March 2018

Using Deep Q-Learning to create a Bot to Play Flappy Bird

PROJECT REPORT

Submitted for CAL in B.Tech Artificial Intelligence(CSE3013)

By

Tushar Pahuja **15BCE1252**

Osho Agyeya **15BCE1326**

Name of faculty: **Dr. R. Bhargavi**

(SCHOOL OF COMPUTING SCIENCE AND ENGINEERING)

CERTIFICATE

This is to certify that the Project work entitled "***Using Deep Q-Learning to create a Bot to Play Flappy Bird***" that is being submitted by "***TUSHAR PAHUJA, OSHO AGYEYA***" for CAL in B.Tech Artificial Intelligence(CSE3013) is a record of bonafide work done under my supervision.

Place: Chennai
Date: 3/04/2018

Signature of Students:

TUSHAR PAHUJA

OSHO AGYEYA

Signature of Faculty:

Dr. R. BHARGAVI

ACKNOWLEDGEMENTS

We thank VIT University (**SCHOOL OF COMPUTING SCIENCE AND ENGINEERING**) for giving us the opportunity to conduct this project and experiment. We also thank our guide for project Dr. R Bhargavi for her constant, good and knowledgeable guidance for the project. Through this project, we learned many of new things about Artificial Intelligence and Deep Learning which will be definitely useful for us.

OSHO AGYEYA
Reg. No. 15BCE1326

TUSHAR PAHUJA
Reg. No. 15BCE1252

ABSTRACT

Deep Q-Network is a learning algorithm developed by Google DeepMind to play Atari Games. It showed how a computer learned to play Atari Video Games by just observing screen pixels and receiving a reward when the game score increased. The algorithm was generic enough to play various games.

The main aim of the project is to study the concepts of Deep Reinforcement Learning and use the learned concepts to build a bot which can play Flappy Bird. Reinforcement learning lies somewhere in between supervised and unsupervised learning. Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all, in reinforcement learning one has sparse and time-delayed labels – the rewards. Based only on those rewards the agent has to learn to behave in the environment.

The project demonstrates the training of the built bot over various environments and how the bot improves with excessive training and ultimately attains an average score of 20-30 in the game.

UNDERLYING PRINCIPLES

Reinforcement Learning

Suppose you want to teach a neural network to play this game. Input to your network would be screen images, and output would be three actions: left, right or fire (to launch the ball). It would make sense to treat it as a classification problem – for each game screen you have to decide, whether you should move left, right or press fire. Sounds straightforward? Sure, but then you need training examples, and a lot of them. Of course, you could go and record game sessions using expert players, but that's not really how we learn. We don't need somebody to tell us a million times which move to choose at each screen. We just need occasional feedback that we did the right thing and can then figure out everything else ourselves. This is the task which Reinforcement Learning tries to solve.

Credit Assignment Problem

One of the initial steps in the construction of a bot is to determine which of the preceding actions was responsible for getting the reward and to what extent. This is generally referred to as the **credit assignment problem.**

Exploration - Exploitation

The game bot built also considers the principle of Exploration and Exploitation.

Exploitation refers to using learned knowledge to predict future actions. This depends on the parameters of the model learned over time.

Exploration refers to making random actions to prevent the algorithm from falling into a local minima/maximum, not sticking to the current best solution and randomly trying new actions in order to attain a better solution.

Markov Decision Process

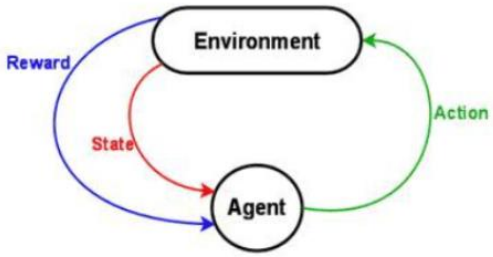
A Markov decision process is a 5-tuple $(S, A, P_\cdot(\cdot,\cdot), R_\cdot(\cdot,\cdot), \gamma)$ where –

- S is a finite set of states,
- A is a finite set of actions from a particular state s.
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time t+1,

- $R_a(s, s')$ Is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a.
- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between future rewards and present rewards.

