Programming Assignment 3

Deep Flappy Bird

April 9, 2021

1 Introduction

In PA 1, you implemented a representation for a simple puzzle game, Drive Ya Nuts, as well as depth-first search with pruning. For PA 2, you created a representation and heuristic algorithms for playing Nannon, a two-player board game with a more complex ruleset. Finally, in PA 3, you will develop a representation and train machine-learning agents to play a full-fledged game that works with visual data: the infamous phone game Flappy Bird.

1.1 Flappy Bird

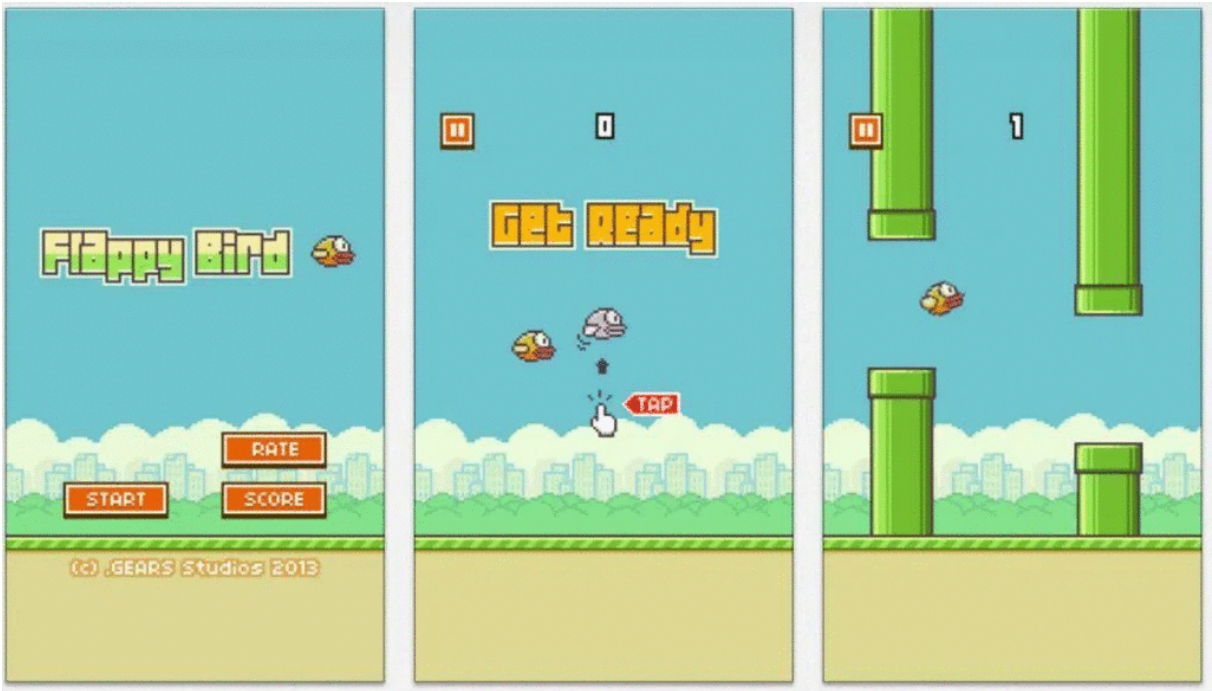

Figure 1: Flappy Bird

Flappy Bird was created by developer Dong Nguyen in two to three days; the gameplay was inspired by the act of bouncing a ping pong ball against a paddle for as long as possible. Initially the game was significantly easier than it became in the final version, however Nguyen said he found this version to be boring. Journalists would later complain about the difficulty while admitting to falling for its addictive nature.

In the world of Flappy Bird, the player controls a pixelated bird named Faby. While flying consistently to the right, the player must navigate through the gaps between randomly generated pairs of pipes. The player receives a point for each successful pass through a pair of pipes, while the game ends if the bird collides with a pipe or the ground.

Your task will be to implement a simplified version of Flappy Bird, and then train an agent to play the game using reinforcement learning techniques.

2 Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning in which intelligent agents must take actions in an environment in order to maximize some reward. With the successful introduction of deep learning techniques, RL has become a focus of attention in the past decade. However, the basic theory and mathematical techniques were established in the 1950s.

