Reinforcement Learning - Individual assignment

Ryad Guezzi[1]

https://github.com/ryadguezzi/flappy-rl

Abstract. We investigate reinforcement learning approaches applied to Text Flappy Bird, a simplified version of the classic Flappy Bird game. We implement two agents — a Monte Carlo-based agent and a Sarsa(λ) agent, both using a linear Q function approximation with Fourier state representation. A comprehensive experimental setup is presented, including a detailed hyperparameter sweep and performance evaluation on varying environment configurations. Results indicate that the Sarsa(λ) agent demonstrates stable convergence and robustness to parameter variations, while experiments with the Monte Carlo agent suggest less promising results.

1 Framework

1.1 Environment

For this work, the text-based Flappy Bird with 2D state representation was used. As a reminder, every state of the game is represented by the horizontal and vertical coordinates of the vector linking the bird to the center of the next gate. The action space has only two elements, 0 and 1, representing whether the bird flaps or not. One might argue this should be enough information to determine the correct action, since intuitively past gates and gates beyond the next have no or negligible influence on the right action; the bird just has to align with the next gate.

However, this representation comes with limitations. Most notably, it ignores the bird's velocity, which plays a crucial role in determining the success of a flap or the decision to wait. The dynamics of the game are such that the bird's movement is not instantaneous — the consequences of an action depend not only on the current position but also on the motion trajectory, making the system partially observable under this simple 2D representation.

The alternative representation is a full rendering of the screen as the state, i.e., a pixel-based image input. While this would capture more relevant information, like where the edges of the gates exactly are, it also significantly increases the dimensionality of the problem. This may require using more powerful function approximators (e.g., deep convolutional networks), longer training time, and more complex architecture, whereas our goal here is to remain within a relatively simple, and computationally efficient framework.

1.2 Function Approximation with Fourier Features

To approximate the action-value function $Q(s, a)$, I used a linear model over Fourier basis features. Usage of Fourier features is mentioned in [1], section 9.5.2. This approach maps each state s (normalized to have coordinates in $[0, 1]$) to a feature vector $\mathbf{x} \in R^(n + 1)$, where n is a fixed integer. Let n be fixed and

$$\{c_{i,j} = (i, j)^T, 0 \leq i, j \leq n\}$$

the set of frequency vectors. Then for $0 \leq i < (n + 1)^2$ we define the feature vector with

$$x_i = cos(\pi s^T c^i)$$

.

We also define two weight vectors, for the two possible actions : w_0 and w_1 that are $(n + 1)^2$-dimensional.

The Q function is then approximated as $Q(s, a) = \hat{Q}(x, a) = w_a^T x$. The goal is to learn w. The larger n is, the better will be the Fourier approximation of the value function.

The motivation behind using Fourier features lies in their ability to approximate any continuous function over a compact domain. In particulat, since the Flappy Bird environment exhibits patterns that are somewhat periodic (gates appearing at regular intervals, bird motion oscillating due to gravity and flapping), Fourier features offer a natural and efficient basis to capture the shape of the optimal value function without needing deep architectures. In addition, since the Fourier basis is fixed and orthogonal, training becomes a matter of learning a simple linear model, which is fast and stable.

This approach also facilitates interpretability, as the influence of each frequency component in decision-making can be studied through the learned weights.

2 SARSA(λ)

The Sarsa(λ) algorithm belongs to the class of eligility-trace algorithms: by keeping track of past returns, one could combine the k-step TD errors for multiple values of k, and do a kind of weighted average of these errors to adjust the Q-value function at current step. This is basically what SARSA(λ) does, with the weight of the k-step error being proportional to λ^k. λ is thus a hyperparameter in $[0, 1]$; with $\lambda = 0$, Q-value update only uses one-step TD error, so it's the classical temporal difference algorithm, while $\lambda = 1$ means only the ∞-step error is counted, so it's a kind of Monte-Carlo method. The implementation of this method uses an eligibility trace vector, a clever way to compute the contribution of past returns with low complexity.

Results The parameters used to train this agent were $n = 15, \lambda = 0.9, \alpha = 0.005, \gamma = 0.99, \epsilon_0 = 0.1$; and the number of episodes in the simulation is 10^4. The implementation uses a decaying exploration rate ϵ, so that the agent explores a lot at the beginning and less once it has experience.

