CSE - 691 Final Project Report

Deep Q Learning Pong Game

By

Akshay

Nishant Agrawal

Sushant Garg

Syed Yaser Ahmed Syed

Prof. Qinru Qiu

Machine Intell w/ Deep Learning

Syracuse University

December 14, 2017

TABLE OF CONTENTS

Introduction

The aim of this document is to explain the technicalities of a Deep Q Learning Pong game application. Here we have described a system that combines Deep neural networks with Reinforcement Learning(RL) to control agents using just the raw pixels as input. However, reinforcement learning presents several challenges from a deep learning perspective. This challenge can be overcome using a convolutional neural network to learn successful control policies in a complex RL environment.

Furthermore, this document demonstrates the architecture of Deep Q learning neural network, trained with a variant of the Q-learning algorithm. Here, the learning and testing has been done using Q function, learning rate & a variety of hyper parameters. The detailed analysis of memory requirement, complexity and structure of code has been done. The same network has later been extended to another game called Flappy bird.

Convolutional Neural networks consume a lot of time in calculations , A simple Q network is much faster. We have compared Convolutional networks and a simple Deep Q network and listed its advantages and disadvantages.

I. Reinforcement Learning

In reinforcement learning the aim is to help software agent to choose the actions to perform its tasks in an environment so as to maximize the rewards. The agent learns from experience using trial and error approach. So here we explore using:

- A set of environment & agent states
- Exploitation/exploration trade-off
- Markov Decision Process for learning tasks.

Another important aspect during training is the epsilon greedy strategy, which is a hyperparameter that facilitates the exploration/exploitation trade-off. In general, we start with a higher value of epsilon(between 0 & 1). A higher value of epsilon refers to more exploration than exploitation. So exploration is done more by taking risks by giving up on immediate rewards. As time passes during training, the epsilon value gradually decreases allowing more exploitation of rewards in the nearby states.

II. Markov Decision Process

A MDP has a set of actions A for the agent and a finite set of states S and has specified probabilities to transition between these states. There are rewards associated for each transition which are aggregated. The aim is to maximize this sum of rewards for the long term. We do this by taking the best possible action in each state.

III. Q-Learning

Q-learning is a model-free reinforcement learning technique. It evaluates which action to take based on Q function that determines the value of being in a certain state and taking a certain action at that state. It is used to find an optimal action-selection policy for a given Markov decision process (MDP). Policies are rules that agent follows while selecting actions. In this case, optimal policy is to select an action such that the quality is maximized, i.e. the q-function.

❖ *Q-Function*

It represents the quality of an action 'a' in a given state 's'. Here, π represents the policy such that the quality is maximized, this is done by getting the best possible score at the end of the game after performing an action. Just like with discounted future rewards, we can express the Q-value of state *s* and action *a* in terms of the Q-value of the next state *s'*.

$$\pi(s) = argmax_a\, Q(s,a)$$

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

❖ *Q-Learning algorithm*

- ➔ Choose an action 'a' to perform in the current state, 's'.
- ➔ Perform 'a' and receive reward R (s, a).
- ➔ Observe the new state, S (s, a).
- ➔ Update Q function
- ➔ If the next state is not terminal, go back to step 1.

IV. Application

The application must be capable of quickly filtering the frames. Our pong game runs at 24 fps, we take 4 frames each making it quicker to be classified by our convolutional layer. We then resize the image to 80x80 to further minimize the computational effort. The efficiency of the convolutional neural network to parse information even with degradation in quality makes it the best image classification algorithm for a game like pong.

❖ *Inputs and Outputs*

We take the last 4 frames of 80x80 pixels each, Convert it to grayscale and normalize the arrays. When passed through the CNN it gets reduced further to 512 floats. The output of the CNN will be 3 possible actions.

❖ *Network Architecture*

Our architecture uses three convolutional layers followed by a fully connected layer which is then resized to a 256. To further reduce computational effort we use a 2x2 max pool layer after every convolutional layer.

The following table explains the architecture:

Input	Layer	Output
80*80*4	8*8 Conv layer with strides 4	20*20*32
20*20*32	2*2 Max pool layer	10*10*32
10*10*32	4*4 Conv layer with strides 2	5*5*64
5*5*64	2*2 Max pool layer	3*3*64
3*3*64	2*2 Conv layer with strides 1	3*3*64
3*3*64	2*2 Max pool layer	2*2*64
2*2*64	Fully connected layer	256
256	Relu Activation	256
256	Output layer	3

❖ ***Learning and Testing***

To train the layer, author has used the below defined Loss function.

