Chapter 1 : Introduction

In this report titled **ANALOGY OF REINFORCEMENT LEARNING**, we present our work done on Reinforcement Learning(RL). In our project we have studied Reinforcement Learning and have done a comparative study on the RL based Q Learning method using conventional and Deep learning algorithms.

Reinforcement Learning is an area of machine learning, where an agent learns by interacting with its environment to achieve an objective. In the conventional method the model uses a Q table to store and retrieve the actions for every state based on the rewards earned by the model in the past. This lookup Q table helps the model decide the future course of action. Though this model can be efficient for small state space systems, a system with large number of state-action pairs cannot be handled using this conventional method. Also this method employs a lookup table for the system which makes it limited to that particular system and for each new system a new model has to be designed which makes it inefficient and makes it distant from human-like learning.

In contrast Deep learning employs neural nets which makes it closer to human-like learning as a single model can be used to work on different environments without any prior knowledge of the environment. The model learns by itself through the rewards it achieves on taking certain actions. Thus a single model can be used to work on similar but different environments viz gaming environments that involve

maximizing a particular score based on the actions taken on any state. Using Deep Q Learning a single model trained for a particular environment can be made to work on other models too which may have different rules, states and set of action. Thus the model is not limited to state-action pairs and is flexible to work on different environments too without any modification.

1.1 Motivation

Machine Learning(ML) is still an emerging concept and over the past decade a lot of work has been done to implement ML. A major breakthrough in the field of ML came in the year 2016 when the world recognised the power of ML when Google owned DeepMind's program AlphaGo beat Lee Sedol 4-1 in the 5 game series of the game Go. Lee Sedol is a professional Go player of 9 dan rank and is one of the strongest players in the history of Go. This was possible due to Deep Learning algorithm and Deep Q Network(DQN) developed by DeepMind in the year 2013. This inception of Deep Q Learning motivated us to do a comparative study of Deep Q learning with the conventional Q learning methods.

Work has been done in the field of Q learning and Deep Q learning individually but a comparative study and contrasting features of both the methods is still missing to evaluate the working of Q learning using conventional and Deep learning methods.

We aim to compare and contrast the applications and results of different algorithms in different situations and to compare the performance and results obtained using the two methods.

1.2 Objective

- To study Reinforcement learning and in particular Q Learning
- To study about conventional Q learning algorithms and to implement the algorithms on smart cab system and openAI gym environments
- To study about Deep Q learning algorithms and to implement the algorithm on openAI gym environments
- To compare and contrast the performance and results obtained using the two different methods.

1.3 Summary of the Report

In this report we have talked about a comparative study done on various environments using conventional Q learning technique and using Deep Q learning technique.

We have used a smart cab simulator and openAI gym environments that includes Cartpole and Mountain Car game environments and then compared and contrasted the performance and the result of the different algorithms.

Chapter 2 : LITERATURE SURVEY

Minh et. al. [1] of DeepMind Technologies along with other developers developed the first ever Deep Q network(DQN) by combining the concept of Q learning and Deep learning. The developed model was used on 7 game environments without any prior knowledge of the game environment or rules of the game. The same model was used on the 7 games without any modification and the DQN model surpassed the previous approaches made on six of the games and surpassed a human expert on 3 of them.

Lalwani and Suni et. al. [14] have presented a similar approach to learn control policies of Atari games using Reinforcement learning and have applied the Q learning based model on 3 games. The model which was trained on just one game gave successful result on all the 3 games.

Chapter 3 : Introduction To Machine Learning

3.1 Machine Learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it learn for themselves.

The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary aim is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly.

Some machine learning methods

Machine learning algorithms are often categorized as supervised or unsupervised.

- Supervised machine learning algorithms can apply what has been learned in the past to new data using labeled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system is able to provide targets for any new input after sufficient

training. The learning algorithm can also compare its output with the correct, intended output and find errors in order to modify the model accordingly.

- In contrast, unsupervised machine learning algorithms are used when the information used to train is neither classified nor labeled. Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabeled data. The system doesn't figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabeled data.
- Semi-supervised machine learning algorithms fall somewhere in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training – typically a small amount of labeled data and a large amount of unlabeled data. The systems that use this method are able to considerably improve learning accuracy. Usually, semi-supervised learning is chosen when the acquired labeled data requires skilled and relevant resources in order to train it / learn from it. Otherwise, acquiring unlabeled data generally doesn't require additional resources.
- Reinforcement machine learning algorithms is a learning method that interacts with its environment by producing actions and discovers errors or rewards. Trial and error search and delayed reward are the most relevant characteristics of reinforcement learning. This method allows machines and software agents to automatically determine the ideal behavior within a specific context in order to maximize its performance. Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal.

Machine learning enables analysis of massive quantities of data. While it generally delivers faster, more accurate results in order to identify profitable opportunities or dangerous risks, it may also require additional time and resources to train it properly. Combining machine learning with AI and cognitive technologies can make it even more effective in processing large volumes of information.

3.2 Reinforcement Learning

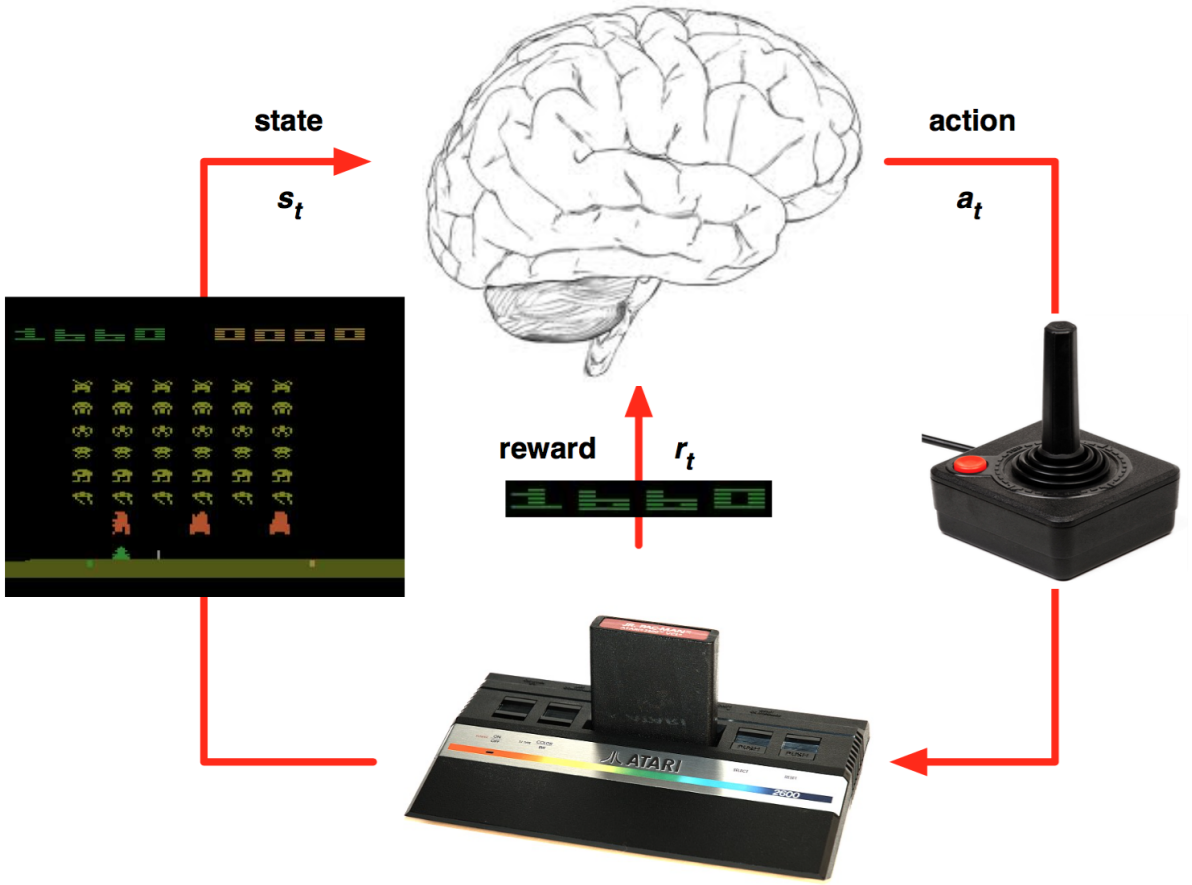


Figure 3.1 Steps for reinforcement learning

Reinforcement Learning is a type of machine learning that allows you to create AI agents that learn from the environment by interacting with it. Reinforcement learning provides the capacity for us not only to teach an artificial agent how to act, but to allow it to learn through it's own interactions with an environment.

Reinforcement learning lies somewhere in between supervised and unsupervised learning. Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all, in reinforcement learning one has sparse and time-delayed labels – the rewards. Based only on those rewards the agent has to learn to behave in the environment.

Just like how we learn to ride a bicycle, this kind of AI learns by trial and error. As seen in the picture, the brain represents the AI agent, which acts on the environment. After each action, the agent receives the feedback. The feedback consists of the reward and next state of the environment. The reward is usually defined by a human.

One of the most important aspects of reinforcement learning is that it takes into account that taking an action does not necessarily only affect the immediate reward, but it may have an impact in the future rewards as well. This way, taking what seems to be the best action now may not be the best decision in the long term. This concept is known as delayed reward.

Another of the key aspects of reinforcement learning is that in order to maximize the reward we must follow what we have learned is best, but in order to learn that we have to also explore randomly sometimes. This is called the exploration-exploitation dilemma and will be discussed in further detail later.

3.3 Markov Decision Process

The mathematical framework for defining a solution in reinforcement learning scenario is called **Markov Decision Process**. This can be designed as:

- Set of states, S
- Set of actions, A
- Reward function, R
- Policy, π
- Value, V

We have to take an action (A) to transition from our start state to our end state (*S*). In return getting rewards (R) for each action we take. Our actions can lead to a positive reward or negative reward.

The set of actions we took define our policy (π) and the rewards we get in return defines our value (V). Our task here is to maximize our rewards by choosing the correct policy. So we have to maximize $E(r_t \mid \pi, s_t)$ for all possible values of *S* for a time t.

3.3.1 States

The **state** is the current situation that the agent (your program) is in. The simplest approximation of a state is simply the current frame in your Atari game. Unfortunately, this is not always sufficient: given the image on the left, you are probably unable to tell whether the ball is going up or going down!

A simple trick to deal with this is simply to bring some of the previous history into your state (that is perfectly acceptable under the Markov property). DeepMind

chose to use the past 4 frames, so we will do the same. 2 frames is necessary for our algorithm to learn about the speed of objects, 3 frames is necessary to infer acceleration. It is unclear to me how necessary the 4th frame is (to infer the 3rd derivative of position? to gain better precision?)

3.3.2 Actions

An **action** is a command that you can give in the game in the hope of reaching a certain state and reward (more on those later). In the case of Atari games, actions are all sent via the joystick.

3.3.3 Rewards

The last component of our MDPs are the **rewards**. Rewards are given after performing an action, and are normally a function of your starting state, the action you performed, and your end state. The goal of your reinforcement learning program is to *maximize long term rewards*.

In the case of Atari, rewards simply correspond to *changes in score*, ie every time your score increases, you get a positive rewards of the size of the increase, and vice versa if your score ever decreases (which should be very rare).

3.3.4 Discounting

In practice, our reinforcement learning algorithms will never optimize for total rewards per se, instead, they will optimize for total *discounted* rewards. In other words, we will choose some number γ (gamma) where $0 < \gamma < 1$, and at each step in the future, we optimize for $r0 + \gamma\, r1 + \gamma^2\, r2 + \gamma^3\, r3\ldots$ (where r0 is the immediate reward, r1 the reward one step from now etc.).

