# CISC 856 – Reinforcement Learning

# Final Project Report

**Mountain Car (Classic control)**

**Group: 5**


| Name | ID |
|---|---|
| Mohammed Abdelghany | **20398561** |
| Mahmoud Kandeel | **20398558** |
| Mostafa Mohamed Amer | **20398564** |

## Introduction

The Mountain Car problem is a classic reinforcement learning control problem, In this project, our motivation is to explore and apply RL technique  to solve the Mountain Car problem and analyze their performance , The primary objective of this project is to understand and investigate the capabilities and limitations of RL algorithm for solving Mountain Car problem , The Mountain Car problem requires the agent to learn a strategy that involves exploration and overcoming initial disadvantages to achieve the goal. By tackling this problem, we can gain insights into the effectiveness of RL algorithm and his ability to learn in sparse reward environments.

We chose this project because The Mountain Car challenge represents a real-life scenario in which a car with limited power must cross difficult terrain. This topic has parallels in real-world applications such as autonomous driving or robotics, where vehicles must overcome impediments or environmental limits to achieve their objectives.

## Problem Formulation

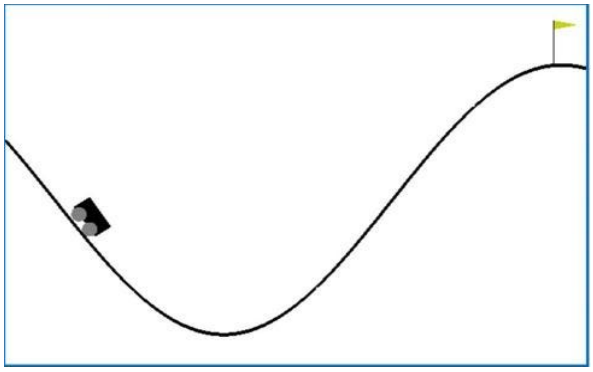
Mountain Car is a classic reinforcement learning task in which agents must learn how to drive a car in a two-dimensional continuous world. The task's purpose is to master the physics of the environment such that the car can climb a steep hill.

A car is positioned in a valley between two mountains in the Mountain Car environment. The car's engine lacks the power to propel the vehicle directly up the hill. The car must drive back and forth in the valley until it reaches the top of the right hill.

Mountain Car’s state space is continuous and characterized by the car's position and velocity. The position is between -1.2 and 0.6, and the velocity is between -0.07 and 0.07. The car begins at the valley's bottom with an initial location and zero velocity.

Mountain Car’s action space is discrete, with three possible actions: accelerate to the left, accelerate to the right, or do nothing.

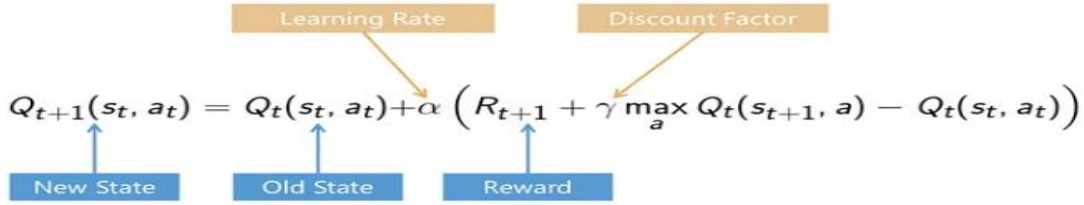
Because the car's acceleration is limited, climbing the hill straight is difficult. To resist gravity forces and reach the top of the mountain, the agent must carefully determine when and in which direction to apply acceleration.



### Solution Overview

We used a reinforcement learning method to solve the Mountain Car challenge, specifically the Q-learning algorithm. Q-learning is a model-free RL technique that learns the Q-function, an action-value function, to predict the expected cumulative reward for performing a certain action in each state.

The formulation of Q-learning:




The Mountain Car environment provides the following key components:

1- State Space: The state space of the environment is continuous and represents the position and velocity of the car.
2- The action space is made up of discrete actions that the agent can take. In this setting, the agent has three options: go to the left, move to the right, or do nothing.
3- Rewards: For each time step until it achieves its goal state, the agent receives a negative reward of one. When the agent achieves its goal state, it receives a 0 reward, and the episode ends.

We used The Mountain Car environment which is part of the OpenAI Gym library as default trial and we used the same environment of the OpenAI Gym in multiple trials with improvement the trial until the agent achieves the goal and arrive to the goal in the fourth Trial.

## Trial 1:

It is a default trail, we used The Mountain Car environment which is part of the OpenAI Gym , in this trial , we don't use any function to update the value state

Start the code by importing the required libraries.

```
# import required libraries
import gym
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from IPython.display import HTML
```

1- We start the code with create an instance from Mountain Car environment called "MountainCar-v0" the reset the environment to initial state.

2- Determine the position for each bucket in discrete state representation with 20 elements and determine the size of each bucket called "discrete_win_size"

3- Create loop that iterate until the episode done, in each trial the agent chooses 2 that means "throttle forward" and return state, reward, and done flag.

4- Close the environment.

