q-Network

Now before we start training the DQN we need to calculate the target Q-values based on current Q-values(`Q_current`) for which create a copy of target Q-values(`Q_Target`) which is initially equal to the current Q-values (`Q_current`)

```
        Q_Target = np.copy(Q_current)
```

Listing 4.4: Create a copy for Q-Target Values

Then as per *Line 13-14* in our Pseudo Code we update the `Q_Target` values based on the whether the next states were terminal states or not. To update the respective target Q-values we index into the target Q-values by creating batch of index values based on the size of the mini-batch which is B. Since we assume that the agent will be taking actions optimally, the agent will

only be keeping the maximum Q-Value for each next state `Q_next` using *np.max*. Next, we use Equation 2.18 to compute the target Q-Value for each experience's state-action pair. And for determining whether the next state S_next is terminal or not `done` values comes in handy and hence the update rule is as follows :

```
batch_index = np.arange(self.B, dtype=np.int32)
Q_Target[batch_index,A_js] = R_js + \
self.gamma *np.max(Q_next, axis=1)*done_js

```

Listing 4.5: Q-Target Values update rule

where gamma is the discount factor (γ).The last and the final step involved in the learning phase is to train the DQN on the batch of estimated target Q-values (`Q_Target`)

```
loss=self.Q.train_on_batch(S_js, Q_Target)

```

Listing 4.6: Perform Gradient Descent updates

`train_on_batch` method in Keras Model's API performs a single gradient update on a single batch of data , minimizing the loss with regard to the Q-Network or model's trainable variables for the Q-Network (Q) and returns calculated `loss` value on every update.

- `train_multiple_episodes` : This method trains the agent for specific number of episodes or till the agent is able to reach the threshold limit sol_th. The methods takes number of episodes as argument

```
def train_multiple_episodes(self,num_episodes=500):
    for ep in range(num_episodes):
        done = False
        ep_reward = 0
        #Reset the environment to starting state
        S = self.env.reset()
        ep_steps=0
        while not done:
            # choose greedy action
            A = self.epsilon_greedy_action(S)
            # execute action A on environment
            S_new, R, done, _ = self.env.step(A)

            ep_reward += R
            ep_steps+=1
            # save experience tuple in Replay Memory D
            self.save_experience(S, A, R, S_new,\
            int(done))
            #make transition to new state
            S = S_new
            # train the agent
            self.learn()
```

Listing 4.7: Trains the agent for multiple episodes

Here from *Line 2-23* we replicate the `Episodic-loop` as explained in the Pseudo Code. So we start by resetting the environment to one of the starting states. *Line 8-22* is the `Step-loop` which iterates until done is False or the episode ends. In the `Step-loop` the agent would choose an action A following the ϵ-greedy action selection method in *Line 10*. Then the agent executes that chosen action A in *Line 12* on the environment using the *step*() method made available through the gym framework.In *Line 17* the agent saves the experience tuple in the Replay Memory D. At the end in Lines 20-22 we make transition in the new state S_new and train the agent using the learn method.

Apart from these five primary methods which are part of the `DQN_Agent`.There are some other utility methods for book-keeping , saving and utilizing trained models in the `DQN_Agent` file as well.This brings us to the end of the implementation of `DQN_Agent`

4.5 Reinforce Policy Gradient

In this Section we will be discussing and implementing the Reinforce Policy Gradient agent which is based on the Policy Based approach. Section 2.3.2 discussed the derivation and the background of Reinforce Policy Gradient Method.The agent we implemented using this approach was named PG_Agent . Since reinforce policy gradient is a type of Monte-Carlo technique , the agent usually starts of by sampling a trajectory τ by following a policy π_θ , where the policy is a neural network named the Policy Network with parameters θ. We generate the trajectory by adding an experience into the trajectory, where the experience is a tuple consisting of the current state S_t, action A_t and reward received R_{t+1} at timestamp t denoted by:

$$(S_t, A_t, R_{t+1}) \tag{4.2}$$

Trajectory τ acts as buffer storing the experiences. The implementation of `PG_Agent` [Fig 1.4] is based on the Pseudo-Code described in the Algorithm 2. Lets walk through the Pseudo Code

- *Line 1-3* : We initialize the Policy $\pi_\theta(a|s)$ or the Policy Network with random weight θ , having a learning rate α .The policy will take the state s of the environment as input and outputs the *action-probability* distribution

- *Line 4-15* : Is the `Episodic-loop` where we generate a trajectory τ by following the policy and as per the trajectory we estimate the gradient to eventually pushing the policy in the right direction

Algorithm 2 Reinforce : Monte-Carlo Policy-Gradient Control (episodic)

1: Input: a differentiable policy parameterization $\pi(a|s)$
2: Algorithm parameter: step size $\alpha > 0$
3: Randomly initialize the policy parameter θ
4: *****Loop forever (for each episode)*****
5: **for** $e = 1, 2, \ldots,$ *episode* **do**
6: Follow policy $\pi(a|s)$ to collect trajectory τ
7: *where* $\tau = S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
8: Estimate the return of the trajectory τ : $R(\tau) = (G_0, G_1, G_2, \ldots, G_{T-1})$
9: *where* G_k is the expected return for the transition k
10: $G_k \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
11: Use trajectory τ to estimate the gradient $\nabla_\theta J(\theta)$
12: $\sum_{t=0}^{T-1} \nabla_\theta \ln \pi_\theta(a_t|s_t) G_t$
13: Update the weights of the policy $\pi(a|s)$ as per the gradient ascent
14: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
15: **end for**

- *Line 6-7* : In every episode, we generate the trajectory τ by following the Policy $\pi_\theta(a|s)$ or the Policy Network
- *Line 8-10* : We estimate the return of the trajectory $R(\tau)$ using the *action's advantage* G_k
- *Line 11-12* : Using the $R(\tau)$ we estimate the the gradient $\nabla_\theta J(\theta)$ for the policy $\pi_\theta(a|s)$
- *Line 13-14* : Then, based on the Gradient Ascent Update rule the policy parameters are tweaked so that probabilities of actions which increased the returns are encouraged and the probabilities of actions which decreased the returns are discouraged.

4.5.1 Python implementation of PG_Agent

The PG_Agent we implemented is present in the `agents` directory in our Github repository.As for the implementation of PG_Agent file is considered it consists of just one class the `PG_Agent`. The implementation took Algorithm 2 as reference.

PG_Agent class Takes care of the implementation of the agent.It is initialized with a number of agent specific hyper parameters like the environment name , learning rate(α), discount factor(γ) , trajectory(τ), policy-network(π). The

policy-network or the Policy π as per *Line 1* of the Pseudo-Code is a deep neural network implemented using tensorflow's Keras library, the weights θ of the Policy are randomly intialized with values which are sampled randomly from a normal distribution as per *Line 3-5* in the Code Snippet here and *Line 2* in the Code-snippet initializes the input layer:

```
self.pi = keras.Sequential([
keras.Input(shape=(self.state_space,), name="inputs"),
keras.layers.Dense(layer1_size, activation='relu', kernel_initializer=keras.
initializers.he_normal(),autocast=False,

keras.layers.Dense(layer2_size, activation='relu', kernel_initializer=keras.
initializers.he_normal()),

keras.layers.Dense(self.action_space, activation='softmax')])
```

Listing 4.8: Policy-Network π_θ

The Policy-Network has of two hidden layers namely `layer1` and `layer2` with corresponding neurons per layer (layer1_size,layer2_size) and the hidden layers use `Relu` as activation function and the output layer has n action-probability-distributions as per the action space of the environment. Output-layer uses soft-max activation function that predicts a multinomial probability distribution.These action-probability-distributions are probability-values for each of the n-actions that the agent can take in a specific state.These probabilities are tweaked through policy's parameters θ, wherein we encourage the probabilities which result in higher returns and discourage ones which result in lower returns.