The right discount rate is often difficult to choose: too low, and our agent will put itself in long term difficulty for the sake of cheap immediate rewards. Too high, and it will be difficult for our algorithm to converge because so much of the future needs to be taken into account. For Atari, we will mostly be using 0.99 as our discount rate.

3.3.5 Discounted Future Reward

To perform well in long-term, we need to take into account not only the immediate rewards, but also the future awards we are going to get. How should we go about that?

Given one run of Markov decision process, we can easily calculate the total reward for one episode:

$R=r1+r2+r3+\ldots+rn$

Given that, the total future reward from time point *t* onward can be expressed as:

$Rt=rt+rt+1+rt+2+\ldots+rn$

But because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions. The more into the future we go, the more it may diverge. For that reason it is common to use discounted future reward instead: $Rt=rt+\gamma rt+1+\gamma 2rt+2\ldots+\gamma n-trn$

Here γ is the discount factor between 0 and 1 – the more into the future the reward is, the less we take it into consideration. It is easy to see, that discounted future reward at time step *t* can be expressed in terms of the same thing at time step *t*+1:

$$R_t=r_t+\gamma(r_{t+1}+\gamma(r_{t+2}+\ldots))=r_t+\gamma R_{t+1}$$

If we set the discount factor γ=0, then our strategy will be short-sighted and we rely only on the immediate rewards. If we want to balance between immediate and future rewards, we should set discount factor to something like γ=0.9. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor γ=1.

A good strategy for an agent would be to always choose an action, that maximizes the discounted future reward.

3.3.6 Policies

Policies are the output of any reinforcement learning algorithm. Policies simply indicate what *action* to take for any given *state* (ie a policy could be described as a set of rules of the type "If I am in state A, take action 1, if in state B, take action 2, etc.").

A policy is called "deterministic" if it never involves "flipping a coin" for deciding the action at any state. It is called "optimal" if following it gives the highest expected discounted reward of any policy.

3.4 Broad categorization of Reinforcement Learning Algorithms

3.4.1 Model-free v.s. Model-based

The model stands for the simulation of the dynamics of the environment. That is, the model learns the transition probability T(s1|(s0, a)) from the pair of current state s0 and action a to the next state s1. If the transition probability is successfully learned, the agent will know how likely to enter a specific state given current state and action. However, model-based algorithms become impractical as the state space and action space grows (S * S * A, for a tabular setup).

On the other hand, model-free algorithms rely on trial-and-error to update its knowledge. As a result, it does not require space to store all the combination of states and actions.

3.4.2 On-policy v.s. Off-policy

An on-policy agent learns the value based on its current action a derived from the current policy, whereas its off-policy counter part learns it based on the action a* obtained from another policy. In Q-learning, such policy is the greedy policy.

3.5 Conventional Reinforcement Learning Algorithms

3.5.1 Monte Carlo

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling. The essential idea behind Monte Carlo is using randomness to solve problems which are deterministic in principle. Since the value of a state is defined as the expectation of the random return when the process is started from the given state, an obvious way of estimating this value is to compute

an average over multiple independent realizations started from the given state. This is an instance of the so-called Monte-Carlo method.

They are mainly used for optimisation , numerical integration, and generating draws from a probability distribution.

3.5.1.1 Limitations of Monte Carlo

Since the variance of the returns can be high, which means that the quality of the estimates will be poor. Also, when interacting with a system in a closed-loop fashion (i.e., when estimation happens while interacting with the system), it might be impossible to reset the state of the system to some particular state. In this case, the Monte-Carlo technique cannot be applied without introducing some additional bias.

3.5.2 Temporal difference learning

It is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Uses ‘bootstrapping’: predictions are used as targets during the course of learning. Considers that subsequent predictions are often correlated in some sense.

3.5.3 TD(λ)

This algorithm aims toward unifying Monte-Carlo and TD(0).

‘λ’ [0,1], refers to the trace decay parameter.

$\lambda = 0$ gives TD(0).

$\lambda = 1$, (TD(1)) is equivalent to a Monte-Carlo method.

3.5.4 Actor-critic methods

In this method the policy is updated before it is completely evaluated. It is also known as Generalized Policy Iteration (GPI). The actor aims at improving the current policy. The critic evaluates the current policy, thus helping the actor.

3.5.4.1 The Actor-Critic Architecture

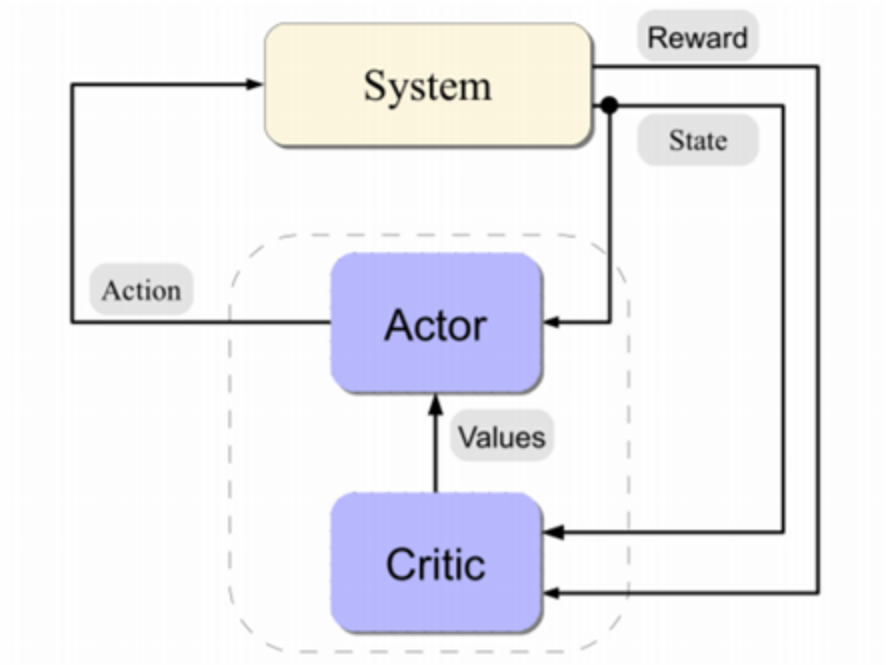


Figure 3.2 Schematic of actor-critic architecture

3.5.5 SARSA

- An extension of TD(λ).
- Current **S**tate, current **A**ction, next **R**eward, next **S**tate and next **A**ction.
- Similar to Q-learning, keeps track of the action-value underlying the possible state-action pairs.
- The difference is in the definition of the TD error.

function SARSAACTORCRITIC(X)
Input: X is the current state
$\omega, \theta, z \leftarrow 0$
$A \leftarrow a_1$ ▷ Pick any action
repeat
  $(R, Y) \leftarrow$ EXECUTEINWORLD(A)
  $A' \leftarrow$ DRAW($\pi_\omega(Y, \cdot)$)
  $(\theta, z) \leftarrow$ SARSALAMBDALINFAPP($X, A, R, Y, A', \theta, z$) ▷ Use $\lambda = 1$ and $\alpha \gg \beta$
  $\psi \leftarrow \frac{\partial}{\partial \omega} \log \pi_\omega(X, A)$
  $v \leftarrow$ SUM($\pi_\omega(Y, \cdot) \cdot \theta^\top \varphi[X, \cdot]$)
  $\omega \leftarrow \omega + \beta \cdot \left(\theta^\top \varphi[X, A] - v\right) \cdot \psi$
  $X \leftarrow Y$
  $A \leftarrow A'$
until True

Figure 3.3 Pseudo code of SARSA

3.5.6 Q-Learning

Q-learning is a popular, off-policy learning algorithm that utilizes the Q function. It is based on the simple premise that the best policy is the one that selects the action with the highest total future reward.

In a reinforcement learning model, an agent takes actions in an environment with the goal of maximizing a cumulative reward. The basic reinforcement learning model consists of: a set of environment states S; a set of actions A; rules of transitioning between states; rules that determine the scalar immediate reward of a transition; and rules that describe what the agent observes.

Q-Learning is a model-free form of Reinforcement Learning. If S is a set of states, A is a set of actions, is the discount factor, is the step size. Then we can understand Q-Learning by this Algorithm:

$Initialize\ \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})\ arbitrarily$

$Repeat\ (for\ each\ episode)$:

$Initialize\ \boldsymbol{S}$

$Repeat\ (for\ each\ step\ of\ episode)$:

$Choose\ \boldsymbol{a}\ from\ \boldsymbol{s}\ using\ policy\ derived\ from\ \boldsymbol{Q}\ (e.\ g. \in -greedy)$

$$Take\ action\ \boldsymbol{a}, observe\ \boldsymbol{r}, \boldsymbol{s'}\ \boldsymbol{Q'}(\boldsymbol{s'}, \boldsymbol{a'}) <-- \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a}) + \boldsymbol{\alpha}[\boldsymbol{r} + \boldsymbol{\gamma}.\ \boldsymbol{max}\ \boldsymbol{Q}(\boldsymbol{s'}, \boldsymbol{a'}) - \boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})]$$

$$\boldsymbol{s} <-- \boldsymbol{s'}$$

$$until\ \boldsymbol{s}\ is\ terminal$$

In Q-learning, the value of the Q function for each state is updated iteratively based on the new rewards it receives. At its most basic level, the algorithm looks at the difference between (a) its current estimate of total future reward and (b) the estimate generated from its most recent experience. After it calculates this difference (a - b), it adjusts its current estimate up or down a bit based on the number. Q-learning uses a couple of parameters to allow us to tweak how the process works. The first is the learning rate, represented by the Greek letter α (alpha). This is a number between 0 and 1 that determines the extent to which newly learned information will impact the existing estimates. A low α means that the agent will put less stock into the new information it learns, and a high α means that new information will more quickly override older information. The second parameter is the discount rate, g. The higher the discount rate, the less important future rewards are valued.

