



DEEP REINFORCEMENT LEARNING IN ATARI GAMES

By

Andre Leonardo Angkawijaya

001201600010

A Thesis
Submitted to the Faculty of Computing
President University
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science
in Information Technology

Cikarang, Bekasi, Indonesia

March 2020

Copyright by
Andre Leonardo Angkawijaya
2020

DEEP REINFORCEMENT LEARNING IN ATARI GAMES

By

Andre Leonardo Angkawijaya

Approved:

Dr. Tjong Wan Sen, S. T., M. T.
Thesis Advisor

Nur Hadisukmana, M.Sc
Program Head of Information Technology

Rila Mandala, Ph.D
Dean of Faculty of Computing

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other University or Institution.

I have read the Thesis Regulations and I am aware of the potential consequences of any breach of them.

Cikarang, March 3, 2020

Andre Leonardo Angkawijaya

ABSTRACT

Data is an important foundation to analyze customer needs, personalize marketing campaign, predict the upcoming business trend, and create various smart system. Many companies such as Amazon, Tesla, and Google have successfully used *artificial intelligence* to benefits human through a lot of smart system. The newest trends show an artificial intelligence *agent* which can learn and choose an action suitable to its current environment by itself. This learning method is called *reinforcement learning* which become the main studies of this thesis.

The author studies the algorithm in the simulated environment, an Atari game called Ms. PacMan, with the goal to develop an agent that excel in the game mechanic. A basic reinforcement learning algorithm called *Deep Q Network* is chosen to be implemented. The agent is trained by the environment raw pixel images and the action list information. The experiments conducted by using this algorithm shows the agent's decision-making ability in choosing a favorable action. However, the agent performance is not stable and perfect which indicates the needs of more uninterrupted training session.

DEDICATION

*For Him, my family, and friends
who always support me through any means.*

ACKNOWLEDGMENTS

The author wishes to express his utmost and genuine gratitude to those who supports him through any means during the studies conducted for this thesis:

1. God almighty, my savior and stronghold, Jesus Christ, for His grace and love that always support me so that I could stand here today.
2. My family who always support me in their prayer, works, and love in my entire life.
3. My thesis advisor, Mr. Tjong Wan Sen, for his guidance during the whole development phase.
4. The President University faculty of computing's lecturer for their time, guidance, and knowledge who shape me into what I am today.
5. All of my Computing friends whom I shared the stories and memories with during the development phase or even the whole university life, especially the Horrible People.

TABLE OF CONTENTS

	Page
DEDICATION	ii
ACKNOWLEDGMENTS	iii
TABLE OF CONTENTS.....	iv
LIST OF TABLES	vii
LIST OF FIGURES	viii
 I. INTRODUCTION	 1
1.1 Background.....	1
1.2 Problem Statement	3
1.3 Research Objective	3
1.4 Scope and Limitation	4
1.5 Methodology	4
1.6 Thesis Outline	6
 II. LITERATURE STUDY.....	 8
2.1 Artificial Intelligence and Neural Network	8
2.2 Machine Learning	11
2.2.1 Supervised and Unsupervised Learning.....	11
2.2.2 Reinforcement Learning	11
2.2.2.2 The Elements	14
2.3 Reinforcement Learning Basics.....	15
2.3.1 The Markov Property	15
2.3.2 The Markov Decision Process	15
2.3.3 Value Function and Q-Value Function	17
2.4 Ms. PacMan Environment	18
2.4.1 States	18
2.4.2 Actions	18
2.4.3 Rewards.....	18

2.5	Reinforcement Learning Algorithm.....	20
2.5.1	Q-Learning.....	21
2.5.2	Deep Q Network	25
2.6	Atari Games Environment	26
III.	SYSTEM ANALYSIS	28
3.1	System Overview	28
3.2	Hardware and Software Requirement.....	28
3.3	Functional Analysis	30
3.4	Use Case Diagram.....	31
3.5	Use Case Narrative	32
3.6	Activity Diagram	37
IV.	SYSTEM DESIGN	40
4.1	User Interface Design	40
4.2	Physical Design.....	40
4.3	Data Design.....	41
4.4	Class Diagram.....	42
V.	SYSTEM IMPLEMENTATION	49
5.1	The Library	49
5.2	The Network Class.....	50
5.2.1	The Constructor	51
5.2.2	The Build Network Function	53
5.2.3	The Train Function	56
5.2.4	The Predict Functions	58
5.3	The Agent Class.....	58
5.3.1	The Constructor	58
5.3.2	Storing Memory Function.....	61
5.3.3	Choosing Action Function	61
5.3.4	Update Epsilon Function.....	63
5.3.5	Sampling Function.....	63
5.3.6	Max Q Function	64
5.3.7	The Learning Function.....	64
5.4	The Main Class	65
5.4.1	The Constant	65
5.4.2	Data Preprocessing.....	66
5.4.3	The Plotting Function	67

5.4.4	The Evaluation Function.....	73
5.4.5	The Main Function.....	81
VI.	SYSTEM TESTING	88
6.1	Testing Environment.....	88
6.2	Testing Scenario.....	88
6.2.1	Training Time and Scenarios.....	89
6.2.2	Initial Result.....	89
6.2.3	Case A.....	91
6.2.4	Case B	92
6.2.5	Default Case.....	93
6.3	Additional Experiment.....	98
VII.	CONCLUSION AND FUTURE WORK	101
7.1	Conclusion	101
7.2	Future Works	102

LIST OF TABLES

TABLE	Page
Table 2.1 Ms. PacMan Reward Space – The Foods and Ghost Eating Scores.....	19
Table 2.2 Ms. PacMan’s Reward Space – The Bonus Fruits.....	19
Table 2.3 Initial Q Value Table of The Dog Agent	23
Table 3.1 Use Case Narrative of Train the Agent Node	32
Table 3.2 Use Case Narrative of Load Training’s Checkpoint File Node.....	34
Table 3.3 Use Case Narrative of Show Agent’s Performance Logs Node	35
Table 4.1 Software Requirements.....	41
Table 4.2 Hardware Requirements	41
Table 4.3 The Data Design	42
Table 6.1 Testing Scenario	89

LIST OF FIGURES

FIGURE	Page
Figure 2.1 The Perceptron.....	9
Figure 2.2 Forward Propagation	10
Figure 2.3 Backward Propagation.....	10
Figure 2.4 Reinforcement Learning Robot Learning to Avoid Bad Decision	12
Figure 2.5 Agent's Exploration and Exploitation Rate Over Training Session.....	14
Figure 2.6 A Non-Exhaustive, but Useful Taxonomy of Algorithms in Modern RL.....	20
Figure 2.7 A Dog Agent in 2x2 Grid World	22
Figure 2.8 Q Value Update Calculation & The Updated Q Values Table.....	24
Figure 2.9 The Algorithm for Deep Q-Learning with Experience Replay [4]	26
Figure 2.10 Gym Retro Screenshots Collage Showing Atari and Sega Games Environment.....	27
Figure 3.1 The Functional Analysis Diagram	31
Figure 3.2 The Use Case Diagram.....	32
Figure 3.3 Initiate Training Activity Diagram	38
Figure 3.4 Begin Training Activity Diagram.....	39
Figure 4.1 The Class Diagram	43
Figure 5.1 The Network Class' Required Library	50
Figure 5.2 The Network's Class Constructor	51
Figure 5.3 The 1 st Section of Build Network Function.....	54
Figure 5.4 The 2 nd Section of Build Network Function.....	55
Figure 5.5 The Model's Architecture	56
Figure 5.6 The Train Function	57
Figure 5.7 The Predict Function	58
Figure 5.8 The Agent's Hyperparameters.....	59
Figure 5.9 The Agent's Memory Properties	59
Figure 5.10 The Agent's Network	60
Figure 5.11 The Agent's Class.....	60
Figure 5.12 The Store Transition Function.....	61
Figure 5.13 The Policy Function	62
Figure 5.14 The Exploration Code.....	62
Figure 5.15 The Exploitation Code.....	63
Figure 5.16 The Update Epsilon Function.....	63
Figure 5.17 Random Mechanism to Choose a Batch of Episodes from Memory	63
Figure 5.18 The Sampling Function	64

Figure 5.19 The Max Q Function	64
Figure 5.20 The Learning Function	65
Figure 5.21 The Constants	65
Figure 5.22 The Preprocess and Shift Function.....	66
Figure 5.23 The Average Scores Graph.....	67
Figure 5.24 The 1 st Section of The Plot Function.....	68
Figure 5.25 The 2 nd Section of The Plot Function.....	68
Figure 5.26 The 3 rd Section of The Plot Function	69
Figure 5.27 The 4 th Section of The Plot Function.....	69
Figure 5.28 The 5 th Section of The Plot Function.....	70
Figure 5.29 The 6 th Section of The Plot Function.....	71
Figure 5.30 The 7 th Section of The Plot Function.....	71
Figure 5.31 The Plotting Function (Part 1).....	72
Figure 5.32 The Plotting Function (Part 2).....	73
Figure 5.33 The 1 st Section of Evaluate Function.....	74
Figure 5.34 The Preprocessed and Stacked initial game frames	75
Figure 5.35 The 1 st Section of Preprocess Function	75
Figure 5.36 The Transformation of Real Game Frame into Preprocessed Frame	76
Figure 5.37 The 2 nd Section of Preprocess Function	76
Figure 5.38 The 2 nd Section of Evaluate Function.....	77
Figure 5.39 The Shift Function.....	78
Figure 5.40 The Shifting Frame Mechanics	78
Figure 5.41 The 3 rd Section of Evaluate Function	79
Figure 5.42 The 4 th Section of Evaluate Function	79
Figure 5.43 The Evaluate Function (Part 1).....	80
Figure 5.44 The Evaluate Function (Part 2).....	81
Figure 5.45 The Main Setup	82
Figure 5.46 The Main Episode Loop.....	83
Figure 5.47 The Agent's Decision-Making Mechanics.....	84
Figure 5.48 The Experience Replay and Training Mechanics.....	85
Figure 5.49 The Agent's Evaluation Mechanics.....	86
Figure 5.50 The Plotting Mechanics.....	86
Figure 5.51 The Output Log	87
Figure 6.1 Testing Agent's Initial Behaviour	90
Figure 6.2 Testing Agent's Collide with a Ghost	90
Figure 6.3 The Agent Out of Lives.....	91
Figure 6.4 The Training Result from Episode 80 – 110 in Case A	92
Figure 6.5 The Training Result from Episode 80 – 110 in Case B.....	93
Figure 6.6 The First Phase Scores History Graph.....	94

Figure 6.7 The Second Phase Scores History Graph	95
Figure 6.8 The Third Phase Scores History Graph	96
Figure 6.9 The Third Phase Average Scores Graph	97
Figure 6.10 The Breakout's Game Screen.....	98
Figure 6.11 The Breakout Average Scores Graph.....	99
Figure 6.12 The Agent's Score Progression Throughout 1000 Breakout's Episodes	100

CHAPTER I

INTRODUCTION

The introduction consists of the Background, Problem Statement, Research Objectives, Scope and Limitation, Research Methodology and Thesis Outline.

1.1 Background

Technology is developed to help human in performing a complicated works that either involved dangerous works or complex computation. Inventions such as computer and smartphone are some good examples of technologies development that enables human to work in a smart, simple, and efficient manner through a variety of smart programs. An example of the smart program is the virtual intelligence assistant developed by Google which can recognize and processed our voice as an input, the Google Assistant. A smart program has a trainable intelligence that will get better in recognizing voice and processing task as long as it got a decent amount of input and training time. This man-made intelligence is called *Artificial Intelligence* (AI).

The long development goal of AI is to achieve the ability for the machine to think and act both rationally and humanly in solving any intellectual human task which is called *Artificial General Intelligence* (AGI). However, the current technology still cannot achieve the main goal of AI as it is a difficult task to create a machine that could think and act humanly.

AI works by receiving inputs, calculates, and then show the predictions as an output. At the beginning, the AI will show a less accurate prediction as they still not have learned enough. Carving an intelligence into machine needs an iteration of learning process which is called *Machine Learning* (ML).

Google DeepMind and OpenAI are companies that shows AI potential in solving problem that can be trained in simulated environment. *Reinforcement Learning* (RL), one of the ML method, is utilized by these companies to train an expert agent that outperforms humans in game. The latest success is shown by the agent created and trained by DeepMind, the *AlphaGo Zero*, which is able to defeat the 18-time world champion Lee Sedol in the game of Go [11]. OpenAI agent is also able to defeat the three best DOTA 2 player in the world in 1v1 match and it puts a tough battle in five bots versus five players mode [7].

The utilization of game in training the RL agent is to describe the complex and high dimensional real-world data. By utilizing game, RL researcher will be able to evade high experimental cost in training an agent to do intelligence task. For example, in the case of a researcher that would want to create a self-driving car. By using game or any simulated environment to train the self-driving car agent will highly reduce the cost that will be used to train the agent in the real life, such as the cost for the hardware material that will be broken during the real-world test. Strategic games can also be utilized to train an agent to

excel in some business problem in different field such as marketing strategies, stock exchange prediction, personalized advertisement, enterprise resource planning, and so on.

Seeing the potential of RL methods, the author studies and implement the *Reinforcement Learning* (RL) algorithm to create an agent that is excel in playing Atari games. The training results will be an agent that is excel in playing different kinds of Atari games environment. To begin with, the agent will be trained to play a single Atari game called *Ms. PacMan*.

1.2 Problem Statement

Data enables business to wins through unique strategy such as predicting the upcoming business trends, creating personalize marketing, constructing stock prediction strategy, etc. AI that is excel in prediction and strategic problem could be utilized to solve problem in various field, even though it will be challenging to be solved as real-world problem has a huge and dynamic environment that made it impossible for researchers to represents in an algorithm. An algorithm that enables AI to explore the environment and create a better strategy to solve a problem is needed to solve a complex real-world problem.

1.3 Research Objective

This study is conducted with the following objectives:

- Analyse reinforcement learning algorithm capability in creating an agent that could solve various problem by exploring different kind of simulated environment.
- To implement reinforcement learning as the machine learning algorithm in creating an agent that could outperform human in Atari game.
- To analyse the agent performance in a limited time training session.
- To provide a reference material for students who are interested in this field that would become a base for a more suitable learning process to solve real world problem.

1.4 Scope and Limitation

This RL algorithm used in this study is the Deep Q Network algorithms that is ran in the Atari games environment. The aspects looked into are the implementation of the algorithm and the agent's performance the long run. The agent may not be perfect as the time and hardware requirement limited the study.

1.5 Methodology

The author uses the Rapid Application Development (RAD) model of software development process. RAD is an agile project management strategy popular in fast paced development environment. RAD reduces planning time and prioritize on delivering project prototype releases. Therefore, RAD ensures an effective communication and easy progress

measurements to check on improvement and issues. In this model, the whole software development process is divided into four main phases:

- **Requirements Planning** – The phase where the current problem's requirements are researched, defined, and finalized. This phase is shorter compared to other requirements planning phase in other software development model. In this phase, everyone has an opportunity to address problem, goals, and expectation to avoid miscommunications.
- **User Design** – The development process started right after the requirements are researched. In user design phase, the client evaluates each prototype that is given by the developer to ensure that the needs are being met at every design step. This method gives the user and developer the experience to learn from the mistakes and reduce undetected errors.
- **Rapid Construction** – The software development team built the working model from the prototype created from the user design phase. The developers began to prepare for the working model construction including the testing. During this process, the user also still able to give input until the end of the development. The user can give any idea, changes, and alterations to help in tackling any problem that arise.

- **Cutover** – The final phase of RAD model is the implementation phase where the developed program is ready to be launched. The developers will continually search and solved any occurring bugs.

1.6 Thesis Outline

The thesis consists of seven chapters, which are:

1. Chapter I: Introduction

Introduction consists of Thesis Background, Problem Statement, Research Objective, Scope and Limitation, Methodology, and Thesis Outline.

2. Chapter II: Literature Study

Literature Study describes the theoretical basis of references and guidance in the thesis creation.

3. Chapter III: System Analysis

System Analysis describes the analysis of the program – its behavior and function. It consists of System Overview, Hardware and Software Requirement, Use Case Diagram, Use Case Narrative, and Activity Diagram.

4. Chapter IV: System Design

System Design describes the definition of the program's architecture, components, and modules. It defines User Interface Design, Physical Design, Data Design, and Class Diagram of the program.

5. Chapter V: System Implementation

System Implementation describes how the application is implemented. It consists of User Interface Development and Application Details.

6. Chapter VI: System Testing

System Testing contains the testing documentation of the application. Included here are Testing Environment and Testing Scenarios, along with the results.

7. Chapter VII: Conclusion and Future Work

This chapter contains conclusion of the research. It also describes possible future improvements in section Future Work.

CHAPTER II

LITERATURE STUDY

Literature study aims to explain the core concept of the program development. The chapter is filled with the summary of the body of knowledge that is researched by the author.

