Practice 3 Discrete Model-Free methods Monte-Carlo, TD, SARSA, Q-Learning

Learning Objectives

The objective of this Practice is implement several Model-free methods in the frozen-lake, taxi and cartpole environments.

You will learn that the loops are easily transferable between methods and how the agent learns on each method.

The algorithms are described in detail in [SB18]. Information about the environments can be found in [Far24b]. [Far24a], [Far25].

Introduction to Model-free algorithms

Model-free algorithms learn optimal policies or value functions directly from interactions with the environment, without requiring a model of the environment. A Model in the RL field means the knowledge on how the environment transitions from one state to the another, (i.e., the dynamics of the environment) and the rewards the agent will receive. Model-free methods do not try to learn this information explicitly. If we could define the main points that define a model free algorithm they would be:

Method	Model-Based / Model-Free	On-Policy / Off-Policy	Practice
Using Gymnasium - Russell's Grid	N/A	N/A	1
Dynamic Programming	Model-Based	N/A (requires a model)	2
Monte Carlo	Model-Free	On-Policy	3
Temporal Difference (TD)	Model-Free	On-Policy	3
SARSA and Q-Learning	Model-Free	On-Policy	3
DQN-Learning	Model-Free	Off-Policy	4

Table 1: Classification of RL Methods and Assignments/Practices

- **No model of the environment** : Do not need to understand or simulate how the environment works
- **Focus on Policy and Value Functions**. In policy-based methods they try to learn the policy, in value-based methods they try to learn the value function

- **On-policy or off-policy**: The agent learns the value of the policy currently used to make decisions (SARSA). Off-policy the agent learns the optimal policy independent of the action is taking (Q-learning)
- **Sample-based**: The agent learns by interacting with the environment.

Monte-Carlo methods

Monte-Carlo methods are a class of computational algorithms that rely on repeated random sampling to solve problems that might be deterministic. The basic concept is to use randomness to solve problems that might be deterministic by nature. The name comes from the Monte-Carlo Casino in Monaco where Stanislaw Ulam (a Polish physicist), who was the primary developer of the method, was inspired by his uncle who was a compulsive gambler in the Casino.

Monte-Carlo methods are often implemented using computer simulations, and they can provide approximate solutions to problems that are otherwise intractable or too complex to analyze mathematically, being this the case for the application to Reinforcement Learning.

1 Files for this Practice

```
    RL-SAMBD-Assignment3.pdf
```

2 Activities

The activities in this Practice are based in in the Taxi Environment [Far24b], in the Frozen Lake [Far25] and in the Cartpole [Far24a]

We will perform 3 activities in this practice.

- Monte-Carlo on Frozen Lake 4x4 and 8x8
- Monte-Carlo for Taxi
- SARSA for Taxi
- Q-Learning on Cartpole
- BONUS: Create 4 learning videos on Frozen Lake with MC, TD, SARSA and Q-Learning

2.1 Activity 1 - Monte-Carlo on Frozen Lake

The objective of this activity is to implement a Monte-Carlo algorithm to solve the Taxi environment. By completing this assignment, you will gain practical experience with Monte-Carlo methods in reinforcement learning and understand how to apply them to evaluate and improve policies.

I provide you with an example of Monte-Carlo for the FrozenLake environment that you can use as a starting example. Bear in mind that the Taxi environment has higher complexity (action space, observation and random start).

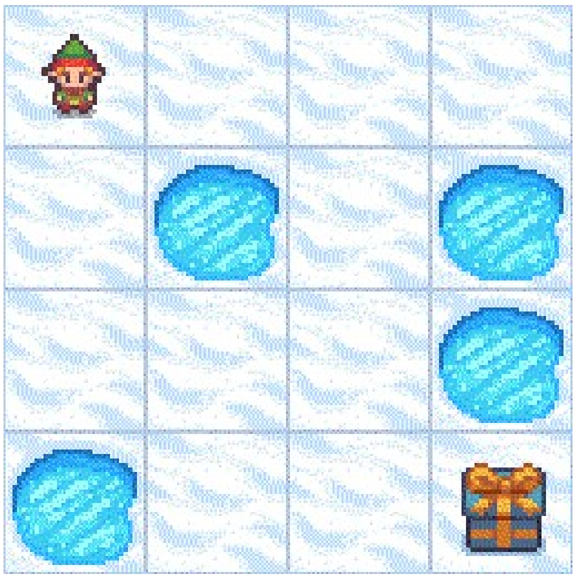

Figure 1: Frozen lake environment human rendering

```
https://github.com/castorgit/RL_course/blob/main/018_MC_Frozenlake.ipynb
```

