Reinforcment-Midterm Excercise

Stav Cohen & Avner Duchovni

January 2025

1 Introduction

In the following exercise, we had to implement an autonomous Agent for the game "Flappy Bird" using reinforcement learning with tabular methods

The main idea of the project is the ability to train the bird to pass the pipes through the gaps without manual intervention and without crashing into a pipe

If a bird crashes into the pipe, the game is over

The measurement of success for this task is passing 10 pipes successfully with a gap size of 65

In this project we used the "Flappy Bird" game from Python PLE library, while the library generates the environment, the pipe and the horizontal velocity of the environment (The pipes "move" rate), and the default vertical velocity of the bird (The default is going down by gravity)

Our task is to find a policy for the bird, deciding whether to flap its wings or not, according to the environment on each step in order to maximize the number of pipes passed without crashing

2 Methodology

There are several challenges during implementation of the bird action for each step:

1. Pre-processing:

   First of all, we have to decide what action to take for each step. However, there are a various number of steps and each step may have a different position. Moreover, the default environment is based on Pixels, which can cause a very huge number of possible states

   The size of the environment is $512X288$

For that, we got 8 parameters we can be based on:

(a) The bird vertical position (y position)

(b) The bird velocity

(c) The horizontal distance between the bird and the next pipe

(d) The vertical position of the top of the next pipe

(e) The vertical position of the bottom of the next pipe

(f) The horizontal distance between the bird and the next-next pipe

(g) The vertical position of the top of the next-next pipe

(h) The vertical position of the bottom of the next-next pipe

The main issue is that each one of these 8 parameters can have up to 500 values, which can lead to up to $512^2 * 288^5 * 20$ possible states, which is not possible to train in a polynomial time. In order to resolve it, first we shrink it from 8 parameters to 5 parameters:

- First, we calculate the vertical distance between the bird and the middle of the next pipe (top + bottom / 2). This methodology reduces 3 parameters into 1
- The same we do for the next-next pipe

However, it still $512^2 * 288^2 * 20$ which is a lot. The main thing we do is to transfer the 512 possible states into a binary state as following:

(a) The horizontal distance from the next pipe is turned into 1 if it's less than 45, and 0 otherwise

(b) The vertical distance from the middle of the next pipe is turned into 1 if it's more than -4 (Which is above, not under) and to 0 otherwise

(c) The velocity is turned to 1 if it's positive (going up) or 0 (going down)

(d) The horizontal distance from the next-next pipe is turned into 1 if it's less than 60, and 0 otherwise

(e) The vertical distance from the middle of the next-next pipe is turned into 1 if it's more than -8 (Which is above, not under) && If it's above the vertical distance from the middle of the next pipe It and to 0 otherwise

Then, we get into possible 2^5 states which is just 32 states, which can be easilly trained

2. Reward Shaping:

   The Reward Shaping is simple. The main idea is to give a small reward (of +1) for each survival step, and a very negative reward (of -50), if we got into a terminal state (Which means that the bird crashed into a pipe)

3. Policy agent implementation

This part includes the policy construction. For this project, we implement the agent using two tabular learning methods: **Q-Learning** and **SARSA**

The policy structure is a dictionary of {stateId: [action0, action1]}, while the state id is a string representing processed values of all the parameters. E.g "0_1_1_0_0" and the actions are weights between 0 and 1

We initiate the first state of "0_0_0_0_0" with zero-weights, and then when we do a new step, if the state of the step is a new state that haven't used yet, we also initiate it with zero-weights

When we chose an action, we chose the action0 (UP) if it's weight is greater than the weight of action1 (DOWN), and otherwise we choose action1 (DOWN)

Each time we also have to update the policy, which is done differently based on the method we use:

- For Q-Learning, the policy update is as following: $Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r_{t+1} + \gamma(max_{a'}Q(s_{t+1}, a)))$

  While α is the learning-rate, γ is the distance factor, $Q(s_t, a_t)$ is the weight of the selected action of the current state, r_{t+1} is the reward of the new state, and $max_{a'}Q(s_{t+1}, a)$ is the max weight among the actions of the new state
- For SARSA, the policy update is as following: $Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r_{t+1} + Q(s_{t+1}, a_{t+1}))$

  While $Q(s_{t+1}, a_{t+1})$ is the selected action of the new state

Reducing the state space dimension is crucial to allow effective learning of policies. Although learning with such a big state space is theoretically possible, it is highly inefficient, taking up impractical amounts of time and resources. The solution for that is reducing the state space dimension in a way that will still allow an agent to learn a good policy.

Our first attempt in reducing the state space was aggressive. We decided to reduce the huge state space into a binary space, no more. We decided on a most basic state space. We extracted the vertical difference between the bird and the middle of the upcoming gap. In practice, we found that checking whether the bird is above or below the next gap mid-point allowed the bird to learn a trivial policy: if the bird is below the middle point - flap its wings to gain height. Otherwise, the bird is above the middle of the next gap, so it should avoid flapping its wings and lose height. This allowed the bird to always move towards the middle of the next gap, resulting with a minimal state space. Surprisingly, this state space allowed the bird to pass a fair amount of pipes when the gap size was 80.

Using a trivial state space allowed us to focus on implementing the actual learning algorithms. However, reducing the gap size to a more challenging size

(65) proved more challenging, driving us to look for more complex state spaces which contain more information of the raw state. Increasing the state space is a tradeoff, allowing better policies to be learned, at the cost of a longer learning process, requiring more delicate consideration of hyperparameters. We examined the bird's weakest link, and allowed repeated 'wrong' behavior (behavior resulting in crashing into pipes) to decide how to enrich the state space and preprocessing function.

