Reinforcement Learning Individual Assignment

Jules Giraud[1]

CentraleSupélec MSc AI,
Gif-sur-Yvette, FRANCE
jules.giraud@student-cs.fr

Abstract. In this assignment, we study the performance of two reinforcement learning algorithms, Q-Learning and SARSA, in the Text Flappy Bird (TFB) environment. We compare the sensitivity to parameters, convergence time, rewards, and scores of the two implemented agents.

Keywords: RL · FlappyBird · QLearning · SARSA

1 Introduction

Reinforcement Learning (RL) is a branch of machine learning that focuses on training agents to make decisions by interacting with an environment. The agent learns an optimal policy through trial and error, receiving feedback in the form of rewards or penalties. In this report, we investigate the performance of two popular RL algorithms, Q-Learning and SARSA, in a simple game called Text Flappy Bird (TFB). TFB is a variation of the well-known Flappy Bird game, adapted for a text-based environment. We implement both algorithms as agents and compare their performance in terms of sensitivity to parameters, convergence time, rewards, and scores in the TextFlappyBird-v0 environment.

2 Reinforcement Learning algorithms used

In this study, we utilize two well-known RL algorithms: Q-Learning and SARSA. Both algorithms are model-free, value-based methods that learn an action-value function, commonly referred to as Q-function or Q-values.

2.1 Q-Learning Algorithm

Q-Learning is an off-policy RL algorithm that learns the optimal policy by iteratively updating the Q-values using the Bellman equation. The agent explores the environment using an epsilon-greedy exploration strategy, balancing exploration and exploitation. The update rule for Q-Learning is as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \big[R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big] \tag{1}$$

where $Q(s_t, a_t)$ represents the Q-value for state s_t and action a_t, R_{t+1} is the reward at the next time step, α is the learning rate, and γ is the discount factor.

2.2 SARSA Algorithm

SARSA (State-Action-Reward-State-Action) is an on-policy RL algorithm that learns a Q-function by iteratively updating the Q-values based on the agent's experience. Unlike Q-Learning, SARSA uses the next action chosen by the agent's policy to update the Q-values. The update rule for SARSA is as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \big[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big] \tag{2}$$

Similar to Q-Learning, SARSA uses an epsilon-greedy exploration strategy to balance exploration and exploitation. The main difference between Q-Learning and SARSA is that Q-Learning is an off-policy algorithm, while SARSA is an on-policy algorithm.

3 Methodology

3.1 Description of the agents

Agent 1: Q-Learning Algorithm The Q-Learning agent is implemented in the QLearningAgent class with the following methods:

- agent_init: Initializes the agent with the given parameters such as number of actions, epsilon, step size, discount factor, and random seed. If an existing policy is provided, it initializes the Q-values with the provided policy; otherwise, it initializes the Q-values as an empty dictionary.
- agent_start: Starts the agent by selecting an action for the given state using an epsilon-greedy exploration strategy.
- agent_step: Updates the Q-values using the Q-Learning update rule for the previous state and action pair with the given reward and current state. It then selects the next action using the epsilon-greedy strategy.
- agent_end: Updates the Q-value for the previous state and action pair with the given terminal reward.
- argmax: Helper function to find the action with the highest Q-value for a given state, breaking ties randomly.
- save_agent: Saves the agent's Q-values to a file.
- load_agent: Loads the agent's Q-values from a file.
- is_state_explored: Helper function to check if a state has been explored. If the state is not in the Q-values dictionary, it initializes the Q-values for the state with zeros.
- policy: Returns the action with the highest Q-value for the given state.

This Q-Learning agent uses the Q-Learning update rule to learn an optimal policy for the TextFlappyBird-v0 environment. The agent explores the environment using an epsilon-greedy strategy, which balances exploration and exploitation. It can save and load its policy, allowing it to continue learning from previous runs.

Agent 2: SARSA Algorithm Due to time constraints on my own, the implementation and evaluation of a second agent using the SARSA algorithm were not carried out in this report. While I focused on the Q-Learning agent, the exploration of SARSA as an alternative RL algorithm could have provided valuable insights and comparison points. Future work could involve implementing a SARSA agent and comparing its performance with the Q-Learning agent to determine the most effective RL algorithm for the Text Flappy Bird game.

