EXAMINING REINFORCEMENT LEARNING METHODS FOR LEARNING FLAPPY BIRD

November 3, 2017

James Robb jamesr15@ru.is
Sigurður Helgason sigurdurhel15@ru.is
Arnar Páll Jóhannsson arnarpj15@ru.is
Reykjavik University
Introduction To Machine Learning T-504-ITML

Contents

1 Introduction

For this project we are given a version of the game Flappy Bird[1] which has been ported to the PyGame Learning Environment (PLE)[2]. Our goal is to examine the results of training multiple agents to play this game with different reinforcement learning algorithms. We have two measure for evaluating the quality of an algorithm and the model it is used with. We measure how far a trained agent can progress through the environment, and how quickly that agent can be trained.

The algorithms we examine for this task are On-Policy Monte Carlo Control and Q-Learning. We begin by using the same initial model for both algorithms, then attempt to use linear function approximation with both algorithms with a provided function, and finally we implement a model of our own using Q-Learning.

2 Environment

In the game of Flappy Bird the player controls a bird that moves across the screen at a fixed horizontal velocity while attempting to navigate through a series of pipes protruding from the top and bottom of the screen. The single means of input is the ability to have the bird flap (ascend). If the bird is not flapping it descends. This is an episodic game meaning after finishing up a game we start again fresh.

Some notable specifications of the environment are that the height of the screen is 512 pixels and its width is 288 pixels and the birds velocity ranges from -8 to 10.

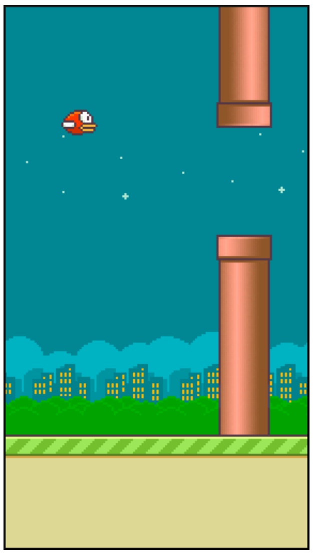

Figure 1: An image of the Flappy Bird environment during game play

The environment has the following characteristics:

- The environment is discrete and finite. The possible velocities of the bird, the bird's position, and the position of the randomly placed pipes are all integer values and could reasonably be stored.

- A state in the environment consists of the bird's position, the velocity of the bird, the positions of the pipes on the screen, and the current score.
- The environment is stochastic in that it is not possible to determine the placement of the pipes until they are seen.
- The environment is episodic in that when the game ends we start with a clean slate at the beginning.
- The environment is Markovian as the distribution of future states is dependent only on the current state, and not on the sequence of actions that brought us to the current state.

3 Learning Methods

3.1 Initial Model and Features

When beginning with the Monte-Carlo and Q-Learning algorithms we are asked to train an agent using four basic features that are provided out of the box by the PLE. They are:

- The current y-position of the bird
- The top y-position of the next gap
- The horizontal distance between the bird and the next pipe
- The current velocity of the bird

While we are only examining four features the possible combinations of these values results in a very large state space. As an initial step to reduce the size of the state space, we discretize these values into buckets of size 15. For example, instead of 288 possible y-positions for the bird, there will only be 15. This reduces the state space to 50625 states.

When implementing the algorithms we store state-action values. For each state described above we store the state along with one of the two possible actions; flap or noop (don't flap). We denote the function that returns the state-action value as $Q(s, a)$.

3.2 Reward Structure

There are two basic ways in which rewards are observed throughout an episode of the game. When the bird successfully navigates through a set of pipes 1 point is given, and -5 points are given when the bird crashes into a set of pipes, the ground, or the sky. Transitioning from one state to another that doesn't result in either a crash (the episode ending) or successfully navigating through a set of pipes yields a reward of 0.

3.3 On-Policy Monte Carlo Control

The first algorithm we examined was On-Policy Monte Carol Control. In particular, we used every visit Monte Carlo. This means we update the values of the state-action values every time we encounter them during the value propagation process. An ϵ-greedy method is used to introduce randomness in the action selection process during training.

When training we set $\epsilon = 0.1$ and the learning rate $\alpha = 0.1$. We trained for one million frames with the initial features described earlier. A plot of the average scored obtained over the training period can be seen below.

