Playing Flappybird using Deep Q-Networks

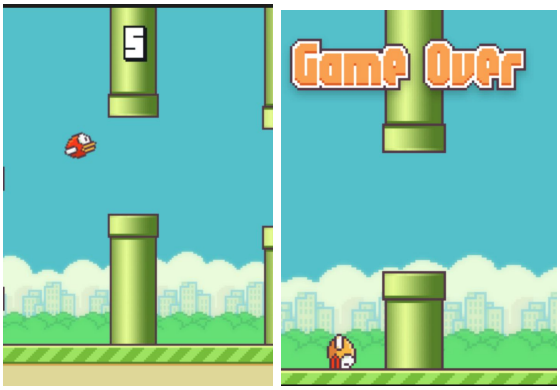
Masashi Omori
1/29/2018

1. Introduction

Starting with the success of DeepMind's artificial intelligence playing Atari games at superhuman skill levels, and the recent success of DeepMind's AlphaGo beating the best go players in the world and OpenAI's bot which plays Dota 2 (Defence of the Ancients) better than world class players, unsupervised machine learning is garnering more attention than ever before. These leaps in machine learning is a big step towards creating artificial general intelligence (AGI) since the bots learned through experience what the best moves to reach optimal rewards. The brain behind these bots are commonly known as policy agent which is a machine learning model that predicts the best action to take given a state of the game. The policy agent learns these actions based on experience alone with minimal human interaction, thus sometimes creating series of moves which no one in the world thought about. With that, the goal of this capstone project is to create deep neural network that will learn and play the game Flappybird from experience. Flappybird will be emulated based on PyGame Learning Environment (PLE) and a wrapper around PLE.

2. Flappybird Analysis

The game of Flappybird has a very simple setup. It is an automatic side scrolling game where the user controls a bird and maneuvers it through the moving environment. There are random pipes which are setup and the user must navigate above/below/between them to avoid contact. By default, the altitude of the bird decreases, and the user can tap on the screen to let it 'flap,' increasing its altitude for a brief moment. Repeated taps will result in steady incline of the bird's location. Hitting any of the pipes or the top/bottom of the game screen will result in termination of the trial, and score is calculated based on how far the bird was able to travel.



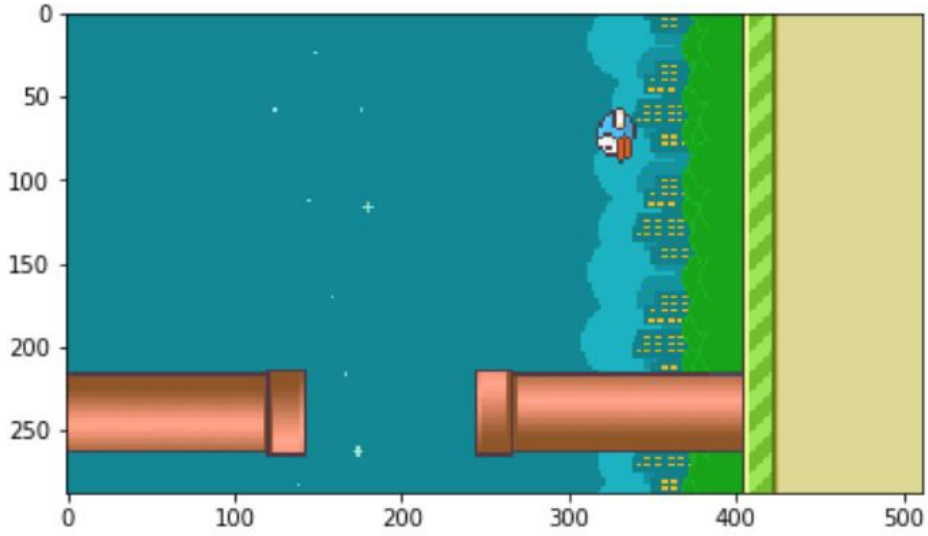
Each frame of the game is setup as a 512 by 288 by 3 dimension matrix of floating numbers, where the last axis accounts for the RGB representation. There are also the allowed actions for this game, which is to move up or down. The pygame library encodes these as 119 (up) or None (down), but for simplicity we've changed it so the actions are encoded in an array of length 2 where one of the value is 1 and the other is 0. If the first index is 1, then we do nothing, and if the second index is 1, then the bird moves up.