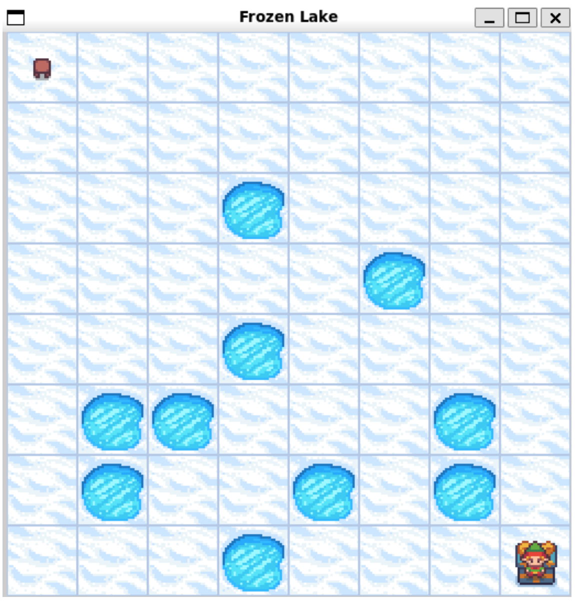

Figure 2: Frozen lake environment 8x8

One activity you can do here is to modify the environment. Change the environment call using the 8x8 map (is much bigger with 64 cells) you will see how convergence takes longer and simulate the result. ¿Do you see anything in the simulation?

```
env = gym.make('FrozenLake-v1', map_name="8x8", is_slippery=False)
```

2.1.1 The Monte-Carlo Algorithm

The Monte-Carlo algorithm is an effective and popular model-free reinforcement learning algorithm.

Monte-Carlo Prediction Algorithm Every Visit (Sutton)

```
Initialize:
    $\pi \leftarrow$ policy to be evaluated
    $V \leftarrow$ an arbitrary state-value function
    $Returns(s) \leftarrow$ an empty list, for all $s \in S$

Repeat Forever:
    Generate an episode using $\pi$
    for each state $s$ appearing in the episode:
        $G \leftarrow$ return following each occurrence of $s$
        Append $G$ to $Return(s)$
        $V(s) \leftarrow$ average $(Returns(s))$
```

Algorithm 1: Monte-Carlo Every Visit Algorithm (source Sutton)

Monte-Carlo (MC) algorithms in reinforcement learning estimate the value of states or state-action pairs by averaging returns observed from sample episodes. Unlike dynamic programming, MC methods do not require a model of the environment and can learn from raw experience. The key steps involve generating episodes using a policy, calculating the return for each state or state-action pair in the episode, and updating value estimates based on these returns. MC methods can be applied in both on-policy and off-policy settings. In on-policy MC, updates are made using returns from the current policy, while in off-policy MC, updates can use returns from different policies. The process iteratively improves the policy by making it greedy with respect to the current value estimates, aiming to converge to the optimal policy over time.

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

Where:

- $Q(s,a)$: Current Q-value for state s and action a.
- α: Learning rate (determines how much new information overrides the old).
- r: Reward received after taking action a in state s.
- γ: Discount factor (how much future rewards are considered).
- $\max_{a'} Q(s',a')$: Maximum Q-value for the next state s' across all possible actions a'.

Assignment steps
Set Up the Environment

- Initialize the environment
- Understand action space, observation space and reward structure of Taxi environment.

Implement Q-Learning Algorithm

- **Initialization**:
  - Create a Q-table with dimensions [number_of_states, number_of_actions] and initialize it to zeros.
  - Set the parameters: learning rate α, discount factor γ, exploration rate ϵ, and number of episodes N.
- **Training Loop**:
  - For each episode:
    1. Initialize the agent's starting state.
    2. While the state is not terminal:
       (a) Choose an action using the epsilon-greedy policy.
       (b) Execute the action, observe the reward, and the next state.
       (c) Update the Q-value using the Q-learning formula.
       (d) Transition to the next state.
  - Decrease the exploration rate ϵ over time to reduce exploration as learning progresses.
- **Evaluate the Policy**:
  - After training, derive the policy from the Q-table by choosing the action with the highest Q-value for each state.
  - Test the learned policy in the environment and measure performance metrics such as total reward and number of steps to reach the goal.
- **Visualize and Analyze**:
  - **Visualize the Q-Table**: Display the Q-values for each state-action pair to understand the learned values.
  - **Visualize the Policy**: Show the optimal policy on the grid, indicating the best action to take from each state.
  - **Analyze Performance**: Compare the performance of the learned policy to a baseline (e.g., random actions) and discuss observations on convergence and effectiveness.