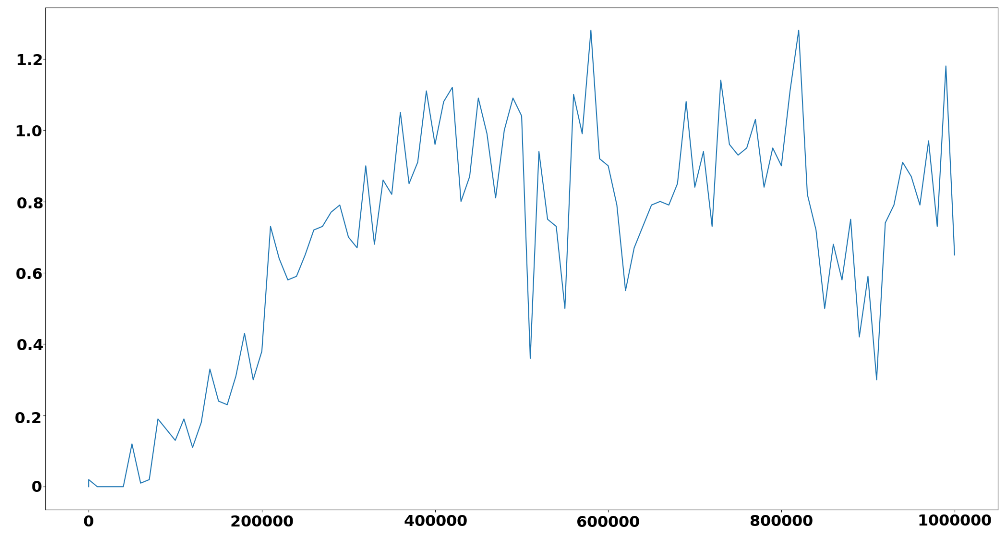

Figure 2: On-Policy Monte Carlo Control learning curve

As can be seen in the graph On-Policy Monte Carlo does not perform well given the four features described earlier. A number of reasons can be attributed to why this is so. The large size of the state space makes it difficult to learn good state-action values in a reasonable amount of time. Monte-Carlo also introduces a lot of variance as the values of states early in an episode depend heavily on the values of states later in the episode. This could be mitigated to some degree with effective reward discounting.

The graph suggests that further training with these features will not yield any significant improvement as the average score visibly levels out.

The agent for this algorithm is `MCAgent` and is located in `MCAgent.py`

3.4 Q-Learning

The second algorithm we examined was Q-Learning. The exact same parameters and features were used as with the On-Policy Monte Carlo algorithm. We trained with one million frames again, and the corresponding graph for the average scored obtained over the training period can be seen below.