Below is the annotated equation for Q-learning:

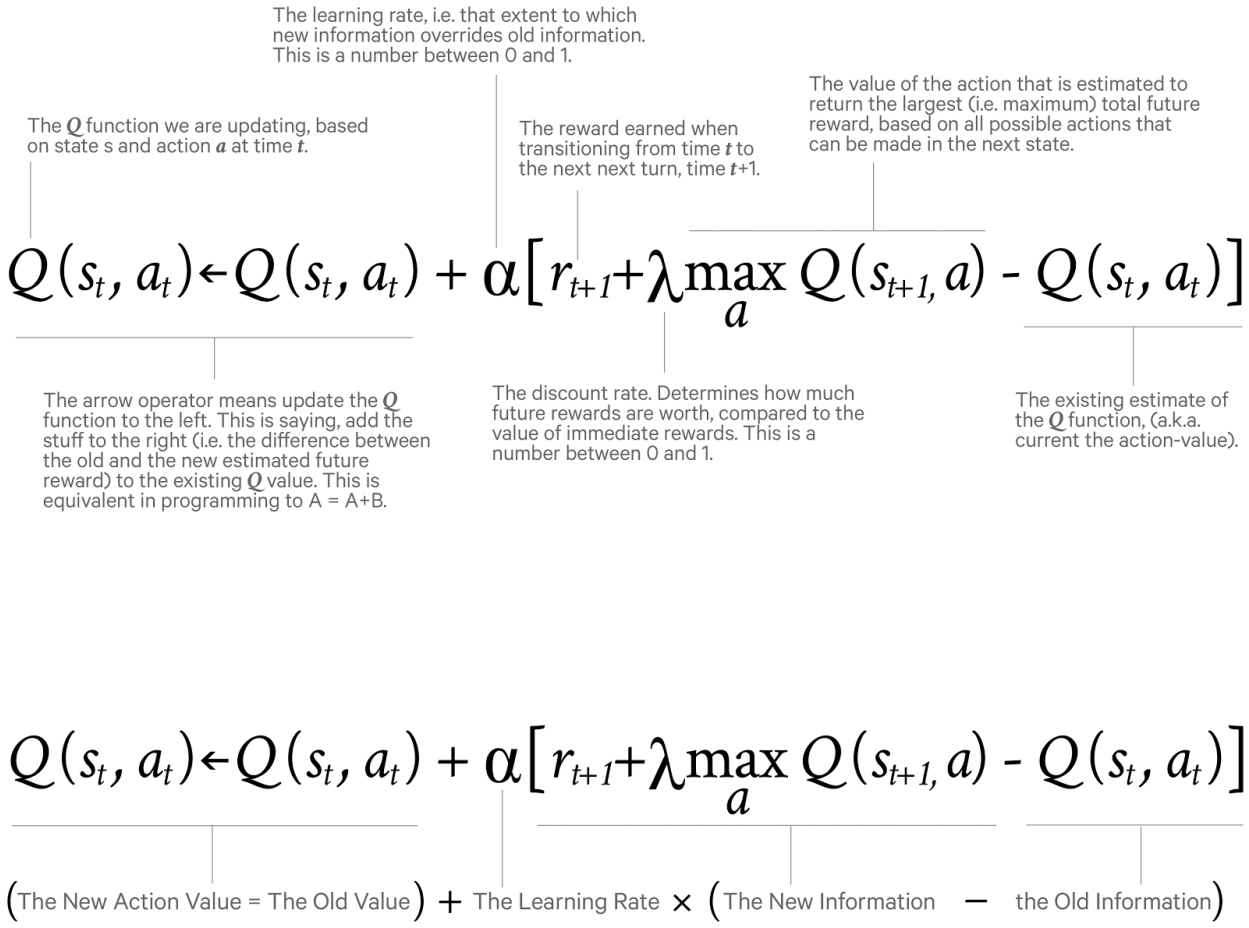


Figure 3.4 Explanation of Q-learnings equation

3.5.6.1 The Q Function

At the heart of Q-Learning is the function *Q(s, a)*. This function gives the discounted total value of taking action *a* in state *s*. How is that determined you say? Well, *Q(s, a)* is simply equal to the reward you get for taking a in state *s*, plus the discounted value of the state *s'* where you end up. Further, the value of a state is

simply the value of taking the optimal action at that state, ie $max_a(Q(s, a))$, so we have:

$Q(s, a) = r + \gamma\, max_a'(Q(s', a'))$

3.5.6.2 Tabular Approaches for Tabular Environments

In its simplest implementation, Q-Learning is a table of values for every state (row) and action (column) possible in the environment. Within each cell of the table, we learn a value for how good it is to take a given action within a given state.We start by initializing the table to be uniform (all zeros), and then as we observe the rewards we obtain for various actions, we update the table accordingly.

We make updates to our Q-table using something called the Bellman equation, which states that the expected long-term reward for a given action is equal to the immediate reward from the current action combined with the expected reward from the best future action taken at the following state. *In this way, we reuse our own Q-table when estimating how to update our table for future actions!* In equation form, the rule looks like this:

Eq 1. $Q(s,a) = r + \gamma(max(Q(s',a'))$

This says that the Q-value for a given state (s) and action (a) should represent the current reward (r) plus the maximum discounted (γ) future reward expected according to our own table for the next state (s') we would end up in. The discount variable allows us to decide how important the possible future rewards are compared to the present reward. By updating in this way, the table slowly begins to obtain accurate measures of the expected future reward for a given action in a given state.

3.5.6.3 Q-Learning with Neural Networks

Now, you may be thinking: tables are great, but they don't really scale, do they? While it is easy to have a 16x4 table for a simple grid world, the number of possible states in any modern game or real-world environment is nearly infinitely larger. For most interesting problems, tables simply don't work. We instead need some way to take a description of our state, and produce Q-values for actions without a table: that is where neural networks come in. By acting as a function approximator, we can take any number of possible states that can be represented as a vector and learn to map them to Q-values.

While the network learns to solve the Frozen-Lake problem, it turns out it doesn't do so quite as efficiently as the Q-Table. While neural networks allow for greater flexibility, they do so at the cost of stability when it comes to Q-Learning. There are a number of possible extensions to our simple Q-Network which allow for greater performance and more robust learning.

3.6 Artificial Neural Networks

Artificial neural networks (**ANNs**) or **connectionist systems** are computing systems vaguely inspired by the biological neural networks that constitute animal brains. Such systems "learn" (i.e. progressively improve performance on) tasks by considering examples, generally without task-specific programming. For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any a priori knowledge about cats, e.g., that they have fur, tails, whiskers and cat-like faces. Instead, they evolve their own set of relevant characteristics from the learning material that they process.

An ANN is based on a collection of connected units or nodes called artificial neurons (a simplified version of biological neurons in an animal brain). Each connection (a simplified version of a synapse) between artificial neurons can transmit a signal from one to another. The artificial neuron that receives the signal can process it and then signal artificial neurons connected to it.

In common ANN implementations, the signal at a connection between artificial neurons is a real number, and the output of each artificial neuron is calculated by a non-linear function of the sum of its inputs. Artificial neurons and connections typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Artificial neurons may have a threshold such that only if the aggregate signal crosses that threshold is the signal sent. Typically, artificial neurons are organized in layers. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first (input), to the last (output) layer, possibly after traversing the layers multiple times.

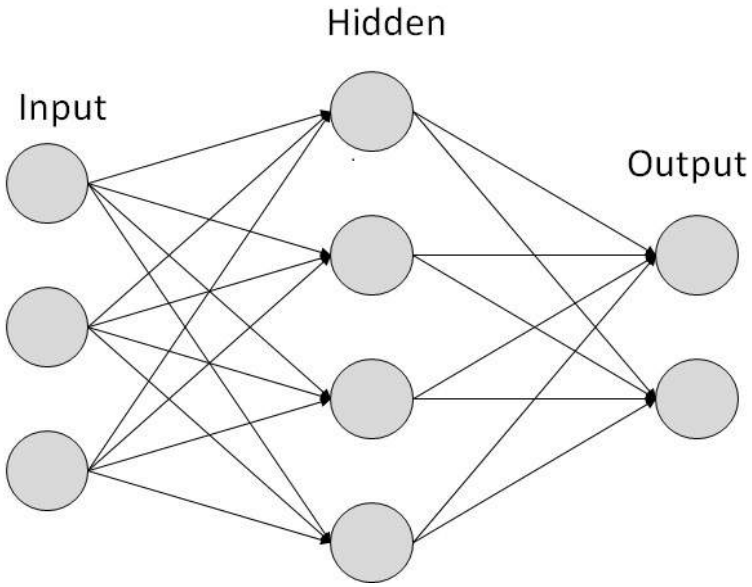


Figure 3.5 Basic Architecture of an Artificial Neural Network

3.7 Deep Reinforcement Learning

Google's DeepMind published its famous paper Playing Atari with Deep Reinforcement Learning, in which they introduced a new algorithm called **Deep Q Network** (DQN for short) in 2013. It demonstrated how an AI agent can learn to play games by just observing the screen without any prior information about those games. The result turned out to be pretty impressive. This paper opened the era of what is called 'deep reinforcement learning', a mix of deep learning and reinforcement learning.

In Q-Learning Algorithm, there is a function called **Q Function**, which is used to approximate the reward based on a state. We call it **Q(s,a)**, where Q is a function which calculates the expected future value from state s and action a. Similarly in Deep Q Network algorithm, we use a neural network to approximate the reward based on the state.

3.8 Deep Q Learning

The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the existence of each individual brick. This intuitive representation is however game specific. Could we come up with something more universal, that would be suitable for all the games? Obvious choice is screen pixels – they implicitly contain all of the relevant information about the game situation, except for the speed and direction of the ball. Two consecutive screens would have these covered as well.

If we would apply the same preprocessing to game screens as in the DeepMind paper – take four last screen images, resize them to 84×84 and convert to grayscale with 256 gray levels – we would have $256^{84\times84\times4}\approx1067970$ possible game states. This means 1067970 rows in our imaginary Q-table – that is more than the number of atoms in the known universe! One could argue that many pixel combinations and therefore states never occur – we could possibly represent it as a sparse table containing only visited states. Even so, most of the states are very rarely visited and it would take a lifetime of the universe for the Q-table to converge. Ideally we would also like to have a good guess for Q-values for states we have never seen before.

This is the point, where deep learning steps in neural networks are exceptionally good in coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q-value. Alternatively we could take only game screens as input and output the Q-value for each possible

action. This approach has the advantage, that if we want to perform a Q-value update or pick the action with highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions immediately available.

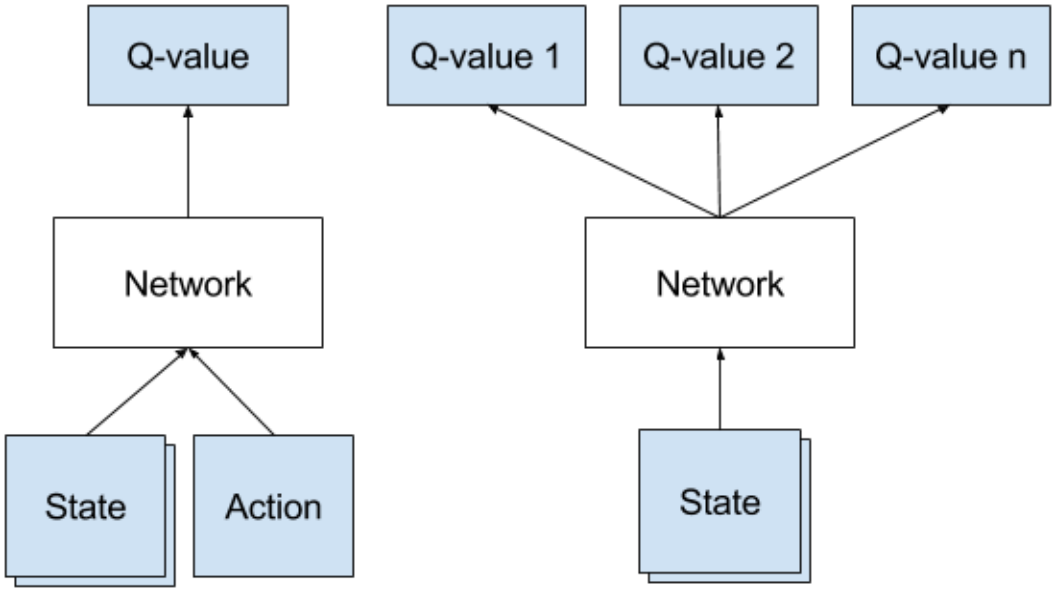


Figure 3.6 Left: Naive formulation of deep Q-network. Right: More optimal architecture of deep Q-network, used in DeepMind paper.

