Laboratory of Reinforcement Learning

Lab1: Q-Learning and SARSA Algorithm for Taxi-V3 environment .

Author:

- **AMMAR KHODJA Rayane**

Supervised by:

- **TABIA Hedi**

Master MMVAI December 2024.
Paris Saclay University.

Laboratoire de Recherche Paris Saclay University, IBISC Evry Val d'Essonne.

Abstract

Reinforcement learning algorithms are important for training agents to make good decisions in changing environments. This study compares two popular reinforcement learning methods - Q-Learning and SARSA - on a taxi navigation challenge.

In this challenge, called Taxi-V3, the agent has to move around a city to pick up passengers and drop them off efficiently. Q-Learning and SARSA work by estimating future rewards for different actions. But they update these reward estimates differently.

This study looks at how well each algorithm does on the taxi challenge. We train them in Taxi-V3 and test factors like: learning rate to adjust new knowledge, discount factor to value future rewards, and the tradeoff between exploration and exploitation.

We compare performance metrics like: average reward over time, how quickly they learn an effective strategy, and overall efficiency in the task. The results show Q-Learning and SARSA exhibit distinct learning patterns and behaviors on the taxi navigation problem.

In summary, this research provides useful insights on when to choose Q-Learning versus SARSA for real-world-inspired decision making scenarios. Comparing them on the Taxi-V3 benchmark sheds light on their adaptability to complex environments and informs selection based on the demands of a particular application.

The goal is to better understand the strengths and weaknesses of these core reinforcement learning techniques on problems that involveplanning sequences of actions to efficiently achieve goals in a dynamic world.

1 Introduction

Reinforcement learning (RL) agents must balance exploitation of known rewards with exploration to find greater rewards. This explores SARSA and Q-learning, two core RL algorithms with distinct approaches, in the cliff walking game—where an agent navigates a grid to a goal while avoiding falling off cliffs. SARSA learns from the current policy, while Q-learning learns from all possible actions.

This lab compares SARSA and Q-learning on key metrics: steps taken, rewards gained, and unsafe actions. We systematically vary learning rates and reward timescales to probe under which conditions each algorithm succeeds. Through these experiments spanning training episodes and parameterized configurations, we assess the strengths and weaknesses of each algorithm for goal-based navigation with irreversible penalties.

The cliff walking game represents a class of applications involving sparse rewards, large penalties, and exploration-exploitation tradeoffs. Our findings provide practical guidance for selecting suitable RL algorithms in these domains. Analyzing SARSA and Q-Learning empirical performance on cliff walking also yields theoretical insights into on-policy vs. off-policy learning. This lab thus contributes actionable evidence and new perspectives to longstanding debates contrasting core RL techniques.

2 Implementation

Before coding, clearly define:

- Environment layout (grid size, cliffs, start/goal locations)
- Reward scheme
- State representation
- Action set (up, down, left, right)
- Hyperparameters (learning rate, discount factor, epsilon decay)
- Training parameters (episodes, max steps per episode)
- Evaluation metrics (rewards, steps taken, convergence rate)

Additionally, we will have dataset logistics established for recording metrics during training, model checkpoints, performance graphs, etc.

3 Background

3.1 Reinforcement Learning

Reinforcement learning is a type of machine learning where an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or punishments, allowing it to learn optimal strategies for maximizing cumulative rewards over time.

3.2 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that aims to learn the optimal action-value function $Q^*(s, a)$, representing the expected cumulative reward of taking action a in state s and following the optimal policy thereafter.

The Q-learning update rule is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [R + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$$

where α is the learning rate, γ is the discount factor, R is the immediate reward, and s' is the next state.

3.3 SARSA

SARSA (State-Action-Reward-State-Action) updates its Q-values based on the policy it is currently following. The SARSA learning rule is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [R + \gamma \cdot Q(s',a') - Q(s,a)]$$

where:

- α is the learning rate,
- γ is the discount factor,
- R is the immediate reward,
- s' is the next state, and
- a' is the next action taken in the next state.

SARSA updates its Q-values by considering the action actually taken in the next state, distinguishing it from Q-learning.

