WHITNEY KING

MACHINE LEARNING ENGINEER NANODEGREE

FEBRUARY 27TH, 2019

CAPSTONE PROPOSAL

TABLE OF CONTENTS

I DOMAIN BACKGROUND

For enthusiasts of machine learning, one of the most captivating aspects of artificial intelligence is infinite number of ways in which it can be applied. Uses making major waves in the field span the gamut from philanthropic, to economic, to humanitarian, to malicious; all of which are catapulting the world into a future of technology only previously imagined through science fiction. For domain related concepts discussed in this project, it's important to understand the nuances differentiating each one:

- **Deep learning**; class of machine learning algorithm known as *neural networks*
    - **Neural networks**; operate in a hierarchical, non-linear fashion
        - Cascade of layers with various abstractions which perform transformations or feature extraction[1]
- **Convolutional neural networks (CNN)**; designed to require little preprocessing by integrating multilayer perceptrons
    - Takes an input, passes it through multiple hidden convolution layers, then returns an output
    - Popular choice for image and text processing[2]
- **Reinforcement learning**; one of the three machine learning paradigms, alongside supervised and unsupervised learning
    - *Model-less*; focuses on performing *actions* which maximize set *rewards* in a specific *environment*
    - Popular choice for simulation and optimization tasks
    - Foregoes classifications or predictions; performs *actions* which maximize set *rewards* in a specific *environment*[3]
    - Strikes balance between *exploration* and *exploitation*
        - **Exploration**; gathering more information
        - **Exploitation**; performing the best action given what is known[4]
- **Deep-Q** ; type of *deep-reinforcement learning* that employs *convolutional neural networks*
    - Operates in a discrete *action space*; such as a play field
    - Utilizes an *action-value* function (known as the $\boldsymbol{Q}$-function) with two inputs; *state* and *action*
    - Outputs the expected *reward* for an *action* given the *state*[5]

I.I MOTIVATION

One of the first game playing engines to make the news was Google's *AlphaGo*, built to beat the best human *Go* players in the world using supervised learning and Deep-Q reinforcement learning. The latest evolution of *AlphaGo* introduced *AlphaGo Zero*, which has not only learned to play *Go*, but also *Chess* and *Shogi* through deep-reinforcement learning; with *zero* human opponents or prior domain knowledge[6]. These examples are undeniably amazing feats of man and machine. For me it's applications such as this that are the most engaging; ones fusing technology, functionality, and entertainment.

While studying reinforcement learning and thinking on positive, light-hearted applications for this type of agent, I was immediately reminded of the ongoing viral social experiment *Twitch Plays Pokémon*[7]. Though this is not an AI driven platform, games are played

[1] (Deep learning, 2019)
[2] (Convolutional neural network, 2019)
[3] (Bajaj, 2015)
[4] (Reinforcement learning, 2019)
[5] (Simonini, 2018)
[6] (Silver, Hubert, & Schrittwieser, 2018)
[7] (Twitch Plays Pokémon, 2019)

using a collaborative engine which performs actions based on commands sent through a Twitch[8] channel chatroom. This sets up a hive of actions all supposedly vying towards the same goal, but inevitably stymied by misplaced actions or bad actors. Setting up a learner to play the entirety of a Pokémon game seemed overly ambitious for a first project of this nature, however the concept reminded me of the unpredictability of reinforcement learning agents. Ultimately, this was the inspiration behind wanting to train an agent that will play a more basic, well-recognized, and easily available game.

I.II ASSUMPTIONS

Flappy Bird is a game where a single-player *"directs a flying bird [ ], who moves continuously to the right, between sets of pipes [ ]. The bird flaps upward each time that the player taps the screen; if the screen is not tapped, it falls because of gravity"*[9].