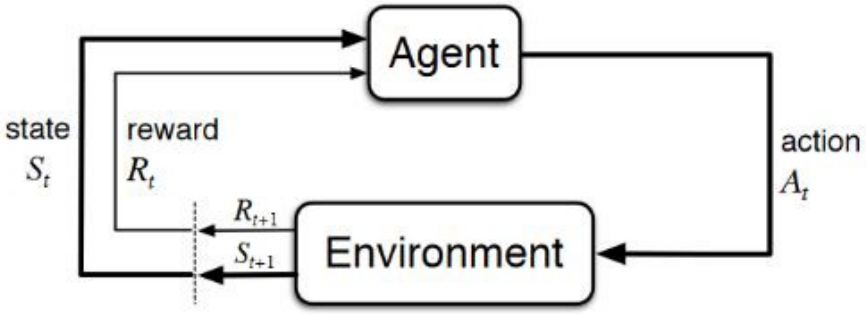

Figure 2: The Markov Decision Process

The central abstraction of RL is the Markov Decision Process (MDP). The components of an MDP include:

- Agent: The learner or decision-maker to be trained
- Environment: Everything outside the agent
- Time: A sequence of discrete time steps, $t \in \{0, 1, 2, \ldots T\}$
- State: A "situation" that the agent may find itself in at a certain timestep. Each state is a member of the larger set of all possible states in the MDP: $S_t \in \mathcal{S}$
- Action: An action that the agent chooses to perform at a given timestep. Likewise, each action is a member of the set of all possible actions: $A_t \in \mathcal{A}$

After each action is performed, the environment returns:

- A reward: a numerical value, $R_{t+1} \in \mathcal{R} \subseteq \mathbb{R}$
- A new state, $S_{t+1} \in \mathcal{S}$

Most RL algorithms work assuming that the problem can be represented as an MDP. A popular toolkit for developing RL algorithms is OpenAI Gym. Gym has been used to represent many

classic Atari games as well as simulated robotics problems. You will be implementing your own environment, similar to those included in OpenAI Gym, for the game Flappy Bird.

2.2 Deep Q Networks

Reinforcement learning had its first notable success in the 1990s with the TD-Gammon, which used an algorithm called TD-Lambda which trained solely through self-play and was able to play backgammon at an expert level. However, progress in the field was relatively slow until 2013, when it was found that the introduction of deep learning techniques dramatically improved the stability and applicability of classic RL approaches.

You will use one algorithm in particular, Deep Q Networks (DQN), to tackle the Flappy Bird problem. **You are required** to read the 2013 paper by Minh, et al., at DeepMind, which showed that DQN could perform surprisingly well on Atari games such as Breakout and SeaQuest. This was followed by a whole host of RL algorithms, many of which use Atari games as their testbed. You can also refer to this more casual introduction to DQNs, and there are many similar blog posts providing intuition for how the algorithm works.

3 Environment Description

You will implement your Flappy Bird environment in the file **`env.py`**. Here we will describe characteristics of the environment before detailing the API you must follow. A Flappy Bird frame is represented as a 50 x 50 grid of numbers taken from $\{0, 1, 2\}$. A 0 value corresponds to empty air, a 1 value to obstacles such as the ground, ceiling, or pipes, while a 2 value corresponds to the body of the bird. This is a simplified representation, but you can imagine that these each could correspond to the pixels you see on a phone screen.

In our "canonical" version of the game, we can moreover specify certain properties:

- The bird is always represented by a 5 x 5 pixel grid.
- The x-position of the bird is always centered.
- There is a border of 2 pixels for the ground and ceiling.
- The starting state consists of the bird centered on the y-position, and with no pipes on the screen. The first set of pipes then spawns immediately on the right.
- Pipes are seven pixels wide.
- There is a gap of 20 pixels between the top pipe and the bottom pipe.
- When centered evenly, each pipe is 15 pixels long. (We include the "ground" in our pipe length)
- Each pair of pipes has some random noise contributing to their vertical length. The top pipe may be between 5 pixels and 25 pixels long; the bottom pipe will grow or shrink to maintain the gap of 20 pixels.
- There is a horizontal gap of 22 pixels between each set of pipes.

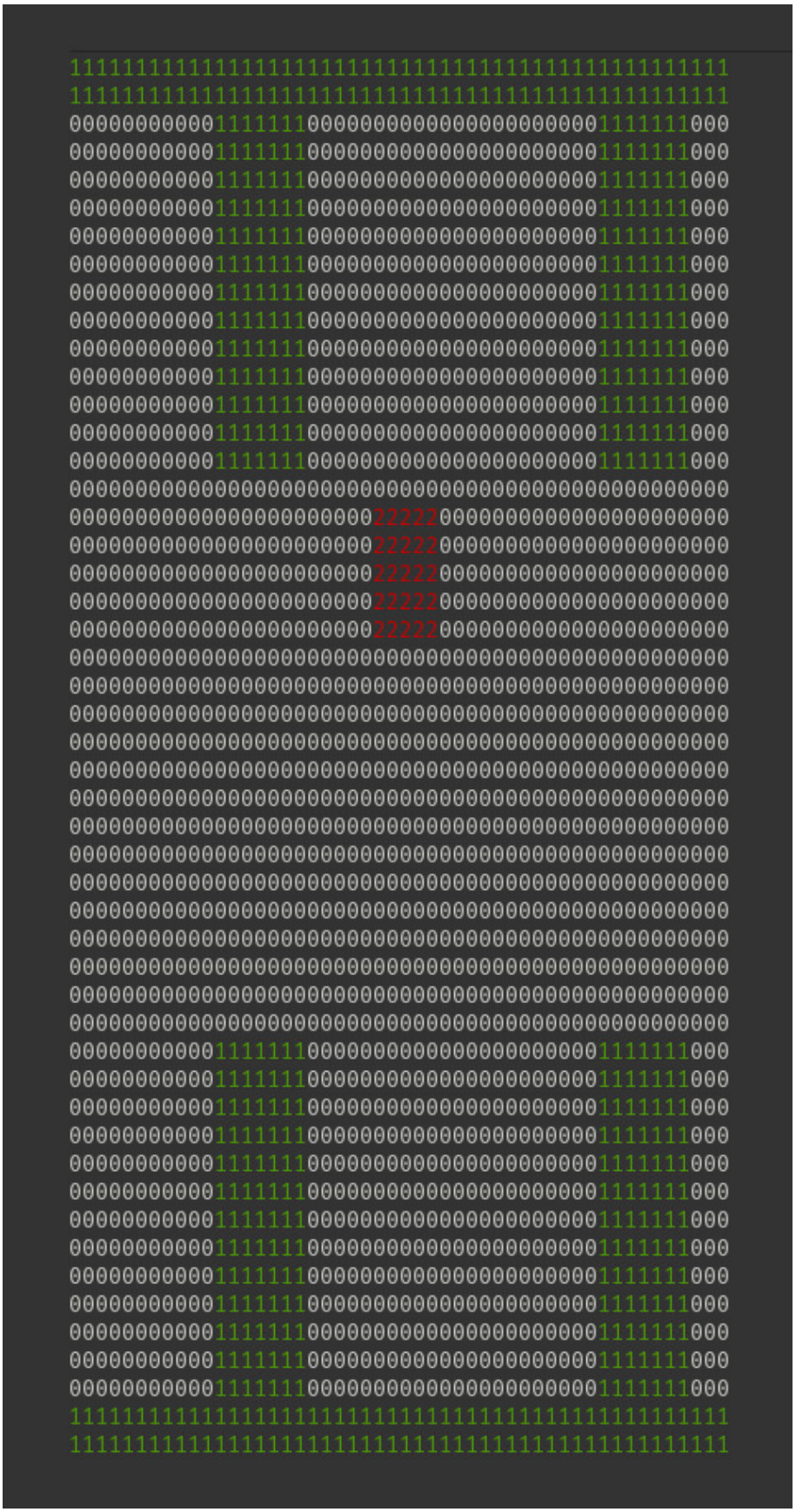

Figure 3: A Flappy Bird frame visually displayed

3.1 Transitions

Our environment must be capable of modeling transitions between states. Similar to the original Flappy Bird, for each state there are two possible actions in $\mathcal{A} = \{0, 1\}$: A 0 action, corresponding to not doing anything, and a 1 action, corresponding to tapping the screen.

Flappy Bird has a very simple physics engine. The bird should appear to continuously fall, but jump when the user taps the screen. This can be accomplished with the following technique:

- The environment should keep track of a variable representing the bird's vertical speed, which is initialized at zero. On each timestep, this should be decremented by 2, such that if the player does nothing, the vertical speed becomes increasingly negative.
- If the player taps the screen, the vertical speed should be reset to a constant value of 5.

