3MD3220: Reinforcement Learning Individual Assignment

Ibrahim Al Khalil RIDENE

March 2025

GITHUB link of the project: **click on me**

Contents

1 Introduction **1**

2 Comparison of TFB Environment Variants **1**

3 Monte Carlo Agent **1**

4 Sarsa(λ) Agent **2**

5 Impact of Hyperparameters **2**
5.1 Impact of Learning Rate 2
5.2 Impact of Epsilon ϵ 4
5.3 Impact of λ 6
5.4 Summary of Best Hyperparameters 6

6 Comparison Between Best Models **7**
6.1 Average Reward 7
6.2 State-Value Function 7

7 Impact of Configuration on Best Model **10**

8 Applicability to the Original Flappy Bird Environment **11**

9 Conclusion **11**

1 Introduction

Reinforcement Learning (RL) is a powerful framework for training agents to make sequential decisions under uncertainty, where an agent learns from interactions with an environment through trial and error [2]. In this project, we focus on a simplified, text-based variant of the classic Flappy Bird game, called *Text Flappy Bird* (TFB), which has been specifically designed for RL experimentation in a discrete and fully observable setting [1].

The TFB environment is provided in two formats: (1) a low-dimensional state representation based on distances from the next pipe gap, and (2) a screen-based version that returns the entire game state as a matrix of integers. The objective of this work is to train and compare multiple RL agents—specifically, a Monte Carlo agent and a Sarsa(λ) agent—on the first TFB environments.

We evaluate the agents on their ability to learn optimal policies, visualize the state-value functions, and perform parameter sweeps over learning rates and trace decay values. In addition, we assess how well these agents generalize across different environment configurations and compare the tabular methods to their potential applicability in pixel-based environments like the original Flappy Bird game [3].

Through this project, we aim to highlight the strengths and limitations of on-policy RL algorithms in discrete dynamic environments, and provide visual and numerical evidence for the behavior of different learning strategies.

2 Comparison of TFB Environment Variants

The Text Flappy Bird (TFB) Gym offers two variants: TextFlappyBird-v0 and TextFlappyBird-screen-v0. The first provides a simplified state space with just two integers: the horizontal distance to the next pipe and the vertical offset from the gap center. The second returns a full 2D grid of the screen, where each cell encodes game elements [1].

`TextFlappyBird-v0` is well-suited for tabular RL due to its low-dimensional state space, making it ideal for educational and algorithmic analysis. In contrast, the screen-based version offers richer, visual-like input but requires state abstraction or function approximation due to its high dimensionality.

This project focuses exclusively on the simplified environment for compatibility with tabular agents.

3 Monte Carlo Agent

We implemented a Monte Carlo (MC) control agent to solve the Text Flappy Bird environment. The agent maintains a state-action value function $Q(s, a)$ and follows an ϵ-greedy policy derived from it. The agent supports both *First-Visit* and *Every-Visit* Monte Carlo variants. This is controlled by a boolean parameter, `first_visit`, in the implementation.

At the end of each episode, the agent computes the return G_t from each time step t onward as:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k,$$

where r_k is the reward at time step k and γ is the discount factor.

For every (or first, depending on the setting) occurrence of each (s, a) pair in the episode, the Q-value is updated using:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(G_t - Q(s, a) \right),$$

where α is the learning rate.

The policy π used for action selection is ϵ-greedy:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|}, & \text{if } a = \arg\max_{a'} Q(s, a'), \\ \frac{\epsilon}{|\mathcal{A}|}, & \text{otherwise.} \end{cases}$$

The MC agent only updates its Q-values after observing complete episodes. This makes it suitable for episodic environments like TFB, but also potentially slower to converge than temporal-difference methods. Despite that, it serves as a strong baseline and provides reliable convergence given enough episodes.

4 Sarsa(λ) Agent

In addition to the Monte Carlo agent, we implemented an on-policy Temporal-Difference (TD) learning algorithm known as Sarsa(λ), which extends one-step Sarsa by incorporating eligibility traces. This approach balances between the bias of TD(0) and the high variance of Monte Carlo methods, enabling more efficient learning in episodic tasks like Text Flappy Bird.

