The Use of Reinforcement Learning in Gaming: The Breakout Game Case Study

Taresh Dewan
COMP5112WC Student
Lakehead University
tdewan@lakeheadu.ca

Manva Trivedi
COMP5112WC Student
Lakehead University
trivedim@lakeheadu.ca

Sabah Mohammed
COMP5112WC Supervisor
Lakehead University
mohammed@lakeheadu.ca

Aloukik Aditya
COMP5112WC Student
Lakehead University
aaditya@lakeheadu.ca

Ao Chen
COMP5112WC Student
Lakehead University
achen11@lakededu.ca

Danning Jiang
COMP5112WC Student
Lakehead University
djiang3@lakeheadu.ca

***Abstract*—Traditionally, reinforcement learning (RL) algorithms are called trial and error learning methods that use real task experience to develop an incremental management policy. The reinforcement learning theory offers a viewpoint in psychology, how agents can maximize their control of an environment. The major difference of reinforcement learning from supervised learning is that a partial feedback is provided to the learner, regarding the learned experiences. An RL agent learns how to map states to optimal action through trial-and-error and over time practices and develops a strategy for long-term rewards. In this paper, we are using an approach which unifies artificial neural networks and reinforcement learning architecture allowing the agent to learn the best possible actions in a virtual environment to achieve their objectives for which we have chosen Breakout – a classic arcade game. We have chosen Breakout as it achieves superhuman play as compared to other games such as Enduro, Time Pilot etc. This paper provides a comparative analysis between Deep Q Network (DQN) and Double Deep Q Network (DDQN) algorithms based on their hit rate, out of which DDQN proved to be better for Breakout game. DQN is chosen over Basic Q learning because it understands policy learning using its neural network which is good for complex environment and DDQN is chosen as it solves overestimation problem (agent always choses non-optimal action for any state just because it has maximum Q-value) occurring in basic Q-learning.**

***Index Terms*—Reinforcement learning (RL), Deep Q Network (DQN), Double Deep Q Network (DDQN), arcade games, Breakout, Atari, agent, action, state, environment, Q-value, rewards**

I. Introduction

A. Overview

Reinforcement learning (RL) refers to goal-oriented algorithms that learn how to accomplish a specific goal or how to optimise over many steps along a dimension; for example, over many moves, they can maximise the points earned in a game. RL algorithms can start with a blank state and achieve superhuman performance under the right conditions. Such algorithms are penalised, like a pet incentivized through scolding and punishment, when they make the wrong decisions and are praised when they make the right ones – this is reinforcement. Performance of RL degrades when the state-action space is too large to be completely known. For this reason, we are using Deep Reinforcement Learning to achieve better performance.

Deep reinforcement learning integrates artificial neural networks with an architecture of RL that allows software-defined agents to learn the best possible behaviours in a virtual environment to achieve their objectives. Instead of using a lookup table to store, index and update all possible states and their values, which is difficult with very big problems, we have trained a neural network on state-action space samples to learn to determine how important those are relative to our aim of enhancing learning.

B. Basic Definitions

Reinforcement Learning can be understood if we know the concepts of agent, state, action, environment and rewards, that is explained below:

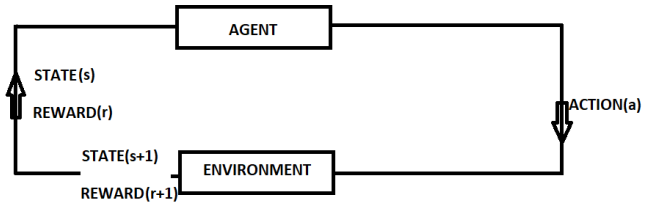

Fig. 1. Reinforcement Learning Process

As shown in figure 1, agent and environment are the key components in reinforcement learning process. ***Agent*** is an entity that takes ***action*** (a set of moves which the agent can make) in an ***environment*** (surroundings the agent is going through and which responds to the agent). A ***state*** is a real and immediate condition in which the agent finds himself; that is, a position or a moment on the basis of which its Q-value is

updated and ***reward*** (input by which we measure the success or failure of an agent's action in each state) is assigned to the agent which can be negative or positive which in turn impacts the ***Q-value*** (Q-value takes the input of two parameters: state and current action a. $Q\pi$(s, a) refers to the long-term return of an action taking action from the current state s under policy π. Q maps the pairs of state action to rewards).

