IMPLEMENTATION OF PONG GAME USING DEEP Q NETWORKS (DQN)

CECS 590 – Deep Learning

Fall 2018

Under the Guidance of Professor ***Dr. Roman Tankelevich***

Akshatha Thonnuru Swamygowda (017532175)

Sarveshwaran Sampathkumar (017387654)

California State University Long Beach

ABSTRACT:

The main idea behind this project is to build a PONG game using Deep Neural Networks (DNN). Especially we will be making use of the principles of Deep Q Networks (DQN) in developing this project. A Deep Q Network makes use of key concepts of the DNN along with the Reinforcement Learning (RL). In order to successfully implement this project, we will have to make use of the pixels to identify the position of the object on the screen and we will associate a score for it. These two data will be used by the model to position the paddle of the player based on the incoming ball. The concept of RL will be used here, where the model learns the movement/position of the paddle over a period by trial and error method. A Convolution Neural Network (CNN) will make use of the pixel and score data and passes these data over a good amount of convolution layers to produce the required output. The activation function of the neurons in the convolution layer will make use of Rectified Linear Unit (ReLU) to process the incoming data and produce a meaningful decision. Using a generalised artificial intelligence by implementing a network as explained above helped in developing this game of pong where the paddle was able to observe the game play for a good amount of time, after which the paddle was able to play the game and, in the end, master the game by itself.

OUTLINE:

The report on “Implementation of PONG game using Deep Q Networks (DQN)” is designed in such a way that it covers the following in the same order as below

- Introduction
- Background
- Reinforcement Learning
- Markov Decision Process
- Q Learning
- Deep Q Network
- Deep Q Learning Algorithm
- Implementation
- Flow of the code
- Preliminary Findings
- Observation and Outcome
- Bibliography
- Reference Paper
- Appendix – Source Code

INTRODUCTION:

The idea behind this project is to demonstrate an example of the “General Artificial Intelligence” and the general idea behind this concept. We will be implementing the game of PONG which makes use of the concept behind general artificial intelligence to learn and master the game. Before we get into the concept of general artificial intelligence, we will first understand what an artificial intelligence is. Artificial intelligence (AI) is the intelligence which is shown by a computer program in mastering a skill or task. For example, we can see the implementation of artificial intelligence in the autonomous vehicles. AI programmed for autonomous vehicles can only be implemented for those vehicles. But in contrast General Artificial Intelligence is where the intelligence exhibited by the program can be applied not only on one but on variety of fields. This idea is further explained in this report along with the implementation of the PONG game using Deep Q Networks.

BACKGROUND:

In 2013, Google bought a London based start up called DeepMind for a whopping 500 million dollars. The reason behind this huge acquisition was because DeepMind was one of the few to work on the General Artificial Intelligence. DeepMind published a paper “Playing Atari with Deep Reinforcement Learning” explaining how their algorithm was designed to play the games by just making use of the pixel information from the screen. They also explained how their algorithm was able to master different games with different goals. This big step towards generalised artificial algorithm attracted Google to acquire them. This project is based on “Human Level Control Through Deep Reinforcement Learning” which explains how the same algorithm could be applied on forty-nine different games to master the game using the pixel information from the screen. PONG was one among the forty-nine games. The game of PONG is implemented as part of this project using Deep Q Network technique along with the Reinforcement Learning techniques. The below graph shows all the games which could make use of the generalised artificial intelligence. It also compares the output efficiency of the Deep Q Network game play versus the human game play which clearly shows that Deep Q Network game play was more efficient than the human game play.

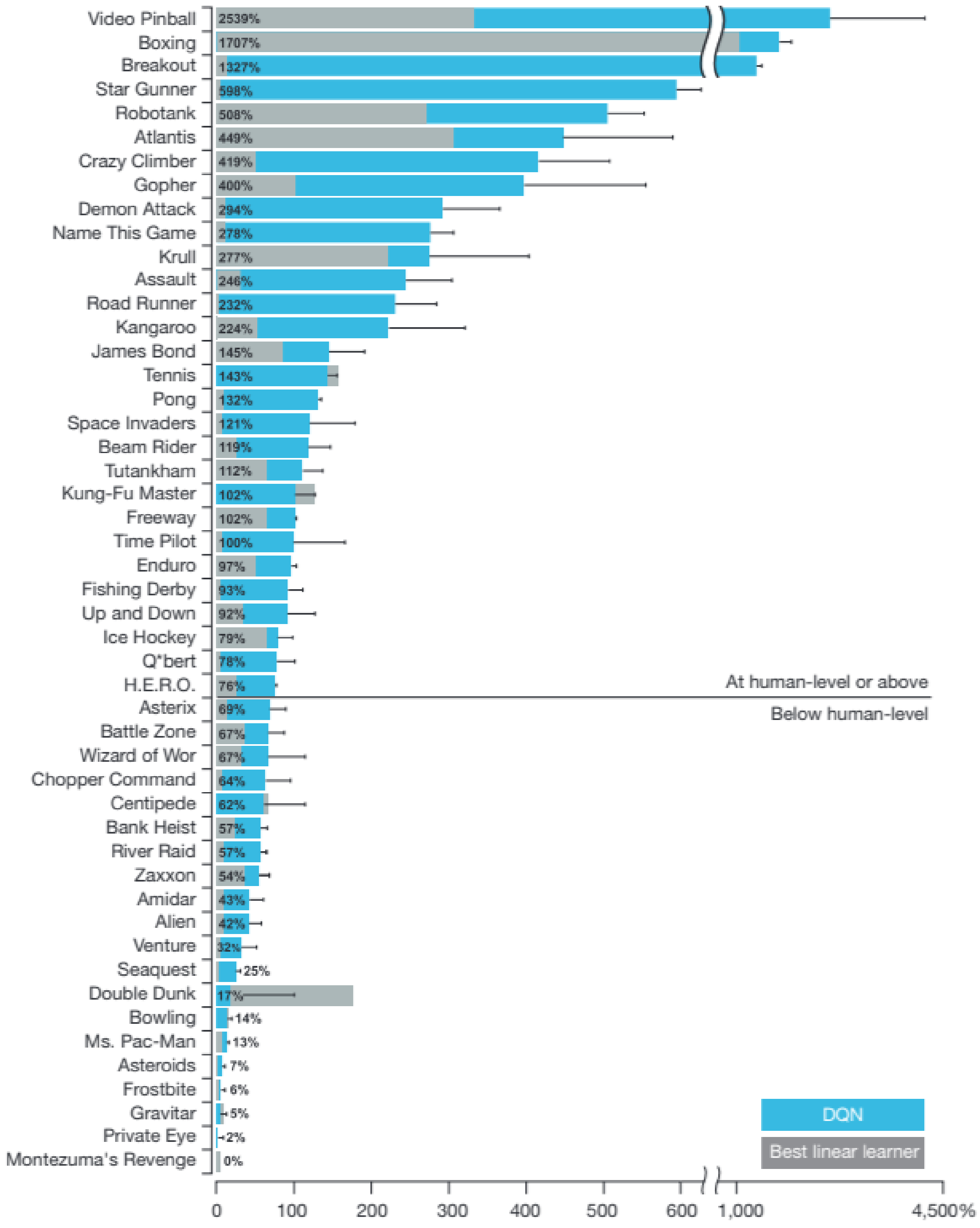


Figure 1 – Game Play Comparison Between Human vs Deep Q Learning Agent

REINFORCEMENT LEARNING:

Before we investigate reinforcement learning, we will first understand what supervised and unsupervised learnings are. In case of supervised learning, we try to find a mapping function between the input and output labels. It is based on classification and regression. This is applied on the labelled data. On the other hand, in case of unsupervised learning, the machine is made to learn the input data which are unlabelled by finding similarities between them. This is usually done by clustering. Reinforcement learning is something which falls in between these two learning techniques. As we see that supervised learning works on the labelled data and unsupervised learning works on unlabelled data, the reinforcement learning works based on the rewards. The concept of reward is explained below.