The Sarsa(λ) agent maintains a Q-table $Q(s, a)$ and an eligibility trace table $E(s, a)$ for every state-action pair. At each time step t, the agent observes a transition $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ and computes the temporal-difference (TD) error:

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t),$$

where γ is the discount factor.

The eligibility trace for the current state-action pair is incremented:

$$E(s_t, a_t) \leftarrow E(s_t, a_t) + 1,$$

and all Q-values are updated according to:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\, \delta_t\, E(s, a),$$

followed by decaying all traces:

$$E(s, a) \leftarrow \gamma\lambda\, E(s, a),$$

where $\lambda \in [0, 1]$ is the trace decay parameter and α is the learning rate.

This backward view of TD(λ) allows the agent to propagate credit across multiple prior state-action pairs, making it faster to learn successful behaviors in dynamic environments. When $\lambda = 0$, the method reduces to the classic one-step Sarsa algorithm, and when $\lambda = 1$ (with $\gamma = 1$), it behaves like Monte Carlo.

The agent uses an ϵ-greedy policy to balance exploration and exploitation throughout training.

5 Impact of Hyperparameters

5.1 Impact of Learning Rate

We evaluate how the learning rate α influences learning dynamics for each agent type. For each configuration, we test $\alpha \in \{0.01, 0.1, 0.5, 1.0\}$ and report average episodic reward across 1000 episodes.

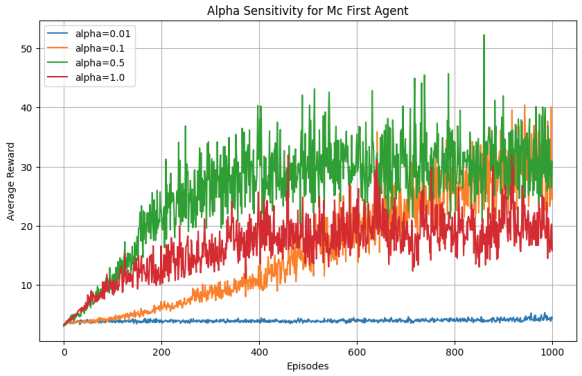

Figure 1: Learning rate sensitivity for Monte Carlo First-Visit agent

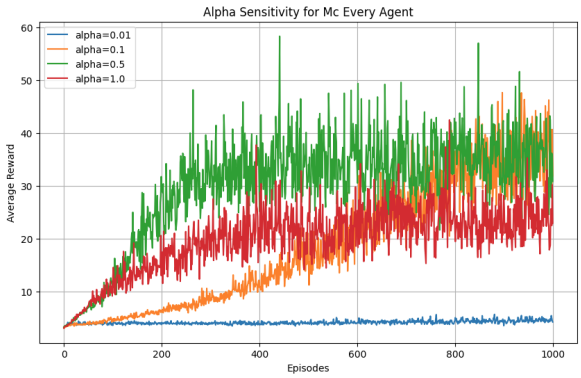

Figure 2: Learning rate sensitivity for Monte Carlo Every-Visit agent

MC First-Visit & MC Every-Visit Analysis:

- $\alpha = 0.01$: Very slow and stagnant learning, showing underfitting.
- $\alpha = 0.1$: More visible learning, but still conservative.
- $\alpha = 0.5$: Yields the best results — high, consistent rewards with stable convergence.
- $\alpha = 1.0$: Fast early gains but unstable performance due to over-updates.

$\rightarrow$ A moderate learning rate such as $\alpha = 0.5$ offers the best trade-off between speed and stability.

Sarsa(0) Analysis:

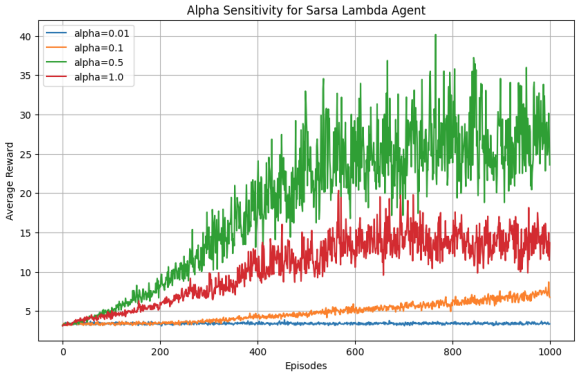

Figure 3: Learning rate sensitivity for Sarsa($\lambda = 0$) agent.