`PG_Agent` class consists of five main methods apart from some other complimentary methods.

- `add_experience_to_trajectory` : Adds experience tuple (S, R, A) into the trajectory τ - tau

```
    def add_experience_to_trajectory(self,S,R,A):
        self.tau['states'].append(S)
        self.tau['rewards'].append(R)
        self.tau['actions'].append(A)
```

Listing 4.9: Add experience tuple to τ

- `empty_the_trajectory` : Vacates the trajectory τ containing the experience related information

```
    def empty_the_trajectory(self):
        self.tau['states'] = []
        self.tau['actions'] = []
        self.tau['rewards'] = []
```

Listing 4.10: Emptys τ

- `get_discounted_rewards` : This function takes the rewards of the trajectory as input and calculates discounted rewards — as per the *Line 8-10* of the Algorithm 2 — based on the discount factor (γ) - gamma. The function also standardize the discounted rewards G

```
        def get_discounted_rewards(self,rewards):
            reward_sum = 0
            G = []
            for reward in rewards[::-1]:
                reward_sum = reward + self.gamma * reward_sum
                G.append(reward_sum)
            G.reverse()
            G = np.array(G)
            # standardise the rewards
            G -= np.mean(G)
            G /= np.std(G)
            return G
```

Listing 4.11: Estimates action-advantage G

- `choose_action` : This function receive a state of the environment and outputs an action based on the action-probability-distribution of the policy π_θ - pi

```
            def choose_action(self, state):
                softmax_out =self.pi(state.reshape((1, -1)))
                selected_action = np.random.choice(self.action_space, p=
    softmax_out.numpy()[0])
                return selected_action
```

Listing 4.12: Chooses an action based on π_θ

- `learn` : The learn function is invoked at the end of every episode , wherein the states , actions , rewards of the trajectory are collected , then the discounted rewards G are calculated based on the rewards. Finally in *Line 4* of the Code Snippet, the policy network π_θ is updated based on the states, actions and discounted rewards.

```
    def learn(self):
        states, actions, rewards = self.tau['states'],self.tau['actions'],
    self.tau['rewards']
        G=self.get_discounted_rewards(rewards)
        loss=self.update_policy_network(np.vstack(states) ,actions,G)
        return loss
```

Listing 4.13: The learn method

- `pg_loss` : This method estimates the objective function $J(\theta)$, based on the Eq 2.30 , where we weight the log-probabilities of the actions chosen with the action-advantages G

```
    def pg_loss(self,aprobs, actions, G):
        indexes = tf.range(0, tf.shape(aprobs)[0]) * tf.shape(aprobs)[1] +
    actions
        responsible_outputs = tf.gather(tf.reshape(aprobs, [-1]), indexes)
        loss = -tf.reduce_mean(tf.math.log(responsible_outputs) * G)
        return loss
```

Listing 4.14: Estimate the objective function $J(\theta)$

- `update_policy_network` : This function updates the weights of the policy-network π_θ by estimating the gradients $\nabla_\theta J(\theta)$ of the loss function `pg_loss`

```
    def update_policy_network(self,states,actions,G):
        with tf.GradientTape() as tape:
            aprobs = self.pi(states)
            loss = self.pg_loss(aprobs, actions, G)
        gradients = tape.gradient(loss, self.pi.trainable_variables)
        self.optimizer.apply_gradients(zip(gradients, self.pi.
    trainable_variables))
        return loss
```

Listing 4.15: Update θ of π_θ by estimating $\nabla_\theta J(\theta)$

- `train_multiple_episodes` : This method trains the agent for specific number of episodes or till the agent is able to reach the threshold limit sol_th. The methods takes number of episodes as argument

```
    def train_multiple_episodes(self,num_episodes=500):
        for ep in range(num_episodes):
            ep_reward = 0
            ep_steps=0
            S = self.env.reset()
            self.empty_the_trajectory()
            done =False
            while not done:
                A = self.choose_action(S)
                S_new, R, done, _ = self.env.step(A)
                #store experience in the trajectory - tau
                self.add_experience_to_trajectory(S,R,A)
                #make transition to a new state
                S = S_new
                ep_reward += R
                ep_steps+=1
            # make policy - pi - learn at the end of episode
            loss=self.learn()....
```

Listing 4.16: Train over multiple episodes

Here from *Line 2-21* we replicate the `Episode-loop` as explained in the Pseudo Code. So, we start by resetting the environment in *Line 5* to one of the starting states. *Line 8-18* is the `Step-loop` which iterates until done is False or the episode ends. In the `Step-loop` the agent selected an action A by following the Policy π_θ. Then the agent executes that chosen action *A* in *Line 10* on the environment using the *step*() method. In *Line 12* the agent would add the experience tuple into the trajectory τ - tau. At the end in *Line 14* we make transition in the new state `S_new`. We make the agent learn from the trajectory using the learn method at the end of the episode

4.6 Summary

In this Chapter we described the implementation details of both the agents namely the `DQN_agent` and the `PG_Agent` with help of python code snippets.In the next Chapter we will elaborate the results found during the training and testing phases of the agents.And then make analysis on the results as per the performance metrics.

CHAPTER 5

RESULTS AND ANALYSIS

In this chapter the results which where found in the two experiments have been elaborated. Chapter is divided into these sections

- **Experiment 1** : Section contains, the results of both the agents `DQN_Agent` and `PG_Agent` for the Cartpole-v1 environment are described.
- **Experiment 2** : Section contains, the results of both the agents `DQN_Agent` and `PG_Agent` for the LunarLander-v2 environment are described.
- **Analysis** : Section describes the analysis of both the experiments based on the performance metrics discussed in the Section 1.3 and Section 1.6
- **Guidelines** : As per the analysis of the results the guidelines for using the two agents are discussed

5.1 Prerequisites for the Experiments

When training the agent : different factors influence the training phase of the agent like — the initial weights of the DNN, random sampling of batches for training (DQN_Agent),random sampling of the actions, variance in the reward functions and the starting state of the environment. Due to these influential factors *every trained model* is sort of `unique` and therefore we collected five different training models with different set of hyper-parameters for each agent in the experiments so that we can get a `generic overview` of the behaviour and the performance of the agent,this also helped in carrying out Hyper-parameter tuning. The performance metrics for comparison of agents performance were mentioned in Section 1.3 and Section 1.6.

5.1.1 Notations for the models

Models which we collected in our experiments had the following set of parameters[1] denoted by corresponding symbols.And L1 and L2 are number of neuron in these two layers respectively.

Table 5.1: Model specific parameters and Symbols

Parameter Name	Symbol
Batch-size	B
Discount factor	γ
Learning rate	α
Threshold Limit	sol_th
Episodes taken by the agent to train	episodes
Epsilon decay rate	eps_dec
Training time for the model	training_time
Neurons in layer_1 and layer_2	(L1 , L2)

During the training `sol_th` is the limit to which the average rewards accumulated by the agent in the last 100 episodes of the training would try to reach and this is the threshold value which the agent needs to reach in order to declare the environment as solved. The number of episodes taken by the agent to train (`episodes`) depend on `sol_th` because till the time the agent does not reach this value of threshold the environment cannot as termed as `solved`. We mentioned in Section 3.3.1 and Section 3.3.2 that the sol_th for Cartpole-v1 and LunarLander-v2 should be 500 and 200 reward points respectively. These threshold limits helps the agent reach a predefined efficiency level.

[1]Every DNN used in our experiments consisted of two hidden layers namely layer_1 and layer_2, because two hidden layers are mostly sufficient.

