# Reinforcement Learning Assignment Report

Raphael Faure (raphael.faure@student-cs.fr)

March 2025

## 1 About the game

Flappy Bird, developed by Dong Nguyen in 2013, is a popular mobile game where players control a bird navigating between green pipes without collision [1]. The bird's automatic descent due to gravity requires players to tap the screen to make it flap and ascend. The game's addictive nature and high difficulty led to millions of downloads and cultural significance, despite being removed from app stores in 2014. Flappy Bird's influence extends to artificial intelligence and machine learning, with a text-based adaptation called TextFlappyBird in the Gymnasium Python package. This version serves as a benchmark environment for training and testing reinforcement learning algorithms.

In the Gymnasium-compatible `text-flappy-bird-gym` package, two versions of the TextFlappyBird environment are available. They share the exact same transition dynamics and differ only in the format of the observations. [2]

The first version, *TextFlappyBird-screen-v0*, returns a 2D array of integers representing the game screen. Each integer corresponds to a character used to render the game in a terminal (like NetHack). This version allows agents to process spatial information but requires more complex state encoding and can increase training time.

The second version, *TextFlappyBird-v0*, returns a simplified state consisting of the horizontal and vertical distance between the player and the center of the closest upcoming pipe gap. This compact representation is more suitable for tabular methods and allows faster training and evaluation.

Overall, *TextFlappyBird-screen-v0* is more expressive and closer to visual environments, but heavier to process. *TextFlappyBird-v0* is minimalistic and efficient, but its simplicity may lead to overfitting or poor generalization in

In this assignment, I have chosen the second version. Indeed, I have enough hardware ressources with Google Colab to train effectively agents on this environment. You can find my work here [3].

## 2 Modelisation

The key features of the game are as follows: the *Agent* is the bird (represented by '@'), controlled by the reinforcement learning algorithm; the *Environment* is defined by the bird's position and the layout of the pipes. The *State Space* is low-dimensional and composed of two discrete variables: the vertical offset $dy \in [-11, 10]$ between the bird and the center of the upcoming pipe gap, and the horizontal distance $dx \in [0, 14]$ between the bird and the pipe. To store values in a Q-table, $dy$ is encoded into the range $[0, 21]$ by mapping negative values with $dy_{\text{index}} = dy + 22$ if $dy < 0$, and $dy_{\text{index}} = dy$ otherwise. The resulting Q-table is thus of shape $[22 \times 15 \times 2]$. The *Action Space* is discrete with two possible actions: flap (ascend) or do nothing (descend). The *Reward Function* assigns a positive reward of $+10$ for successfully passing through a pipe, and a negative reward of $-10$ for collisions, which also terminate the episode.

In terms of implementations, I define a class per agent and an abstract `BaseAgent` class as we did during labs.

## 2.1 Sarsa Agent

The first implemented agent is the SARSA($\lambda$) Agent. It is an approach that incorporates eligibility traces to improve learning efficiency and stability. Indeed, eligibility traces allow SARSA($\lambda$) to update not only the current state-action pair but also past ones, with a decaying influence. This helps the agent assign credit more efficiently over time, speeding up learning in environments with delayed rewards. It updates the value function based on state-action pairs experienced by the agent. The update rule is:

$$e_t(s, a) \leftarrow \gamma\lambda e_{t-1}(s, a) + 1(s_t = s, a_t = a)$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta_t e_t(s_t, a_t)$$

where : $Q(s_t, a_t)$ is the action-value function, $\alpha$ is the learning rate, $\delta_t$ is the temporal difference error, and $e_t(s_t, a_t)$ is the eligibility trace.

The parameter $\lambda$ controls the decay rate of eligibility traces, balancing the influence of past and recent experiences. SARSA($\lambda$) is particularly suitable for scenarios where the policy being evaluated is the same as the policy being improved.

## 2.2 Monte Carlo Agent

The Monte Carlo agent is a reinforcement learning algorithm known for its simplicity. It estimates the value function by averaging returns from complete episodes, updating estimates based on actual observed returns. The update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left(G_t - Q(s_t, a_t)\right)$$

where: $Q(s_t, a_t)$ is the action-value function, $\alpha$ is the step-size, and $G_t$ is the return from time $t$ onwards.