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right],$$

Here,

- ➔ y is the expected reward,
- ➔ Q(s,a) is the result of the Q-function,
- ➔ *theta* is the weight and bias used in the function

For this application, the following hyper parameters were used:

- ➔ learning rate was 1e-6,
- ➔ mini batch size of 100
- ➔ Decay rate of 0.99
- ➔ Initial random probability distribution of 1.0 (100%)
- ➔ Final random probability distribution of 0.05(.05%)

V. Analysis

Since we use CNN model the Deep Q network can efficiently train the AI by reducing computational effort required by each layer as it is the images is processed.

Below figure helps us visualize the convolutional networks better.

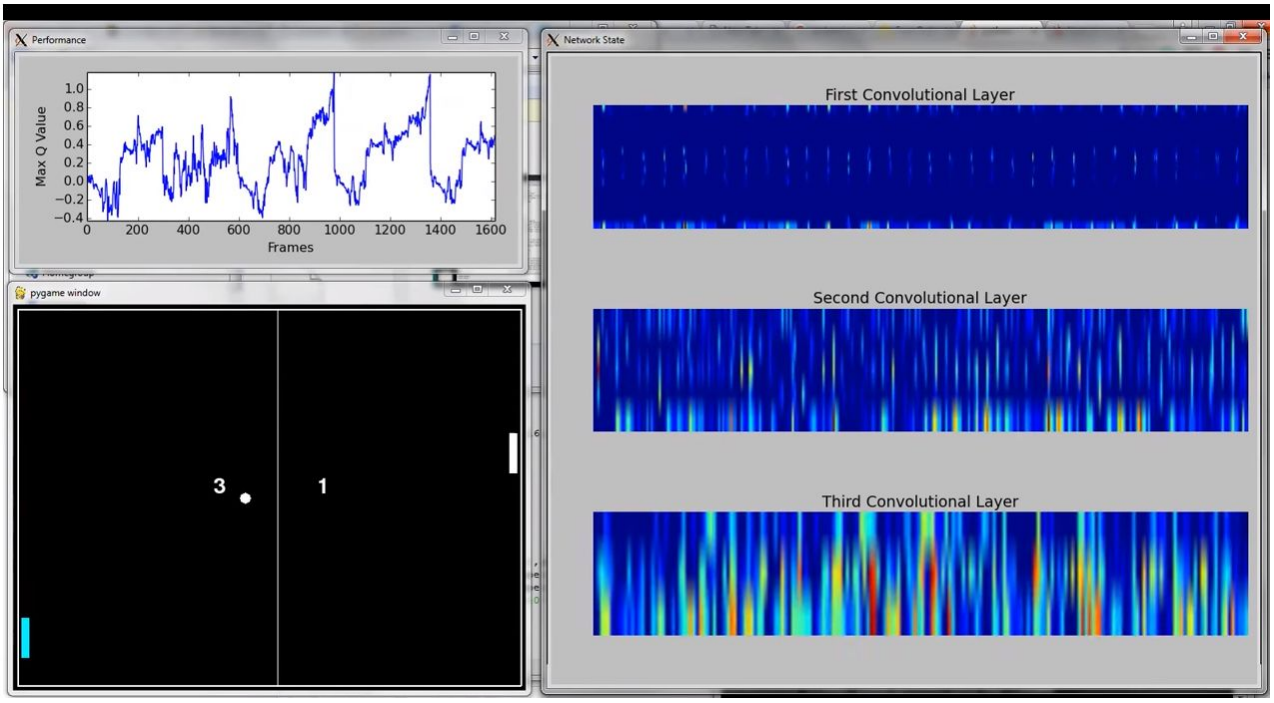


Fig:Visualization of Convolutional layers for pong

❖ ***Memory requirements***

Layer		Parameters
1st Layer	(8*8)+4)*4	272
Max Pool Layer	(2*2)+2)*32	192
2nd Layer	(4*4)+2)*64	1152
Max Pool Layer	(3*3)+2)*64	704
3rd Layer	(2*2)+1)*64	320
Max Pool Layer	(2*2)+2)*64	384