An agent sends feedback in the form of actions to the environment from any given state, and the environment returns the new state of the agent (which resulted from acting on the prior state) as well as rewards if any. Rewards can be staggered or immediate. Effectively they determine the behaviour of the handler. Another term associated with rewards is ***discount factor*** which can be computed by future rewards as discovered by the agent to dampen the effect of that reward on the choice of action by the agent.

In our customised breakout game environment, paddle plays the role of an agent and environment includes wall, bricks and ball. Actions which can be taken by the agent (paddle) are moving the paddle to the left, moving the paddle to the right or let it stay in idle position. States of the game will be, whether the game is ongoing / lost / won, the x and y coordinates of the ball, ball-velocity, the x position of the paddle, array of the coordinates of the remaining bricks, the number of frames since the game started, and the current score of the game.

Our customised breakout environment consists of a paddle, a ball, a wall and a block which consists of bricks with different colors. When a ball hits the paddle, 3 points are rewarded to the agent and if it misses the paddle, 3 points are penalised to the agent. When a ball hits the brick, rewards are awarded to the agent as per the color of the brick (blue=8, green=7, olive=6, yellow=5, orange=4, red=3). Q-value of each (state, action) pair will be fed to the network which will be helpful for learning. After achieving Q-values for each distinct (state, action) pair which will be stored in the network, the action with the best Q-value will be chosen for the progression of the game.

One of the reasons to choose breakout is that for the start, it is better to choose a game that can be altered as per the user and for which different parts of the algorithms like states, reward system can be modified into a better one. In this paper, we have given a comparative analysis of two different algorithms on a classic Atari arcade game - Breakout. We have compared and evaluated the performances of different models namely Conventional Deep Q-Network and Double Deep Q-Network.

C. Asserting Thesis

The "memory" is a key component of DQNs: the trials are used to train the model continuously, as stated earlier. Instead of training on the trials when they get in, however, we add them to memory and train them on a random sample of that memory. The gamma factor reflects this depreciated value for the expected future returns on the state. Value defined for but will vary between 0 and 1. We have followed Epsilon Greedy Search in which value of gamma is 1 in initial stages as the behavior of the paddle will be completely random and it will start decreasing with the ongoing iterations with a value 0.95. For each distinct (state, action) pair, it's Q-value (which is calculated by DQN and DDQN algorithms) will be updated, and that Q-value will be used to take best possible action of the paddle for the next state (velocity, ball co-ordinates).

We chose to implement Deep Reinforcement Learning on Atari games because the environment of Atari games is quite uncertain taking into consideration the states and actions related to the environment which makes it relatable to real life situations.

II. Problem Definition

The purpose of our project is to create virtual environment of a game named "Breakout" that can prove out to be a close replication of the real-life environment, learn the environment with time as it happens in real life and act accordingly. To duplicate a real-life environment and to act as per the changes, the environment has to be learnt properly, which we are going to do so, using the algorithms of Reinforcement Learning mentioned above and showing comparisons which is better for such situations.

First, Reinforcement learning is the problem that we have studied. Reinforcement learning is a branch of machine learning, which is used to describe and solve the problem of agent's interaction with the environment through learning strategies to achieve maximum reward [1]. A classic and standard model of reinforcement learning is the Markov Decision Process (MDP), which is simply a process where an agent takes action to update its state to obtain reward and interact with environment [2].

Second problem being the game itself. We have imitated Breakout as our game. The building blocks of this game are a moving ball, a paddle and 6 rows of bricks. The agent i.e. paddle can move left and right to hit the ball. After hitting the ball, the ball will rebound and then destroy the bricks to achieve reward. This game needs to be able to receive incoming actions and switch to the next frame. If the ball is missed, the game will end and then reset the position of the ball and paddle in the game. The bricks will disappear after being touched by the ball. At the beginning of the game, the ball and paddle are placed in the initial place and all 6 rows of bricks are loaded. The game gives the total number of bricks destroyed, the number of times the ball was missed, and the ball's hit rate.

Third problem is to create a network structure for both DQN and DDQN in our simulated Breakout environment in terms of reward acquisition, loss at each epoch and hit rate to conclude which algorithm performs better.

III. Related Research Work

Reinforcement learning can go back to the implementation of TD-gammon, In 1992, IBM researcher Gerald Tesauro developed an algorithm that combines time difference learning and neural networks, and named it TD-Gammon, specializing in playing backgammon. TD-gammon uses a three-layer neural

network. The backgammon position is represented by 198 units as the input, and there are 40-80 neurons in the middle-hidden layer. The final output is an estimate of the value function [3]. TD-gammon used a model-free reinforcement learning algorithm like Q-learning and approximated the value function using a multi-layer perceptron with one hidden layer.