3.9 Implementing Mini Deep Q Network (DQN)

Normally in games, the *reward* directly relates to the score of the game. Imagine a situation where the pole from CartPole game is tilted to the right. The expected future reward of pushing right button will then be higher than that of pushing the left button since it could yield higher score of the game as the pole survives longer. In order to logically represent this intuition and train it, we need to express this as a formula that we can optimize on. The loss is just a value that indicates how far our

prediction is from the actual target. For example, the prediction of the model could indicate that it sees more value in pushing the right button when in fact it can gain more reward by pushing the left button. We want to decrease this gap between the prediction and the target (loss). We will define our loss function as follows:

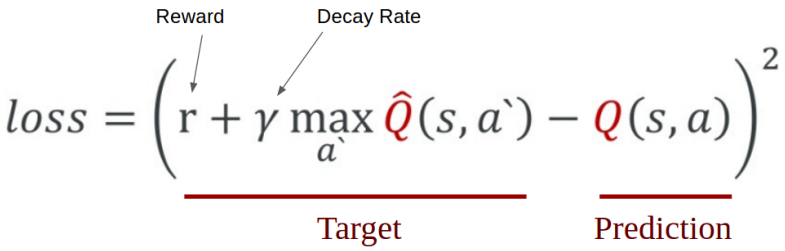


Figure 3.7 Mathematical representation of Q-learning from Taehoon Kim's slides

We first carry out an action *a*, and observe the reward *r* and resulting new state *s`*. Based on the result, we calculate the maximum target Q and then discount it so that the future reward is worth less than immediate reward (It is a same concept as interest rate for money. Immediate payment always worth more for same amount of money). Lastly, we add the current reward to the discounted future reward to get the target value. Subtracting our current prediction from the target gives the loss. Squaring this value allows us to punish the large loss value more and treat the negative values same as the positive values.

Chapter 4 : Description

4.1 Toolkit/API Used

4.1.1 Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano.

Reinforcement learning (RL) is the subfield of machine learning concerned with decision making and motor control. It studies how an agent can learn how to achieve goals in a complex, uncertain environment. It's exciting for two reasons:

- RL is very general, encompassing all problems that involve making a sequence of decisions:for example, controlling a robot's motors so that it's able to run and jump, making business decisions like pricing and inventory management, or playing video games and board games. RL can even be applied to supervised learning problems with sequential or structured outputs.
- RL algorithms have started to achieve good results in many difficult environments. RL has a long history, but until recent advances in deep learning, it required lots of problem-specific engineering. DeepMind's Atari results, BRETT from Pieter Abbeel's group, and AlphaGo all used deep RL

algorithms which did not make too many assumptions about their environment, and thus can be applied in other settings.

However, RL research is also slowed down by two factors:

- The need for better benchmarks. In supervised learning, progress has been driven by large labeled datasets like ImageNet. In RL, the closest equivalent would be a large and diverse collection of environments. However, the existing open-source collections of RL environments don't have enough variety, and they are often difficult to even set up and use.
- Lack of standardization of environments used in publications. Subtle differences in the problem definition, such as the reward function or the set of actions, can drastically alter a task's difficulty. This issue makes it difficult to reproduce published research and compare results from different papers.

Gym is an attempt to fix both problems.By combining the complex representations that deep neural networks can learn with the goal-driven learning of an RL agent, computers have accomplished some amazing feats, like beating humans at over a dozen Atari games, and defeating the Go world champion.

4.1.2 Tensorflow

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and

engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open source license on November 9, 2015.

4.1.3 Keras

Keras is an open source neural network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, Theano, or MXNet. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It was developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System), and its primary author and maintainer is François Chollet, a Google engineer.
Keras contains numerous implementations of commonly used neural network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier.

Keras allows users to productize deep models on smartphones (iOS and Android), on the web, or on the Java Virtual Machine.It also allows use of distributed training of deep learning models on clusters of Graphics Processing Units (GPU).

4.2 Overview and working of the testing environments

4.2.1 Smart Cab

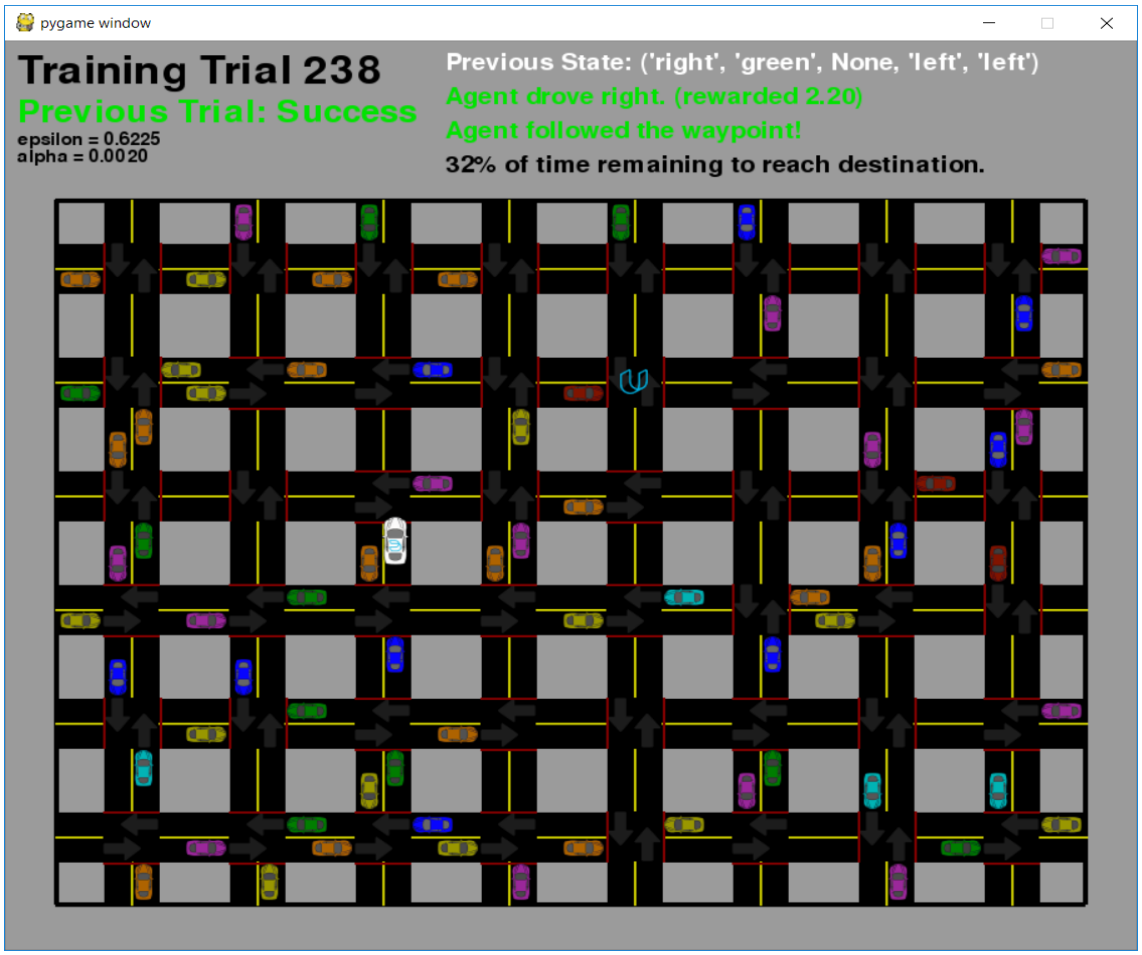


Figure 4.1 GUI environment for SmartCab

Smart Cab simulator is designed and developed in Python. The aim is to construct an optimized Q-Learning driving agent that will navigate a *Smartcab* through its environment towards a goal.

The environment consists of a grid like structure with a number of cars in it with different colors and the Smart Cab is the only car represented in the white color. The cab is initially left in the environment to explore the environment and choose the best policy. The car has to navigate its way into the environment in order to reach the destination while following the traffic rules and also ensuring a safe journey.

Since the *Smartcab* is expected to drive passengers from one location to another, the driving agent will be evaluated on two very important metrics: **Safety** and **Reliability**.

In order for a journey to be considered safe the car must follow traffic rules like stopping on red light, passing on green lights with no oncoming traffic etc. A driving agent that gets the Smartcab to its destination while running red lights or narrowly avoiding accidents would be considered unsafe. Similarly, a driving agent that frequently fails to reach the destination in time would be considered unreliable. A reliable journey would simply mean reaching the destination in stipulated time.

Maximizing the driving agent's **safety** and **reliability** would ensure that Smart-cabs have a permanent place in the transportation industry.

Safety and **Reliability** are measured using a letter-grade system as follows:

Table 4.1 Assignment of grades for SmartCab based on Safety and Reliability

Grade	Safety	Reliability
A+	Agent commits no traffic violations, and always chooses the correct action.	Agent reaches the destination in time for 100% of trips.
A	Agent commits few minor traffic violations, such as failing to move on a green light.	Agent reaches the destination on time for at least 90% of trips.
B	Agent commits frequent minor traffic violations, such as failing to move on a green light.	Agent reaches the destination on time for at least 80% of trips.
C	Agent commits at least one major traffic violation, such as driving through a red light.	Agent reaches the destination on time for at least 70% of trips.
D	Agent causes at least one minor accident, such as turning left on green with oncoming traffic.	Agent reaches the destination on time for at least 60% of trips.
F	Agent causes at least one major accident, such as driving through a red light with cross-traffic.	Agent fails to reach the destination on time for at least 60% of trips.

4.2.2 Cartpole

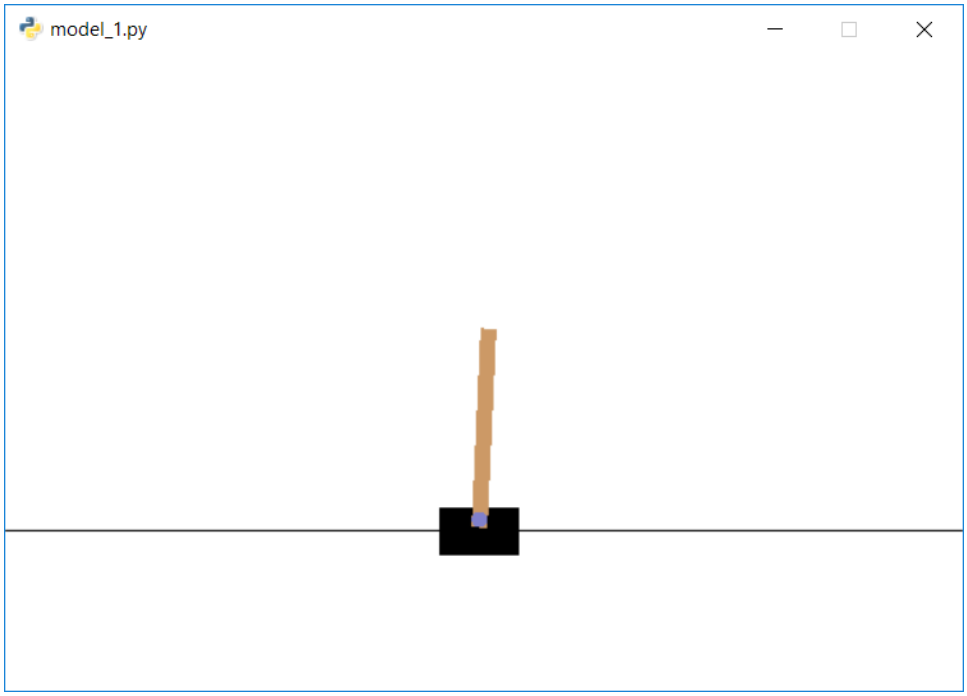


Figure 4.2 GUI environment for Cartpole

CartPole is one of the simplest environments in OpenAI gym (a game simulator). The goal of CartPole is to balance a pole connected with one joint on top of a moving cart. Instead of pixel information, there are 4 kinds of information given by the state, such as angle of the pole and position of the cart. An agent can move the cart by performing a series of actions of 0 or 1 to the cart, pushing it left or right.

4.2.2.1 How The Agent Decides to Act

Our agent will randomly select its action at first by a certain percentage, called ‘exploration rate’ or ‘epsilon’. This is because at first, it is better for the agent to

try all kinds of things before it starts to see the patterns. When it is not deciding the action randomly, the agent will predict the reward value based on the current state and pick the action that will give the highest reward.