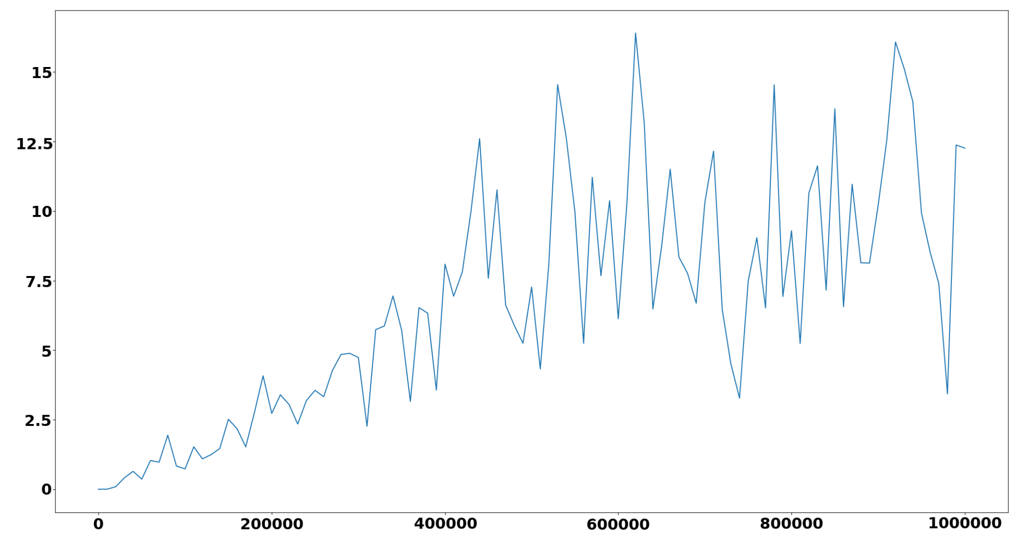

Figure 3: Q-Learning learning curve

The agent for this algorithm is `QLAgent` and is located in `QLAgent.py`

3.5 Monte Carlo vs Q-Learning

We observe that the agent that utilized Q-Learning performed significantly better than the Monte-Carlo agent. We can attribute this to the nature of the game Flappy Bird and the way the algorithms learn. In Flappy bird, what happened earlier in the game is almost entirely independent of what is happening now or what will happen. Monte-Carlo attempts to learn the state-action values by propagating values from events that happened at the end of an episode to the start. Intuitively one can reason that updating state-action values based on what happens through a large series of actions isn't modelling the Flappy Bird problem well. Instead, paying more attention to the values of immediate successor states models the problem more accurately. Q-Learning updates the state-action values in this manner, and as a result performed better.

3.6 Linear Function Approximation

We were asked to implement linear function approximation with both the algorithms discussed so far where the state-value function is defined as

$$Q(s,a) = Q(s,a;\theta_{flap},\theta_{noop}) = \begin{cases} \theta_{flap}^{\top}\phi(s) & \text{if } a = \text{flap} \\ \theta_{noop}^{\top}\phi(s) & \text{if } a = \text{noop} \end{cases}$$

θ_{flap} and θ_{noop} are vectors of four real numbers and act as the weights for the four features denoted by $\phi(s)$.

While if such an approximation was possible, it would definitely reduce the amount of space needed to store the state-action values, and hopefully train faster than the other algorithms. Unfortunately it is not possible to learn such a function as the value function of a state is not linear in the four features we have. The state-action function we want to approximate more closely resembles a step function. An informal example of what this function might look like is displayed below. The result of trying to learn this function was that the weight vectors very quickly became numbers too large for a computer to store using native data types, and an agent that always flapped or never flapped.

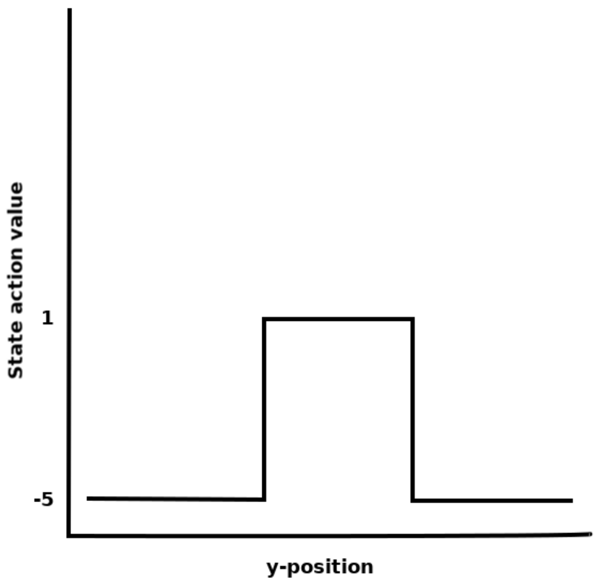

Figure 4: Informal depiction of state-action value function

We could potentially approximate a function for the state-action value for this problem by introducing some additional non-linear features and then learning a linear function in a higher-dimensional space. We made some attempts at this, but the functions learned did not compare with the optimal model we eventually built.

The agent for this algorithm is `LinearApproximationAttempt` and is located in `LinearApproximationAttempt.py`

3.7 Best Agent

3.7.1 Results

This agent performed best by far. With a very small training time, the agent was often good enough at the game to never lose in any of our tests. We allowed the agent to run over-night in one instance at an accelerated frame rate and the agent did not lose.

Unfortunately it is very easy to over-train our agent. Even when training for a relatively small number of frames, the agent quickly goes from being able to play the game perfectly to losing very quickly.

