

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. This 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). The author focused on building an AI that will be called agent that is able to think and act rationally.

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

AI technology shows a promising development that could be utilized in a lot of different field. The study and development of AI should be practiced in order to raise the student knowledge of AI trends and approach.

1.3 Research Objective

This study is conducted with the following objectives:

- 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 Waterfall Model of software development process. In this model, the whole software development process is divided into four phases where the outcome of one phase will become the input for the next phase. The four phases in this model are:

- **Requirement Gathering and analysis** – All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** – the requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** – with inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** – All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.

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 modeled 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 1 below.

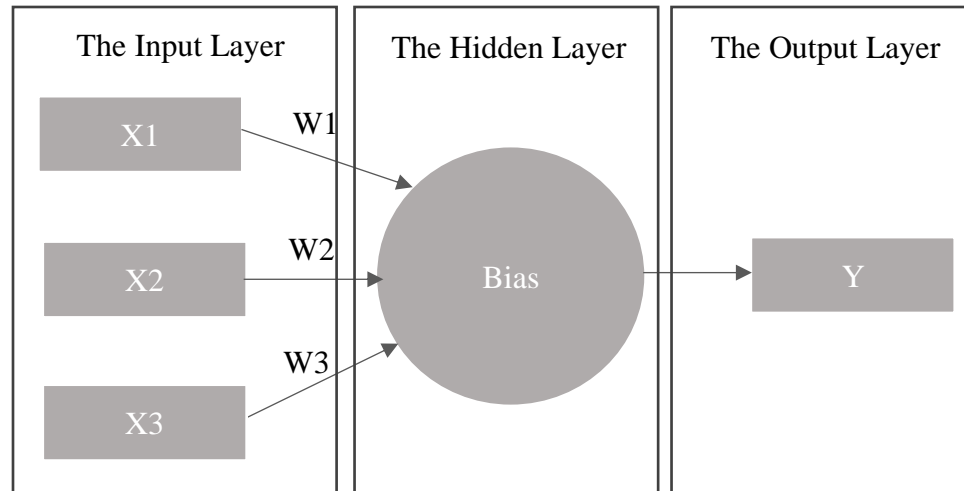


Figure 1. The perceptron

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.

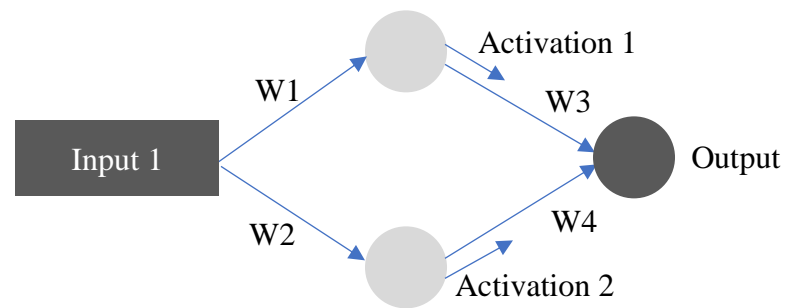


Figure 2. Forward Propagation

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. 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 3.

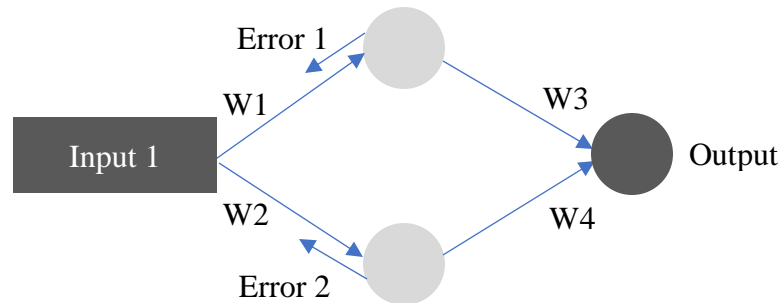


Figure 3. Backward Propagation

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

Reinforcement learning applies the trial and error learning method, where the agents learn the consequences of their known actions in a specific environment with a discrete timestep [1]. At the end of their actions, the state of the environment is evaluated. The agents are given a reward according to the environment's state which can be either positive or negative, the positive reward shown that the agent's actions satisfy our requirement whereas the negative reward do the opposites.

Reinforcement learning cannot be categorized as a form of either supervised or unsupervised learning. The learning algorithm did not require any training data which will be used as a guide by the agent. The agent also did not search, identify, and construct any pattern of the given data. Instead, reinforcement learning is a goal-based agent which maximize the *reward* that the agent can obtain through the means of *exploration* and *exploitation*.

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*.

2.2.2.1 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 1.