However, the applications of TD-gammon into other board games were less successful, which led to a widespread belief that the TD-gammon method only worked in backgammon. This perhaps because the randomness in the dice rolls helps explore the state space and also makes the value function particularly smooth [4].

Q learning is proposed by Watkins in 1989, which has become the popular option for reinforcement learning-based agents, however it is useless for the complicated and high state space problem.

The combination of deep learning and reinforcement learning methods, mainly involving Q learning, was brought forward in a sequence of papers [5]. From what we know that previous reinforcement learning methods had trouble in selecting features, while the deep reinforcement learning approach was found to handle complex tasks successfully, as it can learn from data at different levels of features. Mnih successfully trained a deep RL agent from visual inputs consisting of thousands of pixels. This approach enabled it to reach beyond-human capabilities in playing Atari games, Alpha Go and so on [6]. The Deep Q network agent synthesized by Mnih achieves human-like performance when playing Atari games by using artificial neural networks to process sensory data. In subsequent work, Van Hassel [7] improved the algorithm by implementing double deep Q-Learning which helps generate more accurate estimates by eliminating overestimation.

In the paper "Human-level control through deep reinforcement learning" [5], researchers have shown that the deep Q-network agent, obtaining only the pixels and the game score as inputs, has been able to exceed the efficiency of all previous algorithms and reach a level equal to that of a skilled human games tester using the same algorithm, network architecture and hyperparameters across a range of 49 games [5].

Paper [8], shows comparisons between RELU Neural Network and Spiking Neural Network in terms of rewards achieved for given number of epochs in breakout, using Epsilon greedy approach and conventional greedy approach. Furthermore, with additional benefits of SNNs can supplement the working of DQN when data is noisy and incomplete [8]. Paper [9], shows comparisons of performances in terms of training time, stability and higher score achieved using DQN and Asynchronous Advantage Actor-Critic (A3C) algorithms that too in breakout game. Rewards achieved using A3C algorithm are higher as compared to DQN as A3C uses a multi-core power CPU to work efficiently whereas, DQN needs a powerful GPU to train faster and runs slowly on a CPU [9]. Paper [10], shows a visual DQN approach which helps to control the random actions of DQN and helps domain experts to understand, diagnose, and improve DQN models with four levels of details: overall training level, epoch-level, episode-level, and segment-level. Basic overview of performance results of different algorithms along with their criteria in breakout game is briefly given in Table I.

TABLE I
OVERVIEW OF RELATED RESEARCH WORK IN BREAKOUT

SNo	Criteria for Comparison	Algorithms Compared	Results
1.	Rewards for given number of epochs	SNN | DQN	SNN is better especially for noisy or incomplete data.
2.	High Score, Stability and rewards	DQN | ARC	ARC outperformed DQN with a high score of 79 and rewards increasing gradually.
3.	Epoch level, training level, episode level and segment level	Visual DQN | DQN	Visual DQN is able to control the random actions taken by DQN algorithm.

We have shown performance comparisons for DQN and DDQN algorithms for the training phase, in terms of ***hit rate*** which is calculated by taking the ratio of number of times the ball hits the paddle to the sum of number of times ball hits the paddle and number of times ball misses the paddle, for the given number of episodes and which in our case is 100.

IV. METHODOLOGY

In order to compare the performance of the AI agent in playing the breakout game based on different algorithms, we decided to create our own environment to train the agent, which imitates the environment of the OpenAI Atari game Breakout-ram-v0. After setting up the environment, we built the network for DQN and Double DQN algorithms respectively. The specific steps are as follows:

1) Setting up the environment-Breakout, including the background, the paddle and ball, defining bricks, controlling paddle movement, handling collisions, updating state and environment, using turtle library which is a graphics library in python that can be used to create various objects and shapes, provide animations to them using penup() function, by adjusting the speed during the process of object creation.

2) For defining bricks, controlling paddle movement, handling collisions, updating state and environment, we have defined separate functions namely reset() which will reset the environment if the paddle misses the ball, next_iteration() which will compute the parameters for next state, move_positive_x() will move the paddle to the right and move_negative_x() which will move the paddle to the left.

3) Creating DQN and Double DQN algorithm using the Breakout environment and perform Hyper-parameter tuning (like discount factor gamma, learning rate, epsilon). Using libraries like random, numpy, keras, collections, matplotlib and so on.

4) Training the Agent using environment, within the 1000 steps, calculating reward and loss for each 100 episodes, then save the reward and data loss and plot them.

