Logo, company name

Description automatically generated

**Reinforcement Learning Project**

**Flappy Bird Game with Double Deep Q-Learning using custom environment**

**by/**

**Ahmed Salama**

**Mahmoud Khorshed**

**Zeyad Mohamed**

**1. Introduction**

Flappy Bird is a classic game that involves a bird that must manipulate its way through pipes to stay alive. Using the Reinforcement Learning technique with **Double** **Deep Q-learning** Algorithm, we succeeded to learn the game to play without humans. With the addition of convolutional neural networks, the traditional performance of game-playing programs has seen a great improvement that can even go beyond the ability of even expert-level human players. The novelty thing is that we make our custom environment based on OpenAI gym approach. One of the core challenges with computer vision is obtaining enough data to properly train a neural network, and OpenAI Gym provides a clean interface with dozens of many different environments.

**2. Custom Environment**

Using the Pygame package engine we created a traditional flappy bird game in which there is a bird that tends to pass a maximum number of pipes without any collisions with them or not touch the ground or sky.

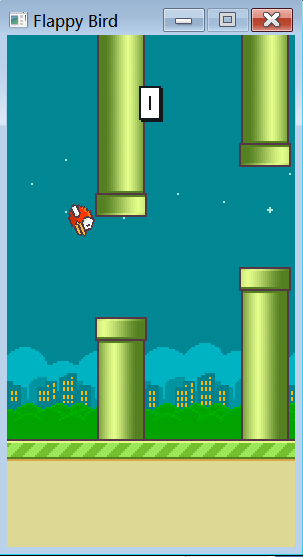
A sample image from a frame generated by the OpenAI Gym environment is shown in figure [1].

Figure 1. Sample frame of Flappy Bird.

The action of 1 causes the bird to go up, while 0 allows gravity to drag it down.

We merged that game with OpenAI Gym to make an agent that can be able to play to be live as possible he can. Our observation space is **864** (screen width) \* **836** (screen height) \***3** (number of channels RGB) representing the pixel values of the frame of the image while the minimum value is zero and the maximum value is **255**. The input to our model will be a sequence of pixel images as arrays (Width x Height x 3) generated by a particular OpenAI Gym environment. The action space is two discrete values (**FLAP** or **IDLE**) **FLAP** means to go up and **IDLE** means do nothing (following the gravity). If the bird takes an action that makes him alive, he will take a reward of **+1** otherwise **-10** If Flappy Bird collides with a pipe or touch the ground.

**3. Input Preprocessing**

Using gym.Wrappers we take the inputs from the environment Once we receive the image from the Gym environment, we apply a couple of steps to format the image to finally use the input to the model. First**, we convert the image to grayscale**, which should reduce complexity. It also reduces the size of the third dimension from 3 for RGB to 1. Next, we **De-noise the image using adaptive thresholding using the CV library**. In Flappy Bird, the background contains some unnecessary pixels that can create some noise for the neural network, such as stars in the night background. Applying **adaptive Gaussian thresholding (CITE)** incorporates each pixel’s neighboring values to determine whether its value is noise or not. This same algorithm generalizes well format other games. as shown in figure [2].

A picture containing text

Description automatically generated

Afterward, we normalize values to be between 0 and 1 and reduce the image to 84\*84 pixels. This allows for consistent input sizes across games as well as decreasing dimensionality of each layer of our network, allowing for faster passes and fewer parameters per layer. Finally, we **stack the last 4 frames** as input to the network using **Gym. Wrappers** to form the 84x84x4 input to the network. For the reward values.

**4. Approach**

**4.1. Neural Network Model**

Our implementation of DQN is using the Pytorch library, and our models are running on Local machines with 8 cores and an Nvidia GeForce RTX 3060 GPU. The Network architecture is outlined in table [1].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Network Architecture | | | | |
| Type | filters | Filter size | stride | activation |
| Conv1 | 32 | 8\*8 | 4 | Relu |
| Conv2 | 32 | 4\*4 | 2 | Relu |
| Conv3 | 64 | 3\*3 | 1 | Relu |
| Fully | 512 |  |  | Relu |
| fully | N \*action |  |  | linear |

Table 1. Details of layers of our current network, based on the DQN model with a larger number of filters in each layer. 

We are using 3 convolution layers with ReLU non-linearity activations, followed by 2 fully connected layers. Fully connected layers are separated by ReLU function like convolution layers. **The final output layer dimension is equal to the number of valid actions allowed in the game**. The values at this output layer represent the Q function given the input state for each valid action. At startup, we also initialize replay memory to 1000 observations. At beginning of training, we first populate the replay memory by choosing random actions for 1000 steps and we are not updating network weights during this preliminary training step. Once the replay memory buffer is partially filled, we start training. We use the **Pytorch Adam optimization** algorithm with an initial learning rate of 0.001 and we tried many learning rates.

**4.2 Deep Q network**

To grasp our DDQN, we must first define DQN. Neural networks are exceptional at learning characteristics from highly structured data. Our Q function is represented by a neural network, which takes the state (the last four game frames stacked) and action as input and returns the associated Q-value. This solution has the advantage of allowing us to execute Q value updates or select the action with the highest Q-value with a single forward pass via the network and have all Q values for all actions available immediately. Input to the network is 4 stacked frames of 84x84 grayscale game screens. The output of the network is Q-values for each possible action (2 for Flappy Bird and Pixel Copter). Q-values are continuous values which makes it a regression task that can either be optimized by using **Huber loss** or **L2** loss.