- **Low α values** result in minimal learning.
- $\alpha = 0.5$ leads to faster and more stable learning.
- $\alpha = 1.0$ becomes erratic after early gains.

$\rightarrow$ $\alpha = 0.5$ again strikes the best balance for TD(0)-based learning.

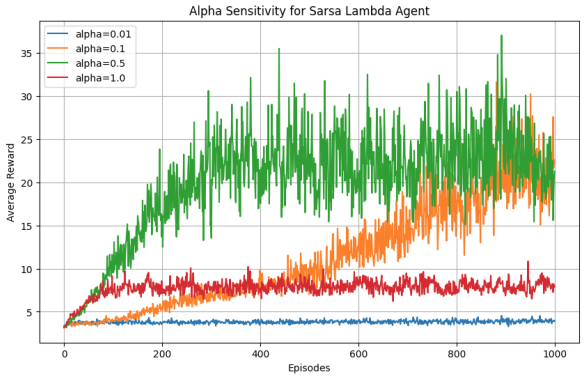

Figure 4: Learning rate sensitivity for Sarsa($\lambda = 0.9$) agent.

Sarsa($\lambda = 0.9$) Analysis:

- Performance improves steadily with increasing α until 0.5.
- $\alpha = 0.5$ offers the highest return and balanced variance.
- $\alpha = 1.0$ performs worse due to unstable Q-updates across traces.

$\rightarrow$ The best performance is again achieved at $\alpha = 0.5$.

Overall Insight:
Across all agent types, $\alpha = 0.5$ consistently provides the most favorable trade-off between learning speed and value stability. Too small values lead to underfitting, and high values cause instability.

5.2 Impact of Epsilon ϵ

We study how the exploration rate ϵ affects learning behavior for each agent. Values tested include $\epsilon \in \{0.0, 0.01, 0.1, 0.4\}$. The average reward across episodes reveals how well agents balance exploration and exploitation.

Monte Carlo (First and Every-Visit) Analysis:

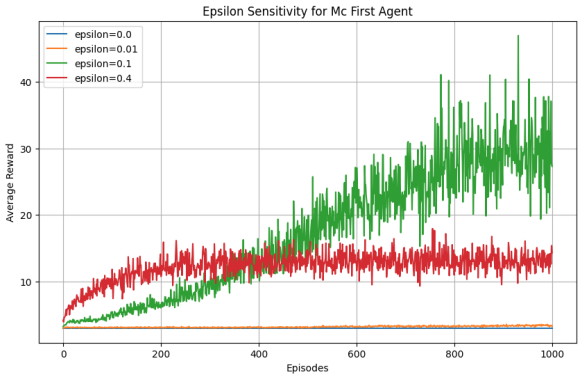

Figure 5: Epsilon sensitivity for MC First-Visit agent.

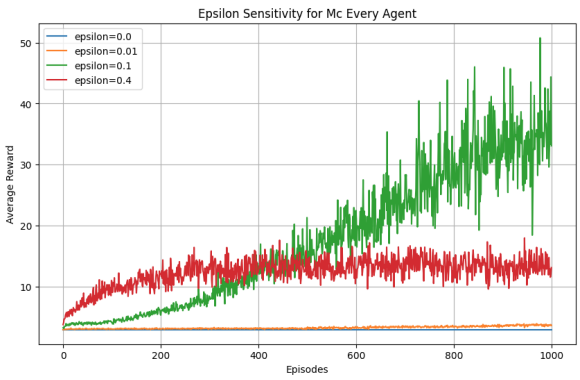

Figure 6: Epsilon sensitivity for MC Every-Visit agent.

- $\epsilon = 0.0$: The agent fails to improve. Without exploration, it remains stuck in poor initial policies.

