RL Assignment : Text Flappy Bird

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

Note : this report is longer than expected in the subject of the assignment, but it was nearly impossible to tackle all the questions with the given template, especially with the images included for the parameter sweeps, that occupy a large part of this document.

1 Agents

The two agents I chose to solve the task are trained following the Q-learning algorithm (see [1] chapter 6), a very classical to estimate the state-action values function Q, and a Policy Gradient agent trained following the REINFORCE algorithm (see [1] chapter 13).

I chose a Q-learning agent because it is a classic of Reinforcement Learning and I thought it was important to get my hands on it, and in the continuation of what we have done in the course labs so far. In the other hand, we have not explored Policy Gradient methods so far, and I found them interesting, especially as they enable to mix a bit of Deep Learning within the learning task. They are very 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 networks that plays the role of a policy function approximator. Their stochastic content is also different, but we discuss it later on.

I implemented the Q-learning from scratch, as it is easy to do it, and I followed a nice tutorial from Huggingface for the REINFORCE implementation, adapting it to our Flappy Bird game, as this type of implementation was quite new for me.

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.
The other agents are able to solve the game, yet they behave quite differently from each other.

Let's start with the Naive agent. 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 scores. This policy passes in average 30 holes, but inevitably fails when it meets a very low hole after a high one. I could not find out why this behavior or fix it without introducing the horizontal information, but it would have made the agent more complicated and not very interesting.

Now let's move to the Q-learning agent. As it's quite flexible, it can adapt to several state spaces. The agent is trained with an epsilon-greedy method to update the Q function, and then applies a greedy policy based on Q to play the game. I first tried to train it on the (dx,dy) space observation, and noticed that it was learning quite fast, though requiring a non-negligible amount of episodes, about 10,000.
I then wondered if a Q-learning agent based only on the vertical information would be able to solve the game. Thus, I retrained 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.

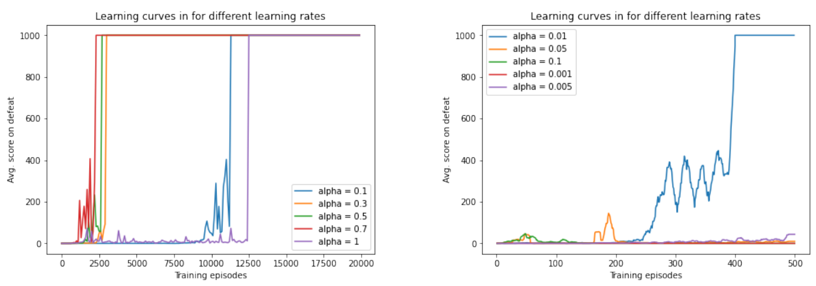
Let's finally talk about the Policy Gradient agent. As the dimension of our observation space is quite low (only 2), it does not require a complicated neural network architecture to approximate the optimal policy. I put an input linear layer of dimensions (2,16), followed by a unique linear layer of dimensions (16,2) and a softmax, that proved to do the job very 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 agents are considered to have converged if they can get a score of 1000 or more over several consecutive games. Thus, their learning curves are cropped to this value, for the execution of the script not to take too long to run (if not forever).

2 Sensitivity Analysis

The two agents of real interest in this report (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 episodes to train on, a learning rate parameter, a reward discount rate and a probability of random action (or exploration rate) for the ϵ-greedy decisions. The Policy Gradient agent does not need the exploration rate, which makes one parameter less than the Q-learning agent 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 agents will be tried with different sets of hyperparameters each time.

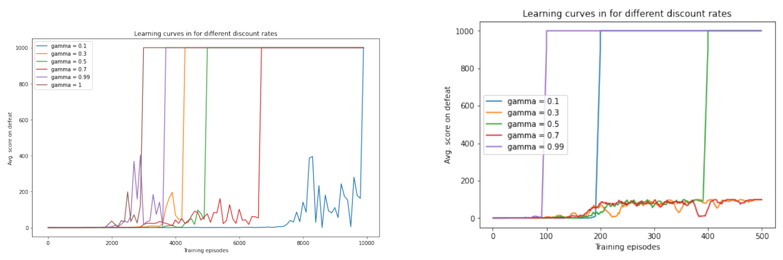
First, both agents are very sensible to the learning rate, though they take different values. As you can see in figure 1, the convergence of the Q-learning agent can be much longer with too low or high values of learning rate (2500 episodes with 0.3 vs. 10000 with 0.1), when it can not converge at all with the Policy Gradient agent (0.01 converges, 0.001 does not).



(a) Q-learning (b) Policy Gradient

Fig. 1: Influence of the learning rate.

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, as you can see in figure 2.