**4.3. Replay Memory**

This can be coined as the most important trick to getting the network to converge. Due to the stochastic nature of the gameplay, approximating Q values using nonlinear functions is not very stable. Replay memory is one of the tricks that help in converging the network and making learning stable. During gameplay, all the experiences < s; a; r; s > are stored in a replay memory of fixed size D = 100000 using (Deque). During training, random mini batches from replay memory are sampled and this helps in breaking the similarity of continuous training samples and helps drive the network towards a local minimum. Replay memory makes the training process like supervised learning and simplifies debugging and testing of the algorithm.

**4.4. Exploration vs Exploitation**

At the beginning of the training cycle predictions of Q network are random due to random initialization of the network. Agent tends to perform exploration in which it tries various actions and observe rewards for these actions. As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. Q-learning incorporates the exploration as part of the algorithm. But this exploration is greedy, it settles with the first effective strategy it finds. A simple and effective fix to this problem is to introduce a hyperparameter epsilon which determines the probability to choose between exploration or exploitation. Decreases overtime from 1 to 0.1 in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed epsilon value.

**4.5 Target Network**

Before we start into DDQN, it's a good idea to grasp the logic behind target networks. During training, a target network is utilized to generate the target-Q values that will be used to compute the loss for each action. Using one network for both estimations is unstable because Q-networks values vary, and if we use a continually shifting set of values to adjust our network values, the value estimations can quickly spiral out of control. Destabilization of the network can occur as a result of feedback loops between the target and estimated Q-values. To reduce this risk, the target network weights are fixed and only periodically updated to the primary Q-network values. Training can then proceed in a more stable manner.

**4.6 Double DQN**

The **Double** **Q learning** algorithm underpins **DQN**, and the purpose for this algorithm is the actions to perform in each state. If all actions were overstated, this would not have been a problem, but that is not always the case. To correct this, the inventor of **DDQN** recommended a simple trick: Instead of determining the **target-Q** value for the training phase using the max over **Q-values**, use your primary network to select an action and your target network to generate the target **Q-value** for that action. As a result, the **Q target** equation can be written as

**y=reward+Qtarget (s0;argmax(Q(s0; a0)))**

**Text, letter

Description automatically generated**

**5. Experiments**

We measure the performance of a particular model using metrics provided by the OpenAI Gym interface, namely the cumulative reward over 10 episodes for train and mean cumulative reward over 5 episodes for the test of a particular environment. We will explore different reinforcement learning techniques as well as experiment with the tuning of hyperparameters (Gamma, Learning rate, Number of episodes).

**Note: we take the maximum number of steps as the optimal value not the minimum number as written in the project instructions file because our goal here is to make the agent takes maximum number of steps, increasing in the number of steps means he was alive for a long time and he didn’t die so fast.**

**First trial curves: -**

Gamma =0.85

Learning rate = 0.05

Number of episodes = 3000

Fig: number of steps per test episodes.

Chart

Description automatically generated

Fig: number of steps per train episodes.

Chart, bar chart

Description automatically generated

Fig: cumulative mean reward per test episodes.

Chart, bar chart

Description automatically generated

Fig: cumulative reward per train episodes.

Chart, bar chart

Description automatically generated

**Result conclusions: -**

We have noticed that

* Number of test steps start to increase after 900 episodes and starts to get better.
* Number of training steps starts to get better from the first episode.
* Mean cumulative reward per each test episode gets higher after 400 test episodes.
* Cumulative reward per each train episode gets higher after 500 train episodes.

**Second trail:-**

Gamma =0.8

Learning rate = 0.01

Number of episodes = 7000

Fig: number of steps per test episodes.

Chart, bar chart

Description automatically generated

Fig: number of steps per train episodes.

Chart, histogram

Description automatically generated

Fig: cumulative mean reward per test episodes.

Chart, bar chart

Description automatically generated

Fig: cumulative reward per train episodes.

Chart, histogram

Description automatically generated

Fig: q value reward per train episodes.

Chart, histogram

Description automatically generated

**Result conclusions: -**

We have noticed that

* Number of test steps start to increase after 300 episodes and starts to get better
* Number of training steps starts to get better from the first episode.
* Mean cumulative reward per each test episode gets higher after 400 test episodes.
* Cumulative reward per each train episode gets higher after 500 train episodes.
* Q value per each train episode gets higher after 1500 episodes.

**Last trail:-**

Gamma =0.8

Learning rate = 0.01

Number of episodes = 7000

Fig: number of steps per test episodes.

Chart, bar chart

Description automatically generated

Fig: number of steps per train episodes.

Chart, bar chart

Description automatically generated

Fig: cumulative mean reward per test episodes.

Chart, bar chart

Description automatically generated

Fig: cumulative reward per train episodes.

Chart

Description automatically generated

Fig: q value reward per train episodes.

Chart, line chart

Description automatically generated

**Result conclusions: -**

We have noticed that

* Number of test steps start to increase after 900 episodes and starts to get better
* Number of training steps starts to get better from the first episode.
* Mean cumulative reward per each test episode gets higher after 300 test episodes and doesn’t increase that much after that
* Cumulative reward per each train episode gets higher from the first train episode.
* Q value per each train episode gets higher after 1000 episodes.

**6. Final thoughts**

From the above plots the worst hyperparameters was in the first trial since they took many episodes to converge, second and last trial results was near to each other, although the last trial is a little bit better, the main key here if we want to make flappy an intelligent agent we should increase the number of training episodes as much as we can, that’s the most important hyperparameter here, but in terms of convergence speed the alpha and gamma play an important rule more than the number of training episodes.