2.1 Artificial Intelligence and Neural Network

Artificial intelligence is created to master a specific task. Similar to human, a machine receives inputs, calculates, and then shows the predictions of the input. Converting intelligence into a machine needs an iterative learning process which is called *machine learning*.

Artificial Intelligence works by the creation of a *Neural Network* which is a collection of *artificial neurons* connected together to do classification and prediction. A neural network is modelled and works after the brain, where the neurons are connected together through synapses to transmit signals to another neuron. The simplest form of a neural network is called a *Perceptron*. A Perceptron can be viewed as a building block that constructs the complex neural network which has four different parts which are input, weights, bias, and output. The perceptron is illustrated through figure 2.1 below.

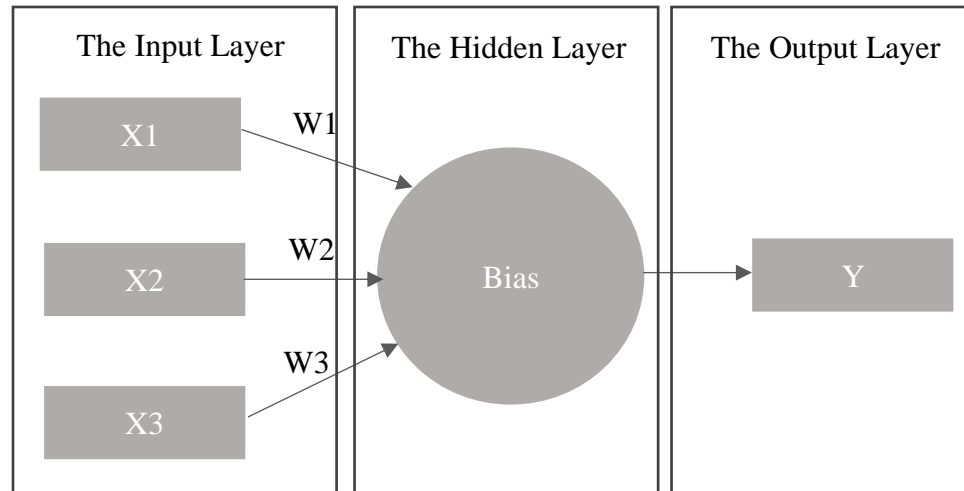


Figure 2.1 The Perceptron

Every neural networks constructed in three different layers. The *input layer* which are the initial data for the network that could be also gotten from the previous network. The *hidden layer* which is an intermediate layer between the input and output layer where all of the calculations is calculated. The *output layer* where the results of the calculations is printed. These layers are connected by a *connection* which is illustrated by the arrow. The connection illustrate the data flow from each node to another node.

The networks worked by calculating the input, weight, and bias that will produce a sum called the weighted sum. Weight is a number that represent the connection strength between nodes. Bias which always have a value 1 is an offset that ascertain the activation in the neuron. Bias allows the activation function to shift to get the better data prediction.

In the figure above, the inputs are denoted by X_1 , X_2 , and X_3 ; the weights are denoted by W_1 , W_2 , and W_3 ; and the output is denoted by Y . The weighted sum then will

be standardized by an *activation function*, which define whether a given neural network's node will be activated or not based on the weighted sum, to produce the output.

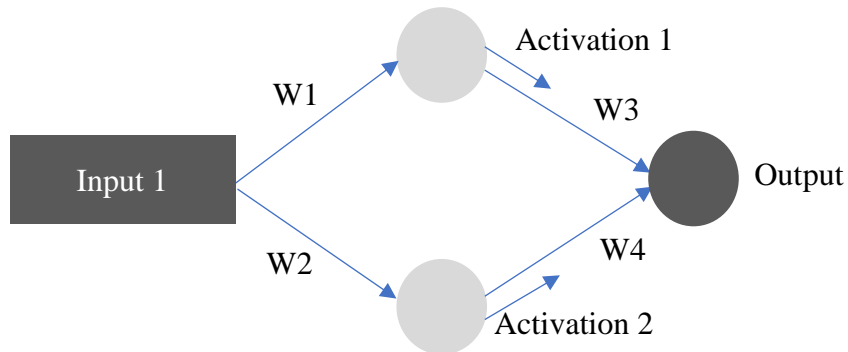


Figure 2.2 Forward Propagation

The process of moving forward the nodes, activates the node, and produce the output is called *forward propagation*, which is shown by the figure 2.2. At the end, the output is evaluated for the means to increase its accuracy. The error calculation will move backward from the output into the input layer through the same connections that hold the same weights that we use in the forward propagation [14]. This method is called *backward propagation*, which is shown by the figure 2.3.

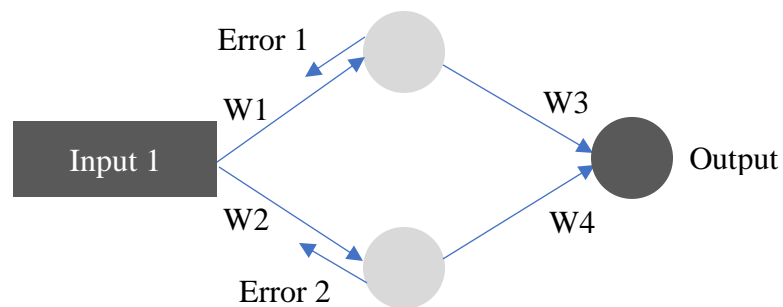


Figure 2.3 Backward Propagation

The errors that are calculated are proportional of a specific node's output. The backward propagation, or in short, *backpropagation*, is useful to allow researcher to calculate the the overall node's error for further tweaking and optimization. One method of tweaking the network is to manipulate the weights which defines the impact of a node to another node.

2.2 Machine Learning

In this section, the author describes four popular machine learning methods which called supervised, unsupervised, semi-supervised, and reinforcement learning [5].

2.2.1 Supervised and Unsupervised Learning

Supervised learning uses labelled or named data to trains the agents to predicts an outcome, for example, the agent is trained with a labelled fruit images to be able to differentiates fruit's name when it receives an image full of different kinds of fruits [3]. On the other hand, unsupervised learning agents does not used labelled data as a training material. The agents try to find the pattern and classify the provided data. Market research, social network analysis, and data clustering are the example of the cases who used unsupervised learning method.

2.2.2 Reinforcement Learning

Different with supervised and unsupervised learning, reinforcement learning applies the trial and error learning method. The agent learn the consequences of their

known actions in a specific environment with a discrete timestep [1]. After taking an action, the agent will receive either a positive or negative reward depending on the new environment's state. The positive reward indicated the taken action satisfies the requirement, whereas the negative reward does the opposite. Figure 2.4 below illustrates the behaviour of a reinforcement learning agent.

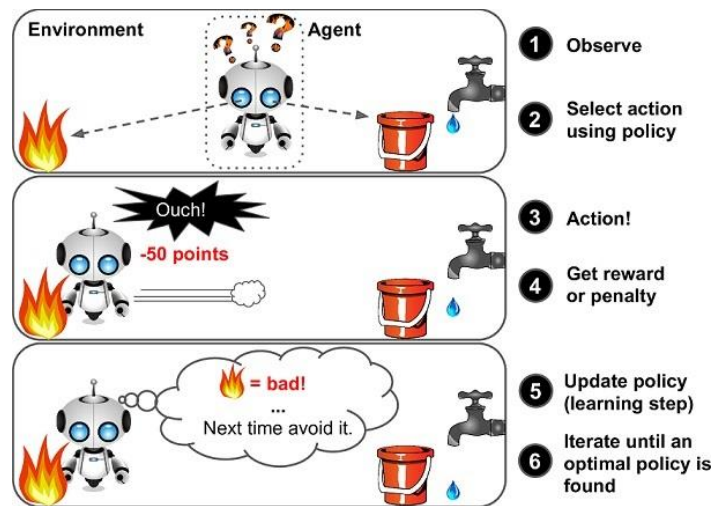


Figure 2.4 Reinforcement Learning Robot Learning to Avoid Bad Decision

The learning algorithm did not require any training data to be gathered before the training is conducted. Instead, a reinforcement learning agent is a goal-based agent which maximizes the *reward* that the agent can obtain through the means of *exploration* and *exploitation*. Through this action of traversing around numerous states, the agent gathers the environment's frame and feeds it into the neural network as the training input.

The learning focuses on addressing the problem of optimization, delayed consequences, exploration, and generalization. The agent should search for an optimal policy to make a good decision in taking an action to get a good reward. However, a good reward does not guarantee the agent's for a way of the expected good future. For example, if the agent have a tendency to always choose the best reward at the beginning of the episode. This behavior could close the agent's possibility to search for another route that could lead to an even greater reward.

In the reinforcement learning environment, the agent was given a set of action that the agent could took. To achieve reward maximization, the agent firstly try to *explore* the environment by taking random action to figure the best action which could lead to maximum the reward. After some time exploring the environment, the agent will goes on to *exploit* the action which could lead to the maximum possible reward that it could get. The researcher has to address the way to handle the trade off between exploration and exploitation to achieve the maximum reward. This problem is called *exploration-exploitation dilemma*. The agent's exploration and exploitation rate over training session is illustrated through figure 2.5 below.



Figure 2.5 Agent's Exploration and Exploitation Rate Over Training Session

2.2.2.2 The Elements

There are important terms that will be used when one studies reinforcement learning. These terms are *model*, *reward*, *policy*, and *value function*.

A *model* in reinforcement learning is a representation of the environment at a given state. The model contains the information of the current state (S), the taken action (A), the future state (S'), and the future reward (R).

A *reward* is the deciding factor that affect the agent behavior in choosing an action. In another word, it is a result of an action that the agent has taken that will decide whether the agent has taken a good or bad action. High or low reward does not indicate the agent successfulness as it depends on the algorithm to determine whether the agent favor a high or low reward. A *value* is not the same as reward, value is a total amount of reward accumulated by the agents for its lifetime while a reward is immediately given when the agent took an action.

A *policy* defines the reinforcement agent's behavior in taking an action at a certain state. Policy is the main aspect of the reinforcement learning where an optimal policy should be search in order to obtain favorable behavior.

2.3 Reinforcement Learning Basics

Few fundamentals theory of reinforcement learning will be briefly explained through this section.

2.3.1 The Markov Property

The state in the reinforcement learning should satisfy the Markov Property. The markov property define that a current state completely characterizes the state of the world, hence the current state is independent both towards the future and past state [15]. In the agent's environment, the agent transition to another state through the taken action. If the next state could be predicted without knowing/dependent to the preceded events, then the mathematical equation of the property is given in equation 2.1.

$$\Pr\{s_{t+1} = \acute{s}, r_{t+1} = r \mid s_t, r_t, a_t, s_{t-1}, a_{t-1}, \dots r_1, s_0, a_0\} \quad (2.1)$$

2.3.2 The Markov Decision Process

Serena Yeung explain that the Markov Decision Process is the mathematical formulation of the reinforcement learning defined by the tuple $(S, A, \mathcal{R}, \mathbb{P}, \gamma)$ [15]:

- S represents the set of possible states
- A represents the set of possible actions

- \mathcal{R} represents the distribution of the reward given a (state, action) pair
- \mathbb{P} represents the transition probability i.e. distribution of the next state given (state, action) pair
- γ represents the discount factor

The Markov Decision Process works will be represented as the main task of the reinforcement learning's agent which is described through the pseudocode below.

- *The agent initializes by sampling the environment initial state $s_0 \sim p(s_0)$.*
- *Then, from $t=0$ until done:*
 - *Agent select action a_t*
 - *Environment samples reward given the state and action given*
 $r_t \sim R(.|s_t, a_t)$
 - *Environment sample the next state $s_{t+1} \sim P(.|s_t, a_t)$*
 - *Agent receives reward r_t and move to the next state s_{t+1}*

Based on this, the agent policy can now be stated as $\pi_t(s, a)$ that specifies the choosing action mechanics for the agents in each state. The objective of the reinforcement learning agent is to find the optimum policy π^* that maximize the cumulative discounted reward $\sum_{t>0} \gamma^t r_t$.

The optimum policy (π^*) should be *stochastic* to be able to fulfill the Markov Property. Thus, to handle the randomness, the maximum expected sum of reward is taken. The optimum policy is shown through the equation 2.2.

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid \pi \right], \text{ with} \quad (2.2)$$

Initial state sampled from the initial state distribution $s_0 \sim p(s_0)$

Action sampled from the policy given the state $a_t \sim \pi(. \mid s_t)$, and

Next state sampled from the transition probability distribution $s_{t+1} \sim p(. \mid s_t, a_t)$

2.3.3 Value Function and Q-Value Function

Finding the optimum policy means that the agent has to learn the goodness of a state and the goodness of a state-action pair. The *value function* (V^π) shown by equation 2.3 is the expected cumulative reward from following the policy from a state that quantifies the good and bad state.

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi \right] \quad (2.3)$$

The *Q-Value function* (Q^π) at state, s , and action, a , is the expected cumulative reward from taking the action a on state s . The Q-Value function is shown in the figure 2.4 below.

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right] \quad (2.4)$$

2.4 Ms. PacMan Environment

Ms. PacMan environment satisfies the Markov Properties, as the agent does not need to know the previous state to predict the next state. For example, the agent does not need to know how the bonus fruit appear in the game, instead it could predict in the future to approach the bonus fruit when it does appear on the game screen.

2.4.1 States

The states that will be used in the thesis will be the preprocessed game frames. The state space consists a lot of states, because Ms. PacMan has 1293 distinct locations in the maze. A complete state of Ms. PacMan's model consists of the location of Ms. PacMan, the ghosts, the power pills, along with the ghost previous move and the information whether the ghost is edible.

2.4.2 Actions

The agent has nine action that could be performed at the game which are represented by a single integer. These actions are ['NOOP', 'UP', 'RIGHT', 'LEFT', 'DOWN', 'UPRIGHT', 'UPLEFT', 'DOWNRIGHT', 'DOWNLEFT'].

2.4.3 Rewards

Ms. PacMan's reward could be obtained by gathering the foods (Pac-Dot), bonus fruit (Fruit), power up item (Power Pellet), eating a ghost, and chain eating the ghosts. The reward list is shown through tables 2.1 and 2.2 below.

Table 2.1 Ms. PacMan Reward Space – The Foods and Ghost Eating Scores














	Pac-Dot	10 Points
	Power Pellet	50 Points
	1 Ghost	200 Points
	2 Ghost	400 Points
	3 Ghost	600 Points
	4 Ghost	800 Points

Table 2.2 Ms. PacMan's Reward Space – The Bonus Fruits

	Cherry	100 Points
	Strawberry	200 Points
	Orange	500 Points
	Pretzel	700 Points
	Apple	1000 Points
	Pear	2000 Points
	Banana	5000 Points

2.5 Reinforcement Learning Algorithm

OpenAI research in reinforcement learning through various papers line up a nearly accurate taxonomy of algorithms in modern reinforcement learning as shown by figure 2.6 below.

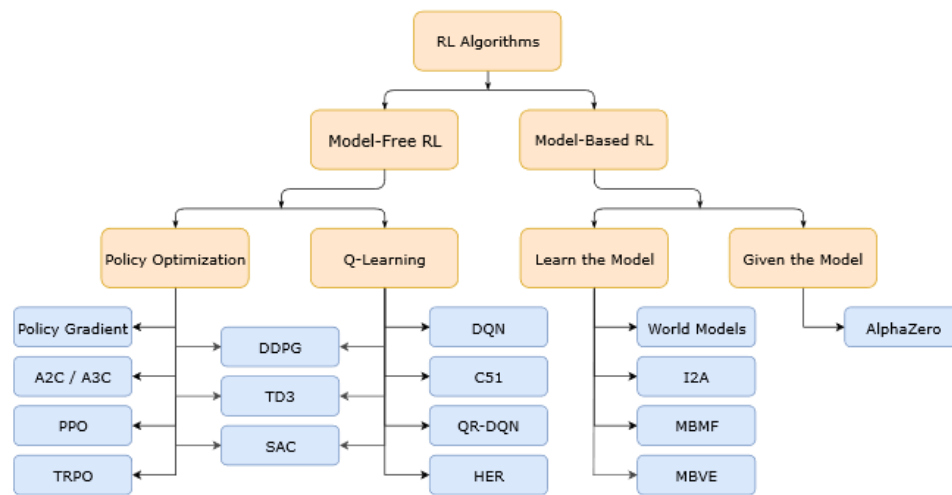


Figure 2.6 A Non-Exhaustive, but Useful Taxonomy of Algorithms in Modern RL

The reinforcement learning algorithms are separated into two different kinds related to the model which are *model-based* and *model-free* [16]. By using a model, the algorithm will be directed to *plan* future action depending on the information the agent has in the model. Model-based agents maintain the transition and reward function that will be used to resemble the world. The algorithm may or may not have a policy or value function.