4 Into coding

We will need to import the necessary environment and import the necessary libraries as follows:

```
import sys, os
if 'google.colab' in sys.modules and not os.path.exists('.setup_complete'):
    !wget -q https://raw.githubusercontent.com/yandexdataschool/Practical_RL/master/setup_colab
    .sh -O- | bash

    !touch .setup_complete

#This code generates a virtual display for rendering game images. It remains inactive
if your machine is equipped with a monitor.
if type(os.environ.get("DISPLAY")) is not str or len(os.environ.get("DISPLAY")) == 0:
    !bash ../xvfb start
    os.environ['DISPLAY'] = ':1'
```

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
!pip3 install -q gymnasium[classic-control]
```

```
 from collections import defaultdict
import random
import math
import numpy as np
```

Now, we need to implement our environment which is Taxi-V3:

```
 import gymnasium as gym
env = gym.make("Taxi-v3", render_mode='rgb_array')

n_actions = env.action_space.n
s, _ = env.reset()
plt.imshow(env.render())
```

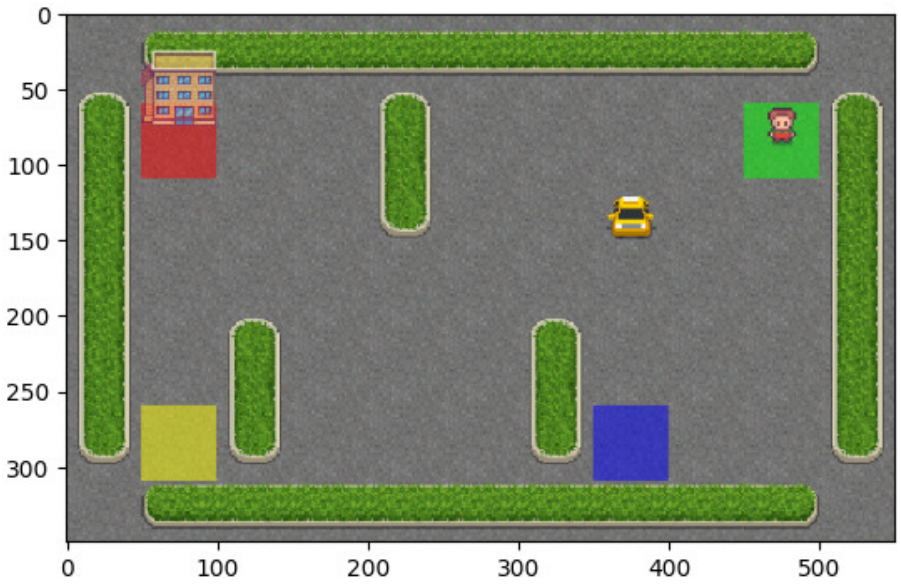

Figure 1: The environment of our example

Taxi-V3 Environment Overview:

The Taxi-V3 environment simulates a taxi navigating a grid to pick up and drop off passengers. Key features include:

- **Gridworld Layout:** A grid with walls, representing obstacles.
- **Taxi:** The agent that moves in four directions: north, south, east, and west.
- **Passenger and Destination:** Randomly placed in the grid; taxi's goal is to pick up and drop off the passenger.
- **Rewards and Penalties:** Positive reward for successful pick-up and drop-off, penalties for time steps without completion.
- **States:** Defined by taxi, passenger, and destination positions.
- **Actions:** Taxi can move in four directions or pick up/drop off the passenger.
- **Objective:** Learn a policy for efficient navigation to maximize cumulative reward.

4.1 Q-learning

Algorithm 1 Q-Learning Algorithm

0 -1 1

Initialize Q-values $Q(s, a)$ arbitrarily

for each episode **do** Initialize the starting state s

while episode is not done **do** Choose action a using an exploration strategy based on Q Take action a, observe next state s' and reward r Update Q-value for the current state-action pair using the Q-learning update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$$

Move to the next state s'

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

The Q-Learning class implements a Q-Learning agent for reinforcement learning. Here are the key functions:

1. **__init__**
   - Initializes the Q-Learning agent with essential parameters.
   - Parameters: `alpha` (learning rate), `epsilon` (exploration probability), `discount` (discount factor), `get_legal_actions` (function to get legal actions).

2. **get_qvalue**
   - Returns the Q-value for a given state-action pair.
   - Parameters: `state`, `action`.

3. **set_qvalue**
   - Sets the Q-value for a given state-action pair.
   - Parameters: `state`, `action`, `value`.

4. **get_value**
   - Calculates the agent's estimation of the state value using the current Q-values.
   - Parameters: `state`.

5. **update**

- Updates the Q-value for a state-action pair using the Q-Learning update rule.
- Parameters: `state`, `action`, `reward`, `next_state`.

6. get_best_action

- Computes the best action to take in a given state using current Q-values.
- Parameters: `state`.

7. get_action

- Determines the action to take in the current state, incorporating exploration using an epsilon-greedy policy.
- Parameters: `state`.

The Python code is:

