Machine Learning Engineer Nanodegree

Zhenhua Ma
December 29, 2017

CAPSTONE REPORT

I. DEFINITION

Project Overview

Machine learning represents a crucial area of artificial intelligence (AI). In recent years, the application of machine learning in learning to play games becomes very popular in AI research and is gaining more and more attentions in both the academic and the industry fields. Computer scientists, machine learning researchers, and software developers work together to introduce machine learning to playing games and have gained tremendous improvements and achievements over the years.

In 1949, Claude Shannon published a research paper titled “Programming a Computer for Playing Chess.” In this paper Shannon estimated that chess has more than 10^120 possible positions. Even with modern supercomputers, it is impossible to “solve” chess with pure brute force [1]. Less then 50 years later, in 1997, IBM’s supercomputer Deep Blue defeated reigning world chess champion Garry Kasparov in a six-game match, marking the first time a chess AI had bested the top human player. Nowadays, the Stockfish, one of the top chess AIs, is about 10 times more in skill level than the human #1 player [2]. It uses a combination of brute force and finely-tuned heuristics to compute numeric evaluations for every legal move in a position. To reduce the massive branching options that occur with each move (any given position may have 25 legal moves; each of those may have 25 legal moves, and so on), Stockfish uses a search algorithm called Alpha-beta pruning to weed out bad moves.

Go is a popular board game (especially in Asia) of even greater complexity: the number of legal Go positions is on the order of 10^170, much more than the complexity of chess. The Google

DeepMind team paved the way for computer dominance in Go in the past two years with the development of AlphaGo, an AI that uses deep neural networks to learn from its own games (a technique known as deep reinforcement learning) as well as games played by top human players (a classic supervised learning technique). In March 2016, it beat Lee Sedol in a five-game match, the first time a computer Go program has beaten a 9-dan professional without handicaps. At the 2017 Future of Go Summit, AlphaGo beat Ke Jie, the world No.1 ranked player at the time, in a three-game match. After this, AlphaGo was awarded professional 9-dan by the Chinese Weiqi Association [3].

Inspired by how Google applies deep reinforcement learning technique in the development of AlphaGo, I would like to apply what I have learn from the Machine Learning Nanodegree program to do a similar project in gaming: implement a neural network program that can learn by itself on how to play the Flappy Bird game.

Problem Statement

Flappy bird is a mobile game developed by Vietnamese programmer Dong Nguyen. The game is a side-scroller where the player controls a bird, attempting to fly between columns of green pipes without hitting them. If the player touches the pipes, they lose. At each instant there are two actions the player can take: to tap the screen so the bird flaps upward, or not to tap the screen so the bird falls because of gravity. Each pair of pipes that the player navigates between earns the player a single point, with medals awarded for the score at the end of the game. No medal is awarded to scores less than ten. A bronze medal is given to scores between ten and twenty. In order to receive the silver medal, the player must reach 20 points. The gold medal is given to those who score higher than thirty points. Players who achieve a score of forty or higher receive a platinum medal.

With reinforcement learning, I will implement a neural network to make the agent teach itself to play the flappy bird game. Without any prior information regarding to the game, the agent shall be able to learn how the game works only by playing it again and again. Eventually after training, the agent shall be able to make the right moves under different scenarios: whether the

bird should flap up or down to pass different pairs of pipes. When the game is finished, the agent shall be awarded a platinum medal.

Evaluation Metrics

When the Flappy bird game is terminated, a medal will be awarded to the player according to the score accumulated through the game. No medal is awarded to scores less than ten. A bronze medal is given to scores between ten and twenty. In order to receive the silver medal, the player must reach 20 points. The gold medal is given to those who score higher than thirty points. Players who achieve a score of forty or higher receive a platinum medal. This medal system or the score itself can be served as the evaluation metric for the performance of the CNN.

I also would like to investigate how much training the CNN is needed in order to achieve certain performance. So another evaluation metric will be the ratio of the number of training to the corresponding score.

II. ANALYSIS

Datasets and Inputs Exploration

The Flappy bird game emulator I am going to used is a python-pygame implementation of the original game. The source code of it can be found at [4].