I.II.I ENVIRONMENT

- Game can be played on three difficulty levels: **easy**, **medium**, or **hard**
  - Controls the number of on screen at any given time
- Level has no set end; continues scrolling forward until the player loses
- Each *always* have the same gap distance; vertical position varies
- automatically moves right →, while the player controls vertical movement
  - Tap at the appropriate intervals to move ↑/↓
  - Avoid

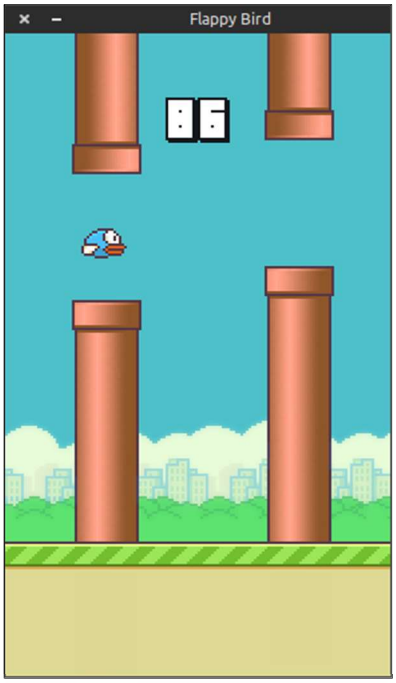

FIG 1: *Flappy Bird* Screenshot (Verma, 2017)

I.II.II RULES

- Each successfully cleared by earns **1 point**
- When touches either , the **game is over**
- Medals are award based on score at the **end** of the game
  - **Bronze**: 10 – 19
  - **Silver**: 20 – 29
  - **Gold**: 30 – 39
  - **Platinum**: 40+

II PROBLEM STATEMENT

This project will build a Deep-Q reinforcement learning agent to play the game *Flappy Bird*. It's repetitive by design, with a consistent and easily defined structure, observable task, and entirely replicable format. These are all characteristics of a good environment to baseline a learner, as well as an ideal candidate for this type of project.

II.I FORM

To structure learning to play *Flappy Bird* as a machine learning problem, Mitchell's Formalism can be followed[10]:

[8] https://www.twitch.tv/twitchplayspokemon/
[9] (Flappy Bird, 2019)
[10] (Brownlee, 2013)

II.I.I TASK

PLAY THE GAME FLAPPY BIRD

II.I.II EXPERIENCE

REINFORCEMENT LEARNING THROUGH REPEATED SELF-PLAY

II.I.III PERFORMANCE

ACHIEVE MAXIMUM SCORE

II.II FUNCTION

To help visualize and understand what makes the Deep-Q algorithm the appropriate choice for an agent playing *Flappy Bird*, a game in which the sole mission is to get the highest possible score, we should examine the algorithms policies in detail.

II.II.I TARGET POLICY (π)

The foundation of Deep-Q learning begins with an *action-value* function, written as $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$. This represents the expected reward from starting in *state* $\boldsymbol{s}$, taking *action* $\boldsymbol{a}$, then following a *target policy* $\boldsymbol{\pi}$. In general, the higher the quality, the better the action. Calculating $\boldsymbol{Q}(\boldsymbol{s}, \boldsymbol{a})$ given $\boldsymbol{\pi}$ is represented[11]:

$$\boldsymbol{Q^{\pi}} : \boldsymbol{S} \times \boldsymbol{A} \rightarrow \mathbb{R}$$

- $\boldsymbol{S}$: set of all possible *states* (positions of  and  on screen)
- $\boldsymbol{A}$: set of all possible *actions* (flap/don't flap)
- $\mathbb{R}$: value of *reward* for $\boldsymbol{s} \in \boldsymbol{S}$ and $\boldsymbol{a} \in \boldsymbol{A}$, following $\boldsymbol{\pi}$ on a forward trajectory

II.II.II BEHAVIOR POLICY (μ)

Deep-Q Convolutional Neural Network

The benefit of Deep-Q learning to this particular problem is that it searches for an optimal *behavior policy* $\boldsymbol{\mu}$ in a given environment by maximizing the expected total reward value of each successive state/action pairing[12]. This is done through iterative value updates using the weighted average of the old value and the new information, passed through convolution layers where the agent will attempt to optimize and follow $\boldsymbol{\mu}$ simultaneously.