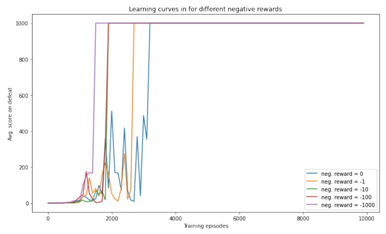


(a) Q-learning (b) Policy Gradient

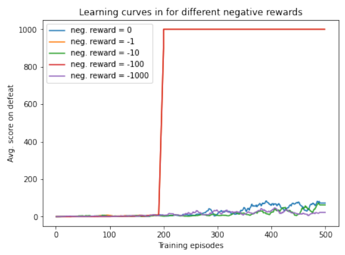
Fig. 2: Influence of the discount rate.

Another parameter not discussed so far 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 (see figure 3.

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 intermediate values (like 0.1), and be on par with typical values (like 0.3). A too little rate is slower to converge (0.05), as in figure 4, but it still does. Indeed, the environment



(a) Q-learning



(b) Policy Gradient

Fig. 3: Influence of the negative reward.

is generated randomly, so even with a high randomness choice of action, 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.

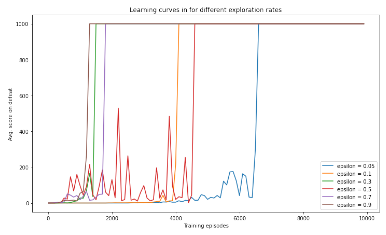


Fig. 4: Influence of the exploration rate over the Q-learning agent.

3 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 except the declaration of the environment. 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 of the Q-learning agent. 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 5. The state values map illustrates that as the agent is close to the hole (low dx values), it values being aligned with it (dy close to 0), but when it is further (high dx values, right after a hole), more dy states are valued. This result is quite 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 choses 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 but it only learnt a policy directly through its neural network. It could be possible to estimate the state-action values function with extra step, but it is quite out of the scope of this work. However, we can take a look at the probability of the actions in each step, as seen in figure 6 with the probabilities of the "fly" action. This table is less sophisticated than the table of the Q-learning agent. Indeed, the frontier between the choice to fly ($P(fly) > 0.5$) and idle ($P(fly) < 0.5$) is "linear". The difference with the frontier of the Naive agent policy (that would be horizontal and centered on -1 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 the previous hole and to crash too fast, as seen with the Naive agent. Thus, we can see the learnt policy as an improved Naive policy.

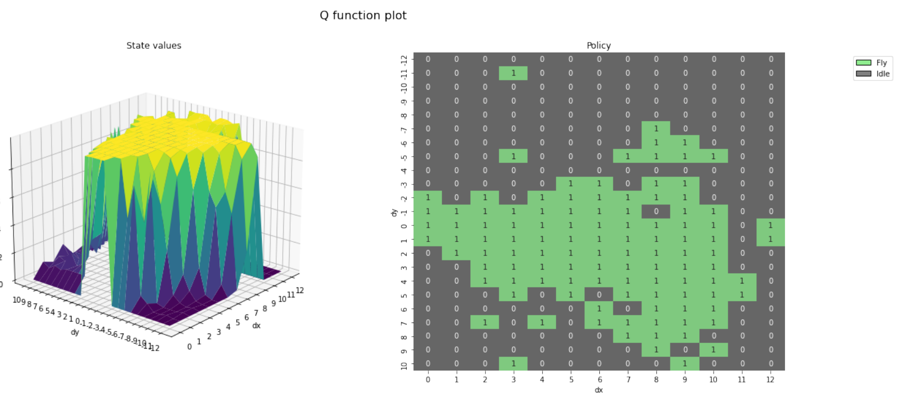


Fig. 5: State-action values representation of the Q-learning agent.

4 Conclusion

In this assignment, we implement two different agents and their training algorithms taken from [1]: a Q-learning agent and a Policy Gradient agent.

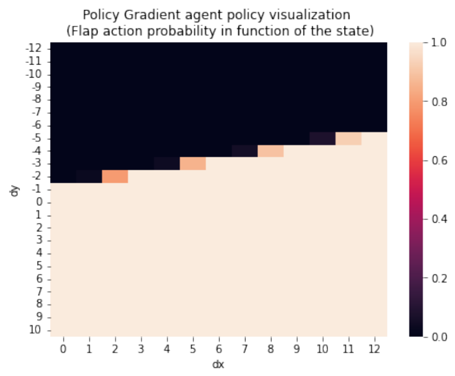


Fig. 6: Policy of the Policy Gradient agent: probability of the ”fly” action for each state of the game.

Both agents are able not only to beat our Naive agent, but 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. The learning rate as well as the exploration rate for the Q-learning agent seem to be the most important parameters, followed by the discount rate.

This approach can be extended to similar environments (the original Flappy Bird), or adapted to other games such as Helicopter or Snake.

References

1. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.