3MD3220: RL Individual Assignment

Antonine Batifol

https://github.com/abatifol/RL-flappy-bird

1 Experimental Setup

In this work, we evaluate two reinforcement learning agents: Monte Carlo (MC) agent and Sarsa(λ) agent on to the Text Flappy Bird (TFB) environment.

1.1 TFB Environments

Two versions of the TFB environment were provided:

- **Text-FlappyBird-simple environment**:This version provides a simplified state consisting of two numerical values: the horizontal and vertical distances between the bird and the center of the upcoming pipe gap. This discretized representation is computationally efficient and allows for faster training but may be limiting if the agent needs more context about the environment. For instance, the environment does not provide information on the upper and lower boundaries of the screen, where the agent fails if it hits it.
- **Text-FlappyBird-Screen environment**: This version returns the complete screen as a 2D array where each element represents a component (e.g., the bird, pipes, and background). It provides a richer, high-dimensional state representation, which offers more detailed context. However, this comes at the cost of increased computational complexity due to the processing of a larger state space. Also it may not scale well with increased screen size, as the state space will grow significantly.

For both versions, the action space is binary: the agent may choose either to *flap* (to ascend) or to remain *idle* (allowing gravity to pull it downward). In the rest of the report we will use the simple environment because of computational limitation, since the parameters hypertuning required dozens of hours.

1.2 Monte-Carlo Agent

The Monte-Carlo Agent is based on episodic learning and Q-values are updated only at the end of each episodes:

$$Q_{t+1}(a, s) = Q_t(a, s) + \alpha(G - Q_t(a, s))$$

where G is the sum of the discounted rewards of the episode. The Q-value for each state-action pair visited during the episode are updated. We repeat this for multiple episodes to refine Q estimation. By using real returns and not estimated

it is not biased. However waiting the end of each episode before updating Q, induced a lots of variance and makes the learning slow. Monte Carlo Agent can only work with episodic environments. In our implementation, the agent selects each next action during the episode using an ϵ-greedy policy. Thus the agent is sensitive to the choice of epsilon (and the decay) which controls the trade-off between exploration and exploitation in the ϵ-greedy policy, of α the stepsize used for each update and γ the discount rate.

1.3 Sarsa(λ) Agent

Sarsa(λ) is an on-policy, step-by-step learning algorithm that combines Temporal Difference (TD) Learning with eligibility traces.

1. Take an action using an ϵ-greedy policy
2. Observe the reward and next state.
3. Choose the next action using the same ϵ-greedy policy
4. Compute the TD error:

$$\delta = R + \gamma Q(S', A') - Q(S, A)$$

5. Update Q-values using eligibility trace: The Q-values are updated at every step using traces that decay over time. The trace allows the agent to learn from both recent and past experiences and is updating during the episode with Q, $e(s, a) = e(s, a) + 1$. Then for all s,a:

$$Q(s, a) = Q(s, a) + \alpha\delta e(s, a)$$

$$e(s, a) = \gamma\lambda e(s, a)$$

6. Repeat for every step until the episode ends.

Unlike the Monte-Carlo Agent, the Sarsa(λ) Agent updates the Q-value continuously during an episode using bootstraping to estimate future rewards. This is supposed to help the agent learning faster. For the Sarsa(λ) Agent, the factor $\gamma\lambda$ controls the influence of past action in the learning process through the eligibility trace.

2 Results and Performance

2.1 Default Parameter Performance

Initially, both agents were trained with the same default parameters: $\epsilon = 1.0$, $\alpha = 0.1$, $\gamma = 0.99$, and ϵ-decay $= 0.999$ (with Sarsa(λ) using $\lambda = 0.5$). Under these settings, both agents seem to perform quite similarly. However, the Monte Carlo (MC) agent's rewards demonstrated more instability, likely due to the higher variance caused by the Q-value updates occurring only at the end of each episode. It does not appear that the Sarsa Lambda agent learns faster than the MC agent as we could have expected, which suggests further parameters tuning.

Policy Analysis Figure 2 in the appendix illustrates the learned policy of the MC and Sarsa agents. As anticipated, when the bird is below the center of the gap, the policy for both agents is in majority to flap. However when the bird is above the center of the gap, it has not yet fully learned to idle in order to fall below. Especially in the case of the MC Agent, the policy is still mixed. Closer to the pipe (x close to 0) the policy still encourages to idle as expected, but gives more freedom farther away from the pipe. However, the MC agent's policy appears more uncertain, likely due to the higher variance in its updates. This suggests that the MC agent might require longer training to reach a more stable and effective policy.