Optimal $\boldsymbol{\mu}$ for the *maximum sum of all attainable reward values* solves the exploration versus exploitation trade-off, and is *greedy* because it always takes actions based on highest possible $\boldsymbol{Q}$[13]:

$$\boldsymbol{Q}^{*}(\boldsymbol{s}, \boldsymbol{a}) = \max_{\boldsymbol{\pi}} \boldsymbol{Q^{\pi}}(\boldsymbol{s}, \boldsymbol{a}),\ \forall_{\boldsymbol{s}}, \boldsymbol{s} \in \boldsymbol{S}, \boldsymbol{a} \in \boldsymbol{A}$$

[11] (Q-learning, 2019)
[12] (Deep reinforcement learning, 2019)
[13] (Oppermann, 2018)<

III DATASETS & INPUTS

This project is designed to be a self-playing reinforcement agent with no opponents or prior domain knowledge; thus, **it will not use a dataset**. However, the agent will need to be provided with an emulated environment running *Flappy Bird*, outlined in VII.II Emulation. Data and inputs used for logic will come directly from experience gained as the agent works through the learning process. It will begin with a blank slate, and information will be tracked in an *experience replay* tuple $\langle \boldsymbol{s}, \boldsymbol{a}', \boldsymbol{r}, \boldsymbol{s}' \rangle$ in the form of a deque, which serves as a simple cache for logging the *state* $(\boldsymbol{s})$, *action taken in* $\boldsymbol{s}$ $(\boldsymbol{a}')$, *next state* $(\boldsymbol{s}')$, and *reward for* $\boldsymbol{a}'$ $(\boldsymbol{r})$[14].

III.I PSEUDOCODE

In his article on self-learning AI agents, Oppermann outlines the pseudocode for a Deep-Q algorithm using experience replay to simultaneously optimize and follow a behavior policy; illustrating how these concepts combine to create and process layers of inputs and outputs[15]:

Initialize replay memory $\boldsymbol{D}$ to size $\boldsymbol{N}$
Initialize action-value function $\boldsymbol{Q}$ with random weights
for episode $= \boldsymbol{1}$, $\boldsymbol{M}$ **do**
  Initialize state $\boldsymbol{s_t}$
  for $\boldsymbol{t = 1}$, $\boldsymbol{T}$ **do**
    With probability $\in$ select random action $\boldsymbol{a_t}$
    Otherwise select $\boldsymbol{a_t} = \max_{\boldsymbol{a}} \boldsymbol{Q}^*(\boldsymbol{s_t}, \boldsymbol{a}; \boldsymbol{\theta})$
    Execute action $\boldsymbol{a_t}$ in emulator and observe reward $\boldsymbol{r_t}$ and state $\boldsymbol{s_{t+1}}$
    Store transition $\langle \boldsymbol{s_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{s_{t+1}} \rangle$ in $\boldsymbol{D}$
    Set $\boldsymbol{s_{t+1}} = \boldsymbol{s_t}$
    Sample random minibatch of transitions $\langle \boldsymbol{s_t}, \boldsymbol{a_t}, \boldsymbol{r_t}, \boldsymbol{s_{t+1}} \rangle$ from $\boldsymbol{D}$
    Set $\boldsymbol{y_j} = \begin{cases} \boldsymbol{r_j} & \text{for terminal } \boldsymbol{s_{t+1}} \\ \boldsymbol{r_j} + \boldsymbol{\gamma} \max_{\boldsymbol{a}'} \boldsymbol{Q}(\boldsymbol{s_t}, \boldsymbol{a_j}; \boldsymbol{\theta}) & \text{for nonterminal } \boldsymbol{s_{t+1}} \end{cases}$
    Perform gradient step on $\left(\boldsymbol{y_j} - \boldsymbol{Q}(\boldsymbol{s_t}, \boldsymbol{a_j}; \boldsymbol{\theta})\right)^2$
  end for
end for

IV SOLUTION STATEMENT

Knowing the $\boldsymbol{Q}$-value for any $(\boldsymbol{s}, \boldsymbol{a})$ pair enables the agent to execute the best action for any given state by utilizing the principle of *temporal difference*; recursively comparing the old reward to the new one, then choosing the highest value[16]. As depicted in Figure 2, the problem will be solved using deep reinforcement learning through iteratively self-playing *Flappy Bird* while building and following a behavior policy targeting highest $\boldsymbol{Q}$-values.

