AHMET AKIN IMPLEMENTATION OF DEEP Q-LEARNING TO THE 3D CONTINOUS ACTION SPACE NEU 2018

IMPLEMENTATION OF DEEP Q-LEARNING TO THE 3D CONTINOUS ACTION SPACE

A THESIS SUBMITTED TO THE GRADUATE SCHOOL OF APPLIED SCIENCES OF NEAR EAST UNIVERSITY

By

AHMET AKIN

In Partial Fulfilment of the Requirements for The Degree of Master of Science In Information Systems Engineering

NICOSIA, 2018

IMPLEMENTATION OF DEEP Q-LEARNING TO THE 3D CONTINOUS ACTION SPACE

A THESIS SUBMITTED TO THE GRADUATE SCHOOL OF APPLIED SCIENCES OF NEAR EAST UNIVERSITY

By

AHMET AKIN

In Partial Fulfilment of the Requirements for The Degree of Master of Science In Information Systems Engineering

NICOSIA, 2018

AHMET AKIN: IMPLEMENTATION OF DEEP Q-LEARNING TO THE 3D CONTINOUS ACTION SPACE

Approval of Director of Graduate School of

Applied Sciences

Prof. Dr. Nadire ÇAVUŞ

We certify this thesis is satisfactory for the award of the degree of Masters of Science in Information Systems Engineering

Examining Committee in Charge:

Assoc. Prof. Dr. Kamil DİMİLİLER	Department of Automotive Engineering, NEU
Assist. Prof. Dr. Elbrus Bashir İMANOV	Department of Computer Engineering, NEU
Assist. Prof. Dr. Boran ŞEKEROĞLU	Department of Information Systems Engineering, NEU

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as require by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name:

Signature:

Date:

ACKNOWLEDGMENTS

I would like to express my sincere gratitude and thanks to my beloved supervisor, Assist. Prof. Dr. Boran Şekeroğlu, for his continuous guidance and assistance during my graduate studies. His inspiring knowledge and creative thinking have been source of encouragement throughout this work. And my thanks and love would be dedicated to my family for their great confidence in me.

To my country and my family…

ABSTRACT

Nowadays Artificial Intelligence usage many applications and researches actively developing. Artificial intelligence branch machine learning have another branch inside as named reinforcement learning. In recent years Reinforcement learning and deep learning or deep neural networks usage of together show us successful performance in video games, robotics, natural language process and etc. Especially one of reinforcement learning method Q learning implementation with deep learning which is named deep q networks human level performance in video games shows us with artificial intelligence research progress how can reach what kind of level. In this study Deep q network implemented in 3D video game with artificial intelligence bot or agent and tested how can perform with different parameters. This results of this tested experiences evaluated, discussed.

Keywords: Deep Learning; reinforcement learning; q learning; neural networks; deep reinforcement learning

ÖZET

Günümüzde yapay zeka bir çok alandaki uygulamalarıyla ve araştırma konuları ile aktif olarak gelişmektedir. Ve yapay zeka dallarından biri olan makina öğrenimin alt dalı sayılan takviyeli öğrenmenin derin öğrenme ile birlikte kullanılmasıyla son yıllarda video oyunlarında, robotikte ve dil işleme gibi vb. alanlarda başarı göstermiştir. Özellikle takviyeli öğrenme methodlarından birisi olan q öğrenmenin derin öğrenme ile birleşerek derin q-ağı olaran adlandırılan methodun video oyunlarda insan seviyesindeki performansı yapay zekanın araştırmaların ilerlemesi ile nasıl bir seviyede performans sergileyebilecegini göstermektedir.Bu çalışmada DQN 3 boyutlu bir video oyundaki yapay zeka botu ile test edilip nasıl bir performans sergilediği farklı parametreler ile test sonuçları değerlendirilip tartışılmıştır.

Anahtar kelimeler: Derin öğrenme; takviyeli öğrenme; q öğrenme; yapay sinir ağları; derin takviyeli öğrenme

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS AND SYMBOLS

Abbreviations:

AI:	Artificial Intelligence
ANN:	Artificial Neural Network
DFN:	Deep Feedforward Network
DNN:	Deep Neural Network
DQN:	Deep Q Network
DRL:	Deep Reinforcement Learning
FNN:	Feedforward Neural Network
MDP:	Markov Decision Process
MRP:	Markov Reward Process
NPC:	Non-playable Character
RL:	Reinforcement Learning
SGD:	Stochastic Gradient Descent
TD:	Temporal Difference

Symbols:

t:	Discrete time step
A_t:	Action at time t
S_t:	State at time t
H_t:	History of reached during current state
O_t:	Observation at time t
R_t:	Reward at time t
π:	Policy decision making rule
$\pi(s)$:	Action taken in state s
$\pi(a\|s)$:	Probability of taking action a, in state s

$v_\pi(s)$:	Value of state s under policy π
γ:	Discount-rate parameter
P:	Probability
S, s, s':	State
A,a	Action
R:	Reward
R_s:	Reward at state
ε :	Probability of taking a random action in an e-greedy policy
v(s):	Value of state s
q(s,a):	Value of action a, in state s
$v_\pi(s)$:	Value of state s under policy π
$q_\pi(s,a)$:	Value of action a, in state s under policy π
$v^*_\pi(s)$:	Optimal value of state s under policy π
$q^*_\pi(s,a)$:	Optimal value of action a, in state s under policy π
Q:	Array estimates of action-value function
y_i:	Vector of output a neural network
x_j:	Vector of input a neural network
b:	Bias term for neural network
L_i:	Define likelihood or loss function
θ_i, θ_i^-:	Define parameters of Q network at iteration i
k:	Number of actions

CHAPTER 1

INTRODUCTION

Nowadays Artificial Intelligence (AI) technologies using at many different area in our lives. And AI observed to close as human intelligence performance for solve specific problems with researches. AI has been divided into many branches with its development since past. Reinforcement learning (RL) is one of the machine learning branch which is machine learning AI branch also. RL techniques and methods successfully used in backgammon, Atari games, robotic and etc.. In RL we have agent and our agent interact with environment. Agent gain experience via trial-and-error and find optimal policies for solve problem. Main purpose of agent get maximum reward with interaction environment. So there is no supervisor in reinforcement learning. While another machine learning methods have supervisor learning and unsupervised learning.

During solving RL problem we formulize mathematically between interaction of agent and environment with Markov decision process (MDP). MDP successfully modelled in robot control learning, planning problem and game playing problems so standard of sequential decision making (Puterman, 1994). RL techniques and methods can solve certain of level small state space MDPs. Because with larger state space MDP data of process and learning time of agent increasing together. For solve this problem and using computer resources more effective we need to better approximation methods. So in RL using neural networks which is successful non-linear differentiable approximator.

In this study we will use Deep neural network (DNN) and RL method Q learning combination of Deep q network (DQN) which is successfully has proven Atari games (Mnih V. , et al., 2013). DQN basically use four technique. They are experience replay, target network, chipping rewards, skipping frames. We implement DQN in 3D continuous action space. Developed 3D space our agent goal is destroy or kill enemies in environment. And agent can move 4 direction. When implementation DQN we use 3 technique from 4, experience replay, target network, clipping rewards. Reason of unused skipping frames need CNN as mentioned in article. While usage of CNN in 2D Atari games get long training time like days in existing experimental system. So while getting days in 2D games, 3D must get longer times. For not extending we use DQN with 3 techniques and implement 3D environment. In this implementation and while experiences we try different parameters and tried to find get maximum reward our agent. And results of these experiments shown with graphics. And we found better parameters for getting maximum reward in continuous action space without using CNN.

CHAPTER 2

BACKGROUND

2.1 Introduction

In this chapter, we will give information about methods and techniques used in this thesis. Firstly we start with reinforcement learning. Using to solve which problems, bring what kind of results, solutions for these problems. And solving or approaching to solution use what kind of techniques and methods. But these explained methods and techniques will be focus on this thesis purposes. Secondly deep learning techniques and methods explained like reinforcement learning. After this chapter we will begin to our founded result from experiments using this techniques and methods.

2.2 Reinforcement Learning

Reinforcement learning (RL) sits in center of many different field like computer science, neuroscience, psychology, mathematic and economic. Which they have same branch in studied in other fields for example game theory, control theory, optimal control or etc. like all this studies main problem is found the way optimal decision making and underlying solution for this problem is RL. Of course we will talk about branch of machine learning RL in computer science.

If start to explain RL problem with an example. We want to build a machine which is play chess board game and this machine trained from a supervisor who is human player. And our supervisor train machine with her game experience with in her gaming experience. After finish this training phase we want to match our machine against best chess player of the world. From

staring to playing game our machine must be stuck to make decision or lose game because human player will find a way to win match against our machine. This problem occur from lack of move or our supervisor experience is not enough to train machine against champion human player (Alpaydın). This examples can be increase. As understanding from example we solve problems like this with RL without any supervisor.

If we start to explain RL roughly, in RL we have an agent and environment our agent make decision in this environment. For example environment is chess board, our agent is black or white player. Our agent main goal is gain maximum reward in this environment. And use RL algorithms, try to find novel ways for reach this goal. For example from popular Alpha Go (Silver, et al., 2016) win the match against world champion go player with unique and unpredictable strategies. Reward is scalar number feedback signal in main goal of agent. Of course, this reward depend agents actions and agent position in environment state. And this rewards signals give our agent an idea about how its actions and states good or bad for reach the goal. If we need to be clearer with examples, for an artificial intelligence (AI) robot positive reward is its move any direction without fall down and bad reward is fall down, for power station is produces power positive reward and overheating from high power produce is negative reward. Agent interaction with environment illustrated in Figure 2.1. .

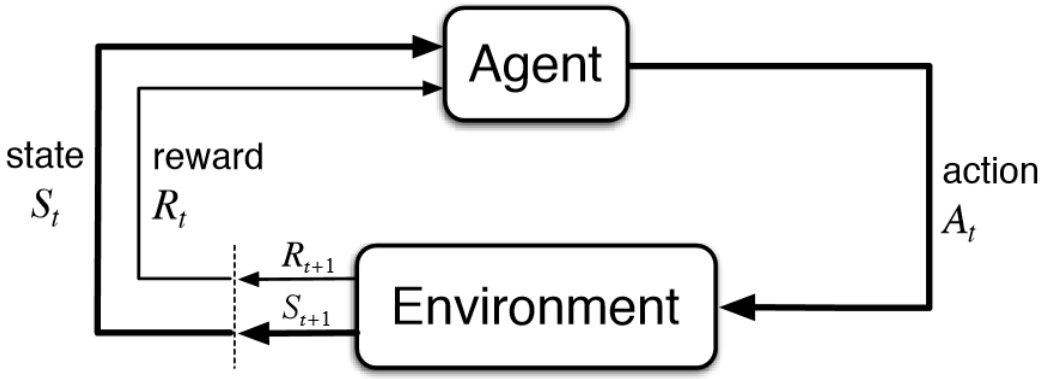

Figure 2.1: RL agent environment interaction (Alpaydın 2010)

If we explain this elements in Figure 2.1, our agent make some sequenced of steps in time t = 0, 1, 2, … N actions A_t in the environment. And get some rewards after making this actions in every time steps, this rewards change in every state S_t after each actions as positive or negative way. As mentioned before our agent's main goal is get maximum reward. Of course this rewards will change in every state with our actions in environment. So agents must select true actions for reach the goal. This observations and rewards can store as history sequenced action in time step:

$$H_t = O_1, R_1, A_1, \dots, A_{t-1}, O_t, R_t \qquad (2.1)$$

And our agent make decision mapping this history H_t. This history function can be called as state S_t.

$$S_t = f(H_t) \qquad (2.2)$$

After define this state there are 3 state environment state which is can know from agent at first time but include rewards and observations, agent state which is used in RL algorithms and store actions and observations in sequence time steps of agent and last one is information state (Markov state) this is can described as formalization of our history in mathematical way. Each observation and information got from state is called Markov property or Markov

Major components of a RL agent is policy, value function, model. For solving any RL problem algorithms include one or more of these components. If we need to explain them;

Policy: we can named as agent's behaviors of sequenced time steps. Kind of map for state to action. This can be deterministic policy $a = \pi(s)$ or stochastic policy $\pi(a \mid s)$.

Value function: is a prediction of future reward from starting state. And give information about badness or goodness about state. E.g.

$$v_{\pi}(s) = E_{\pi}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s] \qquad (2.3)$$

Model: predictions for environment what will do or how to behave and help us to predicts next state and next reward. As show popular example from literature grid world Figure 2.2. , 2.3. and 2.4. We can see our environment have 2 terminate state with +1 and -1 reward. In Figure 2.3. We have our agents' random policies with arrows there for illustrated in mind and Figure 2.4. Show us value of each state.

			+1
	Wall		-1
	Wall		
Agent Start			

Figure 1.2 : Grid world example from literature

→	↔	↓↔	+1
↑	Wall	↕↔	-1
↕	Wall	→↕	←
Agent Start	→	←	↑↑

Figure 2.3: Grid word with random policies π, arrow show as move direction in that state

0.50	0.75	0.45	+1
0.22	Wall	0.30	-1
0.32	Wall	0.75	0.10
Agent Start	0.32	0.60	0.11

Figure 2.4: Grid word with random values v

RL agents can categorized as values based, policy based, action critic, model free and model based .RL algorithms make predictions and control perspective for example temporal

difference TD (0) algorithm make predictions with given policy for solve RL problem we can think +1 reward in grid world example. On the other hand Q Learning algorithm which is we used in this thesis try to find optimal policy for reach to goal (Susson & Barto, 1998).