5.1.2 Training problems

There were other problems which we discovered while training the models which are elaborated with corresponding solutions below:

Why not train for fixed number of episodes ? Having a threshold limit like sol_th for training is beneficial because then the mean rewards accumulated by the agent over last 100 epsiodes of the training would be trending *upwards* and would try to reach this threshold limit, meaning the efficiency of the model would be better as compared to when the mean rewards are trending *downwards* Refer Figure 5.1 . So if we train for fixed number of episodes the model might be having any of upward or downward trending efficiency which is problematic. Therefore, we had to fix the threshold limit sol_th which the agent attempts to reach thereby being in an upward trending zone with an aim to increase its efficiency of prediction.

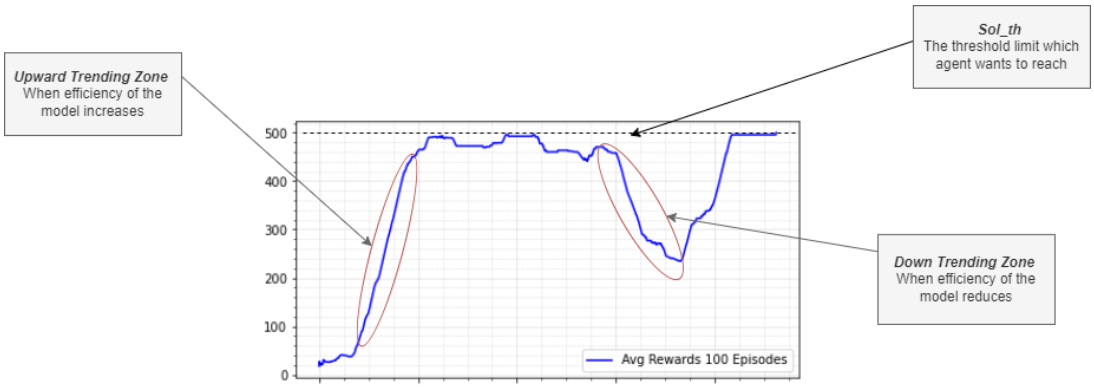

Figure 5.1: Efficiency zones of a model

Seed values When we train the agent on just a single instance of the environment by fixing its seed value — through *env.seed(value)* method — the agent was not able to generalize to the other instances of the environment and the performance on the test instances was really bad. So , better alternative was not to fix the seed value while training and let the agent explore all the possible experiences on various instances of an environment.

Hyper-parameters We made an attempt to determine the best set of hyper-parameters for models via manual tweaks by taking into consideration the hyper-parameters of the previously trained models in an experiment

5.1.3 Performance terms

Training phase : As described before ,in training phase we collected five different training models for each of the agent to get a generic view of how the agent was able to perform and behave.

Testing phase : For the performance testing we considered 16 test-instances or 16 different instances of the same environment. And for generating these instances of the environment we used the $env.seed(val)$ method of the gym-toolkit[Ref 3.4].

Best scenario : Based on the environment , we define best scenario as the scenario which yields the best results.In case of `Cartpole-v1` the number of steps for which the agent is allowed to balance the pole is not more than 500 and this is called the threshold limit. The longer the agent is able to balance the pole the better it is. The *best scenario* for the agent in any particular instance of this environment would be to balance the pole for 500 steps. Similarly, the *best scenario* for the agent in case of `LunarLander-v2`, is the one in which the vehicle lands perfectly inside the landing pad in minimum number of steps and this would be the best landing trajectory because the vehicle takes less time and consumes minimum fuel. The shorter the agent takes to land the vehicle successfully on the landing-pad the better it is.

In this regard we define a term `success_ratio` as below :

$$success\_ratio = \frac{Number\ of\ instances\ with\ best\ scenario}{16} \tag{5.1}$$

Other terms defined in this regard are the mean-reward-points and mean-steps taken across 16 test instances of the environment denoted by `mean_points` and `mean-steps`

$$mean\_points = \frac{sum\ of\ rewards\ for\ 16\ instances}{16} \tag{5.2}$$

$$mean\_steps = \frac{sum\ of\ steps\ for\ 16\ instances}{16} \tag{5.3}$$

5.2 Results from Experiment-1

In this section we will explain the results from the execution of `DQN_Agent` and `PG_Agent` on the `Cartpole-v1` environment. Cartpole-v1 is comparatively simple environment[Refer Sec 3.3.1] which does not have a variable reward function because at every step in an episode the agent receive +1 reward point.

5.2.1 Results of DQN_Agent

Trained Models The five models that we trained and collected using the `DQN_Agent` with there respective hyper-parameters are listed in the Table 5.2 below.

Table 5.2: Hyper-parameters of trained models

#	episodes	α	γ	B	(L1 , L2)	sol_th	time	eps_dec
Model-1	271	0.001	0.99	64	(32,16)	300	03 hr 10 min	0.0001
Model-2	486	0.001	0.99	64	(32,16)	350	13 hr 51 min	0.0001
Model-3	590	0.002	0.99	64	(32,16)	350	06 hr 31 min	0.0001
Model-4	219	0.001	0.99	64	(32,16)	300	01 hr 50 min	0.0001
Model-5	733	0.0003	0.99	64	(64,32)	450	09 hr 24 min	0.0001

Problems while training One of problem which we discovered while training the models with `DQN_Agent` is none of the models could reach the sol_th limit of 500, and the mean-reward for any model over the last 100 episodes(blue)[Example Model-5 in Figure 5.2] kept on fluctuating a lot and never reached 500. These fluctuations or bumps often occur because of phenomenon in neural networks called *Catastrophic Forgetting*[2] , due to which the cumulative rewards may decrease. To tackle this problem we reduced the threshold value sol_th and kept it to a limit of 300 or above as indicated in the Table 5.2.

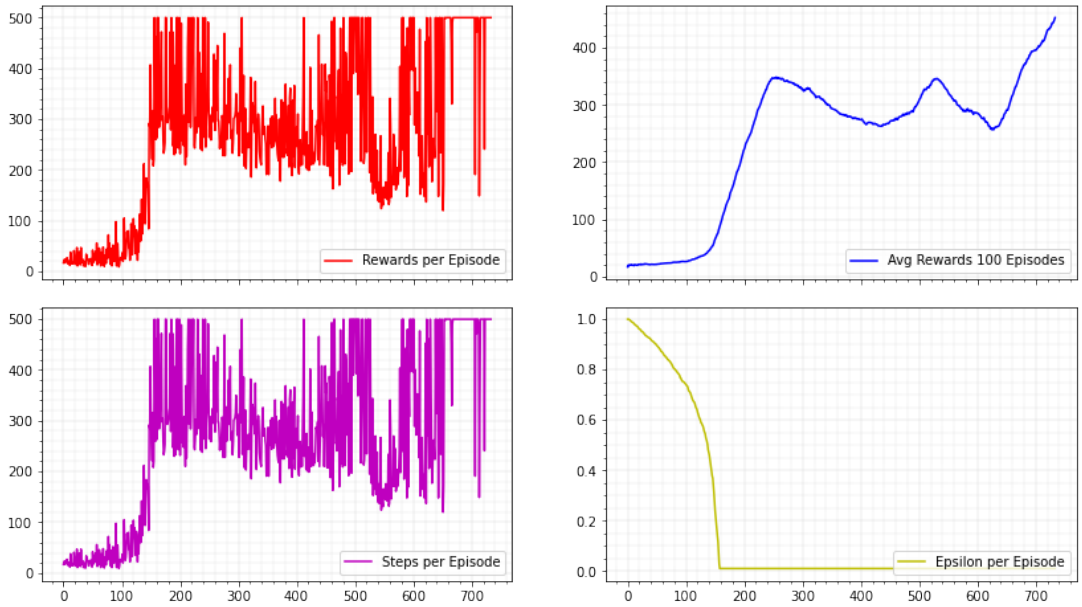

Figure 5.2: Training curve for Model-5

[2]**Catastrophic Forgetting** : Is a phenomenon in which the neural network may forget previously learned experience upon discovering a new experience or due repeated learning of the same experience