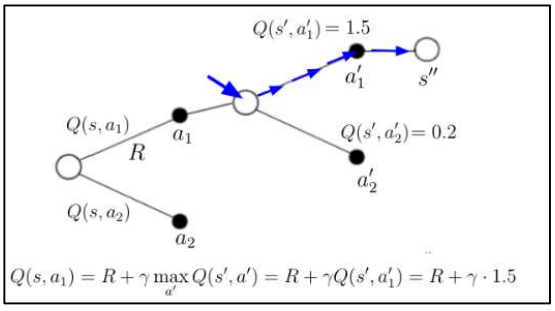

FIG 2: ***Agent following greedy policy for Q*** **(Oppermann, 2018)**

[14] (Simonini, 2018)
[15] (Oppermann, 2018)
[16] (Oppermann, 2018)

V BENCHMARK MODEL

For a continuous control agent such as the one utilized in this project, there is not a set standard for baselining. There are a few tool suites such as `garage`[17] aimed at benchmarking reinforcement learning algorithms performing well defined and widely applied tasks. For our purposes, a specialized benchmarking tool is overkill since the game has integrated award tiers based on score at the end, as described in I.II.II Rules. We can look to these awards to help formulate objective comparisons for the solution. Based on the medals, it can be inferred that scoring less than 10 points is an unsatisfactory performance, while scoring above 40 points is an exemplary one.

VI EVALUATION METRICS

Due to the fact that *Flappy Bird* has a somewhat unique and limited environment, many typical game mechanics such as wins, losses, draws, number deaths, or lives are ruled out right away as possible evaluation metrics; these ideas simply don't exist in this game. During gameplay, flies forward → through repeated accumulating points which cannot be lost. The longer the game is played, the higher the score should be. We have an exact measurement for score via the point system already, so using raw time to estimate score isn't a sensible evaluation metric either. However, abstracting this idea further yields two useful evaluation metrics.

VI.I DISTANCE

The player's tap speed controls the ↑/↓ movement of . The agent will need to learn to vary the strategy it uses for action (flap/don't flap) based on the position of , as well as the distance between each consecutive . Several pipes may be on the screen at once; their gap size is constant, but gap position varies vertically. Flapping must be carefully timed to navigate around and through . The agent ideally should learn to maximize the distance it keeps between the sprite and the pipes.

To translate this concept into a measurable metric, we can calculate the **average distance in pixels maintained between and the pipes** for every episode, and set the agent to self-play many episodes (we'll try 50,000).

VI.II SCORE

We know the game is immediately over when touches , and medals are awarded in tiers based on how many points the player has at the end. From this, it can be deduced that the agent is learning when the average score it's getting each game is climbing; it's learning well if it can consistently attain a score or 40 or greater.

To translate this into a measurable metric, we can calculate the **average score of episodes played**, then ensure our learner is getting better the more it plays.

VII PROJECT DESIGN

Finally, we'll discuss the generalized requirements, structure, and workflow of the *Flappy Bird* playing reinforcement learning agent. Building out this project requires a good amount of prior knowledge in the areas of programming and machine learning.