The working of Reinforcement learning is as follows. The agent works on an environment where there are many possibilities of actions on it. Bases on the action performed on the environment, the state changes and because of which the agent might get a reward.

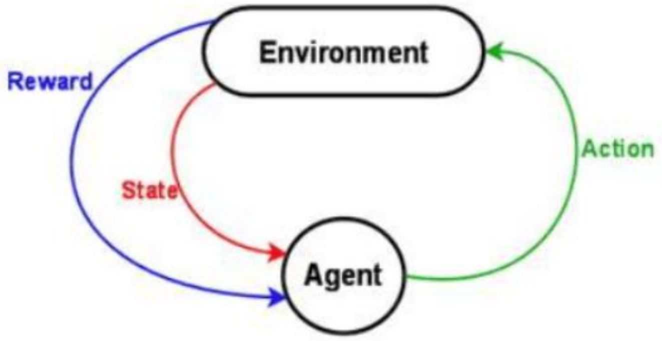


Figure 2 – Reinforcement Learning

MARKOV DECISION PROCESS:

The reinforcement learning technique can be formalised using Markov Decision Process. The rules needed for transforming from one state to another state along with the states and actions involved forms the basis for the Markov Decision Process.

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

The above is termed as one episode where it starts with state (s_0) and end with state (s_n). Here the state is represented by (s_i), action is represented by (a_i) and the reward which is received on doing that action is represented by (r_{i+1}). This Markov Decision Process helps us in evaluating which action to choose based on the state at which the agent is at. For it to be smart it should not only look for short term rewards but for long term rewards. For one episode, the total reward can be expressed as follows:

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

Total future reward can be expressed as follows:

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

Where t represents the time point. Taking discounted future reward into consideration, we can represent R_t as follows:

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

The reason behind using the discounted future reward is that, the more the future we go into, the chances of getting the same reward varies to a greater extend. An agent should choose a action in such a way that the agent can maximise the reward.

Q LEARNING:

As explained above, the agent should look for long term rewards rather than just looking for short term rewards. Whenever the agent wants to choose an action on an environment, the agent picks up an action which has the highest Q (Quality) value from the list of available actions. The basic idea on how this work is given in the below flow chart

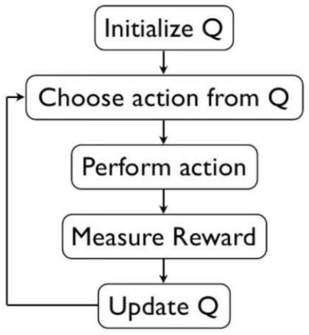
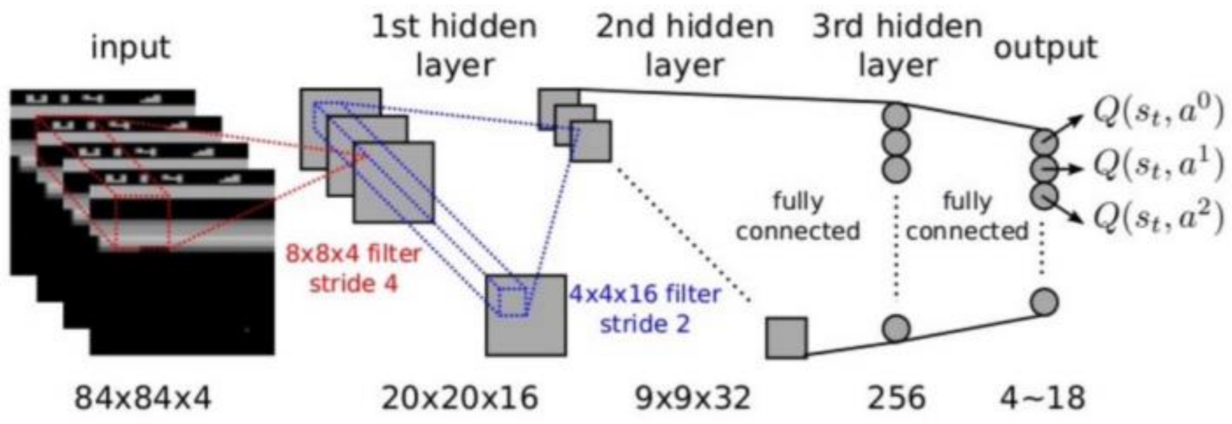


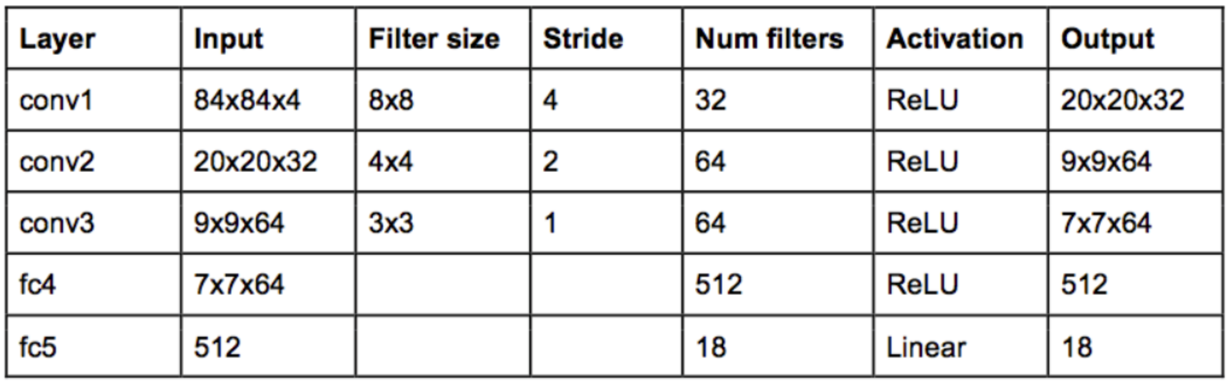
Figure 3 – Q Learning

A Q function is a matrix with states as rows and actions as columns. Initially the Q matrix is initialised with random values. Then the agent is made to pick up an action from the Q matrix which has the highest quality value for that action. Once the action is picked up, the agent is made to perform that action. Based on the performed action on the environment, the state of it changes and the agent observes if he gets a reward for the performed action or not. Based on the reward, the Q matrix will be updated. And the agent performs the same set of actions again. This process keeps repeating and the Q matrix gets updated for every cycle. This is called Q Learning.