The *model-free* agent on the other hand will learn by *trial and error* method to maximize the reward by updating the *policy* on the go. In conclusion, the different aspect

of a model-based and model-free agent will be the agent's capability in determining future state and reward.

For this study, the author used *Q-Learning*, where *Q* stands for *Quality*, method to train the AI. The variation that the author will used is the *Deep Q Network* (DQN) algorithm. The main difference between these two algorithm is in DQN a neural network is used to handle a large number of the states and actions pair. Hence, a Deep Neural Network will functioned as a function approximator. The details on both of this algorithm will be discussed through the sub chapter below.

2.5.1 *Q-Learning*

The Q-Learning use an action-value function, *Q*, to approximate the optimal action-value function, Q^* [9]. Q-learning utilize the *Bellman Equation* which is a mathematical equation that is mainly used to solve optimization problem and it is mainly utilized in *Dynamic Programming*. The equation that is utilized for reinforcement learning is shown below through figure 2.5.

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a) \quad (2.5)$$

The equation above is a processed Bellman Equation that is used to fit a state and action pair into it. The $Q(s, a)$, commonly referred as the Q Value, is calculated through the addition of the immediate reward $r(s, a)$ added by the maximum value of the highest

possible Q Value from the next state (s') in the response of taking an action (a) times a *discount factor* gamma (γ).

A discount factor, a number between 0 and 1, is used to control the importance of the *short-term* and *long-term reward* [17]. A short-term reward will make the agent to greedily take the highest reward as soon as possible when it has the chance to get it which created the exploitation behavior while the long-term reward does the opposites. Thus, the goal of Q-Learning is to maximize the future cumulative reward that could be achieved. The characteristic of this algorithm made it to be called a *greedy algorithm*.

To represent Q Values in solving real world problem, a table form is created to store all possible values from the pair of state and action. The Q Values then will be continuously updated through the game run by using the equation above. To better understand the Q-Learning, an example of a 2x2 grid games is depicted through figure 2.4.

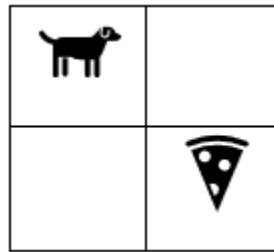


Figure 2.7 A Dog Agent in 2x2 Grid World

A dog is trying to reach the bottom right corner square to eat a slice of pizza. A four-number integer sequence of 0 and 1 is created to represent the environment of this example. The number represent the state of the square where the dog is located with the

sequence of the top-left, top-right, bottom-left, and bottom-right. The square where the dog is located will be labelled with an integer 1 while the square where the dog not in it will be labelled 0. Thus, the picture above is represented by 1000. The table below represent the initial Q Values table for each state and action that the dog can choose.

Table 2.3 Initial Q Value Table of The Dog Agent

Q-Values	1000	0100	0010	0001
Up	-0.1	-0.23	0.4	0
Down	0.75	0.4	-0.8	0
Left	-0.34	0.7	-0.2	0
Right	0.6	-0.23	1.0	0

As shown above, the state 0001 has zero for all of the Q Values showing that the agent has reach the terminal state. The negatives values represent that it is impossible for the dog agent to move beyond the game's border. For each cell that the dog is visited, if the dog did not gain the pizza, then he will get a zero as an immediate reward and one if he got a pizza. Currently, the dog agent is located at the state 1000. From figure 2.7, the dog can only move either to the right or to the bottom. In this scenario, if the dog chooses to move to the right cell, then the current information for the calculation of the Q Values update is stated and will be calculated as shown by the figure 2.8 below.

<i>Current state (s):</i> 1000	Q-Values	1000	0100	0010	0001
<i>Action (a):</i> Right	Up	-0.1	-0.23	0.4	0
<i>Reward (r):</i> 0	Down	0.75	0.4	-0.8	0
<i>Discount factor (γ):</i> 0.9	Left	-0.34	0.7	-0.2	0
<i>Next state (s'):</i> 0100	Right	0.63	-0.23	1.0	0
<i>Max Q:</i> 0.7	$Q(s, a) = r(s, a) + \gamma \max_{\alpha} Q(s', a) = 0 + 0.9 * 0.7 = 0.63$				

Figure 2.8 Q Value Update Calculation & The Updated Q Values Table

The example above shown the simple case scenario of the Q-Learning. To solve the problem of exploration and exploitation dilemma. An *epsilon greedy* approach is introduced. An *epsilon* (ϵ), which is a number between 0 and 1, will be updated periodically to get the probability (p) of the agent to take either exploration and exploitation behavior. The following equation shows one way to update the probability by utilizing epsilon.

$$p = 1 - \epsilon \text{ (Exploitation), or} \quad (2.6)$$

$$p = \epsilon \text{ (Exploration)}$$

However, when the environment has a lot of scenarios that needs to be calculated then Q-Learning will not be good enough in the term of speed and accuracy. Thus, the Deep Q Network is introduced to solve the problem in processing high dimensional data.

2.5.2 Deep Q Network

Deep Q Network, also referred as Deep Q Learning, combine the Deep Neural Network with the Q-Learning mechanics. The Q Values tables will be replaced by the neural network; thus, the neural network will be replacing the table function to approximate Q Values [18]. The goal of this algorithm is to minimize the difference between the initial learning state and the goal state where the Q Values reaches its final converged value, or on the other term, the best performance, which means we try to minimalize the cost function.

The cost function will be similar to the *mean square error* function which is a function to estimates the averages square of error. The cost function of Deep Q Network is defined as follow.

$$Cost = \left[Q(s, a; \theta) - \left(r(s, a) + \gamma \max_a Q(s', a) \right) \right]^2 \quad (2.7)$$

Where $Q(s, a; \theta)$ is the new state-action value function which also takes an additional parameter theta (θ) that symbolized the trainable weights of the neural network. The Deep Q Network is the main research's result presented a paper entitled "Playing Atari with Deep Reinforcement Learning" by Mnih et al [4]. Mnih's team introduces the implementation of Q-Learning variant into Deep RL algorithm, which are Deep Q-Network and Deep Q-Network Best, to create a single agent that is capable to achieve the highest score from seven different Atari Games. In their report, Mnih's team included the algorithm which is shown in the figure 2.9 below.