(Properly speaking, our 50 x 50 Flappy Bird pixel frame is considered a partial observation, as the vertical speed variable is internal to the environment and hidden from the agent.)

At each timestep, your environment should use the action selected by the agent to internally determine a new state. The bird's y-position should be changed by the vertical speed, while its x-position should remain constant. The position of the pipes should be moved by one pixel to the left. A new pipe should spawn once an old pipe has passed out of frame.

For visual problems such as this one, it has been found that multiple observations are needed in order to infer features such as acceleration and direction. It has been found that four frames are sufficient on Atari-style problems. Thus, your environment should keep track of previous frames. For the first few steps, your observation should include copies of the "start" frame to ensure a 4 x 50 x 50 output.

After each transition is calculated, your environment should be able to return the following:

- A 4 x 50 x 50 pixel observation, as a nested Python list of integers, consisting of the previous two frames, the current frame, and the next frame;
- A reward value that is -1.0 if the agent has crashed into a pipe or obstacle, 1.0 if the agent has cleared a pipe, and 0 otherwise;
- A boolean `done` that is `True` if the agent has crashed and the episode is finished, and is `False` otherwise.

4 API

Environments are simplest to implement as classes in Python and as functions operating with global variables in Lisp. There are two functions (or methods) that your environment should expose to the agent: one being named `reset`:

```
1: function RESET
2:     // This should completely reset your environment and return a new, fresh observation.
3:     // This is like quitting and starting a new game.
4:     // The observation should be a 3-D list of dimension 4 x 50 x 50
5:     return observation
```

And the other being `step`.

```
function STEP(action)
    // The input action is an integer in {0, 1} representing the action of the agent
    // The observation should be a 3-D list of dimension 4 x 50 x 50
    // The reward should be a scalar value that is -1.0, 0, or 1.0.
    // done should be a boolean indicating whether the bird has crashed
    return observation, reward, done
```

The basic skeletons for these functions are included in the starter code.

5 Tasks and Implementation

We have provided you with code and default parameters for the DQN algorithm. Your task will be to develop the Flappy Bird environment, and then to perform experiments using DQN. We also include a tool for visualizing the observations produced by your environment. You will need to modify the following files with original code: **`env.py`** and **`dqn.py`**. Detailed instructions on how to use the codebase to carry out these tasks are found in the Appendix.

- PHYSICS (40%): Implement the basic Flappy Bird environment including the borders, the bird, and the physics engine. Your environment should initialize a state with the bird centered on the screen. Implement borders along with collision detection for when the bird hits the border or flies off the screen. Test that this works as expected with **`test_env.py`** as described in the Appendix.
- PIPES (20%): Implement pipes as described above and verify that collision detection works as expected. Your environment should provide the option for whether or not to use pipes. Also test using **`test_env.py`**.
- DQN FLIGHT (10%): Train your bird to fly using DQN! You will need to implement the **epsilon greedy** strategy in **`dqn.py`** first. If you successfully implement epsilon-greedy, and if your `reset` and `step` functions match the API, all should work properly. Use **`train.py`** and **`show_demo.py`** as described in the Appendix.
- DQN PIPES (10%): Train your bird to fly and avoid pipes! This is a more difficult task, and as such it will require more training time. Follow the same procedure using **`train.py`** and **`show_demo.py`**.
- REPORT (20%): Produce a report describing your implementation of the Flappy Bird environment. Then plot the data produced by your training runs; these will be found in **`records/flight/log.csv`** and **`records/pipes/log.csv`**. Describe the behavior of your best-performing runs, and interpret the data in relation to the DQN paper.
- DQN EXPERIMENT (Up to 10% extra credit): Modify your Flappy Bird environment and devise your own experiment that uses DQN. You may opt to change the characteristics of pipes or introduce new obstacles or rewards. The modifications should demonstrate effort, and the experiment should be interesting. In your report, motivate your reason for the experiment, include relevant data, and provide an analysis of the results.