❖ ***Computational complexity***

Computational complexity is calculated with,

R x C x P

Where R = row of input,

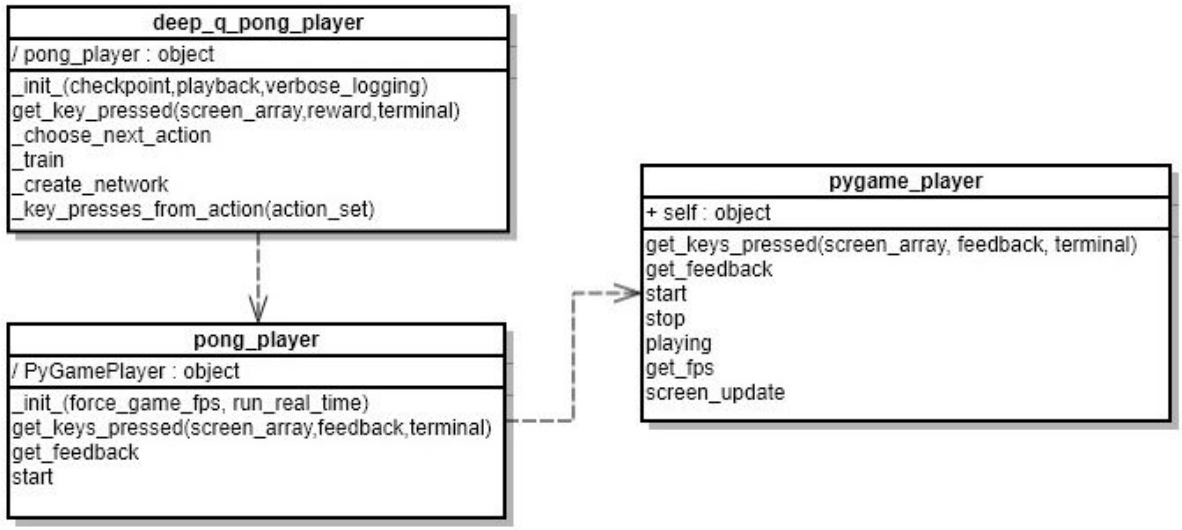
C = column of output,

P = Number of trainable parameters

Layers	Connections
Conv Layer 1	108800
Max Pool 1	19200
Conv Layer 2	28800
Max Pool 2	6336
Conv Layer 3	2880
Max Pool 3	1536

VI Structure of the Code

The application is a pong game modified to get its inputs from the Deep Q-network results. Key pressed function basically acts as a wrapper over the Deep-Q function constantly receiving the possible action to be taken by the AI.



The above figure shows the class diagram of the application. Three modules in total importing libraries like numpy, tensorflow, opencv and pygame.
Deep Q pong player has the application entry which will call functions derived from pong_player. Pong_player is the game engine behind the Deep Q layer.

There is an option to store the checkpoint of the trained AI. The learned weights will be reused to make it easier to track local minima and find ways to avoid them.

- ➔ get_key_pressed function will translate key press to action in the game and it is controlled by the deep Q network. Here the AI will get the input from key_presses_from_action function. This will make the paddle move up or down.
- ➔ Create_network is used to create the architecture for the model used in train
- ➔ train initiates the image classification algorithm where the CNN is used.
- ➔ Key_presses_from_action chooses an action from the 3 possible outputs from CNN.
- ➔ Choose_next_action will calculate an action to take next based on the random probability distribution.

VII. Improvement

Significant improvements on the current network architecture are possible. Our implementation of max pooling may have not been needed, as it caused the convolution kernels in the deeper layers to have a size comparable to that of the previous layer itself. Hence, max pooling may have discarded useful information. Given more time, it would be nice to perform more tuning over the various network parameters, such as the learning rate, the batch size, the replay memory, and the lengths of the observation and exploration phases. Additionally, we could check the performance of the network if the game parameters are tweaked (e.g. the angle of the ball's bounce is changed slightly). This would yield important information about the network's ability to adapt to new situations.

To check if our player is actually learning , we reset the score once it reaches 20.