Training curves The training curves of the five models were quite different, but convey the same characteristics. We took Model-1 as reference [Fig 5.3] for explanation of the training curve. In this model one may notice that until

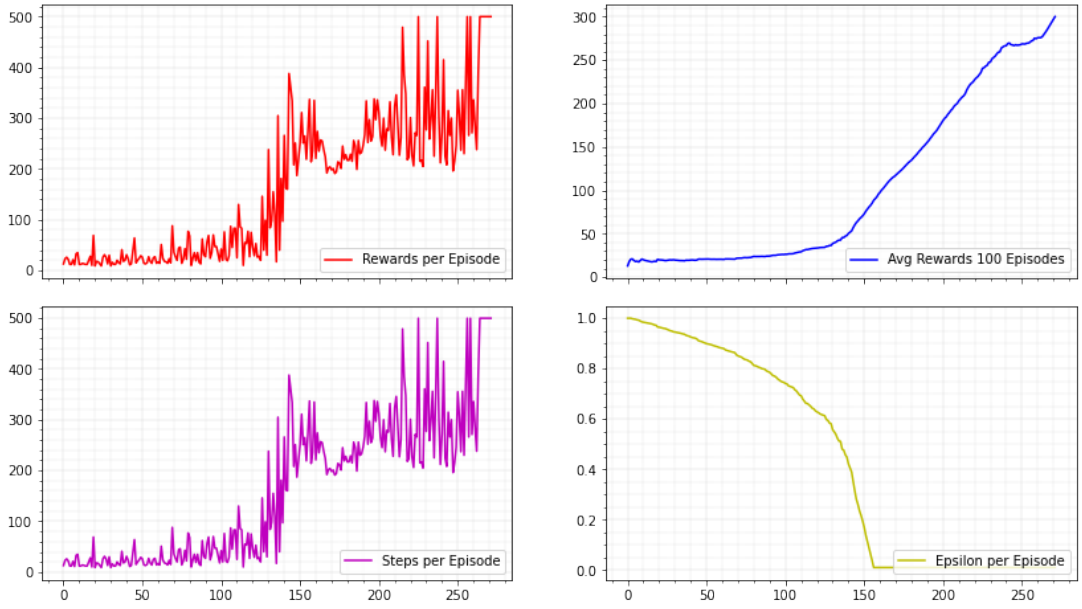

Figure 5.3: Training curve for Model-1

around 150 episodes the value of ϵ (yellow) kept on decreasing (based on epsilon decay rate `eps_dec`) , where initially the agent was exploring more and after 150 episodes it only exploited the information it learnt previously.That is the reason rewards per episodes(red) were almost constant in the first 150 episodes of the training and pumped-up thereafter. If we consider the mean-rewards over the last 100 episodes(blue) ,the mean-rewards kept on increasing almost linearly in this particular model. However, in some other models like Model-5 [Figure 5.2] the mean-rewards over the last 100 episodes were a little different pertaining to *fluctuation* in the cumulative rewards due to *Catastrophic Forgetting*. These fluctuation seems to exist because of the nature of the Cartpole-v1 environment and the Deep Q Learning algorithm, wherein the algorithm samples experiences and updates the weights of the DQN after each `step` , implying too many updates or cramming of experiences leading to fluctuation in cumulative rewards. However, determining the concrete reasons for such behaviour of the training curves is not part of this research.

Performance-testing Figure 5.4 describes how many steps and cumulative reward points Model-1 achieved for the 16 different instances of the environment. The x-axis denotes the seed values of different test-instance. In the right most-bar of the figure,the blue-bar denotes the `mean_points` and the orange-bar denotes the `mean_steps`.

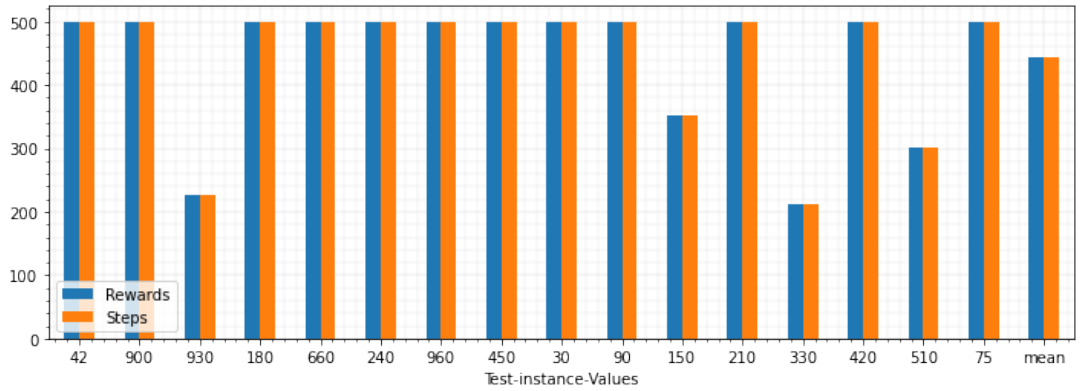

Figure 5.4: Performance on different test-instances by Model-1

Similarly, we evaluated the performances of four other models as well, which are summarized in the table below.

Table 5.3: Summary of performance results

#	success_ratio	mean_points	mean_steps	Remarks
Model-1	12:16	443	443	Good
Model-2	16:16	500	500	Perfect
Model-3	16:16	500	500	Perfect
Model-4	2:16	367	367	Not Good
Model-5	16:16	500	500	Perfect

Behaviour of the agent The trained agents were able to balance the pole using some models for 500 steps. When success ratio in considered, 3 out of 5 models of the `DQN_Agent` were perfectly trained models.

5.2.2 Results of PG_Agent

Table 5.4: Hyper-parameters of trained models

#	episodes	α	γ	(L1 , L2)	sol_th	training_time
Model-1	354	0.01	0.97	(8,8)	495	03 min 12 sec
Model-2	825	0.01	0.97	(8,8)	495	12 min 42 sec
Model-3	630	0.01	0.97	(8,4)	495	06 min 48 sec
Model-4	700	0.01	0.97	(16,8)	500	10 min 56 sec
Model-5	924	0.01	0.99	(16,16)	500	21 min 01 sec

Trained Models The five models which we trained using the `PG_Agent` had the following parameters as shown in Table 5.4. One may notice that the training time for any model was quite less and even with very less number of neurons per

layers the models could reach the sol_th limit near to 500. This means that the convergence of the `PG_Agent` on the `Cartpole-v1` environment was very fast and the PG_agent was able to solve the environment.