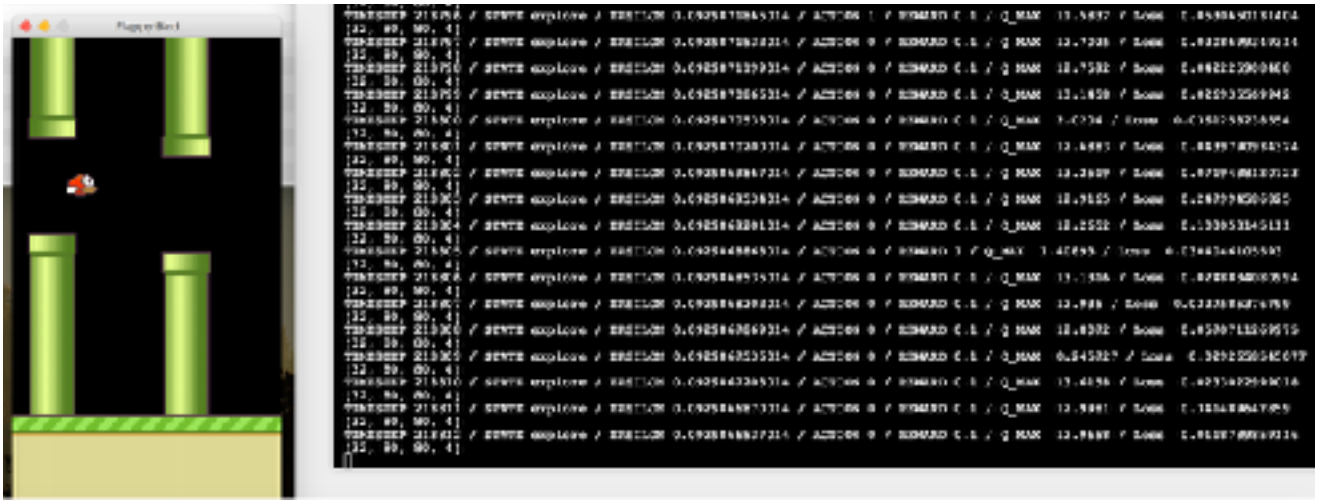
The five inputs to the Q function in the deep reinforcement neural network for playing the Flappy bird are listed below:

- *a*: the bird's action at the current frame, a = 1 means the bird flaps up, while a = 0 means the bird flaps down
- *s*: the bird's state at the current frame
- *r*: the reward the bird receive at the current frame, r = 0.1 if alive, r = 1 if the bird pass the pipe, r = -1 if the bird hit the pipe and die (game terminated)

- s': the bird's state at the next frame
- γ: the discount factor of the Q function

Exploratory Visualization

When the Flappy bird game is going on, those five inputs will also be presented to monitor the training process. The following plot showed an example of how those inputs are presented:



Algorithms and Techniques

I am going to use a convolution neural network to train the agent and make it learn to play the Flappy bird game by itself. The CNN will have multiple hidden layers, convolution layers, pooling layers, and eventually an output layer with a single output for each valid action.

This CNN will be built based on the Keras platform since I have already built many CNNs in there from the Nanodegree projects.

A Q-learning algorithm will be used to train the CNN. The Q function in this case represents the maximum discounted future reward when the bird performs action *a* in state *s*. By taking into account the inputs as mentioned in the "Datasets and Inputs Exploration" section, the Q

function is defined as below. It denotes the maximum future reward for this state and action (*s,a*) is the dimmedte reward *r* plus the maximum future reward for the next state *s*', action *a*'.

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

The specific Q-learning algorithm I used here is the Deep Q-Network [5]. It is a learning algorithm developed by the Google DeepMind team to enable playing games with AI. By using just the screen pixels, receiving a reward associate to the changing game score, the algorithm provide provide a way for a computer to learn to play video games by itself. I was amazed by how well this algorithm works as it turned out to be able to play various games as long as the proper inputs are provided.

Benchmark

The benchmark model in this project will be me playing the Flappy bird many times. In the cases of the Deep Blue and the AlphaGo, AIs played against human and the match results are used to evaluate/demonstrate the performance of the AI. As in this project, I would like to see how good the CNN can perform against me.