Also, we consider the different reward points allowed during the game. From this article, it is recommended (by DeepMind also) to clip the reward values between -1 and 1. Hence, we define the following reward values:

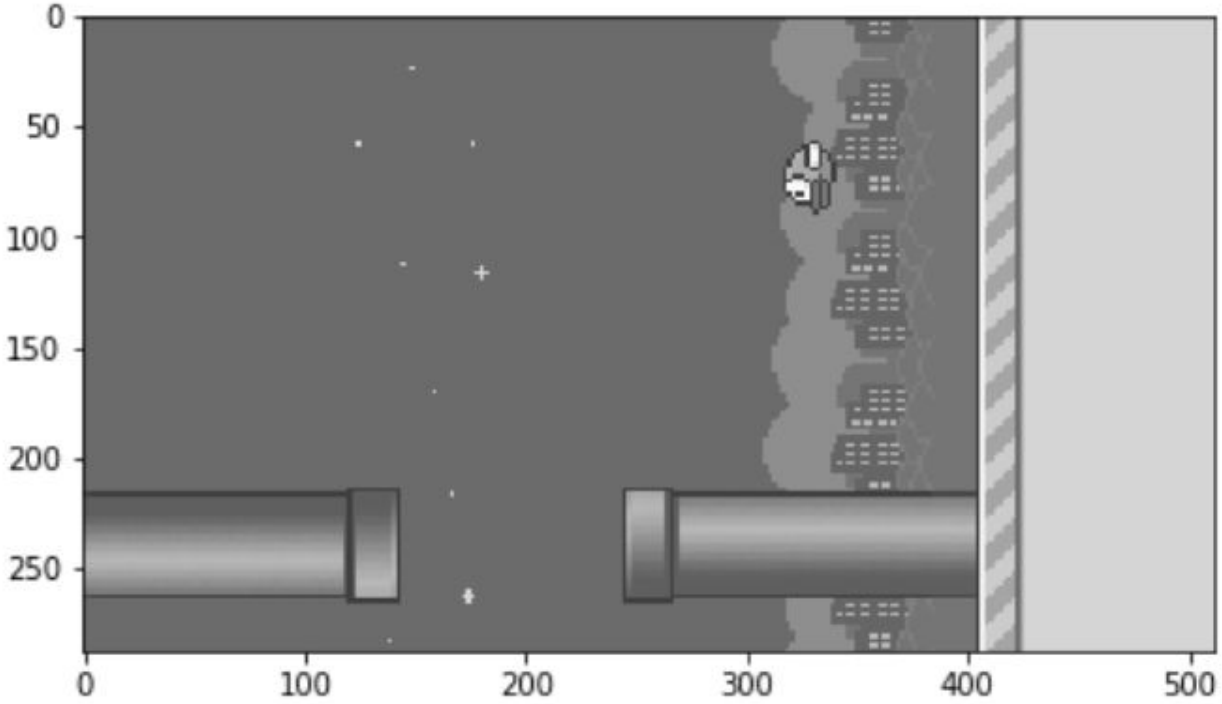
- Each frame survived -> 0.1 points
- Went through a pipe -> 1.0
- Reached game over -> -1.0

3. Data Wrangling

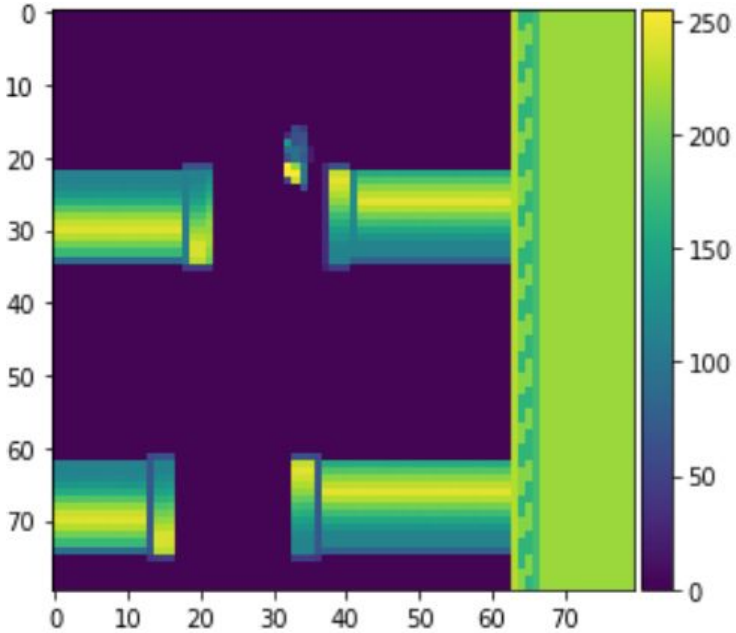
In order to speed up and stabilize training the policy agent, we aim to remove unnecessary information from the state of the game. As a reminder, the original state looks as below, a 512 by 288 by 3 pixel:



We first reduce the dimension of the image by converting it to grayscale, which yields a matrix of just 512 by 288.



Then, the background as blacked out as it is different between day and night time, and reduce the image size to a 80 by 80 matrix. Further the pixel intensity is changed to a range of 0 and 255, which results in the below processed image which is 80 by 80.



One thing to keep in mind at this point was how a model can tell if the bird is moving upwards or downwards. Given the current state, we needed a way to tell the velocity (or trajectory) of the bird, which is why we chose to squash four consecutive frames and define that as our 'state.' This squashed frame has dimensions 80 by 80 by 4.

4. Deep Q Networks

We use the q-learning function to create an optimal action selection policy for the agent. The q-learning function is defined as follows:

$$Q(s, a) = r + \gamma * max(Q(s', a'))$$

Where:
s - current state of the game
a - action taken
$Q(s, a)$ - total reward point for taking action a at state s.
r - current reward value
γ - discount factor to determine the importance of future rewards
s' - next state after taking action a in state s
a' - all possible actions in s'

We model the above equation by representing it in a deep convolutional neural network, and find parameters such that we maximize $Q(s, a)$ for each state s the game encounters. Each s is a (1, 80, 80, 4) matrix (we need the 1 in front as that is a Keras requirement).

A key concept in reinforcement learning is the idea of exploration vs exploitation. Exploration is the concept of choosing a random action to see what new states the agent can be exposed to, where as exploitation is the act of letting the agent choose the action to take. If we rely 100% on exploitation, then the agent will never learn the rewards for certain uncharted scenarios, hence the ratio of exploration and exploitation is a key factor in reinforcement learning. We utilize a method called *epsilon annealing* where we decrease epsilon gradually as the training time goes on.
The epsilon annealing methodology recommends that we start off our epsilon at 1.0, which means we explore 100% of the time, not relying on agent prediction. Then as we observe more scenarios and train the agent, we decrease epsilon until a very small number (in our case, we used 0.0001, which means we explore 0.01% of the time). Intuitively, as the model becomes more trained, we trust its decisions more.

Another concept we used in training the network is batch training. Generally, training a neural network contains the below steps:

1. Feedforward
    a. Input a data sample and process it all the way until the output neuron.

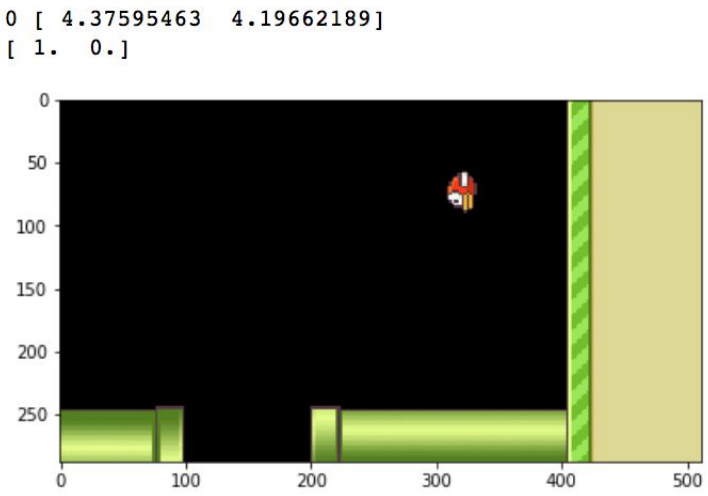
2. Backpropagation
    a. Calculate the gradient of the error function with respect to the weights/biases.
3. Stochastic Gradient Descent
    a. The process of updating weights/biases based on the gradient found from backpropagation.