The most common way of representing a reinforcement learning problem is to represent is as a Markov Decision process.

Suppose you are an agent, situated in an environment (e.g. Breakout game). The environment is in a certain state (e.g. location of the paddle, location and direction of the ball, existence of every brick and so on). The agent can perform certain actions in the environment (e.g. move the paddle to the left or to the right). These actions sometimes result in a reward (e.g. increase in score). Actions transform the environment and lead to a new state, where the agent can perform another action, and so on. The rules for how you choose those actions are called policy. The environment in general is stochastic, which means the next state may be somewhat random (e.g. when you lose a ball and launch a new one, it goes towards a random direction).



Discounted Future Reward

Because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions. The more into the future we go, the more it may diverge. For that reason it is common to use discounted future reward instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

Here γ is the discount factor between 0 and 1 – the more into the future the reward is, the less we take it into consideration. It is easy to see, that discounted future reward at time step t can be expressed in terms of the same thing at time step t+1:

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

If we set the discount factor γ=0, then our strategy will be short-sighted and we rely only on the immediate rewards. If we want to balance between immediate and future rewards, we should set discount factor to something like γ=0.9. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor γ=1.

A good strategy for an agent would be to **always choose an action that maximizes the (discounted) future reward.**

MODEL USED

We have used the standard network architecture that DeepMind have given in their research paper. This is a classical **Convolutional Neural Network** with three convolutional layers, followed by two fully connected layers. We have not used any pooling layers because we know that they lead to translation invariance and the network becomes insensitive to the location of an object in the image. That makes perfectly sense for a classification task like ImageNet, but for games the location of the ball is crucial in determining the potential reward and we wouldn't want to discard this information.

Input to the network are four 84×84 grayscale game screens. Outputs of the network are Q-values for each possible action (2 in case of Flappy Bird). Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss.

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

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

Q LEARNING ALGORITHM

In Q-learning we define a function Q(s, a) representing the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$Q(s_t, a_t) = max\ R_{t+1}$$

The way to think about Q(s, a) is that it is "the best possible score at the end of the game after performing action a in state s". It is called Q-function, because it represents the "quality" of a certain action in a given state.

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

Here π represents the policy, the rule how we choose an action in each state.

OK, how do we get that Q-function then? Let's focus on just one transition <s, a, r, s'>. Just like with discounted future rewards in the previous section, we can express the Q-value of state s and action a in terms of the Q-value of the next state s'.

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

This is called the Bellman equation. If you think about it, it is quite logical – maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state.

The main idea in Q-learning is that we can iteratively approximate the Q-function using the Bellman equation. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns.

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

α in the algorithm is a learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value is taken into account. In particular, when α=1, then two Q[s,a] cancel and the update is exactly the same as the Bellman equation.

The maxa' Q[s',a'] that we use to update Q[s,a] is only an approximation and in early stages of learning it may be completely wrong. However, the approximation get more and more accurate with every iteration and it has been shown, that if we perform this update enough times, then the Q-function will converge and represent the true Q-value.

The final Deep Q-Learning Algorithm used –

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = 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 + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