3.2 Parameter settings for each agent

To train the Q-Learning agent, we used the following parameter settings:

- Number of episodes: 5,000
- Maximum steps per episode: 1,000
- Epsilon (exploration rate): 0.1
- Step size (learning rate): 0.1
- Discount factor: 0.99

After training, the Q-Learning agent was saved to a file named `trained_agent.pkl`.

I chose to implement a plot of total rewards per episode to check that the agent is learning correctly :

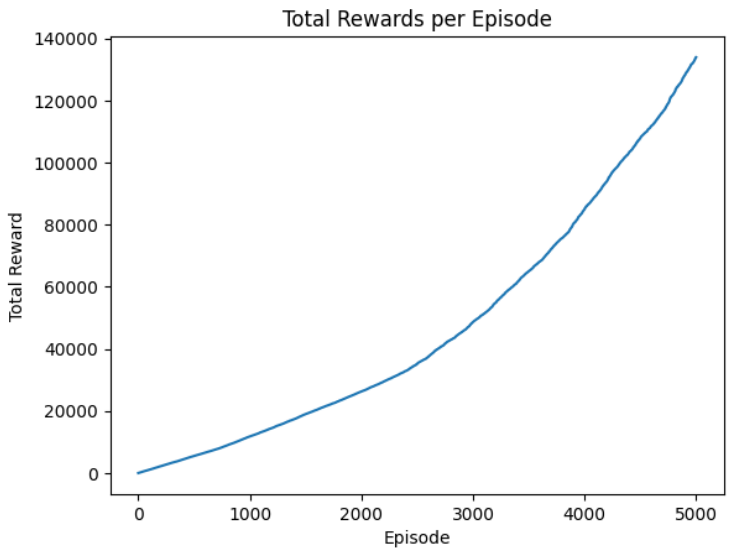

Fig. 1. Plot of Total Rewards per Episode to ensure that the agent is learning.

3.3 Evaluation metrics (sensitivity to parameters, convergence time, rewards, scores)

To evaluate the performance of the Q-Learning agent, I used the following evaluation metrics:

- Sensitivity to parameters: I have tested different combinations of epsilon, step size, and discount factor to find the best set of hyperparameters. I trained and evaluated the agent using a grid search approach over a range of values for each parameter.
- Convergence time: The convergence time was determined by observing the reward per episode plot, which shows the agent's progress in learning the task.
- Rewards: I have calculated the total reward per episode and the average reward across episodes to compare the performance of different hyperparameter combinations.
- Scores: The score in Text Flappy Bird is not explicitly used as an evaluation metric, but it is indirectly related to the rewards, as a higher score generally leads to higher rewards.