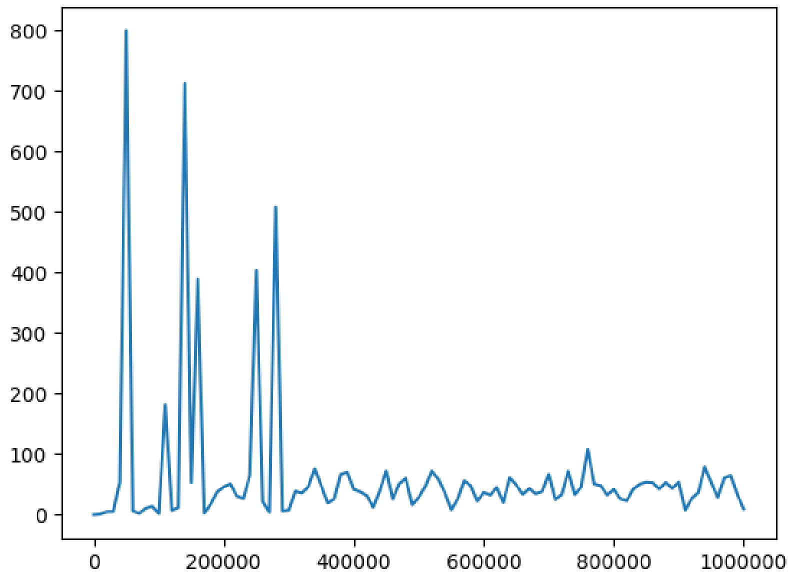

Figure 5: Best agent learning curve

In the graph the highest value on the y-axis (the score) is limited to 800. We had the game artificially end when the agent reached a score of 800 as often the agent performed perfectly and such it would not be possible to extract a result.

3.7.2 Feature Identification and Discretization

We designed a new set of features when creating our best agent. They are:

- The opening between two pipes as a rectangle stretching towards the bird on the x-axis. We call this space the hallway.
- The distance from the bird to the next set of pipes.

- The vertical velocity of the bird.

We employed some discretization on these values. The discretizations are:

- We discretized the vertical velocity of the bird to 8 chunks
- We discretized the distance to the next pipe to be one of the values {close, far}
- We discretized the vertical height of the bird to be one of the values {below, in, above} describing it's position in relation to the hallway
- If distance to the next pipes is close and the bird is in the hallway we discretized the vertical height of the bird further, into 10 chunks.

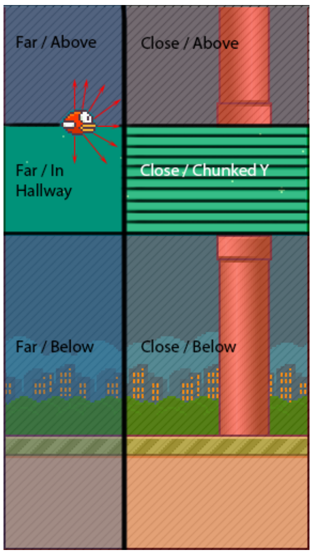

Figure 6: Visualization of the best agent discretization

These discretizations reduced the state-space to ~ 270 states.

3.7.3 Algorithm Description

Our best agent is built on top of the Q-Learning agent examined earlier. The parameters we were asked to tune were ϵ, α, and γ (discounting rate). During the training the we halved the learning rate every 100000 frames. This is done to reduce the fluctuations of Q values over the duration of the training. We let $\epsilon = 0.1$, $\alpha = 0.1$, and $\gamma = 1$, which were the default values for the parameters. While the possibility exists that the agent could have performed even better than it did with different parameter values, we felt the success it demonstrated didn't necessitate exploration of different values.

We proactively initialize the state-action values to favorable values upon their first occurrence. Normally the initial value of a state-action value is 0, but we assign a different initial value based on the following rules:

- If the bird was far and either below or above, we decreased the initial state-action value to -1
- If the bird was close and below and not performing an action (flapping) we decreased the initial state-action value to -2
- If the bird was close and above and flapping we decreased the initial state-action value to -2

The proactive modification discouraged the bird from exploring areas that are never going to lead to a higher goal.

4 Conclusion

4.1 Discussion

A major take away from this assignment was in learning how to properly model and discretize an environment. We attribute a large part of the success we had with our best agent compared to the model outlined in the assignment to how small the state space was, and what each state represented.

When faced with reinforcement learning problems in the future we will endeavour to build the simplest model we feel reasonably describes the environment and state space and slowly add more information or new features as needed.

4.2 Bonus

For both of the bonus problems, we would like the reviewer to evaluate the `QLBestAgent` agent in the file `QLBestAGent.py`.

In order to reach results with the agent that should sufficiently show that we have achieved the goals outlined in the bonus problem, we ask the reviewer to train the agent for only ~ 30000 frames as the agent's performance begins to degrade quickly afterwards.

References

[1] *Flappy Bird.* URL: `https://en.wikipedia.org/wiki/Flappy_Bird`.

[2] *PyGame Learning Environment.* URL: `https://github.com/ntasfi/PyGame-Learning-Environment`.