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

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*
 $r_t \sim R(\cdot | s_t, a_t)$

- *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 should be *stochastic* to be able to fulfill the Markov Property. Thus, to handle the randomness, the maximum expected sum of reward is taken.

$$\pi^* = \arg \max_{\pi} \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid \pi \right], \text{ with} \quad (2)$$

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

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

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

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* is the expected cumulative reward from following the policy from a state that quantifies the good and bad state.

$$V^{\pi}(s) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi \right] \quad (3)$$

The *Q-Value function* at state s and action a is the expected cumulative reward from taking the action a on state s .

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi \right] \quad (4)$$

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 the reward list is shown below.

Table 1. Ms. PacMan's 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. 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 4 below.

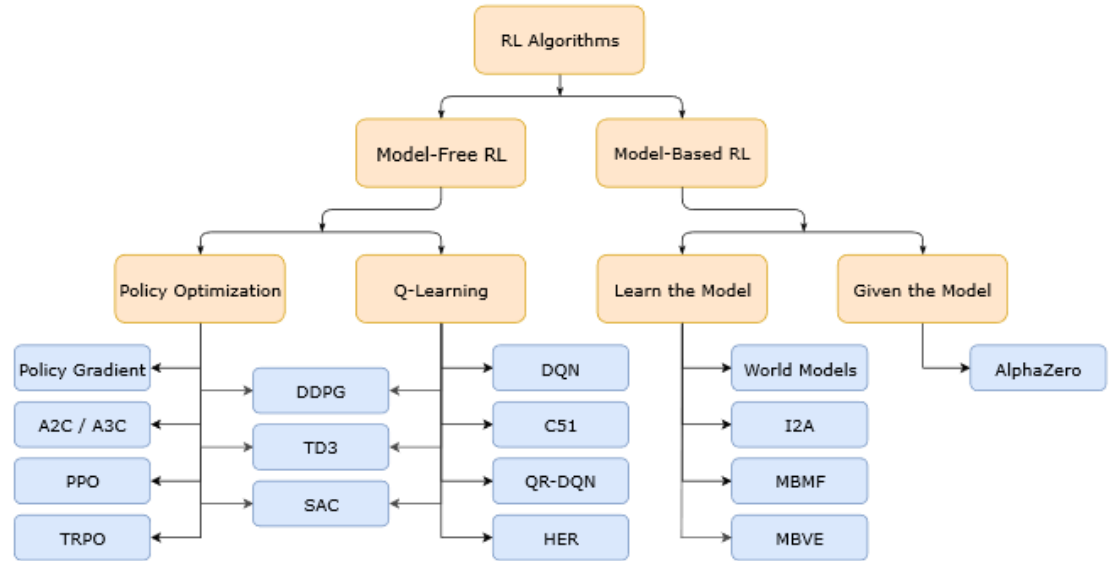


Figure 4. A non-exhaustive, but useful taxonomy of algorithms in modern RL. Reprinted from Part 2: Kinds of RL Algorithm, in OpenAI Spinning Up, 2018, Retrieved September 12, 2019, from https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.

The reinforcement learning algorithm 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 agent 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.

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a) \quad (5)$$

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 is illustrated by the figure of a 2x2 grid games below.

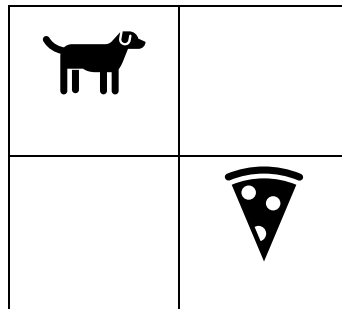


Figure 5. The Dog Agent in 2x2 Grid World

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 3. Initial Q Values Table for 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. In this scenario, if the dog chooses to move to the right cell, then the calculation for the Q Values update will be calculated as the following.

Table 4. The Updated Q Values Table

Current state (s): 1000

Action (a): Right

Reward (r): 0

Discount factor (γ): 0.9

Next state (s'): 0100

Max Q : 0.7

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	0.63	-0.23	1.0	0

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a) = 0 + 0.9 * 0.7 = 0.63$$

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.

$$p = 1 - \epsilon \text{ (Exploitation), or} \quad (6)$$

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

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 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.

$$Cost = \left[Q(s, a; \theta) - \left(r(s, a) + \gamma \max_a Q(s', a) \right) \right]^2 \quad (7)$$

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.

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.