In the beginning, the agent explores by acting randomly.
It goes through multiple phases of learning.

1. The cart masters balancing the pole.
2. But goes out of bounds, ending the game.
3. It tries to move away from the bounds when it is too close to them, but drops the pole.
4. The cart masters balancing and controlling the pole.

After several hundreds of episodes , it starts to learn how to maximize the score.
The final result is the birth of a skillful CartPole game player!

4.2.2.2 Cartpole game Environment

Table 4.2 State Space Representation

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Table 4.3 List of Valid Actions

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

Note: The amount the velocity is reduced or increased is not fixed as it depends on the angle the pole is pointing. This is because the center of gravity of the pole increases the amount of energy needed to move the cart underneath it.

4.2.2.3 Evaluation Criteria

1. Score : The number of time frames that the agent manages to balance the pole.
2. Reward : Reward is 1 for every step taken, including the termination step.
3. Average Score : The mean of the scores till the respective episode.
4. Average Reward : The reward of the scores till the respective episode.

4.2.2.4 Starting State

All observations are assigned a uniform random value between ±0.05

4.2.2.5 Episode Termination

1. Pole Angle is more than ±12°
2. Cart Position is more than ±2.4 (center of the cart reaches the edge of the display)
3. Episode length is greater than 200

4.2.3 Mountain Car

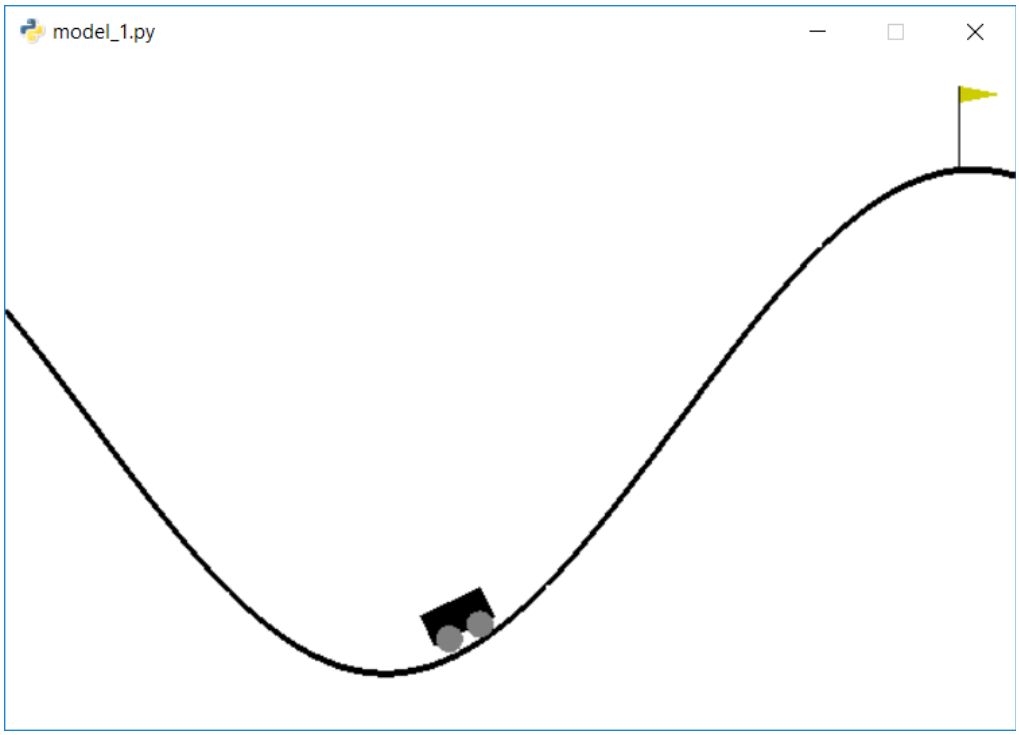


Figure 4.3 GUI environment for Mountain Car

Mountain Car is a problem in which an underpowered car must drive up a steep hill. A car is on a one-dimensional track, positioned between two "mountains".The

goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill.

4.2.3.1 Mountain Car game Environment

Table 4.4 State Space Representation for Mountain Car

Num	Observation	Min	Max
0	position	-1.2	0.6
1	velocity	-0.07	0.07

Table 4.5 List of valid Actions

Num	Observation
0	push left
1	no push
2	push right

4.2.3.2 Evaluation Criteria

1. Score : The highest coordinate that the agent manages to attain .
2. Reward : -1 for each time step, until the goal position of 0.5 is reached.
3. Average Score : The mean of the scores till the respective episode.
4. Average Reward : The reward of the scores till the respective episode.

4.2.3.3 Starting State

Random position from -0.6 to -0.4 with no velocity.

4.2.3.4 Episode Termination

The episode ends when you reach 0.5 position, or if 200 iterations are reached.