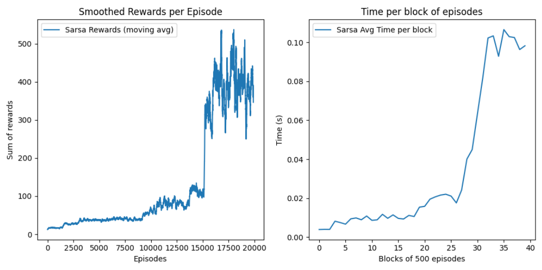
The return $G_t$ is the cumulative reward until the end of the episode, discounted by $\gamma$. The Monte Carlo agent provides an unbiased estimate of the value function without bootstrapping. However, it can be less efficient in environments with long episodes.

## 3 Results over hyperparameters

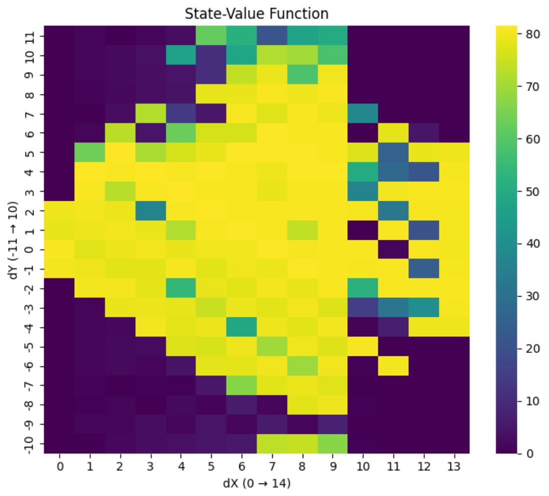
### 3.1 First Hyperparameters

I have been working with the following hyperparameters : $\epsilon = 0.85$, $\epsilon_{min} = 10^{-2}$, $\lambda = 0.8$, $\epsilon_{decay} = 0.999$, $\alpha = 0.05$, discount $= 0.99$, seed $= 1$.

I have trained the Sarsa agent over 20.000 episodes. The figures show that the agent is learning over the steps. The average reward is increasing and the time spent per iteration is longer as the agent learns (Fig. 1). It achieves almost 500 of average reward at the end which is quite impressive without any fine-tuning. The state-value heatmap shows that the SARSA agent learned well. Indeed, when dx is close to 0, only several points are valued, and only 4 at dx = 0, which corresponds to the pipe gap. dy values are reduced as dx tends to 0, as the bird needs to go through the pipe. Just after passing a pipe, the agent is safe, so almost all the positions along the y-axis are valued. On top of that, values are already well approximated with sharp boundaries.



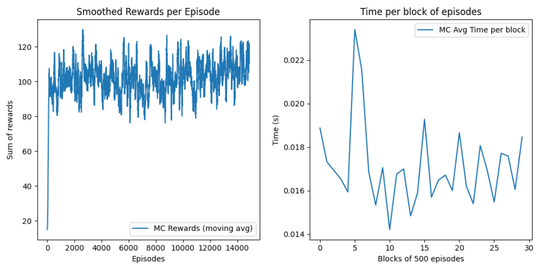
(a) First Sarsa Agent results.



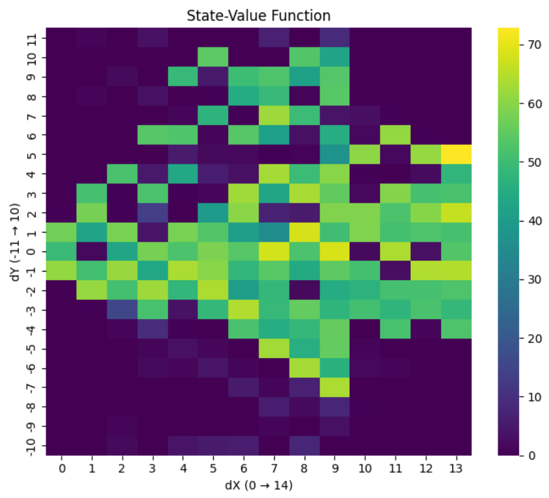
(b) State value

Figure 1: First Sarsa agent

I trained the Monte Carlo agent for 15,000 episodes (Fig. 2). The learning curve remains noisy throughout and shows no real improvement, which suggests unstable learning, likely due to the high variance of the method or poorly tuned hyperparameters. The heatmap has a shape similar to the one from the Sarsa agent, but the values are more spread out, which is consistent with Monte Carlo's tendency to produce higher variance estimates.



(a) First MC Agent results.