Thus, game is utilized as it simulates real-life environment which can be used to trains AI to minimize costs and risks in order to achieve greater task. The author utilizes Arcade Learning Environment (ALE) which is a dedicated simple object-oriented framework for hobbyists and AI researchers to developed AI agents using Atari games as shown by the figure 6 below. Additionally, the author uses OpenAI Gym, which is an enhanced toolkit for creating an agent trained by RL which uses ALE.



Figure 6. Gym Retro screenshots collage showing Atari and Sega games environment. Reprinted from Gym Retro, in OpenAI, 2018, Retrieved from <https://openai.com/blog/gym-retro/>. Copyright 2018 by OpenAI. Reprinted with permission.

2.7 Related Works

The most popular paper that the author finds throughout his research in this topic is the paper called “Playing Atari with Deep Reinforcement Learning” by Mnih et al. [4]. The 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 their reinforcement learning algorithm which is shown in the figure 7 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 7. The algorithm for Deep Q-Learning with experience replay by Mnih et al. (2013)

The Deep Q-Learning with experience replay algorithm will be the main algorithm that the author will try to implements and tweaks in his work. Additionally, the other related works that the author found is the bachelor thesis entitled Deep Q-Learning with Feature Exemplified in Atari Pacman by Meo [9]. Meo works on implementing and compares

linear function approximation and deep q learning method. He experiments between different training setup to benchmark and retrieve the most satisfying performance of his algorithm.

Mnih and Meo's works are the base of the author's thesis. In his work, the author would like to take a single main algorithm and then compares it with different type of existing reinforcement learning algorithm. Then, the author would tweak different scenario in his setting and record it to search for the satisfying performance that his algorithm implementation can deliver.

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 8 below.

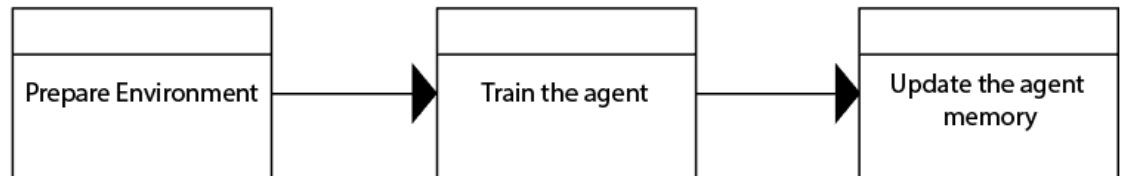


Figure 8. The Functional Analysis Diagram

3.4 Use Case Diagram

Use case diagram, depicted by figure 9, 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.

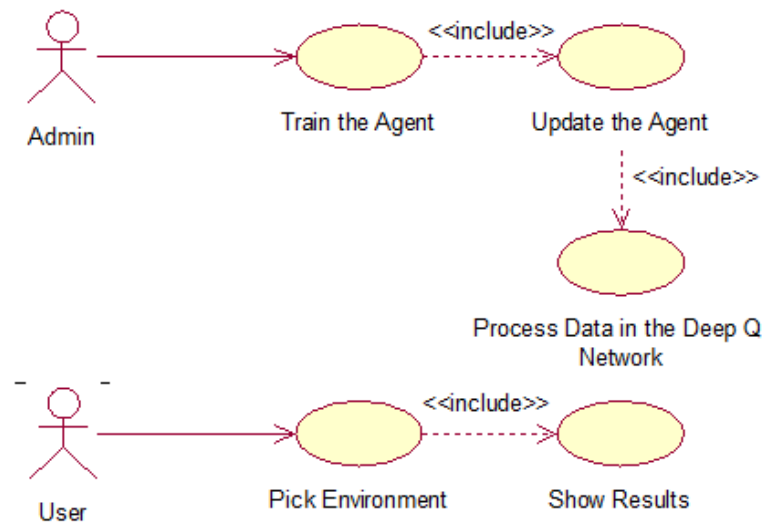


Figure 9. The Use Case Diagram

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 5. Use Case Narrative of Train the Agent Node

Use Case Name	Train the Agent
Goal in Context	The accuracy of the Agent is increased
Primary Actor	The admin
Secondary Actor	None
Precondition	The environment must exist
Trigger	The admin initializes the environment
Scenario	<ol style="list-style-type: none"> 1. The admin initializes the environment 2. The admin starts the training process 3. The application train the agent to maximize the reward it could get 4. The training results are recorded and plotted into a chart
Alternate Scenario	None
Exception	The environment is not exist
Post Condition	The training data are saved