Chapter 5 : Results and Conclusion

5.1 Results for SmartCab (Policy Iteration Q-Learning)

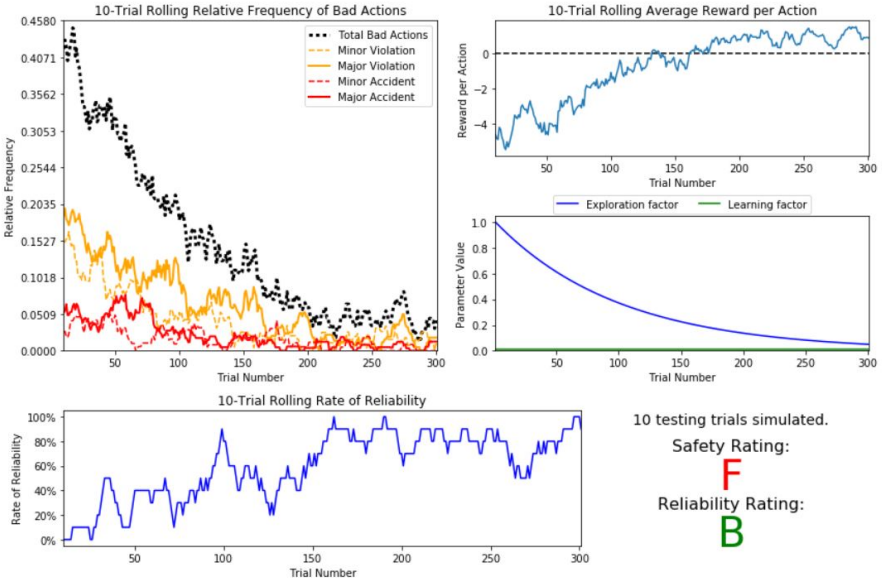


Figure 5.1 : Simulation results of Q- learning agent for learning rate = 0.01

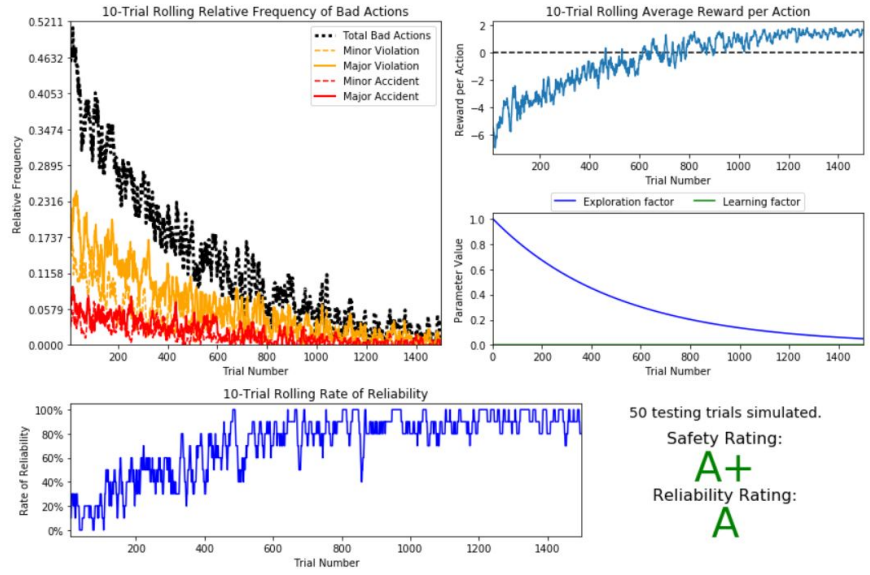


Figure 5.2 : Simulation results of Q- learning agent for learning rate = 0.002

5.2 Cartpole

5.2.1 Learning Curves

5.2.1.1 Policy Iteration Q Learning Agent

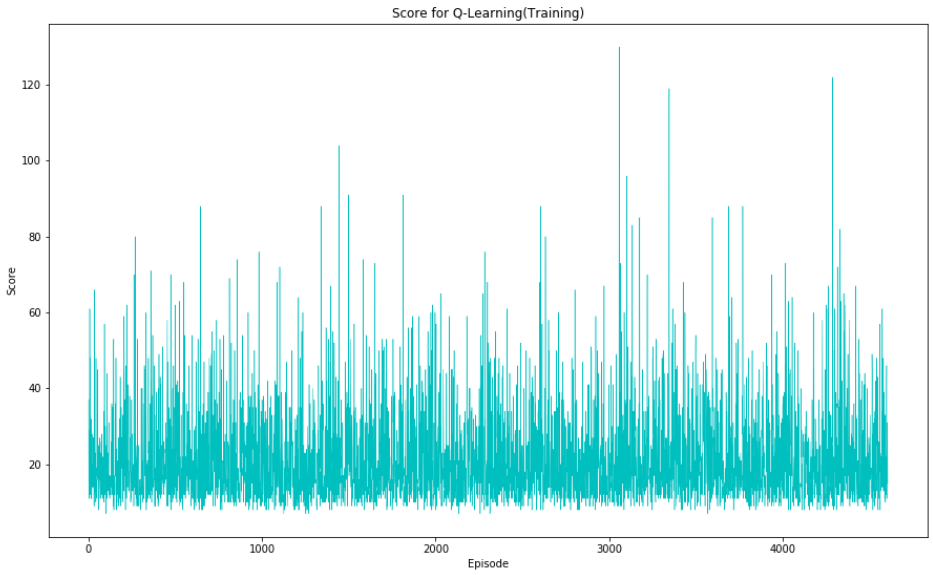


Figure 5.3 : Score vs Episodes

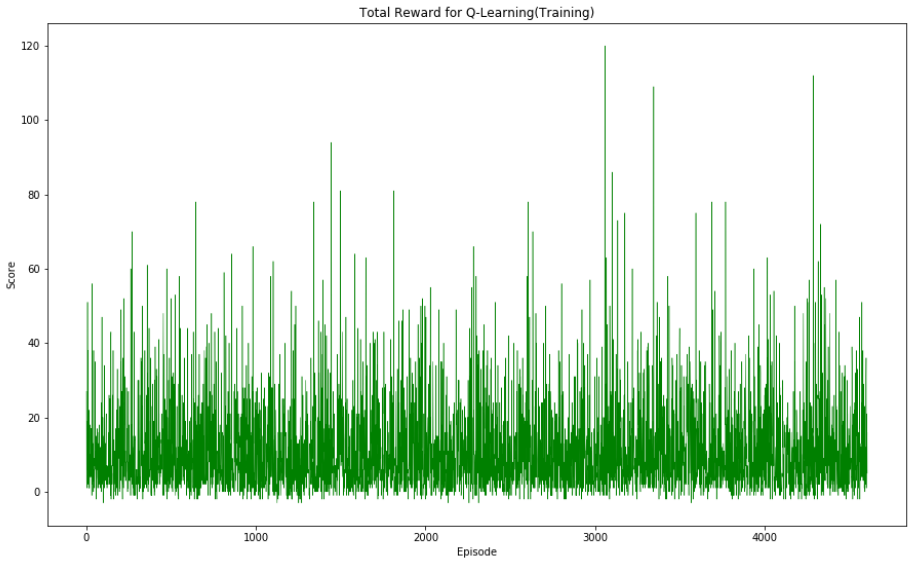


Figure 5.4 : Reward vs Episodes

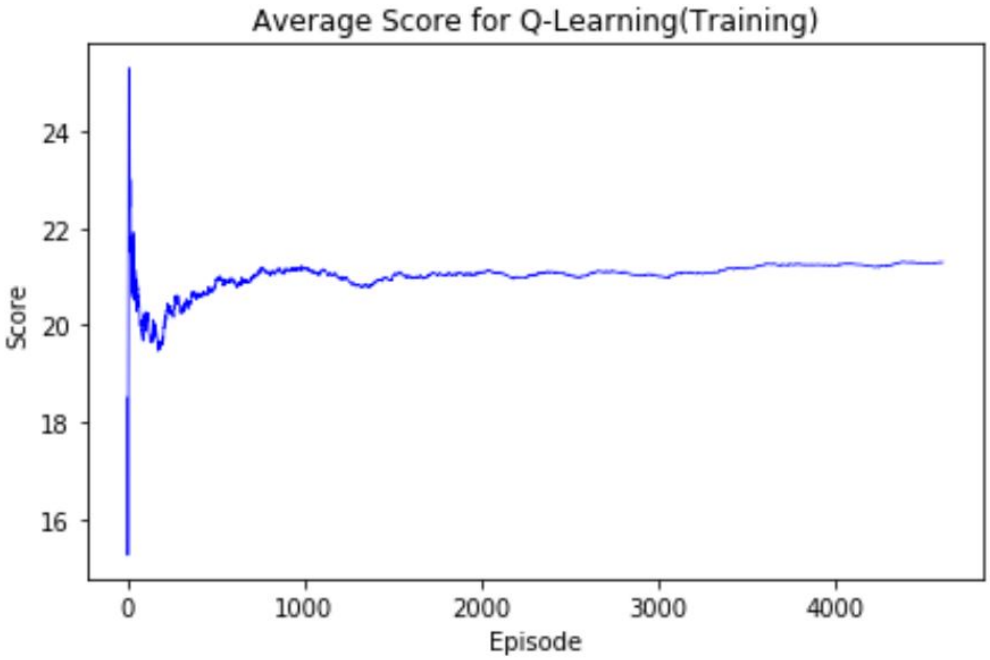


Figure 5.5 : Average Score vs Episodes

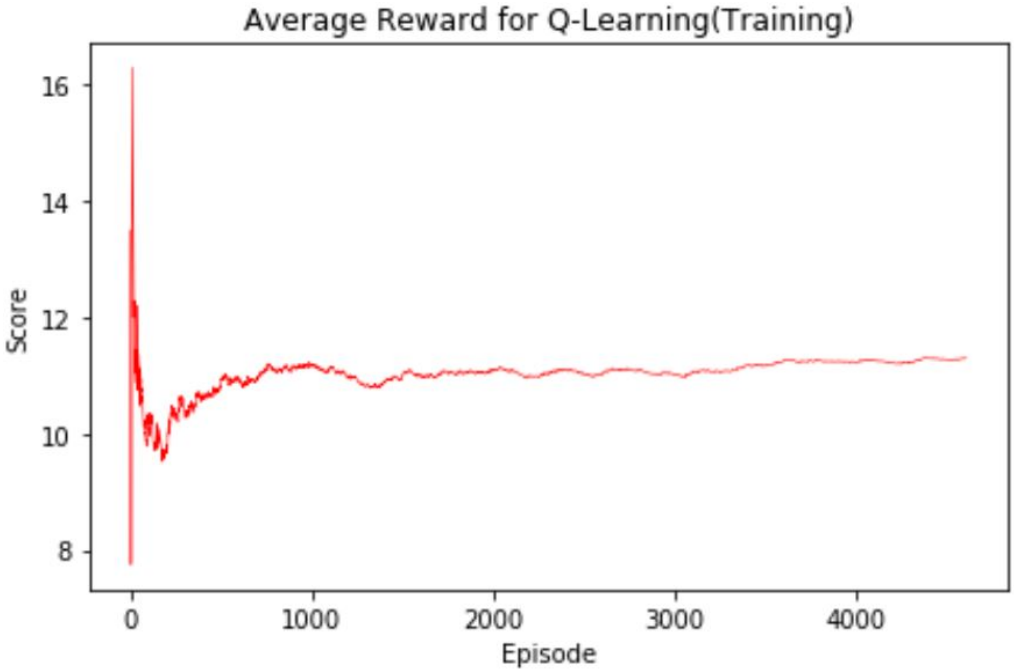


Figure 5.6 : Average reward vs Episodes