2.2.1 Markov decision processes

In RL framework generally decision theory formulized as Markov decision process (MDP) (Boutilier, Dean, & Hanks, 1999). And MDP have an important place in modern RL. Prediction and Control algorithms use MDP try to find results for RL problems. MDP generally include set of states, set of actions, set of rewards state transition probabilities and rewards, discount factor this is shown as 5 tuple <S, A, P, R, γ>. If we explain these elements of MDP:

States: As explained before describe our position or an information in environment. As a Markov property each state connected last state.

$$P\,[S_{t+1} \mid S_t] = P\,[S_{t+1} \mid S_1, \ldots, S_t] \tag{2.4}$$

Transition Probability: Next states can be stochastic and its can be our successor states` and defined by

$$P_{ss`} = P\,[S_{t+1} = s` \mid S_t = s] \tag{2.5}$$

These two element create Markov chain which is sequence of random states S_1, S_2 with Markov property. For understanding the Markov chain Figure 2.5. . As seen in figure if our student start from class 1 from that state probability of transition the next states which is

Facebook or class 2 is 0.5 for each action. And from that point sequence of states episodes can occur as probability aspect and we call this Markov process.

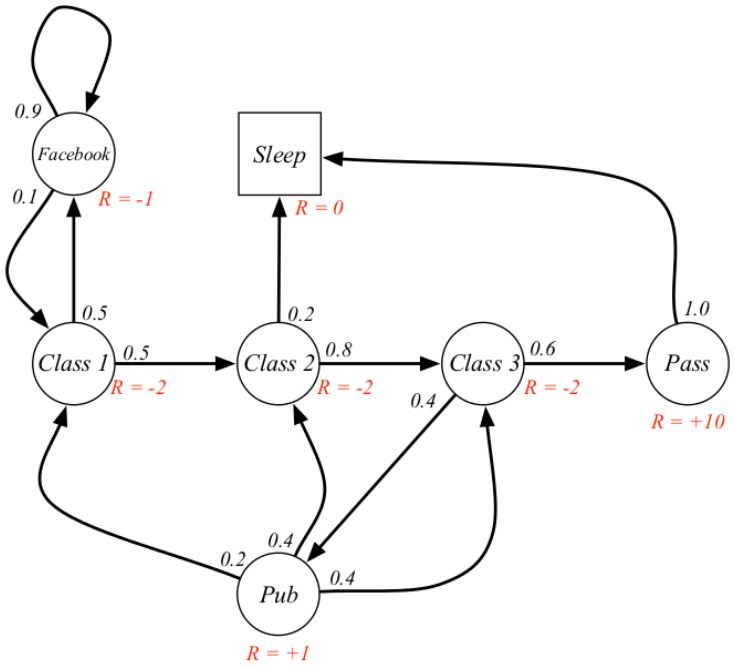

Figure 2.5: Markov chain for student

We try to clear for explain Markov process but there are 2 more element of MDP reward function and gamma γ , these two things called as Markov reward processes(MRP)

Reward function: explain us value of existing agent state as seen Figure 2.6. for each transition between states we get a reward from that state.

$$R_s = E\left[R_{t+1} \mid S_t = s\right] \tag{2.5}$$

Gamma γ: is our discount factor where is $\gamma \in [0, 1]$ and trades off the importance of immediate and later rewards for us.

With these MRP we get return G_t after time-step t by total discounted.

$$G_t=R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2.6)$$

Discount element gamma change 0 to 1 this is help us to evaluate immediate as acted greed or model unknown of environment so we just wait for delayed future reward. This discount used for mathematically convenient purpose.

2.2.2 Value function

Most of RL algorithms evaluating value functions of states. This function give us how state good or bad. If we need to define value function v(s) as respected MDP

$$v(s) = \mathrm{E}[G_t \mid S_t = s] = \mathrm{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s] \quad (2.7)$$

For MRP state value function v(s) is return of from started state the current state. Also we can define similarly value of taking action in a state which is called action-state value function

q(s, a).

$$q(s,a) = \mathrm{E}[G_t \mid S_t = s, A_t = a\ ] = \mathrm{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\ ] \quad (2.8)$$

2.2.3 Policy

As we explained before agents' behaviors of sequenced time steps. A policy π is a distribution over actions given states. In MDP policies are depended the each states so policies different for every states. Of course if model is known by agent these policies can describe again from

earned reward between looking starting state to terminate state. As seen probability of state transition, reward can change by given policy and state transitions can move as policy.

$$P^{\pi}_{ss'} = \sum_{a \varepsilon A} \pi(a|s) P^{a}_{s} \quad (2.9)$$

$$R^{\pi}_{ss'} = \sum_{a \varepsilon A} \pi(a|s) R^{a}_{s} \quad (2.10)$$

Like these two change value functions also can act from give policy and make evaluations.

$$q_{\pi(s,a)=} E_{\pi}[G_t \,|S_t = s\,, A_t = a] \quad (2.11)$$

$$v_{\pi}(s) = E_{\pi}\,[G_t \,|S_t = s] \quad (2.12)$$

Also these last 2 equation can decomposed as bellman equation (Bellman, 1957) 2.13, 2.14. After this point we want to select optimal policies for each value functions and solve the MDP problem. Optimality shown as $V^{*}_{\pi}(s)$ for state value and $q^{*}_{\pi}(s, a)$ for action state value function. So far we explain briefly MDP and bellman equation concepts so RL use these tools for solve the problems with iterative methods for example value iteration, policy iteration , Q learning, Sarsa etc. but we will focus to explain only q learning which is we used in our algorithm for solve RL problem.

$$V_{\pi}(s) = \mathrm{E}_{\pi}[R_{t+1} + \gamma\, v_{\pi}(S_{t+1})\, j\, S_t = s] \quad (2.13)$$

$$q_{\pi}(s, a) = \mathrm{E}_{\pi}[R_t + 1 + \gamma q_{\pi}(S_t + 1;\, A_t + 1)\, j\, S_t = s;\, A_t = a] \quad (2.14)$$

2.2.4 Q Learning

First of all Q learning is a model-free RL method so our agent try to solve the problem without model of environment. While temporal difference (TD) learning try solve problem with given

policy and our agent use this policy as iterative in environment for predict v(s), Q Learning algorithm shown below, make this without any policy it called off-policy with his experiences (Susson & Barto, 1998). And Q learning defined by (Watkins & Dayan, 1992):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \qquad (2.15)$$

Initialize $Q(s, a)$, $\forall s \in S$, $a \in A(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose A from S using policy derived from Q (e.g., e-greedy)
        Take action A, observe R, S_0
        $Q(S, A) \leftarrow Q(S, A) + \alpha R + \gamma \max_a Q(S_0, a) - Q(S, A)$
        $S \leftarrow S$
    until S is terminal

Algorithm 2.1: Q Learning Algorithm (Sutton and Andrew 1998)

With this method or another RL methods can solve small MDP problems without and problem but when MDP begin the increase calculated values, states and actions hold a lot of memory for these things and calculations begin after increasing of MDP will slowdown. For solve this problem and approximation we will use neural networks. We will look it in next titles.

2.3 Deep Neural Networks

Deep neural networks (DNN) help to progress of AI and machine learning for example self-driving cars (Mariusz, et al., 2016), image recognition systems (Simonyan & Zisserman, 2015) and voice recognition (Hinton, et al., 2012) etc. used with really good performance as state-of-art. DNNs provide to as a good structure for approximation the non-linear functions. And in this section we will explain briefly about architecture and techniques.

2.3.1. Neural networks units

Basically artificial neural networks (ANN) modelled the how human brain works. The human brain with complex web of interconnection neurons ability of produce output from given information input. ANN have a human brain like architecture. In ANN have neurons like human brain. These neurons ordered in layers and produces some output values from entered inputs moving in the layers with some calculations as seen in Figure 2.6. . Neuron take vector of inputs x and calculate the weighted sum of inputs, weights denoted as w. The weighted calculation added to bias term and these are passed from an activation function f and neuron produce output y. This calculations equation shown as:

$$y_i = f(\sum_j x_j w_{i,j} + b_i) \qquad (2.16)$$

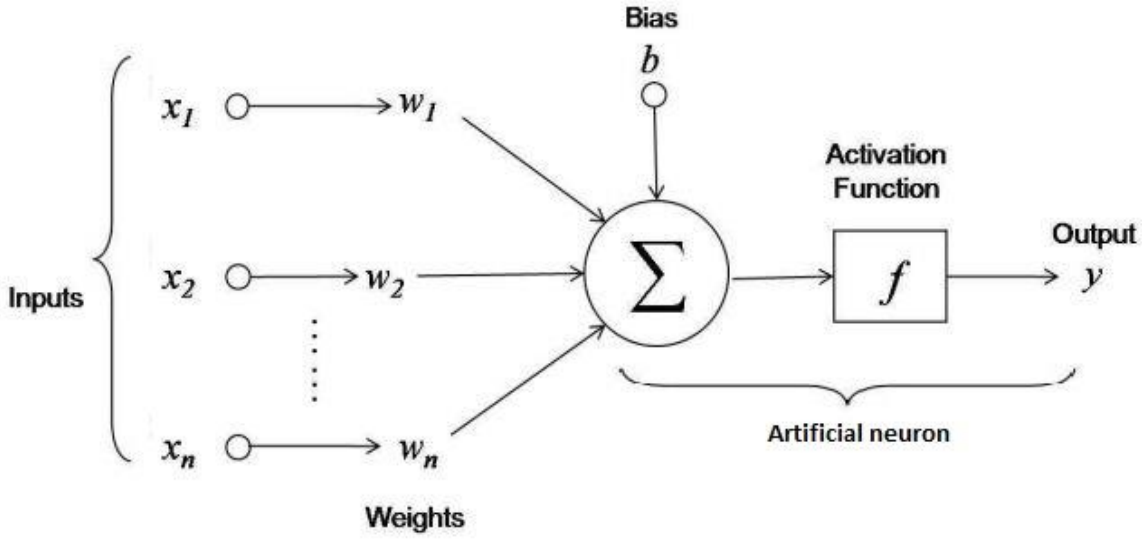

Figure 2.6: An Artificial Neuron (Tanikic & Despotovic, 2012)

And there are three activation function for solve non-linearity in output of artificial neuron. First one is sigmoid neuron result will be 0 to 1,second is tanh neuron this time our output is range between -1 to 1 , another one is restricted linear unit which different from other results

look like hockey stick. We will show these function presentation and figures under this paragraph. (Buduma, 2017) (Ketkar, 2017) (Samarasinghe, 2006)

Sigmoid:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.17}$$

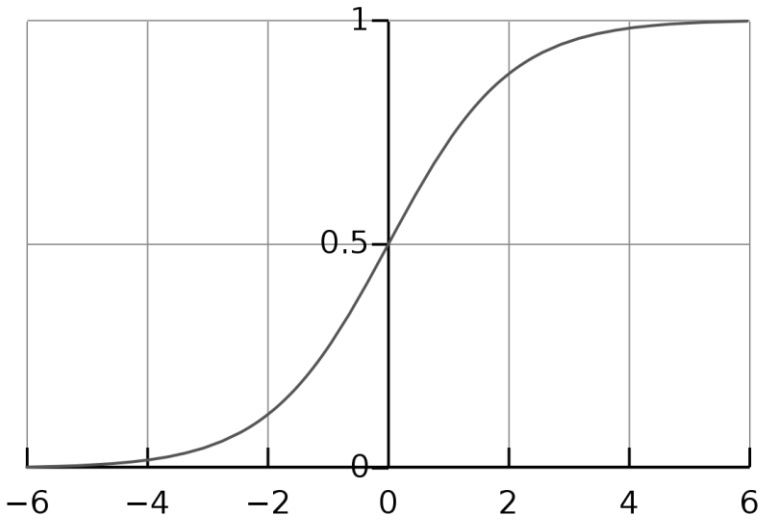

Figure 2.7: Output of sigmoid function varies (Buduma 2017)

Tanh:

$$y = \tanh(wx + b) \tag{2.18}$$

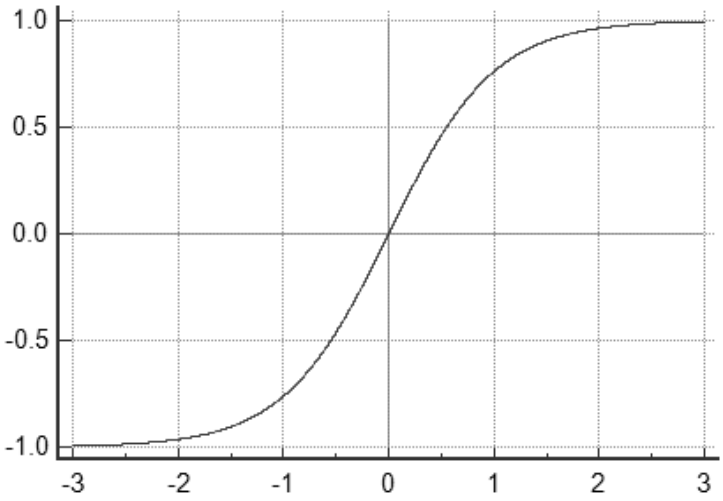

Figure 2.8: Output of tanh function varies (Buduma 2017)