It can be observed that it takes the above three steps per training sample, which can be very computationally expensive. Batch training differs in that stochastic gradient descent is not executed until all the samples in a batch have been passed forward. What this means is that it can drastically decrease the computations required to train the neural network, in general networks train faster with batches.

5. Results

Using the methodology described above, we explored 50k frames of the game and used batch training size of 32. In total, this took roughly 4 hours on my local laptop and 2 hours on AWS p2.xlarge instance. With this, we could not achieve the same level of superhuman results, but noticed very interesting and important results. The below output images contains the following information:

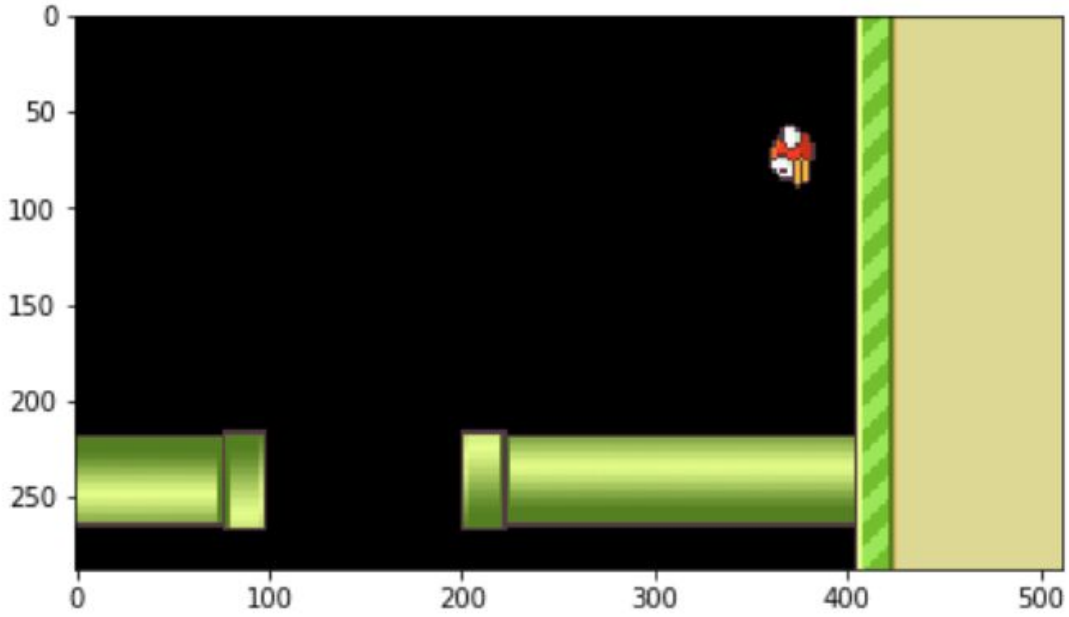
- The prediction output, where 0 means do nothing and 1 means to move up.
- The array which represents the expected reward value of taking a certain action.
- The action array which was passed in to the network. ([1,0] for do nothing or [0,1] for move up)



In the initial stages of the game where the pipes are not close, it does nothing and keeps going down, but learns that hitting the ground results in a terminal state.

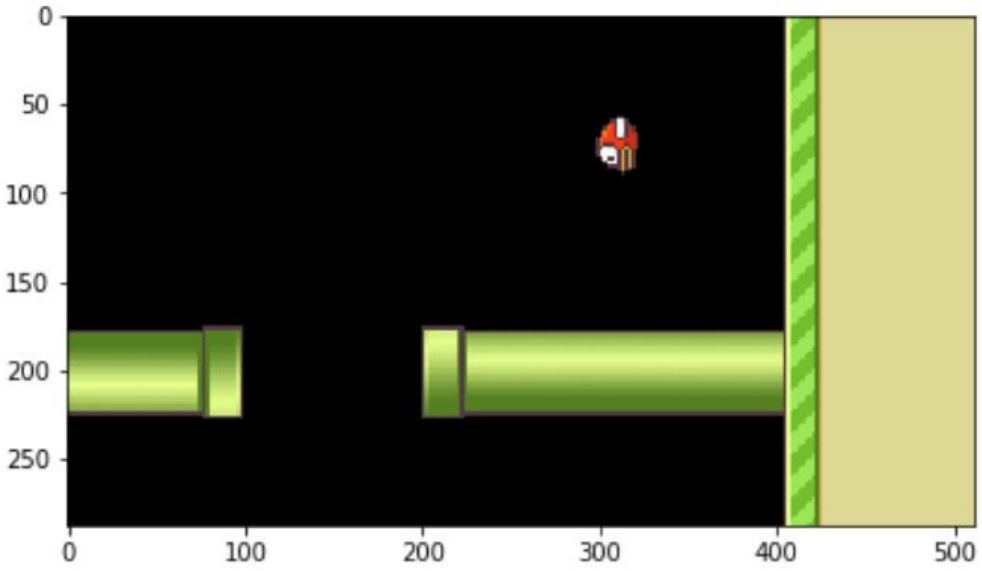
Once it nears the ground, the value in the second index exceeds the first index, and makes the bird move up.

```
1 [ 3.59042621  3.6813426 ]
[ 0.  1.]
```