Algorithm 1 Deep Q-learning with Experience Replay

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
  Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
  for  $t = 1, T$  do
    With probability  $\epsilon$  select a random action  $a_t$ 
    otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
    Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
    Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
    Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
    Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
    Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
    Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
  end for
end for

```

Figure 2.9 The Algorithm for Deep Q-Learning with Experience Replay [4]

The Deep Q-Learning with experience replay algorithm will be the main algorithm that the author will implements.

2.6 Atari Games Environment

The uses of games environment as the means to simulated AI learning progress is a popular practice in training AI, especially to trains the AI that will take on a risky task. For example, the intelligence mechanical hand robot which have task on moving heavy or fragile object. If the training were to be conducted in real life, the business will only suffer a lot of damage through the training process. A lot of product or placeholder will be wasted just to train the hand's capabilities and the hand itself could be damaged through a lot of iterations of learning process.



CHAPTER III

SYSTEM ANALYSIS

3.1 System Overview

This thesis is intended to implement Deep Q Network Algorithm to train an agent in playing Atari game. The system will train the agent to the point that the agent is able to exploit the environment to achieve the best score as fast as possible. The main objective of the system is to create a system that were able to outperform human in playing games.

3.2 Hardware and Software Requirement

Listed below are the software and hardware that are needed to developed this application:

- Personal Computer

A personal computer is where the application resides and be developed, from its earliest stage to its deployment. This application is developed on a PC running 64-bit Windows 10 education.

- JetBrains PyCharm Professional 2019.2.1

PyCharm is a Python Integrated Development Environment (IDE) that can be downloaded for free with a student email. Developed by a Czech company,

JetBrains, PyCharm provides easy code analysis, jupyter notebook supports, an integrated unit tester, integration with version control systems (VCS), and supports web development with Django.

- Google Colaboratory

A free Jupyter Notebook like that allows researcher to write and execute Python code in the browser. The Google Colaboratory gives a powerful host resources that could be utilizes which are 100GB cloud storage and 12GB RAM (upgradeable to 25GB for free). However, the Google Colaboratory only allow 12 hours of active running session and 30 minutes of idle running session.

- Microsoft Office

Microsoft Office application, specifically Microsoft Word is used in the making of the application's documentation.

- Python 3.7.4

Python 3.7.4 is the programming language that is used in the development of the application. Some APIs listed below are used to support the development of the application:

- Tensorflow

Google's open source machine learning platform that could deliver an easy yet robust machine learning model.

- Tensorboard

Tensorboard provides visualization and tools needed for data machine learning experimentation. It can be used to track and plot metrics, save images, view histogram, and so on.

- Keras

A high-level neural network API that run on top of Tensorflow.

- Gym

OpenAI's toolkit for developing and comparing a variety of reinforcement learning algorithm. Using Gym also allows the researchers to create and define their own environment to test their algorithm.

3.3 Functional Analysis

The system will be divided into a smaller functional elements which describes the overall system workflow. The system is divided into three sub parts which are preparing the environment, training the agent, and update the agent. The functional analysis diagram could be seen from the figure 3.1 below.

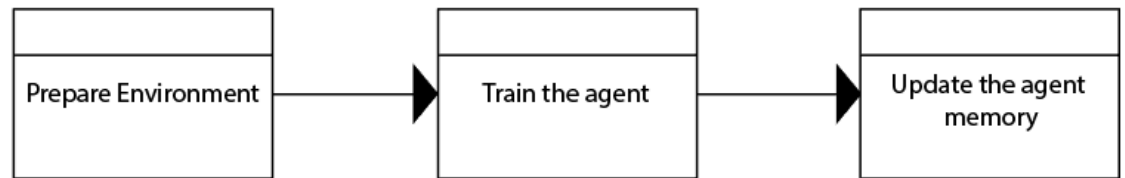


Figure 3.1 The Functional Analysis Diagram

3.4 Use Case Diagram

Use case diagram, depicted by figure 3.2, is a diagram describing dynamic behavior or technical concept of a system. It defines:

- Actors

The internal or external factors interacting with the system depicted by stickman

- Use Cases Nodes

A sequence of actions that provide something of measurable value to an actor depicted by circle.

- Associations Arrow

Dotted arrows which explains a node dependency toward another node.

- Unidirectional Associations Arrow

Arrow which explains a node association toward another figures.

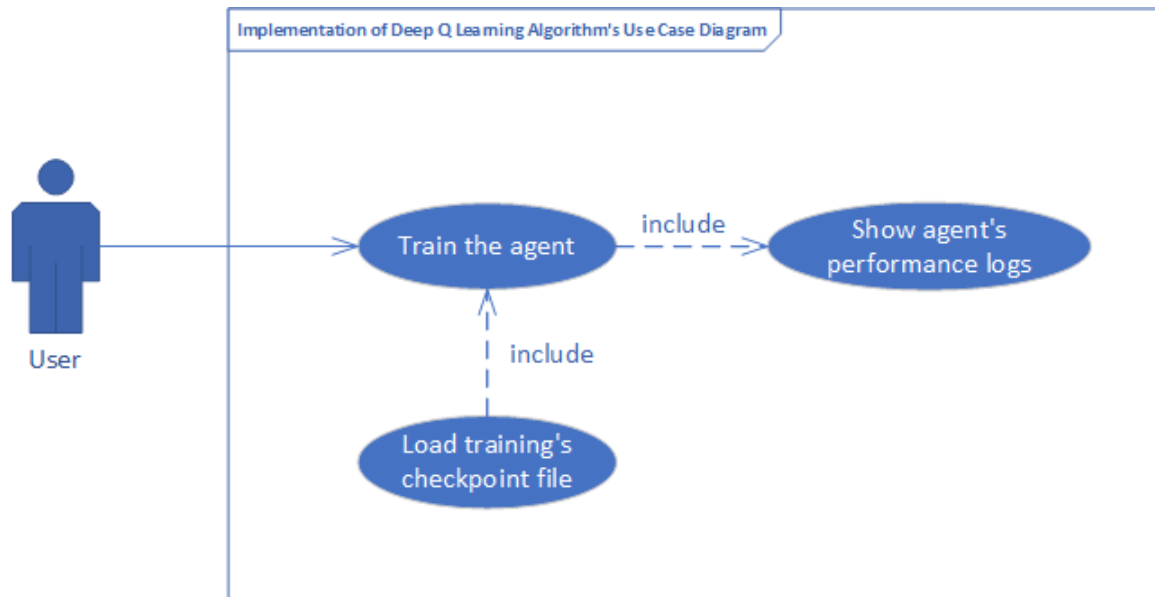


Figure 3.2 The Use Case Diagram

3.5 Use Case Narrative

A detailed textual representation of the sequence of events occurred during the interaction between an actor and the system in use case diagram will be explained in the narratives. The narrative is a table that will explain the prerequisites, conditions, expected results, alternative scenario, exception, goal, and post condition of a use case node. Use case narrative aims to clarify the system's behaviors from the early stages.

Table 3.1 Use Case Narrative of Train the Agent Node

Use Case Name	Train the Agent
Use Case ID	UC01
Priority	High

Primary Business Actor	User	
Primary System Actor	System	
Others Participating Actor	None	
Description	This use case represents the event when the user run the program to start the agent's training process.	
Precondition	The environment must be initialized	
Trigger	User click the run button	
Typical Course of Event	Actor Action	System Response
	<i>Step 1:</i> Type the environment's ID into the defined variable	
	<i>Step 2:</i> Click run button	
		<i>Step 3:</i> Verifies environment's ID
		<i>Step 4:</i> Load the environment
		<i>Step 5:</i> The training started
Alternate Course	<p>Alt 1.0: The user input a wrong environment's ID which cannot be initialized</p> <ul style="list-style-type: none"> - System display error message notifying the user that the environment does not exist <p>Alt 1.1: The user does not fill the environment's ID</p> <ul style="list-style-type: none"> - System display error message notifying the user that the environment ID is not specified 	

	<p>Alt 1.2: The training got interrupted by overheat which resulted in corrupted saved file</p> <p>Alt 1.3: The training got interrupted by memory/storage overflow</p> <ul style="list-style-type: none"> - System display error message notifying the user that the memory/storage is exhausted
Post Condition	The training results (trainable weights) are saved
Implementation Constraint and Specifications	None

Table 3.2 Use Case Narrative of Load Training's Checkpoint File Node

Use Case Name	Load training's checkpoint file	
Use Case ID	UC02	
Priority	High	
Primary Business Actor	User	
Primary System Actor	System	
Others Participating Actor	None	
Description	This use case represents the event when the user loads a specific training's checkpoint file to the application	
Precondition	<ul style="list-style-type: none"> - The environment must be initialized - The checkpoint file must exist in the defined load path 	
Trigger	User click the build and compile button	
Typical Course of Event	Actor Action	System Response

	<i>Step 1:</i> Type the load path into the defined variable	
	<i>Step 2:</i> Click run button	
		<i>Step 3:</i> Verifies checkpoint file existence in the load path
		<i>Step 4:</i> Load the trainable weight from the checkpoint file
Alternate Course	<p>Alt 1.0: The user input a wrong load path</p> <ul style="list-style-type: none"> - System display error message notifying the user that the checkpoint file is not exist in the defined load path <p>Alt 1.1: The user input a wrong checkpoint's file name</p> <ul style="list-style-type: none"> - System display error message notifying the user that the file does not exist 	
Post Condition	The agent's trainable weight is loaded	
Implementation Constraint and Specifications	None	

Table 3.3 Use Case Narrative of Show Agent's Performance Logs Node

Use Case Name	Show agent's performance logs
Use Case ID	UC03
Priority	High

Primary Business Actor	User	
Primary System Actor	System	
Others Participating Actor	None	
Description	This use case represents the event of showing the agent's performance tracker files (graph and log) while the training is still being conducted or terminated	
Precondition	The training is started	
Trigger	User click the build and compile button	
Typical Course of Event	Actor Action	System Response
		<i>Step 1:</i> Conducted training process
		<i>Step 2:</i> Verify whether the end of episode is reached
		<i>Step 3:</i> Log the episode's information and save the video of the agent's performance
		<i>Step 4:</i> Verify whether the currently ending episode is a multiple of 100
		<i>Step 5:</i> Create scores history and average scores graph
Alternate Course	Alt 1.0: The system failed to generate the agent's performance tracking files	

	<ul style="list-style-type: none"> - System display error message notifying the user that the storage is exhausted <p>Alt 1.1: The training process ended unexpectedly</p> <ul style="list-style-type: none"> - System display error message notifying the user that there is an error in the graph axis' elements dimension
Post Condition	<ul style="list-style-type: none"> - The scores history graph and average scores graph is saved as a picture - Every episode's information is logged into the console
Implementation Constraint and Specifications	None

3.6 Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. The diagram defines both concurrent and sequential activities which is used by the developers as a lineup of the technical flow of their works. This sub chapter will explain the activities diagram that are used in building this application. The *Train the agent* node activity flow is illustrated through the figure 3.3 below.

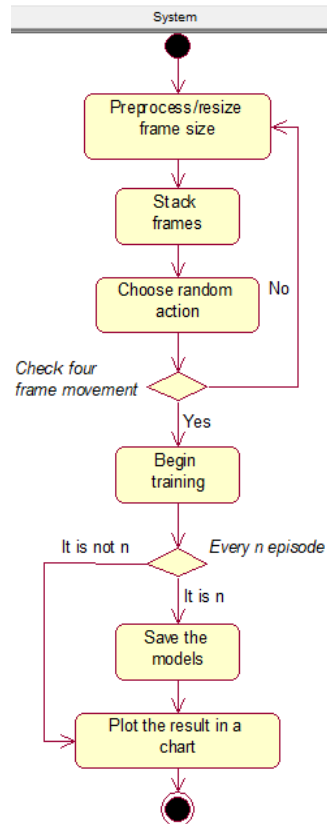


Figure 3.3 Initiate Training Activity Diagram

The system will be the sole actor of the *Initiate Training Activity Diagram*. To begin the training, the system will first *Preprocess/Resize* the environment's frame to delete meaningless details (extra spaces, scoreboard, etc.). Then, the next process is to *Stack Frames* so that the system could get the sense of motion. Next, the system will initiate the agent to take a *Random Action* to update the environment. If the system did not detect an update to the frames in *n-steps*, it will restart the process all over again from the *Preprocess* node. Otherwise, the system will *Begin the Training Session* that will be furtherly explained

in the next section that is depicted by figure 3.5. The last steps that will end the training session are to *Save* the trained model for each defined step and *Plot* the training results into the chart.

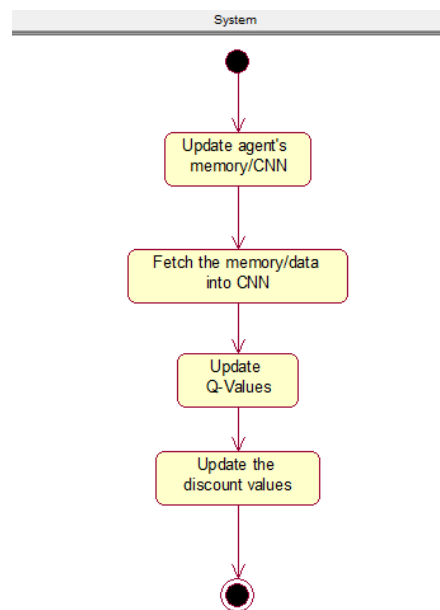


Figure 3.4 Begin Training Activity Diagram

The *Begin Training Activity Diagram* is the last activity diagram in this application. The training is started by initializing or *updating the agent's memory* with the newest taken action. Then, the knowledge that the agent got will be *fetched into the Convolutional Neural Network (CNN)*. After that, the *Q-Values will be updated* by the new reward function and the *Discount Values* will also be updated by the CNN

CHAPTER IV

SYSTEM DESIGN

System Design is the process of defining the architecture, components, modules, and data of a system to satisfy the specified requirements. It can be considered as the intermediate stage between System Analysis and Product Development where the analyses that have been made in the previous chapter are now visualized in detail. The application comprises of four sections: user interface design, physical design, data design, and class diagram.

4.1 User Interface Design

User interface design is one of the important components of any application, be it a desktop or a web application. The user interface will allow users to use the application easily. As the developed program is a neural network agent that is trained to play Atari games, no user interface is created in this study. The result of the training will be shown through the video created during the training session.

4.2 Physical Design

Physical Design defines the minimum requirements of software and hardware used in development process to ensure that the created application runs without problems. Table

4.1 and 4.2 below show respectively the software and hardware requirements to runs this application.

Table 4.1 Software Requirements

No	Field	Description
1	Operating System	Windows 10 Education
2	Programming Language	Python
3	Program Development	JetBrains PyCharm Professional 2019.2.1
		Google Colaboratory
4	Documentation	Microsoft Word 2019

Table 4.2 Hardware Requirements

No	Field	Description
1	Processor	Minimum requirement: Intel Core i series
2	Memory	Minimum requirement: 12GB of RAM
3	Monitor	Minimum requirement: Resolution 1280x720
4	Hard Drive	Minimum requirement: Free space of 100 MB

4.3 Data Design

Database is not needed in this application as the application purpose is only to train the agent to be excel in a specific game environment. Two checkpoint (.h5) data are generated, one is to save the training weights for every iteration and the other is created when highest average score is calculated. The last two data generated by the system are the

preprocessed environment file, which is an image format file (.png), and a video file of the game episode. Table 4.3 below depict the data design for the application.

Table 4.3 The Data Design

No	Filename	Content Description
1	checkpoint.h5	The training checkpoint for a defined number of iterations
2	[high_avg_score]_best.h5	The <i>[high_avg_score]</i> is filled with the actual high score that the agent received
3	preprocessed.png	The preprocessed environment data
4	openaigym.video.mp4	The video file generated by the open ai gym library

4.4 Class Diagram

A class diagram, as specified in UML, is a static diagram that describes the classes of a system. Classes here are classes in Object Oriented Programming context which displayed with its attributes, methods, and relations with other classes. Class diagram of this application is shown in figure 4.1.

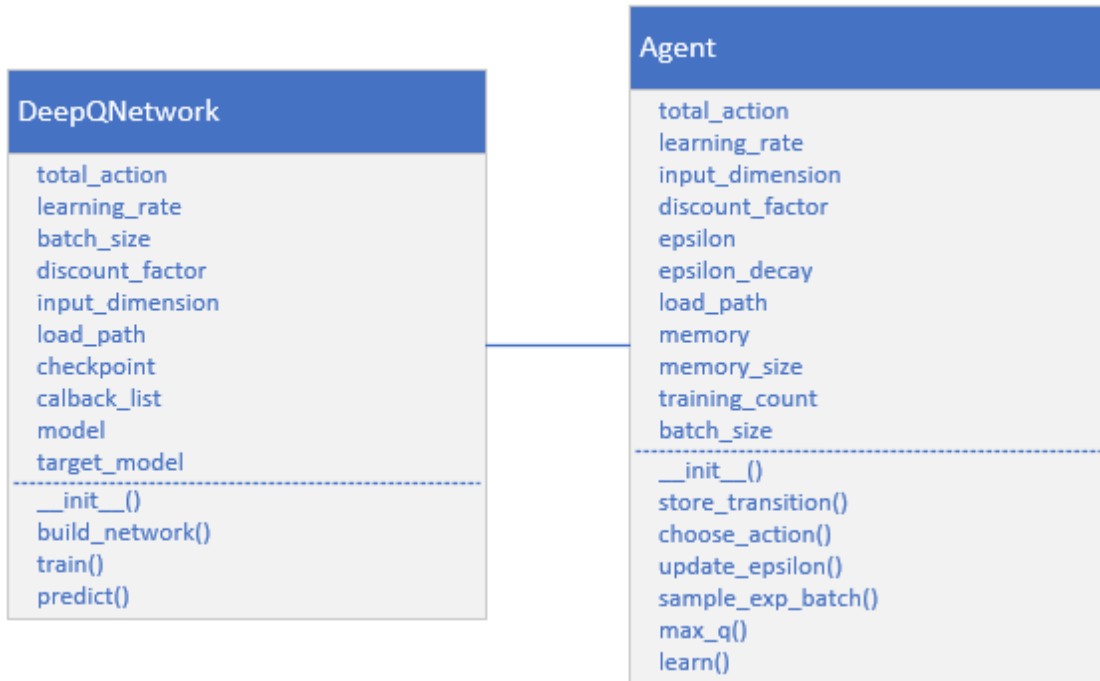


Figure 4.1 The Class Diagram

Two classes, *DQ_Network* and *Agent*, are used in this application. The further explanation of each class will be explained below:

a. DQ_Network Class

The DQ_Network Class main purpose is to build a Convolutional Neural Network for the agent. There exist nine properties to define this class which are:

- total_action

The total_action stands for a number of actions that the agent could take in the specific environment.

- learning_rate

The learning_rate is a variable which hold a floating-point value from 0 to 1 that defines the learning speed of the agent.

- batch_size

The batch_size is a variable whichhold an integer number defining the batch size in the Keras's fit function.

- discount_factor

The discount_factor is a variable which hold a floating-point value from 0 to 1 that defines the agent's behavior in considering long-term or short-term reward of the agent.

- input_dimension

The input_dimension is a variable which hold a three-dimensional array (row, column, channel) that defines the game's frame size. It will be utilized as an input for the neural network.

- load_path

The load_path is a variable which hold a string that contains the checkpoint's file path.

- checkpoint

The checkpoint holds the Keras's checkpoint function to save the agent's training result per epoch.

- model

The model holds the main neural network model that is initialized by `build_network()` function.

- target_model

The model holds the target neural network model that is initialized by `build_network()` function.

Four methods of the *DQ_Network class* are explained below:

- `__init__`

The `__init__` method overwrite the class's default initializer. It is utilized to initialize the class's properties.

- `build_network`

The `build_network` method is the core method of this class where the neural network is configured.

- `train`

The `train` method holds the main algorithm of the Deep Q Network with Experience Replay.

- `predict`

The `predict` method returns the prediction result of the neural network.

b. Agent Class

The *Agent Class* represent the agent with the neural networks. Agent stores the main algorithm parameters included the experiences memory. The mechanism of memory storing, action choosing, sampling experience, updating epsilon, and learning is defined in this function. There exist eleven properties that is created for the *Agent Class* which are:

- total_action

The total_action stands for a number of actions that the agent could take in the specific environment.

- learning_rate

The learning_rate is a variable which hold a floating-point value from 0 to 1 that defines the learning speed of the agent.

- input_dimension

The input_dimension is a variable which hold a three-dimensional array (row, column, channel) that defines the game's frame size. It will be utilized as an input for the neural network.

- discount_factor

The discount_factor is a variable which hold a floating-point value from 0 to 1 that defines the agent's behavior in considering long-term or short-term reward of the agent.

- epsilon

The epsilon stores a floating-point value between 0 and 1 to determine the agent characteristics to either explore or exploit the taken action.

- epsilon_decay

The epsilon_decay stores a floating-point value to control the epsilon's value in by decreasing the epsilon over the gameplay session.

- load_path

The load_path is a variable which hold a string that contains the checkpoint's file path.

- batch_size

The batch_size is a variable whichhold an integer number defining the batch size in the Keras's fit function.

- memory_size

The memory_size defines the memory size that the agent could hold.

- memory

The memory stored the gameplay experience in an array.

- training_count

The training _count is a counter to track the training progression.

Six methods in the *Agent Class* are explained below:

- `__init__`

The `__init__` method overwrite the class's default initializer. It is utilized to initialize the class's properties.

- `store_transition`

The `store_transition` method is utilized to save the taken action result in the agent's memory.

- `choose_action`

The `choose_action` method is used to determine whether the agent should explore or exploit the environment.

- `update_epsilon`

The `update_epsilon` method is used to handle the epsilon decay overtime mechanism.

- `sample_exp_batch`

The `sample_exp_batch` method randomly samples a number of stored experiences from the agent's memory that will be feeds into the neural network.

- `max_q`

The `max_q` method is utilized to get the maximum q value for testing purposes.

- `learn`

The `learn` method is the method where the network will be run and the Q-values will be updated.

CHAPTER V

SYSTEM IMPLEMENTATION

This chapter describes the process of implementing the algorithm of Deep Q Network with Experience Replay in Atari games using Python 3. The algorithm was implemented by the creation of *Network*, *Agent*, and *Main* class. The *Network* and *Agent* classes are the supporting classes that will build the Deep Q Network algorithm in the *Main* class. The *Network* class defines the neural network model that the *Agent* will be using in training. The *Agent* class was used to defines the memory, and decision-making mechanism. In the last class, the *Main* class, was used to define and run the main Deep Q Learning algorithm.

5.1 The Library

The following lines of code shown by figure 5.1 defines the library that are utilized in creating the agent. The Keras library components which are RMSProp, Sequential, Dense, Conv2D, Flatten, VarianceScalling, and Callbacks are used to create the neural network.

The random and randrange are built in python function which handles the random selection in the code. PIL, or Python Image Library, is a free library that adds supports for image processing in python. The OpenAi gym's library which handle the Atari games

environment. The Matplotlib handles data visualization to ease the evaluation of the agent's performance.

```
from keras.optimizers import RMSprop
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
from keras.initializers import VarianceScaling
from keras.callbacks import *
from random import random, randrange
from PIL import Image
import gym
import matplotlib.pyplot as plt
from gym.wrappers import Monitor
```

Figure 5.1 The Network Class' Required Library

5.2 The Network Class

The component to build the agent's neural network model is defined in the Network class. As mentioned before in the previous chapter, the agent utilizes a *Convolutional Neural Network* to recognize the feature inside the Atari game's frame [12]. The Network

class takes the *action* and *input* of the environment then feeds it into the CNN. The model will learn the input and try to predict the action by updating the q values list.

5.2.1 The Constructor

The Network class import the layers, optimizers, initializers, and models from Keras library for the need of building a Convolutional Neural Network model. The class's constructor, depicted by figure 5.2, will be explained in the next paragraph.

```
class DeepQNetwork(object):
    def __init__(self, total_action, learning_rate=0.00025,
                 input_dimension=(210, 160, 4),
                 batch_size=32, discount_factor=0.99,
                 load_path=None):

        # Parameters
        self.total_action = total_action
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.discount_factor = discount_factor
        self.input_dimension = input_dimension
        self.load_path = load_path
        self.checkpoint = ModelCheckpoint("checkpoint.h5",
monitor='val_acc', save_weights_only=True)
        self.callbacks_list = [self.checkpoint]

        # Init DQN
        self.model = self.build_network()
        self.target_model = self.build_network()

        if load_path is not None:
            self.model.load_weights(self.load_path)
```

Figure 5.2 The Network's Class Constructor

The class parameters are described as follow:

1. *total_action* is created without default value to ease the user in using the program to test another Atari games. The parameter stores the total action of the Atari games that will be inputted in the dense layer of the neural network.
2. *learning_rate* is a hyperparameter which determines the learning speed of the neural network. A higher learning rate will be resulted in a faster learning process with a low accurate model while a lower learning rate will be resulted in slow and accurate model. Learning rate will adjust the model trainable weight according to the error.
3. *input_dimension* defines the size of the game environment. The default size of Atari games environment is (210, 160, 4).
4. *batch_size* is a number of samples that will be used in the gradient update that is calculated by Keras' fit function.
5. *discount_factor* is a hyperparameter which defines the agent's behavior in exploration and exploitation. The details of this variable is thoroughly explained in the *Q-Learning* section in the Literature Study.
6. *load_path* is a variable that stores the checkpoint file path. The training session will be periodically stored to ensures the agent's learning continuity.

7. *checkpoint* is a variable which calls the checkpoint function from Keras library.
The variables contain the information of the checkpoint file's name and the property of the file. In this case, the checkpoint file only saves the model's trainable weights.
8. *callbacks_list* is a list that contains the checkpoint's session that is utilized to create checkpoint files for every training conducted.
9. The *model* and *q_target_model* are the variables that contain the current and the target *q_values*. These variables are initialized by the function *build_network* which means that these two variables contain the neural network's model.

5.2.2 The Build Network Function

The build network function is defined through the figure 5.3 and figure 5.4.

```
def build_network(self):  
    # CNN  
    model = Sequential()  
    model.add(Conv2D(32,  
                    kernel_size=8,  
                    strides=(4, 4),  
                    activation='relu',  
                    data_format = 'channels_first',  
                    input_shape=self.input_dimension,  
  
    kernel_initializer=VarianceScaling(scale=2.0)))  
    model.add(Conv2D(64,  
                    kernel_size=4,  
                    strides=(2, 2),  
                    activation='relu',  
                    data_format = 'channels_first',  
                    input_shape=self.input_dimension,
```

Figure 5.3 The 1st Section of Build Network Function

```

kernel_initializer=VarianceScaling(scale=2.0)))
    model.add(Conv2D(64,
                      kernel_size=3,
                      strides=(1, 1),
                      activation='relu',
                      data_format = 'channels_first',

input_shape=self.input_dimension,))

    model.add(Flatten())
    model.add(Dense(512,
                    activation='relu',

kernel_initializer=VarianceScaling(scale=2.0)))
    model.add(Dense(units=self.total_action,

kernel_initializer=VarianceScaling(scale=2.0)))

    if self.load_path is not None:
        model.load_weights(self.load_path)

    model.compile(RMSprop(self.learning_rate),
                  loss='mean_squared_error',
                  metrics=['accuracy'])
    return model

```

Figure 5.4 The 2nd Section of Build Network Function

The CNN are constructed by using three convolutional layers, one flatten layer, and a fully connected layers with 512 nodes. In the end, the convolutional layers are compiled using a *RMSProp* activation function and *mean_squared_error* loss function. Figure 5.5 below shows the summary of the neural network model architecture that is used in this application.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 20, 20)	8224
conv2d_2 (Conv2D)	(None, 64, 9, 9)	32832
conv2d_3 (Conv2D)	(None, 64, 7, 7)	36928
flatten_1 (Flatten)	(None, 3136)	0
dense_1 (Dense)	(None, 512)	1606144
dense_2 (Dense)	(None, 9)	4617
Total params: 1,688,745		
Trainable params: 1,688,745		
Non-trainable params: 0		

Figure 5.5 The Model's Architecture

5.2.3 The Train Function

The training function stores the core calculation of the Q Values. The agent learns to predict the Q Values based on the game stored history that were stored in the *batch* parameter. The calculation for the Q Values in Deep Q Network algorithm is presented below.

$$Q(s, a) = r(s, a) + \gamma \max_{\alpha} Q(s', a; \theta) \quad (5.1)$$

For each episode stored in the memory, the next state will be taken to be predicted by the network. Then, the maximum Q value of the next state is taken to fulfill the

$\max_{\alpha} Q(s', a; \theta)$ calculation. If the agent read the end state, the agent will only take the current reward, or $r(s, a)$, to the q list. The q list then will be fetched into the neural network to be trained by the network. The training code is indicated by the calling of the Keras's `model.fit` function where the training's checkpoint is handled by the *callbacks* parameter of fit function. figure 5.6 below shows the training function.