Training curves The training curves of the all the models were sturdy and had the similar characteristics with subtle difference. We took Model-5 as reference [Fig 5.5] for explanation of how the training curve of any of the five models may look like. In Model-5 the cumulative reward points(red) and the number

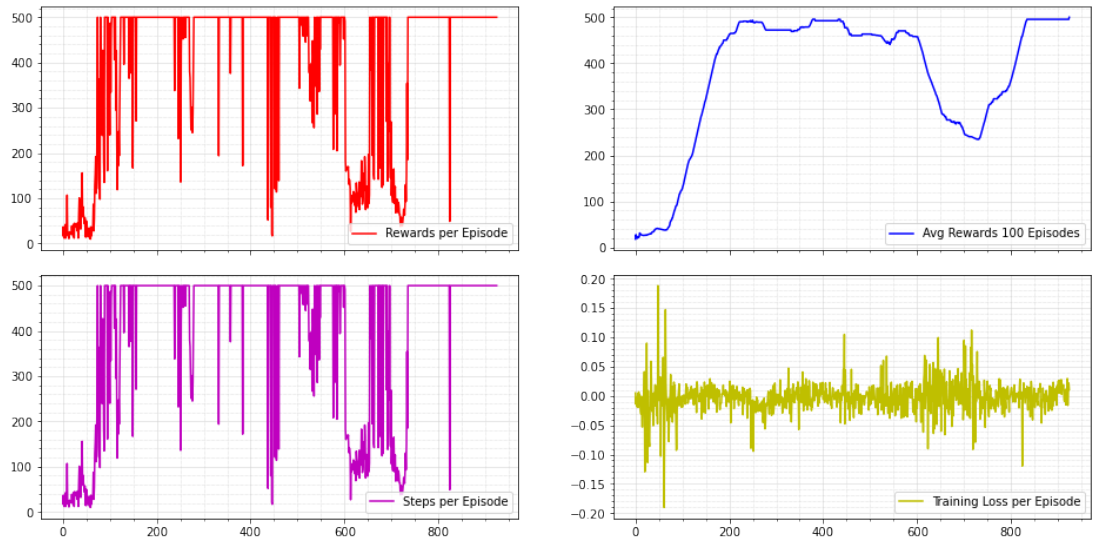

Figure 5.5: Training curve for Model-5

of steps(magenta) per episode are the same [3.3.1] for `Cartpole-v1`. One may notice that loss of the model(yellow) was more during the initial 0-100 episodes of the training phase, in the meanwhile the model could increase its cumulative rewards per episode(red) and simultaneously increase the mean rewards (blue) over the last 100 episodes. The rewards per episode(red) were maximum that is 500 for many episodes indicating that the DNN predicted the *best scenario*.The loss of the model(yellow) pumped up again in between 600-800 episodes of the training when it might have discovered new information from the environment after which the loss stabilized.

Performance-testing When we tested the accuracy of all the five models. We found that **all** the five models where able to generalize perfectly well to all the 16 instances of the environment, wherein all the five models were able to secure `mean_points` and `mean_steps` of 500 each,capturing perfectly-trained models. Figure 5.6 describes how many steps and cumulative reward points Model-5 achieved for the 16 different instances of the environment.In the right most-bar of the figure , the blue-bar denotes the `mean_points` and the orange-bar denotes

the `mean_steps` which were 500 each, indicating that the agent was able to solve all the test-instances perfectly resulting in a success_ratio of 16:16. Similarly, other four models also had success_ratio of 16:16.

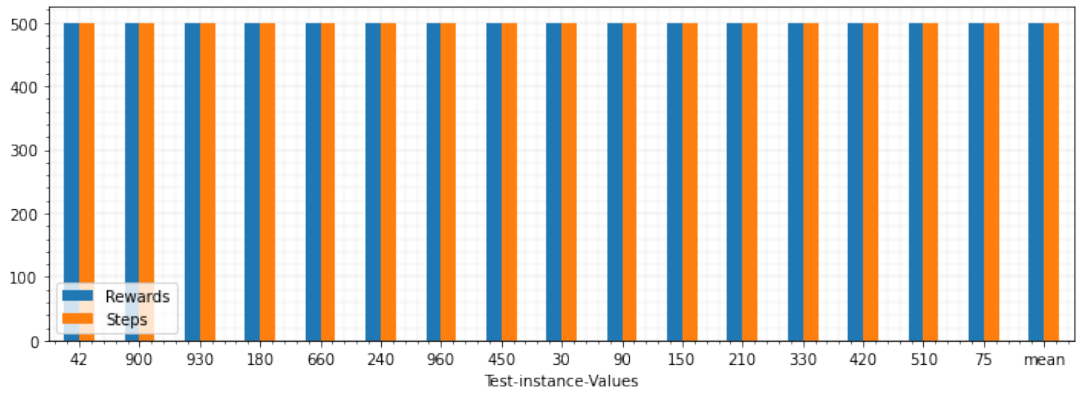

Figure 5.6: Performance on different test-instances for Model-5

Behaviour of the agent All the five models were perfectly trained models having a success-ratio of 16:16. The trained agents were able to balance the pole extremely well for 500 steps. In the animated rendering we could see that the agent was making *smooth* transitions to new states at every step of the episode and this is due to the nature of the algorithm of `PG_agent` which takes small steps in the direction of the gradient following the stochastic policy π_θ

Summary of Experiment-1

The `PG_Agent` exhibited an exemplary performance on `Cartpole-v1` by balancing the pole for maximum number of steps for the environment, wherein all of its models were perfectly trained models which had a success_ratio of 16:16. Moreover, the `PG_Agent` was able to `solve` the environment by reaching the threshold limit (sol_th) of 500 which was not the case with `DQN_Agent`.

5.3 Results from Experiment-2

In this section we will explain the results from the execution of `DQN_Agent` and `PG_Agent` on the `LunarLander-v2` environment. LunarLander-v2 is a complex environment which has a variable reward function [Refer Sec 3.3.2] because reward points depend on the location of the vehicle.

5.3.1 Results of DQN_Agent

Trained Models The hyper-parameters of the five models which were trained using the `DQN_Agent` for `LunarLander-v2` are listed in the Table 5.5.

Table 5.5: Hyper-parameters of trained models

#	episodes	α	γ	B	(L1 , L2)	sol_th	training_time	eps_dec
Model-1	315	0.001	0.99	64	(128,64)	200	08 hr 55 min	0.0001
Model-2	352	0.001	0.99	64	(128,64)	210	11 hr 24 min	0.0001
Model-3	390	0.001	0.99	64	(256,128)	210	10 hr 50 min	0.0001
Model-4	322	0.001	0.99	64	(64,32)	210	11 hr 32 min	0.0001
Model-5	384	0.002	0.99	64	(64,32)	210	13 hr 27 min	0.0001

Problems while training If you notice the sol_th in Model-1 is 200 after training and testing, the results were not up to the mark since the success_ratio was less in the test-instances. Then we incremented the sol_th to 210 for sake of increasing the models efficiency of landing since many a times we could notice the vehicle hovering near landing-pad but not landing on it.We also attempted to increase the number of neurons per layer as in Model-3 , but the the accuracy of prediction plunged [Table 5.6] , so in later models namely Model-4 and Model-5 we decreased the neurons.

Training curves The training curves of the five models were quite sturdy and reflected the same characteristics with very subtle differences. We took the best model as reference[Fig 5.7] for the explanation of the training curves. In Model-2, during the first 60 episodes the value of ϵ (yellow) kept on decreasing(based on epsilon decay rate `eps_dec`),wherein the agent was mostly exploring new states and rarely exploiting the learnt information,whereas *after* 60 episodes when the value of ϵ is constant the agent only exploits the information which it learnt previously.The learning curves of the model seems quite sturdy,where the cumulative rewards (red) kept on increasing persistently,thereby the mean-rewards(blue) over last 100 episodes also kept on increasing. One may notice that the number of steps per episode(magenta) taken by the agent eventually pumped up to the maximum threshold of 1000 steps and later on kept on decreasing which was the main requisite for the agent as per the *best scenario* for `LunarLander-v2`.We didn't notice many fluctuations in the mean reward over last 100 episodes assuming LunarLander-v2 being a complex environment had many states to explore pertaining to which *Catastrophic Forgetting* did not have vital impact on the training curve of mean-rewards (blue) over last 100 episodes.