State-Value Function Figure 3 shows the state-value functions for both agents. Overall both agents seem to have the same coverage for the states visited. However the Sarsa(λ) agent may be a bit more structured in his state visitation (less scattered high-value) except on the upper left, indicating clearer trajectories. The Monte Carlo method updates state values only after entire episodes, leading to this slightly noisier value distribution. Sarsa(λ), using temporal-difference learning with eligibility traces, learns more smoothly across states, producing more continuous and structured value functions.Overall, both agent display expected value-state distribution, as the bird has to be close to the horizontal line Y=0 when it comes near the pipe (X=0) while it has more freedom when it is further away from the pipe. The MC agent seems less sure since it display high-value state above the Y=0 line even close to the pipe.

2.2 Hyperparameter Tuning

Monte Carlo Agent Tuning To optimize both training times and final performance, I introduced a stopping criterion: training terminates once the agent achieves a score above 10,000 on at least 2000 episodes. I conducted parameter sweeps for varying α, γ, and ϵ-decay (see Fig 4). Key observations include:

1. **Learning Progression**: Most configurations exhibit significant improvements after 4000 to 6000 episodes, which highlights slow learning of MC agent due to updates only at the end of each episode. There is exceptions combinations that reached very high score early.
2. **Epsilon Decay Impact**: A slower decay (e.g., 0.9995) maintains exploration longer making learning too long, however a too fast decay (0.99) precipitates earlier exploitation and lead to suboptimal policies. ϵ-decay=0.999 seems like a good compromise.
3. **Learning Rate Influence**: Higher α values (e.g., 0.3 or 0.4) provide rapid initial learning but risk instability, whereas lower α (0.1) results in steadier, but slower, convergence. Overall α does not seem to influence the convergence time but eventually $\alpha = 0.2$ seems to yield the highest rewards globally.

Particularly promising configurations include $\alpha = 0.2$, $\gamma = 0.999$ with ϵ-decay 0.99, and $\alpha = 0.3$, $\gamma = 0.99$ with the same decay rate, though the latter

may risk overfitting to a specific strategy. Otherwise $\gamma = 0.9$ associated with ϵ-decay=0.999 yield the best rewards.

Sarsa(λ) Agent Tuning Due to time constraint, the training of the Sarsa agent was halted once it reaches a score of 10000 at least 50 times. A similar parameter sweep was performed for the Sarsa agent as for the Monte Carlo agent (see Figures 5 and 6). The key findings from this tuning process are as follows:

- **Learning Rate (α)**: $\alpha = 0.2$ and $\alpha = 0.3$ yield faster learning and higher rewards at the end. However, when α was set too high (e.g., α=0.4), instability was introduced, as reflected in the erratic behavior of the reward curves in the graphs.
- **Epsilon and Epsilon Decay Impact**: (ϵ) affects exploration-exploitation balance and is initially set to 1.0 for full exploration. The decay rate of epsilon determines how quickly the agent shifts from exploration to exploitation. We can see that a too fast decay (0.99) leads to bad performance as the agent does not explore enough. For all combinations, a decay of 0.999 yield the best results except for the exception $\alpha = 0.3, \gamma = 0.999$ and a decay of 0.99.

However, we cannot infer the impact of γ from the plot since it does not seem to have a high influence when γ is high enough e.g. 0.9 here. I choose a discount high enough for better long-term reckoning. We also explore a large range of values for λ for two different values of α. However the results do not allow to determine which value for λ yield the best performances. λ is supposed to determine how much past experiences influence updates. Higher λ values (closer to 1) promote longer credit assignment, making learning more stable but potentially slower, while lower λ values (closer to 0) behave more like standard SARSA, updating only based on the most recent transition.

