WHITNEY KING
MACHINE LEARNING ENGINEER NANODEGREE
MARCH 4TH, 2019

CAPSTONE PROJECT

TABLE OF CONTENTS

I DEFINITION

I.I PROJECT OVERVIEW

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.

One of the first game playing agents to garner a large amount of attention was Google's *AlphaGo*; designed to beat the best human *Go* players in the world using a combination of supervised 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 only deep-reinforcement learning; its training was done with *zero* human opponents or prior domain knowledge (i.e. no dataset)[1].

Reinforcement learning (RL) is a model-less machine learning paradigm, focusing on performing *actions* which maximize *rewards* in a specific *environment*[2]. **Deep-Q** reinforcement learning employs **convolutional neural networks (CNNs)**, multi-layer neural networks popular for image and text processing, to calculate the best reward for any given action/state[3]. With one of the most classic applications of reinforcement learning being game playing agents, as well as my career industry of interest being video games, I wanted to train an RL agent to play a well-recognized and easily available game for this project; *Flappy Bird*.

[1] (Silver, Hubert, & Schrittwieser, 2018)
[2] (Bajaj, 2015)
[3] (Simonini, 2018)

I.II Problem Statement

Build a Deep-Q reinforcement learning agent to play the game Flappy Bird

This project was designed to be a self-playing reinforcement agent with no opponents or prior domain knowledge; thus, *it did not use a dataset*. To train the model, I modified a popular game clone, *FlapPy Bird*[4], which provided most of the functionality needed for the agent, and ran that on top of a custom `keras-rl` infrastructure.

This clone was originally truncated for reinforcement learning in Kevin Chen's Deep Learning Flappy Bird on GitHub. I ended up heavily editing the game code, and performing a complete overhaul on the agent and model architecture, and then implementing an entirely new logging and metrics structure for the analysis.

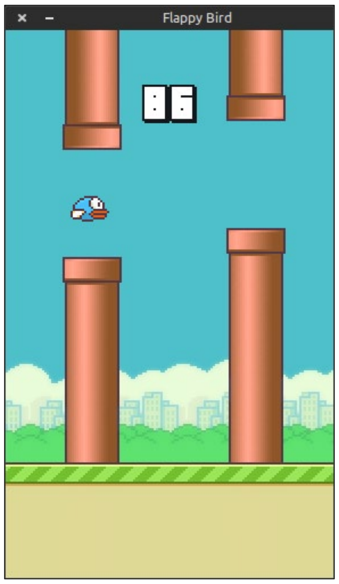


FIG 1: *FLAPPY BIRD* SCREENSHOT (VERMA, 2017)

II.I Game Environment

- automatically moves →, while the agent controls moving ↑/↓ to avoid
- Each *always* have the same gap distance, based on difficulty set
- Level has no set end; continues scrolling to the right until the agent loses
- Each successfully cleared earns **1 point**, but when pipes are touched , *game is over*
- Medals are award based on score at the *end* of the game for humans, where a score of 40 or higher is considered excellent performance

Flappy Bird was chosen because it's *replicable by design, has an observable task, and consistent, easily defined structure*. These are all characteristics of a good environment to baseline a learner, as well as an ideal candidate for this project. Mitchell's Formalism[5] was used to break down learning to play *Flappy Bird* down as a machine learning problem:

Task	Play the Game Flappy Bird
Experience	Reinforcement Learning Through Repeated Self-Play
Performance	Achieve Maximum Score

I.II.I Strategy

1. Clone FlapPy Bird repository using Git, then modify code for CNN ingestion
2. Code Deep-Q RL agent in Python using `keras-rl`, supported by a `TensorFlow` backend
3. Train and test agent by iterating through episodes using random starts and experience replay memory
4. Run benchmarks while agent is working, tweak as necessary until it is learning
5. Perform analysis and create visualizations detailing findings

I.III Metrics

To establish an appropriate reward structure while testing and training the agent, a handful of metrics were created to generate variable rewards, and ultimately help guide the learning process better. Additionally, several metrics were generated and recorded to

[4] https://github.com/sourabhv/FlapPyBird
[5] (Brownlee, 2013)

keep tabs on the model performance during the training process. I'd originally intended to use distance between the center point of the bird and the center point of the closest gap as one of the performance metrics, however in the long run ended up cutting this feature to get other more important parts of the agent working better. Between the *step distance*, *target distance*, and *target delta*, reward values were already pretty consistently in line with where they needed to be for the agent to learn, so continuing to pile more math onto the reward function wasn't necessary.

I.III.I Reward

I.III.I.I Scaled Step Distance

Since the ultimate goal of accumulating the highest score possible is reliant on getting as far in the game as possible, it made sense to generate a measurement scaling step number against the episode score boosting the overall reward. This achieves the goal of increasing the value output the further makes it in the episode.

$$\boldsymbol{stepDist = (1 + stepNb) \cdot (1 + score)}$$

I.III.I.II Target Delta

To generate a penalty modifier, the delta (***Δ***) between the episode score and the target score was calculated. The higher scores, the smaller ***Δ*** will be, which shrinks the overall penalties levied. If the agent doesn't cross into the first pipe gap or crashes, the penalty is multiplied to increase the significance of the penalty received.

$$\boldsymbol{\Delta = target - score}$$

I.III.I.III Target Distance

To get more variation in the output reward values, I wanted to add a measurement scaling the position of the player against their current score and target score. This calculation ended up fairly complex by the time it started generating appropriate values.

$$\boldsymbol{targetDist = \frac{\sqrt{\sqrt{stepDist \cdot step} \cdot |(-target^2 - score)|}}{(1 + \Delta)}}$$

I.III.II Training

To track the performance of the model during training and ultimately gain much of the interesting knowledge about what the model is doing, most of the metrics had to be generated. Custom losses were calculated in a `Lambda()` model layer for some of the training trials, before switching to a double train/test model without the `Lambda()` layer. Additionally, a few custom confusion matrix-based metrics from the `keras-metrics` Python package were implemented, however the `train_on_batch()` function doesn't return any values for anything other than loss. Since the customized `fit()` function results in a memory leak, moss was the primary training performance evaluation metric used.

I.III.II.I Accuracy

This was the single out of the box metric provided by Keras that was added to the metrics. This measure the ratio of correctly predicted values, *true positives* and *true negatives* ($\boldsymbol{TP}$ and $\boldsymbol{TN}$) against the total sample size. This is a very intuitive metric that provides a lot of data, but it is important to use other metrics as well to gain a more well-rounded understanding of what's happened[6].

[6] (Shung, 2018)

I.III.II.II Precision

Precision calculates the ratio of *correctly predicted positives*, to the *total predicted positives*, and helps ensure a low rate of *false positives* ($\boldsymbol{FP}$).

$$\boldsymbol{precision} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FP}}$$

I.III.II.III Recall

This metric is sometimes referred to as sensitivity, and measures the ratio of *predicted positives* against all samples, focusing the lens instead on *true positives* ($\boldsymbol{TP}$).

$$\boldsymbol{recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP} + \boldsymbol{FN}}$$

I.III.II.IV F1-Score

F1 is a good metric to include, since it takes both $\boldsymbol{TP}$ and $\boldsymbol{FP}$ values into account by calculating the weighted average of *precision* and *recall*, and generally gives better information than accuracy alone.

$$\boldsymbol{F1} = 2 \cdot \frac{\boldsymbol{precision} \cdot \boldsymbol{recall}}{\boldsymbol{precision} + \boldsymbol{recall}}$$

I.III.II.V Loss

Loss is ultimately a numerical representation of how good or bad a given prediction is on a given sample. If the option to run dueling networks was selected, the agent calculated training loss as a *mean squared error* of updated Q-values in a custom `Lambda()` layer inserted into the model when it's compiled at runtime, based on either the *max*, *average*, or *naïve* squared loss of each example across the training session. Otherwise, loss is a standard mean square average. The closer the loss value is to zero, the better the model's prediction is; during training, the goal was to fit the model with a set of weights and biases leading to the lowest possible loss. I opted *not* to use dueling networks when performing the finalized training runs, and instead stuck with a double DQN running train and test models simultaneously, *so losses ended up being a regular mean squared error*. This decision was made because there were problems getting `keras` to save the model weights in a way that could be reloaded, and this turned out to be due in part to the dueling network implementation is stateful.

II ANALYSIS

II.I EXPLORATORY VISUALIZATION

The transformations each frame of the game goes through before being passed into the neural network for analysis are somewhat complicated. I spent a good portion of the preliminary preparation stages for this project trying to get a solid grasp on what and why was happening to take normal images and turn them into something computationally interesting for AI. *To lay the groundwork for a solid understanding of these transformations, as well as support the use this generative style of deep learning, I created a diagram, shown in* *Appendix A: Preprocessing Transformations Visualization*, that illustrates the what's happening to each frame to it get it ready for the neural network to use in training, and thus generate data we can use for analysis.

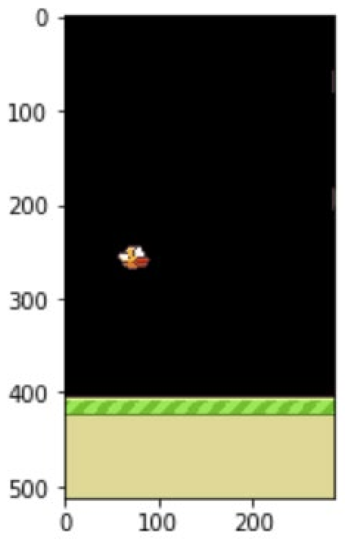


FIG 2: *ORIGINAL PYGAME FRAME*

To take the emulator screen from `pygame` into the neural network, it's first passed through the Python imaging library `OpenCV`. This process changes the color sampling from RGB to BGR, and flips the axis from $(\boldsymbol{X}, \boldsymbol{Y})$ to $(\boldsymbol{Y}, \boldsymbol{X})$. In the grand scheme of things, this color change and inversion did not matter for our agent; its primary concerns are with progress and collision boundaries. However, it's easier to process what is happening from the human side of the equation when the axis is not inverted, so changing the image projection back to (X, Y) was done as one of the transformations.

Before passing the image back to the agent, OpenCV commands are also run to resize the image into an [80px, 80px] square. This does cause some distortion loss, but not enough to negatively affect the learning. Performing this resize helps streamline the neural network architecture and speed up the processing time. Since color isn't of consequence to the agent and would cause additional processing bloat, color sampling is changed to grayscale, and then finally split into two tone values, black or white. This ends up with a frame where the background is shown in black, while the base, and are shown in white.

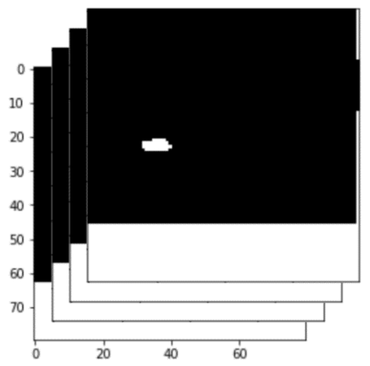


FIG 3: *TENSOR STATE [1, 80, 80, 4]*

Finally, the user defined `state_size` parameter value is used to determine the sequential window length of episode frames to stack to create a state, which is then transformed into a 4-dimensional Tensor for the model to analyze. The diagram shown illustrates a state size of 4, however this value can be any reasonable user defined integer.

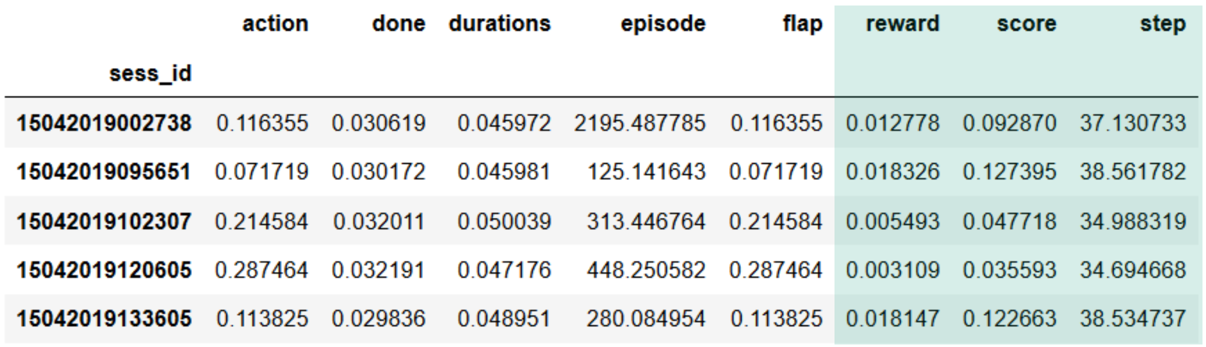
II.I.I Logs

Vital statistics and descriptive information about game state were kept in each step to enable analysis of the model's settings, output, and performance. These are written to JSON files at the end of each training session, then imported into pandas `DataFrames`, saved in a more efficient format using `pickle`, and then finally analyzed using `Python` and `Jupyter Notebook`. In addition to the metrics kept each step the model trains, `DataFrames` allow the information to be grouped by episode and session for aggregate totals.

Unfortunately, I was missing one line of code in the `backward` method that was supposed to copy the metrics over into the logs. I didn't catch that these values were null until near the end of training since I'd been spot checking my work in terminal; this was extremely disheartening, and an immense setback on the analytics side of this project. This code block is now fixed, and appends the custom metrics to the default mean squared error loss, then sends that to the logs each step.

```
if type(metrics) is list:
    metrics = [m for idx, m in enumerate(metrics) if idx not in (1, 2)]
    metrics += self.policy.metrics                # this was missing :(
```

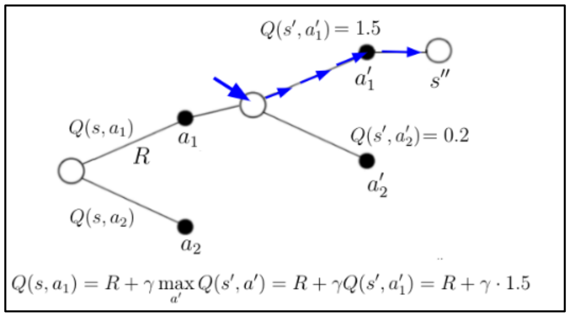
By the end of training, I'd only collected a small fraction of the metrics I'd set out to, and in retrospect should have reserved a lot more time to allow for additional work that's needed fixing bugs in the customized callbacks. Highlighted below from the step logs are the mean reward, scores, and steps across the last 200,000 training steps (roughly 6,760 episodes), *and even without in-depth analysis, it's evident at-a-glance that there are no significant improvements in any categories*.



	action	done	durations	episode	flap	reward	score	step
sess_id								
15042019002738	0.116355	0.030619	0.045972	2195.487785	0.116355	0.012778	0.092870	37.130733
15042019095651	0.071719	0.030172	0.045981	125.141643	0.071719	0.018326	0.127395	38.561782
15042019102307	0.214584	0.032011	0.050039	313.446764	0.214584	0.005493	0.047718	34.988319
15042019120605	0.287464	0.032191	0.047176	448.250582	0.287464	0.003109	0.035593	34.694668
15042019133605	0.113825	0.029836	0.048951	280.084954	0.113825	0.018147	0.122663	38.534737

II.II ALGORITHMS & TECHNIQUES

This proposed solution to the outlined machine learning problem implemented a linearly annealing Deep-Q algorithm, which iteratively self-plays *FlapPy Bird*. Deep-Q learning revolves around an *action-value* function, written as $\boldsymbol{Q(s,a)}$; this is the expected *reward* $\mathbb{R}$ from starting in *state* $\boldsymbol{s}$ from all available states $\boldsymbol{S}$, taking *action* $\boldsymbol{a}$ from all actions $\boldsymbol{A}$, then following *target policy* $\boldsymbol{\pi}$: $\boldsymbol{Q^{\pi}} : \boldsymbol{S} \times \boldsymbol{A} \rightarrow \mathbb{R}$. The agent then searches for an optimal environment *behavior policy* $\boldsymbol{\mu}$ by maximizing total $\boldsymbol{Q}$-value of each successive state/action pairing[7]:



$Q(s', a'_1) = 1.5$
a'_1
s''
$Q(s, a_1)$
a_1
R
$Q(s', a'_2) = 0.2$
a'_2
$Q(s, a_2)$
a_2
$Q(s,a_1) = R + \gamma \max_{a'} Q(s', a') = R + \gamma Q(s', a'_1) = R + \gamma \cdot 1.5$

FIG 2: ***AGENT FOLLOWING GREEDY POLICY FOR Q* (OPPERMANN, 2018)**

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

Deep-Q agents both optimize and follow a behavior policy by simultaneously creating and processing layers of inputs and outputs[8]. The agent begins with a blank slate, then information is tracked in *experience replay* tuples $\boldsymbol{\langle s, a', r, s' \rangle}$, which serve as a simple cache for logging the *state* ($\boldsymbol{s}$), *action taken in* $\boldsymbol{s}$ ($\boldsymbol{a'}$), *next state* ($\boldsymbol{s'}$), and *reward for* $\boldsymbol{a'}$ ($\boldsymbol{r}$)[9]. Knowing the $\boldsymbol{Q}$-value for any $\boldsymbol{(s,a)}$ pairing 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[10]:

$$\boldsymbol{Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\mathbb{R}_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)}$$

II.III.I PSUEDOCODE

Oppermann outlines the generalized pseudocode used in designing the agents fit functionality, as well as its greedy-Q behavior policy. All source code related to this functionality is documented in the companion `Notebook` to this paper, `BetaFlapZero.ipynb`.

```
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 ∈ select random action $\boldsymbol{a_t}$
        Otherwise select $\boldsymbol{a_t = \max a\, Q^{*}(s_t, a; \theta)}$
        Execute action $\boldsymbol{a_t}$ in emulator and observe reward $\boldsymbol{r_t}$ and state $\boldsymbol{s_{t+1}}$
        Store transition $\boldsymbol{\langle s_t, a_t, r_t, s_{t+1} \rangle}$ in $\boldsymbol{D}$
        Set $\boldsymbol{s_{t+1} = s_t}$
        Sample random minibatch of transitions $\boldsymbol{\langle s_t, a_t, r_t, s_{t+1} \rangle}$ from $\boldsymbol{D}$
        Set $\boldsymbol{y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max a'\, Q(s_t, a_j; \theta) & \text{for nonterminal } s_{t+1} \end{cases}}$
        Perform gradient step on $\boldsymbol{\left(y_j - Q(s_t, a_j; \theta)\right)^2}$
    end for
end for
```

[7] (Deep reinforcement learning, 2019)
[8] (Oppermann, 2018)
[9] (Simonini, 2018)
[10] (Oppermann, 2018)

II.III.II Parameters

To help keep track of and fine-tune the values for the large number of customizable parameters in the agent and model, I ended up generating an organized parameter dictionary in a separate file, `flappy_inputs.py`, and then using that to input the hyperparameters into the agent when it's called. Many of these parameters don't have an effect on every run, but are involved in controlling one aspect or another with how the agent, model, memory, and emulator are setup. All vital parameters will be explained alongside the implementation information, and any parameter that went into training the model is specifically labeled in IV.I.I Model Parameters. *An example version of the parameter dictionary is shown in Appendix B: Parameter Dictionary.*

II.IV Benchmark

For a continuous control agent such as the one utilized in this project, there is not a set standard for baselining, and not any generalized feature engineering to be concerned with because it doesn't begin with a dataset. However, since there wasn't any prior domain knowledge, and the game has some built in tiers that act as performance benchmarks, it was assumed that the agent is learning when the average score it's attained each game is climbing, and that it's learned well when it consistently achieves a score of $\mathbf{40}$ or higher.

$\boldsymbol{Scores < 10}$	unsatisfactory performance
$\boldsymbol{Scores >= 40}$	exemplary performance

III Methodology

The framework and code for the DQN agent and model is broken out into four primary Python packages, which combined together I've named *`BetaFlapZero`*. Each of these files contains a class with a customized module that overrides settings in one of the prebuilt `keras` or `keras-rl` modules. To keep the length of this paper reasonable, I've avoided including large chunks of source code in line with this write up, and instead moved previews of substantial code snippets into a `Jupyter Notebook`, `BetaFlapZero.ipynb` where they can be browsed together in a way that makes sense.

`flappy_util.py`	`Utility()` class for storing repetitive and small functional tasks
`flappy_processor.py`	`FlappyProcessor()` class for preprocessing step and reward data
`flappy_callback.py`	`FlappySession()` class for handling logging tasks during training session
`flappy_beta.py`	`NeuralNet()`, and `FlappyDQN()` to build the agent and fit the model

III.I Requirements

This project has the following software dependencies for `Python 3.5.x`:

`keras 2.2.2`	expansive open source library for neural networks
`keras-rl`	open source expansion library for RL learning in Keras
`keras-metrics`	open source expansion library for ML metrics in Keras
`TensorFlow-GPU 1.10`	suite of open source deep learning libraries for GPU processing
`CudNN 7/CUDAToolKit 9`	powerful GPU processing suite for NVIDIA graphics cards (if using GPU*)
`PyGame`	open source game development platform for Python
`OpenCV`	open source image processing library
`Scipy`	open source scientific computing package suite for Python
`NumPy 1.14.5`	scientific computing and calculation package for Python

To create an Anaconda environment preconfigured with the correct dependencies for this project:

1. Download the requirements file **`environment.yml`** to `%ANACONDA_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
   ```

III.II PROCESSOR (PREPROCESSING)

All preprocessing for frame, reward, step, action, and state data was done through the customized `FlappyProcessor()` class in `flappy_processor.py`. Preprocessing for frame and reward data is called through the `process_step()` function:

```
def process_step(self, o, r, d, i):
        observation = self.process_observation(o)
        reward = self.process_reward(r)
        return observation, reward
```

III.II.I OBSERVATION (FRAME)

The transformations used are fairly standard when resampling images to pass through convolution layers to increase processing efficiency. That said, this processing used in the `process_observation()` function was rearchitected and streamlined:

```
def process_observation(self, observation):
        x_t = cv2.transpose(observation)    # flip (Y, X) to (X, Y)
        x_t = cv2.cvtColor(x_t, cv2.COLOR_BGR2GRAY)
        x_t = cv2.resize(x_t, (80, 80))
        x_t = cv2.threshold(x_t, 1, 255, cv2.THRESH_BINARY)[1] # B/W
        return x_t
```

III.II.II ACTION

The processing that occurs within the `process_action()` transforms each action from a scalar action index into a 1-dimensional binary array. The `keras-rl` backend uses a scalar index representing the action selected when it makes predictions, but the rest of the agent uses the binary array. Since the game has two possible actions, that means the action index passed in is either $\mathbf{0}$ (don't flap), or $\mathbf{1}$ (flap). In a binary array, idea can be represented with $[\mathbf{0},\mathbf{0}]$ as an empty array state, $[\mathbf{1},\mathbf{0}]$ when no action is selected, and $[\mathbf{0},\mathbf{1}]$ when flap is selected.

```
    def process_action(self, action, nb_actions):
        '''Transform action from scalar index into binary array'''
        a_t = np.zeros([nb_actions])
        a_t[action] = 1                     # flag action for env
        flap = False if a_t[0] else True
        return a_t, flap                    # action index for experience tuple
```

III.II.III REWARDS

No matter the start position of at the beginning of the episode, collision with the $\boldsymbol{X}$ axis of the first is always at frame $\mathbf{49}$ if the agent does *not* make it into the gap. Thusly, the reward structure was divided so a large penalty is calculated unless frame $\mathbf{50}$ is reached; this was placed in the customized `process_reward()` method. Once past step $\mathbf{50}$, the reward earned each step increases via the weighted metrics, so there is no question for the agent that progress (e.g. max Q values) is the goal.

```
def process_reward(self, reward):
        step_dist = (1 + reward['step']) * (1 + reward['score'])
        tar_delta = reward['target'] - reward['score']
        tar_dist = np.sqrt(   # scaled reward booster
            np.abs((-reward['target'] ** 2 - reward['score'])) *
            np.sqrt(step_dist * reward['step'])) // (1 + tar_delta)
        if reward['step'] < 50:
            mul_penalty = np.sqrt(tar_delta)
        else: mul_penalty = 1
        self.msg = 'Danger zone!'
        award = (tar_dist ** 4) / mul_penalty  # base reward scales w/ dist/score
        if reward['step'] > 49:
            award += 1
        else: award = award / ((1 + tar_delta) * mul_penalty)
        if reward['terminal']:
            award = -100
            self.msg = 'Boom!'
        elif reward['scored']:    # multiplier for scoring
            award += (
                np.sqrt((1 + tar_dist ** 2) * (1 + step_dist ** 2)))
            self.msg = 'You scored!'
```

III.II.IV State Batch

The agent uses the `train_on_batch()` feature of `keras`, which is run by a model with double training/testing networks and an `AdaDelta` optimizer. To increase memory efficiency while training, ***observation***, ***action***, ***reward*** and ***terminal*** are logged inside a `keras-rl` `SequentialMemory()` object as individual `RingBuffer()` objects. States are generated on-demand from stored experiences when a batch is selected for training:

```
def process_state_batch(self, batch):
        # unpack experiences into individual lists
        s0, r, a, t, s1 = [], [], [], [], []
        for e in batch:
            s0.append(e.state0)
            s1.append(e.state1)
            r.append(e.reward)
            a.append(e.action)
            t.append(0. if e.terminal1 else 1.)
        s0 = np.array(s0)
        s1 = np.array(s1)
        t = np.array(t)
        r = np.array(r)

        return s0, r, a, t, s1
```

III.II Environment (Game)

III.II.I Emulation

This project utilized (and heavily modified) a popular clone `FlapPy Bird`[11], built in Python using pygame and available on GitHub under the MIT License[12].

[11] https://github.com/sourabhv/FlapPyBird
[12] (Verma, 2017)

III.II.II Emulator Implementation

For the agent to run `Flappy Bird`, both `flappy.py` and `setup.py` needed to be rebuilt for a looping DQN agent. Some of the highlights of the reengineering efforts on the game emulator are:

- Loading game assets and collision boundaries (e.g. the `HITMASK`) was broken out into `flappy_load.py`
- Welcome screen, quit options, and sounds are all inconsequential to the agents' actions and performance
- All code blocks related to `showWelcomeAnimation()`, SOUNDS, and QUIT were removed
- Randomized difficulty selection, and gym-like methods, `render()`, `reset()`, and `close()` were added to the emulator
    - Randomized bird and pipe color for each new game, and I added a few new color ways for visual interest
- Background images were turned solid black, eliminating noise, and creating a clear collision boundary
- Added tracking for pipe gap locations, and logging output to agent for with each step, as well as save/exit keypress events

The rest of what was already written was significantly streamlined to prepare it for ingestion into the agent, as well as generalize cleanup efforts, such as bringing naming conventions into consistency. The `Environment()` class was added to keep track of progress, and most of the functional parameters were moved into the `__init__()` method. Remaining code blocks were refined and encapsulated into a functional template similar to the `gym` architecture. A new `step()` method was added to handle the processing and output information, then preserve each frame as an image to be transformed and then saved in a state.

III.III Agent & Model

III.III.I Challenges

When I selected this particular project, I settled on doing something that was definitely possible. My worst nightmare was coming up with some grandiose project and build that would be full of exciting ideas, but ultimately be out of reach while building my confidence in both machine and deep learning. When I really got down to research, I discovered that many versions of this project have been done, and yet when dissected they all had a lot of very glaring similarities; down to some of the same bugs and inefficiencies from project to project. This is both a beauty and a curse in the double-edged world of open-source development.

Designing the reward processing systems was one of the most challenging aspects of this project. At a very base level, there isn't necessarily a need for a complicated reward system in order to get the agent learning, however I really felt one was worth consideration. I spent almost two weeks struggling to figure out why rewards were not scaling the way the math was saying they should, and that ended up being largely in part due to not having a full grasp of `pygame` functionality. It's not a challenge that's insurmountable by any means, but it's a nice-to-have feature not a must have one, and was left out of scope for the sake of time. This left a much more truncated rewards system with less variability in the outputted values, but ultimately a more sensible training experience for the AI. What I learned during the struggle of designing the rewards structure, is not to get so focused on 'whether or not I can', and instead focus on 'whether or not I should'. There is a very good reason Occam's Razor is harrowed as a cornerstone philosophy in programming, and time is a very finite resource.

One of the hardest parts of this was getting `TensorFlow` working with GPU processing on an RTX 2070 graphics card. Installing it is one thing, but installing it so the GPU processing actually kicks in is another. There were a lot of packages involved in this agent, all vying for different required versions, and ultimately, I had to build `TensorFlow` from source using `bazel` and Linux subcommands on Windows to make it work; only to promptly have that stop working towards the end of the project when Visual Studio updated. At best, I was only able to get this model to use about 5% of the GPUs processing power during peak, though that isn't too surprising considering the small scale of the model and relative simplicity of the data. This is a saga that could be worthy of its own paper, however I was fortunate to find a handful of great resources as to which command sequences to run to actually make some progress towards having a functioning `TensorFlow` environment with an RTX graphics card on Windows[13]. Many problems still persist through that brute force workaround, and the Python environment is far from what I would consider stable or optimal. I really enjoyed the features and capabilities the `keras-rl` package offered on top of `keras`, but came out of this really frustrated with `TensorFlow`, to the point that given enough time to start anew on this, I would have instead rebuilt entirely in `PyTorch`.

III.III.II AGENT IMPLEMENTATION

For the agent framework, I built out customized `Callback`, `Processor`, `Model`, and `Agent` classes, including rewriting the `fit()`, `forward()`, and `backward()` `keras` methods specifically for *BetaFlapZero*. The meat and potatoes of the agent is split into two main classes: `NeuralNet()`, and `BetaFlapDQN()`. The agent runs a session, logged in `FlappyCallback()`, then populates its hyperparameters, calls a new `Environment()` object, builds the experience replay memory, and finally builds and compiles a convolutional neural network model `NeuralNet()` with user defined parameters from `Inputs()`.

III.III.II.I BUFFER

Most small-scale examples for this kind of learning use a deque data structure to store the replay experience memory, but in my professional opinion deques should be avoided for this kind of application because they are slow when seeking at scale. I experimented with using an in-memory data structure like `redis`, however ended up settling on the `keras-rl` package, because it had well designed, efficient, and easily customizable objects tailored for reinforcement learning with `keras`. For the memory, I used an out-of-the-box `keras-rl` `SequentialMemory()` class, which in turn holds `RingBuffer()` objects holding the experience replay lists as separate elements it draws from at time of sample.

III.III.II.II MODEL

When it came to the neural network architecture, I did a fair amount of experimenting. I tried writing the network using `TensorFlow`, `keras` Sequences, and `PyTorch`, before finally settling on just writing the model layer by layer in `keras`. I really liked `PyTorch`, but was really impressed with the extensibility of `keras`, especially in regard to reinforcement learning agents, as well as the ease of understanding the model workflow.

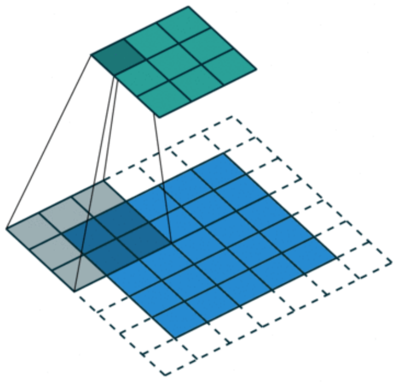


FIG 5: ***3 X 3 FILTER PASSED OVER STATE***
(Kaggle: Amar Jeet)

The model, shown in Appendix C: Model Neural Network Layers, takes a state $\mathbb{S}$ or batches $\mathbb{B}$ of states, $[\mathbb{B}, \mathbf{80}, \mathbf{80}, \mathbb{S}]$, passes a series of filters and transformations over the data, and *returns the same number of dimensions as possible actions;* $\mathbf{2}$. For the final training in *BetaFlapZero*, `state_size` was set to $\mathbf{8}$, `filter_size` was set to $\mathbf{32}$, and `batch_size` was set to $\mathbf{64}$, resulting in an initial kernel of $\mathbf{8x8}$. These parameters values had a nice balance between processing speed and providing the model a good amount of information to make decisions.

III.III.II.III POLICY

This agent uses a `LinearAnnealed()` outer policy, and an `EpsilonGreedyQ()` inner policy with customized parameters while it trains and predicts with the *training model*. The *test model* (which does *not* get trained), makes predictions and validates the outcomes using a much simpler `GreedyQ()` policy.

[13] (Mandal, 2018)