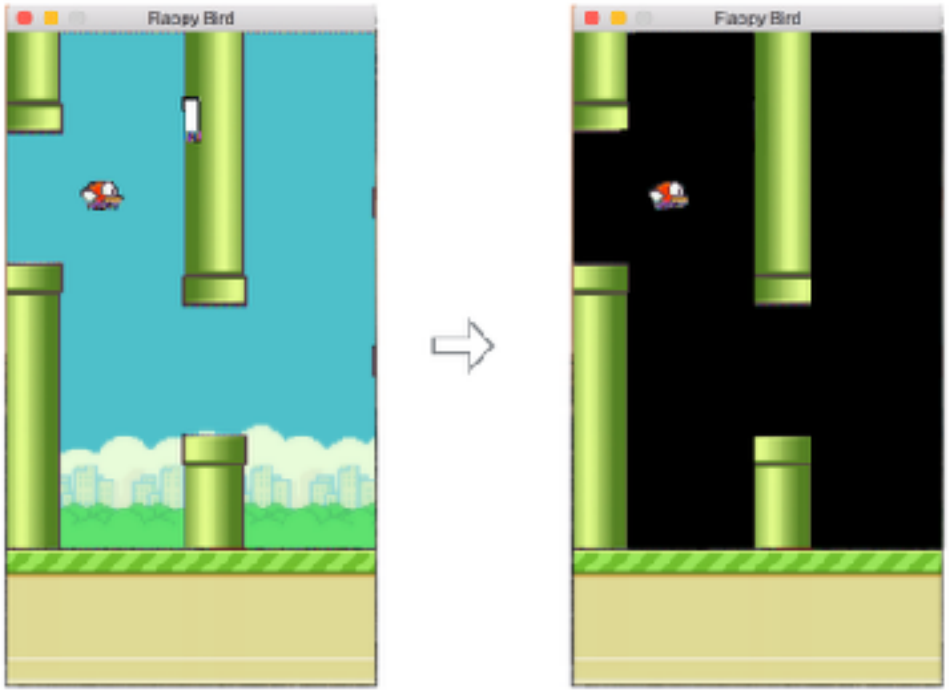
III. METHODOLOGY

Data Preprocessing

To make the training process faster, some image processing are needed before feeding the gaming screens to the neural network. The following steps are done in the image processing step:

- Converting color images from the Flappy bird game into grayscale images
- Cropping down the image size to 80x80 pixels
- Stacking multiple frames together (in order to obtain the velocity information of the bird)

The plot below showed how the Flappy bird game interface changes after the preprocessing:



Implementation

In order to implement this project, the following dependences needed to be installed:

- Python 2.7
- Keras 2.0
- pygame
- scikit-image

First the images from the Flappy bird game [4] will be processed as described in the “Data Preprocessing” section.

After the image processing step, the raw pixels from the images and the inputs mentioned in the “Algorithms and Techniques” section will be used as inputs to a Deep Q-Network as described in the “Solution Statement” section. This Deep Q-Network is a convolutional neural network that is trained with a Q-learning algorithm. Its output is a value function estimating future rewards. The values at the output layer represent the Q function given the input state for

each valid action. At each time frame, the network performs whichever action corresponds to the highest Q value using a ϵ greedy policy.

The pseudo-code for the Deep Q-learning algorithm as listed below are described in [6]:

```
Initialize replay memory D to size N
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialize state s_1
    for t = 1, T do
        With probability ϵ select random action a_t
        otherwise select a_t=max_a  Q(s_t,a; θ_i)
        Execute action a_t in emulator and observe r_t and s_(t+1)
        Store transition (s_t,a_t,r_t,s_(t+1)) in D
        Sample a minibatch of transitions (s_j,a_j,r_j,s_(j+1)) from D
        Set y_j:=
            r_j for terminal s_(j+1)
            r_j+γ*max_(a^' )  Q(s_(j+1),a'; θ_i) for non-terminal
s_(j+1)
        Perform a gradient step on (y_j-Q(s_j,a_j; θ_i))^2 with
respect to θ
    end for
end for
```

Based on the structure of the Deep Q-network, the convolution neural network I builded has the following structures:

```
model.add(Conv2D(filters=32, kernel_size=8, strides=4, padding='same', activation='relu',
                 input_shape=(80, 80, 4)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=4, strides=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, strides=1, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(2, activation='relu'))

adam = Adam(lr=LEARNING_RATE)
model.compile(loss='mse',optimizer=adam)
```

The first layer is a convolution layer that convolves the 4 frames of input images with 80x80 pixels with an 8x8x4x32 kernel with a stride size of 4. The second layer is a 2x2 max pooling layer, which down-sample the output matrix from the first layer, reducing its dimensionality and allowing for assumptions to be made about features contained in the matrix. The third layer is a convolution layer with a 4x4x32x64 kernel with a stride size of 2. The fourth layer is again a 2x2 max pooling layer. The fifth layer is a convolution layer with a 3x3x 64x64 kernel with a stride size of 1. The sixth layer is a max pooling layer. The seventh and the last layer is a fully connected layer with 256 nodes and a 'relu' activation function. Eventually the output layer has the same number of outputs as the number of possible actions the bird can perform: 0 which is doing nothing and 1 which is flying up.

