RL Agent for Flappy Bird
Project Report

Deepak Mittal (CS15S014) and Ghulam Ahmed Ansari (EE12B134)

April 2016

1 Introduction

Reinforcement Learning has been successfully applied to many problems and it has given amazing results. Training AI agents to play games is one such area in which we can leverage the capabilities of Reinforcement Learning to a large extent. Reinforcement Learning provides us tools and algorithms so that we can train agents without knowing the exact model of the environment. There are many model free algorithms available like Q-Learning, Sarsa, Sarsa(λ). Recent breakthrough in this area is its application in Atari Games using Deep Q Networks [4].

In previous mentioned algorithms, we have to hand engineer the features of states to do function approximation, which is a very difficult task. Proposed DQN architecture use deep Neural Networks to find out the features of state ($\Phi(state)$) and learn a function approximator for (state, action) pair values. Goal of this project is to explore all three mentioned algorithms and DQN networks for this Flappy Bird game. This game is not as straightforward. We implemented Q-Learning, Sarsa and Sarsa(λ) algorithms to train RL-agents for FLappy Bird. Experiments with DQN have not been completed yet.

2 Brief Description of Game

In this game, user is given an agent (a bird) to control. User can take two actions : Make a Jump from current state or Do Nothing. Pipes of random height keep on showing from right side and move toward the agent. Pipes always come in pairs. We refer to them as Upper Pipe and Lower Pipe respectively. There is a small space between each pair of pipes. Goal of the user is to prevent the agent from hitting the pipes and ground, and pass the agent through the space between the pipes to get one point. At the end we want to get as much score as possible.

We use the online available emulator to run our experiments [2][1]. This emulator is build using Pygame library.

2.1 Problem Setup

In the first three experiments we have defined the Actions, Rewards and States as follows:

- **Actions** are 0,1. 0 means do nothing where 1 corresponds to make a jump.
- **Rewards** : Agent will get +1 for each time instance it didn't die and -2000 whenever it dies.
- **States** are represented as tuple of two values representing horizontal distance from nearest pipe and vertical distance from the lower pipe In this game agent's x-coordinate is not changing much. Agent is oscillating on the left of the image. Pipes are moving from right to left. So instead of taking agent's sates as its (x, y) coordinate, it makes more sense to take your state relative to the nearest pipe with which you have collision, as this offers repeatability. We keep seeing similar states as the pipes keep coming along. Where as in the former we would have to imbibe the information of the pipes additionally, making the representation more complex.

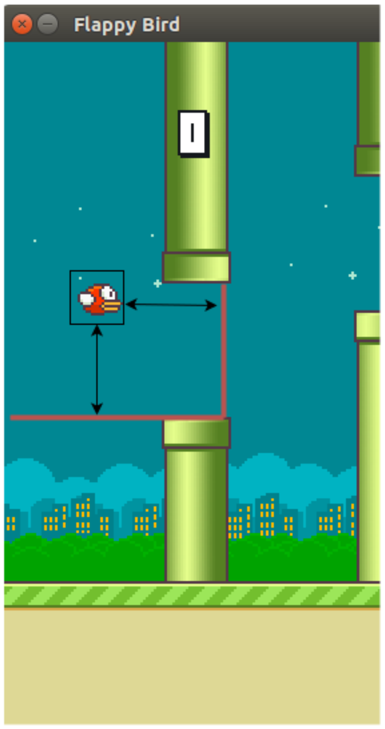

Figure 1: Calculation Of State

- There are roughly 150,000 states if we take each each point as a state directly. We tried training with this set of states and it take a very long time to even reach the first pipe. So there was a need to make the state space smaller. We decided to chose down sampling to reduce the complexity of state space. We **down sample** the space by 10 after trying out different down sampling ratios. The nearby pixels now map to the same state and higher the down sampling ratio, the more is the number of pixels that get mapped to one state. After down sampling the learning is much faster as compared to the last case, but this coarseness comes at the cost of resolution.