ReLU:

$$f(x) = \max(0, wx + b) \tag{2.19}$$

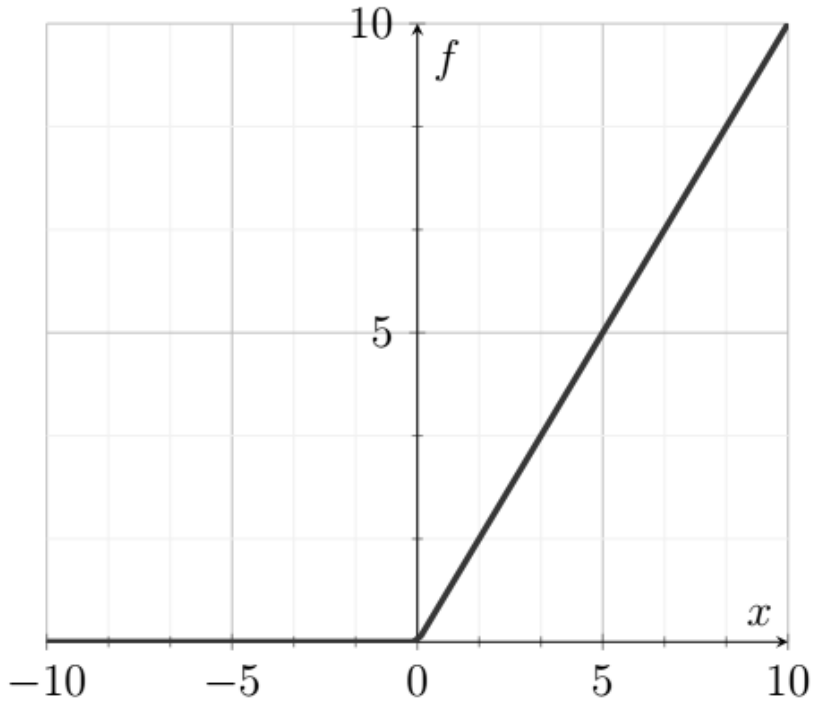

Figure 2.9: The output of a ReLU neuron (Buduma 2017)

2.3.2. Deep Feedforward networks

Deep feedforward networks (DFN) actually called as feedforward neural networks (FNN) this deep word mean that we have more than one hidden layer inside it and a lot of nodes in this hidden layers (Goodfellow, Bengio, & Courville, 2016). So generally as seen Figure 2.10. . FNNs consist 3 layer input layer, hidden layer, output layer. We give some vector x values to input layer this value passing from hidden layer and make calculation through the output layer.

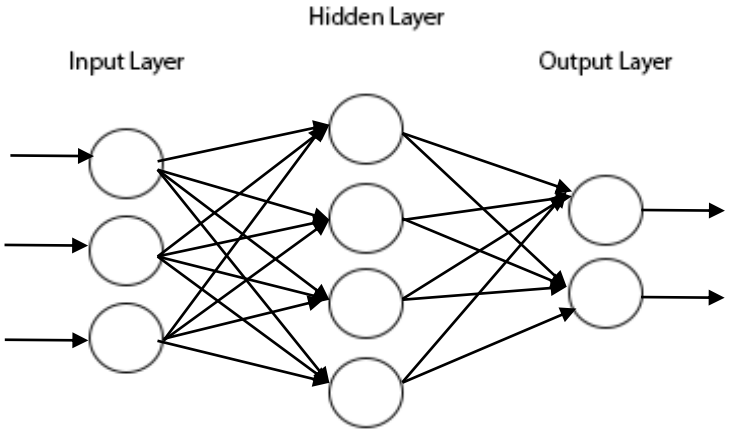

Figure 2.10: Feedforward network layers

As we explained in section 2.3.1. we give this weights as random in feedforward neural network so we need to train this network it's called as backpropagation (BP) Figure 2.11 in literature as named BP make calculations backward from output layer. With BP provide as accuracy for our predicted outputs. And solve this problems in neural networks use gradient descent algorithms during BP. Many years stochastic gradient descent (SGD) has been popular choice for train our feedforward network with BP. But there are developed some other methods from this SGD like ADAM (Kingma & Ba, 2015), AdaGrad (Duchi, Hazan, & Singer, 2011), and Momentum (Sutskever, Martens, Dahl, & Hinton, 2013). And these different methods use learning rate based distribution for training which is we will also compare this methods in thesis experiments.

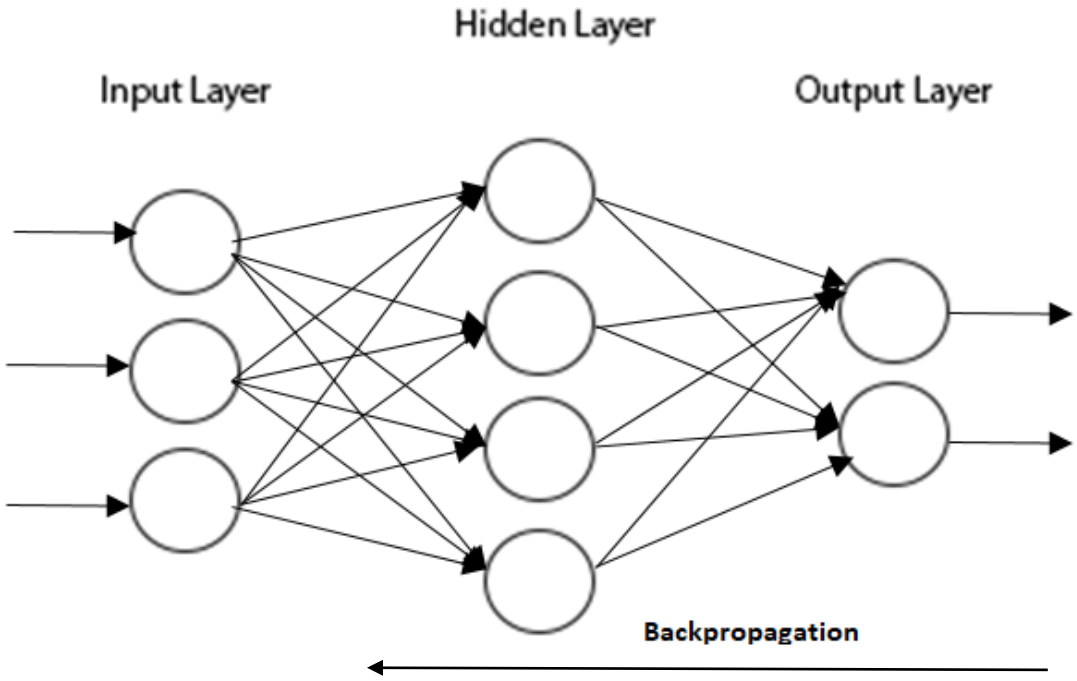

Figure 2.11: Backpropagation

We explain a DNN briefly without any complex formulations for keep to background simple understandable to what we used in our problem solving in work. And there are some problems include this train in DNN how many hidden layer we need to select , which learning rate for gradient descent, which activation function better for algorithm like other parameters have an impact to results (Keller, Liu, & Fogel, 2016). Looking from there adding more hidden layer not means to us all hidden layers nodes work or effect to results (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014).

2.4 Deep Reinforcement Learning

Deep reinforcement learning (DRL) as understanding name and thinking from last titles DNN function approximator usage of in RL value functions and determine the policies. DRL have been making many different success in different areas for example Atari games (Mnih V. , et al., 2013), robotics (Kober, Bagnell, & Peters, 2012), 3D games (Ratcliffe, Devlin,

Kruschwitz, & Citi, 2017) and etc... So as understanding from this examples when we solve small problems with RL, we can solve bigger problems with DRP. Also DRL successfully applied for Q Learning (Gu, Lillicrap, Sutskever, & Levine, 2016) (Mnih V. , et al., 2015).

2.4.1 Deep Q learning

As explained in last sections Q Learning is a model-free RL algorithm. Deep Q Networks (DQN) is combination of Q Learning algorithm and DNN. With usage of DQN observed really good results as human level performance and prove itself (Mnih V. , et al., 2015).

Basically DQN use four main concept while training experience replay, target network, clipping rewards and skipping frames. In our experiences we did not use skipping frames, main reason is Convolutional neural network (CNN) which is used to solve RL problem in Atari games with DQN. Because CNN need really powerful systems to solve problem and used system in this study not enough power render CNN. So there would not any usage in our scope.

With DQN we use experience replay, target network, clipping rewards for stabilize the action value function Q(s, a). As mentioned before Q learning algorithm try to get more reward from experiences. So DQN create some experience replay at buffer for stabilize the q learning problem (Long-Ji, 1993). DQN save RL agents experiences in 4 tuple every time step $<S_t, A_t, R_{t+1}, S_{t+1}>$, and DQN update this experiences with mini batches (or samples) during iteration i. And Q network minimizing a loss function in every iteration i, loss function;

$$L_i(\theta_i) = E_{(s,a,r,s')}\sim U(D)[(r + \gamma max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)^2] \qquad (2.20)$$

γ is discount factor at loss function. θ_i Define parameters of Q network at iteration i and θ_i^- is target network computation at iteration i. θ_i^- target network parameters are updated only with θ_i Q network parameters every defined period time step. As understanding DQN maintains two separate networks. This loss function minimized with using stochastic gradient descent. And behavior policy is use epsilon greedy policy for ensure efficient exploration (Wang, et al., 2016).

Initialize replay memory D
Initialize action value function Q with random weights
Repeat
    Observe initial state s_1
    For t=1:T do
        Select an action a_t using Q (with ε-greedy)
        Carry out action a_t
        Observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
        Sample random transition (s_j, a_j, r_j, s_{j+1}) from D
        Calculate target for each transition
        If s_{j+1} is terminal then
            $y_j = r_j$
        Else
            $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta)$
        End if
        Train the Q network on $(y_j - Q(s_j, a_j; \theta))^2$
    End for
Until terminated

Algorithm 2.2: Deep Q Network, adapted from Mnih V. and others (2015)

CHAPTER 3

RELATED WORK

As explained in last chapter RL agents mostly used in video games. And solve problems using with DNN as a human level. So In this chapter we will use or implement discussed methods and algorithms in our created continuous action space 3D environment without using CNN so from this perspective we look our results and discuss that experienced results.

3.1. Technologies used for in experiments

3.1.1 Unity ml-agents

First of all there not a lot of tools for testing your RL algorithms last years without premade environments like Open AI, hardcoded environment or some other community created environments. Last few years developments in Artificial Intelligence unity game engine deploy an open-source plugin that give chance to develop RL agents for developed games inside. Or in the other perspective develop better game bots, non-playable character (NPC) by developers. Figure 3.1. Shows working principle of this plugin.

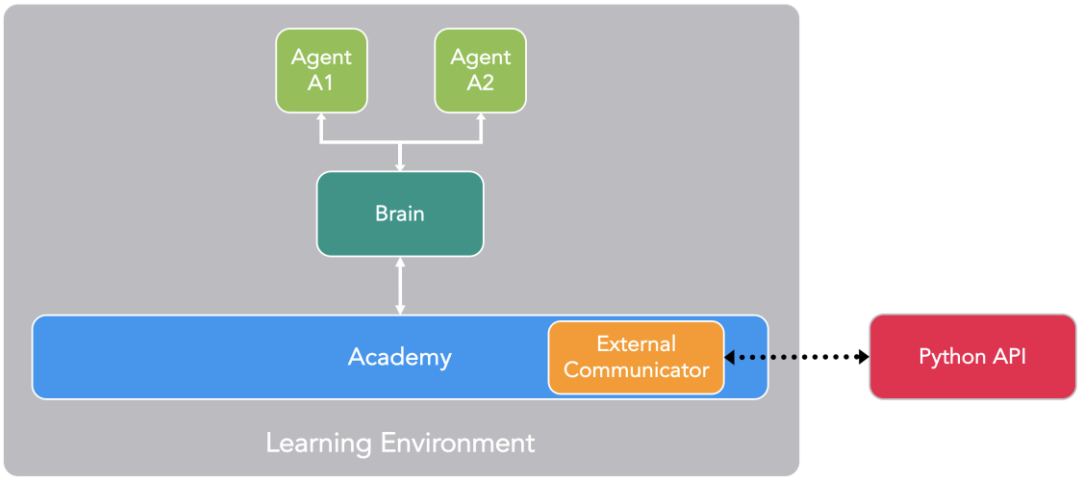

Figure 3.1: General working block diagram of Unity ML Agents working Principle (https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md Reached 14/12/2017)

In unity game engine use c# or javascript for development but as seen Figure 3.1. we have an academy which is help us to communicate environment with our python program create as RL algorithms. And this academy control our agents' brain. There are 4 different types of Brain for developer range of training and inference scenarios:

External: This help us to control agent decisions over python.

Internal: This is where decision are made from Tensorflow model. Basically this plugin create some data after externally trained agent then you can implement that trained data to use inside of developed game.

Player: As understanding control agent as a human when develop environment.

Heuristic: where decision are made using hardcoded behavior of agent.

Briefly Unity ML-Agents help us create different environment for different problems. For example we can simulate our python RL algorithms using this which is we make that in this thesis.

3.1.2. Python Libraries

For communicate with environment and for our algorithm we use different python libraries if we need to briefly mention them Tensorflow library which is help us high performance numerical computations, Numpy library include high-level mathematical functions, Matplotlib library for plot our results.

3.2. Setup

First of all we created a 3D environment using unity game engine. And our agent make its action in this environment space. This environment include an agent, walls, and enemies as our main goal to reach and destroy them. For better understanding environment Figure 3.2..