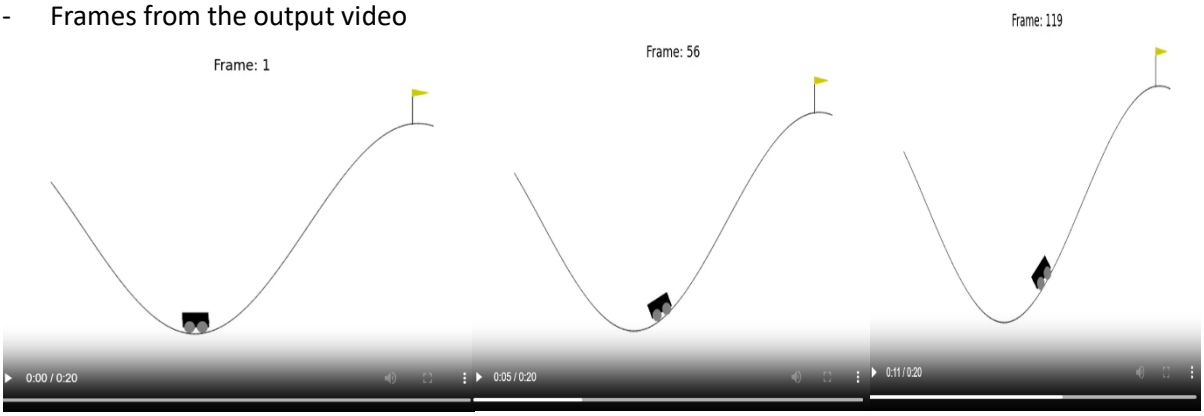
5- Use "update function" that take the frame number as an input and display each frame in "ax", the create animation on all frames using 'FuncAnimation' then use "to_html5_video" to convert the animation into HTML5 video.

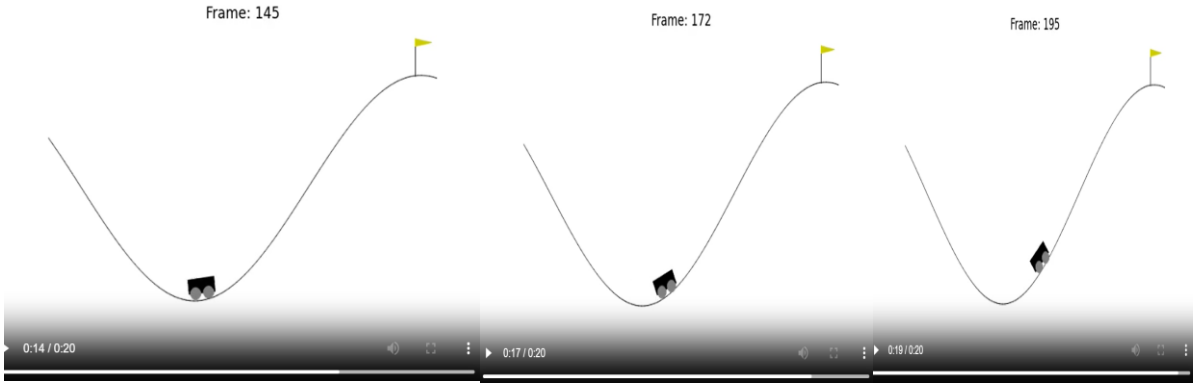
**The Result**

- This Link contain Video that represents the output, click on icon to show video.

https://drive.google.com/file/d/1xqKO4AhywQpAYN2263KGDjvYvI1F8o-9/view?usp=sharing

- Frames from the output video




- from the output, detect that the car trying to get up the mountain, but fail because not enough power helps the car to reach to goal.
- the goal is eventually making the agent arrive to flag area.

## Trial 2

In this trial we make change of the code of previous trial:

1- create q_table, it is a large table that give any combination of states.
2- for every combination of state of position and the velocity, we will look on this table.
3- trying to exploit the environment, we will pick one with the maximum largest q_value
4- initialize q_table with random values.
5- we need to convert the continuous values of states into discrete value.
6- Set the learning rate for updating the Q-values
7- Set the discount factor for future rewards.
8- Set the number of episodes (trials) to run.
9- Determine the position for each bucket in the observation space.
10- Initialize the Q-table with random values.
11- Define a function **get_discrete_state** to convert the continuous state into a discrete state.
12- Get the discrete state of the initial observation.
13- Create an empty list **frame** to store the frames captured from the environment.
14- Run the episodes loop.
15- Set the **done** flag to False at the beginning of each episode.
16- Start the inner loop until the **done** flag is True.
17- Choose the action based on the maximum Q-value for the current discrete state.
18- Perform the chosen action in the environment and get the new state, reward, and done flag.
19- Convert the new state to its corresponding discrete state.
20- Render the environment in RGB array mode and capture the frame.
21- If the episode is not done:
22- Calculate the maximum future Q-value for the new state.
23- Get the current Q-value for the current state and action.
24- Update the Q-value using the Q-learning formula.
25- Update the Q-table with the new Q-value.
26- If the new state's position is greater than or equal to the goal position, set the Q-value to 0.
27- Update the discrete state to the new discrete state for the next iteration.
28- Close the Mountain Car environment.

- The difference between the previous trial and the current trial is that the current trial applies Q-learning to the Mountain Car problem and captures frames to create an animation of the agent's actions and progress. It then saves the animation as an MP4 video.