DEEEP Q LEARNING:

The architecture which was used by DeepMind is as follows:





Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Figure 4 – DeepMind`s Architecture

One thing which is common between all the games is the pixel from the game screen. If we can convert the pixel information of the game from the screen into actions and then input the game to the algorithm, the algorithm will be smart enough to master the game. As shown in the figure 4, we will make use of three layers of Convolution Neural Network (CNN) followed by two layers of fully connected network. The input to this architecture will be four different game screens in order to represent the game characters speed and direction. The output from this will the Q values of all the possible actions. This approximation of Q function is done by using the Deep networks and hence this is called as Deep Q Network.

DEEP Q LEARNING ALGORITHM:

The algorithm used in implementing the PONG game is as follows:

```
initialize replay memory $D$
initialize action-value function $Q$ with random weights
observe initial state $s$
repeat
     select an action $a$
          with probability $\varepsilon$ select a random action
          otherwise select $a = \text{argmax}_{a'} Q(s,a')$
     carry out action $a$
     observe reward $r$ and new state $s'$
     store experience $<s, a, r, s'>$ in replay memory D

     sample random transitions $<ss, aa, rr, ss'>$ from replay memory D
     calculate target for each minibatch transition
          if ss' is terminal state then $tt = rr$
          otherwise $tt = rr + \gamma \max_{a'} Q(ss', aa')$
     train the Q network using $(tt - Q(ss, aa))^2$ as loss

     $s = s'$
until terminated
```

Figure 5 – Deep Q Learning Algorithm

The explanation along with the way it has been implemented in our project is explained in the below sections.

IMPLEMENTATION:

The implementation of PONG game is as follows. The project was implemented in Python. The libraries used are as follows:

- TensorFlow
- CV2 – OpenCV
- Numpy
- Random
- Pygame

The use of each of the above libraries has their own significance. TensorFlow for example is used to implement high complex numerical calculation and to provide the ability to run on the GPUs. OpenCV is used to format the pixel data for the Tensor Flow to understand it. Numpy is used to implement the concept of arrays in the program and Pygame is used to implement the games in python.

The methods defined for the implementation of PONG Game are as follows

- definePingPongBall
- defineLeftPaddle
- defineRightPaddle
- updatePingPongBall
- updateLeftPaddle
- updateRightPaddle
- returnPresentFrameInformation
- returnNextFrameInformation
- tensorFlowGraph
- deepQNetworkGraphTraining

The first three methods – definePingPongBall, defineLeftPaddle and defineRightPaddle are used to define the pingpong ball, left and right paddle. Position of the ball using X and Y coordinates are made use in defining the pong ball. YPosition along with the height and width of the paddle is made use in defining the left and right paddle.

The next three methods – updatePingPongBall, updateLeftPaddle and updateRightPaddle are used to update the pingpong ball, left and right paddle. updatePingPongBall takes care of the direction of the ball based on the position of the ball. When the ball hits the left paddle, it updates the score counter and changes the direction of the ball to right side. Similarly, when the ball hits the right paddle, it updates the score counter and changes the direction of the ball to left side. If the ball doesn't hit the left or right paddle, it only changes the direction of the ball without updating the score. If the ball hits the top window, it changes the direction of the ball to down side. Similarly, if the ball hits the bottom window, it changes the direction of the ball to top side. Based on the action performed by the agent, updateLeftPaddle and updateRightPaddle methods move the direction of the paddle to either up or down. These methods also take care of the paddle not moving out of the window scenario.

The next method – returnPresentFrameInformation provides the pixel information of the current screen whereas the next method – returnNextFrameInformation provides the pixel information of the next screen based on the action performed by the agent. Also, in this method we make calls to defineLeftPaddle, defineRightPaddle and definePingPingBall to define the position of the left paddle, right paddle and the ping pong ball.

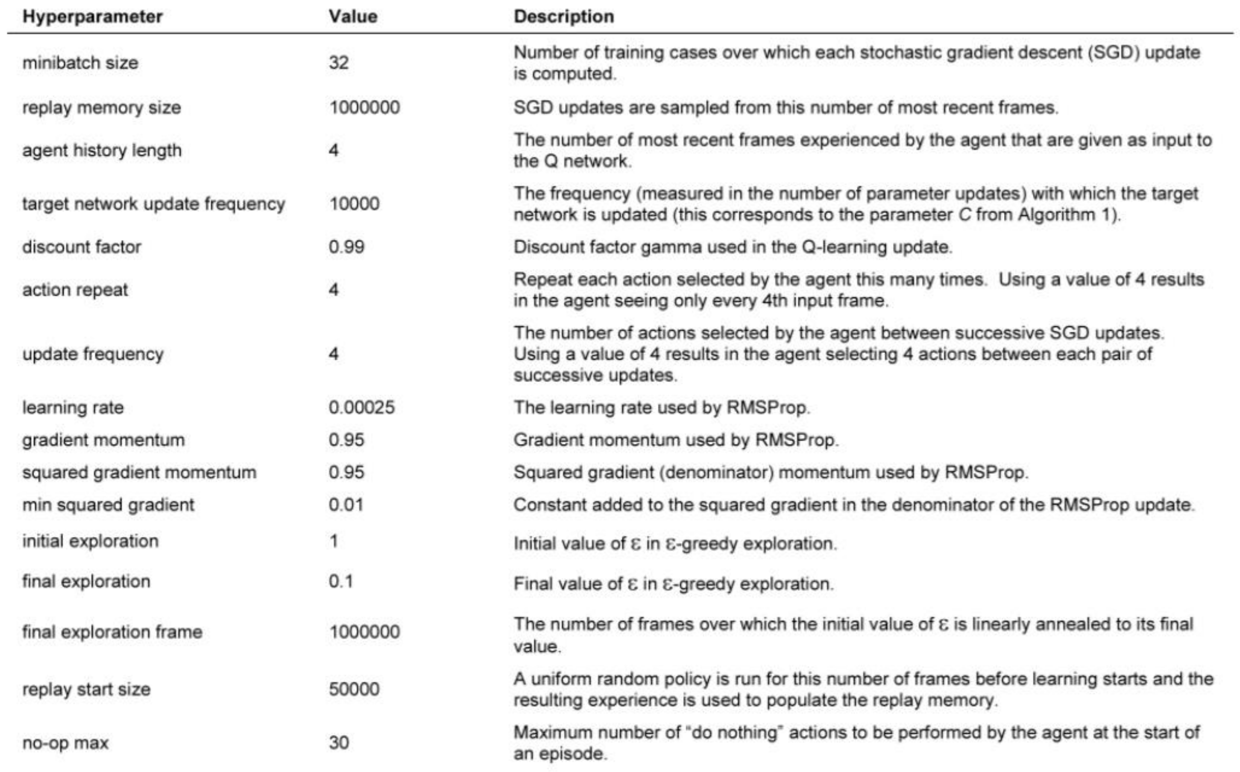
The tensorFlowGraph method is used to define all the 3 convolution layers and the 2 fully connected layers as explained in figure 4. The last method is where the actual Deep Q Network algorithm is implemented which is explained in the following section.