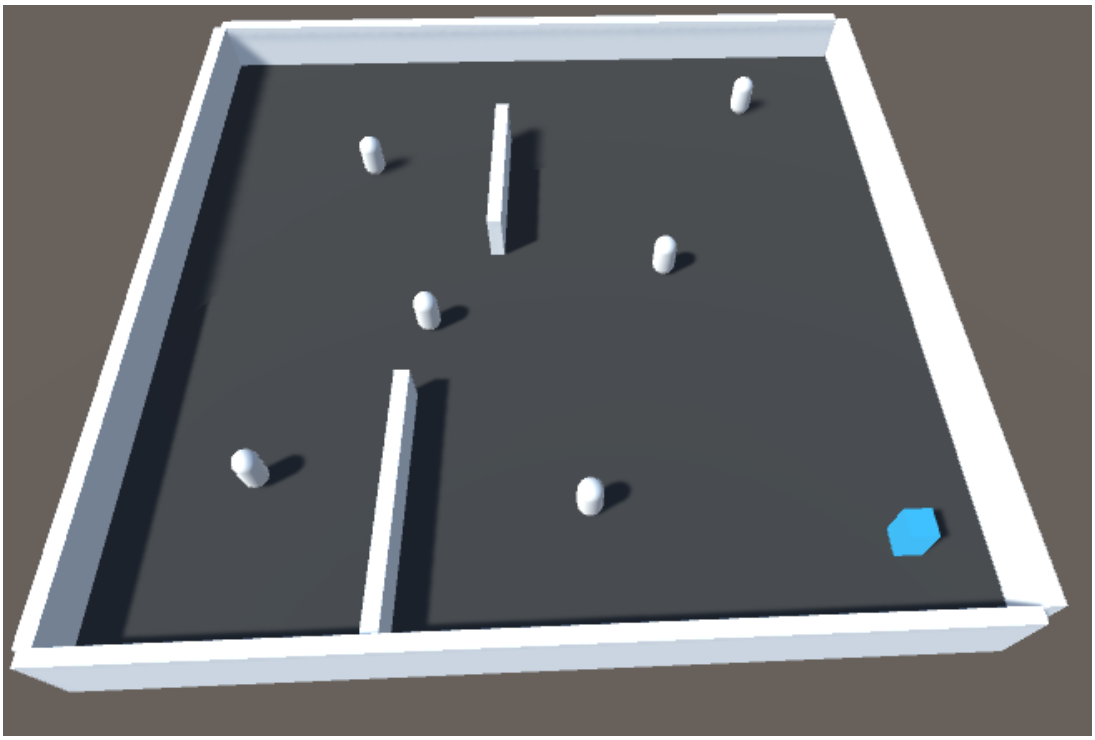

Figure 3.2: Created Environment for work

As seen Figure 3.2. Our agent placed right corner of environment blue cube and it try to destroy six white enemies in environment. If we need to explain this environment as technical way in RL our agent get -0.005 reward every time step in environment. When our agent touch the walls get -0.5 reward for every time step and get +5 point for touching enemies. With respect to RL methods our agent will try to get maximum reward. Termination state of every episode at 2750 step or destroying all enemies from environment. Then our agent observe every step its position and reward at that time step. And our agent have four continuous action left, right, forward and back. Our problem is get maximum reward with using DQN Algorithm

in of environment without using CNN (Mnih V. , et al., 2013). Also another problem for our agent is continuous action space of this environment. So we will try to find best parameters and methods when during experiments.

3.3. Experiments

3.3.1. Experiment 1

In this experiment we tried to find and compared different SGD variants for training like Momentum, AdaGrad, and Adam in our algorithm. Before start we define Maximum 1350 agents experience replay minimum 850 , 170 batch size , 100 copy period for target network, 200 episode and 2 hidden layer with 200 nodes each layer for DQN. These maximum and minimum replay founded by playing game as human player which is explained in 3.2. Than have seen agent finish to destroy all enemies almost at 1300 time step so from this respect maximum, minimum, batch and copy period defined in algorithm. First we used standard gradient optimizer for training and try 5 from 10e-1 to 10e-5 different learning rate. Figure 11 shows results of training DQN with gradient optimizer. This results shows standard gradient optimization get really bad average rewards during training after 200 episodes with different learning rate. And complete 200 episodes around one hour with each different learning rate. Also kill maximum 3 enemies during training and there is not any stabile enemy kill Figure 3.4. .

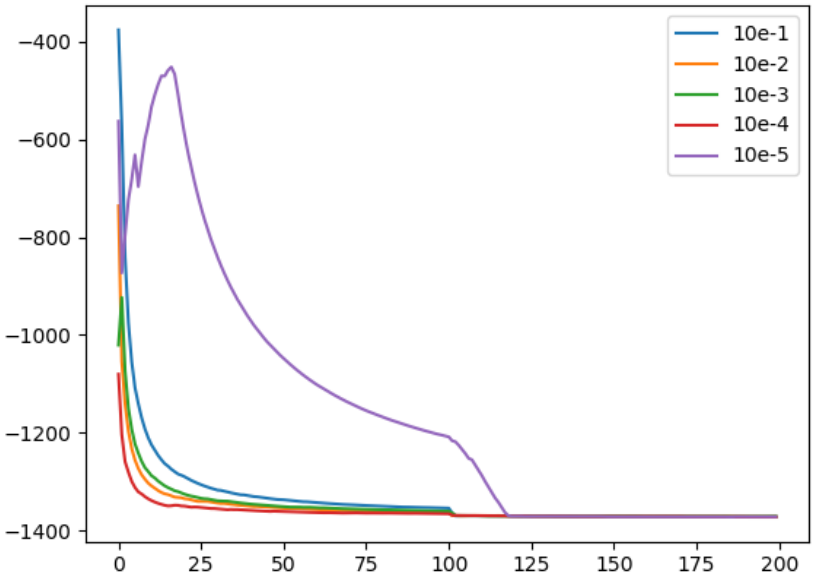

Figure 3.3: Gradient optimizer average reward results in 200 episodes

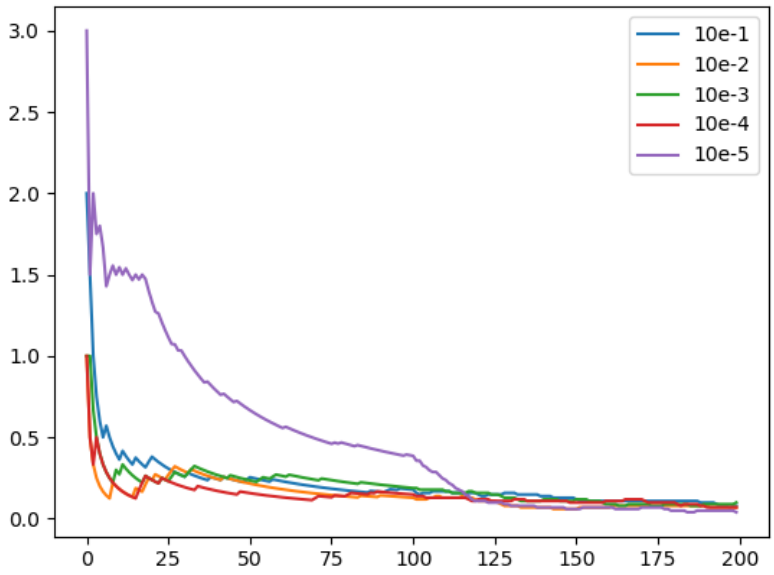

Figure 3.4: Gradient optimizer average enemy kills in 200 Episodes

Secondly we used Momentum optimizer same as gradient Figure 3.5. Show average rewards after

200 episodes. As seen figure our rewards increase with selecting 10e-5 learning rate but again this returned average rewards not good. Momentum optimizer results shows as maximum 4 enemy destroying sometimes Figure 3.6. Shows this as getting average kills in 200 episodes, during training and each training take between 40 minute to 56 minutes for 200 episodes.

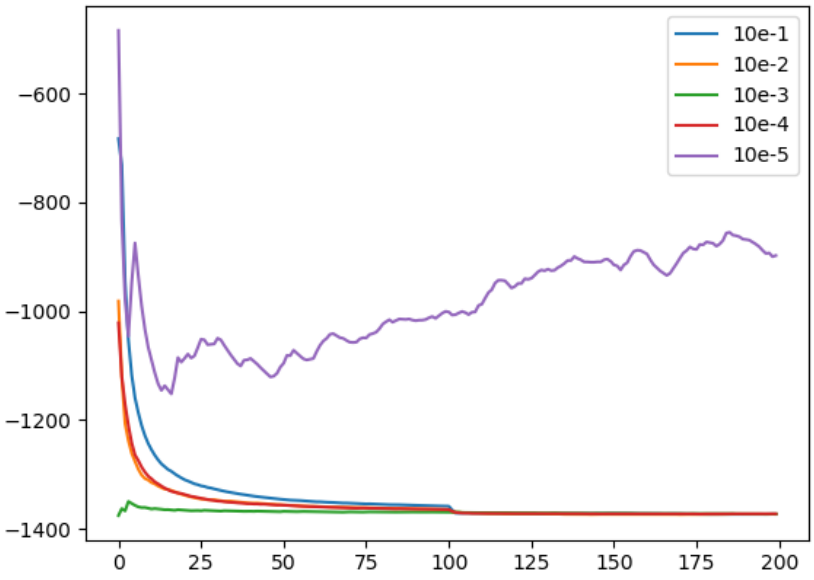

Figure 3.5: Momentum optimizer average rewards in 200 episodes

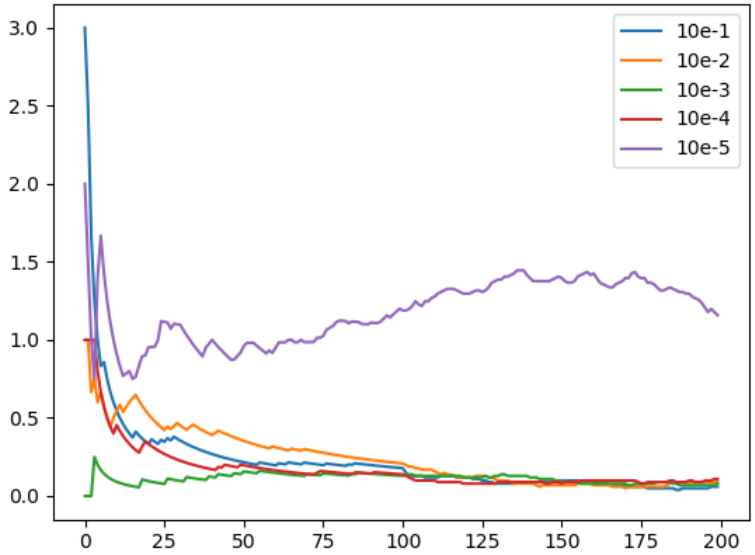

Figure 3.6: Momentum optimizer average kills in 200 episodes

Thirdly we used Adam optimizer for training same as other methods. Figure 18 shows this results in with this. As seen figure Adam optimizer show us much better results than last 2 method. After increase learning rate 10e-3 seem our agent get more reward than before. Adam optimizer training time decrease by increasing learning rate from 50 minutes to 23 minutes. But same as before agent reach really small amount destroy 4 enemy Figure 3.8. Show us average kills or destroys of enemies 200 episodes.

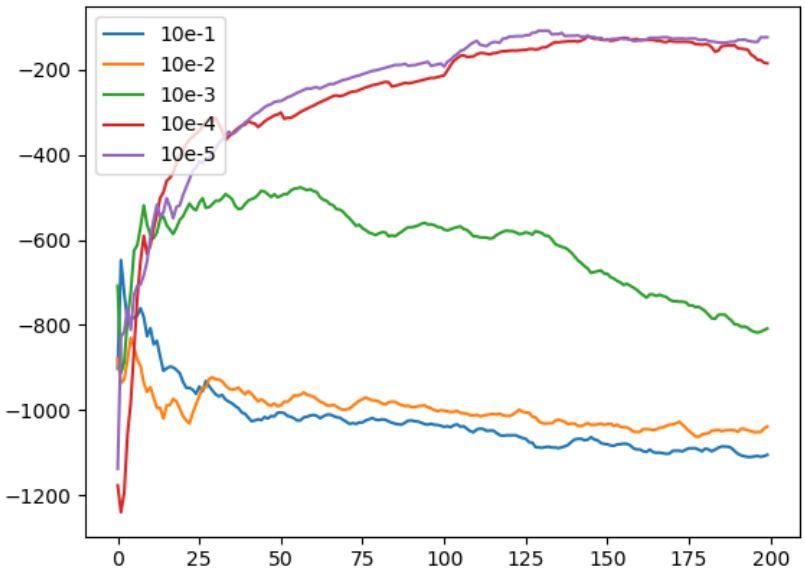

Figure 3.7: Adam optimizer average rewards in 200 episodes

And lastly we used Adagrad optimizer in algorithm for training Figure 3.9. Show us average rewards during training in 200 episodes. Adagrad optimizer show us better results than first 2 and little bit from Adam optimizer method. Adagrad calculate this result between 20 minutes to around 40 minutes and seems little bit good from others. And shows us much stabilize enemy kill then others average of enemy kill shown Figure 3.10. .