5.2.1.2 Deep-Q-Learning Agent

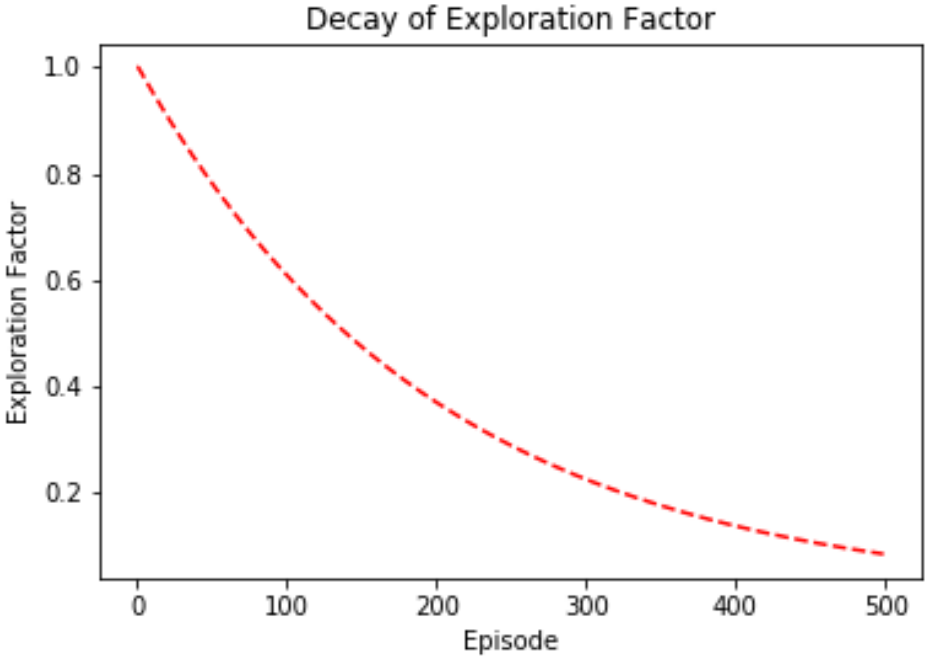


Figure 5.7 : Epsilon Decay rate

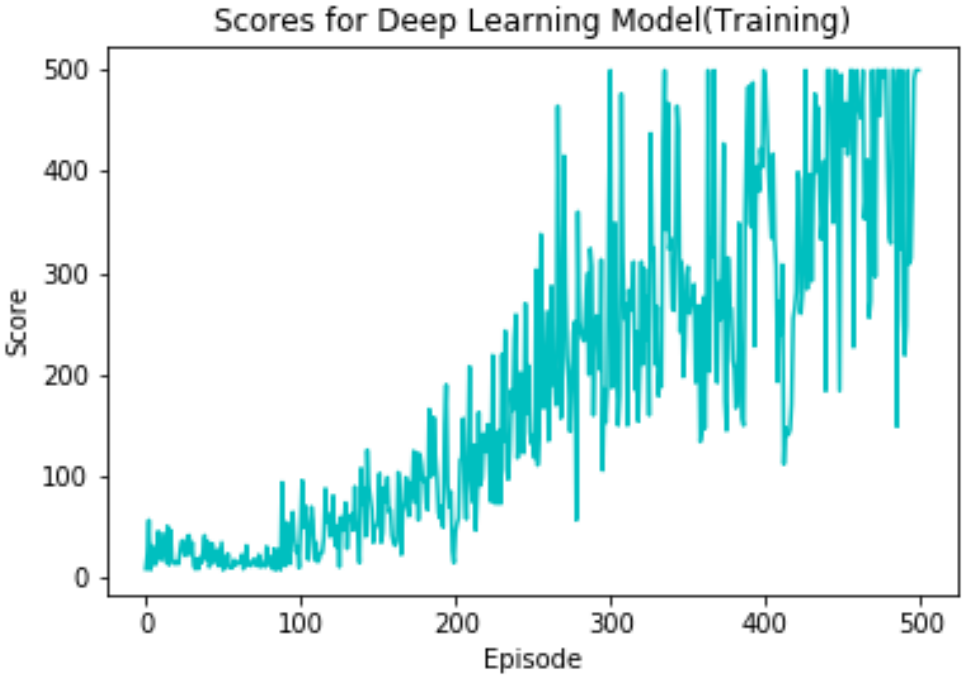


Figure 5.8 : Score vs Episodes

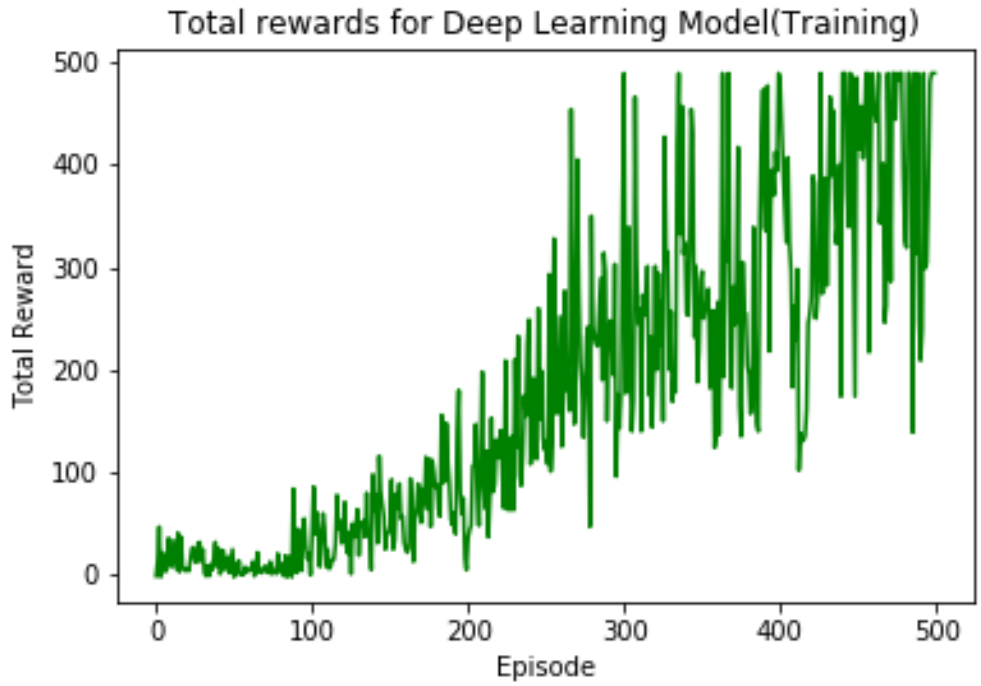


Figure 5.9 : Reward vs Episode

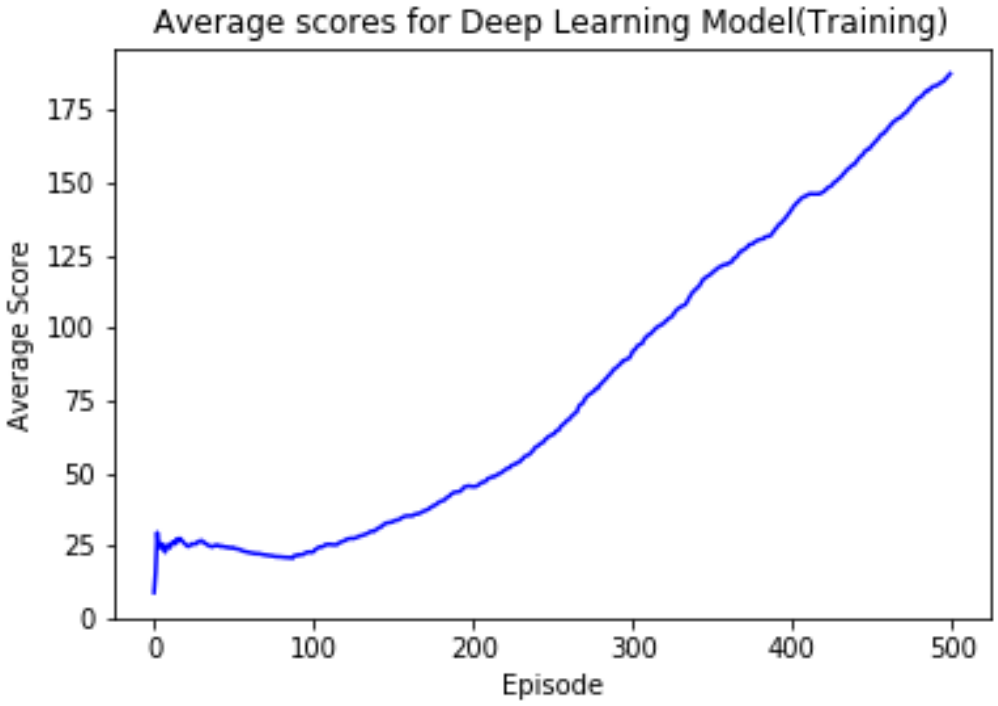


Figure 5.10 : Average Score vs Episode

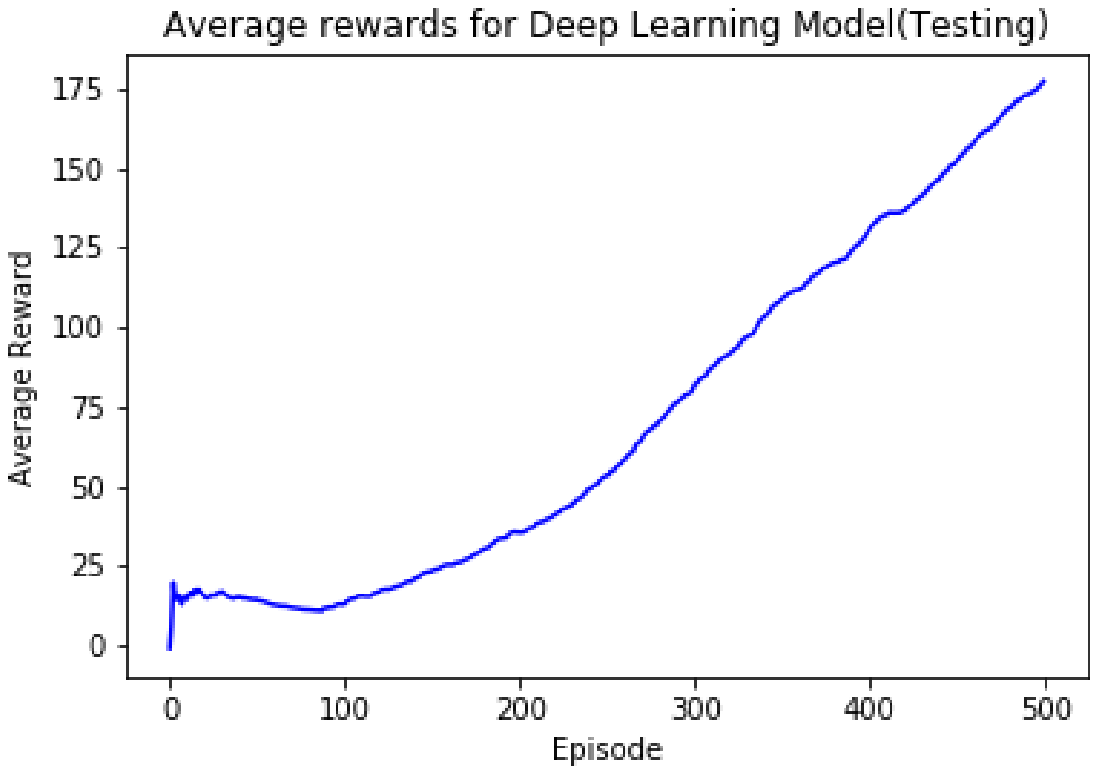


Figure 5.11 : Average Reward vs Episode

5.2.1 Testing Curves (Q-Learning Agent vs DQN Agent)

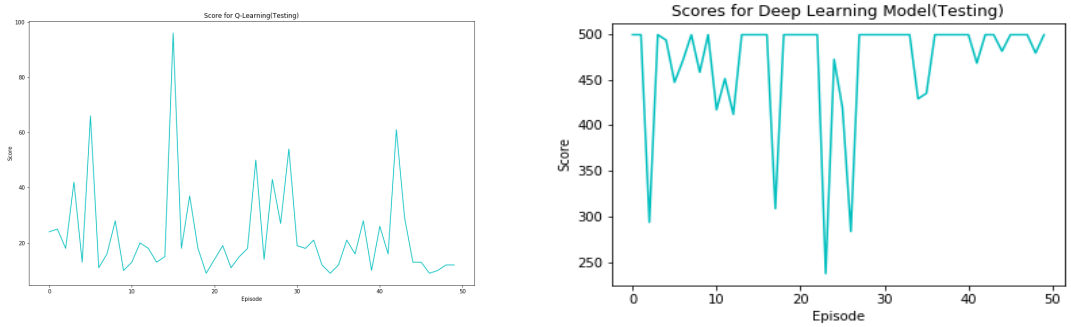


Figure 5.12 : Score vs Episode

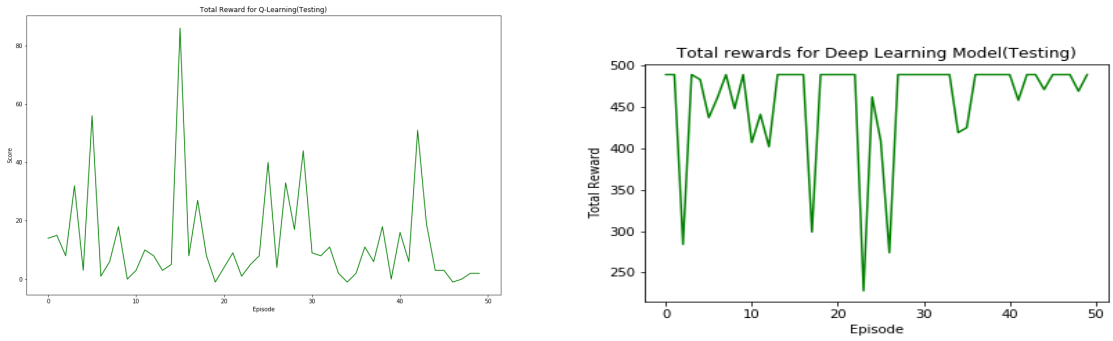


Figure 5.13 : Reward vs Episode

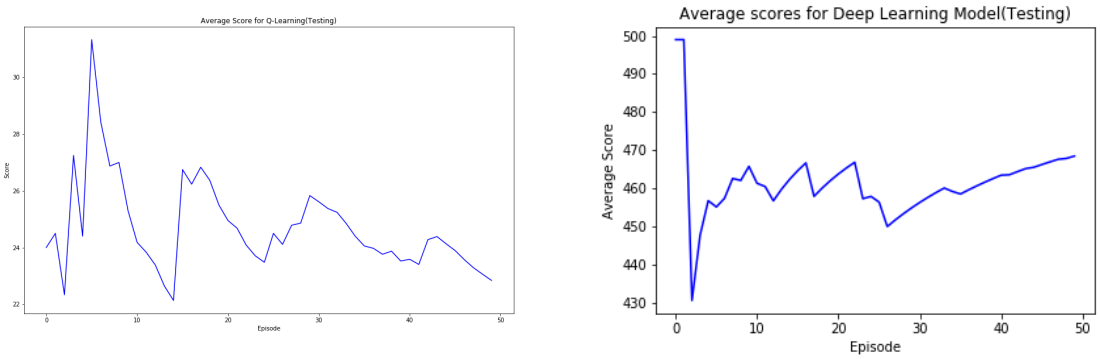
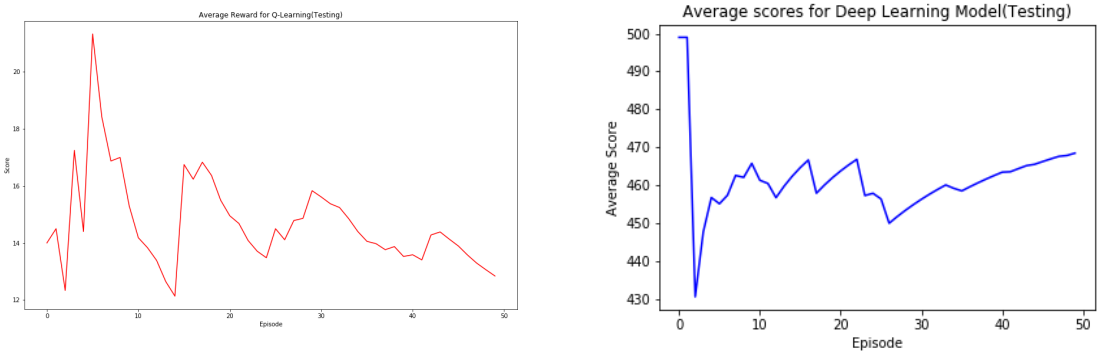