FLOW OF THE CODE:

The game is designed in such a way that the paddle on the right will be able to hit the ball every time the ball reaches the right side of the window without missing it. The one on the left, i.e. the Left Paddle is the one which we are interested in. The left paddle will observe and learn. All the logics are implemented on the left paddle. The We first start by initializing the libraries, the parameter used for this project followed by the networks which we will be setting up for this game. Here we create a network with three CNN layers followed by 2 fully connected layers. And then we input four different game screens in order to represent the game characters speed and direction. The output from this will the Q values of all the possible actions. Experiences are stored in deque data structure. Parameters such as (OBSERVE, EPSILON, GAMMA, BATCH_SIZE, etc) are initialized. Action is set to 3 which can either be up, down or do nothing. Gamma which is the learning rate is set to 0.99. Initial Epsilon is set to 1.0 and the final epsilon is set to 0.05. Explore is set to 500000 steps and the Observe is set to 50000. The game is designed to observe for 50000 steps and then keep exploring for 500000 steps. Batch size is set to 100. The next step is to observe what each action does. We start the game and perform a random action from the list of available actions. Based on the performed action, we will observe the change in state and see if we receive any reward for the performed action. Based on it the Q matrix will be updated as explained in figure 3. The next step is to learn from the observations. This is where we create mini batches and for the Q function, we create a Bellman equation. We train the network.

PRELIMINARY FINDINGS:

In the early stage of development of this game, there were few scenarios which were not considered. Few such scenarios are – paddle going out of the window, ball moving in the wrong direction etc. When the game was run for the first time, the above scenarios did happen in the simulation. The program was then corrected to handle all those scenarios which were not considered while developing the core functionality. Apart from these there were also challenges faced in setting the right set of values for the parameters used in this project. Though the DeepMind provided the set of values which they used when implementing this algorithm, it was not straight forward for us to use the same set of values. This was due to the fact that the given values were for generic scenario and was not specific for the PONG game. Also, due to the restriction on the hardware we had to make few changes in the parameters. List of parameters value provided by DeepMind is shown in the below table (Figure 6). As part of initial inspection, all the above-mentioned items were fixed for the code to run successfully. All the observation of this initial run along with the changes which were made to make it more accurate are explained in the next section.



Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter *C* from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ε is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

Figure 6 – Parameter and their value used by DeepMind

OBSERVATION AND OUTCOMES:

Once the core functionalities of the Deep Q Network were implemented on the game of PONG, we ran the iteration – 1 to see how the system behaved. During the iteration – 1 as explained in the preliminary findings, few of the main scenarios such as paddle going out of the windows were not implemented due to which the execution was a failure, which is clearly depicted in the graph (Figure – 7) the green line shows that there were zero hits by the paddle on all the days. After fixing all the issues in the first iteration, we ran it for the second time (iteration – 2) by setting the observe parameter to 25000 and explore parameter to 100000. i.e. the agent was made to observe the game for 25000-time steps and then it could explore the game for 100000-time steps. When such setup was implemented, the output of the system looked better as shown in the same graph with the blue line. We can see that the left paddle was able to hit the ball on few occasions on day one and as the days passed, it was able to hit it better. And hence, the raise in blue line. To tweak the current configuration to make it work better, we changed the value of the observer parameter to 50000-time steps and the explore parameter to 500000-time steps along with changes to few other parameters. This change helped us in producing better results in third iteration as shown by orange line in the graph below.

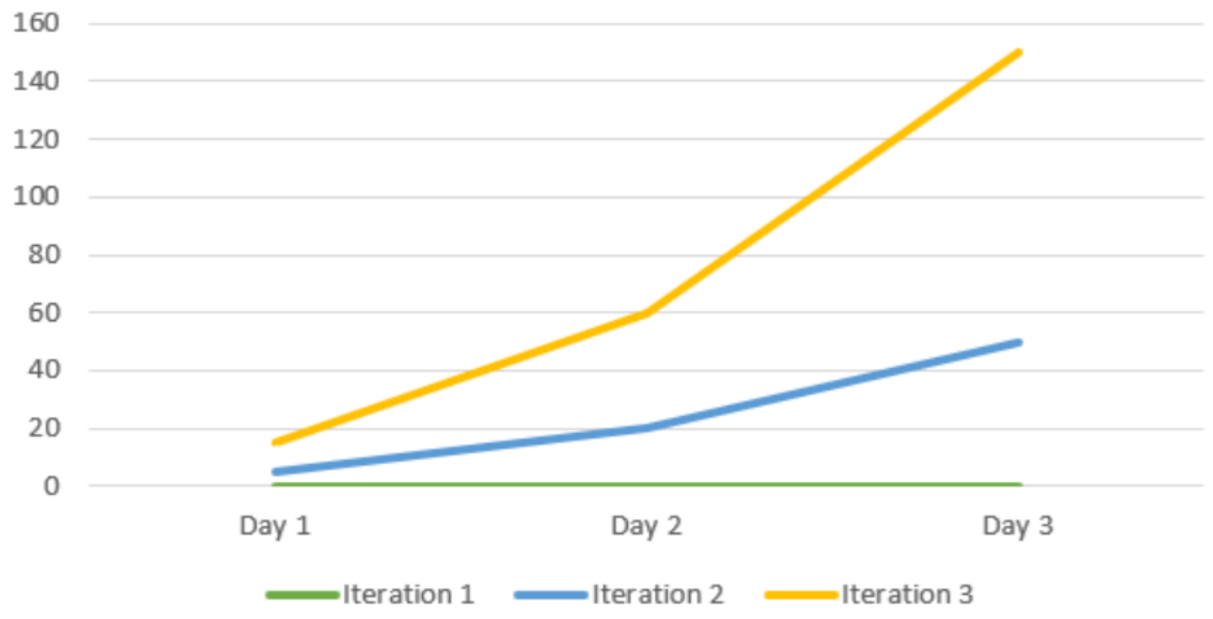
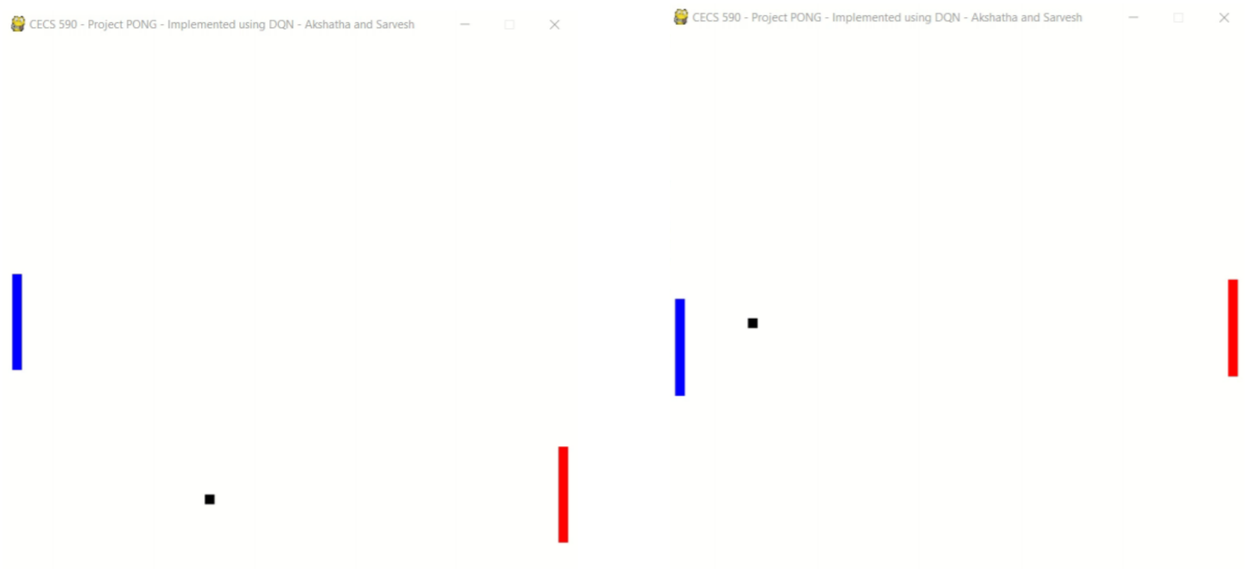