Table 6. Use Case Narrative of Process Data in Deep Q Network Node

Use Case Name	Process Data in Deep Q Network
Goal in Context	The environment data and the agent knowledge are updated
Primary Actor	The admin
Secondary Actor	None
Precondition	The agent is initialized
Trigger	The agent is taking an action that changes the environment
Scenario	<ol style="list-style-type: none"> 1. The agent is taking an action that changes the environment 2. Both of the environment and agent data are processed into the Convolutional Neural Network 3. The agent knowledge base is updated from the retrieved data
Alternate Scenario	None
Exception	None
Post Condition	The agent got the knowledge from the past experience enables it to learn with more accuracy

Table 7. Use Case Narrative of Update the Agent Node

Use Case Name	Update the Agent
Goal in Context	The agent making a progress in every episode
Primary Actor	The admin
Secondary Actor	None
Precondition	The agent is initialized
Trigger	The training to a certain iteration is started by the admin
Scenario	<ol style="list-style-type: none"> 1. The training is started by the admin 2. The agent is initialized with the default environment information 3. The agent takes random action in the environment 4. The action updated the environment which puts the agent in a new environment 5. The agent stores the knowledges and continue taking random action
Alternate Scenario	None
Exception	If the agent exploits the short-term reward and do not seeks a long-term reward, the agent would not get the best performance.

Post Condition	The agent made a progress in making a decision until it get the best performance or until it reaches the end of the iterations.
----------------	---

Table 8. Use Case Narrative of Pick Environment Node

Use Case Name	Pick Environment
Goal in Context	The environment is initialized
Primary Actor	The user
Secondary Actor	None
Precondition	The application is running
Trigger	The user chooses a specific environment from the combo box
Scenario	<ol style="list-style-type: none"> 1. The user chooses an environment 2. The environment is initialized
Alternate Scenario	None
Exception	None
Post Condition	The environment is shown to the user

Table 9. Use Case Narrative of Show Results Node

Use Case Name	Show Results
Goal in Context	The training results is shown to the user through a chart
Primary Actor	The user
Secondary Actor	None
Precondition	The user has chosen a specific environment
Trigger	The environment is initialized
Scenario	<ol style="list-style-type: none"> 1. The environment is shown to the user 2. The results of the agent's training are shown through a chart
Alternate Scenario	None
Exception	None
Post Condition	The result chart is shown to the user

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.

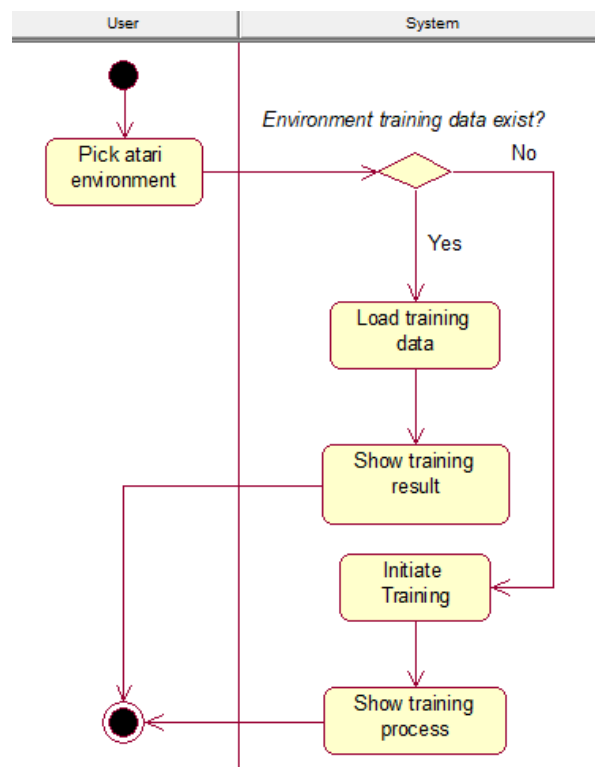


Figure 10. Pick Environment Activity Diagram

Figure 10 above is derived from the *Pick Environment* node of the Use Case Diagram. The user who picks the environment will trigger the system to check whether a training data is exist in the environment. If the training already conducted, then the application will simply show the training result to the user. However, in the case where a training is not yet conducted, then the training will be initiated and the process will be visible to the user through the chart generated on the fly when the training is running. The *Initiate Training* node itself is explained in more detail through the figure 11 below.