2.2 Activity 2 - MC for Taxi

Background The Taxi environment is a classic reinforcement learning problem where the goal is to navigate a taxi to pick up and drop off passengers at specified locations within a grid world. The environment is defined by:

- **State space (S)**: The set of all possible positions of the taxi, passenger locations, and destination locations.
- **Action space (A)**: The set of possible moves (e.g., south, north, east, west, pick up, drop off).
- **Rewards (R)**: Values received after moving from one state to another.
- **Transitions (T)**: The probabilities of moving from one state to another given an action.

The objective of this assignment is to implement the MC algorithm to solve the Taxi environment in Gymnasium. You will gain practical experience with these reinforcement learning algorithms and understand their differences in learning optimal policies. **Assignment Instructions**

Set Up the Environment

- **Initialize the Taxi environment**

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

Define Monte-Carlo Components

- **Initialization**:
  - Create a Q-table with dimensions [number_of_states, number_of_actions] and initialise it to zeros.
  - Set the parameters: learning rate α, discount factor γ, exploration rate ϵ, and number of episodes N.
- **Training Loop**:
  - For each episode:
    1. Initialise the agent's starting state.
    2. Choose an action using the epsilon-greedy policy.
    3. While the state is not terminal:
       (a) Take action, observe reward r, and next state s'.
       (b) Choose next action a' using the epsilon-greedy policy.
       (c) Update the Q-value.
       (d) Set $s \leftarrow s'$ and $a \leftarrow a'$.

  - Decrease the exploration rate ϵ over time to reduce exploration as learning progresses.

Evaluate the Policies

- After training both algorithms, derive the policy from the Q-table by choosing the action with the highest Q-value for each state.
- Test the learned policies in the environment and measure performance metrics such as total reward and number of steps to reach the goal.

Visualize and Analyze

- **Visualize the Q-Table**: Display the Q-values for each state-action pair to understand the learned values.
- **Visualize the Policy**: Show the optimal policy on the grid, indicating the best action to take from each state.
- **Analyze Performance**: Review how the convergence happens. Test the epsilon, gamma, values to increase the convergence speed if necessary.

2.3 Activity 2 - A SARSA agent in the Taxi environment

The objective of this activity is to Implement a SARSA algorithm for the Taxi environment. Through this exercise, you will gain hands-on experience with SARSA, a popular reinforcement learning algorithm.

Later on we will transform this SARSA approach into a Q-Learning (off-policy) and later on we will apply it to the CartPole environment. All the algorithms are described in detail in [SB18].

SARSA: On-Policy TD Control

```
Initialize $Q(s,a), \forall s \in S, a \in A(s)$ arbitrarily and $Q(terminal\_state, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Repeat (for each step of episode):
        Take action A; observe reward, R and next state $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
        $S \leftarrow S'; A \leftarrow A'$
    until $S$ is terminal
```

SARSA Overview

SARSA is an on-policy reinforcement learning algorithm used to learn the value of actions in given states. The core idea is to update the Q-values (state-action values) based on the action actually taken by following the current policy.

You can see the algorithm described in class in the SARSA box.

The SARSA update rule is given by:

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

Where:

- $Q(s,a)$: Current Q-value for state s and action a.
- α: Learning rate (determines how much new information overrides the old).
- r: Reward received after taking action a in state s.
- γ: Discount factor (how much future rewards are considered).
- $Q(s',a')$: Q-value for the next state s' and the next action a'.

Pseudocodde

Set Up the Environment

- Define the Taxi environment
- Initialize the state space, action space, and reward structure.

Implement SARSA Algorithm

- **Initialization**:
  - Create a Q-table with dimensions [number_of_states, number_of_actions] and initialize it to zeros.
  - Set the parameters: learning rate α, discount factor γ, exploration rate ϵ, and number of episodes N.
- **Training Loop**:
  - For each episode:
    1. Initialize the agent's starting state.
    2. Choose an action using the epsilon-greedy policy.
    3. While the state is not terminal:
       (a) Take action, observe reward r, and next state s'.
       (b) Choose next action a' using the epsilon-greedy policy.
       (c) Update the Q-value using the SARSA formula.