```
def train(self, batch, target):
    s_train, q_train = [], []

    for experience in batch:
        s_train.append(experience['state'].astype(np.float64))

        next_state = experience['next'].astype(np.float64)
        next_state_predict = target.predict(next_state).ravel()
        next_q = np.max(next_state_predict)

        q_list = list(self.predict(experience['state'])[0])
        if not experience['terminate']:
            q_list[experience['action']] = experience['reward'] +
self.discount_factor * next_q
        else:
            q_list[experience['action']] = experience['reward']
        q_train.append(q_list)

    s_train = np.asarray(s_train).squeeze()
    q_train = np.asarray(q_train).squeeze()

    self.model.fit(s_train, q_train, batch_size=self.batch_size,
epochs=1, verbose=0, callbacks=self.callbacks_list)
```

Figure 5.6 The Train Function

5.2.4 The Predict Functions

The predict function works by utilizing Keras's function to predict the result according to the neural network's input. Figure 5.7 below shows the predict function for both the current and the target q_value prediction.

```
def predict(self, state):  
    state = state.astype(np.float64)  
    return self.model.predict(state, batch_size=1)
```

Figure 5.7 The Predict Function

5.3 The Agent Class

The agent stores the network's model as the properties. Thus, the agent accepts the hyperparameters of the Q functions which will be used to predict the future rewards in accordance to the current state and action taken. The memory mechanism that will store and sample the history of the episode are defined in this class.

5.3.1 The Constructor

Learning rate, discount factor, epsilon, epsilon decay, total action, and input dimensions are the q function hyperparameters. These hyperparameters are initialized in the constructor of this class depicted by the figure 5.8 below.


```

# Hyper parameters
self.total_action = total_action
self.learning_rate = learning_rate
self.input_dimension = input_dimension
self.discount_factor = discount_factor
self.epsilon = epsilon
self.epsilon_decay = epsilon_decay
self.load_path = load_path

```

Figure 5.8 The Agent's Hyperparameters

The agent's memory is created using an array. The agent has a batch size and memory counter to support the random sampling of mini batches of memory as shown by figure 5.9.

```

# Initialize memory
self.batch_size = batch_size
self.memory_size = memory_size
self.memory = []
self.training_count = 0

```

Figure 5.9 The Agent's Memory Properties

The last properties that needs to be initialized was the agent's neural network. This will be done by instantiating the *Network* class into a variable called networks as shown by the figure 5.10 below while the full *Network* class code is shown through figure 5.11 below.

```

# Initialize network model
self.networks = DeepQNetwork(self.total_action, learning_rate=self.learning_rate,
                             input_dimension=self.input_dimension, size=self.batch_size,
                             discount_factor=self.discount_factor, load_path=self.load_path)
self.networks.target_model.set_weights(self.networks.model.get_weights())

```

Figure 5.10 The Agent's Network

```

class Agent(object):
    def __init__(self, total_action, learning_rate=0.00025, input_dimension=(210, 160, 4),
                 size=32, discount_factor=0.99, size=1024,
                 epsilon=1, epsilon_decay=0.99, load_path=None):
        # Hyper parameters
        self.total_action = total_action
        self.learning_rate = learning_rate
        self.input_dimension = input_dimension
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.load_path = load_path

        # Initialize memory
        self.batch_size = batch_size
        self.memory_size = memory_size
        self.memory = []
        self.memory_count = 0
        self.training_count = 0
        # Initialize network model
        self.networks = DeepQNetwork(self.total_action,
                                     learning_rate=self.learning_rate,
                                     input_dimension=self.input_dimension,
                                     batch_size=self.batch_size,
                                     discount_factor=self.discount_factor,
                                     load_path=self.load_path)
        self.networks.target_model.set_weights(self.networks.model.get_weights())

```

Figure 5.11 The Agent's Class

5.3.2 Storing Memory Function

An episode is a set of *state*, *action*, *reward*, *done variable*, and *next state*. As the memory already initialized in the constructor, the function will append the inputted episode into the memory and increase the memory counter. If the memory exceeds the limit, the least stored memory will be erased from the memory to give a space for the newest memory as shown by figure 5.12.

```
# Storing transition into memory
def store_transition(self, state, action, reward, next, terminate):
    exp = ({'state': state,
            'action': action,
            'reward': reward,
            'next': next,
            'terminate': terminate})
    if len(self.memory) >= self.memory_size:
        self.memory.pop(0)
    self.memory.append(exp)
```

Figure 5.12 The Store Transition Function

5.3.3 Choosing Action Function

A policy to choose an action is defined through the *choose_action* function. The epsilon value that determine the chosen action is dependent to the current agent's mode. If the agent is being evaluated, then the epsilon will have a set value of 0.05. However, if the agent is in the training mode, the epsilon value will follow the current epsilon value that the agent holds. The randomness of our agent decision making is handled through the random function provided by the numpy library. In the conditional code below in figure

5.13, it is shown that the agent will choose either to *explore* or *exploit* depending on the randomness value compared to the defined epsilon value. The breakdown of the full policy function is defined in the next paragraph.

```
# Policy
def choose_action(self, state, testing=False):
    epsilon = 0.05 if testing else self.epsilon
    if random.random() <= epsilon:
        action = np.random.choice(self.total_action)
    else:
        q_val = self.networks.predict(state)
        action = np.argmax(q_val)
    return action
```

Figure 5.13 The Policy Function

An exploration is indicated by the agent taking a random action from the defined action space whenever the random value is less or equal then the epsilon value. The act of the agent exploring the action space is shown through the figure 5.14.

```
if random.random() <= epsilon:
    action = np.random.choice(self.total_action)
```

Figure 5.14 The Exploration Code

On the other hand, an exploitation is defined by a q value network to update the list of q values, then the agent will choose the highest q values which indicated an action that will lead into most favorable reward as shown by figure 5.15.

```

else:
    q_val = self.networks.predict(state)
    action = np.argmax(q_val)
    return action

```

Figure 5.15 The Exploitation Code

5.3.4 Update Epsilon Function

The code shown by figure 5.16 shows the code to updating epsilon value. The epsilon then will be utilized as the real indicator for the agent's behavior that is shown by the figure 5.14.

```

def update_epsilon(self):
    if self.epsilon - self.epsilon_decay > 0.1:
        self.epsilon -= self.epsilon_decay
    else:
        self.epsilon = 0.1

```

Figure 5.16 The Update Epsilon Function

5.3.5 Sampling Function

Randrange built in function is utilized to randomly choice an action in the range of memory length as shown by the figure 5.17.

```

self.memory[randrange(0, len(self.memory))]

```

Figure 5.17 Random Mechanism to Choose a Batch of Episodes from Memory

After a batch of episode have been fetch from the memory, the episode content will be stored into the new container of state, action, done, and next state. The array container

is called *batch*. Then, it will be returned to the new container as shown below in the figure 5.18.

```
# Sampling
def sample_exp_batch(self):
    batch = []
    for i in range(self.batch_size):
        batch.append(self.memory[randrange(0, len(self.memory))])
    return np.asarray(batch)
```

Figure 5.18 The Sampling Function

5.3.6 Max Q Function

The max q function, depicted by figure 5.19, is created to fetch the maximum q values. It is used for the agent evaluation's purpose.

```
def max_q(self, state):
    q_values = self.networks.model.predict(state)
    idxs = np.argwhere(q_values == np.max(q_values)).ravel()
    return np.random.choice(idxs)
```

Figure 5.19 The Max Q Function

5.3.7 The Learning Function

The learning function, depicted by figure 5.20, called the main calculation in the network class. For each training session, the training's counter will keep track the history of the number of training session conducted by the system. The sampling function is called to get the sample for the memory to be fetched in the training function.

```
# Learning with Experience Replay
def learn(self):
    self.training_count += 1
    batch = self.sample_exp_batch()
    self.networks.train(batch, self.networks.target_model)
```

Figure 5.20 The Learning Function

5.4 The Main Class

The *Main* class is utilized to preprocess Atari games frame and to lay out the DQN algorithm implementation.

5.4.1 The Constant

Seven constants which are the hyper parameters of the agent is defined in the beginning of the code as shown by figure 5.21 below.

```
ALPHA = 0.00025
EPSILON = 1
EPSILON_DECAY = 9e-7
GAMMA = 0.99
INPUT_DIMS = (4, 84, 84)
BATCH_SIZE = 32
MEMORY_SIZE = 1024
MAX_EPISODE = 50000
ENV_ID = 'MsPacmanDeterministic-v4'

IMG_SIZE = (84, 84)
```

Figure 5.21 The Constants

5.4.2 Data Preprocessing

The Atari games frames is a high dimensional data which have a lot of information stored in the form of pixels. To minimize the computation, the Atari game' frame is converted from RGB color scheme into grayscale color scheme. The dimensionality is also reduced from 210x164 into 84x84.

Furthermore, to create a sense of movement/velocity, the game' frame will be stacked into four consecutive frames [2]. The stacking procedure was achieved by creating an array using numpy library. The past three frame that already occurred in the game was put into the first three address in the array while the current frame will be stacked into the fourth address. The code of preprocessing the frame of the games and stacking four consecutive frames is shown respectively by the preprocess and stack_frames functions shown by figure 5.22 below.

```
def preprocess(observation):
    global IMG_SIZE
    image = Image.fromarray(observation,
    'RGB').convert('L').resize(IMG_SIZE)
    return np.asarray(image.getdata(),
    dtype=np.uint8).reshape(image.size[1], image.size[0])

def shift(current_stack, observation):
    return np.append(current_stack[1:], [observation], axis=0)
```

Figure 5.22 The Preprocess and Shift Function

In the next function discussion, the shift and preprocess functions mechanism will be illustrated to illustrate with the real example of the initial state and the shifting into the next state.

5.4.3 The Plotting Function

The plot function is created to give an easy visualization of the agent's performance throughout the training session. The function plotted two different graph which are scores history and average score history. The graph created will consist of two different information that will be depicted by a line and a scattered dot. The line will illustrate the reduction of epsilon value while the scattered dot represents either an episode score or an average episode score. The differences between scores history and average scores history is shown in the scattered dot graph. One of the final results of this function, the average scores graph, is shown through the figure 5.23 below.

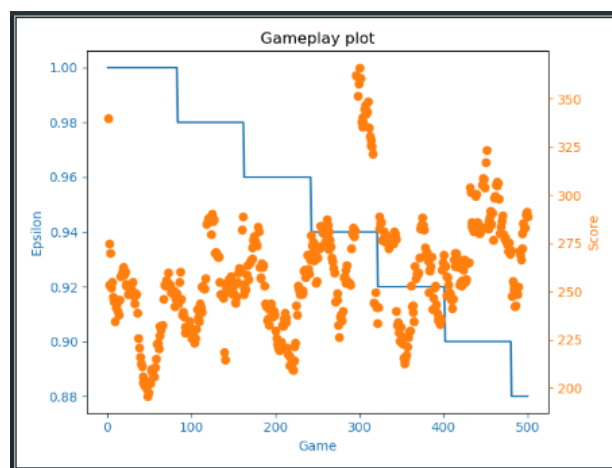


Figure 5.23 The Average Scores Graph

To create the graph, a figure must first be initialized in the beginning of the function hold by a variable called *fig*. Then, two axes represented by variables *ax* and *ax2* is created to hold two different kind of graph in one figure. The beginning of the *plot* function is shown through figure 5.24 below.

```
def plot(x, y, z=None, mode=None, filename=None):
    # The Figure's Frame
    fig = plt.figure()
    ax = fig.add_subplot(111, label="1")
    ax2 = fig.add_subplot(111, label="2", frame_on=False)
```

Figure 5.24 The 1st Section of The Plot Function

The x-axis will be used to track the number of episodes that has ended throughout the training session. The y-axis will be used to represents the score, to balance the axis's elements, the minimum and maximum value will be extracted from the y parameters and putted into *max_score*, *min_score*, *max_index*, and *min_index*. The z-axis, which actually means the second y-axis, is used to store the epsilon value. The code depicted by figure 5.25 below show the extraction of the maximum and minimum value of the y axis.

```
...# The Maximum and Minimum Value for The y Axis
max_score = np.max(y)
min_score = np.min(y)
max_index = np.where(y == max_score)
min_index = np.where(y == min_score)
```

Figure 5.25 The 2nd Section of The Plot Function

Plotting two kind of graph is handled in a similar manner, both scores history and averages graph's x-axis and y-axis are filled with the number of episodes, stored in the x variable, and the epsilons value, stored in the z variable. A built in matplotlib function called *plot* is called to create a basic line graph. The creation of the graph that visualize the reduction of epsilon value is handled through the code depicted by figure 5.26 below.

```
...# The First Plot's Axes – Epsilon's Reduction
ax.plot(x, z, color="C0")
ax.set_xlabel("Game", color="C0")
ax.set_ylabel("Epsilon", color="C0")
ax.tick_params(axis='x', colors="C0")
ax.tick_params(axis='y', colors="C0")
```

Figure 5.26 The 3rd Section of The Plot Function

The scores history is created by mapping the y variable into the scattered dot graph. Creating a second graph in a single graph figure require the utilization of the *ax2* variable that will call *scatter* method from matplotlib library. Another matplotlib function, *text*, is utilized to gives info regarding the maximum and minimum occurring scores. The code to handle the scores history scatter graph is shown by figure 5.27 below.

```
...# The Scores History Scatter Graph
if mode == 0:
    max_score_str = 'Max Score=' + str(max_score)
    min_score_str = 'Min Score=' + str(min_score)
    plt.text(0, max_score, max_score_str)
    plt.text(0, max_score - 300, min_score_str)
    ax2.scatter(x, y, color="C1")
```

Figure 5.27 The 4th Section of The Plot Function

The average scores history is handled similar with the scores graph. The mechanism to obtain the averages score is handled by utilizing numpy built in *mean* function to directly get the score's average from the y variables. The code below depicted by figure 5.28 shown the process to create averages score history scatter graph.

```
elif mode == 1:
    # The Average Scores History Scatter Graph
    N = len(y)
    running_avg = np.empty(N)
    for t in range(N):
        running_avg[t] = np.mean(y[max(0, t - 20):(t + 1)])

    max_avg_score = np.max(running_avg)
    min_avg_score = np.min(running_avg)
    print('avg:', max_avg_score, ' ', min_avg_score)

    max_avg_index = np.where(y == max_avg_score)
    min_avg_index = np.where(y == min_avg_score)
    print('avg:', max_avg_index, ' ', min_avg_index)

    max_avg_score_str = 'Max Score=' + str(max_avg_score)
    min_avg_score_str = 'Min Score=' + str(min_avg_score)
    plt.text(0, max_avg_score, max_avg_score_str)
    plt.text(0, max_avg_score - 80, min_avg_score_str)
    ax2.scatter(x, running_avg, color="C1")
```

Figure 5.28 The 5th Section of The Plot Function

Setting up the scattered graph y-axis is handled outside of the if block. The scatter graph's x-axis is hidden to prevent overlay. The y-axis is set to the right position of the figure to also prevent overlay with the epsilon value that is located at the left-hand side.

The color code for the scattered graph then is set into a different one with the line graph.

The setting of the scattered graph is shown through the figure 5.29 below.

```
...ax2.axes.get_xaxis().set_visible(False)
...ax2.yaxis.tick_right()
...ax2.set_ylabel('Score', color="C1")
...ax2.yaxis.set_label_position('right')
...ax2.tick_params(axis='y', colors="C1")
```

Figure 5.29 The 6th Section of The Plot Function

The figure will not only be shown to the user but it will be saved to the local storage as well. The last section of the plotting function shown by figure 5.30 is to set the figure's title and to save the figure to the local storage *systdef* em only if a filename is provided. The figure is saved first before it is shown to the user so that the *plt* variable which contains the figure will not be emptied by the *show* function.