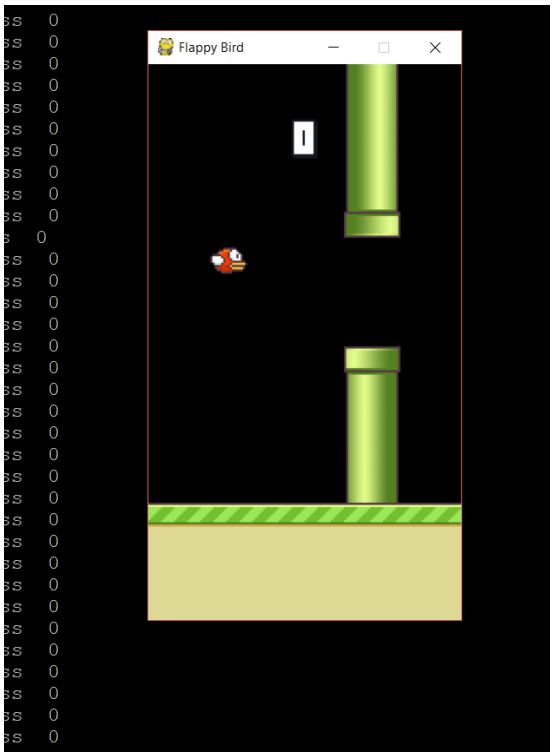
IMPLEMENTATION IN PYTHON

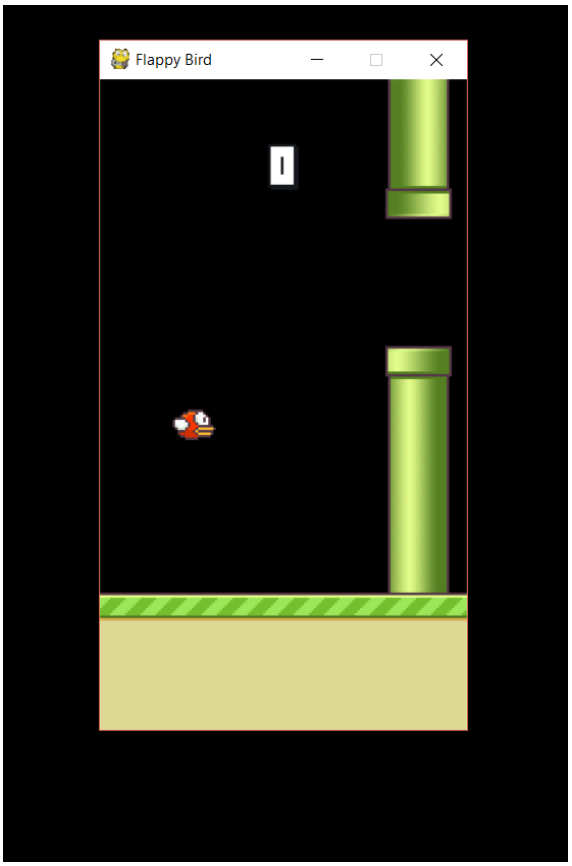
```
    x_t1_colored, r_t, terminal = game_state.frame_step(a_t)

    x_t1 = skimage.color.rgb2gray(x_t1_colored)
    x_t1 = skimage.transform.resize(x_t1,(80,80))
    x_t1 = skimage.exposure.rescale_intensity(x_t1, out_range=(0, 255))

    x_t1 = x_t1.reshape(1, x_t1.shape[0], x_t1.shape[1], 1) #1x80x80x1
    s_t1 = np.append(x_t1, s_t[:, :, :, :3], axis=3)

    # store the transition in D
    D.append((s_t, action_index, r_t, s_t1, terminal))
    if len(D) > REPLAY_MEMORY:
        D.popleft()

    #only train if done observing
    if t > OBSERVE:
        #sample a minibatch to train on
        minibatch = random.sample(D, BATCH)



        inputs = np.zeros((BATCH, s_t.shape[1], s_t.shape[2], s_t.shape[3]))   #32, 80, 80, 4
        print (inputs.shape)
        targets = np.zeros((inputs.shape[0], ACTIONS))                         #32, 2

        #Now we do the experience replay
        for i in range(0, len(minibatch)):
            state_t = minibatch[i][0]
            action_t = minibatch[i][1]   #This is action index
            reward_t = minibatch[i][2]
            state_t1 = minibatch[i][3]
            terminal = minibatch[i][4]
            # if terminated, only equals reward

            inputs[i:i + 1] = state_t    #I saved down s_t

            targets[i] = model.predict(state_t)  # Hitting each buttom probability
            Q_sa = model.predict(state_t1)

            if terminal:
                targets[i, action_t] = reward_t
            else:
                targets[i, action_t] = reward_t + GAMMA * np.max(Q_sa)

        # targets2 = normalize(targets)
        loss += model.train_on_batch(inputs, targets)

    s_t = s_t1
    t = t + 1
```

