3MD4120: Reinforcement Learning
Individual Assignment

Nardone Arthur

CentraleSupélec
arthur.nardone@student-cs.fr

Keywords: Reinforcement Learning · Q-Learning · Expected SARSA

Jupyter notebook and agents implementation can be found in the following repository: `https://github.com/ArtNd/flappy_bird_RL`

1 Description

In choosing which agents to use, I took into account the nature of the game and the mechanics of the environment. As such, I decided to use Q-learning and Expected Sarsa algorithms, which are both suitable for environments with discrete actions and state spaces.

Q-learning is a model-free algorithm that uses the Q-value function to estimate the optimal policy. It is well suited for tasks that involve exploration and exploitation, where the agent must balance between taking actions that have a high expected reward and exploring new actions to discover potentially better policies.
In contrast, Expected Sarsa is a model-based algorithm that uses the expected reward to update the Q-values. It is more suited for tasks that have a limited number of actions and states, where the agent can afford to explore all possible actions.

After considering the complexity of the game and the nature of the reward system, I decided to use both Q-learning and Expected Sarsa algorithms to compare their performance. By implementing both agents, I can evaluate the strengths and weaknesses of each algorithm and determine which one is better suited for the task. Additionally, I can use this comparison to identify the optimal hyperparameters for each algorithm, such as the step-size and the discount factor, which can significantly affect the performance of the agents.

2 Agents

2.1 Q-Learning Agent

The first agent I implemented is the Q-Learning agent. It is a model free agent that uses the Q-value function to estimate the optimal policy.

I fine-tuned the hyperparameters of the agent by doing a grid search between epsilon, step-size and discount. The best hyperparameters combination was evaluated with the mean reward through 10,000 runs and is the following :

- epsilon (ϵ) = 0.05
- step-size (α) = 0.6
- discount (γ) = 1.0

I also fine-tuned all the parameters sequentially, which gave me the same result, and you can see the plots of this tuning in the appendix.
Let's take a look at the State-Value Graph of the agent fine-tuned for 50,000 runs. See the Policy plot in appendix:

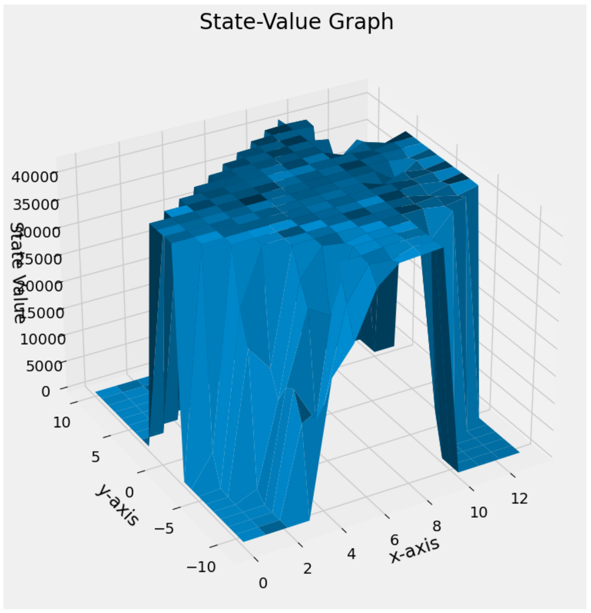

Fig. 1. State-Value Graph of the Q-Learning agent fine-tuned

On the state-value graph we can see that when the agent is very close to the pipe on the x-axis but very far on the y-axis, the state value is null because it is very likely to end. On the contrary, when the agent is in the center of the pipe, the state value is very high.

2.2 Expected Sarsa Agent

The second agent is the Expected Sarsa agent. I fine-tuned the hyperparameters sequentially. The best hyperparameters combination was evaluated with the mean reward through 10,000 runs and is the following :

- epsilon (ϵ) = 0.05

- step-size (α) = 0.5
- discount (γ) = 1.0

Let's take a look at the State-Value Graph of the agent fine-tuned for 50,000 runs. See the Policy plot in appendix:

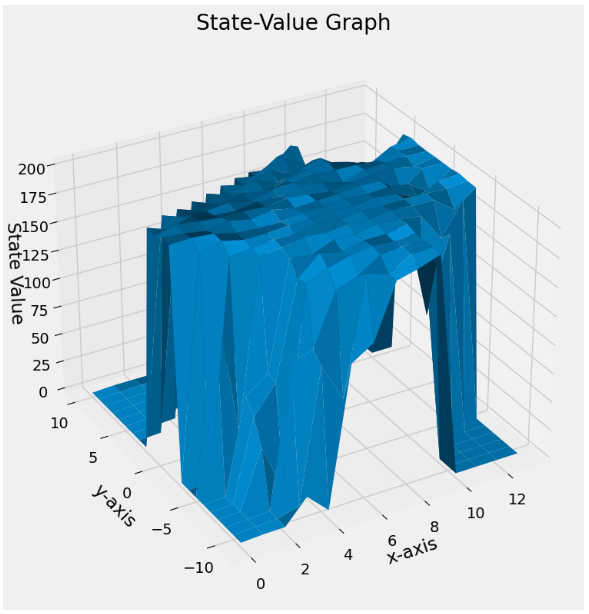

Fig. 2. State-Value Graph of the Expected Sarsa agent fine-tuned

2.3 Agents differences

The two implemented agents, Q-learning and Expected Sarsa, differ in their sensitivity to parameters, convergence time, rewards, and scores. Q-learning tends to converge faster than Expected Sarsa because it updates the Q-values using the maximum expected reward. On the other hand, Expected Sarsa uses the expected value of the reward and, therefore, requires more iterations to converge. We know that Expected Sarsa generally takes fewer risks than Q-Learning, but we can't see it as the Cliff World problem, because it is difficult to represent a "safe" path for the Flappy Bird game. In terms of rewards and scores, both algorithms can achieve similar performance, but Q-learning tends to perform better when the reward structure is sparse.

2.4 Environments differences

The main difference between the two versions of the TextFlappyBird environment is the observation space.

TextFlappyBird-screen-v0 provides a more detailed observation of the game, by returning as an observation the complete screen render of the game. On the other hand, *TextFlappyBird-v0* provides a more limited observation, by returning the distance of the player from the center of the closest upcoming pipe gap. The main limitation of using the same agent for *TextFlappyBird-screen-v0* is the high dimensionality of the observation space, which makes it more difficult to learn an optimal policy.

With an implementation of the original Flappy Bird game environment available, the same agents could be used, but some modifications may be required to adapt to the differences in observation space and action space. The observation space of the original game may require additional processing to extract the relevant features, such as the position of the bird and the pipes. Additionally, the action space may need to be modified to fit the available actions in the game. In the environments proposed by Talendar, it could be possible to use it on the *FlappyBird-v0* environment because the observations returned are numerical information about the game's state. On the contrary, it would be impossible to use it on the other environment because it yields RGB-arrays (images) representing the game's screen. So it would be necessary to use image-based observations and not a simple algorithm like the I used.