```
...
...ax.set_title(filename)
...if filename is not None:
...    plt.savefig(filename)
...plt.show()
```

Figure 5.30 The 7th Section of The Plot Function

As discussed above, the plotting function created is set to produce a figure which has two different graphs. The full code of the plotting function is shown through the figure 5.31 and 5.32 below.

```

plot(x, y, z=None, mode=None, filename=None):
    fig = plt.figure()
    ax = fig.add_subplot(111, label="1")
    ax2 = fig.add_subplot(111, label="2", frame_on=False)

    max_score = np.max(y)
    min_score = np.min(y)
    max_index = np.where(y == max_score)
    min_index = np.where(y == min_score)

    # ax / x = eps, yl = epsilon, bl
    # ax / x = eps, yr = scores, yo
    ax.plot(x, z, color="C0")
    ax.set_xlabel("Game", color="C0")
    ax.set_ylabel("Epsilon", color="C0")
    ax.tick_params(axis='x', colors="C0")
    ax.tick_params(axis='y', colors="C0")

    if mode == 0:
        # The Scores History Scatter Graph
        print('train:', max_score, ' ', min_score)
        print('train:', max_index, ' ', min_index)

        max_score_str = 'Max Score=' + str(max_score)
        min_score_str = 'Min Score=' + str(min_score)
        plt.text(0, max_score, max_score_str)
        plt.text(0, max_score - 300, min_score_str)
        ax2.scatter(x, y, color="C1")
    elif mode == 1:
        # The Average Scores History Scatter Graph
        N = len(y)
        running_avg = np.empty(N)
        for t in range(N):
            running_avg[t] = np.mean(y[max(0, t - 20):(t + 1)])

```

Figure 5.31 The Plotting Function (Part 1)

```

max_avg_score = np.max(running_avg)
min_avg_score = np.min(running_avg)
print('avg:', max_avg_score, ' ', min_avg_score)

max_avg_index = np.where(y == max_avg_score)
min_avg_index = np.where(y == min_avg_score)
print('avg:', max_avg_index, ' ', min_avg_index)

max_avg_score_str = 'Max Score=' +
str(max_avg_score)
min_avg_score_str = 'Min Score=' +
str(min_avg_score)
plt.text(0, max_avg_score, max_avg_score_str)
plt.text(0, max_avg_score - 80, min_avg_score_str)
ax2.scatter(x, running_avg, color="C1")

ax2.axes.get_xaxis().set_visible(False)
ax2.yaxis.tick_right()
ax2.set_ylabel('Score', color="C1")
ax2.yaxis.set_label_position('right')
ax2.tick_params(axis='y', colors="C1")

ax.set_title(filename)
if filename is not None:
    plt.savefig(filename)
plt.show()

```

Figure 5.32 The Plotting Function (Part 2)

5.4.4 The Evaluation Function

The evaluation function is created to evaluate the agent's performance throughout the training session. The function purpose is to calculate the maximum mean score that could be obtained by the agent. The first section of this function is to initialize the needed

variable for evaluation purpose that will be thoroughly discussed in the following paragraph. Below presented by the figure 5.33 is the first section of the evaluation function, the initialization phase.

```

max_mean_score = 0
def evaluate(DQA):
    global max_mean_score
    env = gym.make(ENV_ID)
    scores = list()
    episode = 0

    while episode < 10000:
        obs = preprocess(env.reset())
        # Initialize the first state with the same 4 images
        current_state = np.array([obs, obs, obs, obs])
        t = 0
        score = 0
        live = 3
        done = False

```

Figure 5.33 The 1st Section of Evaluate Function

The maximum mean score then will be stored in the global variable called `max_mean_score`. In the beginning of the function, the environment (`env`), score list (`scores`), and an episode counter (`episode`) will be initialized.

Then, the evaluation will start by calling out a *while loop* to handle episode looping until 10000 iteration. Inside the loop, the observation will be initialized with Ms. PacMan environment by using gym's make function. The observation is stored into `obs` variable. The current state (`current_state`) is represented with a same consecutive four preprocessed

images of the initial Ms. PacMan state. The four preprocessed images that are stored in this variable is shown in figure 5.34.

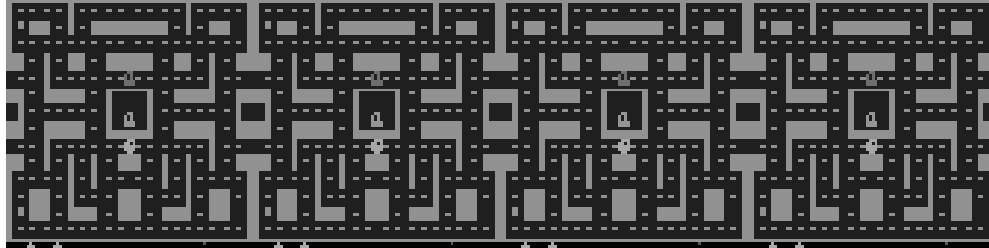


Figure 5.34 The Preprocessed and Stacked initial game frames

The preprocessed image means that the actual RGB color scheme images was converted and resize into a smaller size with grayscale color scheme. The preprocess function takes the observation image, crop the unnecessary black part around the score display, and converted it into an array image with converted color scheme. The code which handles this process is shown in figure 5.35.

```
def preprocess(observation):
    global IMG_SIZE
    image = observation[1:176:2, ::2]
    image = Image.fromarray(image,
        'RGB').convert('L').resize(IMG_SIZE)
```

Figure 5.35 The 1st Section of Preprocess Function

The result of the image color scheme conversion and resize is shown through the figure 5.36.

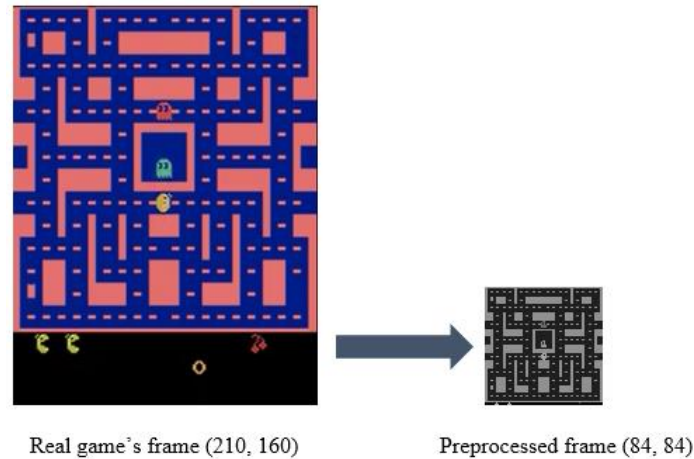


Figure 5.36 The Transformation of Real Game Frame into Preprocessed Frame

The result of the preprocessed frame then will be returned as an array of integer by using numpy library. The code to return the data is shown in the figure 5.37.

```
return np.asarray(image.getdata(),
dtype=np.uint8).reshape(image.size[1], image.size[0])
```

Figure 5.37 The 2nd Section of Preprocess Function

After the current state have been initialized, the other variable which are timestep (t), current score (Score), live counter (live), and the end of episode Boolean (done) are initialized.

The second section of the evaluation function is started when the agent start to play Ms.PacMan. This phase is marks by the second appearance of *while loop* function which is based on the end of episode boolean (done) and the live counter variable (live) which means that an episode will end if the agent reach the terminal state or if the agent run out

of Ms.PacMan's live. The figure 5.38 shows the code of the second section of the evaluation function which contains the second loop.

```
# Start episode
while not done or live > 0:
    action = DQA.choose_action(np.asarray([current_state]),
testing=True)
    obs, reward, done, info = env.step(action)
    live = info['ale.lives']
    obs = preprocess(obs)
    current_state = shift(current_state, obs)
    score += reward
    t += 1
```

Figure 5.38 The 2nd Section of Evaluate Function

The agent calls the choose action function to choose a random action, which is set into testing mode, based on the choose action mechanism that is already discussed in the agent's class section.

The obs, reward, done, and info variable store the result of the agent's taken action. Live variable stored the information of the agent's live count. After taking an action, the next observation will be preprocessed again, then the current_state stack of consecutive four images will be updated. The update is handled by the *shift* function, shown by figure 5.39, that will create a sense of motion so that the neural network could extract the feature of the moving game's frame.

```
def shift(current_stack, observation):
    return np.append(current_stack[1:], [observation], axis=0)
```

Figure 5.39 The Shift Function

Shift function will shift the four images which means the last stored images will be deleted leaving an empty space for the newest state to be stored. The process of shifting the states is illustrated through the figure 5.40 below.

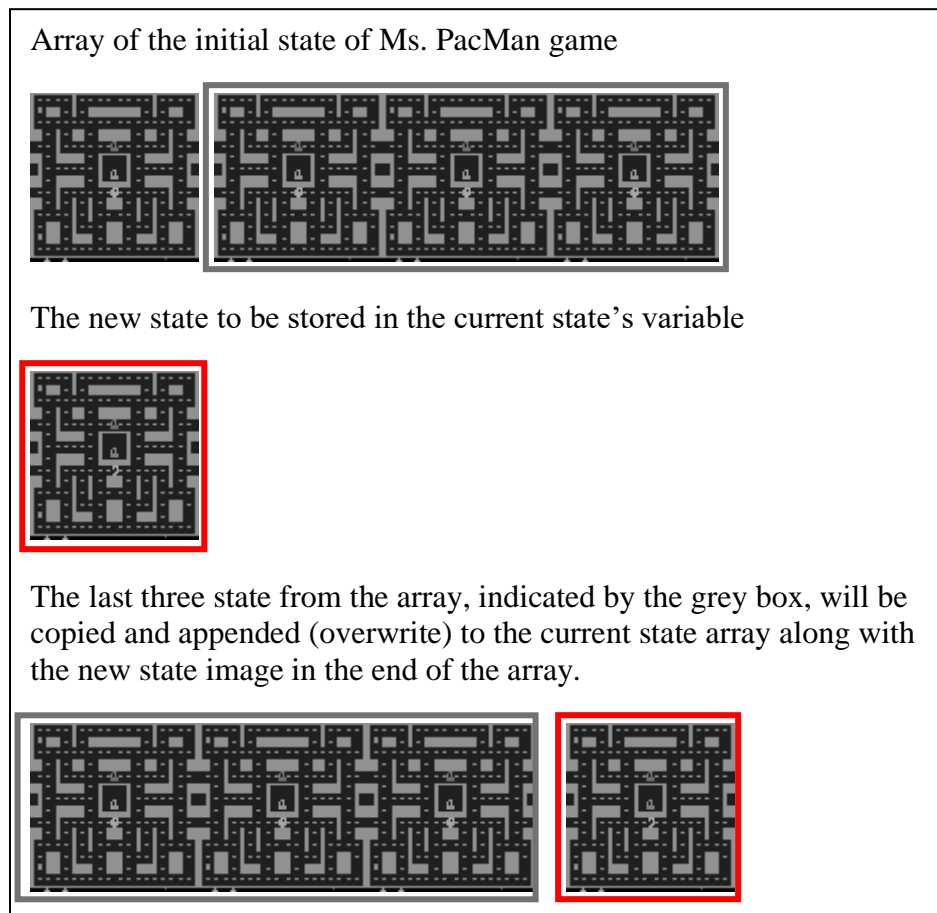


Figure 5.40 The Shifting Frame Mechanics

The score is updated with the immediate reward that the agent gets from taking an action. Then, the algorithm will check whether the episodes reach its end by checking the done variable and the live counter. If the episode reaches it ends, then the score will be appended to the scores list and the episode counter will be incremented as shown by figure 5.41.

```
# End episode
if done or live < 1:
    episode += 1
    scores.append([t, score])
scores = np.asarray(scores)
max_indices = np.argwhere(scores[:, 1] == np.max(scores[:,
1])).ravel()
max_idx = np.random.choice(max_indices)
```

Figure 5.41 The 3rd Section of Evaluate Function

The final phase, depicted by figure 5.42, of the evaluation function is located outside the loops. The maximum score location in the array then will be taken. Then, the best average score that the agent could get is extracted. A checkpoint that contains the weights is also created. The max score then will be returned which mark the end of the function. The full code of the evaluation function is depicted by figure 5.43 and 5.44 below.

```
# Save best model
if max_mean_score < np.mean(scores):
    max_mean_score = np.mean(scores)
    agent.networks.model.save_weights(str(scores[len(scores)-1]) + '_best.h5')
return scores[max_idx, :].ravel()
```

Figure 5.42 The 4th Section of Evaluate Function

```

def evaluate(DQA):
    global max_mean_score
    env = gym.make(ENV_ID)
    # env = Monitor(env, './videos', force=True, video_callable=lambda episode:
    True)
    scores = list()
    frame_counter = 0
    episode = 0

    while frame_counter < 140000:
        obs = preprocess(env.reset())

        frame_counter += 1
        # Initialize the first state with the same 4 images
        current_state = np.array([obs, obs, obs, obs])
        t = 0
        score = 0
        live = 3
        done = False

        # Start episode
        while not done or live > 0:
            action = DQA.choose_action(np.asarray([current_state])), testing=True)
            obs, reward, done, info = env.step(action)
            live = info['ale.lives']
            obs = preprocess(obs)
            current_state = shift(current_state, obs)
            score += reward
            t += 1
            frame_counter += 1

```

Figure 5.43 The Evaluate Function (Part 1)

```

        # End episode
        if done or live < 1:
            episode += 1
            # print('Evaluate Episode %d end\n-----\nFrame counter:
            %d\n' %
                # (episode, frame_counter))
            # print('Length: %d\n, Score: %f\n\n' % (t, score))
            scores.append([t, score])

    # clear_output()
    scores = np.asarray(scores)
    max_indices = np.argwhere(scores[:, 1] == np.max(scores[:, 1])).ravel()
    max_idx = np.random.choice(max_indices)

    # Save best model
    if max_mean_score < np.mean(scores):
        max_mean_score = np.mean(scores)
        agent.networks.model.save(str(scores[len(scores)-1]) + '_best.h5')

    return scores[max_idx, :].ravel()

```

Figure 5.44 The Evaluate Function (Part 2)

5.4.5 The Main Function

The main function implements the Deep Q Network Algorithm. The first step to implement the main algorithm is to setup the environment and initialize all the needed variables. The environment (env) variable hold the environment by using make function from gym. A monitor then will be created to create the video file of an episode by using a Monitor function which also a built-in function from gym library.

The agent class then initialized into the agent variable by calling the agent's class constructor that is filled with the defined constant. Score (score_array), mean score (mean_array), state (state_array), and q (q_array) arrays then initialized alongside the episode (episode_counter) and frame (frame_counter) counter. The full initialization code is shown through the figure 5.45 below.

```
# Setup
# Environment
env = gym.make(ENV_ID)
env = Monitor(env, './videos', force=True, video_callable=lambda episode:
True)
agent = Agent(env.action_space.n, learning_rate=ALPHA,
              input_dimension=INPUT_DIMS, batch_size=BATCH_SIZE,
              discount_factor=GAMMA, memory_size=MEMORY_SIZE,
              epsilon=EPSILON, epsilon_decay=EPSILON_DECAY,
              load_path="[811. 440.]._best.h5")

print('Action space: ', env.action_space.n, '\n',
      'Observation space: ', env.observation_space.shape, '\n')
# Containers
train_result = []
score_array = []
epsilons = []

# Evaluation
t_eval = []
score_eval = []
eps_eval = []

# Counters
episode_counter = 0
frame_counter = 0
```

Figure 5.45 The Main Setup

The next section marks the start of an episode by creating a stacked of preprocessed four initial episode frames. Various loop variables are also being initialized in this first while loop such as score and reward variable to track the cumulative score and immediate reward while an episode is running, the live counter (live) to indicates the Ms. PacMan live which is three lives, and the timestep and frame counter. The info variable is an array that store the agent's live information after the agent has taken an action by using step function from gym library. The code below depicted in figure 5.46 shows the first while episode loop.

```
# Main
while episode_counter < MAX_EPISODE:
    score = 0
    observation = preprocess(env.reset())
    stacked_frames = np.array([observation, observation, observation,
    observation])

    timestep = 0
    done = False
    frame_counter += 1
    live = 3
    reward = 0
    info = []
```

Figure 5.46 The Main Episode Loop

The second loop will run as long as the agent did not exhaust their lives and do not reach the terminal state. The first part of this loop has a similar code with the evaluation

agent second loop which is where the agent will take an action that will update the observation, reward, done, and info variable. The preprocessed image is saved by utilizing the PIL library function for documentation purpose. The full code is shown by figure 5.47.