After removing two max-pooling layers and connected the third Convolutional layer to the Fully Connected Layer, we find that the AI reaches convergence much faster than the previous experiment.The changes made is the code is just commenting of the maxpool layers and adjusting the parameters of the other layers.

```
# hidden layers
h_conv1 = tf.nn.relu(conv2d(s, W_conv1, 4) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2, 2) + b_conv2)
#h_pool2 = max_pool_2x2(h_conv2)

h_conv3 = tf.nn.relu(conv2d(h_conv2, W_conv3, 1) + b_conv3)
#h_pool3 = max_pool_2x2(h_conv3)

#h_pool3_flat = tf.reshape(h_pool3, [-1, 256])
h_conv3_flat = tf.reshape(h_conv3, [-1, 1600])

h_fc1 = tf.nn.relu(tf.matmul(h_conv3_flat, W_fc1) + b_fc1)

# readout layer
readout = tf.matmul(h_fc1, W_fc2) + b_fc2

return s, readout, h_fc1
```

Fig: Code changes for Improvement

The System reaches convergence after 10-14 hours, compared to the previous system which reached convergence after 25 hours(2.5million timesteps).

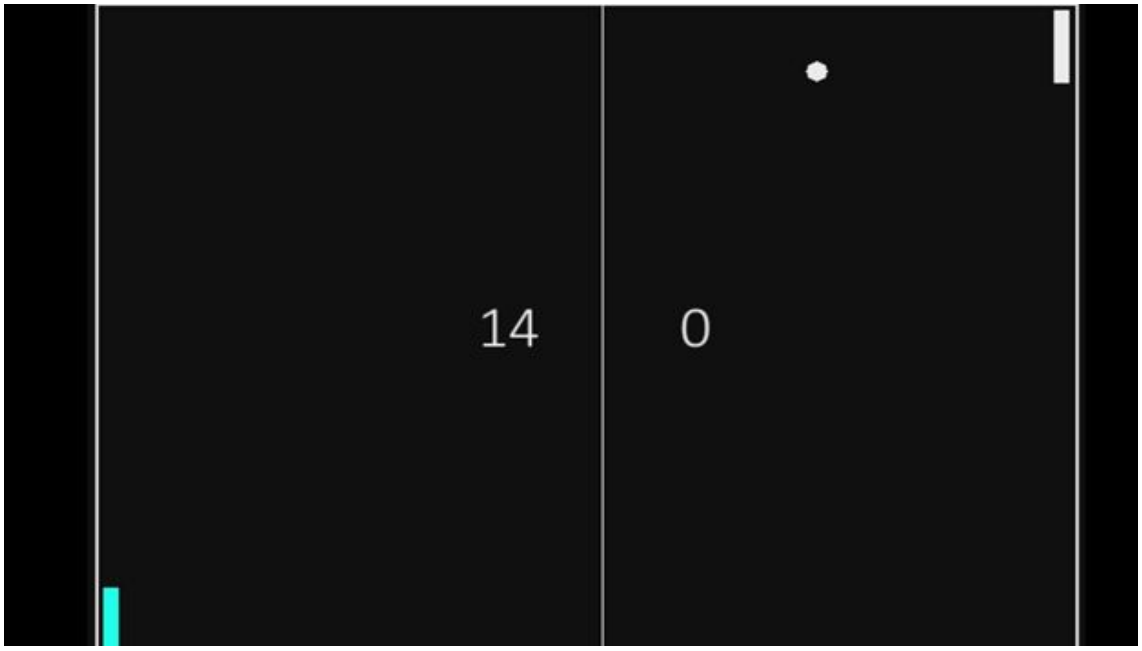


Fig: State of game after 700000th timestep

VIII. Innovation

❖ *Approach*

Convolution Neural Network based Video games take too Long to Learn on standard PCs/GPUs. This is because the information being entered is the entire environment (pixels) and calculation involved is a lot.

For example , The input size for the CNN network used above is 80*80*4 pixels (25600 pixels) . Each Convolutional layer and max pool layer reduce this and finally at the end we get an output (This output tells our player which direction to move in) .

If we can reduce the size of the input , the number of calculations will naturally be reduced.

❖ ***Network***

We removed the Convolutional network and introduced a simple 2 Layered Deep Q network with inputs as the following:

- ➔ Position of Paddle
- ➔ X direction of Ball
- ➔ Y direction of Ball
- ➔ X position of Ball
- ➔ Y position of Ball

The activation function used for each layer is Relu . The first layer contains 64 units and the second layer contains 32 units . Both are dense layers (Fully Connected) which are mapped to a linear output layer which outputs one of three actions :

- ➔ Move up
- ➔ Move down
- ➔ Don't move

We optimize the Mean Square error using the Adam Optimizer.