Figure 7 – Performance of the Game in every iteration


CECS 590 - Project PONG - Implemented using DQN - Akshatha and Sarvesh
CECS 590 - Project PONG - Implemented using DQN - Akshatha and Sarvesh

s

Figure 8 – Two instance of the game play (LeftPaddle – AI and RightPaddle – Computer)

BIBLIOGRAPHY:

The following are the list of sources used for this implementation of PONG game using Deep Q Network:

- Reference Paper – "Human-level control through deep reinforcement learning" by Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis
- Related Paper – "Playing Atari with Deep Reinforcement Learning" by Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller

REFERENCE PAPER:

The below attached are the copy and the link of the paper "Human-level control through deep reinforcement learning" and the paper "Playing Atari with Deep Reinforcement Learning"

Link for the reference paper:

https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf

Copy:

Reference_Paper.pdf

Link for the related paper:

https://arxiv.org/pdf/1312.5602.pdf

Copy:

Related_Paper.pdf

APPENDIX – SOURCE CODE:

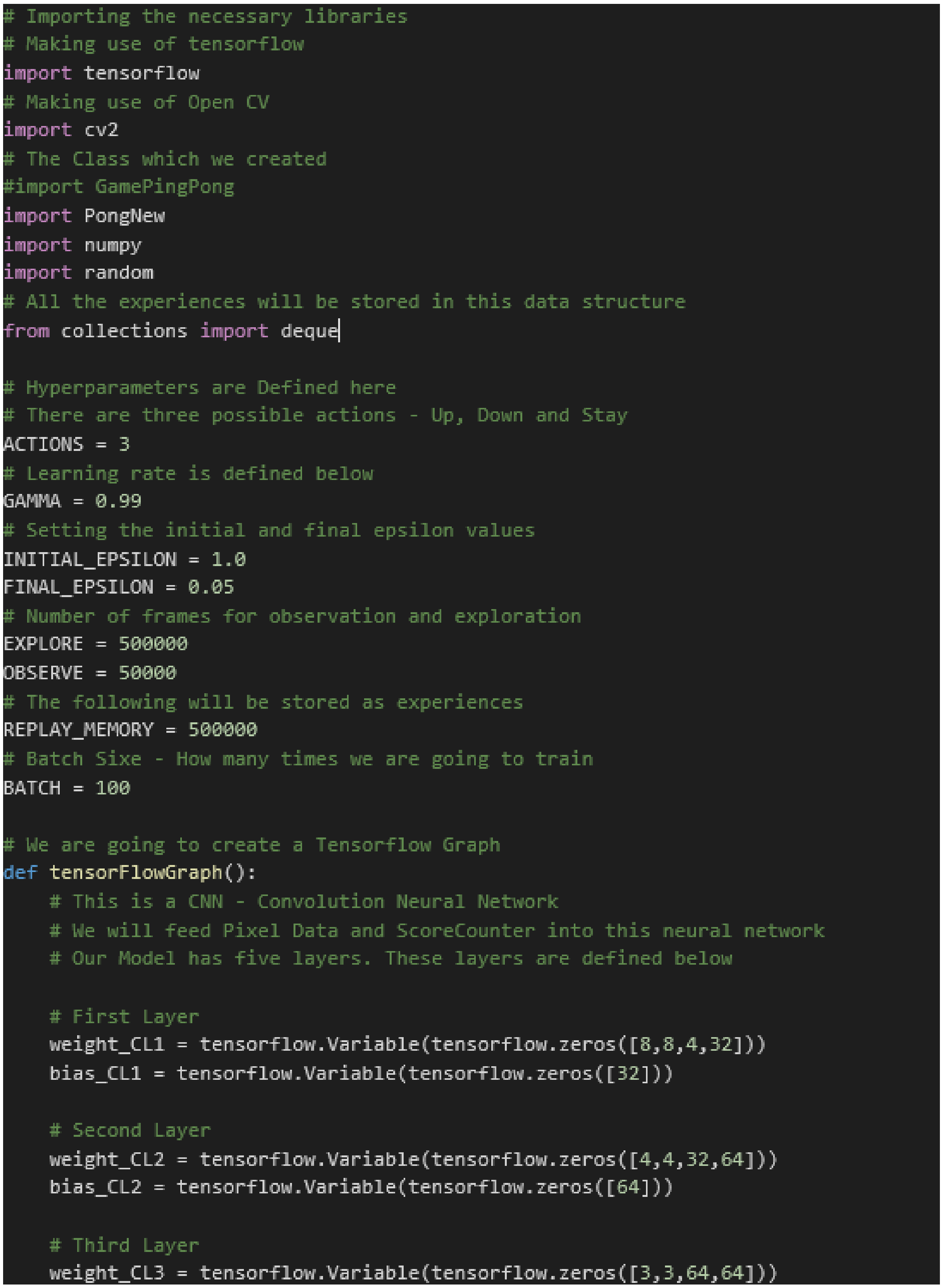
The project consists of two Python files – GamePingPong.py and ReinforcedLearning.py