```

while not done or live > 0:
    # Choose an action
    action = agent.choose_action(np.asarray([stacked_frames]))
    # env.render()

    # Get next observation area and shift the stacked frames
    observation, reward, done, info = env.step(action)
    observation = preprocess(observation)

    im = Image.fromarray(np.uint8(observation))
    if frame_counter < 1000:
        file_name = 'preprocess_' + str(frame_counter) + '.png'
        im.save(file_name)
    next_state = shift(stacked_frames, observation)
    live = info['ale.lives']

    frame_counter += 1

```

Figure 5.47 The Agent's Decision-Making Mechanics

Various Atari games have different reward, to handle this inconsistency, the reward is clipped so to only 0 and 1. The experience is stored into the agent's memory by calling the *store_transition* function. If the memory has reach a certain length, where in this case is 200, the agent will start to learn, or start the neural network model training, for every four timesteps. The Q target weights then will be updated for every 500-training step and

the epsilon will be decayed overtime. Figure 5.48 below shows the code that handle the reward clipping and experience replay main calculation.

```
# Store experience
clip = np.clip(reward, -1, 1)
agent.store_transition(np.asarray([stacked_frames]), action,
                      clip, np.asarray([next_state]), done)

# Train
if len(agent.memory) >= 1000:
    if timestep % 4 == 0:
        agent.learn()

if agent.training_count % 10000 == 0 and agent.training_count >= 10000:
    agent.networks.target_model.set_weights(agent.networks.model.get_weights())
    agent.update_epsilon()
```

Figure 5.48 The Experience Replay and Training Mechanics

The evaluation will start to take place for every 10000 game frames at the end of the episode's loop. The evaluation is done by running the agent throughout the game to obtain maximum average score. The training and score evaluation variables will hold the evaluations score. The second loop end with the update of stacked frame, score, and timestep variables. Figure 5.49 below shows the evaluation and testing mechanics codes.

The last part of the first while loop, depicted by figure 5.50, is ended by plotting function initialization and episode's counter increment. The maximum average score obtain by the evaluation function is also printed at the end of the code. The example of the output log produced by the program is shown by figure 5.51.

```

if frame_counter % 10000 == 0:
    t, s = evaluate(agent)
    t_eval.append(t)
    score_eval.append(s)
    eps_eval.append(episode_counter)

stacked_frames = next_state
score += reward
timestep += 1

```

Figure 5.49 The Agent's Evaluation Mechanics

```

score_array.append(score)
epsilons.append(agent.epsilon)
real_eps = episode_counter + 1
episodes = [i + 1 for i in range(real_eps)]