❖ ***Experiment Results:***

The system converged before 9000 timesteps which is extremely fast when compared to traditional CNN models. Below graph describes the Score (Q-function)

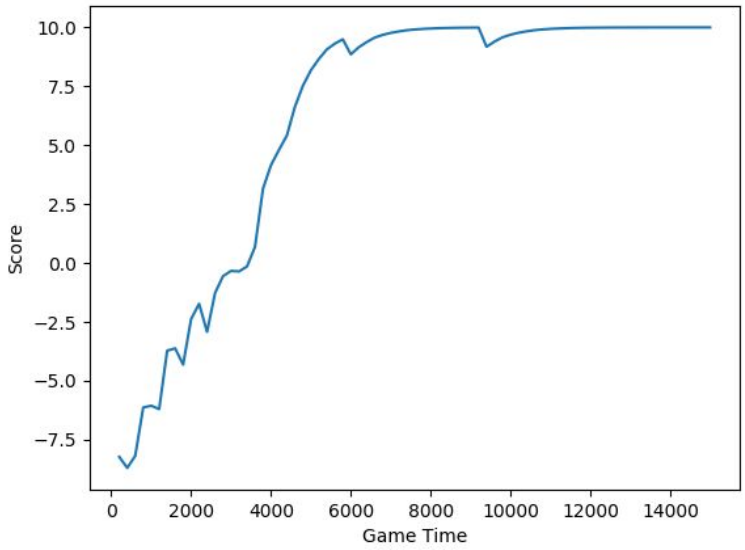


Fig: Score vs Time Map for Pong

- ***Advantages:***
  - ➔ Converges much faster than CNN networks (10000 timesteps)
  - ➔ Works on any GPU/CPU

- ***Disadvantages:***
  - ➔ Cannot be extended to other games

IX. Extension: Flappy Bird

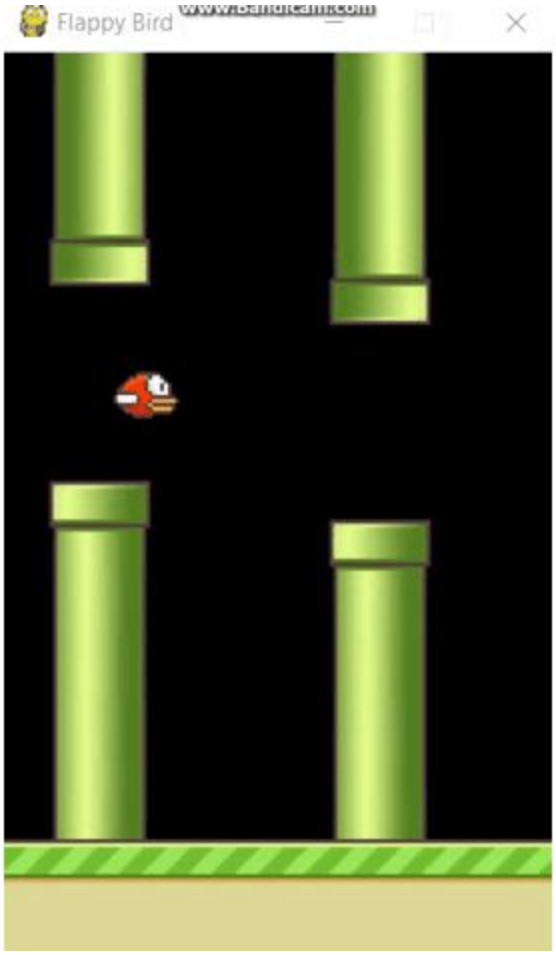


Fig. Screenshot of Model trained after 600k timesteps

❖ *Application*

The aim in the game is to keep the bird flying ahead avoiding all obstacles including the pipes and the walls. During the learning each time the bird collides, penalty is added as negative rewards allowing it to learn which action to take during run. Just like the pong game, we take the last 4 frames to make it easier to be classified by our convolutional layer network.