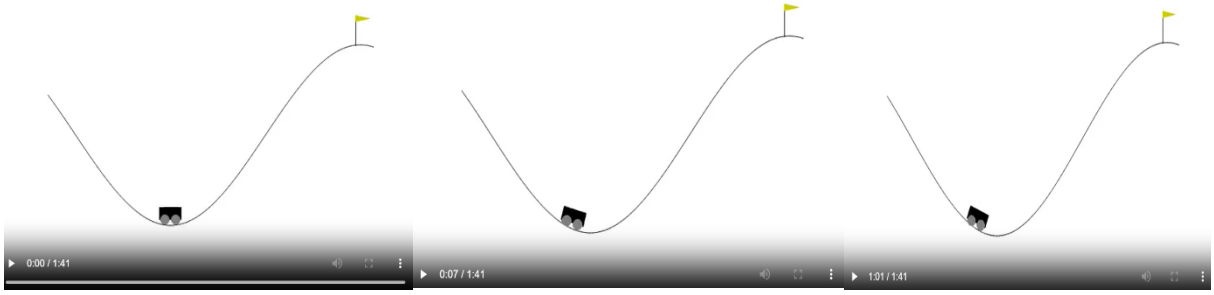
**The Result**

- This Link contain Video that represents the output, click on icon to show video.

https://drive.google.com/file/d/1LkVUN84cd7JWJ20lUHWUdoX7GvZoE27K/view?usp=sharing

- Frames from the output video

- From the previous video, we can detect that the agent starts exploring the environment and make a back movement that help the agent later to reach the goal.




---

## Trial 3

In this trial we need to iterate over episodes to run multiple time, because runs one time that is not useful and not help the agent to get the goal.

1. The number of episodes (trials) has been increased to 25000 instead of 10000.
2. The variable **show_every** has been introduced to print the current episode every 2000 episodes.
3. iterates over the episodes. If the current episode number is divisible by **show_every**, it prints the episode number and sets **render** to True, indicating that the frames should be rendered and stored. Otherwise, **render** is set to False.

4. chooses an action based on the maximum Q-value from the Q-table, takes a step in the environment, and captures the frame if **render** is True.

```
# number of trials (arrive at terminal state)
episodes = 25000
# every 2000 episodes let us know the agent is still alive
show_every = 2000
```

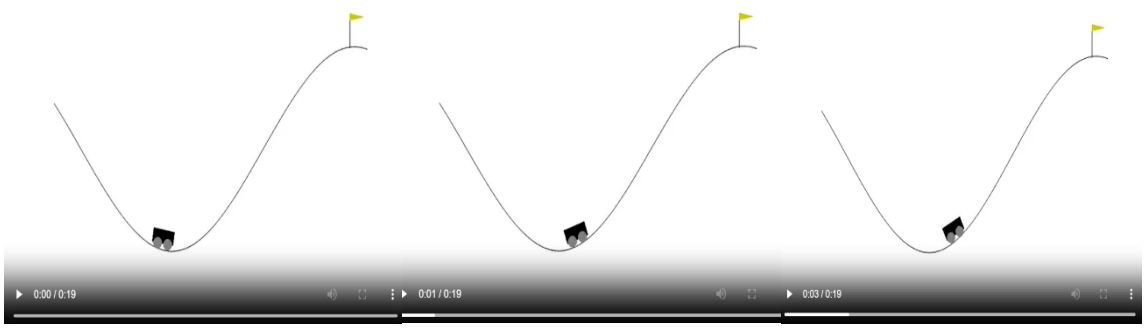
```
# iterate over episodes
for episode in range(episodes):
    if episode % show_every == 0:
        print(episode)
        render = True
    else:
        render = False
```

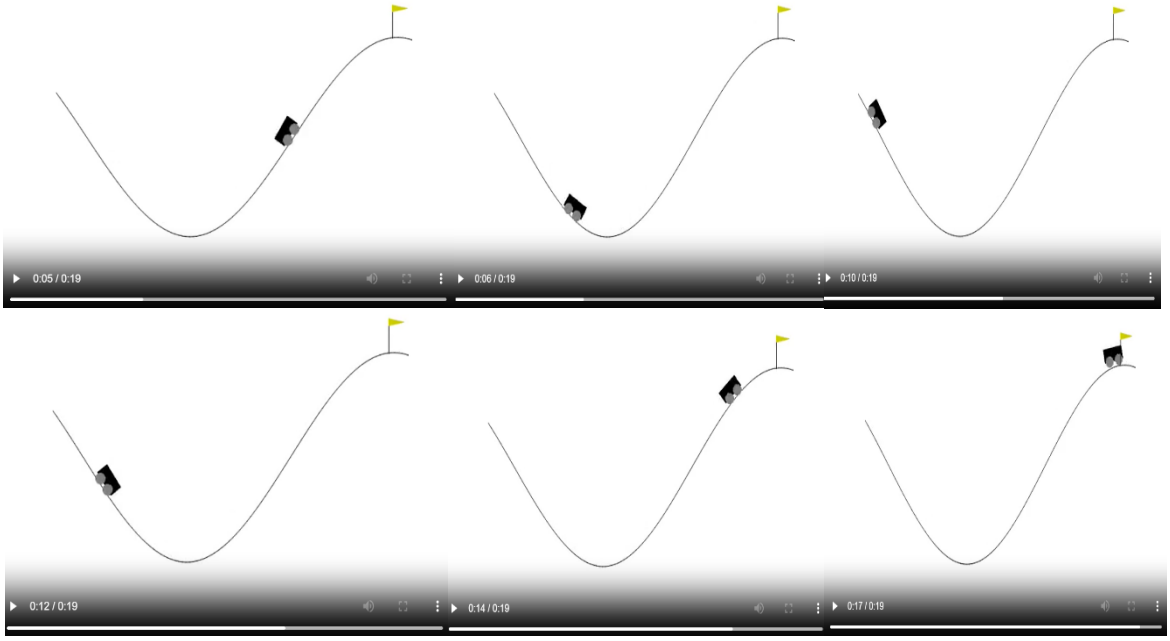
```
if not done:
    # Max future Q multiplied by discount (weight)
    max_future_q = np.max(q_table[new_discrete_state])
    current_q = q_table[discrete_state + (action,)]
    # Formula for calculating the new Q-value
    new_q = (1 - learning_rate) * current_q + learning_rate * (reward + discount * max_future_q)
    # Update q_table with new q_value
    q_table[discrete_state + (action,)] = new_q
elif new_state[0] >= env.goal_position:
    q_table[discrete_state + (action,)] = 0
discrete_state = new_discrete_state
```