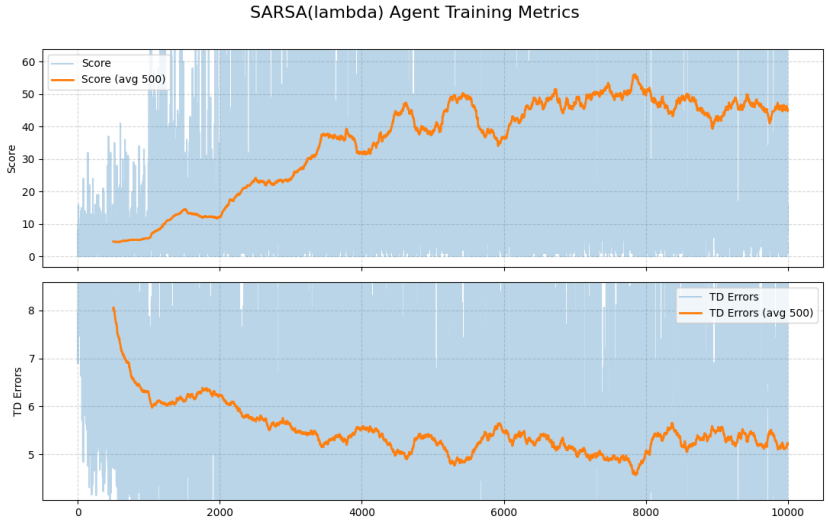

Fig. 1. SARSA(λ) training : smoothed score and avg square TD error per episode

The method works well, with the agent being able to score more than 50 gates reliably (the orange curves are averaged over windows of 500 episodes). We clearly see both the score curve increasing and the average squared TD error decreasing as the training goes. One can estimate the time of satisfactory training at around 5000 episodes, but it is possible that longer training gives arbitrarily high scores: we can reasonably think that if the model can pass 50 gates, it can pass any number of gates.

State-action value function Once the agent trained, we can plot the state-value function for both actions. The variable is the normalized state vector.

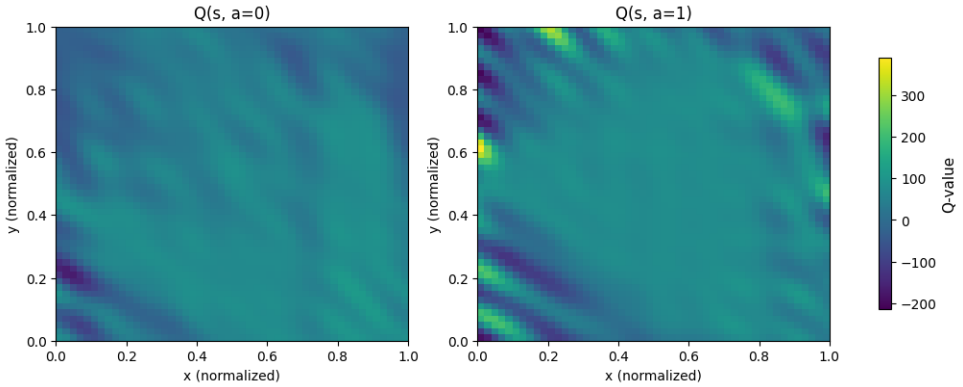

Fig. 2. Heatmap of Q(s,a) for SARSA(λ)

These plots of the learned Q-values are very interesting because they illustrate the periodic nature of the state-action value function, particularly on the left side of each plot. This periodicity is an intuitive result of the game's dynamics, and justifies furthermore the choice of Fourier representation.

Since the bird's movement follows an oscillatory trajectory due to the gravity-flap interaction, the optimal decision to flap depends on whether the induced oscillation aligns favorably with the upcoming pipe gap. For example, when the bird is at the same height as the gap and near the obstacle, the decision to flap is beneficial only if the resulting vertical oscillation allows the bird to pass through the gap at the right time. This creates structured patterns in the Q-function, where certain positions exhibit a preference for flapping, while others favor waiting for the right moment.

Parameters sweeping The effect of the hyperparameters on performance was also studied. The parameters identified as pertinent are λ, n, α and ϵ_{min}, the limit inferior bound of decaying ϵ. We tried to optimize the performance using the following methodology : a few hundreds sets of parameters are picked randomly, with each parameter in a chosen range; then agents are trained in parallel for each of those sets of hyperparameters on 5000 episodes; then they are evaluated by taking the average score on the last 1000 episodes or so as the metric. The top 10 best agents are kept and their score functions are graphed below.