[17] https://github.com/rlworkgroup/garage

VII.I REQUIREMENTS

Visual Studio Code will be used as the IDE for this project, though most any text editor will work for reproducing it. The project has the following dependencies:

- Git
- Python 3 environment with the following packages installed:
  - `PyGame` open source game development platform for Python
  - `Scipy` open source scientific computing package suite for Python
  - `Scikit-Learn` popular machine learning library for Python
  - `Keras` expansive open source library for neural networks
  - `TensorFlow` suite of open source deep learning libraries
  - `OpenCV` open source image processing library

Installing a Python distribution and creating the environment will be done through Anaconda. To automatically create a `flappy` environment preconfigured with the correct dependencies for this project:

1. Download the requirements file `environment.yml` to a `{path}` on your system
2. Edit `environment.yml` so `prefix` matches your Anaconda `\envs` path
3. From *Anaconda Prompt* run:

```
conda env create -f {path}\environment.yml
```

VII.II EMULATION

There are several widely available opensource codebases in existence that emulate *Flappy Bird* and provide all necessary assets. This project will utilize a popular clone *FlapPy Bird*[18], built in Python using `pygame` and available on GitHub under the MIT License[19], which provides all functionality and should require relatively little code modification to make it suitable for an RL agent.

1. From a command terminal `cd` to the project directory and run:

```
git clone https://github.com/sourabhv/FlapPyBird.git
```

2. All graphics are located in the `\assets` folder
3. The game can be launched by running `flappy.py` from your Python terminal
4. The ↑ or **`Space`** keys are used to flap in lieu of tapping on the screen

VII.III MODIFICATIONS

Since the game is repetitive by design, it should be simple to set the agent up to play many times over. Colors and background are of no consequence to an agent designed to achieve the highest possible score. In fact, having to process that data on top of the sprite and object data would hinder the performance of the agent, so the background will be cut out of the game's code. Each frame will be reshaped into a square and converted to grayscale prior to being passed through the convolutional network; it would otherwise greatly impact efficiency by complicating the transformations, and processing each frame as 3Ds volume instead of a 2D plane due to RGB color channels[20].

[18] https://github.com/sourabhv/FlapPyBird
[19] (Verma, 2017)
[20] (Convolutional neural networks (CNN), 2019)

Figure 3 illustrates what a standard sequence of transformations looks like for resampled images passing through convolution layers, the last of which will return an *output with the same number of dimensions as there are actions;* ***2***.

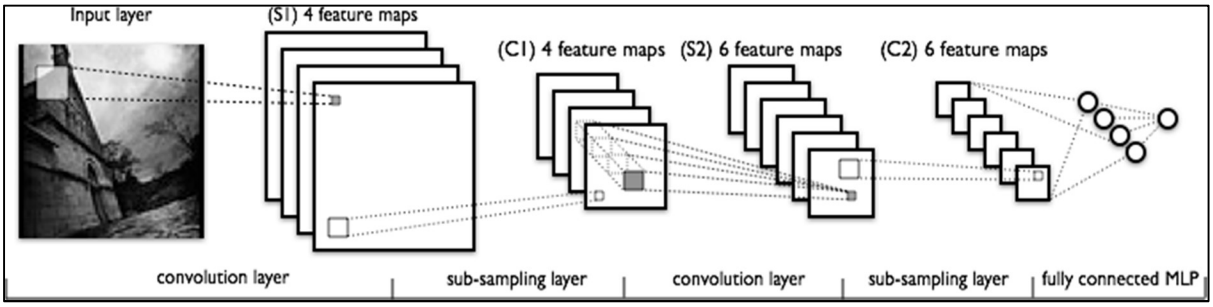

FIG 3: ***CNN Layers of Transformations***
(Convolutional neural networks (CNN), 2019)

VII.IV STRUCTURE

While running, the agent will choose random actions while it learns which ones are the best to take for each given state. To visualize how the learner will work, templates for Python structures are included for major classes being built. The structure of the code in this project will closely resemble what is used for the TensorFlow Implementation Deep-Q Cart-Pole Game in the Udacity Machine Learning Engineer Nanodegree classroom[21].