**The Result**

- This Link contain Video that represents the output, click on icon to show video.

https://drive.google.com/file/d/1N7z4Si9l_MkocLFLI4wKBBWWd8Qm-rN2/view?usp=sharing






- we notice that the agent reaches the goal but make it quickly, because we need one more thing usually and it is called Epsilon that mean how much the exploration(randomness)
- epsilon is a hyperparameter that determines the probability of selecting a random action instead of the action with the maximum Q-value. By adding epsilon, the agent has a chance to explore different actions and states, even if the current Q-values suggest a different optimal action. This exploration is crucial for discovering new, potentially better actions and learning a more accurate representation of the environment.

---

**Trial 4:**

1- Add Epsilon to code.
2- the benefit of these trial, still doing some exploration because as soon as it finds one way to the flag, it will be super optimized very quickly for that way of getting the to flag.

3- calculates the epsilon decay value (“epsilon_decay_value”) by dividing the initial epsilon value by the difference between the starting and ending episodes.And during each episode, if the current episode falls within the range of “ start_epsilon” to “end_epsilon”, it reduces the epsilon value by the decay value

```
# higher epsilon mean perform a random action
epsilon = 0.5
# start epsilon
start_epsilon = 1
# end_ psilon
end_epsilon = episodes // 2
epsilon_decay_value = epsilon / (end_epsilon - start_epsilon)
# initialize q_table
q_table = np.random.uniform(low=-2, high=0, size=(discrete_of_size + [env.action_space.n]))
```

4- Random Action selection:

- checks if a random number is greater than the current epsilon value.
- If it is, it selects the action with the maximum Q-value (**np.argmax(q_table[discrete_state])**), otherwise, it selects a random action (**np.random.randint(0, env.action_space.n)**)

```
while not done:
    if np.random.random() > epsilon:
        action = np.argmax(q_table[discrete_state])
    else:
        action = np.random.randint(0, env.action_space.n)
```

5- **In Training Progress**:
plots the training progress using matplotlib, with the x-axis representing episodes and the y-axis representing reward values. It plots the average, minimum, and maximum rewards over episodes.

```
Episode: 0
average: -0.1
min: -200.0
max: -200.0
Cumulative Reward: -200.0
Number of Steps: 1
```

```
Episode: 2000
average: -192.7035
min: -200.0
max: -116.0
Cumulative Reward: -385607.0
Number of Steps: 2001
```

```
Episode: 4000
average: -178.462
min: -200.0
max: -112.0
Cumulative Reward: -742531.0
Number of Steps: 4001
```

```
Episode: 6000
average: -155.3775
min: -200.0
max: -93.0
Cumulative Reward: -1053286.0
Number of Steps: 6001
```

```
Episode: 8000
average: -142.252
min: -200.0
max: -110.0
Cumulative Reward: -1337790.0
Number of Steps: 8001
```

6- Test Performance Evaluation

- Every ten episodes, the agent's performance on a separate set of test episodes ("test_episodes") is evaluated.
- It keeps track of the rewards and time steps taken in the "test_rewards" and "test_time_steps" lists.

```
if episode % 10 == 0:
    try:
        test_episodes = 5
        test_rewards = []
        test_time_steps = []
        for _ in range(test_episodes):
            state = env.reset()
            done = False
            time_steps = 0
            while not done:
                action = np.argmax(q_table[get_discrete_state(state)])
                state, reward, done, _ = env.step(action)
                time_steps += 1
                test_rewards.append(reward)
            test_time_steps.append(time_steps)

        mean_test_reward = np.mean(test_rewards)
        mean_test_time_steps = np.mean(test_time_steps)

        print(f"\nTest Performance - Episode: {episode} Mean Reward: {mean_test_reward} Mean Time-Steps: {mean_test_time_steps}")
        print("*********************************************")
    except KeyboardInterrupt:
        print("Testing interrupted by the user.")
        break
```

```
Test Performance - Episode: 0 Mean Reward: -1.0 Mean Time-Steps: 200.0
*********************************************

Test Performance - Episode: 10 Mean Reward: -1.0 Mean Time-Steps: 200.0
*********************************************

Test Performance - Episode: 20 Mean Reward: -1.0 Mean Time-Steps: 200.0
*********************************************

Test Performance - Episode: 30 Mean Reward: -1.0 Mean Time-Steps: 200.0
```

```
Test Performance - Episode: 2000 Mean Reward: -1.0 Mean Time-Steps: 176.4
*********************************************

Test Performance - Episode: 2010 Mean Reward: -1.0 Mean Time-Steps: 153.0
*********************************************

Test Performance - Episode: 2020 Mean Reward: -1.0 Mean Time-Steps: 200.0
*********************************************

Test Performance - Episode: 2030 Mean Reward: -1.0 Mean Time-Steps: 200.0
*********************************************
```

```
Test Performance - Episode: 4000 Mean Reward: -1.0 Mean Time-Steps: 169.6
*********************************************

Test Performance - Episode: 4010 Mean Reward: -1.0 Mean Time-Steps: 155.8
*********************************************

Test Performance - Episode: 4020 Mean Reward: -1.0 Mean Time-Steps: 158.8
*********************************************

Test Performance - Episode: 4030 Mean Reward: -1.0 Mean Time-Steps: 174.0
*********************************************
```