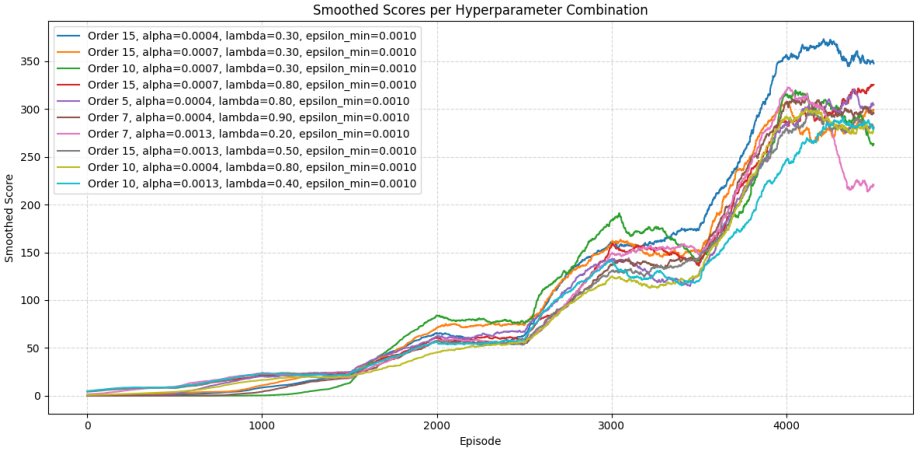

Fig. 3. Smoothed scores learning curves for the ten selected agents.

We see that all these models give impressive results, way above our initial agent, they can learn to get scores up to 350 in only 5000 episodes. It seems the best set of parameters is $n = 15, \lambda = 03, \alpha = 4.10^{-4}, \epsilon_{min} = 10^{-3}$. The values of the parameters ϵ_{min} and α are close for all agents in the top 10, so they have a real effect on performance. Values of λ and n are more diverse so we need to

be careful about overfitting them. The pattern of progression is remarkable and exhibits alternating periods of fast progress and stagnation. The training was stopped in a stagnation period, but it seems like the agents could get infinitely better.

3 Monte-Carlo

A classical on-policy every-visit Monte-Carlo approach was implemented. The Monte Carlo method updates the value of actions based on the total reward received at the end of each episode. The value updates happen only once the episode is completely ended. In our implementation the state-action value approximation and the Fourier representation stay the same, and the goal is still to learn w. The hyperparameters are the same, minus λ.

Results The parameters used to train this agent were $n = 15, \alpha = 0.005, \gamma = 0.99, \epsilon_0 = 0.1$; and the number of episodes in the simulation is again 10^4. The implementation uses a decaying exploration rate ϵ, so that the agent explores a lot at the beginning and less once it has experience.

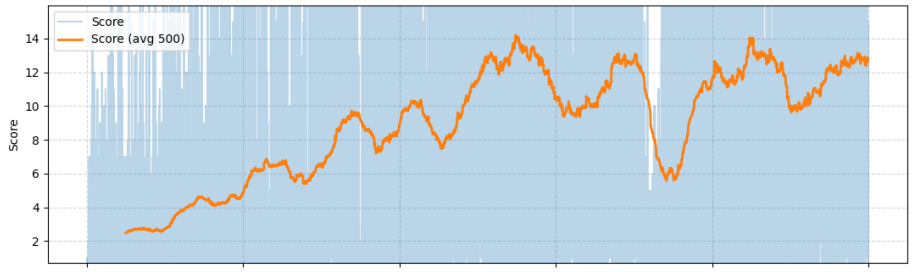

Fig. 4. Monte-Carlo learning : smoothed score per episode

This method still provides a convergent agent and performs decently, obtaining up to 14 of score. The learning profile is somewhat jerky with in particular a pretty large fall in performance around episode 7500. Returns are only updated after full episodes, so the agent's value estimates can shift dramatically from one episode to another, especially under our ϵ-greedy exploration strategy. It is known that Monte-Carlo can exhibit higher variance than other methods, so discovering (and persisting with) a suboptimal action sequence can temporarily degrade performance.