(b) State value

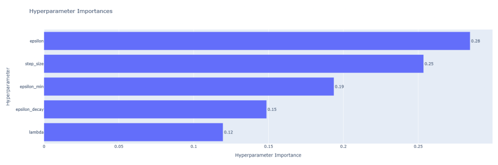
Figure 2: First MC agent

All in all, the Monte Carlo is less performant than the first one in terms of score. The state-value plots show that the Monte-Carlo Agent is more distributed than the Sarsa Agent which is probably due to an higher variance of its $Q$ estimation.

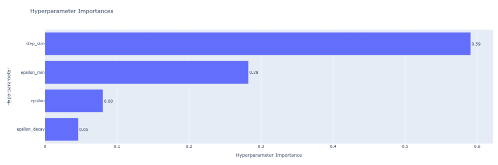
## 3.2 Study of the hyperparameters

In this section, I set the discount to 0.99 and the seed to 1. To perform a fine-tuning of the hyperparameters I used the `Optuna` package. I realised two studies: a first one more exhaustive and a second one more precise on specific hyparameters.

The first one has been performed over 100 trials with 4000 episodes per trial. One can see (Fig. 3) that feature importances are more balanced on the Sarsa Agent's side. On the other hand, the step-size and $\epsilon_{\text{min}}$ are clearly the most important features for the Monte Carlo Agent. As the first one has been performed considering many hyperparameters it let us a better understanding of the role played by each feature. The idea is to perform a second research with only the top-2 of the most important features then train the agents with the best hyperparameters. For each agent, I performed the second study with the best hyperparameter I got from the first one.



(a) Sarsa Agent



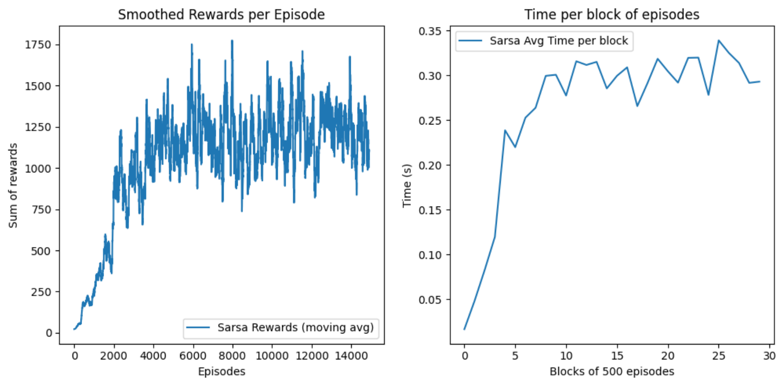
(b) Monte Carlo Agent

Figure 3: Comparison of feature importance for Sarsa and Monte Carlo agents.

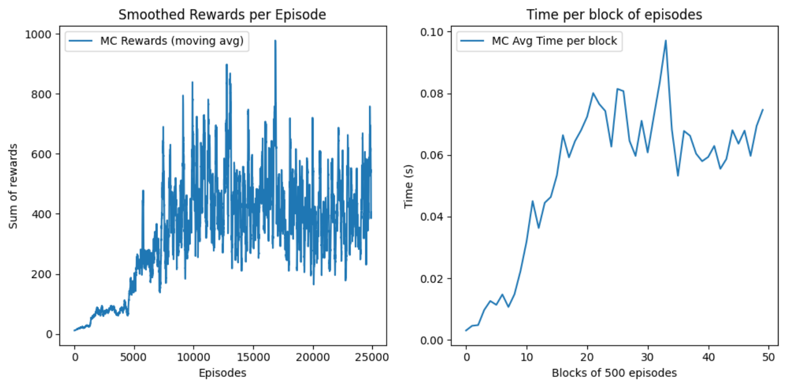
Here are the results from the second study :

| **Agent** | **Epsilon** | **Epsilon Min** | **Epsilon Decay** | **Alpha** | **Lambda** |
|---|---|---|---|---|---|
| Sarsa | 0.351 | 0.005 | 0.990 | 0.415 | 0.800 |
| Monte Carlo | 0.615 | 0.00729 | 0.804 | 0.4733 | X |

Table 1: Hyperparameters for both agents.