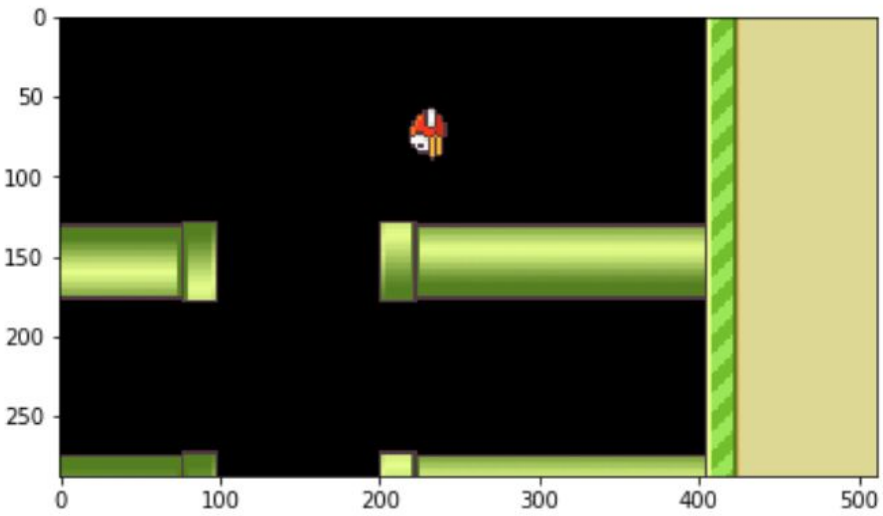


Secondly, it also recognizes when it is nearing the pipe. The next image shows that as it is reaching the pipe and notices it needs to move up to avoid it, it does so.

```
1 [ 4.32769585  4.39619017]
[ 0.  1.]
```

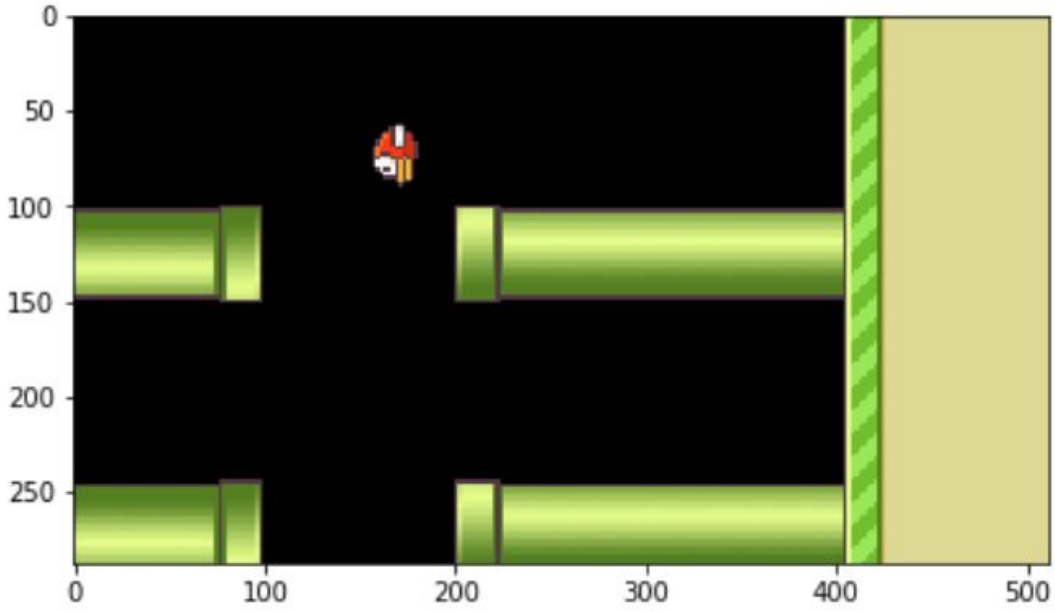


```
1 [ 3.86265206  3.9738481 ]
[ 0.  1.]
```