```
self.policy = LinearAnnealedPolicy(
     inner_policy=EpsGreedyQPolicy(
            eps = self.value_max
    attr='eps',
    value_max=self.value_max,
    value_min=self.value_min,
    value_test=self.alpha,
    nb_steps=self.anneal)
self.test_policy = GreedyQPolicy()
```

For the majority of training, I'd left the game difficulty set to *hard*, under the assumption that it should learn to play the hardest states possible. However, after getting the model tuned to the point learning should have been somewhat evident (over 50 thousand episodes), the agent was still struggling with picking up the signals to aim for the gap; flapping too much, even when epsilon was initialized at $.1$, and annealed down to $.0001$ over the course of a million steps.

My initial workaround was to limit the number of frames per action, but it's not an ideal solution in the long run. The original version of `Flappy Bird` comes in three difficulty levels, *easy*, *medium*, and *hard* however the `pygame` emulation had only implemented settings for hard. The only difference in difficulties is the size of the gap between the pipes: $200px$ for easy, $150px$ for medium, and $100px$ for hard. Towards the end of experimenting with the training flow, I added a function that selects a random difficulty each time an episode is initialized. This added a lot more variability to the replay experience, and having those larger gaps on the playfield helped the agent home in on the gaps and get further along. It wasn't until I changed the size of the pipe gaps in play that I saw any real sustained progress beyond the second set of pipes.

III.III.II.IV CALLBACKS

One of the most important parts of being able to understand what the model is doing; if it is learning and more importantly, if it is learning well, is to be able to process metrics and interesting metadata that gets populated while the model is training. There are a lot of metrics to consider in this model, from benchmarking performance, to model evaluation, to emulator output; and each of those needed to be investigated and validated. A customized `Callback` module, `FlappySession()` was created to handle the read/writes of data to JSON files at the beginning and end of each session, episode, and step. This provided a wealth of information to analyze and use in the evaluation states.

III.III REFINEMENT

I ran a lot of the preliminary training with the dueling network option turned on, which 'duels' by replacing the last `Dense` layer of the models' neural network, with a customized `Lambda` layer when the model is compiled. This layer performs a loss function on affected Q values in a given training batch based on a user defined parameter. This is a built-in feature of the `keras-rl` class `DQNAgent` and available inputs are `max`, `avg`, or `naive`. After discovering that this training methodology was stateful and impacted my ability to easily save model weights, I switched the model to a simpler setup with a double DQN, that uses test and target models to update the model target weights at a set interval; for the final training, model targets were updated every step. Overall, this was a good option to use for this type of learning as it enabled the model to resume from being stopped without losing the weights it had gained in previous sessions. I didn't notice any remarkable decrease in performance after switching the model training structure over to the double DQN. With the double DQN, predictions are made with the test model, which updates a target model at a user defined interval, and then the trainable model slowly converges on the updated targets.

During the model fitting the process, the agent starts each episode with a randomized seed of actions. In this case, for between 1 and 40 steps at the beginning of each episode the agent will perform random actions without commiting them to replay memory. This became an extremely valuable feature very quickly, because there is a huge amount of noise in the first 49 frames via redundant crashing into the top or bottom of the pipes. Staggering the step, action, and position at which the model begins training really helps mix up the actions on the screen and get more entropy in the experience replay buffer. To help build a buffer of richer experiences as minibatches for training, experience is always committed to memory in the event the agent makes it past step 49 in an episode. The notable improvements in training after upping the randomization seeding each episode (notably, stopping the agent from always flapping to the top) also laid the ground work for the idea to put an interval and condition on how often the replay experience saves to memory. I tested out a lot of variation in parameter values for memory interval, and ultimate found putting a reasonable interval value (less than that of the state size) on how often states save wasn't negatively impacting training. The final training saved experience every 4 steps, and used a `state_size` of 8.

While this was a relatively small-scale project, I ended up generating gigabyte sized logs when training less than ten thousand episodes. Through days of iterations with that early on, much of which had zero learning or logging to show for it thanks to weird settings or unnoticed bugs, time didn't allow for really in-depth hyper-parameter optimization with something like `Talos` or `Scikit-Learn` as I'd originally planned as a stretch goal. In general, I believe the parameters used are pretty good, but could absolutely use tuning, especially since I changed the neural network layers significantly from the Google DeepMind research paper that most DQN agent model architecture is generally based on[14].

IV RESULTS

IV.I MODEL EVALUATION & VALIDATION

Most model selections and optimizations were made via trial and error, in addition to researching what worked for other DQN agents.

IV.I.I MODEL STRUCTURE

Fit Type	Double DQN: test and target models w/ target update every iteration
Network Architecture	Appendix C: Model Neural Network Layers
Optimizer	Adadelta
Loss	Mean Squared Error
Metrics	Accuracy, Precision, Recall, F1 Score

IV.I.I MODEL PARAMETERS

State Size	8
Filter Size	32
Batch Size	64
Replay Buffer Size	100,000
Replay Buffer Interval	4
Target Update Interval	1
Training Interval	1
Action Repeat	1
Initial Epsilon	0.1
Terminal Epsilon	0.0001
Epsilon Anneal Steps	500,000

[14] (DeepMind, 2015)

Epsilon Test	0.025
Gamma	0.99
Warmup Steps	1,000
Max Random Episode Steps	40
Gamma	0.99
Warmup Steps	1,000

For the model architecture selection, I experimented with many different layers and several activations. The final network was built with `Convolution2D` layers, followed by `LeakyReLU` activation layers. `MaxPooling2D` and `BatchNormalization` were both integrated by replacing the last `MaxPooling2D` layer typical DQN CNNs with a `BatchNormalization` layer to build a somewhat unique network. The network concludes with `ReLU` activation on duplicated, gradually decreasing `Dense` layers; these layers step down to the final `Linear` activated output, matching the action size of $\mathbf{2}$. In addition to the customized CNN, both `VGG16` and `Resnet50` models were tested, but neither of those seemed to get beyond always flapping to the top of the screen after a reasonable amount of training. For the optimizer, I found `Adadelta` to have the 'smoothest' training results, with a lot less issues getting stuck flapping too much than some of the other options like `SGD`, `RMSprop`, or plain old `Adam`. `Adadelta`, doesn't take a manually set learning rate, so this was one less parameter value to worry about tuning on top of the desirable performance, though it does come at the cost of a longer time to converge.