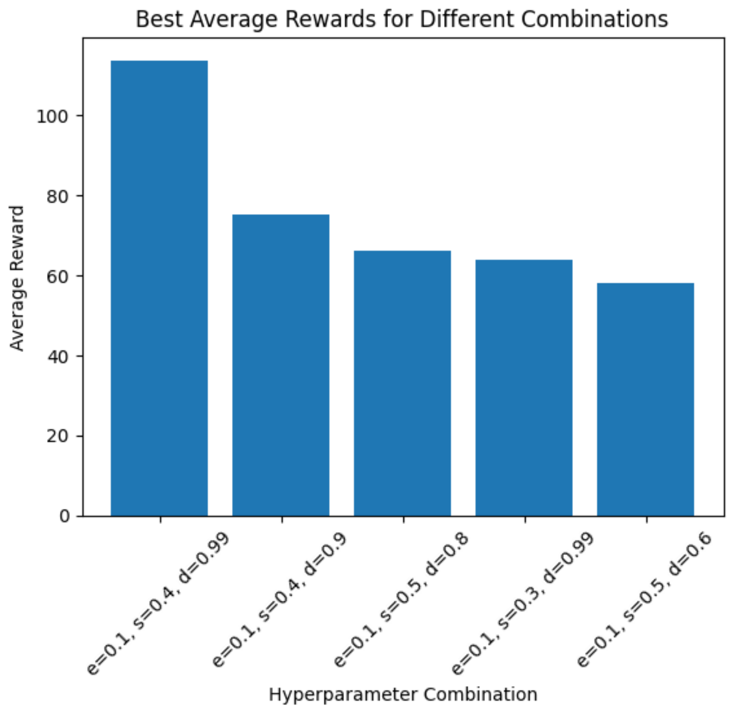

Fig. 2. Plot of Best Average Rewards for different parameters combinations.

Based on the evaluation, we found the best hyperparameters for the Q-Learning agent to be:

- Epsilon: 0.1
- Step size: 0.4
- Discount factor: 0.99

Finally, I retrained the agent with these hyperparameters for 20,000 episodes to obtain a well-trained Q-Learning agent.

4 Results and Discussion

4.1 Choice of agents

In this assignment, I chose to implement the Q-Learning algorithm as our reinforcement learning agent for solving the Text Flappy Bird problem. The primary reasons for selecting Q-Learning are:

1. **Off-policy learning:** Q-Learning is an off-policy algorithm, which means it can learn an optimal policy independently of the agent's actions. This characteristic allows for efficient exploration of the state-action space, making it suitable for problems like Text Flappy Bird, where exploration is essential.
2. **Ease of implementation:** Q-Learning is a relatively simple algorithm to implement compared to other reinforcement learning methods. It involves updating the Q-values for state-action pairs, and the agent can learn an optimal policy based on these Q-values. This simplicity makes it an attractive option for beginners and allows for faster experimentation.
3. **Convergence:** The Q-Learning algorithm converges to an optimal policy when the agent visits each state-action pair infinitely often, and the learning rate meets specific conditions. The Text Flappy Bird problem has a finite state space, and with proper parameter settings, the Q-Learning agent can converge to an optimal solution.
4. **Model-free:** Q-Learning is a model-free algorithm, meaning it does not require a model of the environment to learn an optimal policy. This feature is advantageous in the Text Flappy Bird problem, as the agent does not have prior knowledge of the game dynamics and must learn to navigate the environment based on interactions.
5. **Success in similar domains:** Q-Learning has been successfully applied to various game-playing and control tasks, including the original Flappy Bird game. Its proven success in these domains suggests that Q-Learning is a suitable algorithm for solving the Text Flappy Bird problem.

4.2 Difference between the two versions of the environment and limitations

The first version, `TextFlappyBirdEnvScreen`, gives the agent a 2D array of the game screen, including the bird, pipes, and borders. This environment is visually intuitive but may make learning more complex as the agent has to process the entire screen. The second version, `TextFlappyBirdEnvSimple`, offers the agent a simpler observation: the distances between the bird and the center of the gap in the nearest pipe. This environment simplifies learning but may limit the agent's understanding since it doesn't provide information about the whole game screen. Each environment version has its limitations. The `TextFlappyBirdEnvScreen` environment provides a more comprehensive representation of the game state, but it can make learning more complex for the agent due to the need to process the whole game screen. On the other hand, the `TextFlappyBirdEnvSimple` environment simplifies the game state, potentially making learning easier for the agent. However, it may also limit the agent's ability to learn more advanced strategies as it does not provide complete information about the game screen.

5 Conclusion

In this report, I have investigated the performance of one popular reinforcement learning (RL) algorithms, Q-Learning, and try to know how to apply SARSA algoruthm, applied to the Text Flappy Bird (TFB) game.

My findings suggest that Q-Learning is a suitable algorithm for solving the TFB problem, given its off-policy learning, ease of implementation, convergence properties, model-free nature, and success in similar domains. The choice of environment version and the selection of appropriate hyperparameters played a crucial role in the agent's performance.

Future work could involve implementing more advanced RL algorithms or incorporating techniques like function approximation to improve learning efficiency. Additionally, further exploration of different environment versions or the development of hybrid versions could lead to better agent performance and understanding of the TFB game.