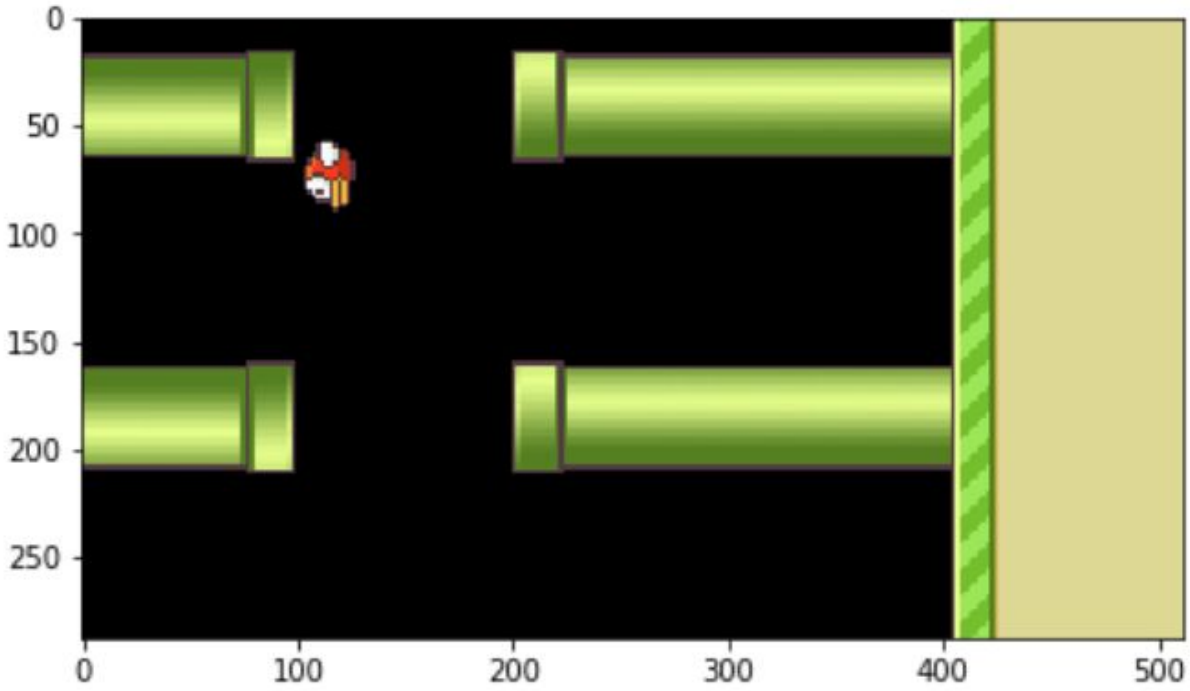
Then once it reaches an altitude higher than the pipe, it stops moving upwards and attempts to go through the section between the upper and lower pipe.

```
0 [ 3.37306714  3.31949496]
[ 1.  0.]
```



Unfortunately, once it passes through the middle of the first pipe, it reaches a terminal state by hitting the ending tip of the first pipe. For the below scenario, it can be easily adjusted with further learning.

```
1 [ 2.82255459  3.11814594]
[ 0.  1.]
```



As a comparison, the article stated above reached superhuman score with roughly one million frames of exploration (on a Titan X GPU), which is 20 times more data than what this project used. By a rough estimate, this would have taken 80 hours of training time, or 40 hours on AWS, which would be very costly. Due to lack of resources, we cut the training short here after achieving some very interesting results depicted above. Future work can include finding out ways to achieve convergence of the network quicker (using prioritized experience replay, double DQN structure), playing around with different CNN parameters and different reward values.