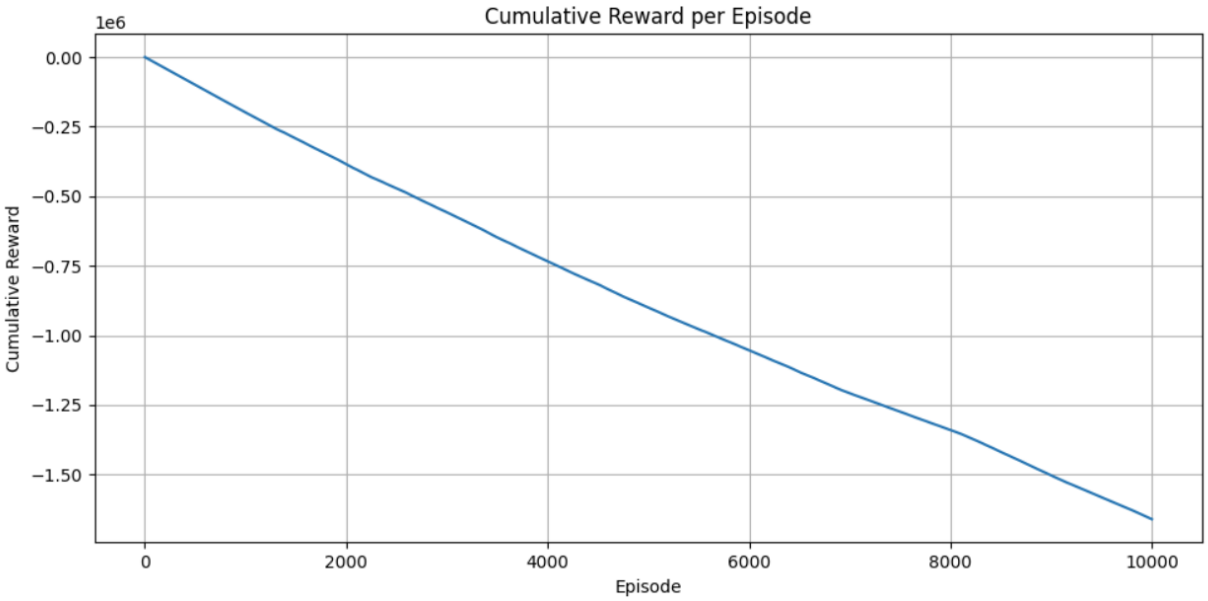
```
Test Performance - Episode: 6000 Mean Reward: -1.0 Mean Time-Steps: 141.4
*********************************************

Test Performance - Episode: 6010 Mean Reward: -1.0 Mean Time-Steps: 142.6
*********************************************

Test Performance - Episode: 6020 Mean Reward: -1.0 Mean Time-Steps: 157.0
*********************************************

Test Performance - Episode: 6030 Mean Reward: -1.0 Mean Time-Steps: 139.0
```

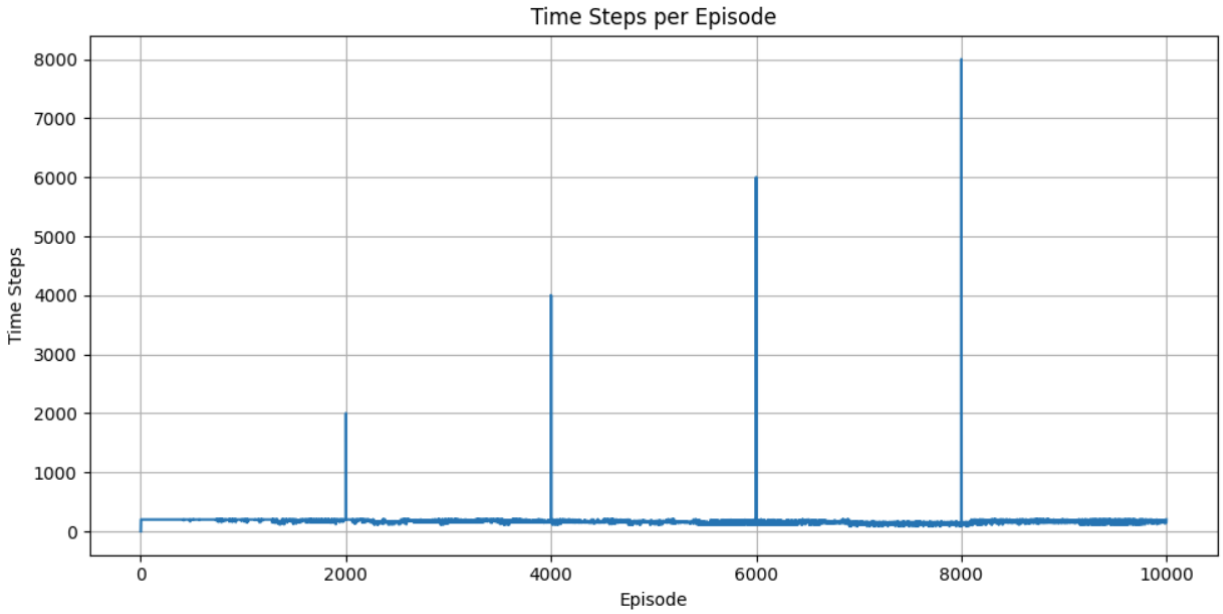
7- Calculate and compute the cumulative reward and number of steps per episode, for the agent performance during training.

```
ep_rewards.append(episode_reward)
if not episode % show_every:
    average_reward = sum(ep_rewards[-show_every:]) / show_every
    aggregate_ep_rewards['ep'].append(episode)
    aggregate_ep_rewards['avg'].append(average_reward)
    aggregate_ep_rewards['min'].append(min(ep_rewards[-show_every:]))
    aggregate_ep_rewards['max'].append(max(ep_rewards[-show_every:]))
    cumulative_reward = sum(ep_rewards)
    time_steps = len(ep_rewards)
    print(f"\n Episode: {episode}\n average: {average_reward}\n min: {min(ep_rewards[-show_every:])}\n max: {max(ep_rewards[-show_every:])}")
    print(f"Cumulative Reward: {cumulative_reward}")
    print(f"Number of Steps: {time_steps}")

cumulative_rewards.append(np.sum(ep_rewards))
time_steps_per_episode.append(time_steps)
```