3 Experiments

We have performed the following experiments in order to learn to play the game:

3.1 Sarsa and Q-Learning

We have used the state, actions and rewards as defined in section 2.1 . We started out with TD(0) updates i.e we trained using Sarsa and Q-Learning Algorithm. we have chosen the following parameters for training:

1. Epsilon(ϵ) 0.4
2. Step-size(α) 0.6
3. gamma(γ) 1. As the problem is episodic.
4. Best down sampling ratio of 10

While training we observed that the agent after a short while learns a good starting policy to get a score of about 4 or 5 but then after a short while, as soon as it begins to encounter the upper pipe, the agent becomes biased over crashing with the upper pipe. We inferred this to be because the length between the upper and lower pipes is greter than the length between two consecutive pipes laterally and so, the number of times the agent gets a reward $+1$ while climbing upwards before crashing is greater than that of the case when it should have continued with the starting policy.

Both Q-learning and Sarsa, in the scope of our training were failing to tackle the above problem. So we decided to use Sarsa λ. Our main motivation was that to penalize crashing with upper pipe whenever it happens, we also need to penalize those state action pairs through which the agent passes before the crashing state. Thus we wanted to use a method with TD(λ) kind of updates

Apart from all these we also tried some hacks and tricks:

1. We were giving positive rewards only if the agent is alive. So we decided to give large positive rewards when agent successfully passed a pair of pipes. But we did not see any significant improvement

2. Initially Q-learning was not performing very well. The Agent was not even able to cross first pair of pipes. So we tried giving manual inputs so that agent crosses first pipe and then learn by itself thereafter. We did this only for the Q-learning case, and have yet to try it for Sarsa and Sarsa(λ) case. We hope that doing this will speed up learning.

3.2 Sarsa Lambda

We have used the state, actions and rewards as defined in section 2.1 . We have chosen the following parameters for training:

1. Lambda(0.4)
2. Epsilon(ϵ) 0.4
3. Stepsize(α) 0.6
4. gamma(γ) 1. As the problem is episodic.
5. Best down sampling ratio of 10.

We trained for about 4 hours. The agent performed much better than the last time. There is significant improvement, but the problem of section 3.1 did not extinguish completely. We pondered a lot about this and came to a conclusion that, this could happen because our state parametrization is not complex enough, and we should have also considered the velocity of the agent while modelling the state.

3.3 DQN

In this experiment we wanted to train a DQN. The screen size is 512*288 which is to be resized to 84*84 as done in [4]. We intend to use same Network Size to process screenshots as DQN[4]. We are stil doing this set of experiments.
In DQN experiments we want to explore two main points:

1. **State Abstraction**: In Flappy Bird to derive a optimal policy only the distance of agent from pipes and ground matters. Other objects like background objects such as trees, buildings, sky etc does not matter. If we will train a vanilla DQN network i.e. present the raw image to network as current state then its CNN part will unnecessarily spend time in modelling background objects also in state representation. Over the time we can make DQN learn that which parts of state are relevant to perform the required task. We as human beings do this a lot i.e. while performing a task we concentrate on small part of environment and completely ignore most part.

2. **Selective Replay Memory**: DQN samples uniformly from the tuples stored in replay memory. Uniform sampling will give equal importance to all the transitions. However we may not require this. Authors has referenced a way to overcome this, Prioritized Sampling [3].

References

[1] *http://sarvagyavaish.github.io/FlappyBirdRL/g.*

[2] https://github.com/sourabhv/FlapPyBird. *FlappyBirdClone.*

[3] Andrew Moore and Chris Atkeson. *Prioritized sweeping: Reinforcement learning with less data and less real time.* Machine Learning, 13:103–130, 1993.

[4] David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller Volodymyr Mnih, Koray Kavukcuoglu. *Playing Atari with Deep Reinforcement Learning.* https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf.