RL Assignment : Text Flappy Bird

Louis Chirol, CentraleSupélec, louis.chirol@student-cs.fr

The notebook with more plots can be found on this repository.

1 Agents and sensitivity analysis

The two algorithms I chose are Q-learning (Sutton&Barto chapter 6), and Policy Gradient with the REINFORCE algorithm (Sutton&Barto chapter 13).

I chose Q-learning because it is a classic of Reinforcement Learning in the continuation of what we have done in the course labs so far. In the other hand, Policy Gradient enables to mix a bit of Deep Learning within the learning task, which I found interesting. They are different by nature as the first one learns a state-action values function Q, whereas the second one directly learns a policy through a neural network.

I implemented the Q-learning from scratch, as it is easy to do it, and I followed this tutorial from Huggingface for the REINFORCE implementation, adapting it to our Flappy Bird game. Both agents are considered to win as they can get a score of 1000 or more over several consecutive games.

As a bonus, I also wrote a third Naive Agent, with a predefined policy based on the height of the bird, as a baseline to compare to the two agents. I implemented its policy based on few tries on the game, to fly when it's below the hole, and idle when above, calibrating the threshold to maximize the score. This policy passes in average 30 holes, but inevitably fails when it meets a very low hole after a high one (it just crashes down), which is probably a bug.

Now let's move to the Q-learning agent. The agent is trained with an epsilon-greedy method to update the Q function, and then applies a greedy policy to play the game. I first tried to train it on the (dx,dy) space observation, and noticed that it was learning well, though requiring a non-negligible amount of episodes, about 10,000. I also tried to train some agents cropping the observation to the dy value each time. I did not manage to train a successful agent this way, it dies the same way the Naive agent does, due to the low holes. The horizontal dimension is thus necessary.

Let's finally talk about the Policy Gradient agent. As the dimension of our observation space is only 2, it does not require a big neural network architecture to approximate the optimal policy. I put an input linear layer of dimensions (2,16), followed by a unique linear layer of dimensions (16,2) and a softmax, that worked well. For the training, the agent samples its actions based on the output probabilities of the softmax layer, and then plays the game with a greedy policy as well (taking the argmax of the probabilities). Surprisingly, this agent takes a much lower number of episodes to learn and beat the game (<500). Yet, the NN related computations increase the computation time, making it comparable to the Q-learning agent training time to beat the game. However, it is obviously much faster when using a GPU (as Colab enables).

The Q-learning and Policy gradient agents take a different set of parameters as input. The Q-learning agent requires the specification of a number of training episodes, a learning rate, a reward discount rate and an exploration rate for the ϵ-greedy. The Policy Gradient agent does not need the exploration rate, which makes one parameter less to set. In this part of the report, we examine the sensitivity of the agents to these parameters. As the algorithms are quite different from each other, each agent will be tried with different sets of hyperparameters each time.

First, both agents are very sensible to the learning rate, though they take different values. The convergence of the Q-learning agent can be much longer with too low or high values of learning rate (2500 episodes with 0.3 vs. 10000 with 0.1), and it can not converge at all with the Policy Gradient agent (0.01 converges, 0.001 does not).

The discount rate also plays an important role. The optimal values seem to be the highest (0.99 or 1) for both agents, yet it is unclear for lower values (between 0.3 and 0.7) which performs best over our tries. Too low values (0.1) converge much more slowly or not at all.

Another parameter is the negative reward following a death of the agent. Increasing it can help devaluating dangerous states. Indeed, for both agents, the convergence is as fast as the negative reward is low: the lower the reward, the better for the Q-learning agent. It is less clear with the Policy Gradient agent, as the outcome is more random, but all rewards produce satisfactory results.

Finally, the exploration rate for the Q-learning agent also influences the results a lot. Surprisingly, high values (0.7 or 0.9) can outperform typical values (like 0.3). A too little rate is slower to converge (0.05) but it still does. Indeed, the environment is generated randomly, so even with a high exploration, the agent will be able to pass through all the states possible. The Policy Gradient agent does not need this parameter because it is intrinsically stochastic and exploratory, as we sample an action based on the probability distribution of the output of the last layer during the training.

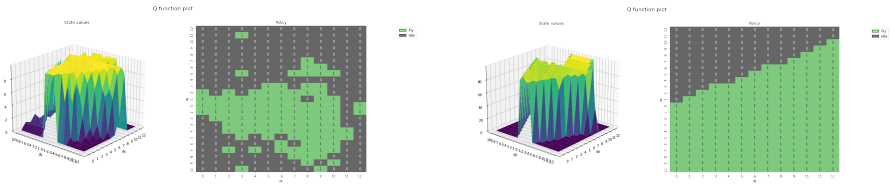
2 Discussion

It should totally be possible to use the original Flappy Bird environment as suggested in the assignment. Indeed, the observations returned by this environment are the same as the one we used, so that no adaptations should be required. However, using the TextFlappyBird-screen-v0 environment would require more work, as the observations of the environment are very different (it is the whole screen encoded). So, the agents cannot be used as is.

An interesting plot is the one of the Q function. From the state-action values dictionary of the agent, we can derive a mapping of one state to its maximum action value, and the policy of the agent for this state with the argmax. This plot is presented in figure 1. For the Q-learning agent, the state values surface illustrates that as the agent is close to the hole (low dx), it values being aligned

with it (dy $\simeq$ 0), but when it is further (high dx, right after a hole), more dy states are valued. This result is logic: we get aligned with the hole as the danger gets closer. The policy makes sense as well as when the bird is above the hole (dy<-2), it will never fly except in some rare states, far from the hole. It is surprising that with dy>1 (below the hole) and dx<8, the agent chooses to idle, but these states have low values and thus won't be considered by the agent.

Such a plot is not immediately available for the Policy Gradient agent, as it has no Q function implemented. We implemented a Monte-Carlo method to estimate this function. It is very interesting to compare this estimation and the policy learnt to what the one of the Q-learning agent, still in figure 1. Indeed, the frontier between the choice to fly and idle is "linear". The difference with the frontier of the Naive agent policy (that would be horizontal) is that the Policy Gradient agent still flies when it's above the next hole (dy< -2) when it's far from the pipe (dx> 5). This prevents the bird from doing the idle action right after a hole and to crash like the Naive agent: the learnt policy is an improved Naive policy. Moreover, the state-action values surface has a shape close to the one of the Q-learning agent. The conclusion here is that at the opposite of the Q-learning agent that learns the values of states and then ignores low values states (thus leaves holes in its policy), the Policy Gradient agent first learns a strong policy for all the states, even those which will have a low value afterward with the Monte Carlo estimation of Q.



(a) Q-learning (b) Policy Gradient

Fig. 1: State-action values and policy

3 Conclusion

We implemented two different agents and their training algorithms taken from Sutton&Barto: a Q-learning agent and a Policy Gradient agent. Both agents are able to beat our Naive agent and to solve the game, with a short training duration. This training can be optimized through the diverse hyperparameters involved in those agents, as the parameters study we led has shown. This approach can be extended to similar environments (the original Flappy Bird), or adapted to other games such as Helicopter or Snake.