The model tuning came in the form of sensitivity analysis around the environment seed, as well as the reward parameters and logic; much of which unfortunately didn't make it into logs due to unseen bugs in code at that time. A lot of configurations were experimented with to try and pinpoint a good pattern to seed episodes not resulting in the agent always taking off with the same action from the same spot. What made the largest difference was randomizing not only the first action the agent trains on, but also at which step in the episode it begins training. I kept that randomization range less than the first terminal pipe in the game, seeding the first $\max_{\mathbf{n}} \mathbf{40}$ steps with entirely random actions, and no training.

This made a noticeable difference in the sensitivity of the agent, and the amount it reached at least the first pipe gap; as well as stopped it from getting stuck always flapping to the top of the screen when left to train overnight. Still, after training over fifty thousand episodes the agent wasn't converging on the gap, making any kind or improvement, or ever getting beyond a score of $\mathbf{2}$. I performed further input manipulation by randomizing the difficulty setting each episode starts with, thus changing the size of the gap between the pipes each episode. Not only did this significantly increase the frequency in which the agent scored, but also that of which it got a score $\geq \mathbf{2}$ on any of the difficulty modes. Overall, this improved model robustness, helped the model generalize better to unseen inputs, but didn't end up showing evidence of any form of learning. It goes without saying that larger gaps naturally lend themselves to the same behavior when random actions are taken (at least on easy and medium difficulties), but what this did succeed in doing was provide the agent with less noise and more entropy in the replay buffer to pull from during training. Given the opportunity to train continually for several days, maybe around a million episodes, I believe this model would get a lot closer to the initial benchmarks and expectations. As it currently stands, there have been some areas of improvement over some DQN implementations, but by in large, those are outweighed by the enormous number of flaws introduced from adding many layers of complexity to the agent. The solution isn't anywhere near robust enough to be considered an overall improvement over existing DQN agents. Coupled with the fact that the majority of useful data tracking performance metrics didn't make it into the logs until training was practically over, the results from the model cannot currently be trusted.

IV.II Justification

There were a lot of moments where I grappled with sunk cost fallacies and whether or not to try something new. On many occasions, I was rewarded with efforts that paid off (such as the improved replay buffer, defined agent architecture, and random episode seeds), and just as many where I had to finally cut my losses and say 'enough' (such as GPU processing, and gap distance), and yet others that didn't turn out the way I'd hoped (custom metrics turning out null with `train_on_batch()`, and the memory leak with the custom `fit()` method). The final solution did not come anywhere close to meeting the expectations set by my chosen benchmark, finding a significant solution that consistently plays *`Flappy Bird`* well, nor actually learning anything at all. This was partially due to my not understanding the amount of time would be lost to troubleshooting bleeding-edge hardware, partially due to several full agent architecture tear downs and rebuilds, and partially to my not understanding the time complexity involved in training something like this for tens of thousands of iterations. In the end, I ran out of time because of scope creep, and lost sight of making sure the information returned was useful, and the model weights were updating and saving as expected. I ran into far more problems than I'd expected, burned through the month extension I was given on this working at least 8 hours a day, and still have a laundry list of things I'd like to continue working on to make this agent into what it can and should be. The results were so glaringly unremarkable that there was no sense in even trying to calculate actual statistics on any of the data that did make it into the logs, because it was obvious based on simple at-a glance averages of the reward, score, and step logs that there was not any significant changes to be analyzed.

To say I'm disappointed in the outcome would be an understatement; however, my knowledge and fundamental understanding around the logic and data structures involved in deep learning processes has increased exponentially. That said, with a very limited action space, comparing the learning over all episodes was not significantly better than any progress seen based on purely randomized actions. There were moments when monitoring that it seemed like the agent may be learning, however there was no evidence to support those instances being anything other than random chance. For a model such as this one, 50 thousand-episode iterations just isn't enough to get it trained up on the action-space to make reliably good decisions. It would be really interesting to see what happens to the agents learning when letting it run for longer extended durations; as it stands, it never got trained to the point that it was able to converge.

V Conclusion

This process implemented a DQN reinforcement learning agent that attempted to learn how to play Flappy Bird at a level matching "good" human players. To execute the learner, I built out a custom agent and model using `keras`, `keras-rl`, and `TensorFlow`. I'd been looking forward to using this project to test new NVIDIA RTX graphics card hardware with this deep learning project, but quickly got in over my head trying to force it to work for this application when `TensorFlow` isn't updated to a place where it reliably supports this latest series of cards yet. A huge amount of time was lost troubleshooting hardware and start-stopping agent training to fix ongoing issues, that other important considerations had to be truncated or shelved, and some vital features, such as logging for performance metrics ended missing pieces because bugs not caught until too late. Training for this agent was extremely slow on CPU; had I anticipated this outcome, I would have changed the backend to `PyTorch`, moved the training environment to AWS, or picked a different project all together with less features and weights for the model to keep track of. What was most frustrating is the complete lack of metrics that ultimately came out of the prebuilt `train_on_batch()` function. All of the metrics populate with correct values when passed through the `fit()` function, but that option can't be reliably used for training due to an unresolved memory leak.