VII.IV.I ALGORITHM

A high-level view of Deep-Q algorithm is outlined in III.I Pseudocode. This logical structure will be called from the `DeepQ()` class.

```
import tensorflow as tf

class DeepQ:
    def __init__(self, learning_rate=0.01, state_size=4,
                 action_size=2, hidden_size=10,
                 name='DeepQ'):
        # state inputs to the Q-network
        with tf.variable_scope(name):
            self.inputs_ = tf.placeholder(tf.float32,
                                          [None, state_size],
                                           name='inputs')

            # One hot encode actions to later choose Q-value for the action

            # Target Q values for training

            # ReLU hidden layers

            # Linear output layer

            ### Train with loss (targetQ - Q)^2
            # output length 2, for two actions
```

[21] (Udacity)

VII.IV.II AGENT

The `Agent()` class will consist of functions that call a decision whether or not to flap, and then update the experience replay.

```
class Agent(object):
     def flap(self, game):
         # Returns either flap (1) or not flap (0) based on DeepQ() output

     def update(self, game):
         # Called at the end of each episode to update neural net based on state
```

`Agent()` can then be run n times over, enabling it to carry out the trials and learning.

```
class PlayOver(object):
    n = 250
    agent = Agent()
    for t in range(n):
        game = Flappy()
        while not game.over():
            flap = agent.flap(game)
            game.action(flap)
        agent.update(game)
```

VII.IV.III REPLAY

The experience replay memory will be initialized as a simple deque data structure. As the agent takes random actions, it will populate with $\langle \boldsymbol{s}, \boldsymbol{a}', \boldsymbol{r}, \boldsymbol{s}' \rangle$ tuples, which in turn direct the agent towards higher quality actions.

```
from collections import deque

class Cache():
    def __init__(self, max_size=1000):
        self.buffer = deque(maxlen=max_size)

    def add(self, experience):
        self.buffer.append(experience)

    def batch(self, batch_size):
        idx = np.random.choice(np.arange(len(self.buffer)),
                               size=batch_size,
                               replace=False)
        return [self.buffer[ii] for ii in idx]
```

VII.V SUMMARY

- **PROBLEM** II Problem Statement
- **LEARNER** Deep Q Convolutional Neural Network
- **ALGORITHM** $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left(R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right)$

REFERENCES

Bajaj, P. (2015, October 29). *Reinforcement learning.* Retrieved from Geeks for Geeks: https://www.geeksforgeeks.org/what-is-reinforcement-learning/

Brownlee, J. (2013, December 23). *How To Define Your Machine Learning Problem.* Retrieved from Machine Learning Mastery: https://machinelearningmastery.com/how-to-define-your-machine-learning-problem/

Convolutional neural network. (2019, February 22). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Convolutional_neural_network

Convolutional neural networks (CNN). (2019, February 23). Retrieved from skymind: A.I. Wiki: https://skymind.ai/wiki/convolutional-network

Deep learning. (2019, February 24). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Deep_learning

Deep reinforcement learning. (2019, February 25). Retrieved from skymind: A.I. Wiki: https://skymind.ai/wiki/deep-reinforcement-learning

Flappy Bird. (2019, February 22). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Flappy_Bird

Oppermann, A. (2018, October 28). *Self Learning AI-Agents Part II: Deep Q-Learning.* Retrieved from Towards Data Science: https://towardsdatascience.com/self-learning-ai-agents-part-ii-deep-q-learning-b5ac60c3f47

Q-learning. (2019, February 20). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Q-learning

Reinforcement learning. (2019, February 18). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Reinforcement_learning

Silver, D., Hubert, T., & Schrittwieser, J. (2018, December 7). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science Magazine, 362*(6419), pp. 1140 - 1144. Retrieved from Science Magazine: http://science.sciencemag.org/content/362/6419/1140

Simonini, T. (2018, April 10). *Diving deeper into Reinforcement Learning with Q-Learning.* Retrieved from freeCodeCamp: https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe

Twitch Plays Pokémon. (2019, February 23). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Twitch_Plays_Pokémon

Udacity. (n.d.). *TensorFlow Implementation.* Retrieved from Deep-Q Learning: https://tinyurl.com/udacitydeepq

Verma, S. (2017, September 3). *FlapPy Bird.* Retrieved from Github: https://github.com/sourabhv/FlapPyBird