This output of this CNN is the value of the Q function of the Deep Q-network: instead of getting the full Q-table of Q(s, a), the CNN is used to compress the full Q-table with different weightings inside the neural network. As a result, by fine tuning the weightings used in the CNN, the optimal value of the Q-function can be found using the training process of the CNN.

Refinement

At the very beginning, all weightings in the CNN are initialized randomly with a normal distribution that has a standard deviation of 0.01. The initialization of the network is very important since neural networks are usually trained using back propagation, which is a non-convex optimization problem for most of the loss functions. As there are multiple local minima, non-convex generally converge to different optimal points for different initial conditions. So it not only affects the speed of the convergence but the network's optimality also. Initial parameters of neural networks are as important as the network architecture. The non-linear activation functions (such as sigmoid) are generally tricky as gradient of these functions impacts the weight updates in gradient descent. A small weights leads to very small gradient, so small inefficient steps. On the other hand, large weights saturates the activations, which again leads to small updates. Hence it is desirable to keep the weights in such a way that activation functions are in linear zone, so that gradients are optimal. It is also advisable to have both positive and negative values as initial weights. Also generalization (avoiding overfitting) is achieved through magnitude control of the weights (norm regularization), to keep them in this

optimal zone. That's why I choose to initialized the CNN with a normal distribution with a standard deviation of 0.01.

During the training, the ϵ value is changed from 0.1 to 0.0001 over the total of 3000000 frames. In this way the agent can choose an action every 0.03 second during the game, which is more than enough to control the actions of the bird and proceed in the game. I also found out that the value of ϵ cannot be set too high, otherwise the bird will keep flying up in the game and stay at the top of the screen. Eventually the bird will hit the top of one pipe and end the game. Such a condition will make the Q function converge very slowing at the early stage of the training. Only when the ϵ value becomes smaller then the bird will try different actions, that's why I choose to start the ϵ value around 0.1 and gradually reducing it.

When preparing for this project, I read one of Google's DeepMind Atari paper [6]. In there a concept of "Experience Replay" is presented and explained very well. In Q-learning, the experiences recorded in a sequential manner are highly co-related. If they are used in the same sequence to update the DQN parameters, the training process will be hindered and unstable. Similar to sampling a mini-batch from a labeled data set to train a classification model, some randomness shall be brought in when selecting the experiences for updating the DQN. In order to overcome this problem, instead of running Q-learning on state/action pairs as they occur during simulation or actual experience, the system stores the data discovered for [state, action, reward, next_state] in a table. Note this does not store associated values - this is the raw data to feed into action-value calculations later. The learning phase is then logically separate from gaining experience, and based on taking random samples from this table. You still want to interleave the two processes - acting and learning - because improving the policy will lead to different behavior that should explore actions closer to optimal ones, and you want to learn from those. However, you can split this how you like - e.g. take one step, learn from three random prior steps etc. The Q-Learning targets when using experience replay use the same targets as the online version, so there is no new formula for that. The loss formula given is also the one you would use for DQN without experience replay. The difference is only which (s, a, r, s', a') you feed into it. In this project, during the gameplay all the (s, a, r, s') inputs are stored in replay memory. When the training begins, random mini-batches from the replay memory are used instead of the most recent transition. Experience replay breaks up the correlation in data

and helps improve the networks convergence as you are sampling batch of experiences randomly from a large memory pool. In this way one avoids the non stationary aspect of the training data and greatly improve the stability of the approximation of the Q-value.
An adaptive learning algorithm “ADAM” is used in the CNN I built to do the optimization. [7] provides very good descriptions of different gradient descent optimization algorithms. After reading through it and comparing different optimizations algorithms, I decided to use ADAM since it has been shown empirically that ADAM works well in practice and compares favorably to other adaptive learning-methods algorithms.