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

       (d) Set $s \leftarrow s'$ and $a \leftarrow a'$.
  - Decrease the exploration rate ϵ over time to reduce exploration as learning progresses.

- **Evaluate the Policy**:
  - After training, derive the policy from the Q-table by choosing the action with the highest Q-value for each state.
  - Test the learned policy in the environment and measure performance metrics such as total reward and number of steps to reach the goal.
- **Visualize and Analyze**:
  - **Visualize the Q-Table**: Display the Q-values for each state-action pair to understand the learned values.
  - **Visualize the Policy**: Show the optimal policy on the grid, indicating the best action to take from each state.
  - **Analyze Performance**: Compare the performance of the learned policy to a baseline (e.g., random actions) and discuss observations on convergence and effectiveness.

2.4 Activity 3 - A Q-Learning Agent in the Taxi environment

As we have learned in class Q-learning is a model-free reinforcement learning algorithm used to learn the value of actions in given states. The core idea is to update the Q-values (state-action values) based on the rewards received and the estimated optimal future values.

The Q-learning update rule is given by:

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

Where:

- $Q(s,a)$: Current Q-value for state s and action a.
- α: Learning rate (determines how much new information overrides the old).
- r: Reward received after taking action a in state s.
- γ: Discount factor (how much future rewards are considered).
- $\max_{a'} Q(s',a')$: Maximum Q-value for the next state s' across all possible actions a'.

Q-learning detailed algorithm for implementation

```
Require:
  Environment with state space $\mathcal{S}$, action space $\mathcal{A}$
  Learning rate $\alpha \in (0, 1]$
  Discount factor $\gamma \in [0, 1]$
  Exploration rate $\epsilon \in [0, 1]$
  Number of episodes $N$
Ensure: Optimal Q-table and derived policy
  Initialize Q-table: $Q(s, a) \leftarrow 0, \forall s \in \mathcal{S}, a \in \mathcal{A}$
  for episode $= 1$ to $N$
    Initialize state $s$
    while $s$ is not terminal
      With probability $\epsilon$: $a \leftarrow$ random action
      With probability $1 - \epsilon$: $a \leftarrow \text{argmax}\, a'Q(s, a')$
      Take action $a$, observe reward $r$ and next state $s'$
      Update Q-value:
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
      $s \leftarrow s'$

  return Q-table and $\pi(s) = \text{argmax}\, aQ(s, a)$
```

Assignment steps

Set Up the Environment

- Define the Taxi-v3 environment, including the grid layout, obstacles, rewards, and goal state.
- Initialize the state space, action space, and reward structure.

Implement Q-Learning Algorithm

- **Initialization**:
  - Create a Q-table with dimensions $[\text{number\_of\_states}, \text{number\_of\_actions}]$ and initialize it to zeros.
  - Set the parameters: learning rate α, discount factor γ, exploration rate ϵ, and number of episodes N.
- **Training Loop**:
  - For each episode:
    1. Initialize the agent's starting state.
    2. While the state is not terminal:

    (a) Choose an action using the epsilon-greedy policy.
    (b) Execute the action, observe the reward, and the next state.
    (c) Update the Q-value using the Q-learning formula.
    (d) Transition to the next state.

  - Decrease the exploration rate ϵ over time to reduce exploration as learning progresses.

- **Evaluate the Policy**:
  - After training, derive the policy from the Q-table by choosing the action with the highest Q-value for each state.
  - Test the learned policy in the environment and measure performance metrics such as total reward and number of steps to reach the goal.
- **Visualize and Analyze**:
  - **Visualize the Q-Table**: Display the Q-values for each state-action pair to understand the learned values.
  - **Visualize the Policy**: Show the optimal policy on the grid, indicating the best action to take from each state.
  - **Analyze Performance**: Compare the performance of the learned policy to a baseline (e.g., random actions) and discuss observations on convergence and effectiveness.

2.5 Activity 4 - Q-Learning Approach for the Cartpole env

2.5.1 CartPole environment Background

The CartPole environment in Gymnasium [Far24a] is a classic control problem where the objective is to balance a pole on a cart. The cart can move left or right along a frictionless track, and the pole is attached to the cart by an unactuated joint. The system receives a reward for every time step the pole remains upright, with the goal being to maximize this reward by preventing the pole from falling over. The environment is characterized by continuous state space representing the cart's position and velocity, as well as the pole's angle and angular velocity, while actions are discrete, consisting of applying a force either left or right on the cart. The challenge lies in developing a control strategy that effectively maintains the pole's balance despite the inherently unstable nature of the system.