3 Real Flappy-Bird Environment

Our agents designed for the Text Flappy Bird environment may not directly work in the original Flappy Bird game without modifications, because the original game environment includes a **continuous** state space, where bird position, velocity, and pipe distances are represented as floating-point values, whereas the text-based version uses a **discretized** state space. Our Monte Carlo and Sarsa(λ) agents are standard tabular RL methods, which will struggle with large or continuous state spaces because they require explicit storage of every state-action value. Therefore it would require to use function approximation techniques such as Neural Networks (Deep Q-Learning) or Tile Coding. We may also adapt

4 Conclusion

Overall the Monte Carlo and the Sarsa(λ) agents exhibit distinct sensitivities to hyperparameters. For both agents, a too high learning rate (α) can lead to

instability and high variance, while lower α allow more stable learning, however too low values will yield slower convergence. A high discount factor (γ) is necessary for both agents to be able to take into account long-term rewards. Both agents depend on an ϵ-greedy which should allow for enough exploration during the early episode. This requires a slow decay of ϵ to maintain sufficient exploration and prevent premature exploitation (sub-optimal policy). Additionally, the Sarsa(λ) agent leverages an eligibility trace parameter (λ) to propagate learning across past actions; higher λ values facilitate broader credit assignment at the potential cost of stability, while lower values limit this propagation, leading to slower overall learning. Both Agent managed to reach 10000 score multiple times (see Fig 7 and Fig 8) for different parameters.

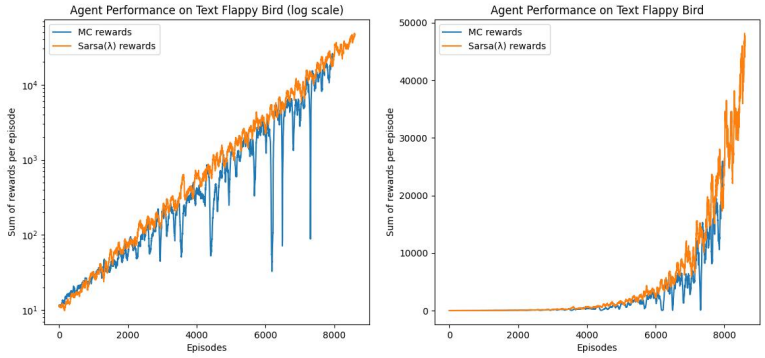


Fig. 1: Training of Monte-Carlo Agent and Sarsa(λ) Agent with default parameters

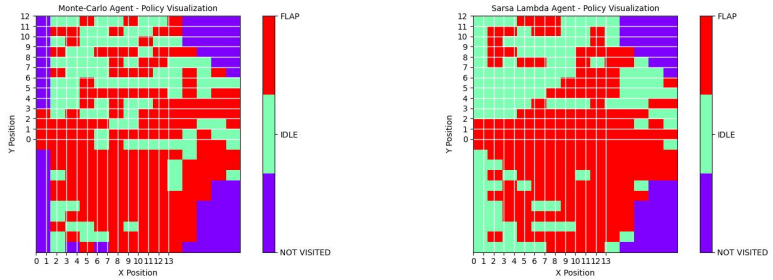


Fig. 2: Policy of the Monte-Carlo Agent and Sarsa(λ) Agent trained with default parameters

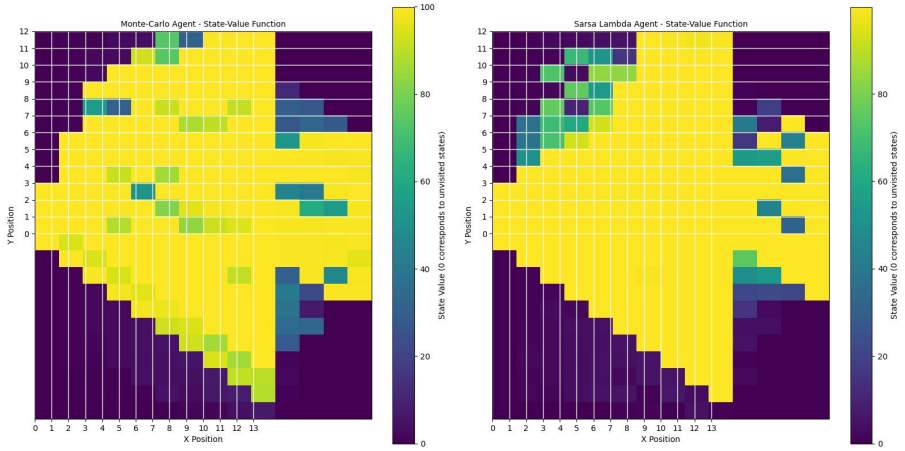


Fig. 3: State-value functions of the Monte-Carlo Agent and Sarsa(λ) Agent trained with default parameters