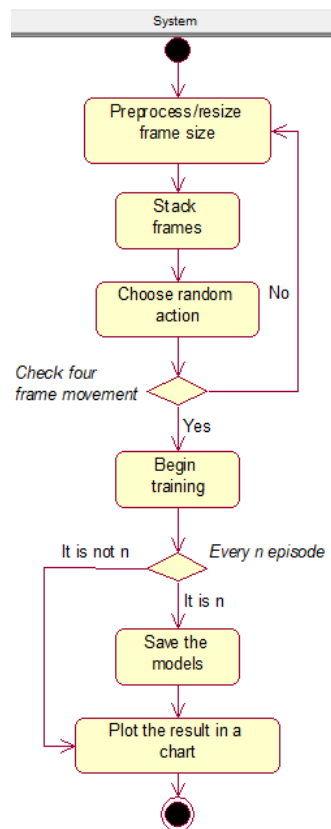


Figure 11. Initiate Training Activity Diagram

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 12. 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.

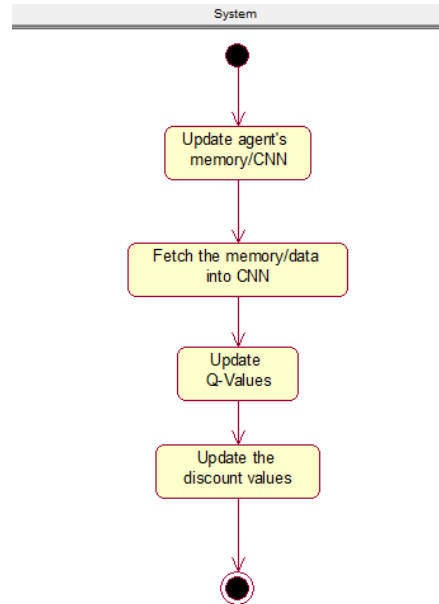


Figure 12. Begin Training Activity Diagram

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 10 and 11 below show respectively the software and hardware requirements to runs this application.

Table 10. 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 11. 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 12 below depict the data design for the application.

Table 12. 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 13.

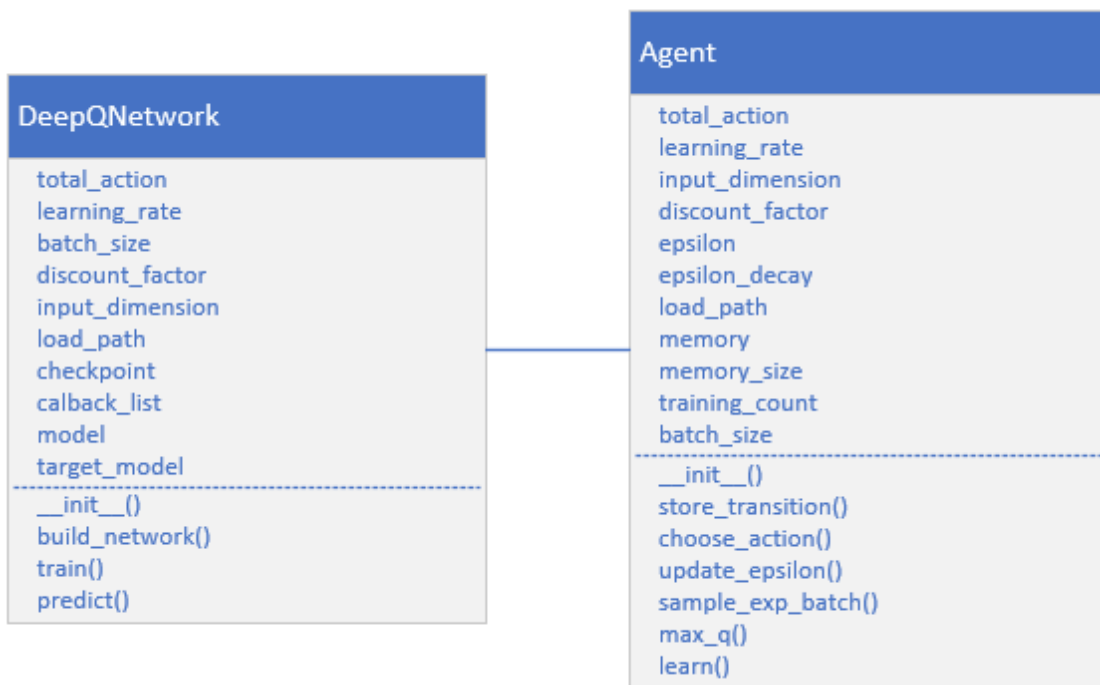


Figure 13. The Application Class Diagram

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 th 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 14 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 rest of the library calling are the OpenAi gym's library which handle the Atari games environment.

```
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

from gym.wrappers import Monitor
```

Figure 14. 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 15, 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 15. The Network's Class Init Constructor

The class parameters are explained below:

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 16.


```

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 16. The 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 17 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 17. 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.

$$Q(s, a) = r(s, a) + \gamma \max_{\alpha} Q(s', a; \theta) \quad (4)$$

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. In the training code indicating by the calling of the Keras's `model.fit` function, the checkpoint for the training is also handled by the *callbacks* parameter. figure 18 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 18. 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 19 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 19. 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 stores 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 that are initialized in the constructor of this class as shown by the figure 20.