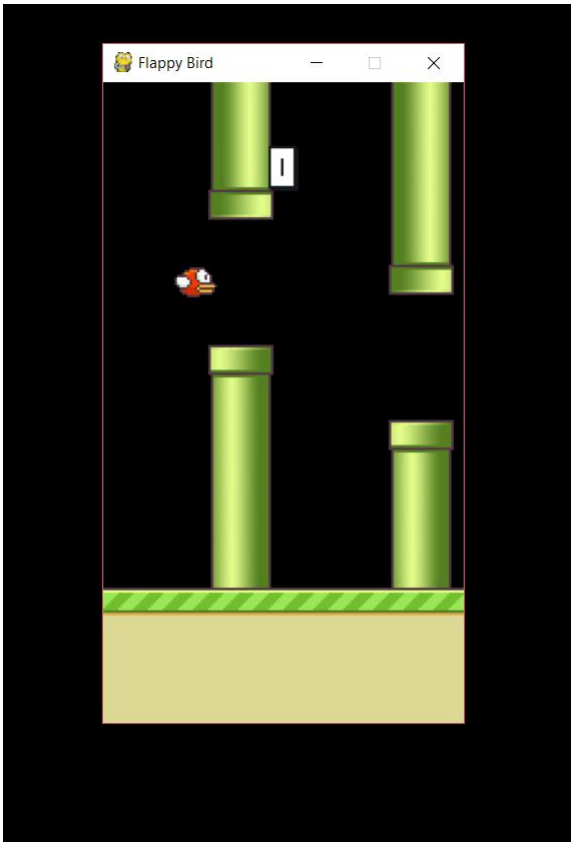
SCREENSHOTS

Training the Model

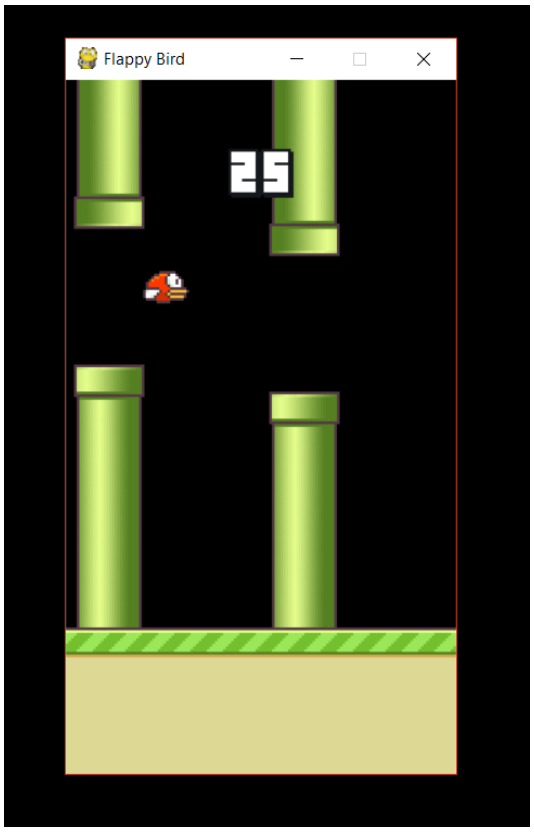
```
C:\Windows\system32\cmd.exe - python  qlearn.py -m "Run"
TIMESTEP 1961 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1962 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1963 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD -1 / Q_MAX   0 / Loss  0
TIMESTEP 1964 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1965 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1966 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
----------Random Action----------
TIMESTEP 1967 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1968 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1969 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
----------Random Action----------
TIMESTEP 1970 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1971 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1972 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1973 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1974 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1975 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1976 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1977 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1978 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1979 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1980 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1981 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
----------Random Action----------
TIMESTEP 1982 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1983 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1984 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1985 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
----------Random Action----------
TIMESTEP 1986 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1987 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1988 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1989 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1990 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1991 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1992 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1993 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1994 / STATE observe / EPSILON 0.1 / ACTION 1 / REWARD 0.1 / Q_MAX   0 / Loss  0
TIMESTEP 1995 / STATE observe / EPSILON 0.1 / ACTION 0 / REWARD 0.1 / Q_MAX   0 / Loss  0
```