3 Appendix

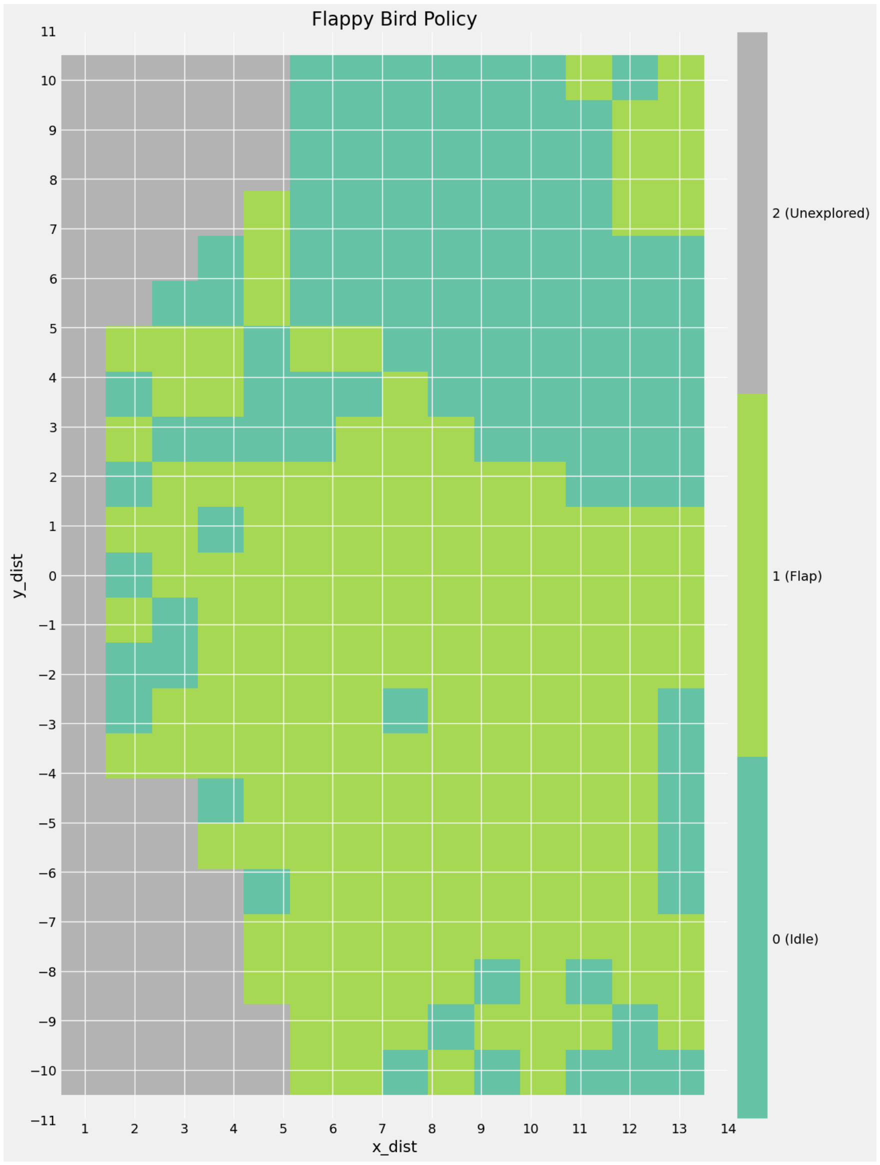

Fig. 3. Q-Learning policy plot

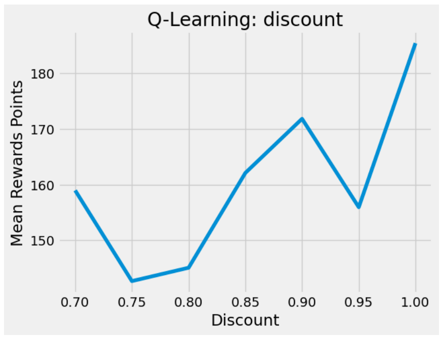

Fig. 4. Q-Learning Discount tuning

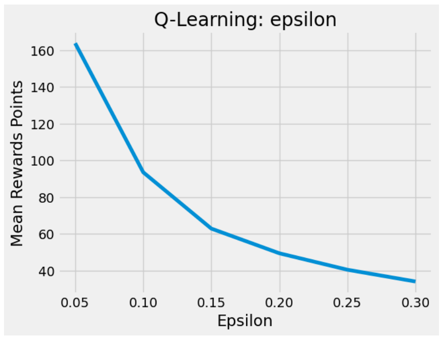

Fig. 5. Q-Learning Discount epsilon

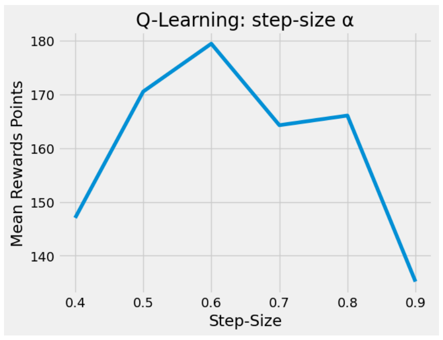

Fig. 6. Q-Learning Step-size tuning