```
# 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 20. The agent' 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 21.

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

Figure 21. The agent's memory

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 22 below.

```
# 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 22. The agent's network

The full code of the *Agent* class constructor is shown through the figure 23 below.


```

class Agent(object):
    def __init__(self, total_action,
                 learning_rate=0.00025, input_dimension=(210, 160, 4),
                 batch_size=32, discount_factor=0.99,
                 memory_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 23. The agent's constructor

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 24.

```
# 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 24. The agent' memory storage 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 25, 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 25. 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 26.

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

Figure 26. The exploration codes

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 27.

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

Figure 27. The exploitation codes

5.3.4 *Update Epsilon Function*

The code shown by figure 28 which defines the way to handle exploration and exploitation behavior is implemented in update epsilon function as shown by the figure below.

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

Figure 28. The exploitation codes

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 29.

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

Figure 29. Randomly choosing an episode from the 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 30.

```
# 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 30. The sampling function

5.3.6 Max Q Function

The max q function, depicted by figure 31, is created to fetch the maximum q values that the agent has by prediction. It is used for the agent evaluation's purpose.

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

Figure 31. The max Q function

5.3.7 The Learning Function

The learning function, depicted by figure 32, called the main calculation in the network class. For each training session, the training's counter will keep track of the history of the number of training sessions 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 32. 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 33 below.

```
ALPHA = 0.00025
EPSILON = 1
EPSILON_DECAY = 9e-7
GAMMA = 0.99
INPUT_DIMS = (4, 110, 84)
BATCH_SIZE = 32
MEMORY_SIZE = 1024
MAX_EPISODE = 50000
ENV_ID = 'MsPacmanDeterministic-v4'

IMG_SIZE = (84, 110)
```

Figure 33. The hyper parameters as 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 34 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 34. The preprocess and frame stacking functions

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 Evaluation Function

The evaluation function is created to evaluate the agent's performance throughout the training session. The function purpose is to calculates the maximum mean score that

could be obtain 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 35 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 35. The 1st section of evaluation 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 36.

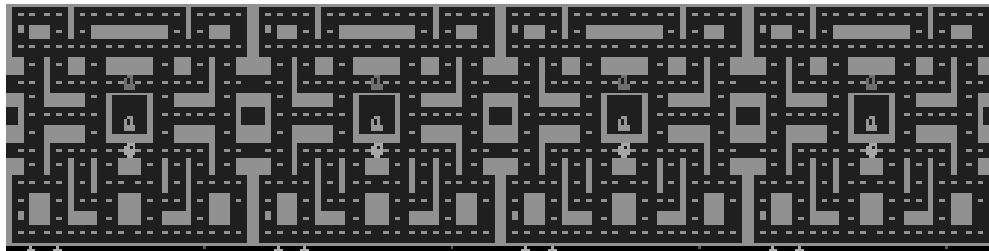


Figure 36. The preprocessed 4 stacked initial game frame

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 37.

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

Figure 37. The 1st section of preprocess function

The result of the image color scheme conversion and resize is shown through the figure 38.