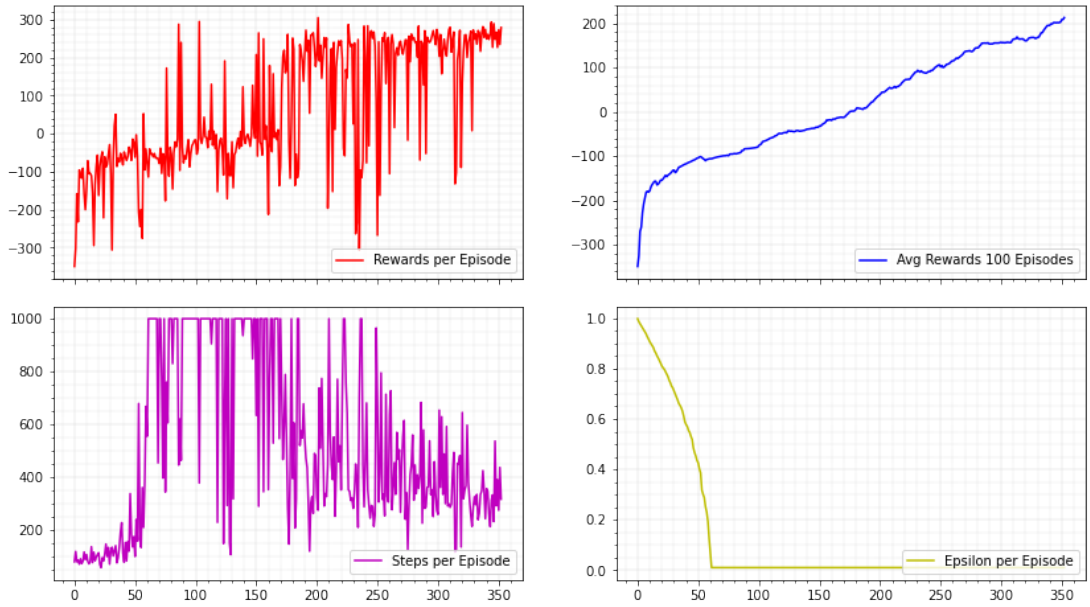

Figure 5.7: Training curve for Model-2

Performance-testing Figure 5.8 describes how many steps and cumulative reward points Model-2 achieved for the 16 different instances of the environment. The x-axis denotes the seed values of different test-instance. In the right most-bar of the figure,the blue-bar denotes the `mean_points` and the orange-bar denotes the `mean_steps`. In all the 16 test-instances the agent was able to land the vehicle perfectly on the landing-pad thereby, achieving a success-ratio of 16:16. Similarly, we evaluated the performances of four other models as well, which are summarized in the Table 5.6.

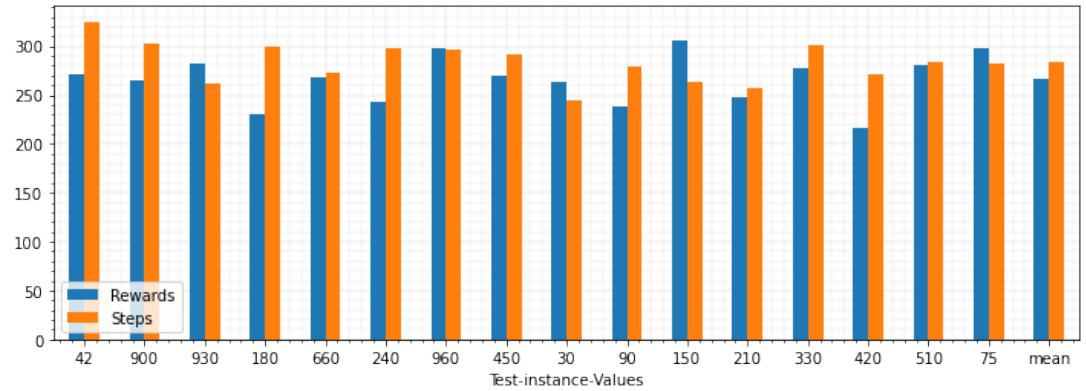

Figure 5.8: Performance on different test-instances for Model-2

Behaviour of the agent The trained agents were able to discover trajectories for the vehicle which had *minimum* number of steps as possible , thereby replicating the *best scenario*. And if we consider the success ratio, 3 out of 5 models of the `DQN_Agent` gave handsome performance. So, acting greedily by

Table 5.6: Summary of performance results

#	success-ratio	mean-points	mean-steps	Remarks
Model-1	4:16	160	583	Not Good
Model-2	16:16	265	283	Perfect
Model-3	6:16	190	365	Not Good
Model-4	16:16	257	366	Perfect
Model-5	15:16	251	310	Good

choosing actions which eventually maximized the Q-values proved worthy in case of `LunarLander-v2`

5.3.2 Results of PG_Agent

Trained Models In case of the `PG_Agent` the models which we collected for the `LunarLander-v2` are listed in Table 5.7 with there respective hyper-parameters.

Problems while training The agent was quite *sensitive* to hyper-parameters and it was really difficult to determine the best set of hyper-parameters for the models. A lot of *variance* could be noticed in the training curves with the four features in any of the five models that we collected — the cumulative rewards per episode , mean-rewards , the number of steps per episode and the loss of the network [Refer figure 5.9]. This induced variance is assumed to be because of the *slow but steady* mathematical convergence of `PG_Agent` and also because of high variance in reward function of `LunarLander-v2`, wherein the model may get stuck in a local-minima for long time and may eventually not to be able to figure out the global-minima. Sometimes but not always, the learning-rate α could be considered as parameter which can increase the convergence of the model which is what we did in Model-5 where we increased α to 0.002.

Table 5.7: Hyper-parameters of trained models

#	episodes	α	γ	(L1 , L2)	sol_th	time
Model-1	13852	0.001	0.99	(128,64)	200	08 hr 38 min
Model-2	8941	0.001	0.99	(128,64)	200	09 hr 29 min
Model-3	9817	0.001	0.99	(64,32)	200	07 hr 14 min
Model-4	3523	0.001	0.99	(32,16)	200	02 hr 20 min
Model-5	4644	0.002	0.99	(64,32)	210	01 hr 54 min

Training curves The training curves of the models of the `PG_Agent` in the case of `LunarLander-v2` were almost the same. Considering Model-4 as example [Fig 5.9] during the training in most of the episodes the number of steps per episode(magenta) was 1000 which is the maximum number of allowed steps for the

environment. However, by the end of the training phase model started gathering more than 200 cumulative rewards(red) per episode and meanwhile reducing the number of steps(magenta) taken as well. This means that the model learnt more about the environment in the final phase of the training meanwhile pumping up the loss of the network(yellow), and the model was also able to converge quickly in last 500 episodes once it determined the correct trajectory. However, if we take the number steps into consideration , the steps(magenta) consumed by the agent during the finally phase of training decreased marginally as compared to models from `DQN_Agent`, in turn not resulting in *best scenario*.