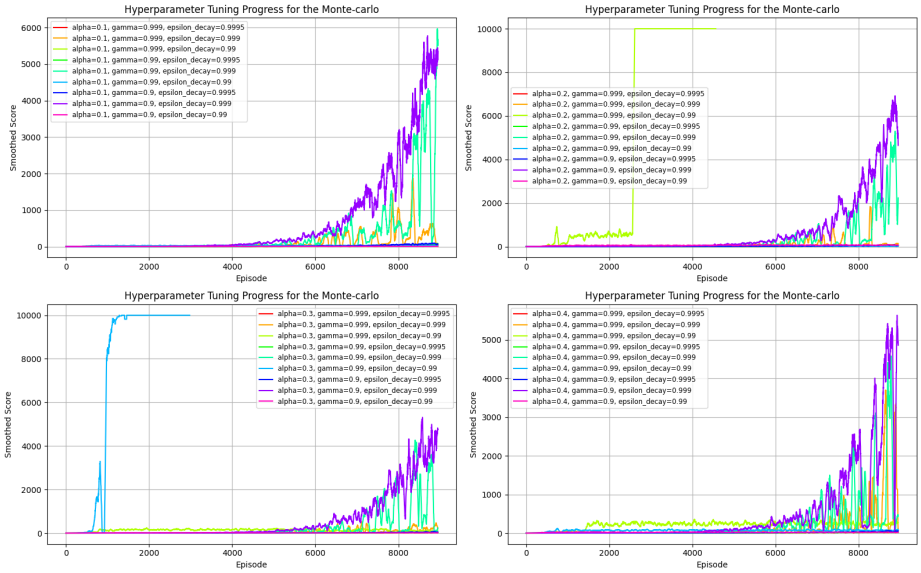


Fig. 4: Hyperparameters tuning for the Monte-Carlo Agent

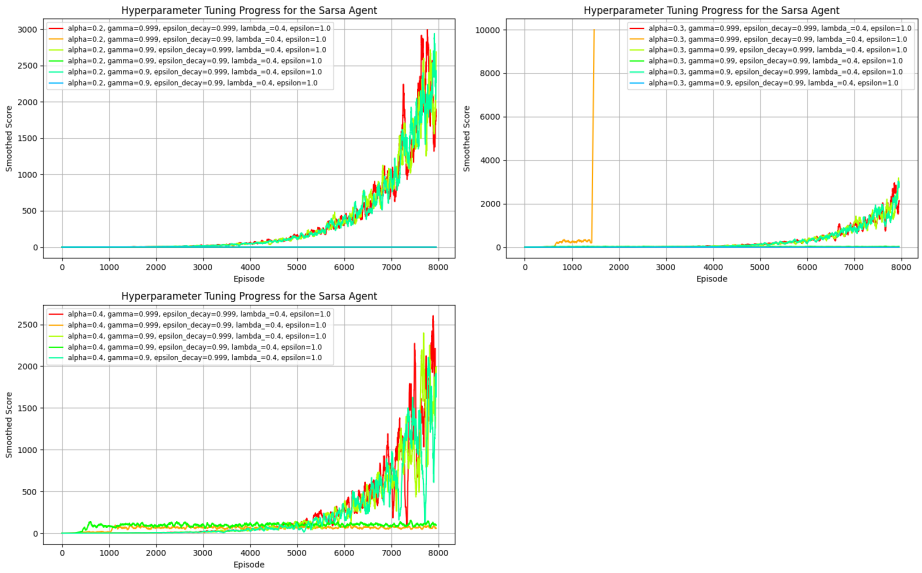
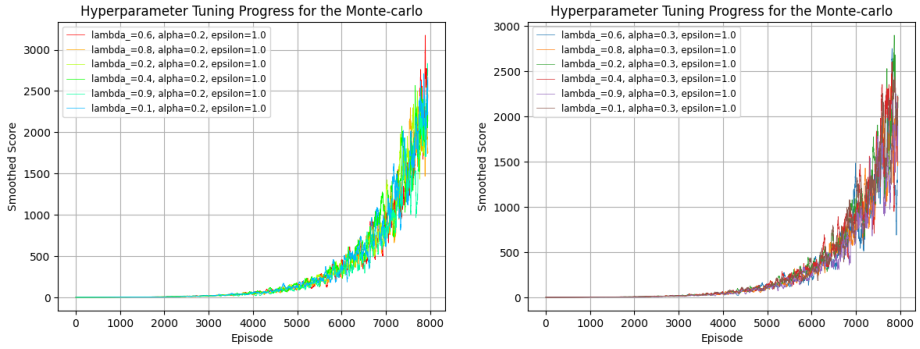