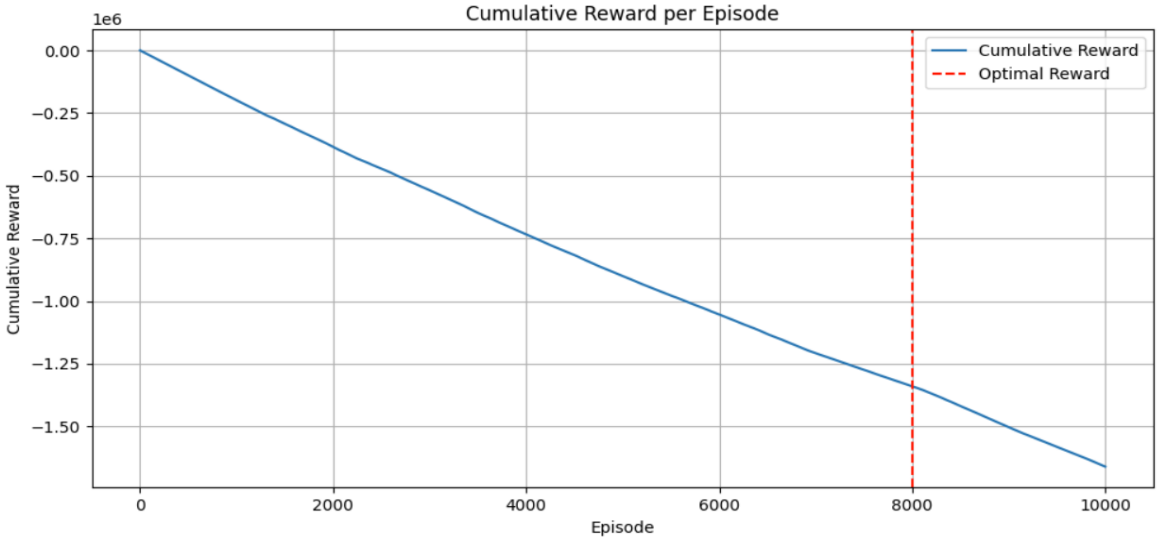
```
# plotting training progress
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 10))

# Plot cumulative reward per episode
ax1.plot(np.arange(episodes), cumulative_rewards)
ax1.set_xlabel("Episode")
ax1.set_ylabel("Cumulative Reward")
ax1.set_title("Cumulative Reward per Episode")
ax1.grid(True)
```

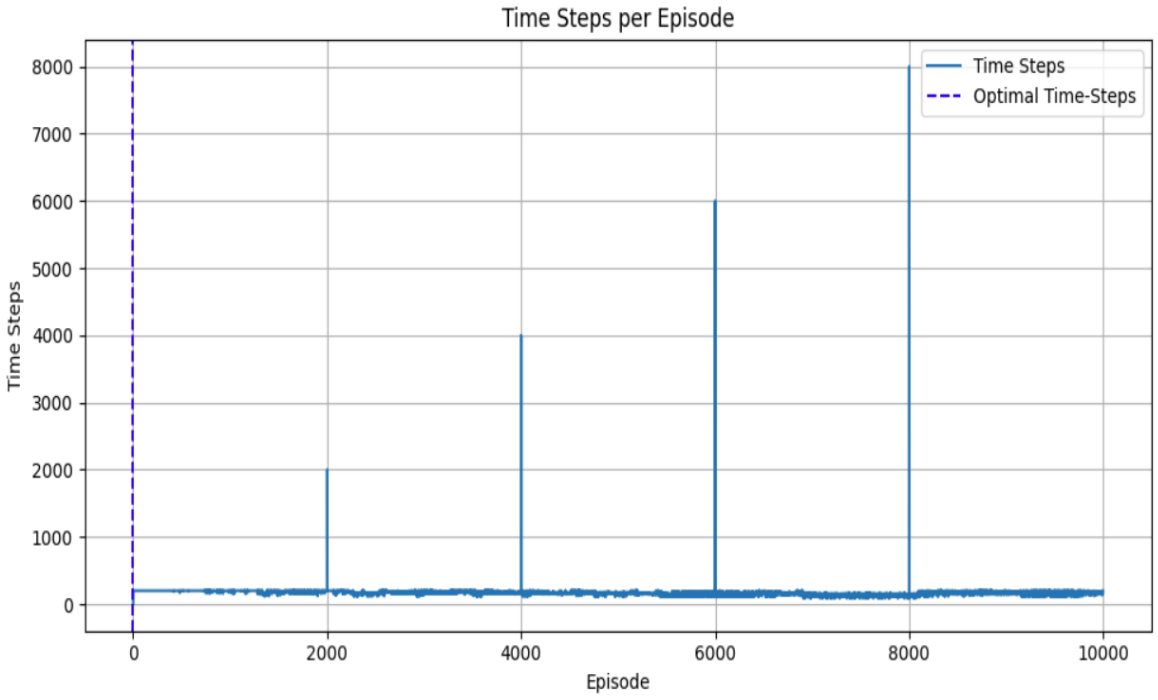
```
# Plot time-steps per episode
ax2.plot(np.arange(episodes), time_steps_per_episode)
ax2.set_xlabel("Episode")
ax2.set_ylabel("Time Steps")
ax2.set_title("Time Steps per Episode")
ax2.grid(True)

plt.tight_layout()
plt.show()
```