There is another issue in the reinforcement learning algorithm which called Exploration vs. Exploitation. How much of an agent’s time should be spent exploiting its existing known-good policy, and how much time should be focused on exploring new, possibility better, actions? We often encounter similar situations in real life too. For example, we face on which restaurant to go to on Saturday night. We all have a set of restaurants that we prefer, based on our policy book Q(s,a). If we stick to our normal preference, there is a strong probability that we’ll pick a good restaurant. However, sometimes, we occasionally like to try new restaurants to see if they are better. The RL agents face the same problem. In order to maximize future reward, they need to balance the amount of time that they follow their current policy (this is called being “greedy”), and the time they spend exploring new possibilities that might be better. A popular approach is called ϵ greedy approach. Under this approach, the policy tells the agent to try a random action some percentage of the time, as defined by the variable ϵ, which is a number between 0 and 1. The strategy will help the RL agent to occasionally try something new and see if we can achieve ultimate strategy.

IV. RESULTS

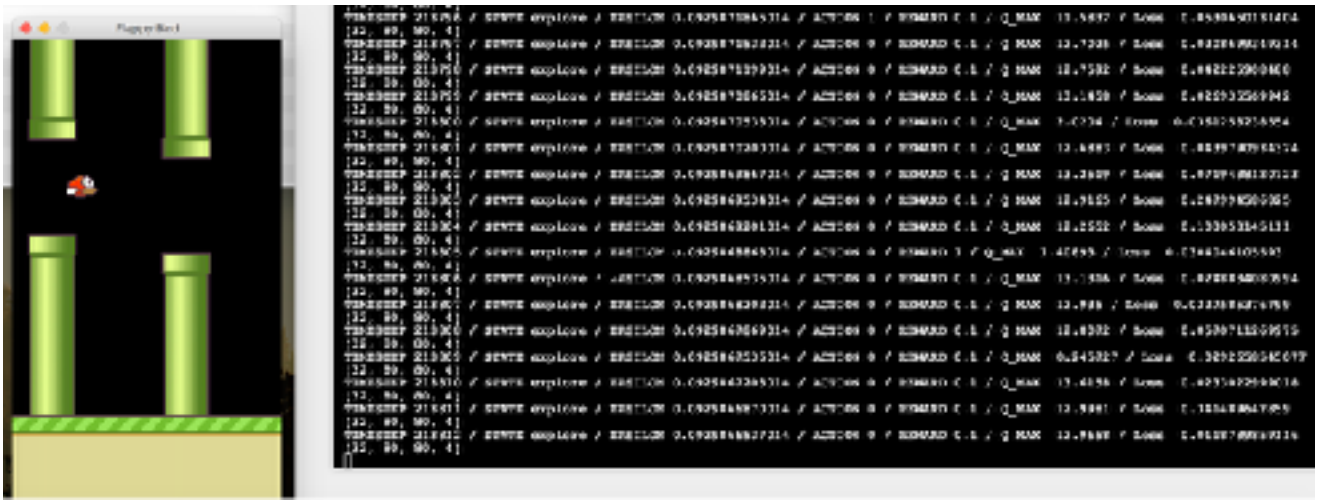
Model Evaluation and Validation

The final model of the CNN has the following parameters:

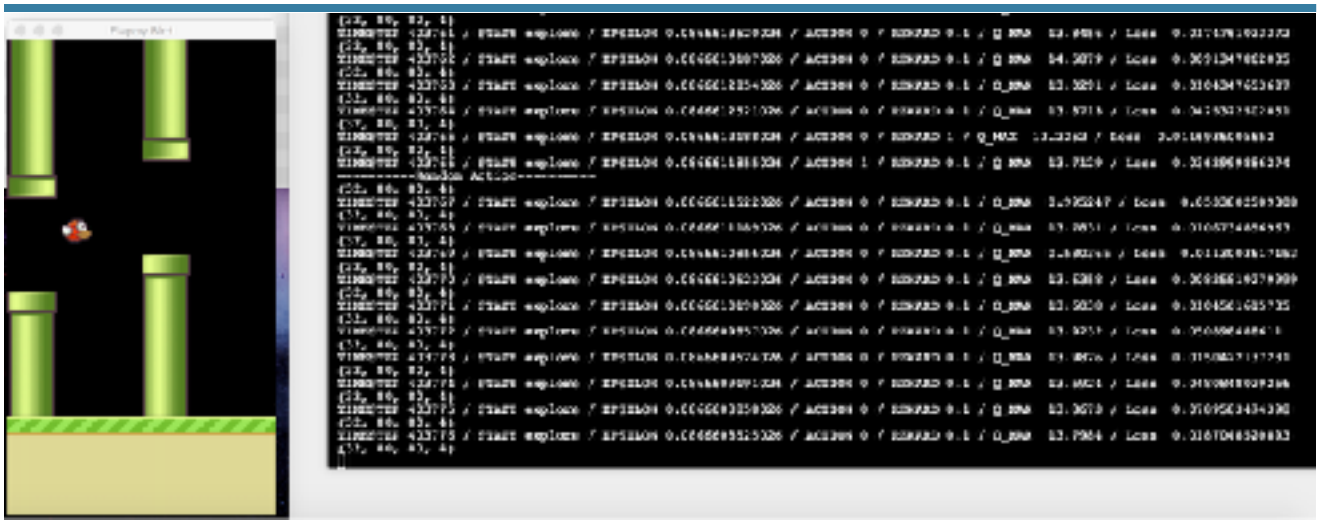
```
GAME = 'flappy_bird' # the name of the game being played for log files
CONFIG = 'nothreshold'
ACTIONS = 2 # number of valid actions
GAMMA = 0.99 # decay rate of past observations
OBSERVATION = 3200. # timesteps to observe before training
EXPLORE = 3000000. # frames over which to anneal epsilon
FINAL_EPSILON = 0.0001 # final value of epsilon
INITIAL_EPSILON = 0.1 # starting value of epsilon
REPLAY_MEMORY = 50000 # number of previous transitions to remember
BATCH = 32 # size of minibatch
FRAME_PER_ACTION = 1
LEARNING_RATE = 1e-4

img_rows , img_cols = 80, 80
#Convert image into Black and white
img_channels = 4 #stack 4 frames of images as input
```

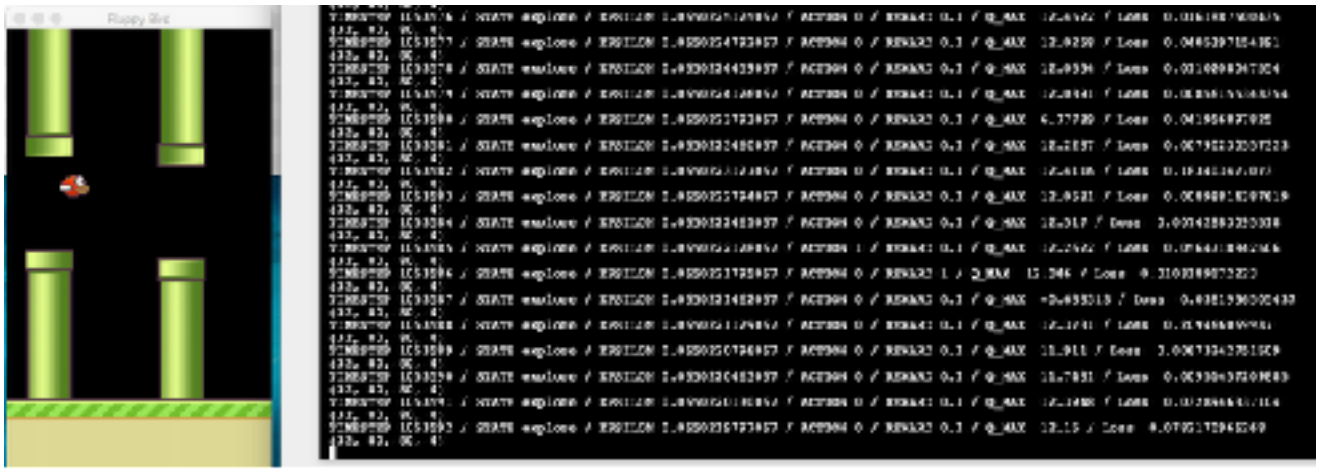
During the training, a few screen shots were taken to record the training process and monitor the performance of the agent.