if real_eps >= 100 and real_eps % 100 == 0:
    try:
        plot(np.asarray(episodes), np.asarray(score_array), epsilons, 0, "train plot_" +
str(episode_counter))
    except ValueError as e:
        print(e)
    except Exception as e:
        print(e)
    try:
        plot(np.asarray(episodes), np.asarray(score_array), epsilons, 1, "average plot_"
+ str(episode_counter))
    except ValueError as e:
        print(e)
    except Exception as e:
        print(e)
    episode_counter += 1
print('Max_mean_score:', max_mean_score)

```

Figure 5.50 The Plotting Mechanics

Score: 240.0	Score: 230.0
Eps: 366 Training # 45806	Eps: 370 Training # 46320
Score: 180.0	Score: 530.0
Eps: 367 Training # 45930	Eps: 371 Training # 46477
Score: 310.0	Score: 370.0
Eps: 368 Training # 46067	Eps: 372 Training # 46624
Score: 340.0	Score: 260.0
Eps: 369 Training # 46205	Eps: 373 Training # 46750

Figure 5.51 The Output Log

CHAPTER VI

SYSTEM TESTING

System Testing refer to a detailed procedure of comprehensive quality assurance process to ensure the whole application runs according to its function. The objective of this process is to find defect/bugs/error of the created application and ensure that the system satisfies its requirements.

6.1 Testing Environment

The testing environment was conducted locally using JetBrains PyCharm. Below the specification required to run the program.

1. 64-bit Microsoft Windows 10 Operating System
2. 12 GB of RAM for the program
3. Python 3.7.4
4. JetBrains PyCharm Professional 2019.2.1

6.2 Testing Scenario

The testing was conducted by the conducted by the observing the output of the neural network per game episode. The goal of the agent is to master and exploit Ms. PacMan to get the highest possible score through any means. Therefore, the agent behavior will also be tracked in the evaluation. The following evaluation shows the agent's behavior and improvement in different learning rate and epoch.

6.2.1 Training Time and Scenarios

In the default setting for the algorithm to work, the training should be set into 1 Epoch, 0.0025 learning rate. After testing the However, the author also experimented with two different scenarios in training the agent and compares the results.

The first scenario, **Case A**, is to set 1 Epoch per training scenario with 0.3 learning rate. The time used to train 100 Episodes with 1 Epoch and 0.3 learning rate is 6 hours.

The second scenario, **Case B**, is to set 100 Epoch per training scenario with 0.025 learning rate. The time used to train 100 episodes is 1 Day and 2 Hours, or 26 Hours in total.

Table 6.1 Testing Scenario

	Epoch	Learning Rate
Case A	1	0.3
Case B	100	0.025
Default	1	0.025

6.2.2 Initial Result

The initial result when the first episode run shows that the agent still trying to explore the surrounding area. The agent movement is not smooth which means that the agent did not take 1 action per direction it's heading into. Instead, the agent trying different

random action that cause it to stuttering around and end up dying while it gathers 90 score.

The agent performance at initial episode is shown through figures 6.1 and 6.2.



Figure 6.1 Testing Agent's Initial Behaviour

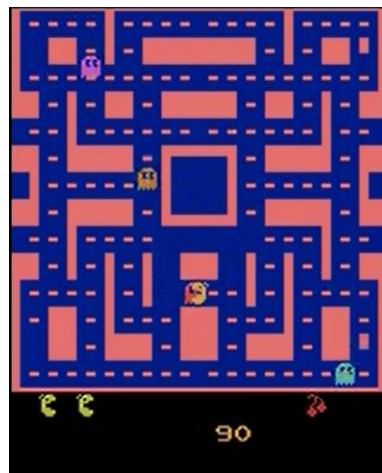


Figure 6.2 Testing Agent's Collide with a Ghost

At the end of its live, the agent manages to gathers 220 points, shown by figure 6.3, without any utilization of **power up** bullet which can power up Ms. PacMan to eat the ghost without dying.

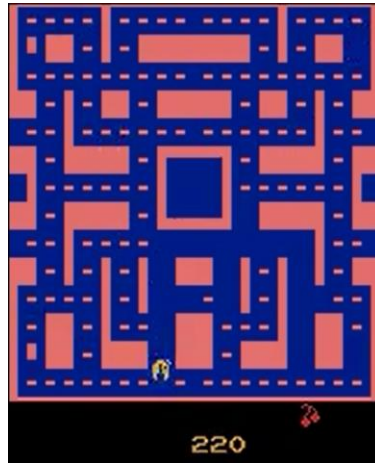


Figure 6.3 The Agent Out of Lives

6.2.3 Case A

During its 100th gameplay, the agent has undergone 12595 training session. The agent got 180 score when it reached the 100th episode. The scores from episode 80 to episode 110 mostly dominated around the range of 200 – 250, The highest score that the agent is able to reach occurred at Episode 97 with the score 590. In this case, the agent still has the tendency to act like the initial test. The agent still does not recognize the enemy, the power up food, and the edible enemy in this scenario. The training result is shown through figure 6.4 below.

Score: 270.0	Score: 240.0	Score: 180.0
Eps: 80 Training # 10030	Eps: 90 Training # 11281	Eps: 100 Training # 12595
Score: 170.0	Score: 220.0	Score: 230.0
Eps: 81 Training # 10126	Eps: 91 Training # 11420	Eps: 101 Training # 12700
Score: 220.0	Score: 240.0	Score: 310.0
Eps: 82 Training # 10238	Eps: 92 Training # 11558	Eps: 102 Training # 12828
Score: 240.0	Score: 210.0	Score: 270.0
Eps: 83 Training # 10363	Eps: 93 Training # 11685	Eps: 103 Training # 12958
Score: 260.0	Score: 200.0	Score: 270.0
Eps: 84 Training # 10537	Eps: 94 Training # 11819	Eps: 104 Training # 13085
Score: 270.0	Score: 140.0	Score: 240.0
Eps: 85 Training # 10664	Eps: 95 Training # 11935	Eps: 105 Training # 13192
Score: 270.0	Score: 290.0	Score: 210.0
Eps: 86 Training # 10785	Eps: 96 Training # 12055	Eps: 106 Training # 13305
Score: 190.0	Score: 590.0	Score: 240.0
Eps: 87 Training # 10901	Eps: 97 Training # 12216	Eps: 107 Training # 13425
Score: 240.0	Score: 240.0	Score: 200.0
Eps: 88 Training # 11022	Eps: 98 Training # 12346	Eps: 108 Training # 13553
Score: 230.0	Score: 250.0	Score: 180.0
Eps: 89 Training # 11151	Eps: 99 Training # 12480	Eps: 109 Training # 13670
		Score: 240.0
		Eps: 110 Training # 13786

Figure 6.4 The Training Result from Episode 80 – 110 in Case A

6.2.4 Case B

During its 100th gameplay, the agent has undergone $13023 * 10^2$ training session. In this scenario, the agent behavior has resulted in a more consistent exploration, detected from a lot of the road is emptied from the food that is previously lying around. In this scenario, there exist three high score achieved by the agent which are found at episode 85, 100, and 104 with a value 940, 1040, and 1340. The training result is shown through figure 6.5 below.

Score: 240.0	Score: 230.0	Score: 1040.0
Eps: 80 Training # 9993	Eps: 90 Training # 11200	Eps: 100 Training # 12472
Score: 160.0	Score: 240.0	Score: 260.0
Eps: 81 Training # 10102	Eps: 91 Training # 11319	Eps: 101 Training # 12596
Score: 170.0	Score: 250.0	Score: 170.0
Eps: 82 Training # 10210	Eps: 92 Training # 11420	Eps: 102 Training # 12712
Score: 160.0	Score: 220.0	Score: 190.0
Eps: 83 Training # 10299	Eps: 93 Training # 11538	Eps: 103 Training # 12819
Score: 210.0	Score: 250.0	Score: 1340.0
Eps: 84 Training # 10421	Eps: 94 Training # 11658	Eps: 104 Training # 13023
Score: 940.0	Score: 280.0	Score: 160.0
Eps: 85 Training # 10550	Eps: 95 Training # 11785	Eps: 105 Training # 13145
Score: 560.0	Score: 240.0	Score: 190.0
Eps: 86 Training # 10699	Eps: 96 Training # 11909	Eps: 106 Training # 13272
Score: 200.0	Score: 310.0	Score: 180.0
Eps: 87 Training # 10822	Eps: 97 Training # 12043	Eps: 107 Training # 13402
Score: 210.0	Score: 270.0	Score: 310.0
Eps: 88 Training # 10934	Eps: 98 Training # 12165	Eps: 108 Training # 13547
Score: 200.0	Score: 220.0	Score: 150.0
Eps: 89 Training # 11099	Eps: 99 Training # 12296	Eps: 109 Training # 13647

Figure 6.5 The Training Result from Episode 80 – 110 in Case B

6.2.5 *Default Case*

The training is conducted in three different phases because the author encountered technical error during training session which interrupted the training session. The first phase has 547 episodes which is running for 29 hour and 27 minutes. The second phase has 601 episodes which is running for 9 hour and 43 minutes. The third phase has 1001 episodes which is running for 20 hour and 8 minutes. Hence, in total, the default scenario was going through 2149 episodes in 59 hour and 19 minutes.

The default scenario is set to have 1 epoch and 0.025 learning rate. In this scenario, the agent is able to reach 100 episode/hour. However, as the training goes on, the training

speed declined into around 35-40 episodes/hour starting from the 700th – 1001th episode of the second phase. The training speed declined as the agent consumed a lot of memory and storage throughout the training session. At the start of a training session, the agent only needs less than 2 GB of RAM where at the halfway of the training, around the 600th episode, the agent consumes around 6.5 – 8.7 GB of RAM.

At the first phase, the agent was exploring any possible actions it could take which is shown by the low score achieved (ranged around 200-400 points) and stuttering behaviour. Thus, the number of high scores achieved at this point cannot be said as a result of an intelligent decision. The average scores graph, depicted by figure 6.6, shows the trends of improvement of the agent's performance through 547 episodes.

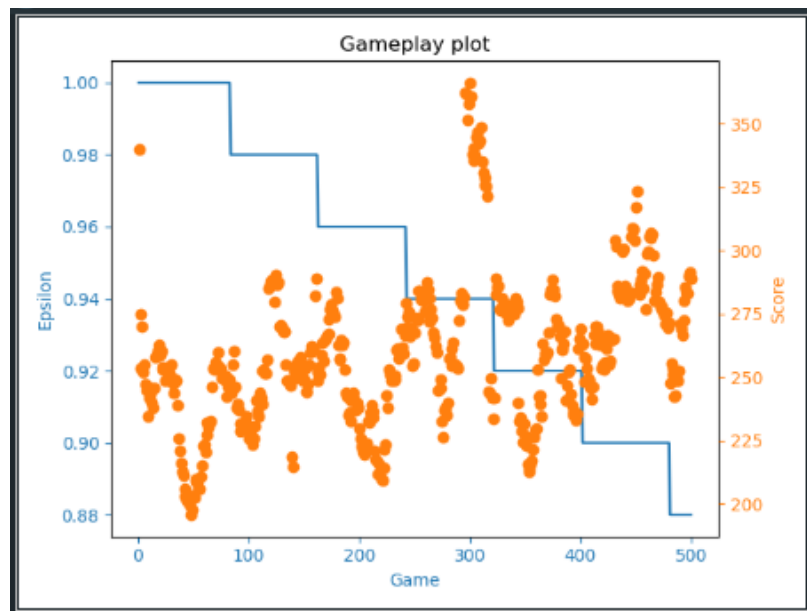


Figure 6.6 The First Phase Scores History Graph

The agent started to frequently achieving score more than 400 points as shown in the scores history graph depicted by figure 6.7. The number of high scores achieved also shown an improvement as the score graph is updated to more than 2500 points. The epsilon is set to follow the progress that has been made in the first phase with 0.05 values addition.

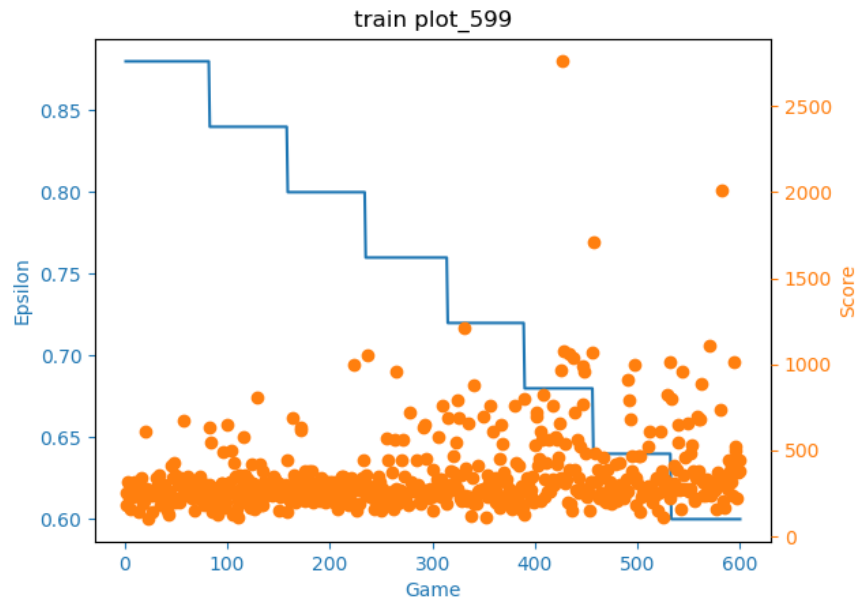


Figure 6.7 The Second Phase Scores History Graph

At the third phase of the training session, the agent is also loaded with the second phase training's weight and the latest epsilon of the second phase. The agent performance at the beginning is much more improved then the agent's performance at the beginning of the first phase. At the end of the third phase, the agent reached a maximum score at the 462nd episode with 4020 points as shown by figure 6.8. Around 700th episode, the

maximum average score is peaked to 877.7. The second evaluation agent's gain 651.8 evaluation points.

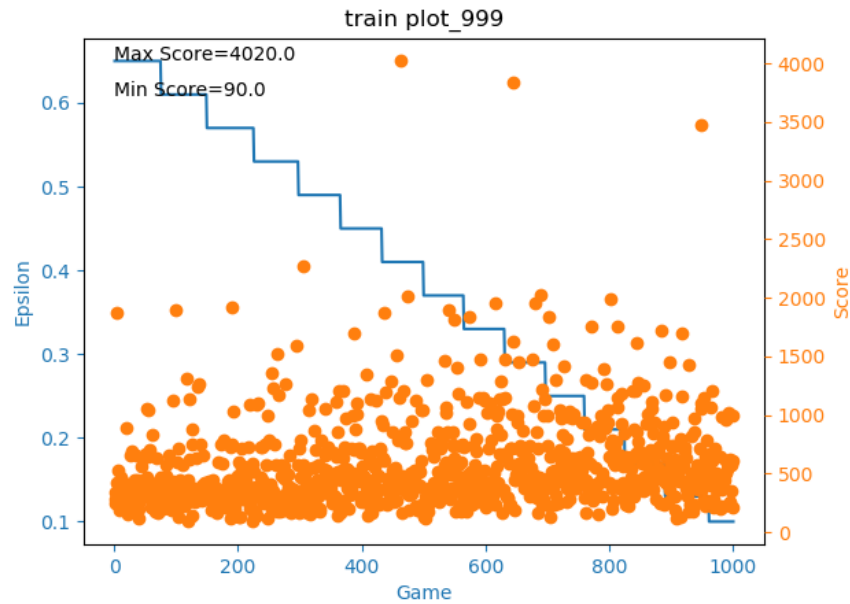


Figure 6.8 The Third Phase Scores History Graph

The agent performance in the default case shows a promising behaviour (indicated by the agent's performance improvement in achieving new higher score). The average scores history graph is presented to gives more details on the agent's performance through figure 6.9 below. The figure shows that throughout the learning the agent has shows a good trend of improvement. However, after the agent pass 500th episodes, the agent shows stagnant performance that almost lead to performance deterioration.

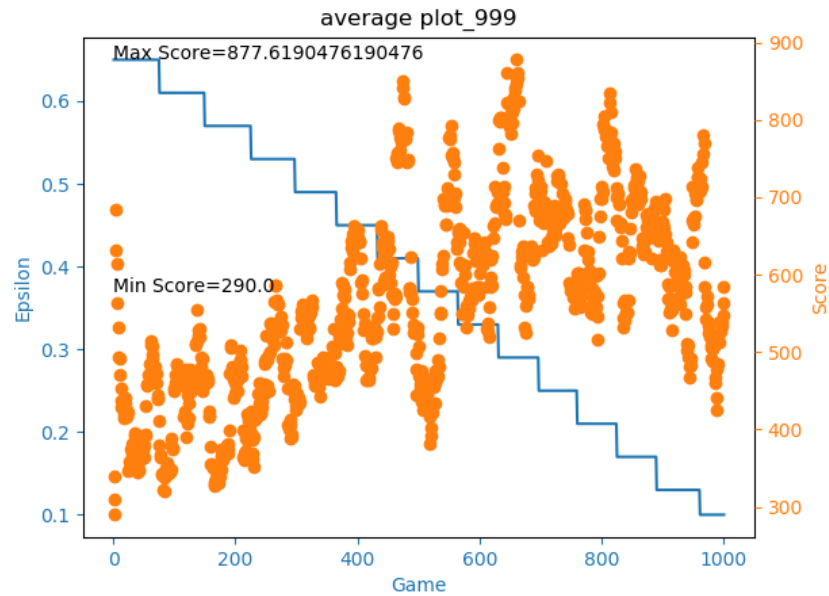


Figure 6.9 The Third Phase Average Scores Graph

The agent's stagnant score after passing 500th episodes and the downfall in the latest episode is assumed to happen because of the agent's behaviour to *exploit* the taken action that is not resulting in desirable result as the agent's performance and behaviour around 1000 episodes is not perfect which means the agent is actually still learning. Hence, the agent which exploit the non-desired action got a downgrade in its performance.

The author assumption is gotten from the behaviour of the algorithm, where epsilon indicated the agent's behaviour of exploring and exploiting the taken action. Figure 6.9 also support the assumption, the figure shown that the agent began to give a static score since episodes 500.

6.3 Additional Experiment

The last experiment is conducted to test the code performance in a different Atari games environment. Breakout is a game where a layer of bricks is stationed on the top third of the screen and the goal of the game is to destroy all of the bricks using a bouncing ball that the player can hit using a panel that could move horizontally. The breakout game screen is shown through the figure 6.10.

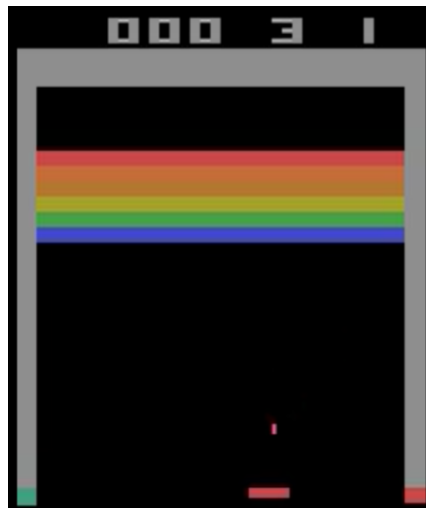


Figure 6.10 The Breakout's Game Screen

The game of BreakOut is relatively short. The test of 1000 episodes that was conducted in the Google Colaboratory only lasted for 3 hours and 28 minutes. As the rate of the epsilon decay is pretty slow compared to the agent's speed in playing an episode, the agent is still at the phase of exploration at the end of the experiment. However, from

1000 episodes observation, it is shown that the agents delivers a similar performance in the different Atari game environment.

Figure 6.11 shown that through the means of exploration, the agent was able to show improvement of hitting the bricks after 600 episodes. Before the end of the experiment, the agent is shown to be taking a possible exploration route that could lead it to achieve a better score indicated by the downfall of the performance.

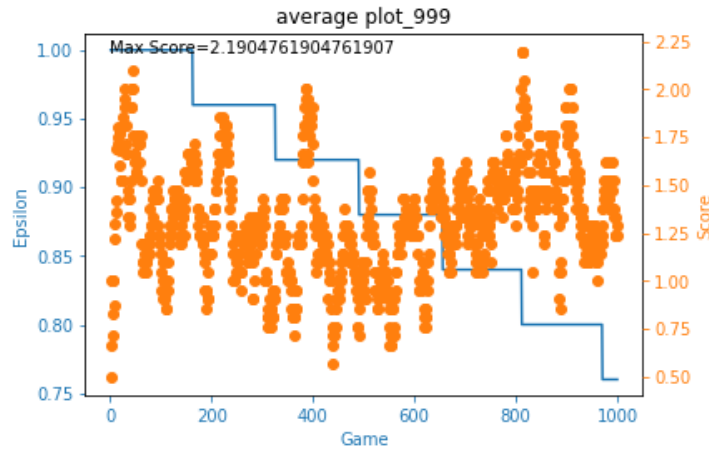


Figure 6.11 The Breakout Average Scores Graph

To support the author assumption, the figures of the agent's score history are presented. These figures, depicted by figure 6.12, shows the agent's progression after passing 100, 400, 700, and 1000 episodes. The first 100 episodes shows that the agent's mainly got to hit zero to two bricks. After 400 episodes has passed, the score history shows that the agent has hit zero to two bricks a lot of time, however the agents also show improvement in hitting three to five bricks that is shown by the increasing number of the

recorded score. Passing the 700 episodes, the agent shown to be increasing the number of the score in hitting to three to five bricks, similar to the 400 episodes graph. However, at the end of the episodes indicated by the 1000 episodes graph, the number of recored score in hitting six and seven bricks started to increase.

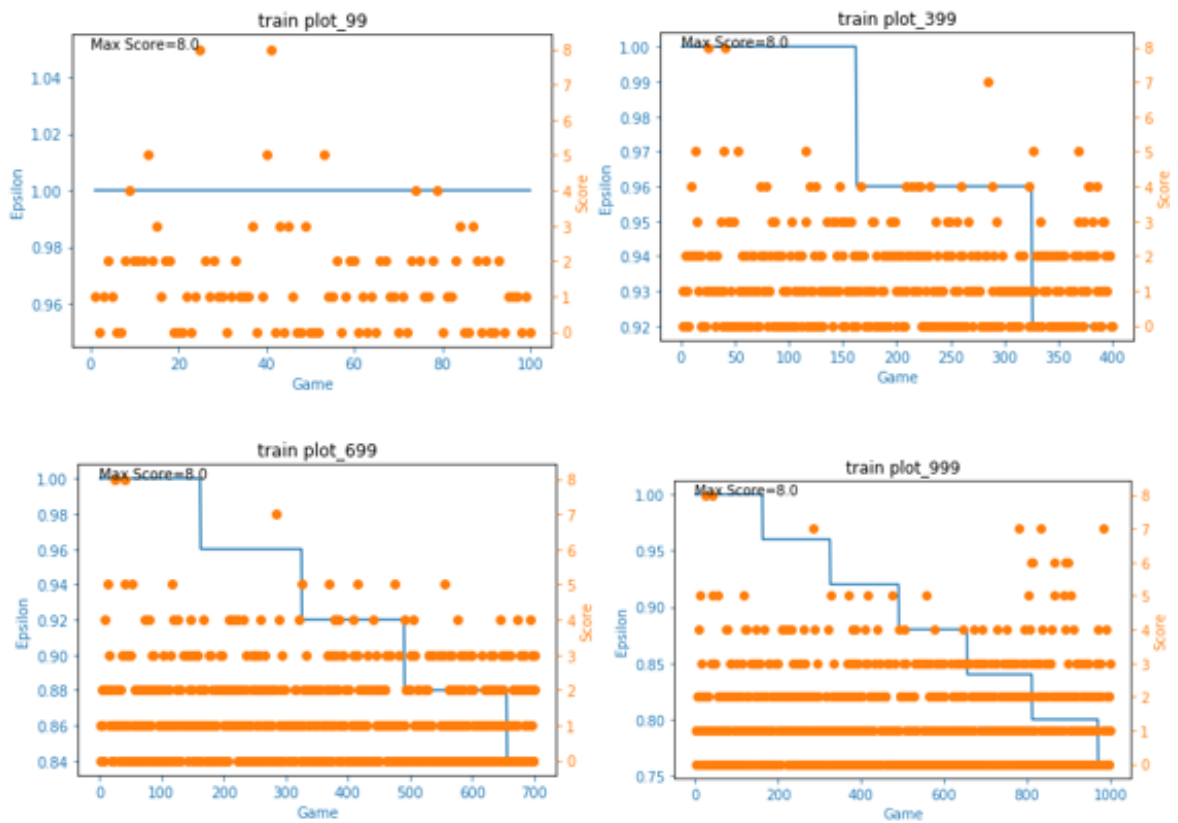


Figure 6.12 The Agent's Score Progression Throughout 1000 Breakout's Episodes

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Even though the agent training progress shows a good trend of improvement, the thesis cannot be said successful in creating an agent that could master and exploit Ms. PacMan. The author did implement the algorithm featured in Mnih paper in the presented code. However, due to the experimental parameters, which is a combination of an experimental and recommended values, the code's performance may not be optimal. There are a lot of sources that gives a better approach to either construct the algorithm or set the recommended parameters, but these approaches require more expert knowledge that needs to be understood and studied which requires a lot of time.

The author found that the choosing action mechanism works as intended. The agent firstly discovers new possibilities that it could achieve in the environment until the agent exploit a good approach that it could do in order to get a better result. This behavior proved that the algorithm works as intended. At the ends of the experiment, the agent is able to move with less stuttering per frame and explore a large amount of area and utilizes some power pills before it run out of lives.

Looking at the results of the experiment, the author believes that the reinforcement learning could be a great approach in solving real world problems. The author encourages

another student to studies this field. It should be noted that a further knowledge is required as real-world problem is more complicated than a game which means that basic reinforcement learning algorithm will only be a fundamental to solve the problem.

7.2 Future Works

As mentioned before, the main limiting factor of the thesis is the hardware requirements. The improvement of this thesis could be done by running the program in a stationary and powerful computer, or at least a computer that could run without any interruption, that could handle a hundred thousand gameplays. Besides hardware supports, the researchers could also build multiple agents supported by model-based algorithm for Ms. PacMan [19]. The idea of using multiple agent is to divide the agent's job, therefore the Ms. PacMan agent has a more focused simpler tasks rather than using one agent to learn to avoid the enemy, gather up pellets, gather bonus fruits, and exploit power pellets at once.

There are several techniques that could be used to improve DQN, as DQN itself is a simple algorithm that has a few disadvantages. However, not every improvement algorithm really improved the performance, as there are many trades off that needs to be considered. Two algorithms that could be used to improves DQN are:

1. Maximum Q values bias reduction (Double Deep Q Network Algorithm)

The Deep Q Network algorithm update the Q Values by taking the maximum next Q values. This behavior causes an issue of overestimating a slightly lower

Q Values that could work better than the maximum Q values. Rather than using target network to get the maximum Q Values, the main network will search for the estimation for the maximum Q values and the target network will be asked the of how high the Q value for the estimated action is.

2. Removing the need of target network (DeepMellow Algorithm)

Seungchan Kim mention two main drawbacks of using target network [13]. First, the target network unable to continually updating the Q-function without time delay. Second, having two network uses a lot of memory resources. Thus, Seungchan Kim proposes the improvement of DQN by simply remove the needs of target networks and utilize the new softmax operator called *Mellowmax*.

REFERENCES

- [1] Andrew N. G, Lecture Notes, Topic: “Part XIII Reinforcement Learning and Control.” CS229, Stanford Engineering, Sept., 19, 2019. [Online]. Available: <http://cs229.stanford.edu/notes/cs229-notes12.pdf>
- [2] D. Takeshi, *Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games*, Nov. 25, 2016. [Online]. Available: <https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/>
- [3] E. G. Learned-Miller, *Introduction to Supervised Learning*, University of Massachusetts, February 17, 2014. [Online]. Available: <https://people.cs.umass.edu/~elm/Teaching/Docs/supervised2014a.pdf>
- [4] Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec., 19, 2013. [Online]. Available: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>
- [5] Musumeci et al. (2018). An Overview on Application of Machine Learning Techniques in Optical Networks [PDF document]. Retrieved from <https://arxiv.org/pdf/1803.07976.pdf>
- [6] N. Tziortziotis, K. Tziortziotis, and K. Blekas. (2014). Play Ms. Pac-Man Using an Advanced Reinforcement Learning Agent. 71-83. 10.1007/978-3-319-07064-3_6.
- [7] OpenAI Five. (n.d.). Retrieved from <https://openai.com/five/>
- [8] Pfau et al. (2018). *Gym Retro screenshots collage showing Atari and Sega games environment*. [Screenshots collage]. Retrieved from <https://openai.com/blog/gym-retro/>
- [9] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge: The MIT Press, 2015. Accessed on: Nov 19, 2019. [Online]. Available: <https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf>

- [10] R. Meo, “Deep Q-Learning with Features Exemplified By Pacman,” B.S. Thesis, Sch. of Eng. and Appl. Sciences., Hamburg Univ., Cambridge, MA, n. d. Accessed on: Sept., 19, 2019. [Online]. Available: http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4168/pdf/Roland_Meo_BA_Thesis.pdf
- [11] The Google DeepMind Challenge Match. (n.d.). Retrieved from <https://deepmind.com/alphago-korea>
- [12] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, University of Montreal, January 12, 2018. [Online]. Available: <https://arxiv.org/pdf/1603.07285.pdf>
- [13] S. Kim, K. Asadi, M. Littman, and G. Konidaris, “DeepMellow: Removing the Need for a Target Network in Deep Q-Learning”, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pp. 2733-2739, 2019. Accessed on, Jan. 6, 2020. [Online]. Available doi: <https://doi.org/10.24963/ijcai.2019/379>
- [14] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*, Cambridge: Cambridge University Press, 2014. Accessed on: Nov 29, 2019. [Online]. Available: <https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf>
- [15] Stanford University School of Engineering. *Lecture 14 / Deep Reinforcement Learning*, 2017. Accessed on: Nov. 1, 2019. [Video File]. Available doi: <https://www.youtube.com/watch?v=lvoHnicueoE&t=1415s>
- [16] Standfordonline. *Stanford CS234: Reinforcement Learning / Winter 2019 / Lecture 1 - Introduction*, 2019. Accessed on: Nov. 7, 2019. [Video File]. Available: <https://www.youtube.com/watch?v=FgzM3zpZ55o>
- [17] Standfordonline. *Stanford CS234: Reinforcement Learning / Winter 2019 / Lecture 2 – Given a Model of the World*. Accessed on: Nov. 9, 2019. [Video File]. Available: <https://www.youtube.com/watch?v=E3f2Camj0Is>

- [18] Standfordonline. *Stanford CS234: Reinforcement Learning / Winter 2019 / Lecture 6 - CNNs and Deep Q Learning*. Accessed on: Nov. 10, 2019. [Video File]. Available: https://www.youtube.com/watch?v=gOV8-bC1_KU&t=4150s
- [19] H. V. Seijen et al. (2017). Hybrid Reward Architecture for Reinforcement Learning [PDF document]. Retrieved from <https://arxiv.org/pdf/1706.04208.pdf>