Fig. 5: Hyperparameters tuning for the Sarsa Agent



$\alpha = 0.2$ $\alpha = 0.3$

Fig. 6: Tuning λ for the Sarsa(λ) Agent

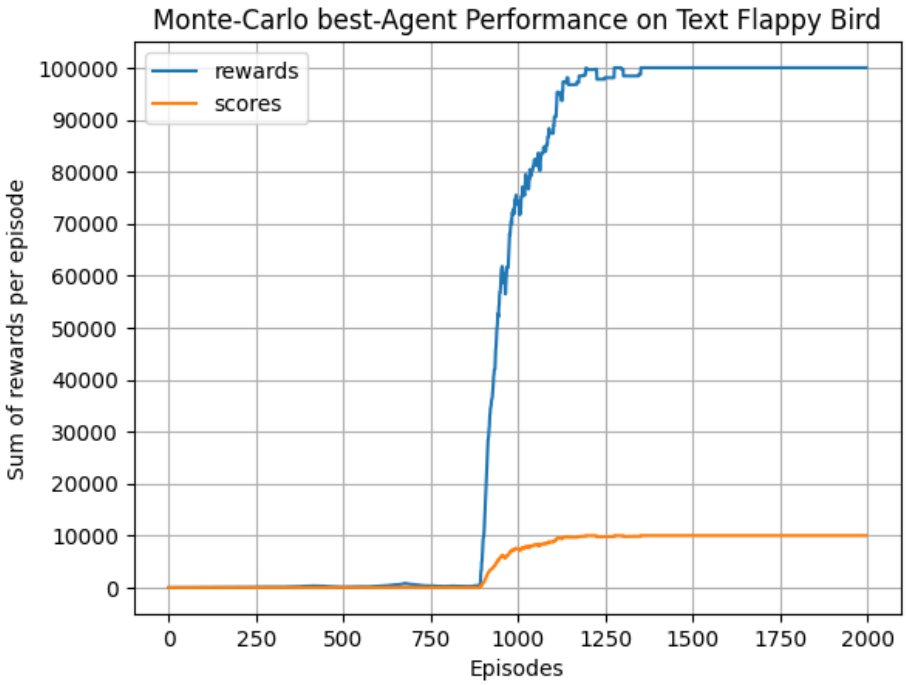


Fig. 7: Training of Monte-Carlo Agent with best parameters

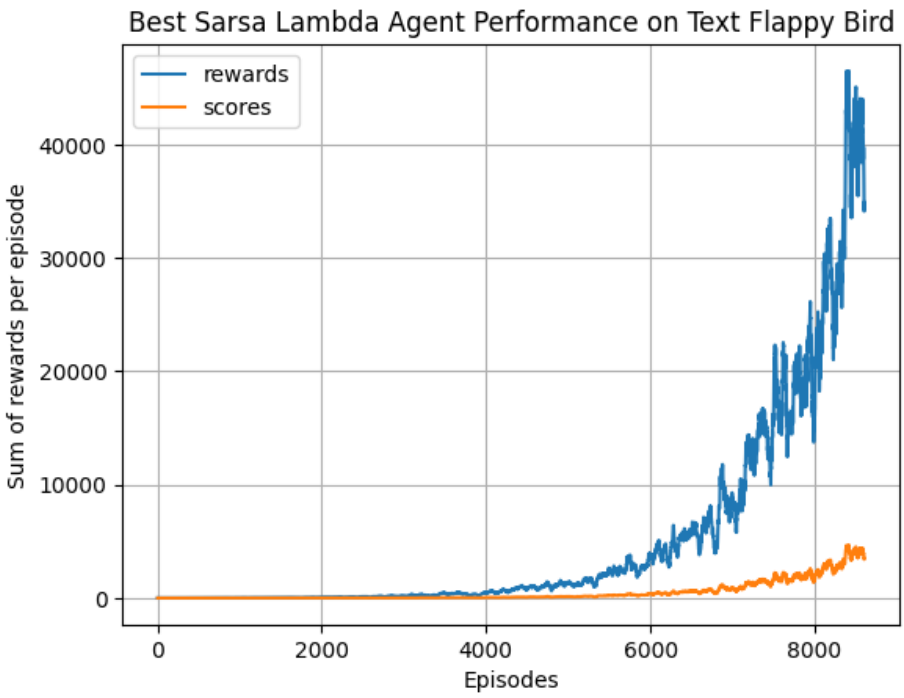


Fig. 8: Training of Sarsa Agent with best parameters