6 Appendix

6.1 Installation

- To complete this assignment, you will need to create a virtual environment and install a number of Python packages.
- After unzipping pa3.zip, in the terminal, navigate to the root directory of the project.
- Run the following command: `python3 -m venv flappy_env`
- Activate the environment with the following command: `source flappy_env/bin/activate`
- You should see, on the left side of your terminal, the name of the environment in parentheses: `(flappy_env)`
- Now you must install the required packages. This includes the common data manipulation library `pandas`, as well as the deep learning library `tensorflow`. Tensorflow does not require a GPU to run, and it can perform all the operations needed in this assignment using your CPU only.
- Run the following command: `pip install -r requirements.txt`
- It may take some time for the packages to finish downloading.
- Test that both libraries can be imported and used by Python. Start a python shell by running `python`. Then run `import pandas` followed by `import tensorflow`. If you experience no errors, the installation was successful. If there is a problem, do not hesitate to contact a TA! Leave the shell by entering `quit()`.

6.2 Environment

An existing codebase in Python has been made available.

- This assignment uses the Keras API with TensorFlow. You will continue to learn more about neural networks through the semester. Connect the code presented here with the material you learn in class. See a TA if you want to learn more about how it works.
- Tensorflow will work even if your computer does not have a GPU! However, it may do its work a bit slower. This should not prohibit you from completing the assignment, however.
- Implement the Flappy Bird physics engine as described above using the `FlappyEnv` skeleton code found in `env.py`.
- You can see expected behavior by running `python show_demo.py --filename=test.p`
- You can slow down or speed up visualizations using the `--speed` argument. E.g., `python show_demo.py --filename=test.p --speed=0.5`
- Test that your environment produces output as expected by running `python test_env.py`. This should produce an animation in the terminal that shows the bird "flapping" up and down.

- Note: You might get an error that the screen size is too small. Try full-screening your terminal and reducing the terminal font size if needed to ensure that the animation fits in the screen.
- Implement pipes in your environment. Notice that the `FlappyEnv` class is created with two arguments: `use_pipes` and `deterministic`. When `use_pipes` is `True`, the environment should produce and keep track of the pipes. When `deterministic` is `True`, the top and bottom pipes should always be equal length.
- You can see expected behavior by running `python show_demo.py --config=pipes --filename=test.p`
- Run `python test_env.py --use_pipes` in order to visualize your bird flying through deterministically rendered pipes.

6.3 Training

- Trace through the code in `train.py` to gain a basic understanding of how this particular implementation of DQN works.
- Note that the `configs` directory contains configuration files that determine the parameters used by algorithm.
- You will need to implement the **epsilon greedy** strategy starting on line 44 of **`dqn.py`**. More detailed instructions can be found in the comments of that file.
- Although Flappy Bird is a proper infinite-horizon game, we cap the number of steps per episode at 250 for **flight** and 1,000 for **pipes**.
- The `records` directory is structured to store training logs, backups of the neural network weights, as well as result trajectories. Within `records` are the subdirectories `flight`, `pipes`, and `custom`. Each of these will store data associated with each type of run, as well your own custom extra credit if you so choose.
- To run DQN using your environment with `use_pipes` set to `False`, run `python train.py`. This should take less than 30 minutes. You can monitor the print output in the console, then inspect the log produced in `records/flight/log.csv`. Use these CSV files for producing your graphs.
- The training code automatically saves the best-performing trajectory as `best.p` in `records/episode_states` for each config. Upon completing training, visualize your DQN bird's flapping performance by running `python show_demo.py`
- To run DQN using your `FlappyEnv` with `use_pipes` set to `True`, run `python train.py --config=pipes`. This run will take much longer, roughly 4-8 hours depending on your machine.
- Likewise inspect the log in `records/pipes/log.csv`. Do not expect perfection! The original DQN paper used a training time of roughly 50 hours; we've changed the parameters a bit to get more interesting results faster.
- Check a visualization of the best trajectory seen with `python show_demo.py --config=pipes`. You can also run this in a separate terminal while the network is training to see the best run so far in real time.

- You can create additional visualizations using the best network weights by running, for example, `python create_demo.py --config=pipes --output_file=new_demo.p`, and watch them later by running, for example, `python show_demo.py --config=pipes --filename=new_demo.p`.
- If you choose to implement a custom project, then use the `configs/custom` file and alter the existing codebase to accommodate your work.
- Have fun!

7 Submission

- Start early!
- Submit the entire codebase, as well as your report, as a `.zip` file
- Important: Do **NOT** include virtual environment in your submission! That would include the entire Tensorflow codebase in your submission, which is too large for Latte and not needed for us.
- The assignment is due **May 4, 11:55 pm**.