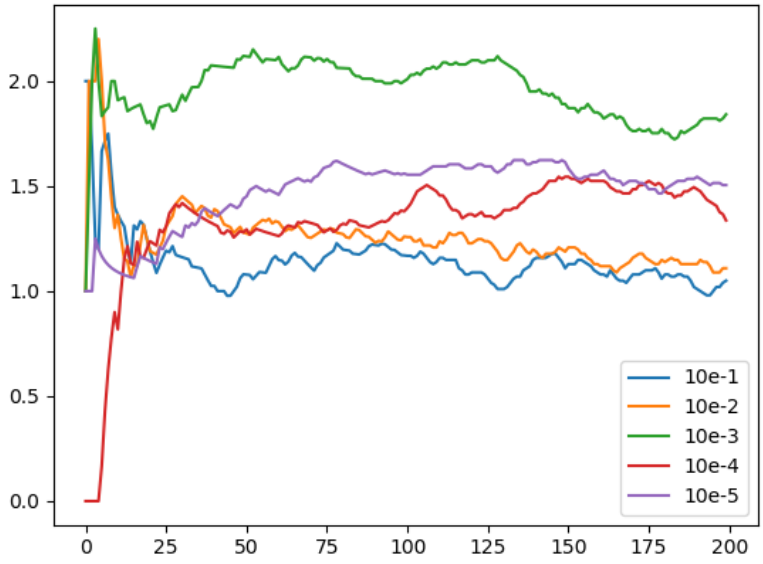

Figure 3.8: Adam optimizer average kills in 200 episodes

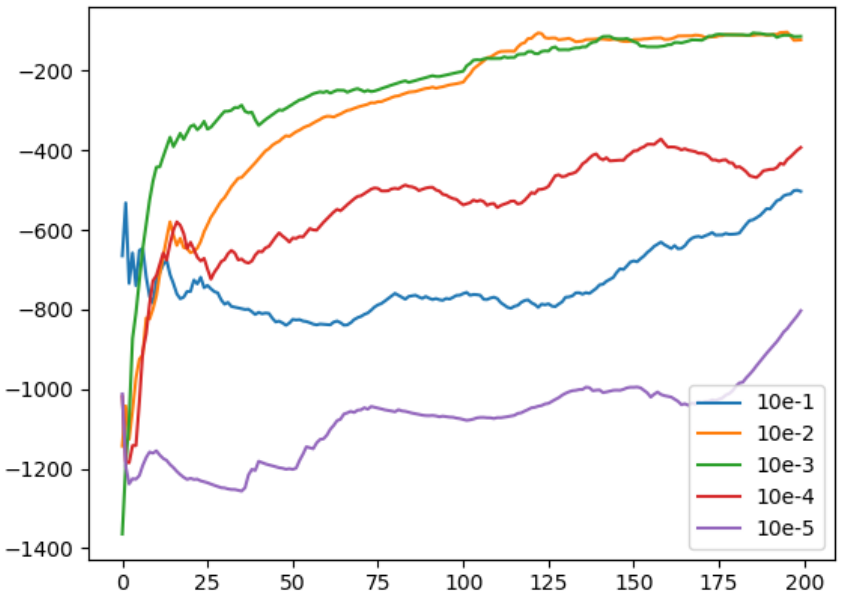

Figure 3.9: Adagrad optimizer average reward in 200 episodes

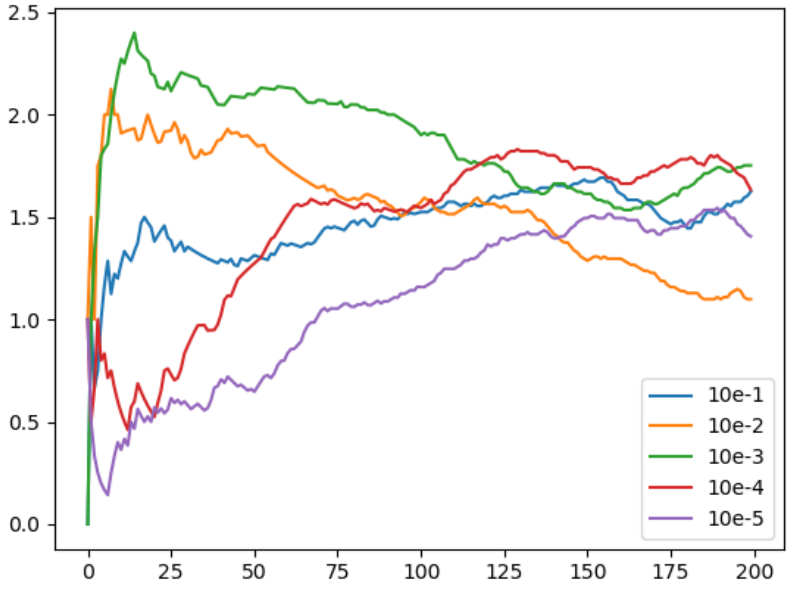

Figure 3.10: Adagrad optimizer kill average in 200 episodes

3.3.2. Experiment 2

After some experiments we saw Adagrad provide to reach faster and better average reward in 200 episodes. So move with Adagrad in experiment 2. Also last section we saw kill average are seem really bad in 200 episodes with this parameters. So for find better stabilize kill average for change this problem and increase average kill counts of enemies we increase our experiment replay as 500 thousand maximum experience, 50 thousand minimum experiment 170 mini batch size and 100 copy period for target network as seen Figure 3.11. Our agent reach better rewards after complete collecting first 500 thousand experiments in 90 episodes and its. Also as seen Figure 3.12. Much better average kills after that point same as rewards. And this is shows us increasing different parameters make huge effect on training. So with bigger experiment replay our agent get better and incremental average reward and kills. This parameters find after 9 different try so other results shown at appendix 3 for avoid complexity.

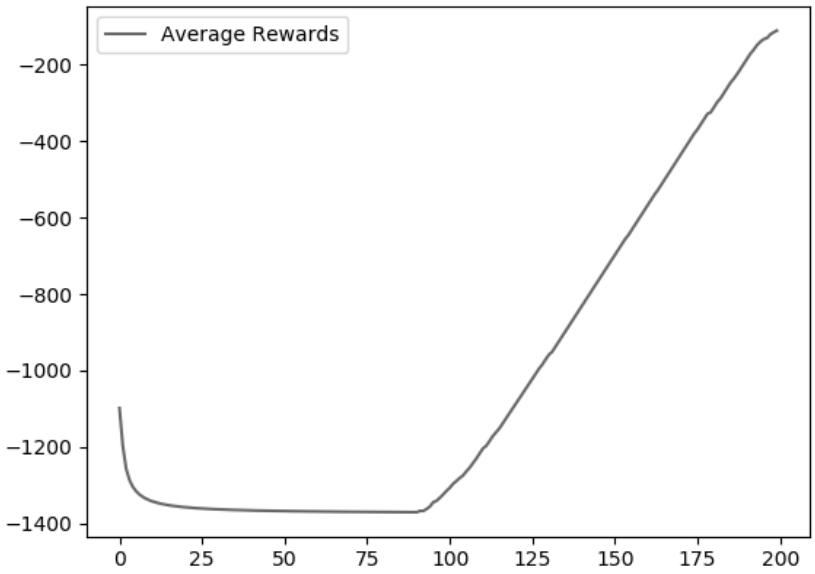

Figure 3.11: Average rewards with Adagrad optimizer changing experiment replay

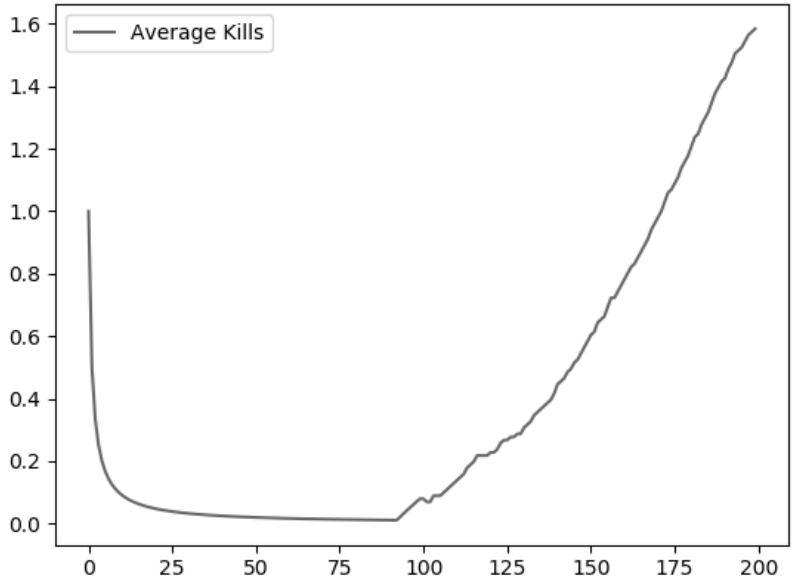

Figure 3.12: Average kills with Adagrad optimizer after changing experiment replay

CHAPTER 4

CONCLUSIONS AND FUTURE WORKS

In this study we implement reinforcement learning method q learning and deep neural network combination named as deep q network successfully 3D environment. Our agent main goal was get destroy all enemies in environment. Also our agent was getting some reward positive or negative way during searching enemies in environment like negative reward if touch wall or positive if destroy enemy. So for reach the goal tried many different parameters like different experience replay, gradient methods, learning rates. While changing this parameters collect some information to compare our results. When compare Adagrad, Momentum and Adam gradient variant methods results shows us our agent get more stabilize average rewards and enemy kills during training. So from this aspect continued with Adagrad optimizer for training our network. While getting more stabilize average reward and enemy kills with Adagrad there was a problem which is when total six enemies in environment our agent stuck to find all enemies between 2 and 4 when getting average of enemy kills from episodes. For solve this problem tried 9 different experience replay and clipping rewards like increasing reward by +5 to +500 for each enemy kill or -0.5 to -5 for touching walls but reward changes not effect this result and we get same result after training. With experience replay changes make really good changes in average enemy kills when we increase it. And our agent has achieved success.

While training we use only 3 methods with DQN when there are 4 method specified in articles. Unused method was skipping frames which is use CNN. Main reason of this, experiment computer system was not powerful enough. So we can get more successfully result with powerful experiment system using with CNN. And solve this problem with that. For

example we can change camera view to agent as first person for give agent acting like a human perspective in future works.

There are many different approaches with combine different methods like tree search, recurrent neural networks, convolutional neural network, imitation learning or etc. if we want compare our study. And many research platforms for solving problems or created artificial intelligence agents like arcade games, racing games, first-person shooters games, open-world games, real-time strategy games and more (Justesen, Bontrager, Togelius, & Risi, 2017). Also there are many open challenges in this platforms like multi-agent learning, adoption in the game industry, computational resources, creating different types of video games or etc. And also this researches with video games will occur new studies and research methods to implementation another fields like military, robotics or cinema industry for example movies created by artificial intelligence. As future plans this works can extended with convolutional neural network, tree search or supervised methods. Also maybe environment can develop like real first person game and agent may train with different methods for find better algorithms.

REFERENCES

Alpaydın, E. (. (n.d.). *Introduction to Machine Learning 2nd ed.* Massachusetts Institute of Technology.

Bellman, R. E. (1957). A Markov decision process. *Indiana University Mathematics Journal*, 679-689.

Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research*, 1-94.

Buduma, N. (2017). *Fundamentals of Deep Learning.* California: O'Reilly Media, Inc.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methords for Online Learning and Stochastic Optimization. *Machine Learning Research*, 2121-2159.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning (Adaptive Computation and Machine Learnign Series).* USA: The MIT Press.

Gu, S., Lillicrap, T., Sutskever, I., & Levine, S. (2016). Continuos Deep Q-Learning with Model-based Acceleration. *33nd International Conference on Machine Learning*, *48*, pp. 2829-2838. New York, USA.

Hinton, G., Deng, L., Yu, D., Dahl, G., Mohamed, A.-r., Jaitly, N., . . . Kingsbury, B. (2012, November). Deep Neural Networks for Acoustic Modelling in Speech Recognition. *IEEE Signal Processing Magazine*, *29*(6), 82-97.

Justesen, N., Bontrager, P., Togelius, J., & Risi, S. (2017, 12 10). *Deep Learning for Video Game Playing.* Retrieved from Cornell University Library: https://arxiv.org/abs/1708.07902

Keller, J., Liu, D., & Fogel, D. (2016). *Fundamentals of Computational Intelligence.* New Jersey, USA: John Wiley & Sons, Inc.

Ketkar, N. (2017). *Deep Learning with Python.* New York: Springer Science+Business Media.

Kingma, D., & Ba, J. L. (2015). ADAM: A Method for Stochastic Optimization. *3rd International Conference for Learning Representations.* San Diego, USA.

Kober, J., Bagnell, A., & Peters, J. (2012). Reinforcement Learning in Robotics: A survey. *Reinforcement Learning*, 579-610. doi:https://doi.org/10.1007/978-3-642-27645-3

Long-Ji, L. (1993). *Reinforcement Learning for Robots Using Neural Networks.* Pittsburg, PA, USA: Carnegie Mellon University.

Mariusz, B., Testa, D., Dworakowski, D., Firner, B., Fleep, B., Goyal, P., . . . Zieba, K. (2016, April 25). *End to End Learning for Self-Driving Cars.* Retrieved December 14, 2017, from Cornell University Library: https://arxiv.org/abs/1604.07316

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wiersta, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *Conference of Neural Information Processing Systems.* Nevada, USA.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., . . . Hassabis, D. (2015). Human-Level control through deep reinforcement learning. *Nature*, 529-533.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* New York, NY, USA: John Wiley & Sons, Inc. .

Ratcliffe, D., Devlin, S., Kruschwitz, U., & Citi, L. (2017). Cylde: A Deep Reinforcement Learning DOOM Playing Agent. *What is the Next for AI in Games Workshops at the 31st AAAI Conference on Artificial Intelligence.* San Francisco, USA.

Samarasinghe, S. (2006). *Neural Networks for Applied Science and Engineering.* New York: Auerbach Publications.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G. v., . . . Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 484-489.

Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations.* San Diego: ICLR 2015.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Reseach, 15*(1), 1929-1958.

Susson, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* Cambridge , Massachusetts , London, England: The MIT Press.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning*, (pp. 1139-1147). Atlanta, USA.