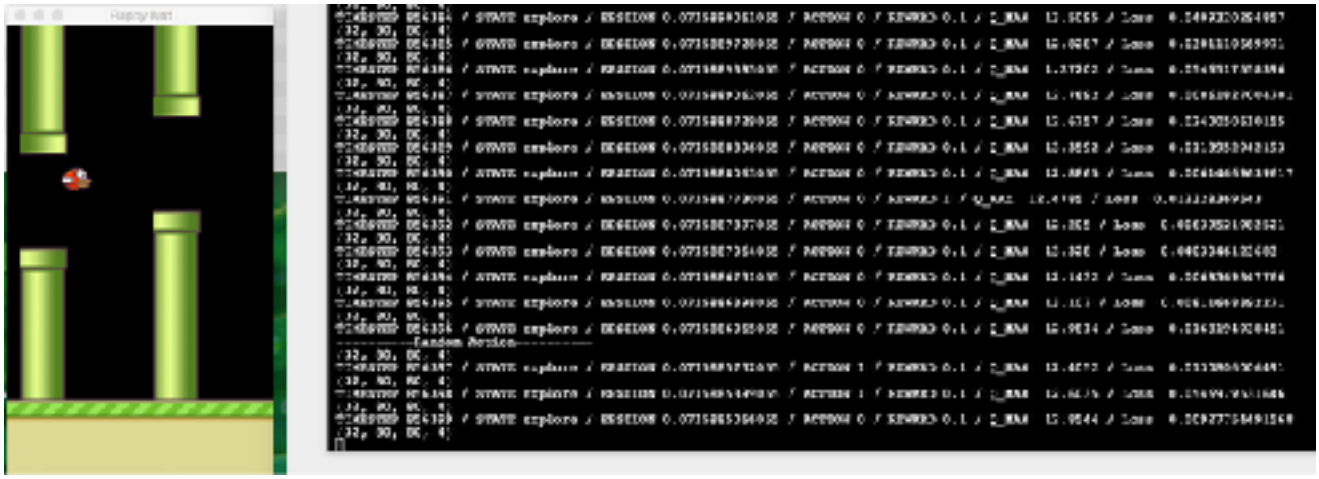
About 200,000 frames in, the agent started to show behaviors indicating that the CNN is learning to find the relative position of the bird with respect to the top, bottoms of the screen, and to the pair of pipes that it has to pass.



About 400,000 frames in, the agent was doing a better job in passing more pair of pipes. This behavior was expected since the CNN was trained with more frames at this moment.



About 800,000 frames in, the agent is doing much better than it was at the very beginning, more trails of passing multiple pipes in one go can be observed. However, sometimes the agent will still made simple mistakes such as going straight to the top of the screen and eventually smashed into the top of the upper pipe. This might be caused by the random action taken by the agent to explore more possibilities.



About 1000,000 frames in, the behaviors of the agent was quite mature, indicating the CNN has achieved a good level of learning of the game. At this stage, the ϵ value has became smaller (about 0.06) and fewer random actions is taken.

Eventually, after processing 3000,000 frames, the ϵ reached the value of 0.0001. One thing to note here is that the training is in fact very slow: it took my iMac computer about 6 days working non-stop to finish the training. Part of the reason is I didn't utilize the GPU resource of the computer, which shall speed up the training process effectively.

Justification

After the training is finished, I make the agent to play 20 flappy bird games and recorded its scores. Then I compared the scores with the bench mark: scores of mine after playing 20 flappy bird games by myself. The comparisons can be found in the following table.

Scores	Human (Me)	DQN Agent
Average Score	6.5	41
Maximum Score	23	59

Table 1: Average and maximum scores from 20 flappy bird games played by me and the DQN agent after training.

From the table above one can see that after the training process, the DQN agent have learned the game very well and beat me (by a lot!) in both the average and maximum scores categories.

On one hand, I am not surprised at all at this result because the agent practically have played the flappy bird game millions of times. On another hand, I am truly amazed by this result since the agent learned how to play the game all by itself through reinforcement learning.

I searched the internet for the average scores of playing flappy bird by human but there is not much information. I did see a few flappy bird players post their scores stating they can score over 100 on the game. From this aspect, the DQN agent is still not doing as good as a human player can be. However, I am sure with some fine-tunings of the model and its parameters the DQN agent can improve its performance and reach that level without a problem. Overall, the model I built is quite promising and definitely has the capacity to be better.