- $\epsilon = 0.01$: Slight improvement, but still unable to find better strategies reliably.
- $\epsilon = 0.1$: Yields the best performance, combining exploration and exploitation effectively.
- $\epsilon = 0.4$: Leads to faster early learning but prevents long-term policy exploitation due to excessive randomness.

→ An intermediate exploration rate like $\epsilon = 0.1$ provides the most stable and high-performing behavior for both MC variants.

Sarsa($\lambda = 0$) Analysis:

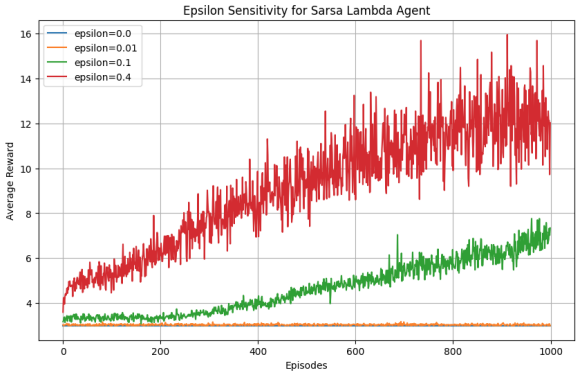

Figure 7: Epsilon sensitivity for Sarsa($\lambda = 0$) agent.

- $\epsilon = 0.0$: No learning occurs — the agent gets stuck in its initial policy.
- $\epsilon = 0.01$: Very slow learning and low overall rewards.
- $\epsilon = 0.1$: Some improvement with moderate exploration.
- $\epsilon = 0.4$: Strongest performance — exploration allows effective policy discovery and consistent improvement.

→ For TD(0), a higher exploration rate like $\epsilon = 0.4$ is essential for meaningful learning and success.

Sarsa($\lambda = 0.9$) Analysis:

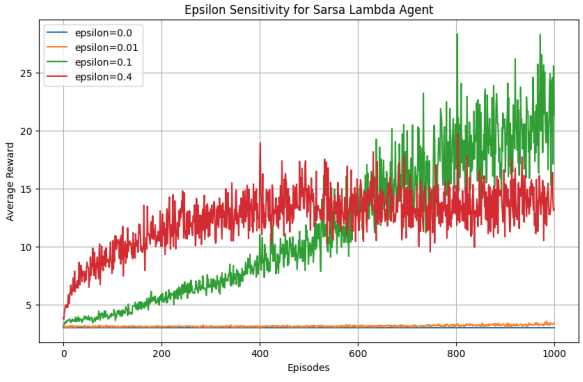

Figure 8: Epsilon sensitivity for Sarsa($\lambda = 0.9$) agent.

- Moderate values of ϵ (particularly 0.1) perform best.
- Very low ϵ inhibits discovery of effective strategies.
- Very high ϵ leads to noisy learning and lower returns due to insufficient exploitation.

$\rightarrow$ Sarsa($\lambda = 0.9$) performs best with $\epsilon = 0.1$, leveraging exploration while stabilizing learning via eligibility traces.

5.3 Impact of λ

We analyze the effect of the eligibility trace parameter λ on the performance of the Sarsa(λ) agent. The plot below shows the average reward across 1000 episodes for varying values of $\lambda \in \{0.0, 0.3, 0.6, 0.9, 1.0\}$.

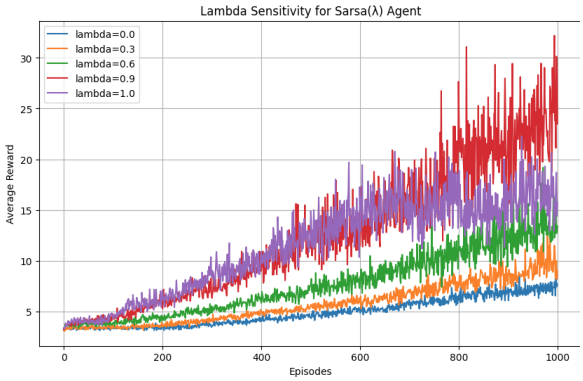

Figure 9: Lambda sensitivity for Sarsa(λ) agent.