State-action value function We can once again graph the Q function heatmaps :

The global tendency to periodicity in the particular diagonal direction is still visible, so like with SARSA, higher coefficients are given to features associated with $c_{i,j}$ vectors pointing diagonally. If we compare it to the one for SARSA(λ),

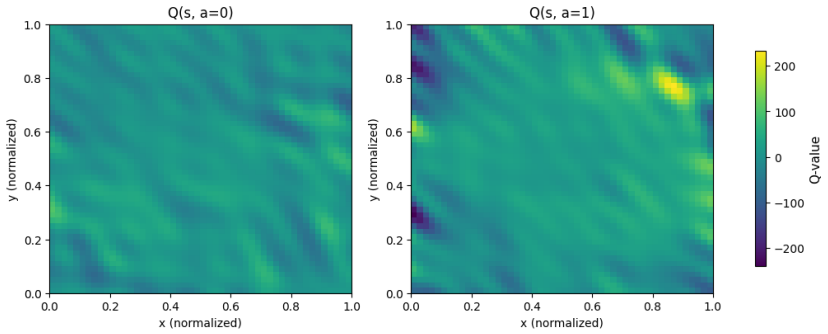

Fig. 5. Heatmap of Q(s,a) for Monte-Carlo

we see that the patterns are way less striking. This can confirm the Monte-Carlo agent hasn't completely learned the game mechanics.

Parameters sweeping The same procedure was executed with Monte-Carlo agents.

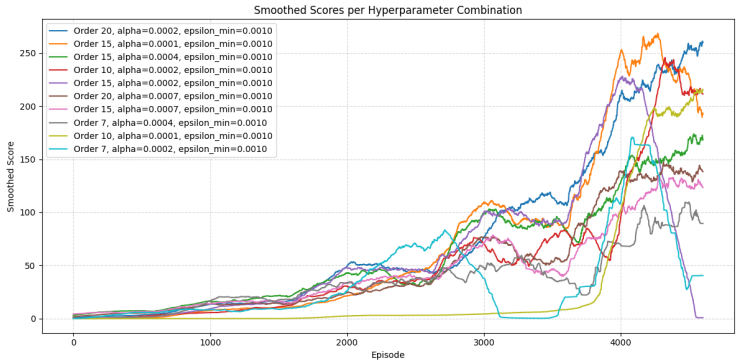

Fig. 6. Smoothed scores learning curves for the ten Monte-Carlo selected agents.

As explained earlier, we see dramatic changes during the learning, with some models even unlearning everything. This might be due to the decaying ϵ that happens every 1000 steps. Also this time having a larger n seems to benefit performance. This high variance phenomena might mean the parameters are actually overfit.

4 Discussion

Robustness with the environment We discussion in section 1 about the two environments types and their advantages and drawbacks. Now that we trained our

models, we can ask ourselves is the agents are robust to changes in that environment. To test that, we created a new instance of the environment with different window size, then a new Sarsa Agent instance with $\alpha = 0$ so it doesn't learn, and we initialized its weight vector with the one obtained after training the first agent (knowledge transfer). We might except that the agent still solves the problem since coordinates are noralized anyways so it shouldn't be too dependant on the screen size. Here are the settings and results : **Agent 2**: width=25, height=30, pipe gap=8; SCORE = 60 **Agent 3**: width=10, height=15, pipe gap=4; SCORE = 0.55

So it seems the model generalizes well with bigger windows, but can't adapt to smaller ones, there may be too harsh discontinuities in such an environment. Now, if we consider the original flappy bird environment, it would be pretty straightforward to transform a video input to the s vector, and then using the trained agent at every step to tell wether to flap or not. The computation is possible in real-time since once the weight fixed, there are just two small dot products to evaluate at each step.

Comparison of the two agents Overall, the SARSA agent exhibits more consistent and robust performance compared to the Monte Carlo agent. The SARSA method benefits from continuous updates, allowing it to adjust its Q-value estimates more smoothly and with lower variance, leading to faster convergence and higher, more stable scores. In contrast, the Monte Carlo agent, while still convergent, updates its value estimates only at the end of each episode, resulting in more abrupt changes in the learning curve. This can even cause the Monte Carlo method to temporarily unlearn previously acquired knowledge. Despite these differences, both agents ultimately develop similar state-action value functions, suggesting that they capture the underlying dynamics of the game in comparable ways.

References

1. R.S. Sutton, A.G. Barto: Reinforcment learning : an introduction. The MIT Press, 2018
2. Christodoulidis Stergios, Reinforcment learning, Course at CentraleSupélec, 2025