Figure 5.14 : Average Score vs Episode



5.15 : Average Reward vs Episode

5.2.1 Comparison of testing results of Q-Learning Agent vs DQN-Agent vs Random Exploration.

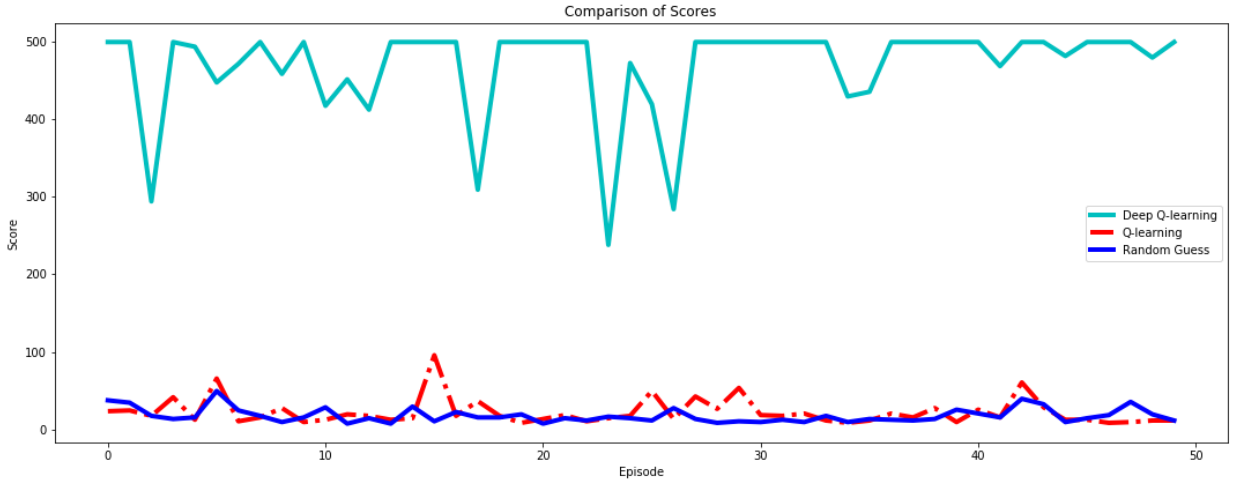


Figure 5.16 : Score vs Episode

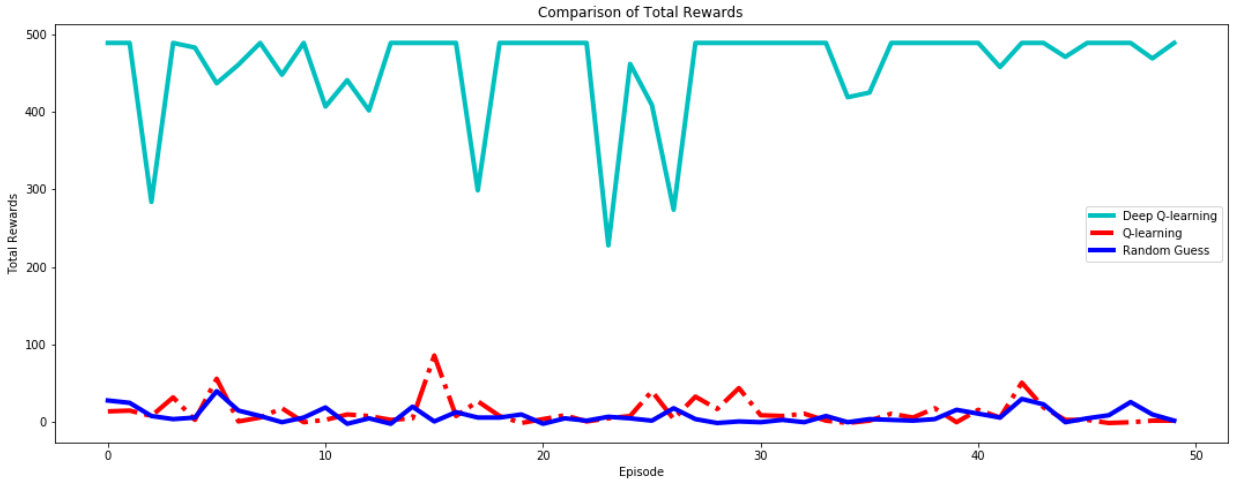


Figure 5.17 : Reward vs Episode

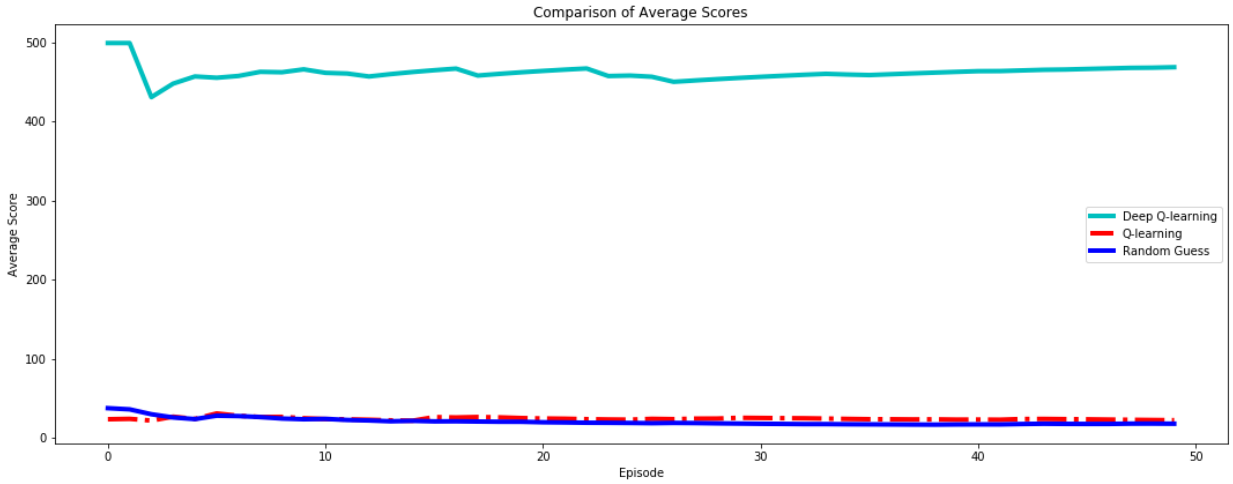


Figure 5.18 : Average Score vs Episode

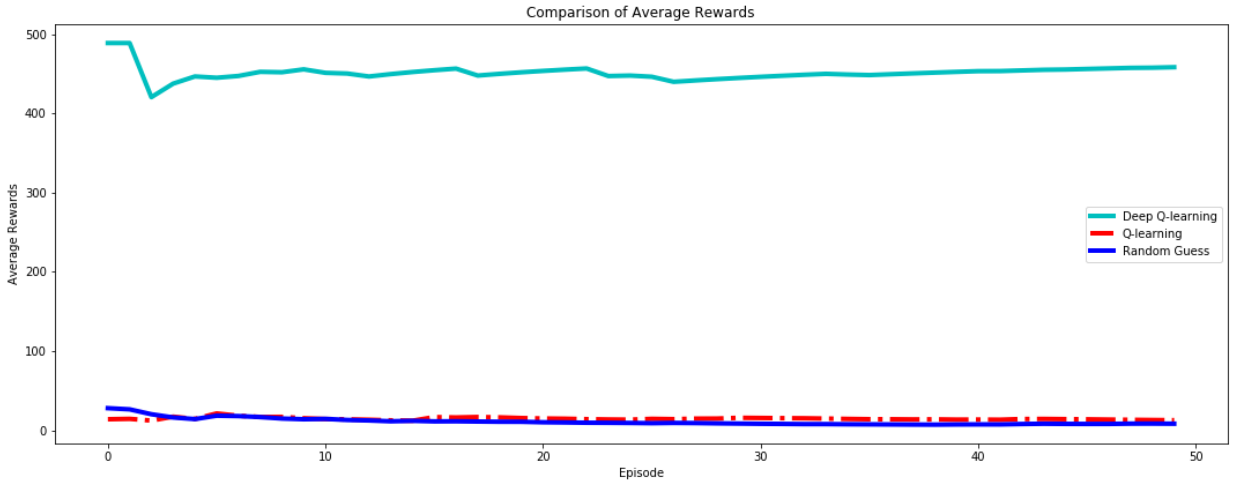


Figure 5.19 : Average Reward vs Episode

5.2 Conclusion

The Results from the above simulation clearly show that Q-Learning shows remarkable performance in environments with a small State-Space (here, Smartcab : state-space-size : 384 States), however fails to achieve the same in environments of higher dimensions.

The Q-Learning agent does performs better than Random-Exploration, but fails to achieve higher rewards. The reason behind the ‘Curse of Dimensionality ’, due to which the size of the Q-table increases exponentially and hence it becomes impossible to iterate over all the possible policies even with a high number of training trials resulting in the poor performance in such high dimensional spaces.

Deep-Reinforcement Learning (here, Deep-Q Learning) manages to overcome this problem by following generalization. The aim of the model is to figure the latent trends in the data gathered by exploring the environment and hence generate a model that can predict the possibly correct (not necessarily optimal) action.

The results substantiate the same since the model manages to achieve very high scores and in many instances ‘a perfect score’ even in much lesser training trials. This strengthens the fact that the DQN-Agent overcomes the limitation of Policy Iteration Q-learning, to explore all states individually, and even with a much limited exploration can offer remarkable results and hence outperforms the Policy Iteration Q-Learning algorithm.

REFERENCES

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. Playing Atari With Deep Reinforcement Learning. NIPS Deep Learning Workshop, 2013.

[2] M.G. Bellemare, Y. Naddaf, J Veness and M. Bowling.The Arcade Learning Environment: An Evaluation Platform for General Agents. Journal of Artificial Intelligence Research 47, Pages 253-279. 2013

[3] Kristjan Korjus, Ilya Kuzovkin, Ardi Tampuu, Taivo Pungas. Replicating the Paper "Playing Atari with Deep Reinforcement Learning". Technical Report, Introduction to Computational Neuroscience, University of Tartu. 2013

[4] Marc G Bellemare, Joel Veness, and Michael Bowling. Investigating contingency awareness using atari 2600 games. In AAAI, 2012.

[5] Matthew Hausknecht, Risto Miikkulainen, and Peter Stone. A neuro-evolution approach to general atari game playing. 2013.

[6] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[7] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, MA: MIT Press, 1998.

[8] K. Arulkumaran, N. Dilokthanakul, M. Shanahan, and A. A. Bharath, "Classifying options for deep reinforcement learning," in Proc. IJCAI Workshop Deep Reinforcement Learning: Frontiers and Challenges, 2016.

[9] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep Q-learning with model-based acceleration," in Proc. Int. Conf. Learning Representations, 2016.

[10] [14] Hamid Maei, Csaba Szepesvari, Shalabh Bhatnagar, Doina Precup, David Silver, and Rich Sutton. Convergent Temporal-Difference Learning with Arbitrary Smooth Function Approximation. In Advances in Neural Information Processing Systems 22, pages 1204–1212, 2009.

[11] M. Kearns, Y. Mansour, and A. Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. Machine Learning, 49(2-3):193–208, 2002.
[12] [Thrun et al., 2000] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust monte carlo localization for mobile robots. Artificial Intelligence, 101:99–141, 2000.
[13] [Astrom, 1965] K. J. Astrom. Optimal control of Markov decision processes with incomplete state estimation. J. Math. Anal. Applic., 10:174–205, 1965.
[14] Varsha Lalwani and Masare Akshay Suni. Playing Atari Games with Deep Reinforcement Learning. Technical Report, IIT Kanpur. 2015
[15] M. Denil, P. Agrawal, T.D. Kulkarni, T. Erez, P. Battaglia, N. de Freitas, "Learning to perform physics experiments via deep reinforcement learning", *Proc. Int. Conf. Learning Representations*, 2017.
[16] J. Foerster, Y.M. Assael, N. de Freitas, S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning", *Proc. Neural Information Processing Systems*, pp. 2137-2145, 2016.
[17] N. Heess, J.J. Hunt, T.P. Lillicrap, D. Silver, "Memory-based control with recurrent neural networks", *NIPS Workshop on Deep Reinforcement Learning*, 2015.
[18] M. Jaderberg, V. Mnih, W.M. Czarnecki, T. Schaul, J.Z. Leibo, D. Silver, K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks", *Proc. Int. Conf. Learning Representations*, 2017.
[19] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. de Maria, V. Panneershelvam, M. Suleyman et al., "Massively parallel methods for deep reinforcement learning", *ICML Workshop on Deep Learning*, 2015.
[20] E. Tzeng, C. Devin, J. Hoffman, C. Finn, X. Peng, S. Levine, K. Saenko, T. Darrell, "Towards adapting deep visuomotor representations from simulated to real environments", *Workshop Algorithmic Foundations Robotics*, 2016.
[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, “Playing Atari with Deep Reinforcement Learning”, arXiv:1312.5602v1 [cs.LG] ,2013