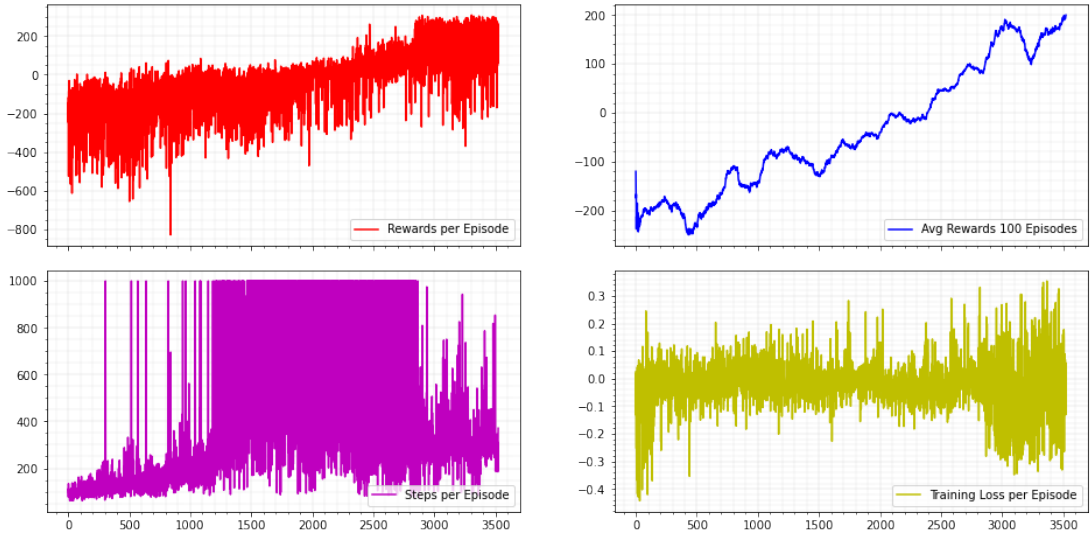

Figure 5.9: Training curve for Model-4

Performance-testing When we tested the accuracy of the five different models trained using `PG_Agent` on 16 different test instances of the environment,we discovered the following results.

Table 5.8: Summary of performance results

#	success-ratio	mean-points	mean-steps	Remarks
Model-1	15:16	179	672	Good
Model-2	13:16	210	547	Good
Model-3	16:16	164	770	Perfect
Model-4	4:16	48	946	Not Good
Model-5	12:16	245	302	Good

In Model-1 and Model-2 in many test-instances of the environment the agent was found consuming a lot of steps in figuring out the center of the landing-pad at coordinates (0,0) even after landing on the landing-pad ,assuming the agent got stuck in local-minima. For Model-4 ,the vehicle kept hovering over the

landing-pad struggling to figure out the landing-position, so reducing the neurons by too much made it hard for agent to discover new states. In Model-5 , now in certain test-instances the agent could be seen slipping down the landing-pad. Since Model-3 consumed a lot of steps , we tried to increase the learning-rate α to check if we were able to reduce the number of steps, which did work for an α of 0.002 in Model-5, which reduced to 245 mean_points and 302 mean_steps [Fig 5.10]. But success-ratio of the Model-5 decreased to 12:16. Since now in certain cases the agent could be seen slipping down the landing-pad.

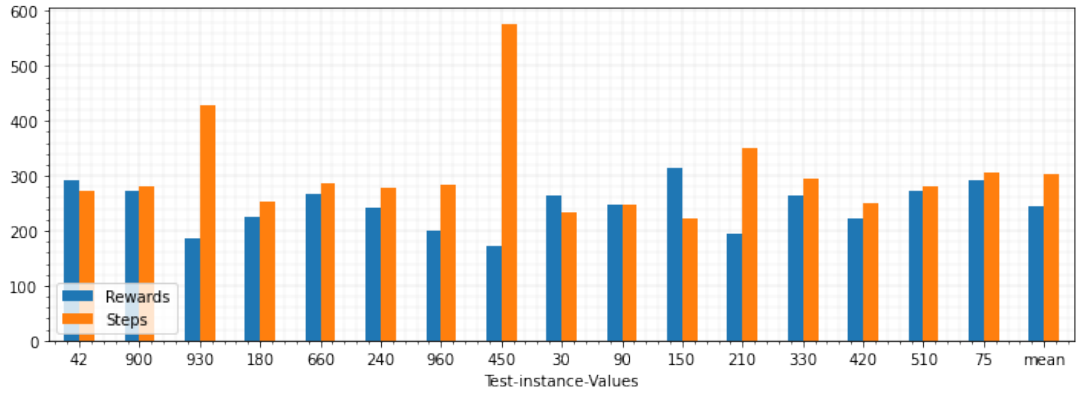

Figure 5.10: Performance on different test-instances for Model-5

Behaviour of the agent The trained agent were able to discover trajectories for the vehicle but consumed more number steps which is a problem if the vehicle has limited fuel. However, the success ratio, 4 of 5 models of the `PG_Agent` was good. This means that, if the models were trained using the `PG_Agent` then the agent was able to land the vehicle but in most test-instances consumed more number of steps in turn not replicating the *best scenario* for LunarLander-v2.

Summary of Experiment-2

It is hard to say which agent namely the `DQN_Agent` or `PG_Agent` performed better on `LunarLander-v2` .If *best scenario* is considered and meaning conveyed by training curves is taken into consideration `DQN_Agent` is a better choice for `LunarLander-v2`, since the *training curve* of `DQN_Agent` make more sense where the number of steps taken by the vehicle to find the right trajectory decreases *substantially* by the end of the training. Moreover, `DQN_Agent` did not encounter any problems of slow convergence which we encountered while training the models with `PG_Agent`, since the `DQN_Agent` acted greedily by choosing actions with maximized the Q-values.

5.4 Analysis of the Results

From the results of two experiments we were able to make the following analysis regarding the performance of agents and their behaviour.

In *Experiment-1* , we found that in case of `Cartpole-v1` environment `PG_Agent` performed far better than the `DQN_Agent`. `PG_Agent` not just trained models within minutes rather than hours , all of the five models trained using `PG_Agent` for `Cartpole-v1` were perfectly trained models with success-ratio of 16:16 and the agent was able to balance the pole for 500 steps. Moreover, we could even train models which had fewer number of neurons per layer indicating there was a feasibility to reduce the complexity of the trained models. When we trained the models for `Cartpole-v1` using the `DQN_Agent` ,none of the models were able to `solve` the environment assuming the DNN encountered *Catastrophic Forgetting*. Only some of the trained models were able to secure a success-ratio of 16:16. Moreover, we had difficulties determining the suitable set of hyper-parameters as well for the models in case of `DQN_Agent`. Therefore,taking all these facts into consideration, in case of `Cartpole-v1` environment `PG_Agent` was a better alternative.

From *Experiment-2*, we found that in case of `LunarLander-v2` environment the `DQN_Agent` and `PG_Agent` both had some pros and cons. `DQN_Agent` was able to generalize the idea of having a trained agent that could land the vehicle on the landing-pad in minimum number of steps — *best scenario* —, this is supported by the fact that 3 of the 5 trained models were able to capture this idea and had fewer problems in figuring out the right trajectory.Moreover , the training curves of the models which were trained using the `DQN_Agent` conveyed a better meaning in the training phase. On the other hand , 4 of 5 models trained using `PG_Agent` for `LunarLander-v2` had good success-ratio, but consumed more number of steps in turn not reflecting the *best scenario* and sometimes had problems figuring out the right trajectory because of *slow but steady* convergence of `PG_Agent`. In addition to this the models trained with `PG_Agent` were very sensitive to hyper-parameters assuming the variance in the reward-functions of the `LunarLander-v2` and it was very difficult to determine a suitable set of hyper-parameters to train these models. Conclusively, taking into account these pros and cons, `DQN_Agent` could be considered as a better alternative as compared to `PG_Agent` in case of `LunarLander-v2`.

5.5 Guidelines

From the analysis of the results we can certainly say that the performance of agents, which are based on different RL algorithms can vary significantly not just due to mathematical formulation of algorithms but also due to variation in the configuration and the complexity of the environments.Moreover,with the help of the analysis done in the previous sections, the following guidelines can be setup for the choice of the two agents namely DQN_Agent and PG_Agent in turn laying down the guidelines for the choice of algorithms.

5.5.1 Guidelines for DQN_Agent

The following guidelines were laid down from our analysis for the choice of DQN_Agent and its algorithm.