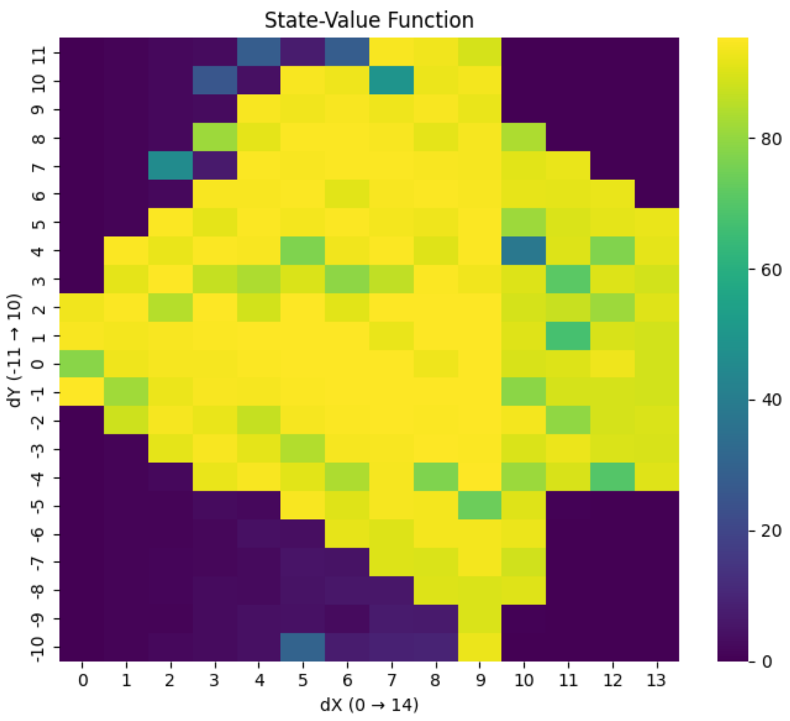


(a) Sarsa Agent

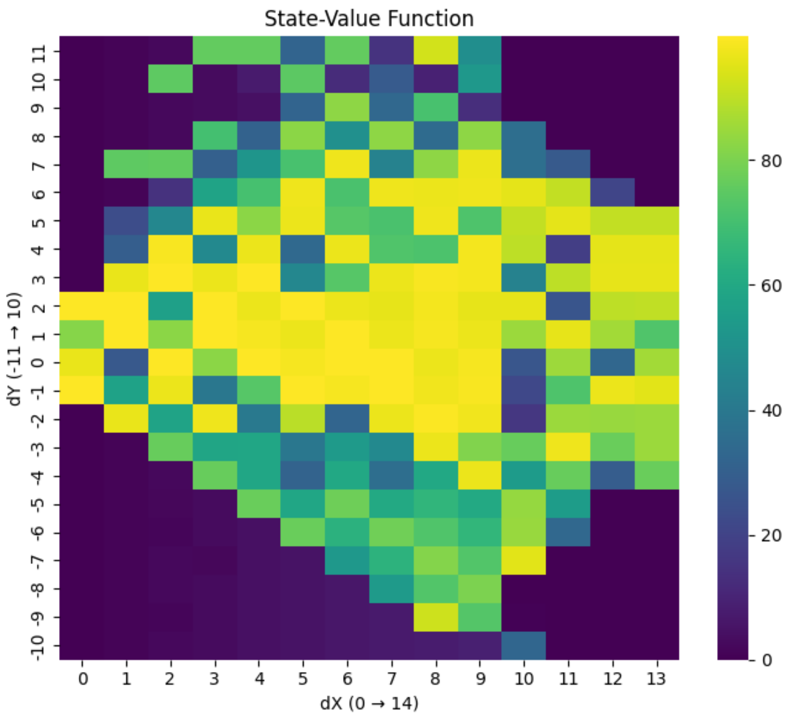


(b) Monte Carlo Agent

Figure 4: Results of the best agents over 25.000 episodes.



(a) Sarsa Agent



(b) Monte Carlo Agent

Figure 5: Results of the best agents over 25.000 episodes.