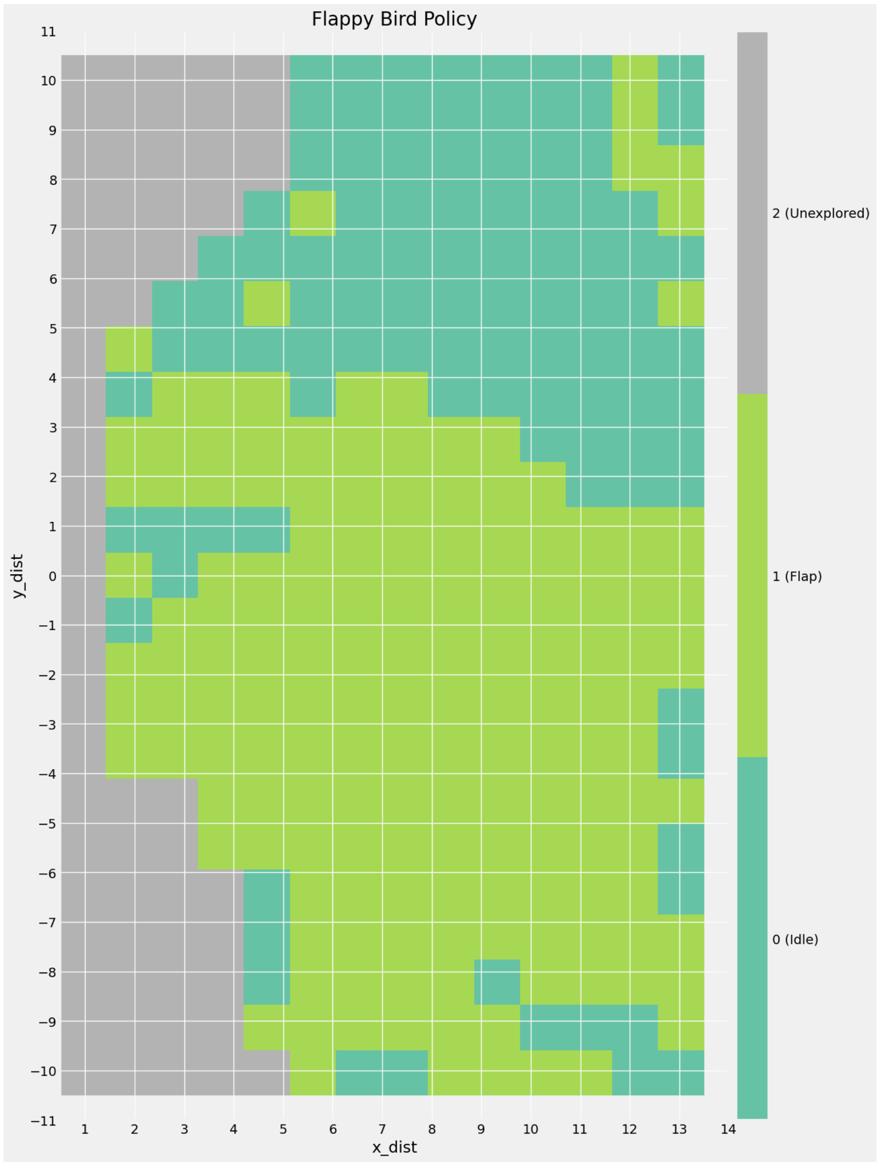

Fig. 7. Expected Sarsa policy plot

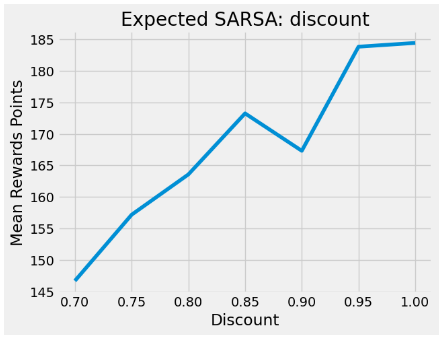

Fig. 8. Expected Sarsa Discount tuning

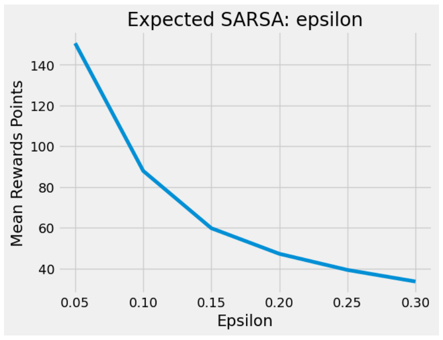

Fig. 9. Expected Sarsa Discount epsilon

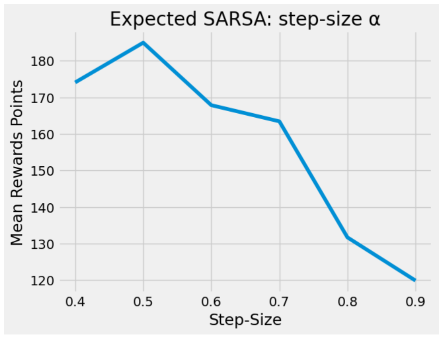

Fig. 10. Expected Sarsa Step-size tuning