- ➔ Input: We take the last 4 frames of 80x80 pixels each, convert and normalize it to a grayscale image.
- ➔ Output: 2 possible actions: Flap (goes up) or not flap (goes down)

❖ ***Network Architecture***

Our architecture uses 6 hidden layers between input and Q layer which are as follows:

- ➔ Convolution with filter size 8, stride 4
- ➔ Avg. pool with filter size 2, stride 1
- ➔ Convolution with filter size 4, stride 2
- ➔ Convolution with filter size 3, stride 1
- ➔ 2 FC layers with Relu activation

The following table explains the architecture:

Input	Layer	Output
80*80*4	Conv layer- 8*8, stride 4	20*20*32
20*20*32	2*2 Avg. pool layer	10*10*32
10*10*32	Conv layer- 4*4, stride 2	5*5*64
5*5*64	Conv layer- 3*3, stride 1	4*4*64
4*4*64	FC layer with ReLU	512
512	FC layer with ReLU	100
100	Q layer	2

❖ ***Learning and Testing***

To train the layer, Q function is used

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Following are the HyperParameters used:

- ➔ Discount factor, γ =0.99
- ➔ Observe timesteps = 100000
- ➔ Explore timesteps = 200000
- ➔ Replay memory =50000
- ➔ Batchsize = 32
- ➔ Initial epsilon = 0.1
- ➔ Final epsilon = 0.0001
- ➔ Frames per sec = 1

❖ ***Memory requirements***:

Layer		Parameters
Conv layer 1	(8*8+4)*4	272
Avg. pool layer	(2*2+2)*32	192
Conv layer 2	(4*4+2)*64	1152
Conv layer 3	(3*3+1)*64	640

Layer	Connections
Conv layer 1	108800
Avg. pool layer	19200
Conv layer 2	28800
Conv layer 3	6336

❖ Code Structure:

In the code for flappy bird application, we define the deep Q network in DQN class. This class has methods to create and train the Q network and other helper methods. FlappyBirdDeepQN is where the execution starts, where the input is preprocessed by the preprocess() method and then the game is started in the playFlappyBird() method. The methods for both DQN class and the executive are given under with their functionalities:

- ➔ createQNW() - creates the Q network
- ➔ getAction() - returns the derived action, called in the playFlappyBird() method
- ➔ setInitState() - sets the initial state, called in the playFlappyBird() method
- ➔ weight_var() - helper method called in createQNW() to define weights for the various layers.
- ➔ bias_var() - helper method called in createQNW() to define biases for the layers.
- ➔ conv2d() - helper method do the computation at the convolution layers.
- ➔ avg_pool_2x2() - another helper method to compute at the pool layer.
- ➔ copyTargetQNW() - starts session by using a copy of the target Q network operation.
- ➔ createTrainingMethod() - sets actionInput and yInput, defines the Q action and cost function to be used for training
- ➔ trainQNW() - does the training for the network by getting a random minibatch and saves the weights for every 100000 iterations.
- ➔ setPerception() - calls the trainQNW() and sets the network perception, i.e whether to observe, explore or train.
- ➔ preprocess() - preprocesses the input to a 80*80 grey-scale image.
- ➔ playFlappyBird() - starts the game, sets the initial state and start training the network.

X. Summary

- ❖ Understood the basic application requirements and the input/output data format as well
  - Input is 4 frames of 80x80 normalized grayscale.
  - Output is 3 possible actions.
- ❖ Studied the architecture of Deep Q learning neural network used for this application
  - Application uses 3 convolutional layers with 2x2 max pool layers after every layer.
  - A 256 fully connected layer after the last max pool layer.
  - A flatten layer to get the 3 possible actions.
- ❖ Learned and performed testing using loss function, learning rate & a variety of hyper parameters
  - Learning rate used was 1e-06 and Epsilon of 0.99 was used.
- ❖ Analyzed memory requirement, complexity and structure of code.
- ❖ Able to run code successfully & demonstrated the same.
- ❖ Extended the application by using it in Flappy Bird game.
- ❖ Retrained & Improved the network.
- ❖ Completely removed Convolutional layers and made new Q network.

XI. References

1. https://machinelearningmastery.com/
2. https://en.wikipedia.org/wiki/Reinforcement_learning
3. https://github.com/asrivat1/DeepLearningVideoGames
4. https://github.com/llSourcell/pong_neural_network_live
5. https://jaromiru.com/2016/10/03/lets-make-a-dqn-implementation/
6. https://www.youtube.com/watch?v=V1eYniJ0Rnk
7. https://github.com/yenchenlin/DeepLearningFlappyBird
8. https://medium.com/machine-learning-for-humans/reinforcement-learning-6eacf258b265