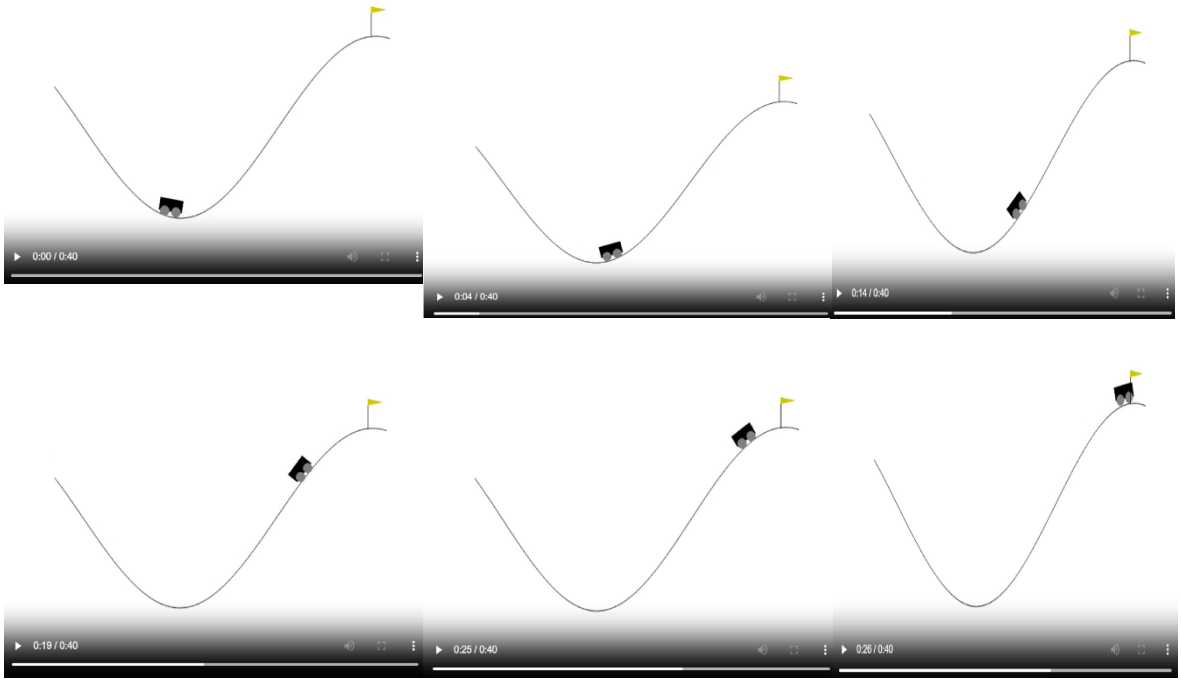

8- -plotting the cumulative reward and time steps per episode separately to provides a clearer view of the agent's learning progress in terms of both cumulative reward and the number of time steps taken per episode.
The vertical lines indicate episodes with the optimal reward and time steps, making it easier to identify the best-performing episodes.






**The Result**

- This Link contain Video that represents the output, click on icon to show video.
https://drive.google.com/file/d/1JklzYfqyrvNOEzArN9eMt3LaPSOtUtTQ/view?usp=sharing




- The current Trial is the best Trial because of optimizing the agent's performance, the agent success to reach the goal in the best way and we notice that the agent learns how to reach the goal better than the previous Trial.

## Hyper parameter search

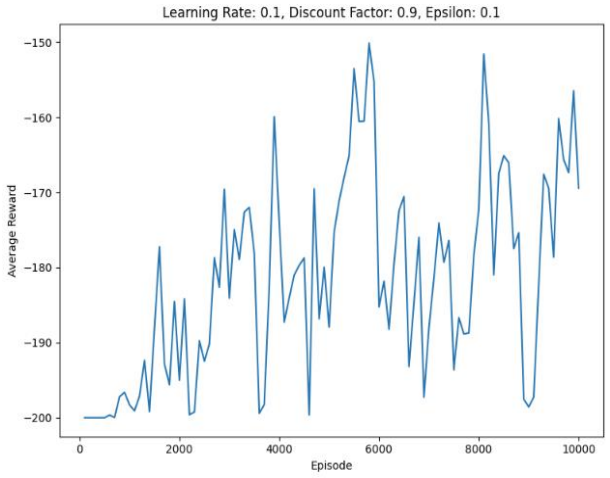
Learning rates, discount factors, epsilon values, the number of episodes, and the frequency of showing the average reward are all defined.