```
class QLearningAgent:
    def __init__(self, alpha, epsilon, discount, get_legal_actions):

        self.get_legal_actions = get_legal_actions
        self._qvalues = defaultdict(lambda: defaultdict(lambda: 0))
        self.alpha = alpha
        self.epsilon = epsilon
        self.discount = discount

    def get_qvalue(self, state, action):
        """ Returns Q(state,action) """
        return self._qvalues[state][action]

    def set_qvalue(self, state, action, value):
        """ Sets the Qvalue for [state,action] to the given value """
        self._qvalues[state][action] = value

    #---------------------START OF YOUR CODE---------------------#

    def get_value(self, state):
        """
        Calculate your agent's estimation of V(s) using the current q-values:

        \[ V(s) = \max_{\text{over\_action}} Q(\text{state, action}) \]

        Note: Consider that q-values may be negative.
        """
        possible_actions = self.get_legal_actions(state)

        # If there are no legal actions, return 0.0
        if len(possible_actions) == 0:
            return 0.0

        # Calculate V(s) by taking the maximum Q-value over all possible actions
        max_q_value = max(self._qvalues[state][action] for action in possible_actions)

        return max_q_value
```

```
    def update(self, state, action, reward, next_state):
        """
        You should do your Q-Value update here:
           Q(s,a) := (1 - alpha) * Q(s,a) + alpha * (r + gamma * V(s'))
        """

        # agent parameters
        gamma = self.discount
        learning_rate = self.alpha

        current_q_value = self._qvalues[state][action]
        legal_actions = self.get_legal_actions(next_state)

        if not legal_actions:
            # If there are no legal actions in the next state, set the Q-value to the
            immediate reward
            new_q_value = reward
        else:
            # Update the Q-value using the Q-learning update rule
            max_next_q_value = max(self._qvalues[next_state][next_action] for
            next_action in legal_actions)
            new_q_value = current_q_value + self.alpha * (reward + self.discount *
            max_next_q_value - current_q_value)

        # Update the Q-value for the current state-action pair
        self._qvalues[state][action] = new_q_value

        self.set_qvalue(state, action, new_q_value)

    def get_best_action(self, state):
            """
            Compute the best action to take in a state (using current q-values).
            """
            possible_actions = self.get_legal_actions(state)

            # If there are no legal actions, return None
            if len(possible_actions) == 0:
                return None

            q_values = np.array([self.get_qvalue(state, action) for action in possible_actions])
            best_action = np.argmax(q_values)

            return best_action

    def get_action(self, state):

            # Pick Action
            possible_actions = self.get_legal_actions(state)
            action = None

            # If there are no legal actions, return None
            if len(possible_actions) == 0:
                return None

            # agent parameters:
            epsilon = self.epsilon
            random_number = random.uniform(0,1)
            if random_number < epsilon:
              chosen_action = random.choice(possible_actions)
            else:
              chosen_action = self.get_best_action(state)


            return chosen_action
```

4.2 Results of Q-learning

Now we need to create our agent from the Class:

```
agent = QLearningAgent(
    alpha=0.5, epsilon=0.25, discount=0.99,
    get_legal_actions=lambda s: range(n_actions))
```

Then we create our function for displaying the results:

```
def play_and_train(env, agent, t_max=10**4):
    """
    This function is designed to:

    - Execute a complete game, with actions determined by the agent's epsilon-greedy policy.
    - Train the agent using agent.update(...) whenever applicable.
    - Return the total reward obtained during the game.
    """
    total_reward = 0.0
    s, _ = env.reset()

    for t in range(t_max):
        # get agent to pick action given state s.
        a = agent.get_action(s)

        next_s, r, done, _, _ = env.step(a)

        # train (update) agent for state s
        agent.update(s, a, r, next_s)

        s = next_s
        total_reward += r
        env.render()  # Render the environment on the screen

        if done:
            break

    return total_reward
```

Function: `play_and_train`
The `play_and_train` function executes a complete game within the environment, with the agent's actions determined by an epsilon-greedy policy. Its main objectives are to train the agent using the Q-learning update rule and to return the total reward obtained during the game.

- **Inputs:** `env` (environment), `agent` (Q-learning agent), `t_max` (maximum time steps).
- **Outputs:** `total_reward` (total reward obtained during the game).
- **Process:**
    1. **Environment Reset:** Initial state `s` is obtained by resetting the environment.
    2. **Game Execution Loop:** Iterates for a maximum of `t_max` time steps.
    3. **Environment Interaction:** Agent selects action `a` based on epsilon-greedy policy, interacts with the environment, and obtains `next_s`, `r`, and `done` (episode completion).
    4. **Agent Update:** Q-values are updated using the Q-learning rule considering `s`, `a`, `r`, and `next_s`.
    5. **State Transition:** `s` is updated to `next_s`.
    6. **Reward Accumulation:** `r` is added to `total_reward`.
    7. **Environment Rendering:** The environment is rendered for visualization.

8. **Termination Check:** Loop continues until `t_max` or episode completion (`done`).
9. **Total Reward Return:** Returns `total_reward`.

Usage Example:

```
total_reward = play_and_train(env, agent)
```

This function facilitates game execution, Q-value updates, and provides insight into the agent's performance. **The performance**

```
from IPython.display import clear_output

rewards = []
for i in range(1000):
    rewards.append(play_and_train(env, agent))
    agent.epsilon *= 0.99

    if i % 100 == 0:
        clear_output(True)
        plt.title('eps = {:e}, mean reward = {:.1f}'.format(agent.epsilon,
        np.mean(rewards[-10:])))
        plt.plot(rewards)
        plt.show()
```

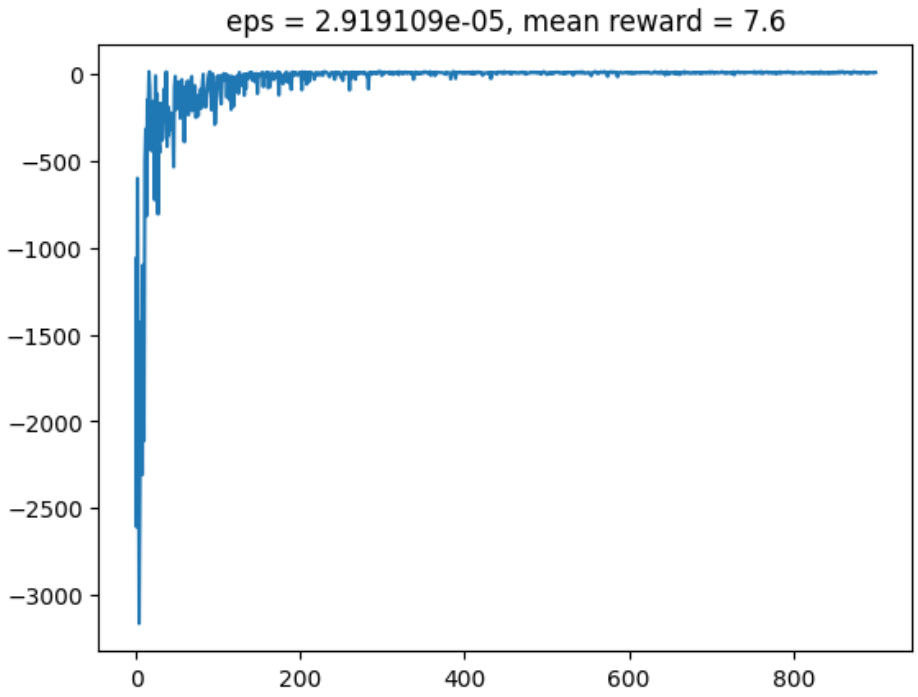

Figure 2: The results of our training with Q-learning Algorithm

4.3 SARSA Algorithm

Algorithm 2 SARSA Algorithm

0 -1 -2 -2 -1 0 Initialize Q-values arbitrarily: $Q(s, a)$ for all s in the state space and a in the action space episode Initialize state s Choose action a using an exploration strategy time step Take action a, observe reward r and next state s' Choose next action a' using the same exploration strategy Update Q-value for the current state-action pair:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma Q(s', a') - Q(s, a)\right)$$

Update s and a to be s' and a'

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

We changed the update function of Q-learning Agent to the following class:

```
    def update(self, state, action, reward, next_state):
        """
        SARSA Update:
           Q(s,a) := (1 - alpha) * Q(s,a) + alpha * (r + gamma * Q(s', a'))
        where a' is the action taken in state s'
        """
        # Next action, chosen using the same policy
        next_action = self.get_action(next_state)

        # SARSA update formula
        gamma = self.discount
        learning_rate = self.alpha

        q_value = (1 - learning_rate) * self.get_qvalue(state, action) + \
                  learning_rate * (reward + gamma * self.get_qvalue(next_state, next_action))

        self.set_qvalue(state, action, q_value)
```

4.4 Results of SARSA

Now we changed the play function to:

```
def play_and_train_sarsa(env, agent, t_max=10**4):
    total_reward = 0.0
    state, _ = env.reset()

    for t in range(t_max):
        action = agent.get_action(state)
        next_state, reward, done, _, _ = env.step(action)

        # SARSA update
        agent.update(state, action, reward, next_state)

        state = next_state
        total_reward += reward
        if done:
            break

    return total_reward
```

where the agent is:

```
agent = SARSAAgent(
    alpha=0.5, epsilon=0.25, discount=0.99,
    get_legal_actions=lambda s: range(n_actions))
```

And to do **the performance**:

```
rewards = []
for i in range(1000):
    rewards.append(play_and_train_sarsa(env, agent))
    agent.epsilon *= 0.99  #Epislon Decay

    if i % 100 == 0:
        clear_output(True)
        plt.title('eps = {:e}, mean reward = {:.1f}'.format(agent.epsilon,
        np.mean(rewards[-10:])))
        plt.plot(rewards)
        plt.show()
```

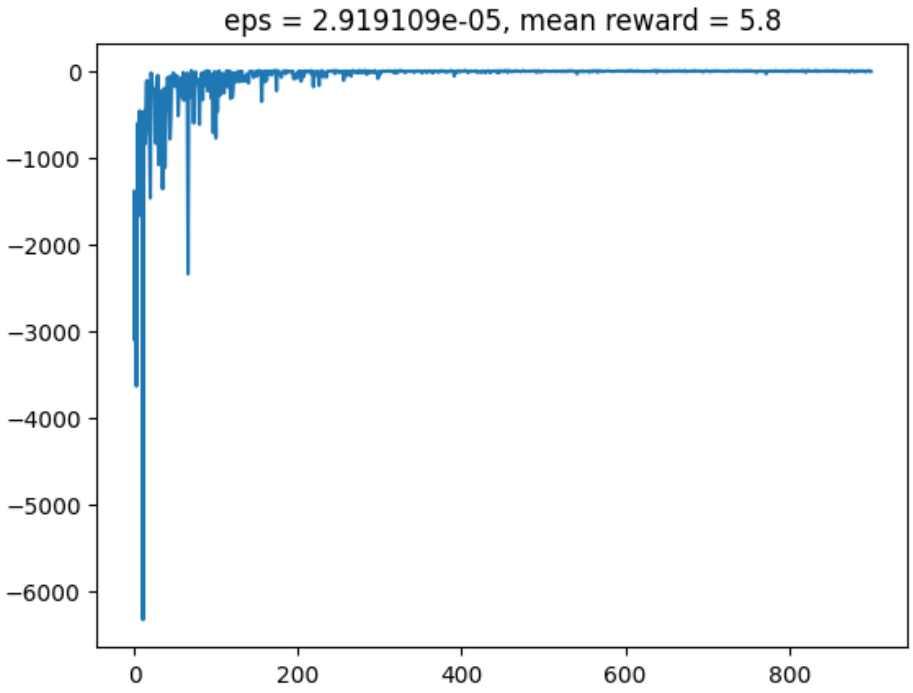

Figure 3: The results of our training with SARSA Algorithm

5 Discussion

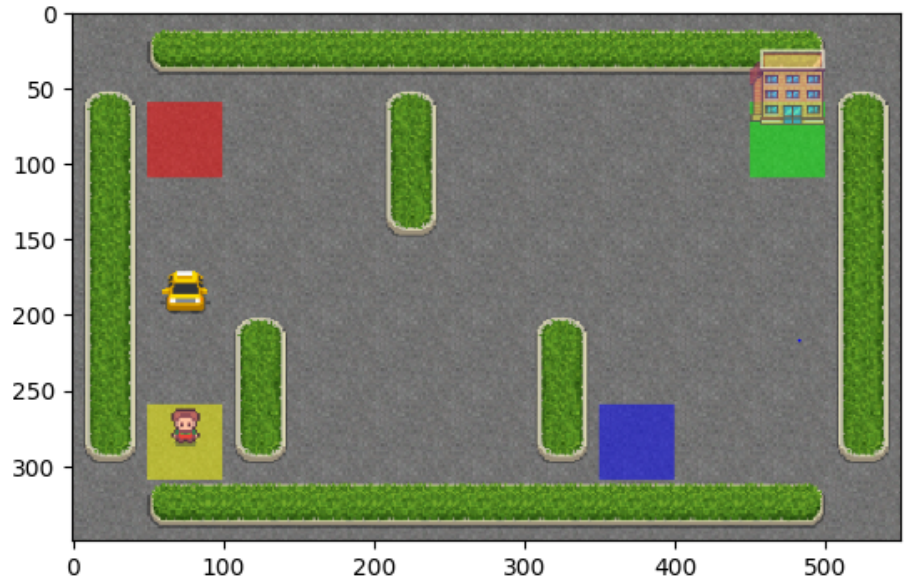

Figure 4: The results after both performances

The performance of SARSA and Q-learning was compared in terms of mean final value and convergence. SARSA demonstrated more cautious learning by considering the actual next action during updates, leading to a smoother convergence and often achieving a higher mean final value. Q-learning, on the other hand, exhibited a more explorative approach, occasionally resulting in larger fluctuations during training. While Q-learning might find optimal policies more quickly, SARSA tends to converge more steadily, ensuring a more reliable outcome. The choice between SARSA and Q-learning depends on the specific characteristics of the problem and the trade-off between exploration and exploitation desired in the learning process.

6 Generale conclusion

In the end, SARSA and Q-learning take different approaches to a common reinforcement learning challenge: balancing trying new things to get ahead faster with playing it safe as you go. SARSA is like the steady tortoise - it sticks closer to the path it knows, cautiously building on what has worked so far. This can pay off when there's a tricky terrain where straying too far off track gets you in trouble. SARSA keeps finding incremental gains without major mishaps. Meanwhile, Q-learning is more like the hare, eagerly bounding off to check out new territory to possibly find bigger rewards, even if it means risking the unknown. This exploration can lead Q-learning to race ahead quickly when there's no major danger that way. At their core, they have the same aim - maximizing rewards by balancing existing gains with new discoveries. Just different styles - the tortoise vs the hare!