The results show a clear positive correlation between higher λ values and performance. When $\lambda = 0$ (standard one-step Sarsa), the agent learns slowly and converges to a lower average reward. As λ increases, the agent benefits from longer eligibility traces, which help assign credit across multiple steps, improving learning efficiency.

- $\lambda = 0.6$ **and** 0.9 show strong performance, with $\lambda = 0.9$ achieving the best final rewards and steepest improvement. - $\lambda = 1.0$ performs well initially but shows slightly more variance, possibly due to full returns causing instability in certain transitions.

$\rightarrow$ Setting λ to a high value such as 0.9 provides the most effective trade-off between long-term credit assignment and learning stability. Lower λ values lead to slower, less effective learning in this environment.

5.4 Summary of Best Hyperparameters

Based on the sensitivity analysis of learning rate α and exploration rate ϵ, the following hyperparameters were found to yield the best results for each agent:

Agent Type	**Best α**	**Best ϵ**
MC First-Visit	0.5	0.1
MC Every-Visit	0.5	0.1
Sarsa($\lambda = 0$)	0.5	0.4
Sarsa($\lambda = 0.9$)	0.5	0.1

Table 1: Optimal hyperparameters for each agent based on performance experiments.

6 Comparison Between Best Models

6.1 Average Reward

We compare the average episodic reward of the best hyperparameter configurations for each agent across 1000 episodes.

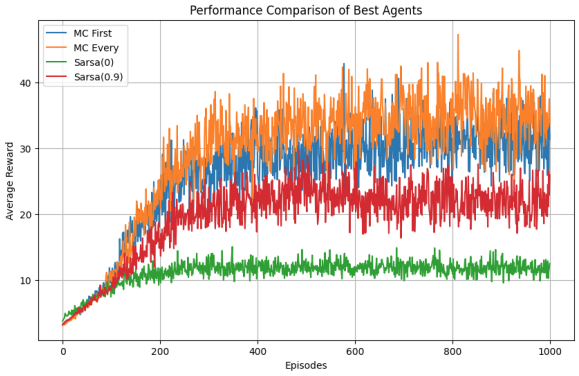

Figure 10: Average reward comparison of best-tuned agents.

- **MC Every-Visit** is the top performer. It converges quickly and maintains the highest and most consistent reward, thanks to frequent value updates and balanced exploration.
- **MC First-Visit** also performs strongly but with slightly lower and more variable rewards than its Every-Visit counterpart.
- **Sarsa($\lambda = 0.9$)** reaches moderate rewards and shows stable, smooth learning. Eligibility traces help assign credit across time, but performance is below that of MC agents.
- **Sarsa($\lambda = 0$)** performs the worst overall. Its reliance on one-step updates and high ϵ makes it slow to converge and prone to suboptimal learning.

The MC agents are generally more sensitive to early parameter tuning but converge faster with higher scores. Sarsa(0.9) trades off some reward for stability and broader generalization. Sarsa(0), with limited credit propagation, lags behind in both learning speed and total reward.

Conclusion:
MC Every-Visit offers the best average reward performance overall, combining efficient exploration and value estimation.

6.2 State-Value Function

The heatmaps below visualize the learned state-value function $V(s) = \max_a Q(s, a)$ for each of the best-trained agents. States are defined by:

- **x-axis**: Horizontal distance to the next pipe - **y-axis**: Vertical offset from the center of the pipe gap

Monte Carlo First-Visit Agent

- Higher values are concentrated in regions with moderate horizontal distance ($x = 4$–6) and vertical offset close to 0 or slightly positive.
- Indicates the agent values states where it is aligned or slightly above the pipe gap, near an obstacle.

Figure 11: State-value function of MC First-Visit agent.

- Low values in zones with poor alignment or early horizontal positions ($x = 0$–2) indicating poor expected return or early termination.

→ The MC First-Visit agent learns to value well-aligned and reactive positions, showing a sharp understanding of critical decision states.