Tanikic, D., & Despotovic, V. (2012). Artificial Intelligence Techniques for Modelling of Temperature in the Metal Cutting Process. In *Metallurgy Advances in Materials and Processes* (p. Chapter 7). London: IntechOpen Limited. doi:http://dx.doi.org/10.5772/47850

Wang, Z., Schaul, T., Hessel, M., Hasselt, v., Lanctot, M., & Freitas, N. (2016). Dueling Network Architecture for Deep Reinforcement Learning. *33rd International Conference on Machine Learning*, (pp. 1995-2003).

Watkins, C. J., & Dayan, P. (1992, May). Q-Learning. *Machine Learning, 8*, 279-292.

APPENDICES

APPENDIX 1

PYTHON CODES

```
"""
@author: ahmetakin
"""

import os
import sys
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from datetime import datetime
import csv

from unityagents import UnityEnvironment

print("Python version:")
print(sys.version)

class hidden_layer:
    def __init__(self,L1,L2,f=tf.nn.tanh,use_bias=True):
        self.W=tf.Variable(tf.random_normal(shape=(L1,L2)))
        self.params=[self.W]
        self.use_bias=use_bias
        if use_bias:
            self.bias=tf.Variable(np.zeros(L2).astype(np.float32))
            self.params.append(self.bias)
        self.f=f
    def forward(self,X):
        if self.use_bias:
            a=tf.matmul(X,self.W)+self.bias
        else:
            a=tf.matmul(X,self.W)
        return self.f(a)

class DeepQNetwork:
    def
__init__(self,D,K,hiddenlayersizes,gamma,max_exp=50000,min_exp=5000,batch_s
z=750):
        self.K=K

        self.layers=[]
        L1=D
        for L2 in hiddenlayersizes:
            layer=hidden_layer(L1,L2)
            self.layers.append(layer)
            L1=L2

        layer=hidden_layer(L1,K,lambda x:x)
```

```
        self.layers.append(layer)

        self.params=[]
        for layer in self.layers:
            self.params+=layer.params

        self.X=tf.placeholder(tf.float32,shape=(None,D),name='X')
        self.G=tf.placeholder(tf.float32,shape=(None,),name='G')
        self.actions=tf.placeholder(tf.int32,shape=(None,),name='actions')

        Z=self.X
        for layer in self.layers:
            Z=layer.forward(Z)
        Y_hat=Z
        self.predict_op=Y_hat

        selected_act_values=tf.reduce_sum(
            Y_hat*tf.one_hot(self.actions,K),
            reduction_indices=[1]
            )

        cost = tf.reduce_sum(tf.square(self.G-selected_act_values))
        #self.train_o=tf.train.GradientDescentOptimizer(10e-
5).minimize(cost)
        self.train_o=tf.train.AdagradOptimizer(10e-3).minimize(cost)
        #self.train_o=tf.train.MomentumOptimizer(10e-
5,momentum=0.9).minimize(cost)
        #self.train_o=tf.train.AdamOptimizer(10e-5).minimize(cost)
        self.exp = {'s': [], 'a': [], 'r': [], 's2': [], 'done': []}
        self.max_exp=max_exp
        self.min_exp=min_exp
        self.batch_sz=batch_sz
        self.gamma=gamma

    def set_session(self,session):
        self.session=session

    def copy_from(self,other):
        ops=[]
        my_params=self.params
        other_params=other.params
        for p,q in zip(my_params,other_params):
            actual=self.session.run(q)
            op=p.assign(actual)
            ops.append(op)
        self.session.run(ops)

    def predict(self,X):
        X= np.atleast_2d(X)
        return self.session.run(self.predict_op,feed_dict={self.X: X})

    def train(self,target_n):
        if len(self.exp['s']) < self.min_exp:
```

```
            return

        idx =
np.random.choice(len(self.exp['s']),size=self.batch_sz,replace=False)
        states = [self.exp['s'][i] for i in idx]
        actions = [self.exp['a'][i] for i in idx]
        rewards = [self.exp['r'][i] for i in idx]
        next_states= [self.exp['s2'][i] for i in idx]
        dones=[self.exp['done'][i] for i in idx]
        next_Q = np.max(target_n.predict(next_states),axis=1)
        targets = [r + self.gamma * next_q if not done else r for
r,next_q,done in zip(rewards,next_Q,dones)]


self.session.run(self.train_o,feed_dict={self.X:states,self.G:targets,self.
actions:actions})

    def add_experience(self,s,a,r,s2,done):
        if len(self.exp['s']) >= self.max_exp:
          self.exp['s'].pop(0)
          self.exp['a'].pop(0)
          self.exp['r'].pop(0)
          self.exp['s2'].pop(0)
          self.exp['done'].pop(0)
        self.exp['s'].append(s)
        self.exp['a'].append(a)
        self.exp['r'].append(r)
        self.exp['s2'].append(s2)
        self.exp['done'].append(done)

    def sample_action(self,x,epsilon):
        if np.random.random() < epsilon:
            return np.random.choice(self.K)
        else:
            X = np.atleast_2d(x)
            return np.argmax(self.predict(X)[0])

def play_one(env,model,tmodel,epsilon,gamma,copy_period):
    train_mode=True
    default_brain = env.brain_names[0]
    env_info = env.reset(train_mode=train_mode)[default_brain]
    done=False
    totalreward=0
    iters=0
    kill=0
    obsv=env_info.vector_observations[0]
    while not done:
        action=model.sample_action(obsv,epsilon)
        prev_observation = obsv
        action1=int(action)
        env_info = env.step(action1)[default_brain]
        obsv= env_info.vector_observations[0]
        reward = env_info.rewards[0]
```

```
        done = env_info.local_done[0]
        totalreward +=reward
        if reward > 4:
            kill+=1
        if done:
            reward = -200

        model.add_experience(prev_observation,action1,reward,obsv,done)
        model.train(tmodel)

        iters +=1

        if iters % copy_period ==0:
            tmodel.copy_from(model)
    return totalreward ,kill

def plot_running_avg(totalrewards):
  N = len(totalrewards)
  running_avg = np.empty(N)
  for t in range(N):
    running_avg[t] = totalrewards[max(0, t-100):(t+1)].mean()
  plt.plot(running_avg)
  plt.title("Running Average")
  plt.show()

def main():
    env_name="shooter"
    env = UnityEnvironment(file_name=env_name)
    gamma=0.99
    copy_period=100

    D = 3
    K = 4
    sizes =[200,200]
    model = DeepQNetwork(D,K,sizes,gamma)
    tmodel =DeepQNetwork(D,K,sizes,gamma)
    init=tf.global_variables_initializer()
    session=tf.InteractiveSession()
    session.run(init)
    model.set_session(session)
    tmodel.set_session(session)

    N=200
    totalrewards=np.empty(N)
    epsilons=[]
    costs=np.empty(N)
    totalrewards2=[]
    kills=[]
    timetaken=[]
    startTime=datetime.now()
    for n in range(N):
        epsilon=1.0/np.sqrt(n+1)
        epsilons.append(epsilon)
```

```
        totalreward ,
kill=play_one(env,model,tmodel,epsilon,gamma,copy_period)
        kills.append(kill)
        totalrewards[n]=totalreward
        totalrewards2.append(totalreward)
        timedif=datetime.now() - startTime
        timetaken.append(str(timedif))
        print("episode:", n, "total reward:", totalreward, "eps:", epsilon,
"avg reward:", np.mean(totalrewards2),"kill count",kill, "Time taken:",
datetime.now() - startTime,"\n")

    print("Avg reward for last 100 episodes:", totalrewards[-100:].mean())

    rewardsfile = open('shooterlogs/rewards.csv','w',newline='')
    rewardsfilewrite = csv.writer(rewardsfile)
    rewardsfilewrite.writerows(map(lambda x: [x], totalrewards2))
    rewardsfile.close()

    epsilonfile = open('shooterlogs/epsilon.csv','w',newline='')
    epsilonfilewrite = csv.writer(epsilonfile)
    epsilonfilewrite.writerows(map(lambda x: [x], epsilons))
    epsilonfile.close()

    killsfile = open('shooterlogs/kills.csv','w',newline='')
    killsfilewrite = csv.writer(killsfile)
    killsfilewrite.writerows(map(lambda x: [x], kills))
    killsfile.close()

    timetakenfile = open('shooterlogs/timetaken.txt','w')
    for i in range(len(timetaken)):
        timetakenfile.write(timetaken[i]+"\n")
    timetakenfile.close()

    plot_running_avg(totalrewards)


if __name__ == '__main__':
    main()
```

APPENDIX 2

C# CODES USED FOR CREATING ENVIRONMENT

For Agent

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;


public class shooteragent : Agent {

      RayPerception rayPer;

      public GameObject bulletExit;
      public GameObject bullet;
      public float bulletForwardForce;

      public float agentRunSpeed;

      public GameObject enemyOne;
      public GameObject enemyTwo;
      public GameObject enemyThree;
      public GameObject enemyFour;
      public GameObject enemyFive;
      public GameObject enemySix;

      public Vector3 enemy3From;
      public Vector3 enemy3To;
      public float enemy3Speed;

      public Text Text;
      public Text Text1;

      Rigidbody rBody;

      void Start () {
            rBody = GetComponent<Rigidbody>();
            StartCoroutine (MoveEnemy3(enemy3From));
      }
      IEnumerator MoveEnemy3(Vector3 target){
            while (Mathf.Abs ((target -
enemyThree.transform.localPosition).x) > 0.20f) {
                  Vector3 direction = target.x == enemy3From.x ?
Vector3.left : Vector3.right;
                  enemyThree.transform.localPosition += direction *
(enemy3Speed * Time.deltaTime);
                  yield return null;
            }
```

```
            yield return new WaitForSeconds(0.2f);

            Vector3 newTarget = target.x==enemy3From.x ? enemy3To :
enemy3From;

            StartCoroutine(MoveEnemy3(newTarget));
      }
      public void MoveAgent(float[] act)
      {
            Vector3 dirToGo = Vector3.zero;
            Vector3 rotateDir = Vector3.zero;

            int action = Mathf.FloorToInt(act[0]);

            switch (action)
            {
            case 0:
                  dirToGo = transform.forward * 1f;
                  break;
            case 1:
                  dirToGo = transform.forward * -1f;
                  break;
            case 2:
                  dirToGo = transform.right * -0.75f;
                  break;
            case 3:
                  dirToGo = transform.right * 0.75f;
                  break;
            case 4:
                  fire ();
                  break;
            }
            transform.Rotate(rotateDir, Time.fixedDeltaTime * 200f);
            rBody.AddForce(dirToGo *
agentRunSpeed,ForceMode.VelocityChange);

      }

      public float detectLenwall;
      public float rewardd;
      public override void AgentAction(float[] vectorAction, string
textAction)
      {
            Text.text = string.Format ("Reward at Step {0}",GetReward());
            Text1.text = string.Format ("Step {0}",GetStepCount());
            RaycastHit hit;
            Ray wallDetectorforward = new Ray (transform.position,
transform.TransformDirection(Vector3.forward));
            Debug.DrawRay (transform.position,
transform.TransformDirection(Vector3.forward), Color.red);

            var cross = (transform.forward + transform.right).normalized;
```

```
            Ray wallDetectorforwardcross = new Ray (transform.position,
transform.TransformDirection(cross).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross), Color.red);

            var cross2 = (transform.forward - transform.right).normalized;
            Ray wallDetectorforwardcross2 = new Ray (transform.position,
transform.TransformDirection(cross2).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross2), Color.red);

            var cross3 = (-transform.forward - transform.right).normalized;
            Ray wallDetectorforwardcross3 = new Ray (transform.position,
transform.TransformDirection(cross3).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross3), Color.red);

            var cross4 = (-transform.forward + transform.right).normalized;
            Ray wallDetectorforwardcross4 = new Ray (transform.position,
transform.TransformDirection(cross4).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross4), Color.red);

            var cross5 = Quaternion.AngleAxis(22.5f, transform.up) *
transform.forward;
            Ray wallDetectorforwardcross5 = new Ray (transform.position,
transform.TransformDirection(cross5).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross5), Color.green);

            var cross6 = Quaternion.AngleAxis(67.5f, transform.up) *
transform.forward;
            Ray wallDetectorforwardcross6 = new Ray (transform.position,
transform.TransformDirection(cross6).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross6), Color.green);

            var cross7 = Quaternion.AngleAxis(112.5f, transform.up) *
transform.forward;
            Ray wallDetectorforwardcross7 = new Ray (transform.position,
transform.TransformDirection(cross7).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross7), Color.green);

            var cross8 = Quaternion.AngleAxis(157.5f, transform.up) *
transform.forward;
            Ray wallDetectorforwardcross8 = new Ray (transform.position,
transform.TransformDirection(cross8).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross8), Color.green);

            var cross9 = Quaternion.AngleAxis(202.5f, transform.up) *
transform.forward;
```

```
            Ray wallDetectorforwardcross9 = new Ray (transform.position,
transform.TransformDirection(cross9).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross9), Color.green);

            var cross10 = Quaternion.AngleAxis(247.5f, transform.up) *
transform.forward;
            Ray wallDetectorforwardcross10 = new Ray (transform.position,
transform.TransformDirection(cross10).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross10), Color.green);

            var cross11 = Quaternion.AngleAxis(292.5f, transform.up) *
transform.forward;
            Ray wallDetectorforwardcross11 = new Ray (transform.position,
transform.TransformDirection(cross11).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross11), Color.green);

            var cross12 = Quaternion.AngleAxis(337.5f, transform.up) *
transform.forward;
            Ray wallDetectorforwardcross12 = new Ray (transform.position,
transform.TransformDirection(cross12).normalized);
            Debug.DrawRay (transform.position,
transform.TransformDirection(cross12), Color.green);