V.I Free-Form Visualization

By an unfortunate stroke of irony, the most interesting data visualization to come out of this cemented the agent's lack of learning that was observed over the course of training . From the data salvaged out of the tail of the training logs, we can see a pretty clear and consistent standard reward per episode, with small number of those appearing to have some moments of random luck, but not enough to be significant or worthy of additional investigation. This visualization stacks episodes across training sessions, then plots rewards for any episodes with a reward > 0.011. This cutoff point was chosen because it's score threshold between frame 49 and 50 if the agent makes it into the gap. The 'heat' spot at the lower left is where the most data was able to be pulled in. Overall, there is almost no variation seen in the spread of attained reward values, and no indicators of learning having occurred.

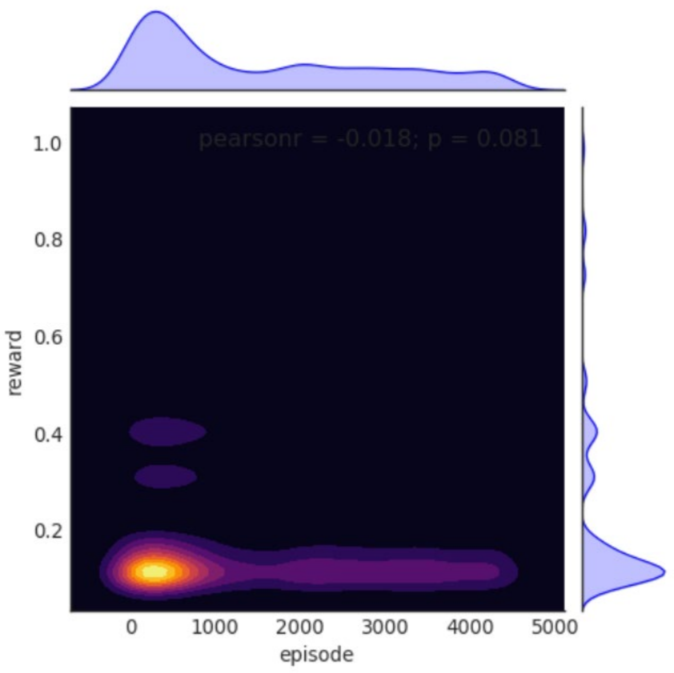


FIG 6: ***JOINTPLOT OF AGENT REWARDS PER EPISODE***

V.II Reflection

I was following relatively simplistic tutorials and examples to help build out this framework, and performance issues were apparent very early on. Much of what I ended up doing came from pouring over the class and method structures of the package source code files, reading through limited documentation of their intended functionality, and likewise with the other packages they call. The amount of scope creep was much larger than expected; I absolutely underestimated the amount of reworking I would be compelled to attempt while putting this together. It was a balancing game of tradeoffs, because performance, composition, and logging all drastically improved, but some features I was hoping to implement were left out for now.

When I'd originally scoped out this project, I had anticipated the agent learning how to navigate the obstacles within 25 – 50 thousand episodes. What I didn't anticipate was *not* being able to run the hundreds of iterations of testing involved with validating the model's progress on GPU. In the end, the GPU solution is still nowhere near stable and continues to drop my graphics card off the PCI bus thanks to versioning inconsistencies between environment packages. Almost two months of time was split between getting `TensorFlow` working with RTX, and rearchitecting everything into a more complex, but richly featured system.

All of these factors lead me down a road of rearchitecting the entire project three times on my quest to build an improvement over what already existed. Now, at least if there are remaining bugs and inefficiencies, they are *my* bugs and inefficiencies resulting from a great deal of effort. I do have the takeaway of a much better understanding around deep learning agents and models than I'd expected to attain when I selected `Flappy Bird` as my capstone subject. I do not believe that this solution in it's current for would be the best possible candidate to solve the problem statement, due to a combination of environment infrastructure problems and additional refinements needed in the training and rewards processes. However, it is a good starting point with some real areas of improvement over more simplified reinforcement learning models, so with some targeted focus and polish in the problem areas, it could absolutely be shaped into a decent generalized reinforcement learning game player.

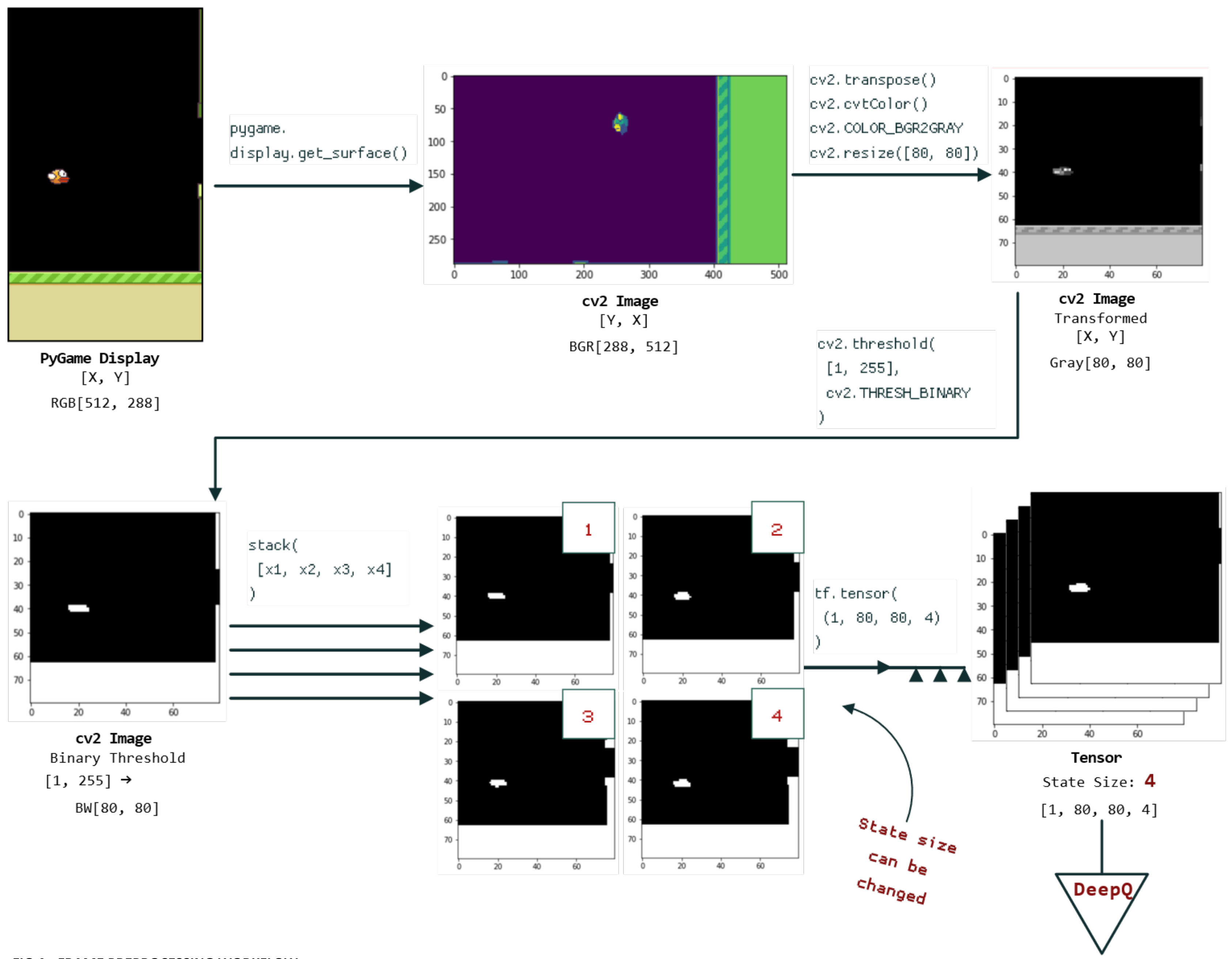
V.III Improvement

For this agent and really any machine learning or deep learning model that would be under consideration for application as a solution to a real-world problem, it would be imperative to improve the hyperparameter optimization This would help give clear guidance on which values are the best to use, which neural network architecture is the most effective, how well the reward system is working, which optimizer is performing the best, and ultimately allow very granular control over the settings used during training. With a model like this, where in a perfect world training is left running and takes multiple days to converge , it's important to go into the process confident that you have a model, agent, and settings that are going to perform in line with expectations. If this model were to be pushed forward with GPU processing, I would move it off of a system with an RTX card in lieu of something else until `TensorFlow` is updated; or switch the codebase to `PyTorch`.

Additionally, moving logging into a database instead of `JSON` flat files would make data analysis and model evaluation much more efficient. `JSON` is really easy to work with, but it's not a very light weight storage option. Loading logs from `JSON` flat files into `pandas` `DataFrames` for analysis is extremely time and resource intensive. To make reading the logs efficient, I truncated the information being logged, broke sessions down into batches of 1000 episodes for the fitting process, then had the agent iterate over that process 50 times. This makes smaller files but generates a lot of them, leading to the same issue with really unreasonable data read times; and cutting potentially valuable data from logging isn't a great idea if this solution were to be used in real-world applications.

Finally, the thing I'd most like to improve about this model is getting the agents' customized `backward()` method working similarly to the built in `fit()` method in `keras`; adding things like the option for a user defined validation train/test split, training epochs, and verbose output while training is in progress. This could be supported by a custom `keras` `Generator` object to create randomized *`Flappy Bird`* episode seeds. At the moment, `backward()` works well for what it was built to do; that's run the prebuilt agent fitting process, which actually runs the `model.train_on_batch()` function. Calling the `model.fit()` method in the current form will trigger a memory leak in RAM once training commences, in turn causing an out of memory error within five thousand episodes. Being an engineer means having the resourcefulness to build and fix things outside of your current scope of knowledge. I've researched this problem quite a bit, and don't know the exact solution for it as of yet, but I will continue trying to find it; unfortunately, that will just have to be outside of the confines of this project deadline. There is a lot I would change if I could go back and do this project again, however the amount of new knowledge I've come out of this armed with is undeniable. As a perfectionist, it's painful to have to complete this project in the current state, but sometimes our biggest learning experiences come from the tasks we underestimate.

Appendix A: Preprocessing Transformations Visualization