Monte Carlo Every-Visit Agent

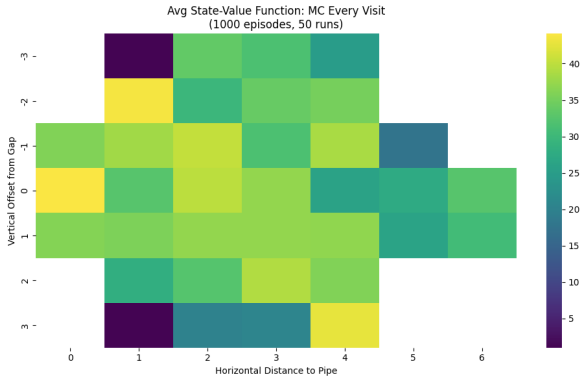

Figure 12: State-value function of MC Every-Visit agent.

- Values are more uniformly high across a broader area.
- Highest values occur for vertical offsets near zero and horizontal positions from $x = 1$ to 4.
- Unlike MC First, this agent learns to assign high value even when the pipe is farther, suggesting a more general understanding of good positions.
- There's less sharp contrast between "good" and "bad" zones, implying smoother generalization.

→ Frequent updates lead to better generalization, producing a smoother, more robust value landscape than the First-Visit variant.

Sarsa($\lambda = 0$) Agent

- Overall value estimates are lower and more sparse than Monte Carlo agents.

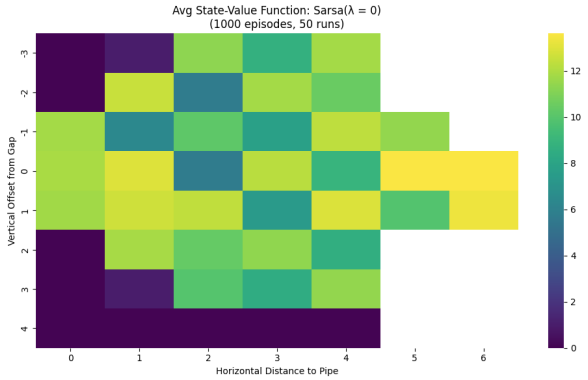

Figure 13: State-value function of Sarsa($\lambda = 0$) agent.

- High-value states are concentrated around the **centered vertical offset (-1 to +1)** and **moderate-to-high horizontal distances**, where the bird is better aligned and has time to act.
- Several corner states (e.g., edges and far vertical deviations) are poorly estimated or valued as 0, which may result from insufficient visits or early termination in these states.
- The map is more **sparse and abrupt**, showing limited generalization and coverage.

→ Sarsa(0), while capable of learning local transitions, struggles to propagate long-term credit across multiple steps. This results in a more conservative and fragmented value map, consistent with its lower episodic reward performance.

Sarsa($\lambda = 0.9$) Agent

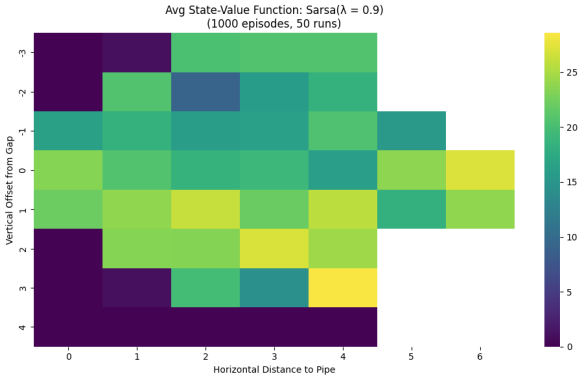

Figure 14: State-value function of Sarsa($\lambda = 0.9$) agent.

- High-value zones appear near optimal alignment and pipe proximity.
- The heatmap is smoother and more complete than Sarsa(0).
- Eligibility traces enable broader credit assignment across sequential states.

→ Sarsa($\lambda = 0.9$) balances local accuracy with smoother generalization, enabling it to learn more complete policies than its TD(0) counterpart.

7 Impact of Configuration on Best Model

Based on the subsection 6.1 , we found that MC Every-Visit offers the best average reward performance overal. Thus, we will use this agent in this experiment.
This plot compares the MC Every-Visit agent's performance on a variety of environment configurations, using the same trained policy from the baseline setup (`height=15`, `width=20`, `pipe_gap=4`). Rewards are averaged over multiple evaluation runs, with error bars representing the standard deviation.