            Ray wallDetectorback = new Ray (transform.position,
transform.TransformDirection(Vector3.back));
            Debug.DrawRay (transform.position,
transform.TransformDirection(Vector3.back), Color.red);

            Ray wallDetectorleft = new Ray (transform.position,
transform.TransformDirection(Vector3.left));
            Debug.DrawRay (transform.position,
transform.TransformDirection(Vector3.left), Color.red);

            Ray wallDetectorright = new Ray (transform.position,
transform.TransformDirection(Vector3.right));
            Debug.DrawRay (transform.position,
transform.TransformDirection(Vector3.right), Color.red);


            if ((Physics.Raycast (wallDetectorforwardcross8, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross7, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross6, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross12, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross11, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross10, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross9, out hit,
detectLenwall)  ) || (Physics.Raycast (wallDetectorforwardcross5, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross4, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross3, out hit,
```

```
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross2, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforwardcross, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorforward, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorback, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorleft, out hit,
detectLenwall)) || (Physics.Raycast (wallDetectorright, out hit,
detectLenwall)) ) {
                  if (hit.collider.tag == "wall") {
                        AddReward (-0.5f);//50
                  } else if (hit.collider.tag == "enemy") {
                        hit.collider.gameObject.SetActive (false);
                        AddReward (5f);
                  }

            }
            MoveAgent(vectorAction);
            AddReward(-0.00005f);
            if (!enemyOne.activeInHierarchy && !enemyTwo.activeInHierarchy
&& !enemyThree.activeInHierarchy && !enemyFour.activeInHierarchy &&
!enemyFive.activeInHierarchy && !enemySix.activeInHierarchy) {
                  Done ();
            } else {
            }
      }
      public void spawnEnemies(){

            enemyOne.SetActive(true);
            enemyTwo.SetActive(true);
            enemyThree.SetActive(true);
            enemyFour.SetActive(true);
            enemyFive.SetActive(true);
            enemySix.SetActive(true);
            enemyOne.SetActive(true);
            enemyTwo.SetActive(true);
            enemyThree.SetActive(true);
            enemyFour.SetActive(true);
            enemyFive.SetActive(true);
            enemySix.SetActive(true);
            enemyOne.SetActive(true);
            enemyTwo.SetActive(true);
            enemyThree.SetActive(true);
            enemyFour.SetActive(true);
            enemyFive.SetActive(true);
            enemySix.SetActive(true);
            enemyOne.SetActive(true);
            enemyTwo.SetActive(true);
            enemyThree.SetActive(true);
            enemyFour.SetActive(true);
            enemyFive.SetActive(true);
            enemySix.SetActive(true);
      }
      public override void AgentReset(){
            spawnEnemies();
```

```
            transform.position = new Vector3(3.994858f,0.275f,4.061983f);
      }
      public float detectLen;
      public override void CollectObservations()
      {

            AddVectorObs (transform.position);
      }

      public void fire(){
            GameObject temp_bullet;
            temp_bullet = Instantiate (bullet,
bulletExit.transform.position, bulletExit.transform.rotation) as
GameObject;
            temp_bullet.transform.Rotate (Vector3.left * 90);

            Rigidbody temp_rigidbody;
            temp_rigidbody = temp_bullet.GetComponent<Rigidbody> ();
            temp_rigidbody.AddForce (transform.forward *
bulletForwardForce);
            Destroy (temp_bullet, 2.0f);

      }
      public void OnCollisionEnter(Collision collision){
            if (collision.gameObject.tag == "wall") {
                  AddReward (-0.005f);
            }

      }

}
```

For Dynamic Enemy

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class enemy_dynamic : MonoBehaviour {

      [SerializeField] Vector3 position1;
      [SerializeField] Vector3 position2;
      [SerializeField] float speed;
      // Use this for initialization
      void Start () {
            StartCoroutine (Move(position1));
      }

      // Update is called once per frame
      void Update () {

      }
```

```
      IEnumerator Move(Vector3 target){
            while (Mathf.Abs ((target - transform.localPosition).x) >
0.20f) {
                  Vector3 direction = target.x == position1.x ?
Vector3.left : Vector3.right;
                  transform.localPosition += direction * (speed *
Time.deltaTime);

                        yield return null;
            }
            print ("reacted the target");

            yield return new WaitForSeconds(0.2f);

            Vector3 newTarget = target.x==position1.x ? position2 :
position1;

            StartCoroutine(Move(newTarget));
      }
}
```

APPENDIX 3

OTHER EXPERIEMENT RESULTS WITH ADAGRAD

Numbers shown as under figure explanation as (Maximum experience replay, Min experience replay, Mini-batch, Copy target, Learning rate, Optimization method)

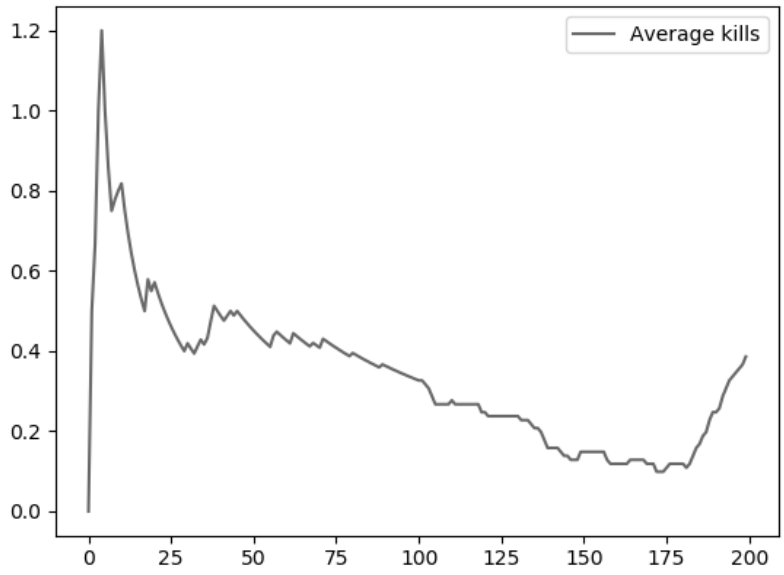

Figure A3.1: (1 Mil., 100 Thousand, 170, 100, 10e-3, Adagrad)

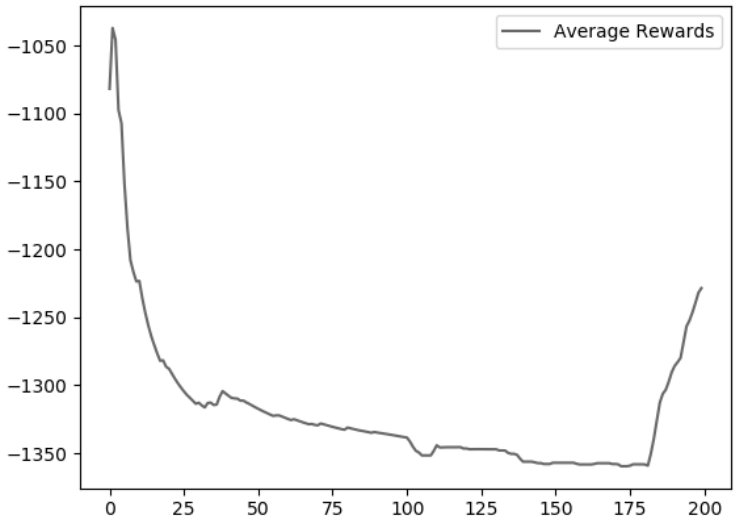

Figure A3.2: (1 Mil., 100Thousand, 170, 100, 10e-3, Adagrad)

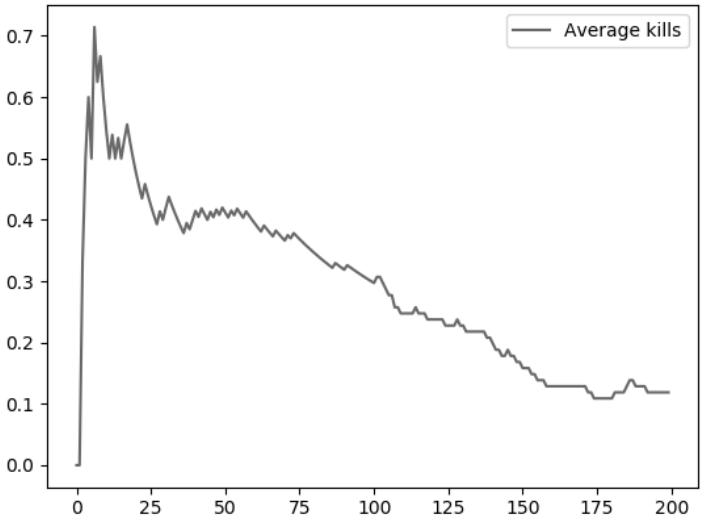

Figure A3.3: (2 Mil., 250 Thousand, 170, 100, 10e-3, Adagrad)

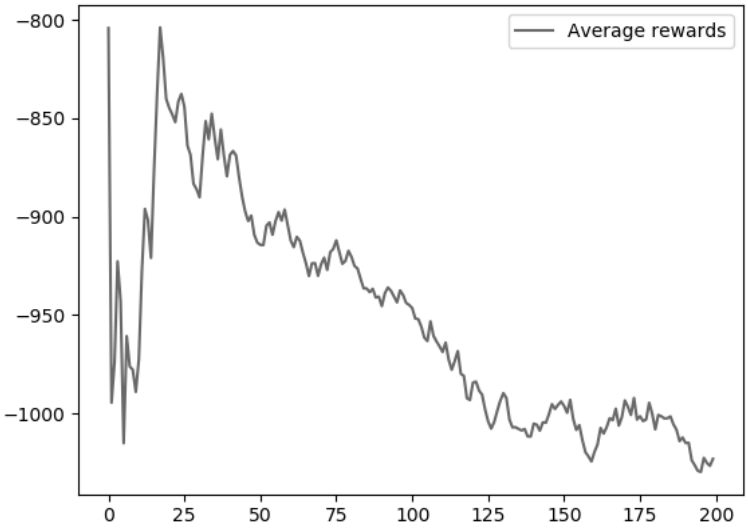

Figure A3.4: (2Mil, 250 Thousand, 170, 100, 10e-3, Adagrad)

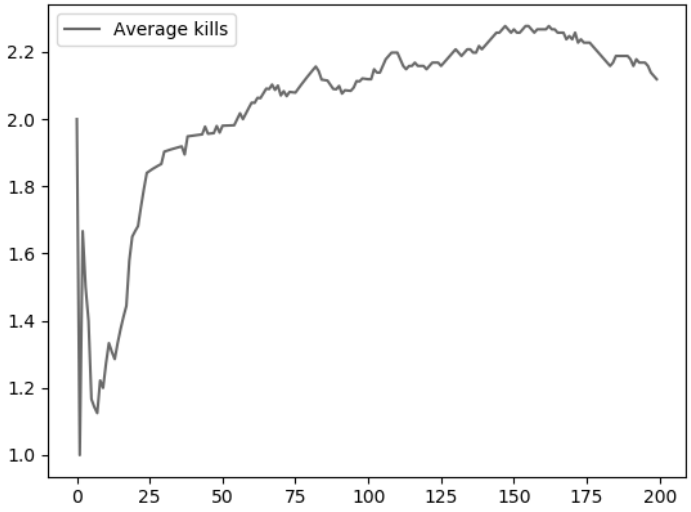

Figure A3.5: (25 Thousand, 2.5 Thousand, 170, 100, 10e-3, Adagrad)

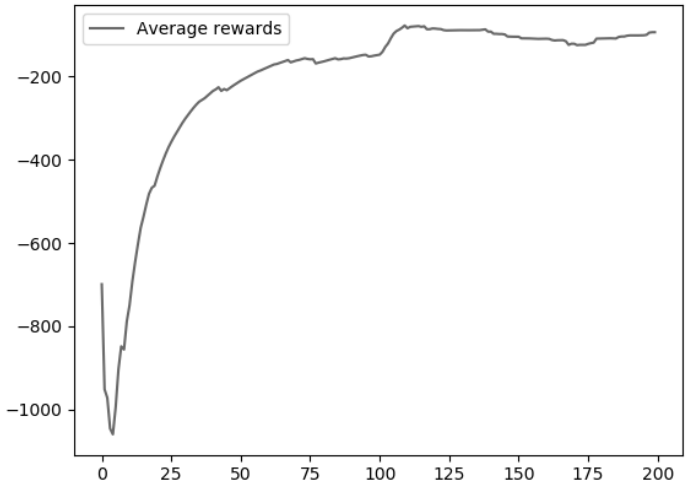

Figure A3.6: (25 Thousand, 2.5 Thousand, 170, 100, 10e-3, Adagrad)

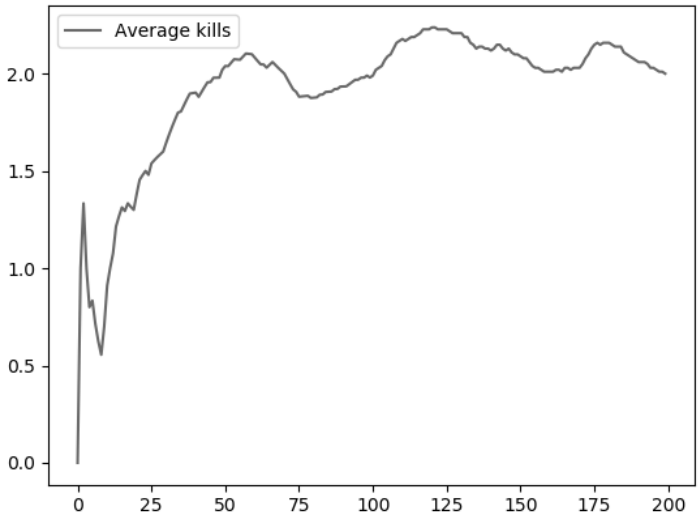

Figure A3.7: (50 Thousand, 5 Thousand, 170, 100, 10e-3, Adagrad)