V. CONCLUSION

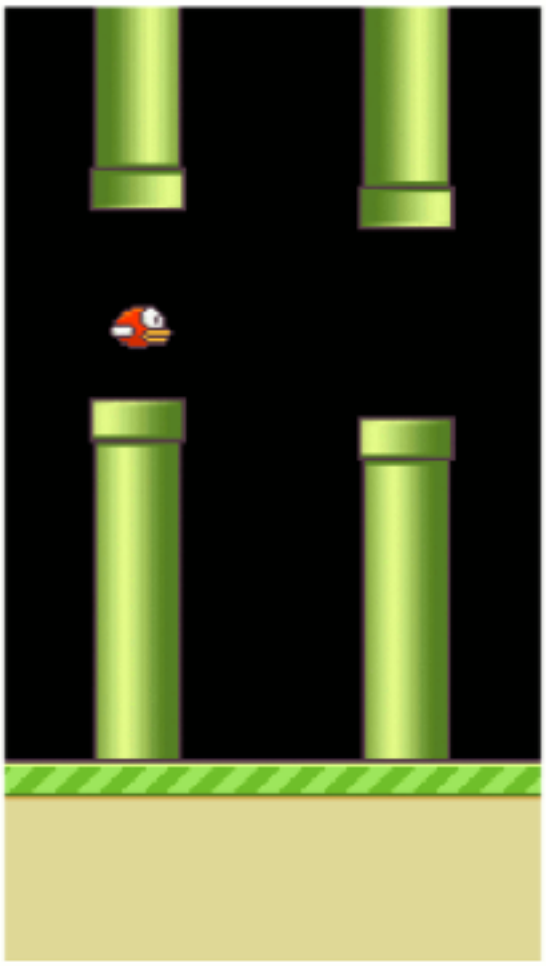
Free-Form Visualization

The DQN agent can be called to play the flappy bird game with the following steps:

1. After installing the dependences mentioned in Section “Implementation” in Page 6, download and go into the attached folder “AI_flappy_bird”
2. In the terminal, type in: python CNN_flappybird.py -m “Run”

A pygame window shall then appeared and the bird will start flying through the pipes, as shown in the figure below.

The agent can be trained again by calling the command: python CNN_flappybird.py -m “Train”. The file “model.h5” shall be deleted first before the training process in order to have all new weightings of the model.



Reflection

With reinforcement learning, deep Q-network and CNN, I was able to implement a DQN agent who can teach itself to play the famous flappy bird game. Without any prior information regarding to the game, the agent was able to learn how the game works only by playing it again and again (over millions of games played in my case). Eventually after training, the agent was able to make the right moves under different scenarios: whether the bird should flap up or down to pass different pairs of pipes. Over the 20 games the DQN agent played, its average score is 41 with the highest score being 59. On the average, the agent was able to achieve a level that will be awarded a platinum medal, just as I expected while writing the proposal of they project. The agent was also beat me in this game by a big margin. Overall, I think the model built in this project is very promising and its performance is beyond my expectation.

The most interesting and also the hardest part of this project is to learn about and use the deep Q-network. It is a learning algorithm developed by the Google DeepMind team to enable playing games with AI. By using just the screen pixels, receiving a reward associate to the changing game score, the algorithm provide provide a way for a computer to learn to play video games by itself. I was amazed by how well this algorithm works as it turned out to be able to play various games as long as the proper inputs are provided. The idea and algorithms behind the DQN is not trivial, but fortunately for me, there are many resources available on the internet regarding to the deep Q-network. Through reading many articles such as [5] and the paper [6], I was able to have an idea of it and implemented it in this project successfully.

Improvement

There are a few things that I would like to investigate more to further improve the CNN and the performance of the DQN agent:

1. The training process is very slow. Part of the reason is that I didn't utilize the GPU resource. I would like to investigate on how to use the GPU to speed up the training process. One possible way is to go with the Theano library in Python [8].
2. At this point, I am not sure whether the CNN model I built is the optimal or even the sub-optimal CNN or not. I would like to find out whether there is a way to quantify the performance of the CNN.
3. During the training, due to the "Exploration vs. Exploitation" consideration, a random action will be taken by the agent sometime. And due to the likelihood of this random action, even in the later stage of the training I still observed some simple mistakes made by the agent. I wonder if there is a better way to do the "Exploration vs. Exploitation" trade-off and improve the training process of the CNN.

VI. REFERENCE

[1] https://www.bigfishgames.com/blog/the-past-and-present-of-machine-learning-in-gaming/

[2] https://www.techemergence.com/machine-learning-in-gaming-building-ais-to-conquer-virtual-worlds/

[3] https://en.wikipedia.org/wiki/AlphaGo

[4] https://github.com/TimoWilken/flappy-bird-pygame.git

[5] https://www.intelnervana.com/demystifying-deep-reinforcement-learning/

[6] Mnih Volodymyr, 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, and Demis Hassabis. **Human-level Control through Deep Reinforcement Learning**. Nature, 529-33, 2015.

[7] http://ruder.io/optimizing-gradient-descent/index.html#adam

[8] http://deeplearning.net/software/theano/