Careful examination of the bird's behavior using the trivial state space revealed a weak spot - whenever the next-next gap was much higher than the current gap, the bird was unable to gain height in time and crashed below the next-next gap. Staying in the middle of the current gap did not allow us to increase height in time to get to the next pipe within the gap boundaries.

Our solution for this problem was to give the bird a little more information regarding its state. Until this point it was totally ignorant to the position of the next-next gap, and was only aware of its position compared to the current gap's middle. We had to allow the bird to 'prepare' during the current gap towards the next-next pipe. The solution was two fold. First, we increased the information the bird has regarding its vertical position as compared to the next pipe. Instead of a boolean feature, signifying 'above' or 'below' the middle point, we broke the difference to three parts. One part was the upper part of the gap, i.e. above the middle plus a constant c_1, the second part was the middle part of the gap, i.e. below the middle plus c_1 and above the middle minus another constant c_2. The third part was the lower part of the gap, i.e. below the middle minus c_2. This allowed our bird to know for any given point in time whether it's in the upper, middle, or lower part of the gap.

In addition to this change, we added another boolean value. If the middle point of the next gap is m_1 and the middle point of the next-next gap is m_2, we added to our state space a boolean signifying whether $m_1 < m_2$. This allowed the bird to learn two distinct behaviors. Whenever the next-next gap is higher than the current gap, the bird learn to oscillate between the upper bound, and whenever the next-next gap was below the current gap, it learned to oscillate between the lower bound. This allowed it to come adjust its behavior in the current gap accordig to the next-next gap's position.

3 Results

4 Discussion

Creating an autonomous Reinforcement Learning-based agent to play Flappy-Bird consisted of three main challenges - reducing the state space, shaping the reward function, and implementing the learning algorithm. Although these efforts are distinct, they are intertwined, meaning that they must compliment one another, and must be all solved in order to make any progress.

Surprisingly, implementing the actual algorithms was quite straight-forward. The algorithms are well defined, and all that was required was translating the mathematics into code, which although not trivial, didn't prove to be very hard. Additionally, the reward shaping was also pretty simple. Since the goal is directly correlated with how long the bird survives, it was clear that we need to give some positive reward for every step it survived. On the other hand giving a negative reward to losing a game seamed pretty reasonable. After few experiments we were able to balance these rewards, seeing no need to further refine them.

The most challenging and interesting aspect proved to be the preprocessing of the environment. The initial state space is huge, and didn't allow for any effective learning. Tackling this difficulty by dramatically reducing the state space proved useful. We took that to the extreme, reducing the state space to the lowest possible dimention that doesn't result in a trivial policy - a binary state space.

We reduced the state space to a binary state space. Although this might seem extreme, with careful design of the processing function we were actually able to create a space space so simple that it was extremely easy to learn, but still allowed the bird to learn a valid policy, resulting in passing multiple gaps. This extreme approach payed off, since it allowed us to shape the rewards and implement the learning algorithms, and have the agent up and running, achieving decent results with minimal effort and need for hyper-parameter tuning.

Having had this trivial preprocessing function in place allowed us to shift our attention to reward shaping. We've implemented a naive learning agent (inspired by those learned in class), which was able to learn the 'optimal' policy given the extremely-reduced state space. After all, having a binary state space allow you to learn which action is best for each of the two states, overall 4 possible policies, 3 of them resulting in imminent lose, and the optimal one allowing decent performance.

With this over-simplified preprocessing function and a naive learning algorithm we could focus on reward shaping, which after very few tweaking was enough to allow the bird to pass multiple pipes. Having these preprocessing and rewards in place, we could implement the Q-learning and SARSA algorithms and make sure they work as expected, without worrying about other moving parts at the same time.

After having implemented the proper algorithms we were free to return our attention to the preprocessing function. This proved to be a challenging effort, requiring creativity and attention to details. We had to understand what are the policy's weak points, and how can we increase the state space in a minimal

way which will allow the agent to learn more effective policies.

5 Conclusion

The learning algorithms we were asked to implement, namely Q-Learning and SARSA are both beautiful. They are smart and effective algorithms, allowing to solve extremely complicated problems. However, these algorithms were already invented. Implementing them required us to convert a couple of formulas into Python code, a task perhaps challenging, but not one requiring much creativity or complex problem solving.

One may expect that implementing such algorithms would be very hard, but our experienced showed that these are in fact solved problems. On the other hand, there are hidden challenges in using the algorithm's implementation, specifically the reward shaping and the preprocessing. The algorithms are extremely smart - but given a big enough state space using them is simply infeasible. Additionally, they depend on proper reward shaping as well.

We were surprised to find that it is the preprocessing, and not the algorithm implementation that proved to be the most challenging. In fact, reducing a state space of over 10^{18} distinct states to something that's actually learnable proved to be almost an artistic effort. It turned to be a balancing act, that on one end has a too-big of a state space, hindering any learning efforts, and on the other hand over simplification, resulting in a hard limit for how rich and effective learning is.

One of the main difficulties were that the three parts of the solution - preprocessing, rewards, and a learning algorithm have to all work together. Strategically setting up dumbed-down version of learning algorithms and preprocessing functions allowed us to rapidly achieve a baseline on which we could implement gradual improvements. Using a trivial binary state space with a naive learning allowed us to find a proper reward system. With a simple state space and a reward function we were able to focus on implementing the learning algorithms, with the option to make sure they work as expected. Then, with the rewards and learning algorithms ready, we could focus on the most challenging part, the preprocessing.

We found it very effective to enrich the state space gradually, based on close inspection of our agent's behavior, particularly its most common mistakes. Understanding our agent's weaknesses and enriching the state space with information which can allow the agent to find a better policy proved effectively, reaching a best score of over 60 pipes, which we found to be impressive.