In Search of Action and Reward: The Keys to Reinforcement Learning

Pedro Luiz Silva

January 15, 2024

Contents

1 Introduction

In the field of statistical learning, there are several subdivisions: supervised learning, unsupervised learning, and reinforcement learning.

In the context of reinforcement learning, there are no available annotations but rather rewards provided by the environment. Generally, accomplishing a task using reinforcement learning proves to be much more complex than through supervised learning. This difficulty arises from the fact that rewards are not as explicit to generalize the problem, necessitating thorough exploration.

Nevertheless, reinforcement learning techniques are widely employed in the field of control and also in video games. Their popularity surged after the publication of the paper on Atari games in 2013 [MKS+13].

In this text, we aim to solve simple reinforcement learning problems and study the basic techniques in this subject[1].

2 Background

2.1 The environment and the agent

We will delve into the Markov Decision Processes (MDPs), a problem that involves evaluating the feedback from an environment based on the state we are in and the action we are going to take.

In this context, we will adopt the following notation: at a given time t, given a state S_t and an action A_t taken, the environment reacts by providing a reward R_{t+1} and by modifying the state in which our agent is located (Figure 1).

To formalize mathematically, we define the dynamics of the MDP as the probability of reaching state s' at time $t+1$ with a reward r, given that the agent was in state s at time t and took action A_t. This notation tells us that state S_{t+1} depends solely on S_t and A_t.

$$p(s', r|s, a) = \mathbb{P}\left(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\right) \tag{1}$$

2.2 Episodes and Rewards

So far, we have discussed our intention to maximize the reward for our agent, but we have not formally defined it. By considering the rewards received after time t, such as $R_{t+1}, R_{t+2}, \ldots, R_T$, we can seek to maximize the sum of these rewards.

$$G_t = R_{t+1} + \cdots + R_T \tag{2}$$

This definition works well when there is a well-defined final time T. In other words, when the interaction between the agent and the environment naturally

[1] All code used here is accessible on github

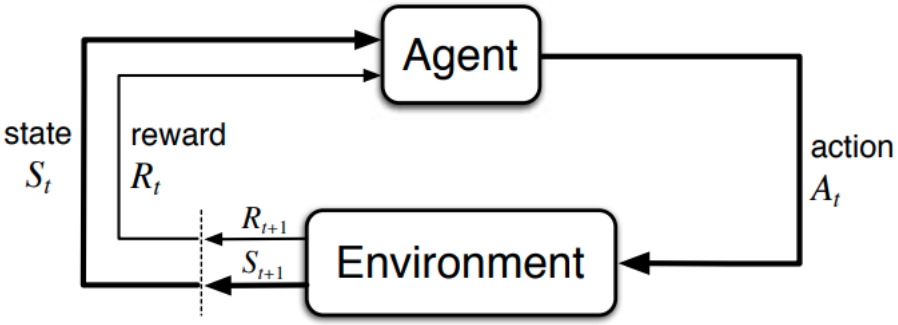

Figure 1: Caption

breaks down into distinct sequences, which we refer to as episodes, such as sessions of a game.

However, in the context of a continuous process lacking a precise notion of a final moment, such as a robot's walk or a control task, the final time T is infinite (∞), and the series $\sum_{\tau=1}^{\infty} R_{t+\tau}$ does not systematically converge (for example, imagine that our agent receives a reward $R_{t+\tau} = r$ for every $\tau \in \mathbb{N}$). Thus, we will introduce the concept of a discount factor $\gamma \in (0, 1)$, indicating our intention to now maximize a sum weighted by γ (equation 3).

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{\tau=1}^{\infty} \gamma^{\tau-1} R_{t+\tau} \tag{3}$$

In this context, the sum G_t converges when the rewards are bounded, which is typically the case. In practical applications, a simplified form of equation 3 is often used, as follows:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{4}$$

2.3 Policies and Value Functions

Value functions are defined as functions that estimate the reward of a given state after taking an action $A_t = a$.

Formally, a policy (also denoted as π) can be defined as a function that assigns the probability that an agent chooses action $A_t = a$ when it is in state $S_t = s$ (in this case, denoted as $\pi(a|s)$). For instance, considering a policy π and a state $S_t = s$, the expected reward can be expressed using the definition of p provided in equation 1.

$$\mathbb{E}_\pi[R_{t+1}|S_t = s] = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a) \tag{5}$$

The value function of a state s under a policy π, denoted as $v_\pi(s)$, represents the expectation of G_t when we are in the state $S_t = s$ and follow the policy π.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{\tau=1}^{\infty} \gamma^{\tau-1} R_{t+\tau} \middle| S_t = s\right] \end{aligned} \tag{6}$$

Let's also define the function $q_\pi(s, a)$ as the expectation of the reward given that we are in the state $S_t = s$ and choose the action $A_t = a$, following the policy π .

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}\left[\sum_{\tau=1}^{\infty} \gamma^{\tau-1} R_{t+\tau} \middle| S_t = s, A_t = a\right] \end{aligned} \tag{7}$$

We can also express v_π in terms of π and q_π:

$$v_\pi(s) = \sum_a \pi(s|a) q_\pi(s, a) \tag{8}$$

In practical contexts, the functions v_π and q_π are typically estimated through experience. In other words, solving a reinforcement learning problem means finding an optimal policy π_* defined as the policy that maximizes the value function v_π.

$$\forall \pi, v_{\pi_*}(s) \geq v_\pi(s) \tag{9}$$

We also define $v_* := v_{\pi_*}$ and q_* :

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s) \\ q_*(s, a) &= \max_\pi q_\pi(s, a) \\ &= \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \end{aligned} \tag{10}$$

2.4 Dynamic Programming

Let's first examine equation 7. If we utilize equation 5, we get:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[\sum_{\tau=1}^{\infty} \gamma^{\tau-1} R_{t+\tau} \middle| S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1}|S_t = s\right] \\ &= \sum_a \pi(s, a) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']] \\ &= \sum_a \pi(s, a) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{11}$$

Therefore, we observe that the value function $v_\pi(s)$ depends on the value function of adjacent states s'. This equation is known as the Bellman equation. An equivalent equation for q_π is provided in 12.

$$q_\pi(s,a) = \sum_r \sum_{s'} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(s'|a') q_\pi(s'|a') \right] \tag{12}$$

According to the Bellman equation, we can find the value function of an optimal policy by selecting the action a that maximizes v_π:

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_*(s')] \tag{13}$$

And for q_π:

$$q_* = \sum_r \sum_{s'} p(s', r|s, a) \left[r + \gamma \max_{a'} q_*(s'|a') \right] \tag{14}$$

Therefore, our optimal policy will be given by $\pi(s) = \text{argmax}_a q_*(s, a)$.

2.5 ε-Greedy Policy

The ε-greedy policy is a commonly used strategy in reinforcement learning, balancing exploration and exploitation. It involves selecting the optimal action with a high probability $(1-\varepsilon)$ based on current estimates, while a random action is chosen with probability ε. This policy allows for exploring new actions while exploiting those that appear to be the best. By gradually decreasing ε towards zero over time, the ε-greedy policy typically converges to an optimal policy.

2.6 Environment and Agent Data Structure

Given the mathematical modeling of the reinforcement learning problem, it is necessary to represent the entities in code, namely the agent and the environment.

The agent observes the environment and takes actions according to an ϵ-greedy strategy. With each action, the environment evolves and provides a reward to the agent, contributing to the learning of optimal choices. The `Agent` class thus provides two main methods: `act` and `learn`.

The `act` function takes the current state of the environment where our agent is located as a parameter and returns the action chosen by the agent.

The `learn` function accepts parameters for the state of the environment before and after the action, as well as the reward S_{t+1}, S_t, A_t, R_{t+1}.

The environment has a main method: `step`, which takes the action A_t as input and computes the next state S_{t+1}, the reward R_{t+1} if the game is over, and other information such as the overall score.

In general, a reinforcement learning agent training step is given by the code below.

Algorithm 1: Train a reinforcement learning agent.

Initialize an estimator $\hat{Q}$ for Q (either a function or a table).
for *episode* **do**
    Initialize the environment with the initial state s_0.
    while $True$ **do**
        Select a random action a_t with a probability of ε.
        Select $a_t = \max_a \hat{Q}(s, a)$ with a probability of $1 - \varepsilon$.
        Execute action a_t, observe the reward r_{t+1}, and the next state s_{t+1}.
        Learn from the data $\{s_t, a_t, r_t, s_{t+1}\}$.
        $s_t \leftarrow s_{t+1}$
        if s_t *is terminal* **then**
            Save the reward
            break
        end
    end
end

3 Studied Approaches

3.1 Monte Carlo

In the Monte Carlo method, we perform a kind of average of returns. Given an episode i, we visit all states $S_0, A_0, R_1, S_1, A_1, \ldots, S_{T-1}, A_{T-1}, R_T$ in that episode. Therefore, we estimate G as a weighted sum of R_t.

Algorithm 2: Monte Carlo Algorithm for Reinforcement Learning

Initialize an arbitrary policy $\pi(s)$.
Initialize $Q(s, a)$ randomly for all $(s, a) \in \mathcal{S} \times \mathcal{A}$.
Initialize $Returns(s, a)$ as an empty list for all $(s, a) \in \mathcal{S} \times \mathcal{A}$.
for *episode* **do**
    Generate the states $S_0, A_0, R_1, S_1, A_1, \ldots, S_{T-1}, A_{T-1}, R_T$.
    $G \leftarrow 0$
    for $t = T - 1, T - 2, \ldots, T_0$ **do**
        $G \leftarrow \gamma G + R_t$
        Add G to $Returns(S_t, A_t)$.
        $Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$
        $\pi(S_t) = \arg\max_a Q(S_t, a)$
    end
end

There are two major drawbacks in Monte Carlo methods, which will be addressed by the upcoming methods:

- Generating complete episodes is necessary, which can be time and resource-intensive, especially in environments where episodes are long or the cost of interacting with the environment is high.
- Monte Carlo is generally less efficient for problems with an infinite horizon or for environments where convergence is slow because it updates values only at the end of each episode.

3.2 TD Learning

The Temporal Difference (TD) learning approach is one of the simplest reinforcement learning algorithms.

The algorithm aims to predict V, the value function v_π of a policy. After transitioning from state S_t to S_{t+1} and knowing the reward R_{t+1}, the algorithm updates the function V such that:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{15}$$

To assess the effectiveness of our algorithm, we often refer to the temporal difference error (**TD error** or δ_t), defined as:

$$\begin{aligned} \delta_t &:= R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \\ &= R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \end{aligned} \tag{16}$$

The temporal difference error becomes particularly significant in approximate methods, where optimizations are performed using gradient calculations.

3.3 SARSA

SARSA, an approach in Temporal Difference (TD) learning, considers the states, actions, and rewards at each step $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$ (hence its name SARSA). This method iteratively updates $Q(s_t, a)$ with these elements according to the equation 17.

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1} - Q(s_t, a_t)\right) \tag{17}$$

The convergence of SARSA towards the optimal policy ($Q(s, a) \to Q^*(s, a)$) is guaranteed when each pair (s, a) is visited an infinite number of times. This can be ensured by choosing a value of ε_t such that $\sum_t \varepsilon_t = +\infty$ [SB18].

3.4 Q-Learning

When our environment is discrete, Q-Learning is an algorithm that stores information about the most recent rewards in a state-action pair to predict the reward $Q(s_t, a)$. Rewards are updated at each iteration according to equation 18.

$$Q^{\text{new}}(s_t, a) \leftarrow Q(s_t, a) + \alpha \left(r_t - Q(s_t, a) + \gamma \max_a Q(s_{t+1}, a) \right) \tag{18}$$

The convergence towards the optimal function $Q_i(s, a) \rightarrow Q^*(s, a)$ is also guaranteed if we visit all state-action pairs (s, a) infinitely often [SB18].

SARSA is often characterized as an "on-policy" algorithm, meaning it uses the current policy for action selection. In contrast, Q-learning relies on a "greedy" policy for action selection and is considered "off-policy." SARSA is more cautious and promotes a more thorough exploration of the environment by following a given policy. On the other hand, Q-learning may be more effective in determining the optimal policy but can also be more volatile due to its strong reliance on a "greedy" policy.

3.5 Q Networks

In the Q-Learning and SARSA approaches, our agent aimed to predict the function Q using a table of past experiences. In practice, this basic approach is entirely inefficient because the action's value function is estimated separately for each sequence without any generalization. Instead, it is common to use a function approximation to estimate the action's value function, $Q(s, a, \theta) \approx Q^*(s, a)$. Typically, in the context of reinforcement learning, these functions are either linear or nonlinear, represented by neural networks (referred to as Q-networks) [MKS+13].

In essence, we choose a loss function suitable for the problem and attempt to minimize the temporal difference error (equation 16) using an optimization algorithm (gradient descent, ADAM, etc.).

4 Studied problems

4.1 Blackjack

Blackjack is a well-known casino game. Its objective is to beat the dealer by aiming for a score of 21 points.

The game starts with the dealer having one card face up and one card face down, while the player has two cards face up (see figure 2).

The player can request additional cards (hit) until deciding to stop (stick) or exceeding 21 (bust, immediate loss). Once the player stops, the dealer reveals their face-down card and draws cards until their total is 17 or more. If the dealer exceeds 21, the player wins. If neither the player nor the dealer exceeds 21, the result (win, lose, tie) is determined by whoever is closest to a total of 21.

In blackjack, a usable Ace (or "soft Ace" in English) refers to an Ace that can be counted as having a value of 11 without exceeding 21, allowing flexibility in the hand's total. When a player has an Ace that can be counted as 11 without exceeding 21, it provides a strategic advantage as the player can adjust the Ace's value based on other cards in hand to achieve the most favorable total, offering

flexibility to avoid exceeding 21. This also enhances the player's chances of achieving a blackjack (a hand with a value of 21 with only two cards, an Ace, and a 10-value card).

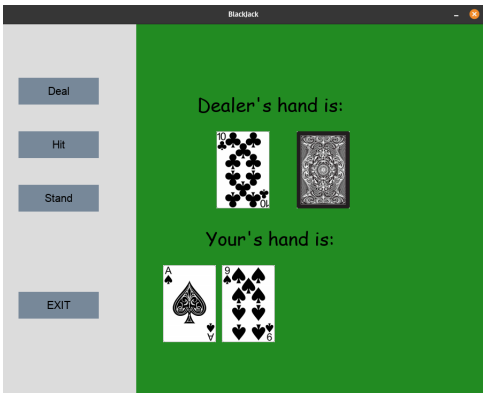

Figure 2: Illustration of a blackjack game

Observation	Domaine
Sum of the cards in the player's hand	$\mathbb{N}$
Card exposed by the dealer	$\{1, \ldots, 10\}$
Usable Ace	$\{0, 1\}$

Table 1: Observation space of the blackjack problem

The objective here is to demonstrate the application of simpler reinforcement learning algorithms: we will compare the Monte Carlo methods with Q-Learning and SARSA.

In the appendix (see 6.1), we can observe the estimated value functions ($V(s)$) as well as the optimal actions for each state. By examining Figure 3, we find that, as expected due to its exploration, the Q-Learning method proved to be the best among the three methods studied.

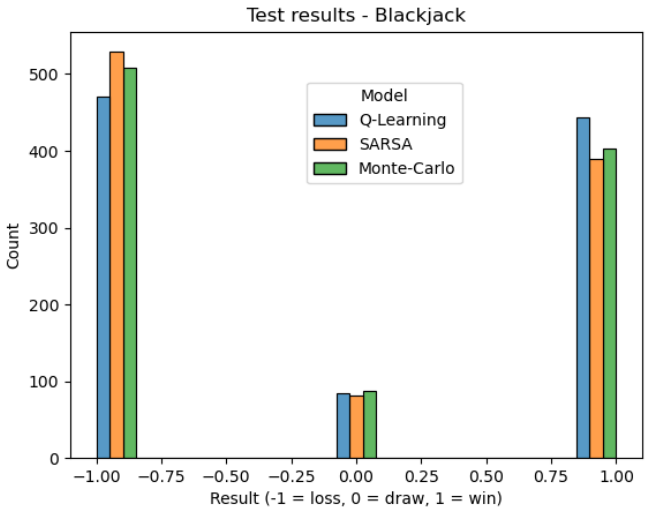

Figure 3: Results of blackjack tests using three distinct methods examined. The x axis indicates whether the player won, tied, or lost.

4.2 Flappy Bird

Sure, here is the rewritten text in LaTeX format:

Here, we will discuss the well-known video game Flappy Bird. The objective of the game is to prevent the bird from colliding with pipes by making it jump. Each successfully passed pipe earns the player one point.

The environment modeling is already available in various independent projects. For this work, the Text Flappy Bird library was chosen, available here. Our state vector had a size of 2, representing the distance in x and y relative to the center of the gap between pipes, while the possible actions were to do nothing or to jump.

Observation	Domain
Δx	$\mathbb{N}$
Δy	$\mathbb{N}$

Table 2: Observation space of Flappy Bird

The rewards provided by the environment are $R_{t+1} = 1$ during the time the bird is still alive. The maximum score is 100 (to prevent the bird from staying alive indefinitely, which occurs with the DQN method).

The training results of the Q-Learning and SARSA techniques are illustrated in Figure 4. It can be observed that Q-Learning outperforms SARSA. This is

attributed to the update strategy of the Q function. Q-Learning uses an off-policy update strategy, meaning it selects its actions based on the best expected reward for the next state, even if it follows a different policy to explore the environment. On the other hand, SARSA follows an on-policy update strategy, where actions are chosen according to the current policy. This difference allows Q-Learning to explore more efficiently and converge more quickly to an optimal solution in certain situations, particularly when exploration is more challenging or when non-optimal policies are explored.

Furthermore, the Q values for each of the methods are available in the appendix (see 6.2).

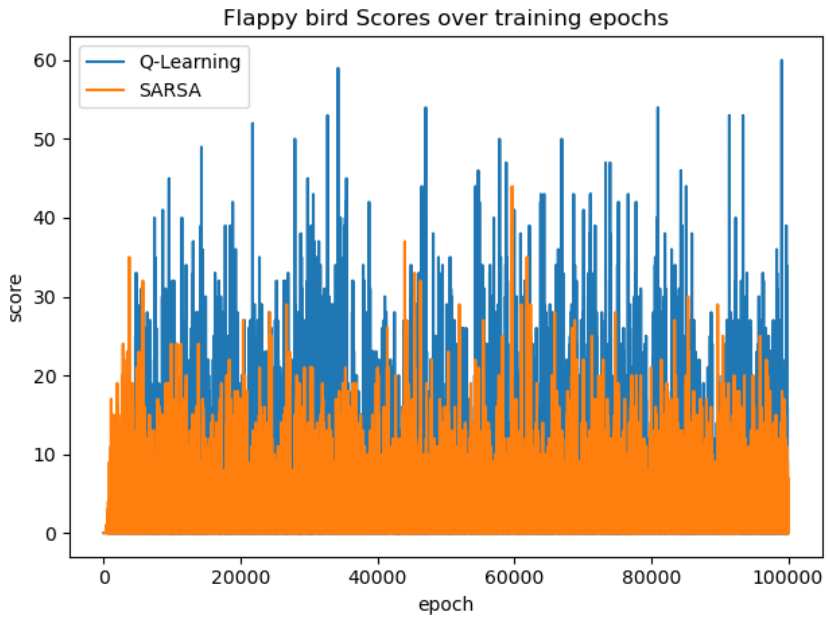

Figure 4: Performance of SARSA and Q-Learning with Flappy Bird. The ordinate axis (y) represents the number of pipes the bird successfully passed before dying.

The figure 5 presents the training results with the DQN method. Despite using a complex model (around 17,000 parameters), a rapid convergence was anticipated, requiring only 200 training epochs. The policy was optimal between epochs 110 and 160. Peaks in the graph (e.g., at epoch 125) illustrate moments when the agent poorly explored. However, after epoch 175, the model exhibits overfitting, capturing the noise in the training data and compromising its generalization. To avoid this overfitting, we will save the best-performing model and use it during the testing phase.

For the test evaluation, we will set $\varepsilon = 0$ (no exploration). Here, we do not want our model to learn from the test data, but rather, we seek to evaluate

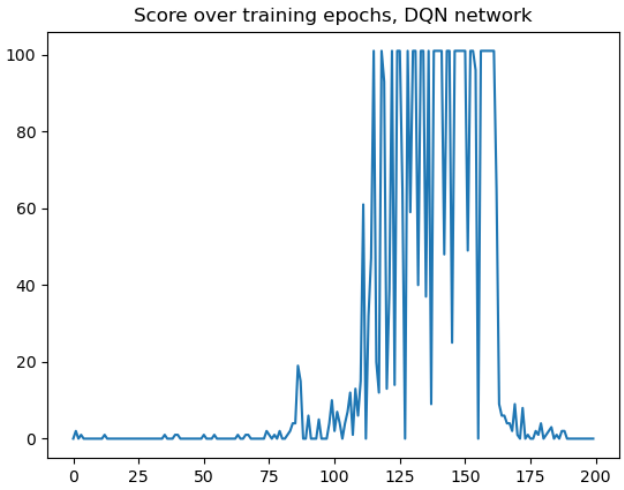

Figure 5: Performance of DQN learning with Flappy Bird. The ordinate axis (y) represents the number of pipes the bird successfully passed before dying.

its ability to generalize the knowledge acquired during the training steps. See figure 6.

Interpreting the results, we note that DQN performed perfectly in the test: it succeeded in all epochs with a maximum score.

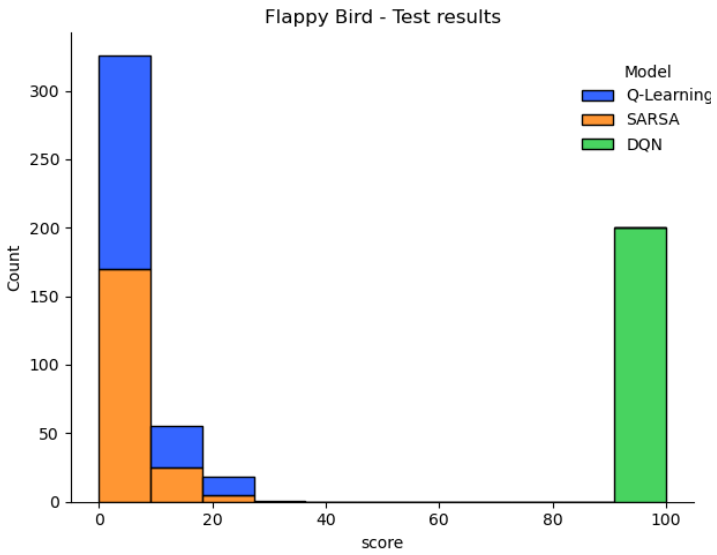

Figure 6: Results of the test phases for the Flappy Bird challenge.

4.3 Lunar Lander

In this game, we will study the use of rewards to guide our agent in its learning. The agent must pilot a rocket attempting to land on the moon. Our goal is to achieve a successful landing between two yellow flags as quickly as possible while avoiding the destruction of the rocket (figure 7).

In this context, the state of the rocket is represented by eight variables: the coordinates (x, y), the linear velocities (v_x, v_y), the angle θ, and the angular velocity $\dot{\theta}$ of the rocket with respect to the y-axis, as well as two booleans indicating whether the rocket has landed or not. After observing the environment, the agent will choose one of the four possible actions: do nothing, fly to the right, fly to the left, or fly upwards.s

Here we have a more complex problem with many more variables, so the environment must provide more structured rewards:

- In case of rocket destruction, $R_{t+1} = -100$
- Landing yields $R_{t+1} = 200$
- Moving the rocket upwards, left, or right results in $R_{t+1} = -0.3$ (the goal is to land as quickly as possible)
- If the rocket lands between the yellow flags, $R_{t+1} = 200$

We can observe the training results in Figure 8.

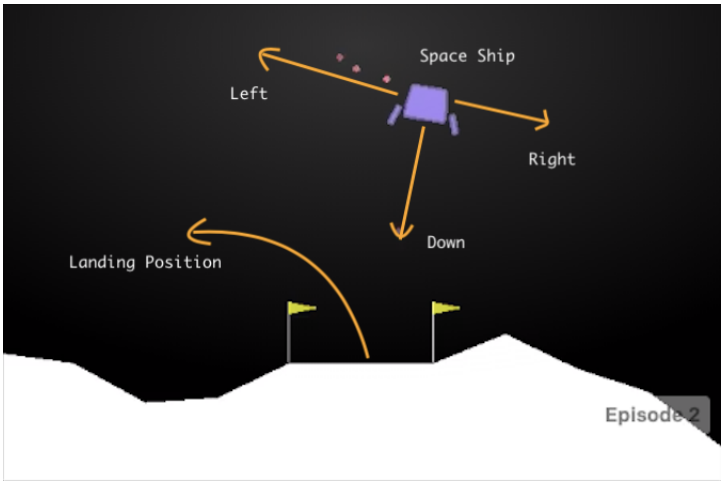

Figure 7: Illustration of the Lunar Lander problem. The goal is to land the rocket (in purple) between the two flags without incurring damage as quickly as possible.

Observation	Domain
x	$[-1.5, 1.5]$
y	$[-1.5, 1.5]$
v_x	$[-5, 5]$
v_y	$[-5, 5]$
θ	$[-\pi, \pi]$
$\dot{\theta}$	$[-5, 5]$
Si la jambe gauche est en contact avec le sol	$\{0, 1\}$
Si la jambe droite est en contact avec le sol	$\{0, 1\}$

Table 3: Espace des observations du problème du Lunar lander

After 500 training episodes, the agent managed to achieve 247 reward points during the test, demonstrating its success in landing effectively (here is a video showing it).

Notice that the convergence is significantly disrupted, which can be attributed to various reasons:

- The use of the ε-greedy policy makes our agent less stable.
- The dimension of the neural network leads to learning noise in the data (overfitting).

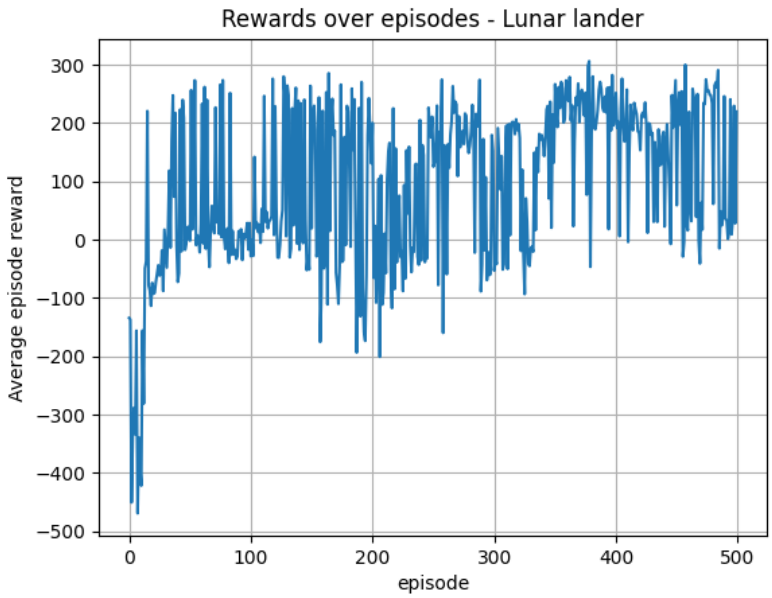

Figure 8: Rewards per episode on the lunar lander

5 Conclusion

Reinforcement learning is a complex and extensive field that combines the realms of mathematics and computer science. The convergence and instability resulting from the choice of hyperparameters are crucial aspects to consider, given the variability of convergence across different contexts. Therefore, understanding how to model these problems becomes essential to fully leverage this approach.

This study has elucidated the fundamental methods of reinforcement learning. By addressing three distinct problems, each challenge was approached with specific learning objectives.

Beyond the satisfactory interpretation of the obtained results, this study has demonstrated the effectiveness of reinforcement learning methods in optimally solving real-world problems in applied mathematics. This underscores the importance and relevance of this approach in various application domains.

6 Appendix

6.1 Results of each approach in the blackjack problem.

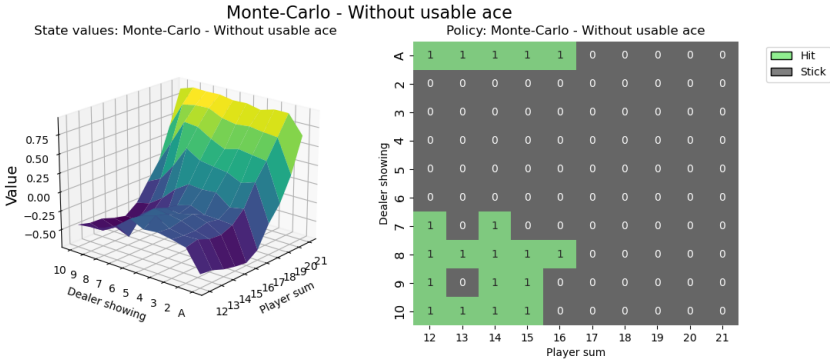

Figure 9: Left: Value function $V(s)$ for each state. Right: Optimal actions $\arg\max_a Q(s,a)$ for the same states, generated using the **Monte Carlo** method **without the possibility to use Ace as 11.**

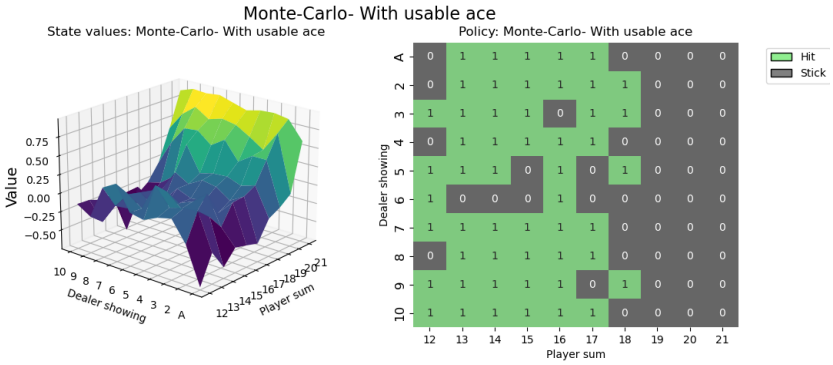

Figure 10: Left: Value function $V(s)$ for each state. Right: Optimal actions $\arg\max_a Q(s,a)$ for the same states, generated using the **Monte Carlo** method **with the possibility to use Ace as 11.**

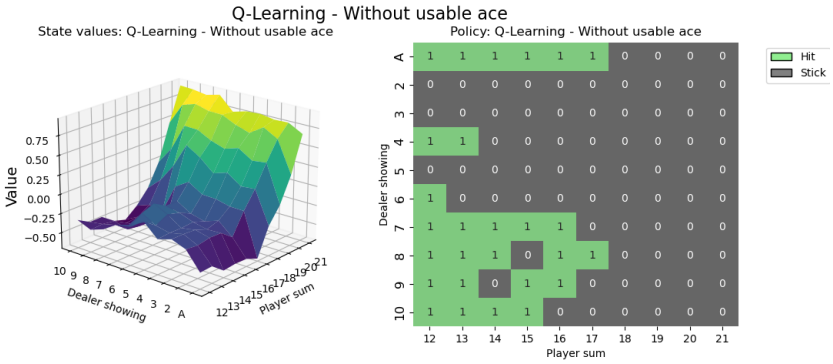

Figure 11: Left: Value function $V(s)$ for each state. Right: Optimal actions $\arg\max_a Q(s,a)$ for the same states, generated using the **Q-Learning** method **without the possibility to use Ace as 11.**

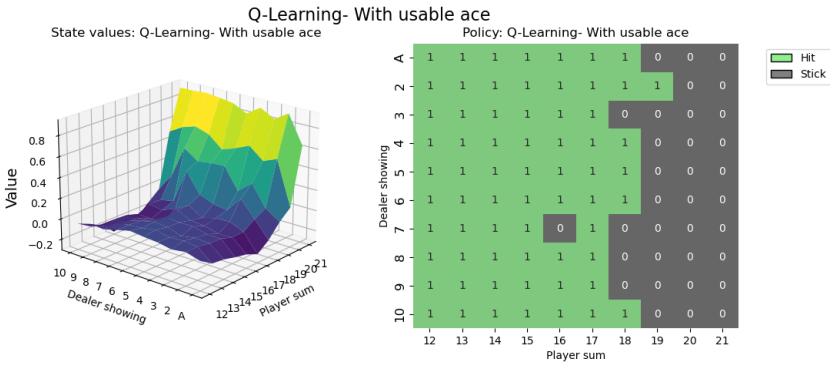

Figure 12: Left: Value function $V(s)$ for each state. Right: Optimal actions $\arg\max_a Q(s,a)$ for the same states, generated using the **Q-Learning** method **without the possibility to use Ace as 11.**

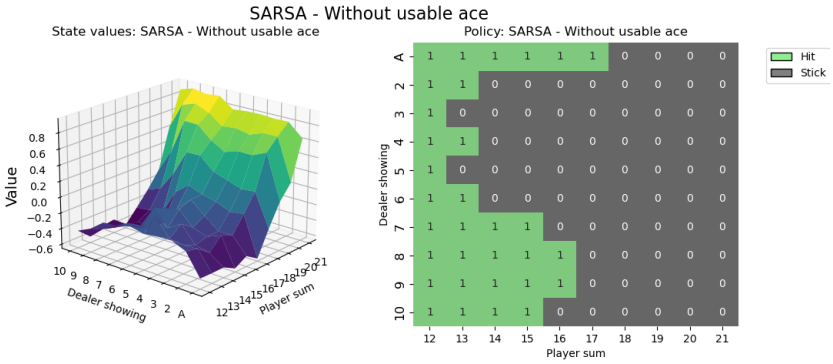

Figure 13: Left: Value function $V(s)$ for each state. Right: Optimal actions $\arg\max_a Q(s,a)$ for the same states, generated using the **SARSA** method **without the possibility to use Ace as 11.**

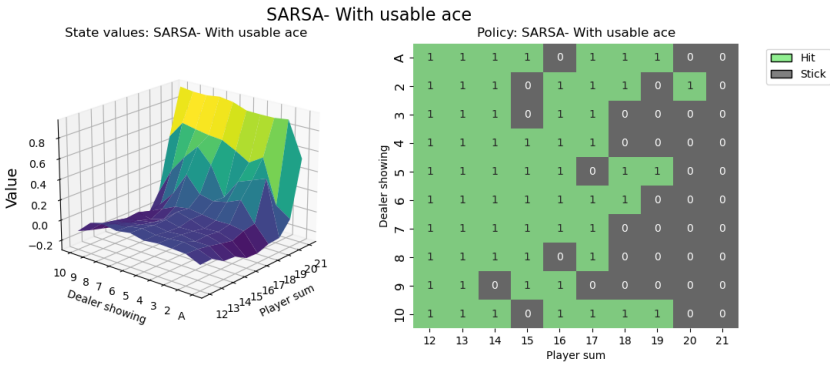

Figure 14: Left: Value function $V(s)$ for each state. Right: Optimal actions $\arg\max_a Q(s,a)$ for the same states, generated using the **SARSA** method **without the possibility to use Ace as 11.**

6.2 Optimal actions for Flappy Bird

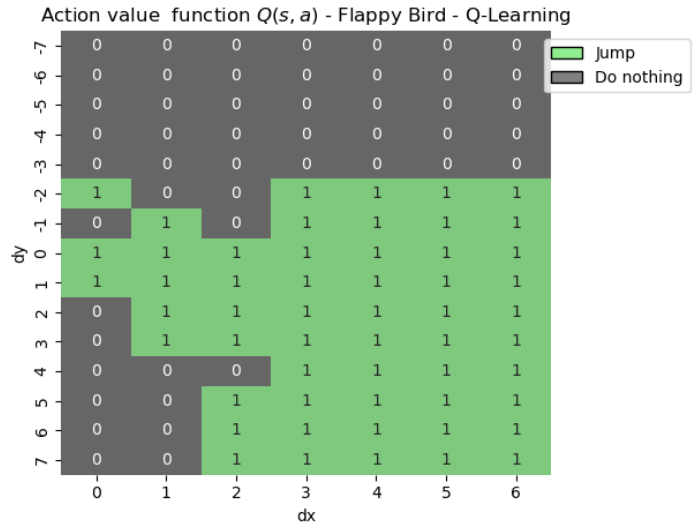

Figure 15: Optimal actions for each state $s = (\Delta x, \Delta y)$ given by the action-value function $\arg\max_a Q(s, a)$ from **Q-Learning**.

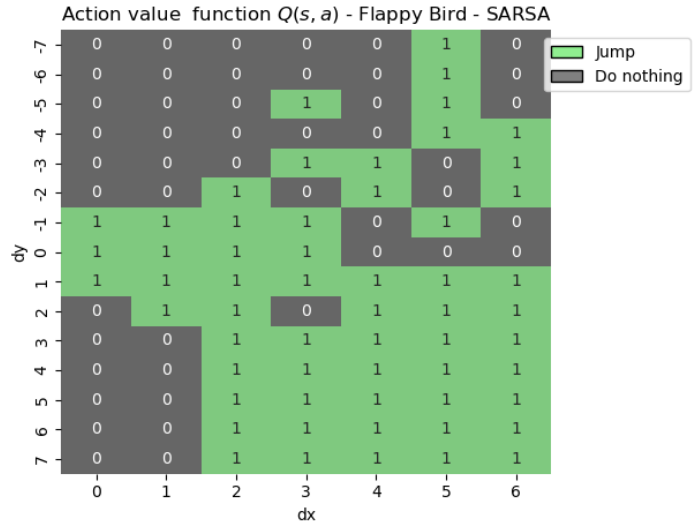

Figure 16: Optimal actions for each state $s = (\Delta x, \Delta y)$ given by the action-value function $\arg\max_a Q(s, a)$ from **SARSA**.

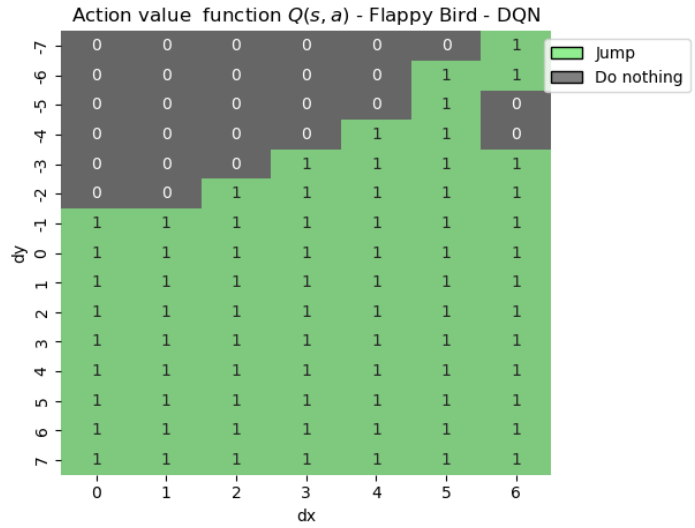

Figure 17: Optimal actions for each state $s = (\Delta x, \Delta y)$ given by the action-value function $\arg\max_a Q(s, a)$ from **DQN**.

References

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. cite arxiv:1606.01540.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[Sze10] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan Claypool Publishers, 2010.