The control objective is to keep the pole in the vertical position by applying horizontal actions (forces) to the cart. The **action space** consists of two actions

- Push the cart left – denoted by 0
- Push the cart right – denoted by 1

The **observation space** for the states is:

1. Cart position, denoted by x in Fig. 3. The minimal and maximal values are -4.8 and 4.8, respectively.

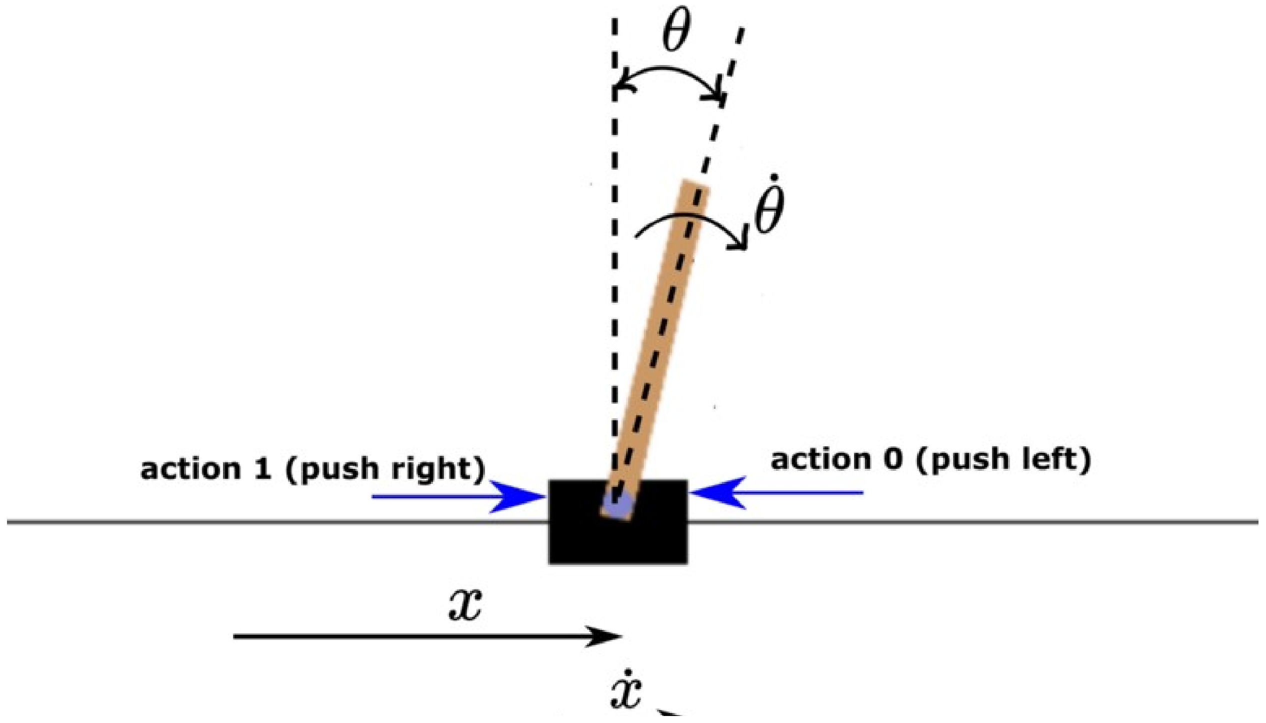

Figure 3: Cartpole environemnt and movement dynamics

2. Cart velocity, denoted by $\dot{x}$ in Fig. 3 The minimal and maximal values are $-\infty$ and ∞, respectively.
3. Pole angle of rotation (measured in radians), denoted by θ. The minimal and maximal values are -0.418 radians (-24 degrees) and 0.418 (24 degrees).
4. Pole angular velocity, denoted by $\dot{\theta}$. The minimal and maximal values are ∞ and ∞, respectively.

It is also important to emphasize that initial states (initial observations) are completely random and the values of states are chosen uniformly from the interval (-0.05,0.05).

An episode ends when: 1) the pole is more than 15 degrees from vertical; or 2) the cart moves more than 2.4 units from the center. The problem is considered "solved" when the average total reward for the episode reaches 195 over 100 consecutive trials

In the next section you have a complete list of step-by-step instruction to perform the practice.