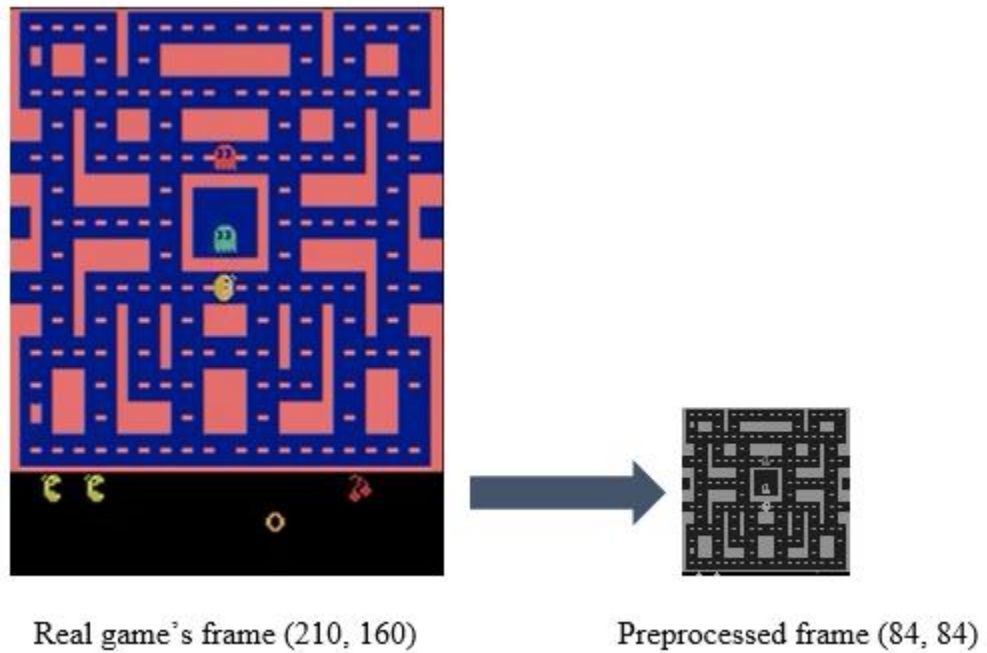


Figure 38. The transformation of preprocessed game frame

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 39.

```

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

```

Figure 39. The 2nd section of the 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 40 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 40. The 2nd section of the evaluation function

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 41, 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 41. 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 42 below.

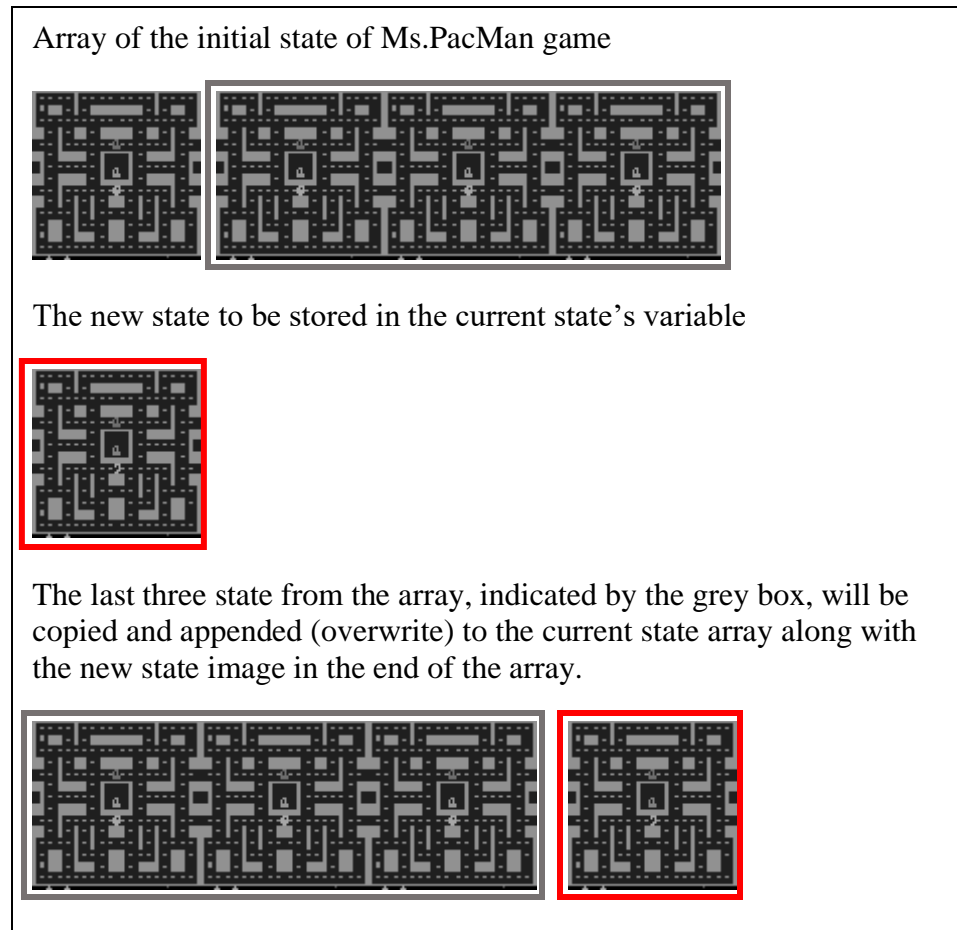


Figure 42. The shifting frame mechanics

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 43.

```

        # 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])

```

Figure 43. The 3rd section of the evaluation function

The final phase of the evaluation function is located outside the loops. The maximum score and its location in the array then will be taken. Then, the loop will continuously search for the best average score that the agent could get. A checkpoint that contains the weights is also created. The max score then returned which mark the end of the function. The final phase of the evaluation function is shown through figure 44 below.

```

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_weights(str(scores[len(scores)-1]) + '_best.h5')

return scores[max_idx, :].ravel()