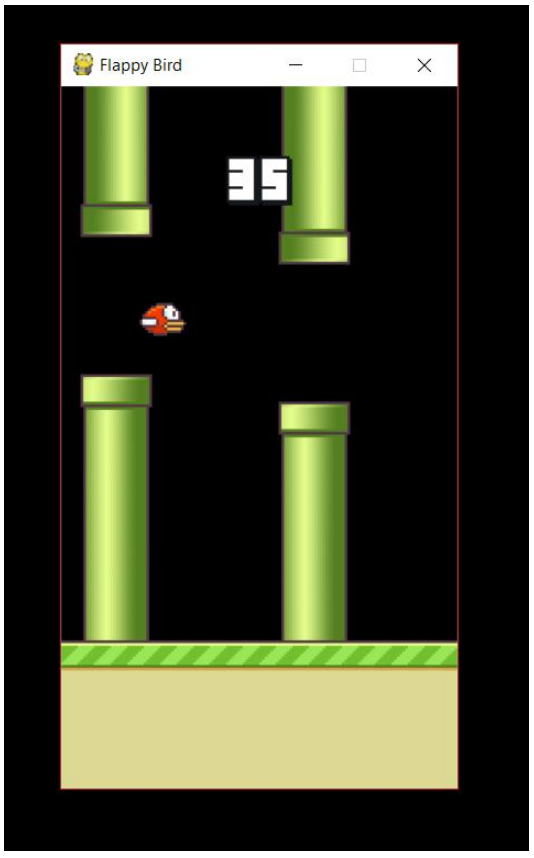


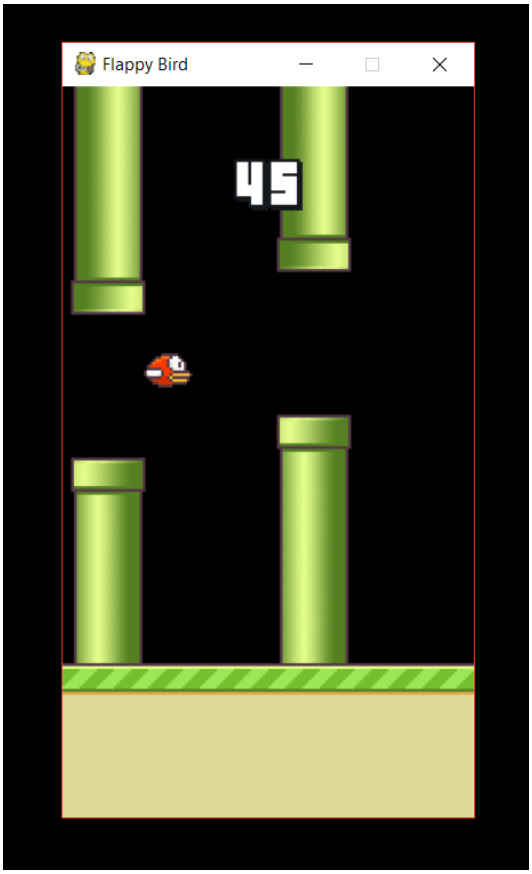

Flappy Bird


Flappy Bird

Running the trained Model







CONCLUSION

The final model after training for 2-3 days is able to play the Flappy Bird game effectively reaching a game score of around 20-50.

This illustrates the effectiveness of Q-Learning algorithm in learning the game environment predicting the best possible action at a particular time step in order to maximize the future discounted reward.

REFERENCES

[1] https://ai.intel.com/demystifying-deep-reinforcement-learning/
[2] http://sarvagyavaish.github.io/FlappyBirdRL/
[3] https://threads-iiith.quora.com/Neuro-Evolution-with-Flappy-Bird-Genetic-Evolution-on-Neural-Networks