```
pygame.
display.get_surface()
```

```
cv2.transpose()
cv2.cvtColor()
cv2.COLOR_BGR2GRAY
cv2.resize([80, 80])
```

```
cv2.threshold(
 [1, 255],
 cv2.THRESH_BINARY
)
```

```
stack(
 [x1, x2, x3, x4]
)
```

```
tf.tensor(
 (1, 80, 80, 4)
)
```

FIG 4: *FRAME PREPROCESSING WORKFLOW*

APPENDIX B: PARAMETER DICTIONARY

```
params = {
        'game': {
            'name': 'FlappyBird',           # name of game being played
            'fps': 30,                      # frames per second
            'tick': 4,                      # clock ticks per second
            'target': 40,                   # target score
            'difficulty': 'hard',           # gap size [easy, medium, hard]
        },
        'agent': {
            'name': 'DeepQ',                # name of the agent
            'action_size': 2,               # number of possible actions
            'delta_clip': np.inf,           # constrain reward range
            'session': {
                'max_ep': 1000,             # max episodes to play
                'episode': {
                    'max_time': 5,          # max minutes per episode
                    'keep_gif_score': 4,    # save gif of episode
        }}},
        'model': {
            'type': 'Custom',               # neural net architecture
            'filter_size': 64,              # filters on state
            'optimizer': 'adadelta',        # [adam, adamax, adadelta, rmprop, sgd]
            'regulizer': 0.01,
            'alpha': 0.05,                  # test value
            'gamma': 0.99,                  # reward discount factor
            'momentum': 0.01,               # SGD
            'decay': 0.001,                 # SGD
            'target_update': 100,           # update model targets interval
            'dueling_network': False,       # DQN
            'dueling_type': 'max',          # DQN
            'training': {
                'verbose': 0,               # 0, 1, or 2, output detail
                'interval': 25,             # train every n steps
                'action_repetition': 4,     # frames per action
                'warmup': 1000,             # observation steps
                'max_ep_observe': 40,       # episode random seed (<=40)
                'learn_rate': 0.001,        # optimizer learn rate
                'initial_epsilon': .1,      # probability of random action
                'terminal_epsilon': 0.001,
                'anneal': 20000,            # steps to cool down epsilon
                'epochs': 1,                # training iterations on batch
                'split': .1,                # train/test split
                'validate': True,           # split and validate batch
                'shuffle': False,           # shuffle batch
                'training': False,          # force training
            },
            'save': {
                'save_n': 1000,             # save model steps interval
                'log_n': 1,                 # log model steps interval
                'ftype': '.json',           # model log type
                'save_full': True,          # full model
                'save_weights': True,       # model weights
                'save_plot': True,          # neural net diagram
        }},
        'memory': {
            'state_size': 8,                # observations to stack for state
            'batch_size': 64,               # states in a training batch
            'limit': 50000,                 # max entries in memory
            'interval': 4,                  # obs to memory every n steps
}}
```

Appendix C: Model Neural Network Layers

```
def _create_model(self, S, A, H, lr, alpha=0.05, reg=0.01):
        inputs = Input(self.shape)
        x = Conv2D(filters=H,
                   kernel_size=(8, 8),
                   strides=(2, 2),
                   use_bias=True,
                   bias_initializer=Constant(value=H),
                   padding='same',
                   kernel_regularizer=l2(reg)
                   )(inputs)
        x = MaxPool2D(pool_size=(2, 2),
                      strides=(2, 2))(x)
        x = BatchNormalization()(x)
        x = LeakyReLU(alpha=alpha)(x)
        x = Conv2D(filters=H * 2,
                   kernel_size=(4, 4),
                   strides=(2, 2),
                   use_bias=True,
                   bias_initializer=Constant(value=H * 2),
                   padding='same',
                   kernel_regularizer=l2(reg)
                   )(x)
        x = BatchNormalization()(x)
        x = LeakyReLU(alpha=alpha)(x)
        x = Conv2D(filters=H,
                   kernel_size=(3, 3),
                   strides=(1, 1),
                   use_bias=True,
                   bias_initializer=Constant(value=H),
                   padding='same',
                   kernel_regularizer=l2(reg)
                   )(x)
        x = BatchNormalization()(x)
        x = LeakyReLU(alpha=alpha)(x)
        x = Flatten()(x)
        x = Dense(256, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(256, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(64, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(64, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(16, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(16, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(4, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(4, activation='relu', kernel_regularizer=l2(reg))(x)
        x = Dense(A, activation='linear', kernel_regularizer=l2(reg))(x)

        return inputs, x
```

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

DeepMind. (2015, February 26). *DQN Nature Paper.* Retrieved from Google: https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

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

Mandal, A. (2018, 10 17). *How to Install Tensorflow GPU with CUDA 10 for Python on Windows.* Retrieved from Python Tutorials: https://www.pytorials.com/how-to-install-tensorflow-gpu-with-cuda-10-0-for-python-on-windows

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

Shung, K. P. (2018, March 15). *Accuracy, Precision, Recall, or F1?* Retrieved from Towards Data Science: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

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

Srivatsan, A. (2016, June 20). *Using Deep-Q Networks to Learn Video Game Strategies.* Retrieved from Github: https://github.com/asrivat1/DeepLearningVideoGames

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