2.5.2 Steps for assignment 3

Initialize the Environment

Use OpenAI's Gym library to create the CartPole environment (`gym.make("CartPole-v1")`). Understand the observation space (which provides continuous variables) and the action space (which includes discrete left and right actions).

Discretize the State Space

Q-learning works with discrete state spaces, but the CartPole environment provides continuous observations. Implement a method to discretize these continuous values into a finite set of bins, which will be used as states in the Q-table.

You would have to discretize the continuous dimensions to a number of buckets. In general, you want to have fewer buckets and keep the state-space as small as possible. Having fewer optimal polices to find means faster training. However, discretizing the state-space too coarsely might prevent convergence as important information might be discretized away.

Here is the function that will take the observation from our model and produce a tuple of 4 integer values:

```
def discretize(x):
    return tuple((x/np.array([0.25, 0.25, 0.01, 0.1])).astype(np.int))
```

Let's also explore another discretization method using bins:

```
def create_bins(i,num):
    return np.arange(num+1)*(i[1]-i[0])/num+i[0]

print("Sample␣bins␣for␣interval␣(-5,5)␣with␣10␣bins\n",create_bins((-5,5),10))

ints = [(-5,5),(-2,2),(-0.5,0.5),(-2,2)] # intervals of values for each parameter
nbins = [20,20,10,10] # number of bins for each parameter
bins = [create_bins(ints[i],nbins[i]) for i in range(4)]

def discretize_bins(x):
    return tuple(np.digitize(x[i],bins[i]) for i in range(4))
```

Play with both discretize functions to understand very well how the conversion continuours-discrete is accomplished as this is the key transformation in this example

3. Initialize the Q-Table

Create a Q-table initialized with zeros. This table will have dimensions based on the discretized state space and the number of actions (2 for left and right). Each entry in the Q-table represents the Q-value for a specific state-action pair.

Implement the Q-Learning Algorithm

- **Hyperparameters**: Define the learning rate (α), discount factor (γ), and exploration rate (ϵ) for the algorithm.
- **Epsilon-Greedy Policy**: Implement an epsilon-greedy policy for action selection. With probability ϵ, select a random action to encourage exploration; otherwise, select the action with the highest Q-value for the current state.

- **Update Rule**: After each action, update the Q-value for the (state, action) pair according to the Q-learning formula:

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

  where s is the current state, a is the chosen action, r is the reward received, s' is the new state, and a' is the next action.

Training Loop

Implement a loop that trains the agent over multiple episodes. For each episode:

- Initialize the environment and get the initial state.
- Run steps until the episode ends (the pole falls or the cart goes off track), updating the Q-table using the Q-learning update rule.
- Gradually reduce ϵ to encourage exploitation over exploration as training progresses.

Evaluate the Performance

After training, evaluate the performance of your Q-learning agent by running it without exploration (i.e., $\epsilon = 0$). Record the total reward over multiple episodes to assess how well the agent has learned to balance the pole.

Expected Outcome / Challenge accomplished

Upon completion, your agent should be able to keep the pole balanced for a significant amount of time in the CartPole environment. Compare your results with the environment's success threshold (a score of 195 over 100 consecutive trials) to evaluate if the agent successfully learned a balancing policy.

3 BONUS: Create 4 learning videos with Frozen-Lake

As you have the algorithms ready, now create videos or GIFs on the learning (first make sure the algorithms converge in a reasonable amount of iterations) use Frozen_Lake 4x4 is_slippery = False.

Watching the videos you will have an idea on how the agent moves in the environment to learn.

4 Questions to answer

Respond 3 questions (you can do it inside the notebook)

- Create a comparative table between the 4 methods (number of iterations, time, ...)
- Which method shows better convergence?
- How long did it take your cartpole to converge?

- Which method you think has better results?
- How long it took your cartpole to converge with Q-learning?

5 Deliverables and submission

Submit one zip file.

The ZIP file must contain the notebooks for the 4 assignments and the BONUS.

Include your name in the titles

- `Lab3_Solutions_Jose_Morales.zip`

Include your name in the title

References

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[Far24a] Farama. *Cartpole environment documentation*. https://gymnasium.farama.org/environments/classic_control/cart_pole/. Accessed: 2024-08-09. 2024.

[Far24b] Farama. *Taxi environment documentation*. https://gymnasium.farama.org/environments/toy_text/taxi/. Accessed: 2024-08-09. 2024.

[Far25] Farama. *Frozen Lake environment documentation*. https://gymnasium.farama.org/environments toy_text/frozen_lake/. Accessed: 2025-03-28. 2025.