A. Deep Q Network

In traditional reinforcement learning like Q learning, we use table to store Q value. But it has a limitation. The problem today is too complicated to use tables to store the Q values of each state and action. No matter how much memory the computer has, it becomes time consuming to search for the corresponding state in such a large table. When reinforcement learning is combined with deep learning, neural network can solve this problem. Because we can just input the state value, output all the Q values of each action, and then directly select the action with the maximum value as the next action according to the principle of Q learning.

Table II describes the structure of Deep Q Network we use in our project.

TABLE II
MODEL STRUCTURE

Layer (type)	Output Shape	Activation function	Param #
dense_1 (Dense)	(None, 64)	RELU	384
dense_2 (Dense)	(None, 64)	RELU	4160
dense_3 (Dense)	(None, 3)	Linear	195

The input of this network is a state, the output is the Q values of 3 actions. Neural network is what we use to process the state and predict Q value. We also need to update the Q value and train the network. Equation (1) below describes how Deep Q Network update Q value.

$$Q_{new} = R + \gamma maxQ(S, a) \tag{1}$$

R means the current reward, γ means parameter gamma , max Q(S, a) means the action a with maximal Q value in the state S, Q_{new} means the updated Q value. Then we can use state S as input, updated Q value as output to train the network.

B. Double DQN

The network structure of Double DQN is the same as Deep Q Network. But Double DQN requires 2 networks. The one is the main network which also has updated parameters, the other one is target network which has old parameters.

In Double DQN, we use main network to get the Q value using state as input. The equation (1) to update Q value has changed to equation (2):

$$Q_{new} = R + \gamma Q(S, argmaxQ(S, a; \theta_{main}), \theta_{target}) \tag{2}$$

R means the current reward, γ means parameter gamma , Q_{new} means the updated Q value, S means the input state, θ_{main} means using the main network, θ_{target} means using the target network. In equation (2), we use state S as input and main network to select the action a which has the largest Q value. Then use target network to get the Q value of the selected action a. Then we can update the Q value and use updated Q value and input state S to train the network.

V. PROTOTYPING

A. Detailed Design:

In our project, we have used three classes to implement the code (Paddle, DQN and DDQN) as shown in figure 2.

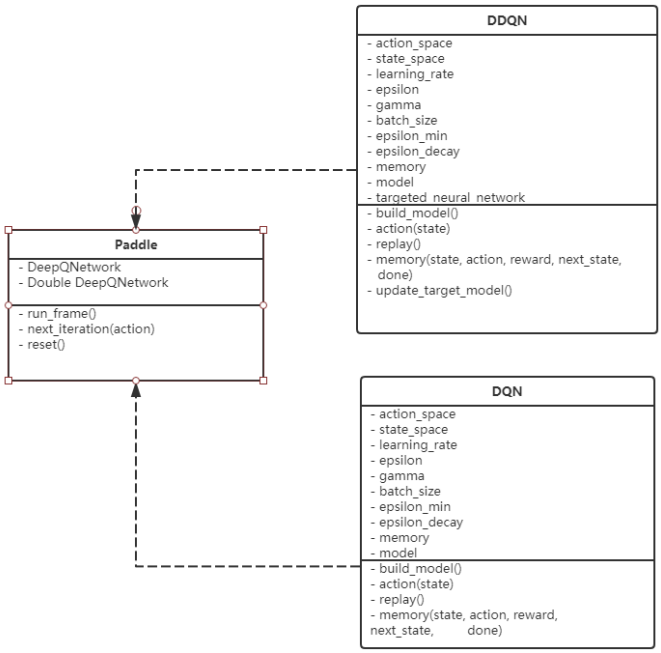

Fig. 2. Class Diagram

B. Building the major classes:

As shown in figure 3, using the import turtle we are adding the turtle library into our python environment. The turtle module provides turtle graphics primitives, in both object-oriented and procedure-oriented ways.

```
import turtle as t
```

Fig. 3. Importing turtle

As shown in figure 4, we are adding the bricks using for loop. We have assigned a variety of colors to the bricks. Using x_cor and y_cor, we have assigned the coordinates of the bricks, and each Brick is added after 110 pixels on the x-axis.

```
bricks_colour=["red","orange","yellow","olive","green","blue"]
count=1
dictionary_bricks={}
list_id=[]
for colour in bricks_colour:
    x_cor=-340
    y_cor=280-bricks_colour.index(colour)*25
    for j in range(0,7):
        count=Brick("square", colour, x_cor, y_cor)
        x_cor=x_cor+110
        if colour not in dictionary_bricks.keys():
            dictionary_bricks[colour]=[]
        else:

            dictionary_bricks[colour].append(count)
```

Fig. 4. Building Bricks

We have used Brick () function to provide its dimensions, shape, colors, and coordinates.