GamePingPong.py

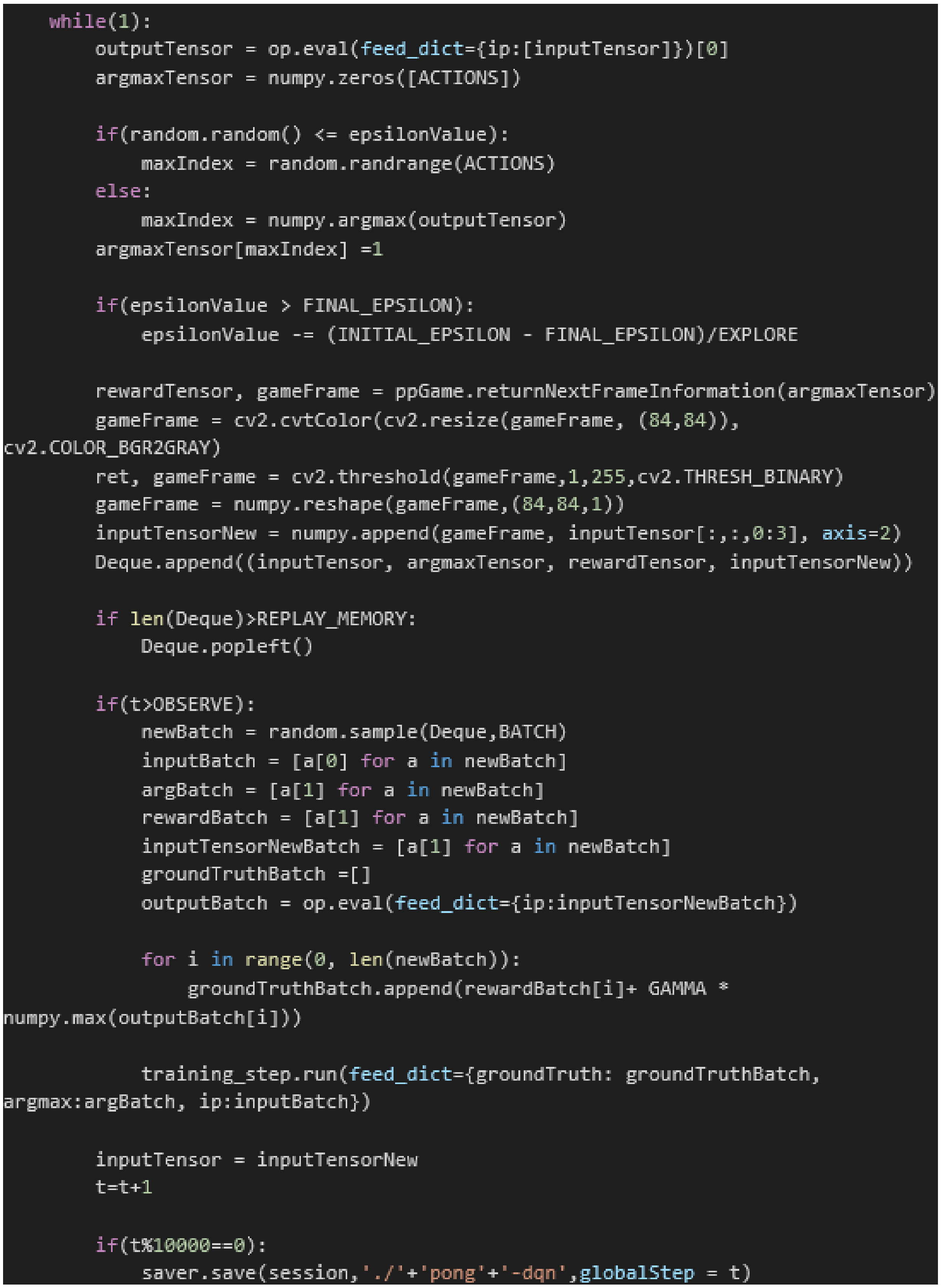
```
# PYGAME is used here to implement Ping Pong Game in Python
import pygame
# Random is used here to define the direction the ball is going to move
import random

# Variable initialization
# Below defines how fast the game is going to move
FrameRatePerSecond = 60
# Window Size
HeightOfWindow = 400
WidthOfWindow = 400
# Paddle Size
HeightOfPaddle = 60
WidthOfPaddle = 10
DistanceFromWindow = 10
# Ball Size (Small Rectangle)
HeightOfBall = 10
WidthOfBall = 10
# Paddle and Ball Speed
SpeedOfPaddle = 2
SpeedOfBallInXDirection = 3
SpeedOfBallInYDirection = 2
# Colors for the Ping Pong Game
WhiteColor = (255,255,255)
BlackColor = (0,0,0)

# Game Screen Initialization
GameScreen = pygame.display.set_mode((WidthOfWindow,HeightOfWindow))

# Ball Initialization
def definePingPongBall(XPositionOfBall,YPositionOfBall):
    # Rect function of the PYGAME is used to draw a rectange (ball)
    PingPongBall = pygame.Rect(XPositionOfBall, YPositionOfBall, WidthOfBall,
HeightOfBall)
    pygame.draw.rect(GameScreen, WhiteColor, PingPongBall)

# LeftPaddle Initialization - This Paddle learns by itself
def defineLeftPaddle(YPositionOfLeftPaddle):
    LeftPaddle = pygame.Rect(DistanceFromWindow, YPositionOfLeftPaddle,
WidthOfPaddle, HeightOfPaddle )
    pygame.draw.rect(GameScreen, WhiteColor, LeftPaddle)

# RightPaddle Initialization = This will be the unbeatable paddle
def defineRightPaddle(YPositionOfRightPaddle):
    RightPaddle = pygame.Rect(WidthOfWindow - DistanceFromWindow - WidthOfPaddle,
YPositionOfRightPaddle, WidthOfPaddle, HeightOfPaddle )
    pygame.draw.rect(GameScreen, WhiteColor, RightPaddle)

# Update the position of the ball
def updatePingPongBall(YPositionOfLeftPaddle,
YPositionOfRightPaddle,XPositionOfBall, YPositionOfBall, XDirectionOfBall,
YDirectionOfBall):
    XPositionOfBall = XPositionOfBall + XDirectionOfBall *
SpeedOfBallInXDirection
    YPositionOfBall = YPositionOfBall + YDirectionOfBall *
SpeedOfBallInYDirection
    ScoreCounter = 0
    # If the ball hits the left window, then change the direction of the ball to
right and update the ScoreCounter accordingly
    if (XPositionOfBall <= DistanceFromWindow + WidthOfPaddle and YPositionOfBall
+ HeightOfBall >= YPositionOfLeftPaddle and YPositionOfBall - HeightOfBall <=
YPositionOfLeftPaddle + HeightOfPaddle):
        XDirectionOfBall = 1
    elif (XPositionOfBall <= 0):
        XDirectionOfBall = 1
        ScoreCounter = -1
        return[ScoreCounter, YPositionOfLeftPaddle, YPositionOfRightPaddle,
XPositionOfBall, YPositionOfBall, XDirectionOfBall, YDirectionOfBall]
    # If the ball hits the right window, then change the direction of the ball to
left and update the ScoreCounter accordingly
    if (XPositionOfBall >= WidthOfWindow - WidthOfPaddle - DistanceFromWindow and
YPositionOfBall + HeightOfBall >= YPositionOfRightPaddle and YPositionOfBall -
HeightOfBall <= YPositionOfRightPaddle + HeightOfPaddle):
        XDirectionOfBall = -1
    elif (XPositionOfBall >= WidthOfWindow - WidthOfBall):
        XDirectionOfBall = -1
        ScoreCounter = 1
        return[ScoreCounter, YPositionOfLeftPaddle, YPositionOfRightPaddle,
XPositionOfBall, YPositionOfBall, XDirectionOfBall, YDirectionOfBall]
    # If the ball hits the top window, then change the direction of the ball to
down
    if (YPositionOfBall <= 0):
        YPositionOfBall = 0
        YDirectionOfBall = 1
    # If the ball hits the bottom window, then change the direction of the ball
to up
    elif (YPositionOfBall >= HeightOfWindow - HeightOfBall):
        YPositionOfBall = HeightOfWindow - HeightOfBall
        YDirectionOfBall = -1
    return[ScoreCounter, YPositionOfLeftPaddle, YPositionOfRightPaddle,
XPositionOfBall, YPositionOfBall, XDirectionOfBall, YDirectionOfBall]

# Update the positon of the LeftPaddle
def updateLeftPaddle(action, YPositionOfLeftPaddle):
    # If the ball moves up, then move the positon of the left paddle up
    if(action[1] == 1):
        YPositionOfLeftPaddle = YPositionOfLeftPaddle - SpeedOfPaddle
    # If the ball moves down, then move the positon of the left paddle down
    if(action[2] == 2):
        YPositionOfLeftPaddle = YPositionOfLeftPaddle + SpeedOfPaddle
    # To make sure that the left paddle doesnt go out of the window
    if (YPositionOfLeftPaddle <0):
        YPositionOfLeftPaddle =0;
    if (YPositionOfLeftPaddle > HeightOfWindow - HeightOfPaddle):
        YPositionOfLeftPaddle = HeightOfWindow - HeightOfPaddle
    return YPositionOfLeftPaddle