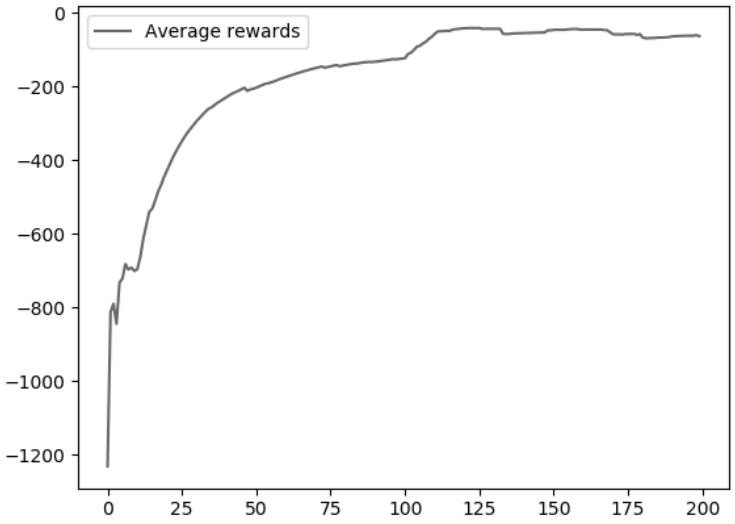

Figure A3.8: (50 Thousand, 5 Thousand, 170, 100, 10e-3, Adagrad)

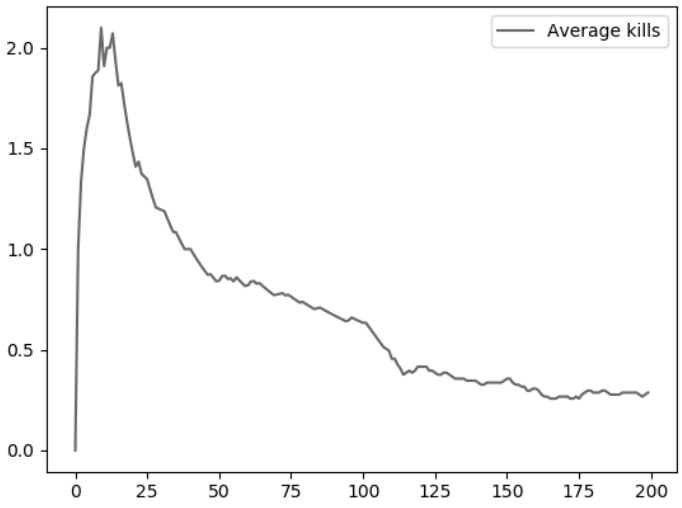

Figure A3.9: (50 Thousand, 5 Thousand, 7.5 Thousand, 100, 10e-3, Adagrad)

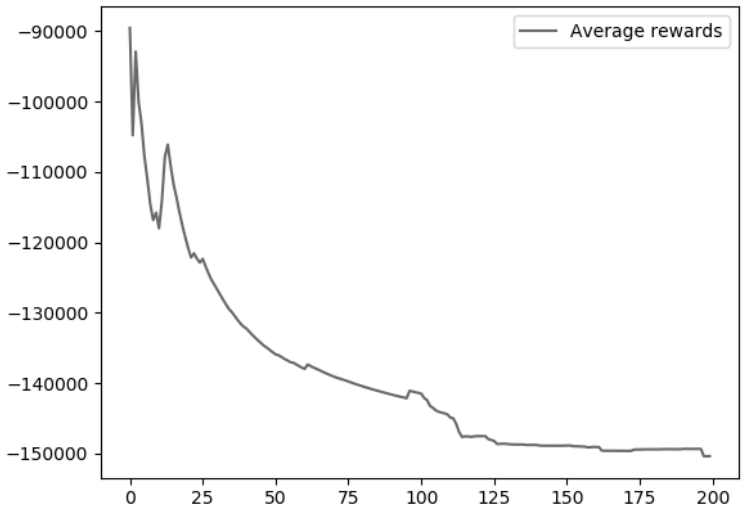

Figure A3.10: (50 Thousand, 5 Thousand, 7.5 Thousand, 100, 10e-3, Adagrad)

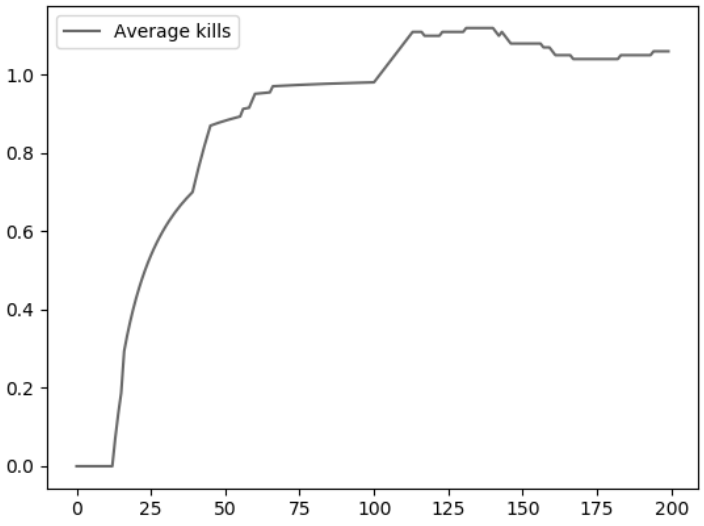

Figure A3.11: (50 Thousand, 5 Thousand, 7.5 Thousand, 100, 10e-3, Adagrad)

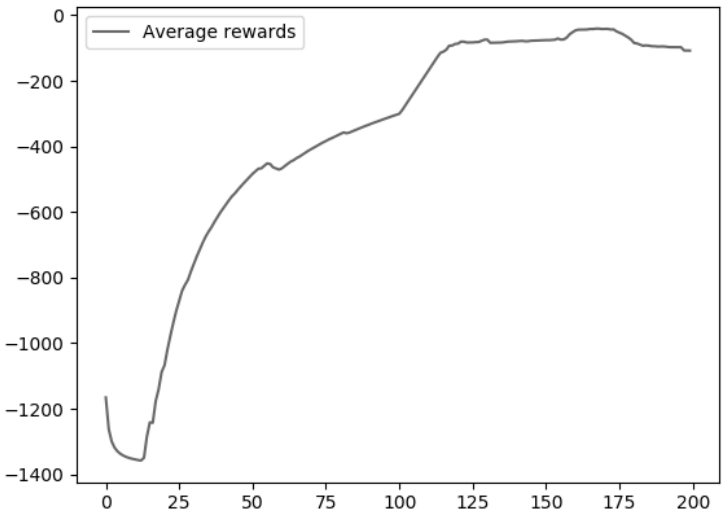

Figure A3.12: (75 Thousand, 7.5 Thousand, 170, 100, 10e-3,Adagrad)

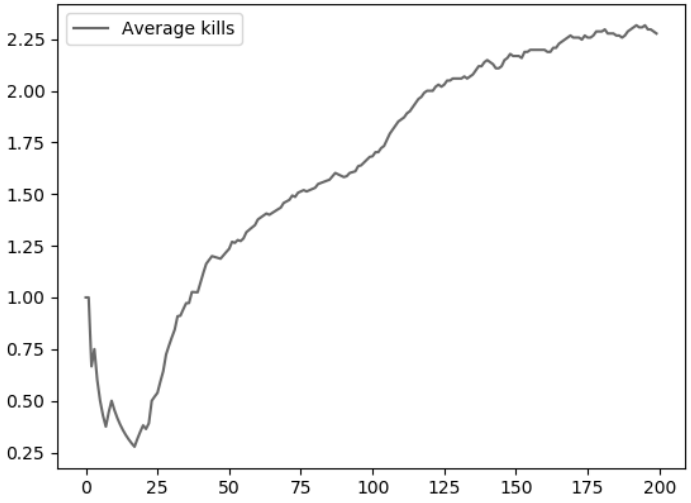

Figure A3.13: (75 Thousand, 7.5 Thousand, 170, 100, 10e-3,Adagrad)

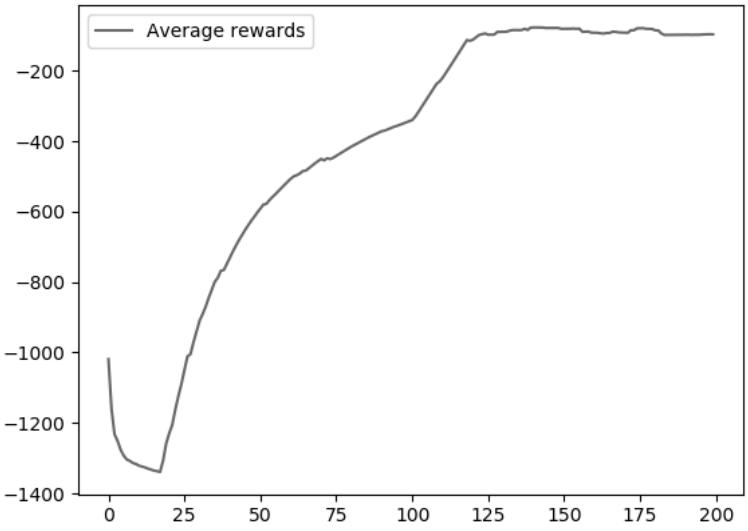

Figure A3.14: (100 Thousand, 10 Thousand, 170,100,10e-3,Adagrad)

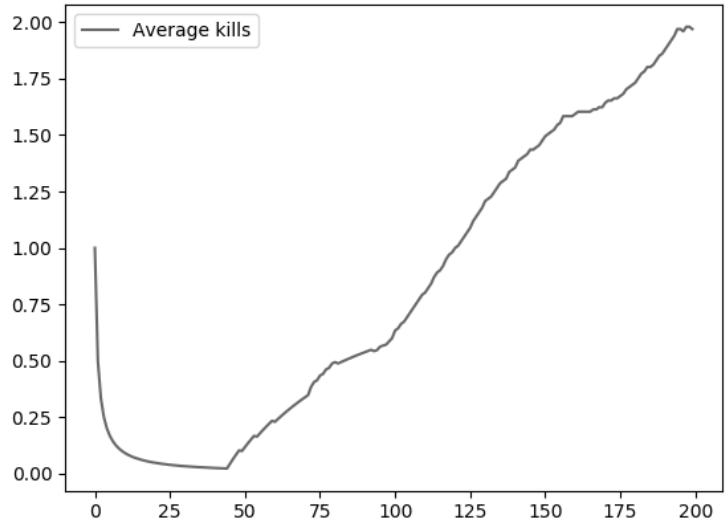

Figure A3.15: (250 Thousand, 25 Thousand, 170, 100, 10e-3, Adagrad)

.

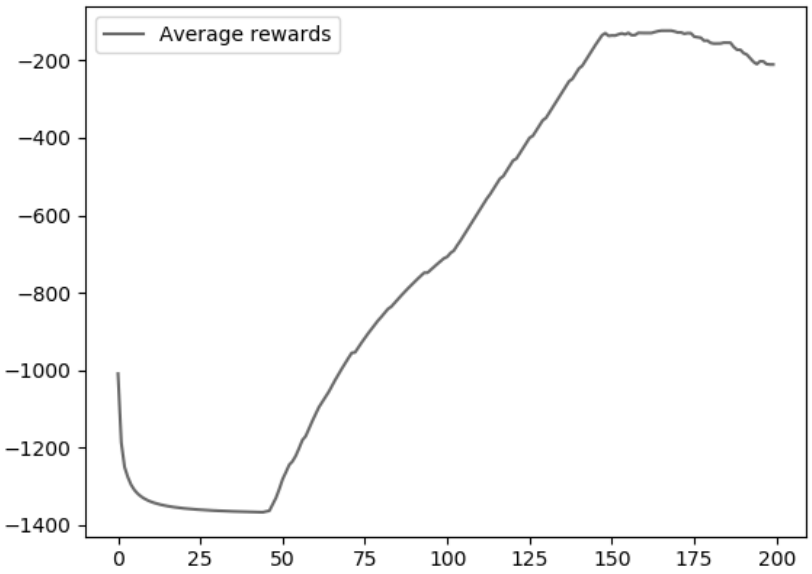

Figure A3.16: (250 Thousand, 25 Thousand, 170, 100, 10e-3, Adagrad)

IMPLEMENTATION OF DEEP Q-LEARNING TO THE 3D CONTIN...

INBOX | NOW VIEWING: NEW PAPERS ▾

Submit File

Online Grading Report | Edit assignment settings | Email non-submitters

AUTHOR	TITLE	SIMILARITY	GRADE	RESPONSE	FILE	PAPER ID	DATE
Ahmet Akın	Abstract	0%	--	--		973227303	07-Jun-2018
Ahmet Akın	Chapter1-Introduction	0%	--	--		973227452	07-Jun-2018
Ahmet Akın	Chapter4 - Conclusions	0%	--	--		973228037	07-Jun-2018
Ahmet Akın	Chapter3	1%	--	--		973227792	07-Jun-2018
Ahmet Akın	All Thesis	7%	--	--		973228339	07-Jun-2018
Ahmet Akın	Chapter2	12%	--	--		973227629	07-Jun-2018