As shown in figure 5, we then initialize the configuration of the ball and paddle. Talking about the ball, we have decided to take its shape as a circle and color to be red. dx and dy are the coordinates of the ball, and the ball will be moving so the values of its coordinates change accordingly. Using setpostion() function, we set the initial position of the ball.

```
# Number of hits successfuly made by the ball width he paddle
self.ball_hit=0

#creating paddle and ball for the game as per the parameters
self.paddle=ttl.Turtle()
self.ball=ttl.Trutle()
self.paddle.color("white")
self.paddle.shape("square")
self.ball.color("red")
self.ball.shape("circle")
self.padddle.penup()
self.paddle.turtlesize(stretch_wid=1,stretch_len=6)
self.ball.penup()
self.ball.speed(0)
self.paddle.speed(0)
self.paddle.setposition(0,-290)
self.ball.setposition(0,0)

# will be used to calculate the number of tuime th ball is missed i.e. it either hits the ground
self.ball_miss=0

#defining the rate of change of posititon that is the velocity in x as well as y direction
self.ball.dx=3
self.ball.dy=-3
```

Fig. 5. Building Paddle and Ball

The paddle has some similar configuration to the ball, but its movement is on the x-axis, and its shape is square. Using shapesize() we stretched the square length and created a rectangle. We decided to select its color as white; it is more visible on black background.

Figure 6 shows how the final display screen is created. Bgcolor() function selects the background color of the entire game, which is black in our case. The screen resolution is decided using win.setup() function, and we chose our screen size to be 800 by 600.

In Paddle class, it has 3 functions:

1) Reset: The returned value of the function will be X value of the paddle, X value of the ball, Y value of the ball. This function is to reset the position of the paddle and the ball in the game. After this function is called, the paddle will return to the mid of the screen and the ball will go back to the initial position and begins to fall.

```
#Creating the basic window which will be behave as the display of the game
self.window=ttl.Screen()

self.window.bgcolor("#000000")
#specifying the width of the window
width_window=600
#specifying the height of the window
height_window=600
#creating the window as per the dimensions mentioned
self.window.screensize(width_window,height_window)
self.window.tracer(0)
```

Fig. 6. To create main screen

2) Run_frame: This function is used to make the ball move then check if any brick is touched by the ball and if the ball is missed. If the brick is touched by the ball, the ball will bounce and the brick will disappear. And the game will record this ball hit and calculate the hit rate. If the ball is missed, the game will call the reset function and record the number of missed ball.
3) Next iteration: This function requires action as an input and the input parameter action is an integer type. It represents which action to take. In our game, we totally have 3 actions. The range of action is from 0 to 2. 0 means paddle should move left; 1 means paddle should do nothing; 2 means paddle should move right. After the game receives the input action, the game will call the run_frame function to move to next frame.

In DQN class, parameters that have been used are:

- action_space: It is an integer type; it represents how many actions we have in the game. In our game, we totally have 3 actions, which means paddle should move left, do nothing and move right.
- state_space: It is an integer type; it represents the dimension of the state.
- Epsilon: It is a float. When we choose an action, we will randomly create a number to compare with it.
- Gamma: It is a float type. This number is one of the parameters from the equation to update the Q value.
- Batch_size: It is an integer type. This is the max batch size of the batch.
- Epsilon_min: It is a float. This is the minimal number of epsilon.

- Epsilon_decay:It is a epsilon. We will use epsilon *=epsilon_decay to decrease the epsilon.
- Learning_rate: It is a float. It is the learning rate for updating the network.
- Memory: It is a deque. The max length is 100000. We use it as the batch.
- Model: It is a keras.model class. This is the model of Deep Q network.

Functions of this class used in our prototype are:

1) Build_model: The output model is a model class we import from keras and is to build the structure of the Deep Q network. It has 3 layers. The input of the network is the state. The state include the X value of the paddle, X value of the ball, Y value of the ball. The first and second hidden layers both have 64 n. The output layer has 3 neuron, which represents the 3 Q values of the 3 actions in the game. Because the agent will choose the action which has the largest Q values.
2) Remember: We will store the state, action, reward, next_state and done in a batch. Because we will train the network by batch.
3) Act: First, we randomly create a number from 0 to 1. Then check this random number is larger than epsilon or not. If its not, we random output an action. If it is larger than epsilon, we let the input state go through the Deep Q network and predict an action then output this action.
4) replay: We train the network by batch and update the weight in Deep Q network.