# Update the position of the RightPaddle
def updateRightPaddle(YPositionOfRightPaddle, YPositionOfBall):
    # To move the right paddle down based on the position of the ball
    if(YPositionOfRightPaddle + HeightOfPaddle/2 < YPositionOfBall +
HeightOfBall/2):
        YPositionOfRightPaddle = YPositionOfRightPaddle + SpeedOfPaddle
    # To move the right paddle up based on the position of the ball
    if(YPositionOfRightPaddle + HeightOfPaddle/2 > YPositionOfBall +
HeightOfBall/2):
        YPositionOfRightPaddle = YPositionOfRightPaddle - SpeedOfPaddle
    # To make sure that the right paddle doesnt go out of the window
    if(YPositionOfRightPaddle < 0):
        YPositionOfRightPaddle = 0;
    if(YPositionOfRightPaddle > HeightOfWindow - HeightOfPaddle):
        YPositionOfRightPaddle = HeightOfWindow - HeightOfPaddle
    return YPositionOfRightPaddle

#Class for the GamePingPong
class GamePingPong:
    def __int__(self):
        # Random will be made used to initilise the ball's initial direction
        randomNum = random.randint(0.9)
        # Tally will be used to keep track of the ScoreCounter
        self.tally = 0
        # Left and Right Paddle will be placed in the middle of the left and
right window
        self.YPositionOfLeftPaddle = HeightOfWindow/2 - HeightOfPaddle/2
        self.YPositionOfRightPaddle = HeightOfWindow/2 - HeightOfPaddle/2
        # Initialize the Direction of the Ball
        self.XDirectionOfBall = 1
        self.YDirectionOfBall = 1
        # Initialize the Starting Point of the Ball
        self.XPositionOfBall = WidthOfWindow/2 - WidthOfBall/2

        # The below will decide the direction of the movement of the ball
randomly
        if(0 < randomNum < 3):
            self.XDirectionOfBall = 1
            self.YDirectionOfBall = 1
        if(3 <= randomNum < 5):
            self.XDirectionOfBall = -1
            self.YDirectionOfBall = 1
        if(5 <= randomNum < 8):
            self.XDirectionOfBall = 1
            self.YDirectionOfBall = -1
        if(8 <= randomNum < 10):
            self.XDirectionOfBall = -1
            self.YDirectionOfBall = -1
        randomNum = random.randint(0,9)
        self.YPositionOfBall = randomNum * (HeightOfWindow - HeightOfBall)/9

    # Gives the present frame at which we are currently at
    def returnPresentFrameInformation(self):
        pygame.event.pump()
        # Fill the screen with black color
        GameScreen.fill(BlackColor)
        # Define the Left Paddle, Right Paddle and the PingPong Ball by calling
their respective functions
        defineLeftPaddle(self.YPositionOfLeftPaddle)
        defineRightPaddle(self.YPositionOfRightPaddle)
        definePingPongBall(self.XPositionOfBall, self.YPositionOfBall)
        # Get the pixel information of the frame and store it in a three
dimension array
        pixelDataOfFrame = pygame.surfarray.array3d(pygame.display.get_surface())
        # Below is used to update the window
        pygame.display.flip()
        # Return the pixel information
        return pixelDataOfFrame

    # Gives the next frame
    def returnNextFrameInformation(self, action):
        pygame.event.pump()
        # Initialise the ScoreCounter with 0
        ScoreCounter = 0
        # Fill the screen with black color
        GameScreen.fill(BlackColor)
        # Define the Left Paddle, Right Paddle and the PingPong Ball by calling
their respective functions
        self.YPositionOfLeftPaddle = updateLeftPaddle(action,
self.YPositionOfLeftPaddle)
        defineLeftPaddle(self.YPositionOfLeftPaddle)
        self.YPositionOfRightPaddle =
updateRightPaddle(self.YPositionOfRightPaddle, self.YPositionOfBall)
        defineRightPaddle(self.YPositionOfRightPaddle)
        [ScoreCounter, self.YPositionOfLeftPaddle, self.YPositionOfRightPaddle,
self.XPositionOfBall, self.YPositionOfBall, self.XDirectionOfBall,
self.YDirectionOfBall] = updatePingPongBall(self.YPositionOfLeftPaddle,
self.YPositionOfRightPaddle, self.XPositionOfBall, self.YPositionOfBall,
self.XDirectionOfBall, self.YDirectionOfBall)
        definePingPongBall(self.XPositionOfBall, self.YPositionOfBall)
        # Get the pixel information of the frame and store it in a three
dimension array
        pixelDataOfFrame = pygame.surfarray.array3d(pygame.display.get_surface())
        # Below is used to update the window
        pygame.display.flip()
        self.tally = self.tally + ScoreCounter
        print("Final Score: " + str(self.tally))
        # Return the pixel information and ScoreCounter
        return pixelDataOfFrame
```