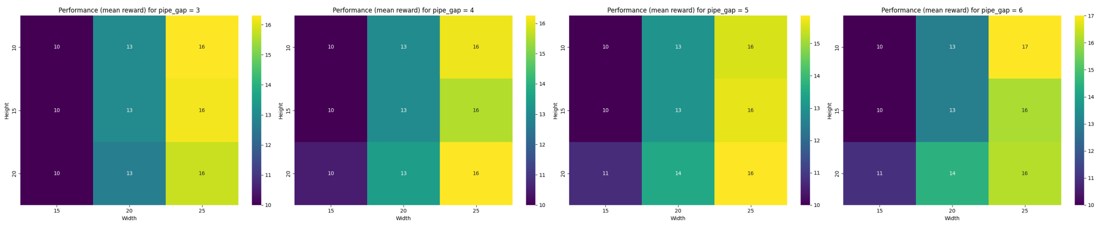
The previous studies have led to a better understanding of each feature and improve the agents performances (Fig. 4). Indeed, the Sarsa Agent achieves in average 1500 average-reward. On the other hand the Monte Carlo agent has improved and achieves 600 average-reward in average but is still very noisy. Both state-value heatmaps highlight similar overall patterns, with high-value areas concentrated around specific (dX, dY) positions, mostly corresponding to short-range, well-aligned approaches to the pipe gap. However, as before, the Sarsa agent's heatmap shows much sharper contrast between high- and low-value zones, with well-defined vertical regions and consistent structure. This suggests more confident and stable estimates. In contrast, the Monte Carlo agent's heatmap is noisier and more diffuse, which reflects the higher variance of its full-episode return estimates. While both agents capture the same global structure, the MC agent is clearly less confident in its value estimates, which makes it less suited for this kind of task.

As there are many hyperparameters, the previous studies might have led to not the best ones for both agents. With more time, they could be extended to all the hyperparameters and augmenting the number trials to 1000 or 10000 should lead to better hyperparameters.

## 4 Extending the work

### 4.1 Robustness to the environment hyperparameters

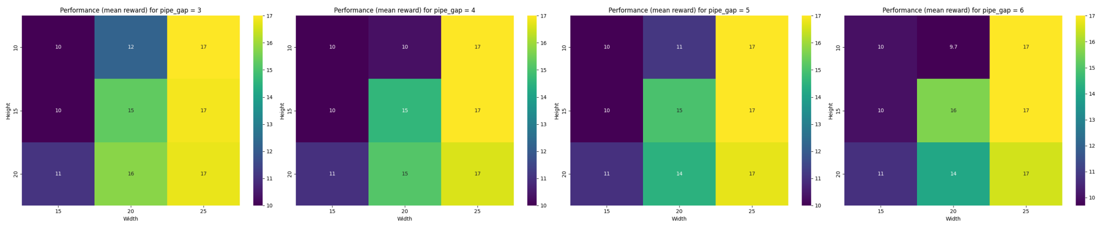
In this section I used the agents that I trained in the last section. I tested the agents on $n_{episodes} = 100$, with $\max_{steps} = 1000$ and I take the average reward. The $Q$ function is frozen (to avoid updating) and the states are clipped with a clip function.



Pipe gap = 3 Pipe gap = 4 Pipe gap = 5 Pipe gap = 6

Figure 6: Mean reward heatmaps for different pipe gap values using the Sarsa agent.

We observe that larger pipe gaps consistently lead to higher mean rewards, suggesting that wider openings make the game easier for the Sarsa agent. On the other hand, the agent performed poorly with a lower width which is logical as the agent had less time to anticipate. Interestingly, the agent is quite robust to height variations.



Pipe gap = 3 Pipe gap = 4 Pipe gap = 5 Pipe gap = 6

Figure 7: Mean reward heatmaps for different pipe gap values using the Monte Carlo agent.

The Monte Carlo agent is less robust to height variations, as it requires a large width value to remain stable. Increasing the gap does not have a significant impact, which is somewhat surprising. With more time, I would have conducted additional tests. Overall, performance is less stable across configurations, with significant variance between grid positions. This suggests a higher sensitivity to environment parameters.

## 4.2 What about the original environment?

Agents trained on `TextFlappyBird-v0` cannot be used as-is on the original Flappy Bird game, since the observation spaces are fundamentally different.

In our setup, the agent receives a compact state made of two values: the horizontal and vertical distances to the center of the next pipe gap. The original game, on the other hand, outputs either raw RGB frames or internal variables, which are not directly compatible.

To reuse the trained policy, one would need to extract from the original game's observations a similar pair of features, either by processing the visual input or accessing internal state variables. Once this mapping is established, the policy can be applied without retraining from scratch.

This kind of preprocessing enables partial transfer and can serve as a starting point for fine-tuning on the real game.

## References

[1] Flappy Bird Wikipedia :https://en.wikipedia.org/wiki/Flappy_Bird

[2] TextFlappyBird documentation :https://gitlab-research.centralesupelec.fr/stergios.christodoulidis/text-flappy-bird-gym

[3] My work : https://colab.research.google.com/drive/1t3KU-t-81Vzfb4Jb0_BM9Roedx2k_gql?usp=sharing