On the other hand in DoubleDQN class, parameters that have been used are:

- action_space: It is an integer type, it represent how many action we have in the game.In our game, we totally have 3 actions, which means paddle should move left, do nothing and move right.
- state_space: It is an integer type, it represent the dimension of the state.
- Epsilon: It is a float. When we choose an action, we will randomly create a number to compare with it.
- Gama: It is a float type. This number is one of the parameter from the equation to update the Q value.
- Batch_size: It is an integer type. This is the max batch size of the batch.
- Epsilon_min: It is a float. This is the minimal number of epsilon.
- Epsilon_decay:It is a epsilon. We will use epsilon *=epsilon_decay to decrease the epsilon.
- Learning_rate: It is a float. It is the learning rate for updating the network.
- Memory: It is a deque. The max length is 100000. We use it as the batch.
- Model: It is a keras.model class. This is the main network of Double Deep Q network.
- targeted_neural_network: It is a keras.model class. This is the target network of Double Deep Q network. Both main network and target network have the same structure.

Functions of this class used in our prototype are:

1) Build_model: The output model is a model class we import from keras. The function is to build the structure of the main network and target network. It has 3 layers.The input of the network is the state. The state include the X value of the paddle, X value of the ball, Y value of the ball. The first and second hidden layers both have 64 n. The output layer has 3 neuron, which represents the 3 Q values of the 3 actions in the game. Because the agent will choose the action which has the largest Q values.
2) Remember: We will store the state, action, reward, next_state and done in a batch. Because we will train the network by batch.
3) Act: First, we randomly create a number from 0 to 1. Then check this random number is larger than epsilon or not. If its not, we random output an action. If it is larger than epsilon, we let the input state go through the main network of Double Deep Q network and predict an action then output this action.
4) replay: We train the network by batch and update the weight in main network.
5) updated_taget_neural_network: Update the weights in target network. Because the replay function only updates the weights in main network. Double DQN use 2 different way to update the main and target network. That is the difference between Deep Q network.

C. Design to train the network:

At first, we create the paddle class. We will train the network for 1000 epochs. In each epoch, we will use reset function to initialize the game and get the first state as the first observation. Then we use act function in the DQN class or DDQN class to predict which action to take. Then use this action as input and call the next_iteration function in paddle class to make the game move to next frame. We will get a new state as a new observation and the reward of this action. Then we use memory function in the DQN class or DDQN class to store the state, action , new state and reward and done in the batch. If the batch is full , it means we have enough experience to let the network learn knowledge from the batch. We use replay function to train the network. If we use Double network, replay function only train the main network, and we will use updated_target_neural_network of DDQN class to update the target network in each 20 epochs. If the game is over we move to next epoch. When the all epochs are finished, we save the weights of the network as shown in figure 7.

D. Experiment Setting:

The code runs on Windows 10. The environment is Python 3.6 and Keras 2.24. Parameters setting is that epsilon is 1, gamma = .95, batch_size = 64, epsilon_min = .01, epsilon_decay = .995, learning_rate = 0.001.