End of GamePingPong.py

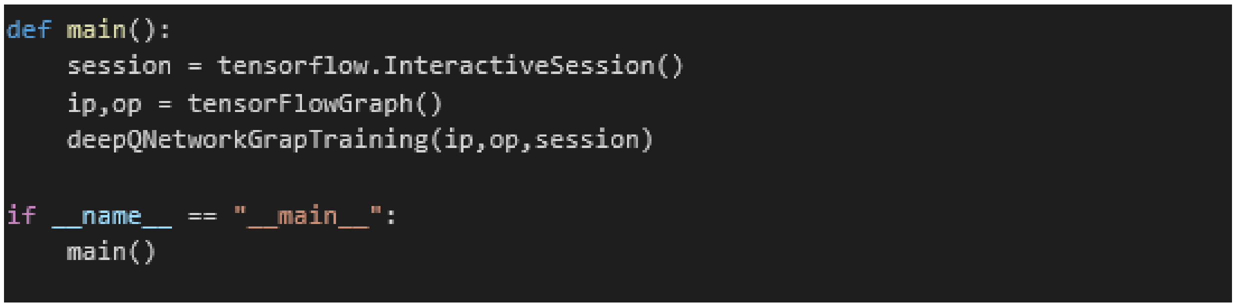
ReinforcedLearning.py



```
# Importing the necessary libraries
# Making use of tensorflow
import tensorflow
# Making use of Open CV
import cv2
# The Class which we created
#import GamePingPong
import PongNew
import numpy
import random
# All the experiences will be stored in this data structure
from collections import deque

# Hyperparameters are Defined here
# There are three possible actions - Up, Down and Stay
ACTIONS = 3
# Learning rate is defined below
GAMMA = 0.99
# Setting the initial and final epsilon values
INITIAL_EPSILON = 1.0
FINAL_EPSILON = 0.05
# Number of frames for observation and exploration
EXPLORE = 500000
OBSERVE = 50000
# The following will be stored as experiences
REPLAY_MEMORY = 500000
# Batch Sixe - How many times we are going to train
BATCH = 100

# We are going to create a Tensorflow Graph
def tensorFlowGraph():
    # This is a CNN - Convolution Neural Network
    # We will feed Pixel Data and ScoreCounter into this neural network
    # Our Model has five layers. These layers are defined below

    # First Layer
    weight_CL1 = tensorflow.Variable(tensorflow.zeros([8,8,4,32]))
    bias_CL1 = tensorflow.Variable(tensorflow.zeros([32]))

    # Second Layer
    weight_CL2 = tensorflow.Variable(tensorflow.zeros([4,4,32,64]))
    bias_CL2 = tensorflow.Variable(tensorflow.zeros([64]))

    # Third Layer
    weight_CL3 = tensorflow.Variable(tensorflow.zeros([3,3,64,64]))
    bias_CL3 = tensorflow.Variable(tensorflow.zeros([64]))

    # Fourth Layer - Fully Connected Layer
    weight_FC4 = tensorflow.Variable(tensorflow.zeros([3136, 784]))
    bias_FC4 = tensorflow.Variable(tensorflow.zeros([784]))

    # Fifth Layer - Fully Connected Layer
    weight_FC5 = tensorflow.Variable(tensorflow.zeros([784,ACTIONS]))
    bias_FC5 = tensorflow.Variable(tensorflow.zeros([ACTIONS]))

    # Feed the Pixel Data into this input
    ip = tensorflow.placeholder("float",[None,84,84,4])

    #Every layer we will perform ReLU function
    CL1 = tensorflow.nn.relu(tensorflow.nn.conv2d(ip, weight_CL1,
strides=[1,4,4,1], padding = "VALID") + bias_CL1)
    CL2 = tensorflow.nn.relu(tensorflow.nn.conv2d(CL1, weight_CL2,
strides=[1,2,2,1], padding = "VALID") + bias_CL2)
    CL3 = tensorflow.nn.relu(tensorflow.nn.conv2d(CL2, weight_CL3,
strides=[1,1,1,1], padding = "VALID") + bias_CL3)
    CL3_Reshape = tensorflow.reshape(CL3,[-1,3136])
    FC4 = tensorflow.nn.relu(tensorflow.matmul(CL3_Reshape, weight_FC4) +
bias_FC4)
    FC5 = tensorflow.matmul(FC4,weight_FC5) + bias_FC5
    return ip, FC5

def deepQNetworkGrapTraining(ip, op, session):
    argmax = tensorflow.placeholder("float",[None,ACTIONS])
    groundTruth = tensorflow.placeholder("float",[None])
    action = tensorflow.reduce_sum(tensorflow.multiply(op, argmax),
reduction_indices=1)
    costFunction = tensorflow.reduce_mean(tensorflow.square(action -
groundTruth))
    training_step = tensorflow.train.AdamOptimizer(1e-6).minimize(costFunction)
    ppGame = PongNew.PingPong()
    Deque = deque()
    gameFrame = ppGame.returnPresentFrameInformation()
    gameFrame = cv2.cvtColor(cv2.resize(gameFrame, (84,84)), cv2.COLOR_BGR2GRAY)
    ret, gameFrame = cv2.threshold(gameFrame,1,255,cv2.THRESH_BINARY)
    inputTensor = numpy.stack((gameFrame, gameFrame, gameFrame, gameFrame),
axis=2)
    saver = tensorflow.train.Saver()
    session.run(tensorflow.initialize_all_variables())
    t = 0
    epsilonValue = INITIAL_EPSILON
```



```
    while(1):
        outputTensor = op.eval(feed_dict={ip:[inputTensor]})[0]
        argmaxTensor = numpy.zeros([ACTIONS])

        if(random.random() <= epsilonValue):
            maxIndex = random.randrange(ACTIONS)
        else:
            maxIndex = numpy.argmax(outputTensor)
        argmaxTensor[maxIndex] =1

        if(epsilonValue > FINAL_EPSILON):
            epsilonValue -= (INITIAL_EPSILON - FINAL_EPSILON)/EXPLORE

        rewardTensor, gameFrame = ppGame.returnNextFrameInformation(argmaxTensor)
        gameFrame = cv2.cvtColor(cv2.resize(gameFrame, (84,84)),
cv2.COLOR_BGR2GRAY)
        ret, gameFrame = cv2.threshold(gameFrame,1,255,cv2.THRESH_BINARY)
        gameFrame = numpy.reshape(gameFrame,(84,84,1))
        inputTensorNew = numpy.append(gameFrame, inputTensor[:,:,0:3], axis=2)
        Deque.append((inputTensor, argmaxTensor, rewardTensor, inputTensorNew))

        if len(Deque)>REPLAY_MEMORY:
            Deque.popleft()

        if(t>OBSERVE):
            newBatch = random.sample(Deque,BATCH)
            inputBatch = [a[0] for a in newBatch]
            argBatch = [a[1] for a in newBatch]
            rewardBatch = [a[1] for a in newBatch]
            inputTensorNewBatch = [a[1] for a in newBatch]
            groundTruthBatch =[]
            outputBatch = op.eval(feed_dict={ip:inputTensorNewBatch})

            for i in range(0, len(newBatch)):
                groundTruthBatch.append(rewardBatch[i]+ GAMMA *
numpy.max(outputBatch[i]))

            training_step.run(feed_dict={groundTruth: groundTruthBatch,
argmax:argBatch, ip:inputBatch})

        inputTensor = inputTensorNew
        t=t+1

        if(t%10000==0):
            saver.save(session,'./'+'pong'+'-dqn',globalStep = t)
```



```python
def main():
    session = tensorflow.InteractiveSession()
    ip,op = tensorFlowGraph()
    deepQNetworkGrapTraining(ip,op,session)

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

End of ReinforcedLearning.py