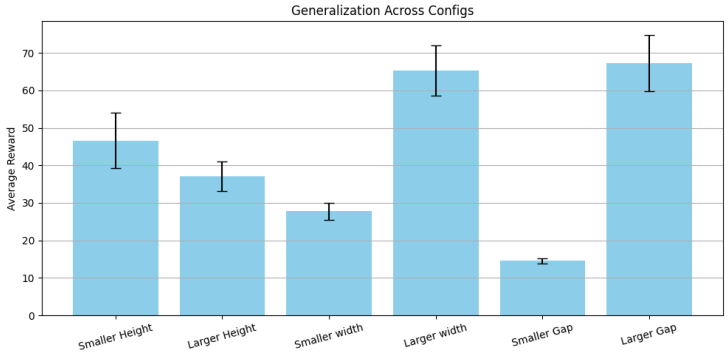

Figure 15: Generalization across environment configurations for the MC Every-Visit agent.

Height Modifications

- **Smaller Height:** The agent performs strongly with a high average reward and moderate variance. The reduced vertical space does not hinder the learned policy significantly and may even simplify the game by constraining the movement range.
- **Larger Height:** Performance drops slightly compared to the smaller height setup. The increased vertical space introduces more room for error, especially if the agent was trained in a more compact vertical range.

→ The agent generalizes decently to both height changes, with better adaptation to smaller vertical bounds.

Width Modifications

- **Smaller Width:** Performance declines notably. The reduced horizontal space offers less time for the agent to react between pipes, breaking its timing-based policy and resulting in lower rewards.
- **Larger Width:** The agent performs exceptionally well here — even better than the baseline. The increased width gives more reaction time and smoother transitions between obstacles, allowing the learned policy to succeed more reliably.

→ The agent generalizes poorly to narrower widths but excels in wider ones, benefiting from longer decision windows.

Gap Modifications

- **Smaller Gap:** The most challenging configuration. The agent performs poorly here, with both low rewards and low variance. This indicates consistent failure due to tighter navigation space — the agent likely crashes frequently.

- **Larger Gap:** The highest-performing configuration. The easier task makes the learned policy highly effective, even if slightly misaligned. The agent performs reliably and achieves high rewards.

→ The agent struggles when the pipe gap is narrower than during training but thrives when the gap is larger.

Final Insight

The MC Every-Visit agent shows **strong generalization** to configurations that make the environment **more forgiving** (larger width/gap), but suffers under **tighter constraints** (narrower pipes or width). This highlights the need for retraining or function approximation when transferring agents to more difficult levels.

8 Applicability to the Original Flappy Bird Environment

The agents implemented in this project were designed for the simplified version of the Text Flappy Bird environment, where the state is represented by discrete features such as horizontal and vertical distances to the next pipe. While these agents can, in theory, be applied to the original Flappy Bird Gym environment [3], doing so would require adapting the input space. The original environment provides continuous observations, so discretization or function approximation (e.g., neural networks) would be necessary for the agents to operate effectively. The action space remains compatible, but the current tabular implementations would need modification to handle the richer state representations.

9 Conclusion

This project explored reinforcement learning approaches for solving the Text Flappy Bird game using Monte Carlo and Sarsa(λ) agents. After evaluating different hyperparameters, the Monte Carlo Every-Visit agent emerged as the best performer.

We analyzed agent behavior through learning curves, state-value functions, and generalization tests. Results show strong performance in forgiving environments, but limitations under tighter constraints. Overall, the study highlights the effectiveness of simple RL methods and the importance of tuning and environment design.

References

[1] Stergios Christodoulidis. Text flappy bird gym environment. https://gitlab-research.centralesupelec.fr/stergios.christodoulidis/text-flappy-bird-gym, 2025. GitLab Repository.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT press, 2nd edition, 2018.

[3] Talendar. Flappy bird gym environment. https://github.com/Talendar/flappy-bird-gym, 2018.