Table 5.9: Summary of Guidelines for DQN_Agent

Factors	Characteristics of RL problem
Reward Functions	high variance in rewards
Complexity of RL Problem	complex RL problems
Definition of RL Problem	greedy behaviour
Convergence	slow, irrespective of RL problem

According to our summary of guidelines , DQN_Agent is a better choice for RL problems which have variable reward functions and the agent appears to be suitable for complex RL problems having substantial observation space, like LunarLander-v2 had. Moreover, If the problem definition requires the problem to be solved following a greedy behaviour — say minimizing the required steps — than DQN_Agent becomes a preferable choice.However, the convergence of DQN_Agent is slow irrespective of the problem definition.

5.5.2 Guidelines for PG_Agent

The following guidelines were laid down from our analysis for the choice of PG_Agent and its algorithm.

Table 5.10: Summary of Guidelines for PG_Agent

Factors	Characteristics of RL problem
Reward Functions	low variance in rewards
Complexity of RL Problem	simpler RL problems
Definition of RL Problem	stochastic behaviour
Convergence	depends on the RL problem

According to our summary of guidelines , PG_Agent is a better choice for RL problems which have uniformity in the reward functions and the agent appears to be suitable for simple RL problems not having substantial observation space, like Cartpole-v1 had. In addition to this, If the problem definition requires the problem to be solved following a stochastic behaviour — in turn maximizing the required steps — than PG_Agent is a preferable choice.If we take the convergence into consideration then, the convergence of PG_Agent seems to be fast only if the RL problem is less complex and is having uniform reward function, otherwise the convergence is quite slow.

5.6 Summary

In this chapter we found the results by performing the two experiments and analysed them. After analysing the results , in the last section of this chapter we setup guidelines for the choice of the agents and there respective algorithms. Conclusively, as per the definition of the RL problem, if the above mentioned guidelines are followed then one can certainly differentiate between the choice of the agents including its algorithm.

CHAPTER 6

CONCLUSION

In this chapter we discussed about the conclusion we made from the research and the possible future work.

- Section 6.1 : This section puts light on the possible future works that could be done from this research.
- Section 6.2 : This section describes the conclusion made from the research.

6.1 Future Work

Use of Tensorboard One of the major concerns which we discovered from our analysis in Chapter 5 is the results of trained models very much dependent on the hyper-parameters , since variations could be observed in the training curves of various trained models. To identify best set of hyper-parameters for a model , a lot of experimentation is needed. And this experimentation is known as `Hyperparameter Tuning`. For instance , we noticed variations in the training curves the models trained using `PG_Agent` in case of LunarLander-v2 and determining suitable hyper-parameters for it was quite difficult.

Tools such as `Tensorboard` could prove beneficial in this concern as the HParams dashboard in TensorBoard provides several tools to help one find the most promising set of hyperparameters. So, as part of the future work one could try integrating Tensorboard [29] in the pre-written jupyter notebooks , conduct a number of experimentation using the dashboard and determine the promising hyper-parameters by training different models for each of the agents on different environments.

Reduce Catastrophic Forgetting As discussed in Chapter 5 that during the training phase of the `DQN_agent` a lot of bumps and fluctuations [Sec 5.2.1] could be noticed in the training curves for Cartpole-v1. In Q-learning mechanism, we updated exactly one state or action value at each timestep , but in Deep Q-learning where we replaced a Q-value function with a DNN, we were updating many state or action values at the same time.This affects the state or action values of the very next state instead of it being stable as those values are in case of simplistic Q-learning approach. This is also the reasons for the induction of *Catastrophic Forgetting*. Some suggest that the remedy for this complication, is to use a *target network* as an error measure. The usage and advantages of this technique were not discussed as part of this research but could be considered as an upcoming challenge too.

6.2 Conclusion

From the implementation and research work which we conducted it adequate to say that Deep Reinforcement Learning has applications across fields. Nonetheless, the algorithms using which the agents are constructed whether it is Policy based or Value based has a considerable impact on the behaviour and the performance of the agent. For instance as per the analysis of the results in Chapter 5 the `PG_Agent` which was the agent built on top of Policy based algorithm named

Reinforce Policy Gradient performed better in case of `Cartpole-v1` , on the other hand `DQN_Agent` based on the Value based Q-learning algorithm performed better on `LunarLander-v2`. It is not just the complexity of the environments that makes the difference in these agents performance but also the variation in the reward functions. `Cartpole-v1` for instance had a simplistic reward function where in agent just received +1 reward point for every step the agent was able to balance the pole, but the reward function in case of `LunarLander-v2` had variation in reward points in turn having variable reward function.

Policy evaluation becomes inefficient and slow for [Sec 2.3.3] high-variance in rewards, consequently making `PG_Agent` a preferrable choice for `Cartpole-v1`, because of less variance in its reward functions. Whereas `DQN_Agent` based on Value based Q-learning mechanism can tackle high-variance in reward function resulting in better learning curves and trained models.Thereby, making `DQN_Agent` suitable for `LunarLander-v2`. The results from Chapter 5 also suggest the same idea.

The two control system environments namely `Cartpole-v1` and `LunarLander-v2` were high-dimensional problems for which DNN's turned out to be suitable function approximators since the agents were able to perform and generalize well to unseen instances of the environment as per test instances. However, determining suitable set of hyper-parameters for DNN's or models is challenging.

Unlike the simulation environments, manifesting trained agents to real world control systems for solving real world problem is difficult due to underlying physics and additionally we might require some pre-processing in terms of control *inputs*. However, the agents can be trained and replicated using the same RL algorithms, following the analysis of our results in Chapter 5 and following the guidelines manifested in the final section of the same chapter.

APPENDIX A

BIBLIOGRAPHY

[1] deep reinforcement learning for classic control tasks andrew zhang - google search. `https://cs230.stanford.edu/projects_fall_2018/reports/12449630.pdf`. (Accessed on 02/23/2022).

[2] Machine learning market size, share and growth - 2022.

[3] What is machine learning: Definition, types, applications and examples, 12 2019.

[4] Optimal control, Dec 2021.

[5] *Machine learning*. 01 2022.

[6] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[7] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684, 1957.

[8] Wikipedia Contributors. Control system, 05 2019.

[9] Wikipedia Contributors. Deep learning, 05 2019.

[10] Mohit Deshp and e. Perceptrons: The first neural networks, 09 2020.

[11] Ibm Cloud Education. Deep Learning, 08 2021.

[12] stack exchange. python - what is batch size in neural network?, 01 2022.

[13] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. Solving the lunar lander problem under uncertainty using reinforcement learning. *CoRR*, abs/2011.11850, 2020.

[14] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. Solving the lunar lander problem under uncertainty using reinforcement learning. In *2020 Southeast-Con*, volume 2, pages 1–8. IEEE, 2020.

[15] Paul A Gagniuc. *Markov chains: from theory to implementation and experimentation*. John Wiley & Sons, 2017.

[16] A. Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Incorporated, 2019.

[17] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2016.

[18] DeepMind Hado van Hasselt : Senior Staff Research Scientist. Reinforcement learning lecture 1: Introduction.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 12 2013.

[21] OpenAI. Github repository of open ai gym.

[22] OpenAI. Gyms docs for cartpole environment.

[23] OpenAI. Gyms docs for lunarlander environment.

[24] OpenAI. Open ai gym docs.

[25] OpenAI. Open ai gym enviroments.

[26] openai. Part 3: Intro to policy optimization — spinning up documentation. `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html`. (Accessed on 03/04/2022).

[27] D.I. Poole, D.L. Poole, R.A. GOEBEL, D. Poole, A.K. Mackworth, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

[28] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[29] tenosorflow.com. Hyperparameter tuning with the hparams dashboard | tensorboard | tensorflow. `https://www.tensorflow.org/tensorboard/hyperparameter_tuning_with_hparams`. (Accessed on 02/21/2022).

[30] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, mar 1995.

[31] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[32] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.