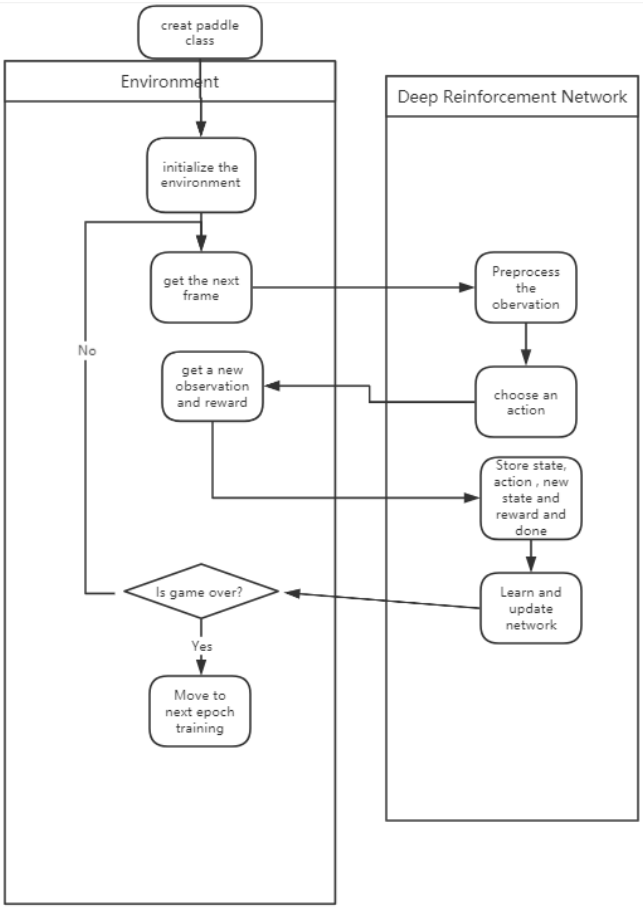

Fig. 7. Flow diagram to train the network

E. Graphical User Interface(GUI - Breakout):

Graphical User Interface of our customized breakout environment is shown in figure 8.

F. Results

After evaluating the performance, we found that the performance of Double Deep Q-learning was better because It uses two neural network models that are similar. During the experience replay, the other one is a replica of the first model's last state. This second model calculates the Q-value. In DQN, Q-value is determined with the reward added to the cumulative Q-value of the next state. If each time the Q-value determines a high number for a given state, the value derived for that particular state from the performance of the neural network will become higher each time. Every output value of the neuron will become higher and higher until the difference between each output value is high. DDQN is better because it reduces overestimations by decoupling agent selection function and target function.

The Conventional Deep Q-learning performance lacked because it is unable to prevent overfitting problems and because the targets would be the Q-values of each of the actions for training the neural network. As shown in Figure 9, the score

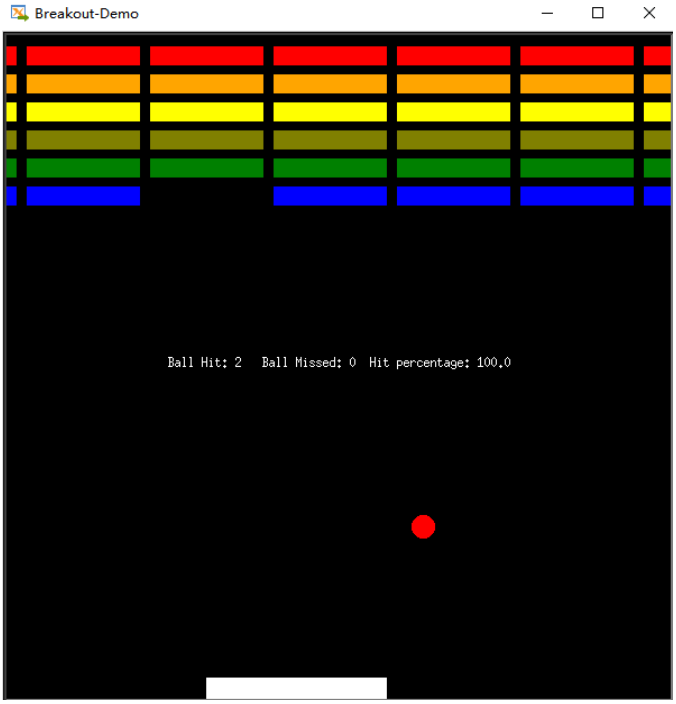

Fig. 8. GUI – customized environment of Breakout

and and Hits percentage of double DQN is higher (83.66 %) than DQN (78.47 %).

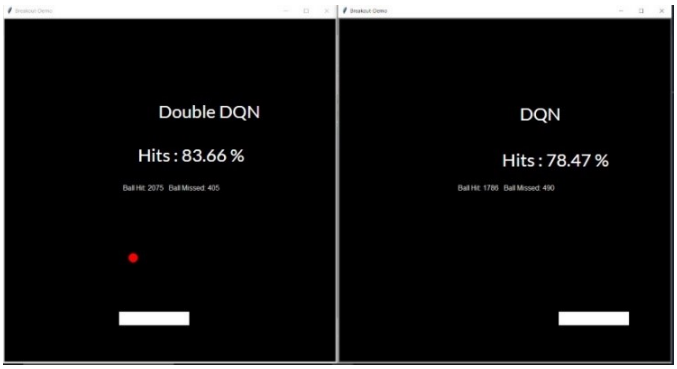

Fig. 9. Performance Comparison (Hit Rate)

In Figure 10 graphs, the rewards are much higher in the case double DQN compares to DQN. The left side shows the highest reward as 10 whereas the right side shows 5 to be max. The X-axis in graph represents episodes which 1000 in both cases. In each episode, we have used reset function to initialize the game and got the first state as the first observation. Then we have used an act function in the DQN class or DDQN class to predict which action to take.