```

Figure 44. The 4th section of the evaluation function

The whole evaluation function is shown through the figure 45 below.

```

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

```



```

        # 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 45. The evaluation function

5.4.4 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 46 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
score_array = []
mean_array = []
state_array = []
q_array = []

# Counters
episode_counter = 0
frame_counter = 0
```

Figure 46. 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 47 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 47. 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 48.

```

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 48. 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 utilization is started by storing the transition into the agent's memory by calling the *store_transition* function. When the memory has 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 49 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) >= 200:
    if timestep % 4 == 0:
        agent.learn()

if agent.training_count % 500 == 0 and agent.training_count >= 500:
    agent.networks.target_model.set_weights(agent.networks.model.get_weights())
    del score_array[:]
    del mean_array[:]

    agent.update_epsilon()
    stacked_frames = next_state
    score += reward
    timestep += 1

```

Figure 49. The experience replay and training mechanics

The evaluation will start to take place for every 2000 game frames at the end of the episode's loop. The training and score evaluation variables will hold the evaluations score taken from the calling of the evaluation function. The testing states will be generated 30 times, then the scores will be updated along with the mean Q Values. The first loop end with the episode counter being incremented and some information have been printed into the console.

```

if frame_counter % 10000 == 0:
    t_evaluation, score_evaluation = evaluate(agent)
    print('t_evaluation:', t_evaluation)
    print('score_evaluation:', score_evaluation)

if len(state_array) < 30:
    for x in range(random.randint(1, 5)):
        state_array.append(agent.memory[randrange(0, len(agent.memory))]['state'])
else:
    score_array.append(score)
    q_array = [agent.max_q(state) for state in state_array]
    mean_array.append(np.mean(q_array))

print('Score:', score)
# print('Lives:', info['ale.lives'], 'Done:', done)
print('Eps:', episode_counter, 'Training #', agent.training_count)
episode_counter += 1

```

Figure 50. The evaluation and testing mechanics

The model then will be trained according to the number of episodes defined. The example of the output log is shown by the figure below.

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 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 13. Testing scenario

	Epoch	Learning Rate
Case A	1	0.3
Case B	100	0.025

6.2.2 Both Scenarios Initial Result (Result at the 1st Episode)

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.



Figure 52. The agent initial episode behavior



Figure 53. The agent run into the threat which demised the agent's live

At the end of its live, the agent manage to gathers 220 episode without any utilization of **power up** bullet which can power up Ms.PacMan to eat the ghost without dying.

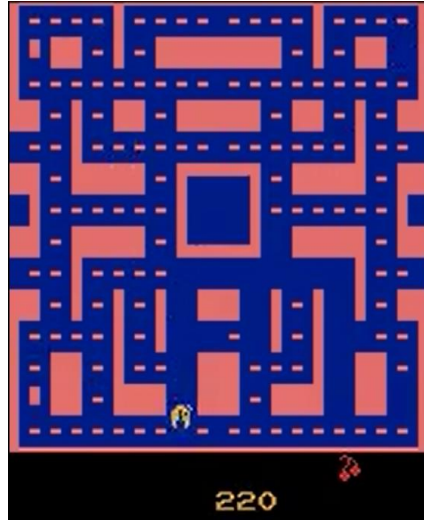


Figure 54. The agent's out of live

6.2.3 Case A - The Agent at the 100th Episode – 12595 Training Session

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

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 55. The training result from episode 80 - 110 from Case A

6.2.4 Case B - The Agent at the 104th-Episode – 13023 * 10² Training Session

During its 100th gameplay, the agent has undergone 13023 * 10² 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.

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 56. The training result from episode 80 - 110 from Case B

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 Conclusion

The thesis is not yet succeeded in creating an agent that could master Ms. PacMan. The agent performance shows a little improvement from the first episode. The best result is shown when the agent is trained using 0.0025 and 100 epoch learning rates. From 100 episode, the training resulted in the agent stable behavior in gathering the food. Though the agent behavior of eating the power up food and eliminating the ghost is more likely to be still a pure coincidence. The main limiting factor in this thesis is the hardware requirement issues. The author encounters with a few of human error that resulted in a handful corrupted checkpoint file with good training history that is worth of 3-4 days training session.

The settings that the author set for the neural network is experimental. The author combines both recommended value and experimental value for the agent parameters. Thus, the parameters used may also not be resulted in an optimal value. The agent's training session however shows a good beginning for it to be a successful agent's that could exploit the game's environment, showed by the Case B testing result, as long as the agent could get a decent number of gameplays.

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. 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 works 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