Table III, shows the hyperparameters of both Double DQN and DQN. Hyperparameters are vital because they directly control the behaviour of the training algorithm and have a significant impact on the performance of the model is being trained. Hyperparameter can impact greatly on the model.

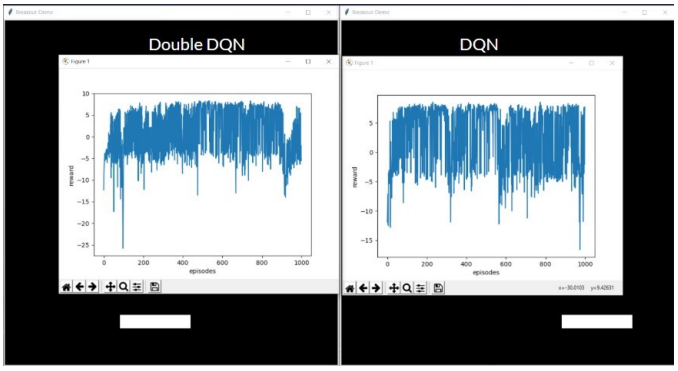

Fig. 10. Performance Comparison (Graph)

TABLE III
HYPER-PARAMETER TUNING

Hyper-parameter	Value
Epsilon	1
gamma	0.95
batch_size	64
epsilon_min	0.01
epsilon_decay	0.995
learning_rate	0.001

VI. CONCLUSION

To put it into a nutshell, after running the game for both models (Deep Q-learning and Double deep Q-learning) for the training phase using hyper-parameters as mentioned in Table III, DDQN performed better in terms of hit-rate which is evident from figure 9. Hit rate is calculated by taking the ratio of number of times the ball hits the paddle to the sum of number of times ball hits the paddle and number of times ball misses the paddle, for the given number of episodes and which in our case is 100. In DDQN highest reward assigned to a particular action was better as compared to DQN which varied from (-10,10) for DDQN and (-5,5) for DQN which states that DDQN explores the environment better than DQN because of its target network which is evident from figure 10.

REFERENCES

[1] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.
[2] Gagniuc, Paul A. Markov chains: from theory to implementation and experimentation. John Wiley & Sons, 2017.
[3] Gerald Tesauro. Temporal difference learning and td-gammon. Communications of the ACM, 38(3):58–68, 1995.
[4] Jordan B. Pollack and Alan D. Blair. Why did td-gammon work. In Advances in Neural Information Processing Systems 9, pages 10–16, 1996.
[5] Human-level control through deep reinforcement learning. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Nature. 2015 Feb 26; 518(7540): 529–533. doi: 10.1038/nature14236.
[6] Silver, David & Huang, Aja & Maddison, Christopher & Guez, Arthur & Sifre, Laurent & Driessche, George & Schrittwieser, Julian & Antonoglou, Ioannis & Panneershelvam, Veda & Lanctot, Marc & Dieleman, Sander & Grewe, Dominik & Nham, John & Kalchbrenner, Nal & Sutskever, Ilya & Lillicrap, Timothy & Leach, Madeleine & Kavukcuoglu, Koray & Graepel, Thore & Hassabis, Demis. (2016). Mastering the game of Go with deep neural networks and tree search. Nature. 529. 484-489. 10.1038/nature16961. Van Hasselt, H., Guez, A., & Silver, D. Deep reinforcement learning with double qlearning. Paper presented at the Thirtieth AAAI Conference on Artificial Intelligence, 2016.
[7] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double Q-Learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16). AAAI Press, 2094–2100.
[8] Patel, Devdhar & Hazan, Hananel & Saunders, Daniel & Siegelmann, Hava & Kozma, Robert. (2019). Improved robustness of reinforcement learning policies upon conversion to spiking neuronal network platforms applied to Atari Breakout game. Neural Networks. 120. 10.1016/j.neunet.2019.08.009.
[9] Jeerige, Anoop, Doina Bein, and Abhishek Verma. "Comparison of deep reinforcement learning approaches for intelligent game playing." In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), 2019.
[10] DQNViz: A Visual Analytics Approach to Understand Deep Q-Networks.Junpeng Wang, Liang Gou, Han-Wei Shen, Hao Yang. In IEEE Transactions on Visualization & Computer Graphics, 2019 Jan;25(1): 288-298.