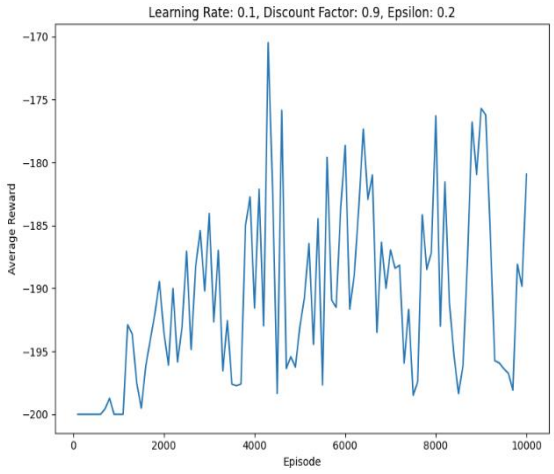
```
# Hyperparameters
learning_rates = [0.1, 0.2]
discount_factors = [0.9, 0.95]
epsilon_values = [0.1, 0.2]
episodes = 10000  # Define the number of episodes
show_every = 100
```

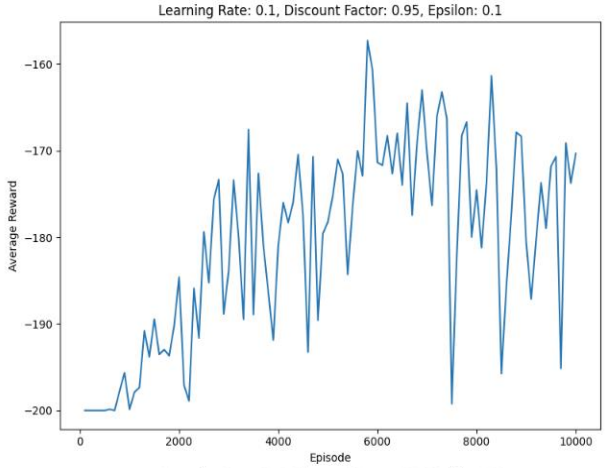
The “train_agent” function is used to train the agent using the specified hyperparameters. In each episode, it initializes the Q-table, tracks episode rewards and average rewards, and runs the Q-learning algorithm, Using nested loops, the training process will be carried out for different combinations of hyperparameters, prints the episode number and the average reward over the last **show_every** episodes, Progress updates that display the episode number and the average reward over the last show_every episode.

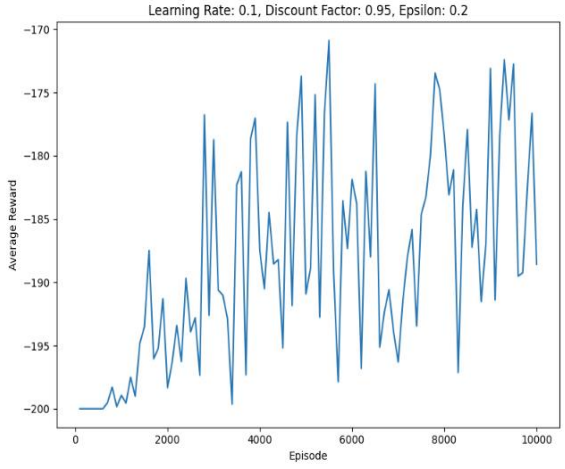
```
# Function to train the agent with given hyperparameters
def train_agent(learning_rate, discount_factor, epsilon):
    # Initialize Q-table
    q_table = np.zeros(discrete_size + [env.action_space.n])

    # Tracking performance over episodes
    episode_rewards = []
    avg_rewards = []

    for episode in range(episodes):
        # Initialize episode-specific variables
        episode_reward = 0
        state = get_discrete_state(env.reset())
        done = False

        while not done:
            if np.random.random() > epsilon:
                action = np.argmax(q_table[state])
            else:
                action = np.random.randint(0, env.action_space.n)

            new_state, reward, done, _ = env.step(action)
            episode_reward += reward

            new_state = get_discrete_state(new_state)

            # Update Q-table using Q-learning algorithm
            q_table[state + (action,)] += learning_rate * (reward + discount_factor * np.max(q_table[new_state]) - q_table[state + (action,)])

            state = new_state

        episode_rewards.append(episode_reward)

        if (episode + 1) % show_every == 0:
            avg_reward = np.mean(episode_rewards[-show_every:])
            avg_rewards.append(avg_reward)
            print(f"Episode: {episode+1} Average Reward: {avg_reward}")

    return avg_rewards
```

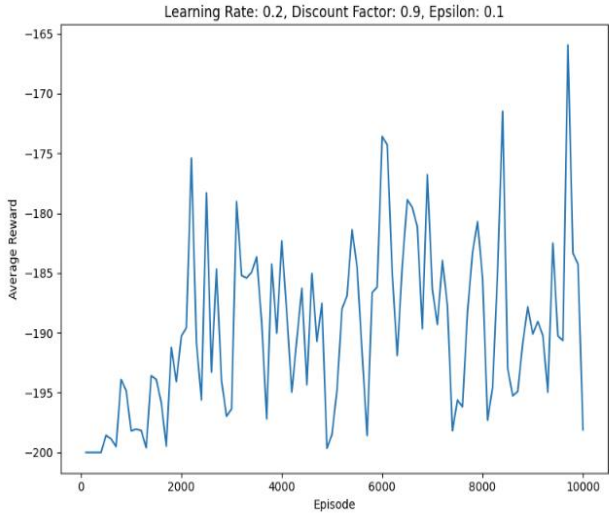
After each collection of hyperparameters, a visualization is generated displaying the average reward over episodes for that specific combination.

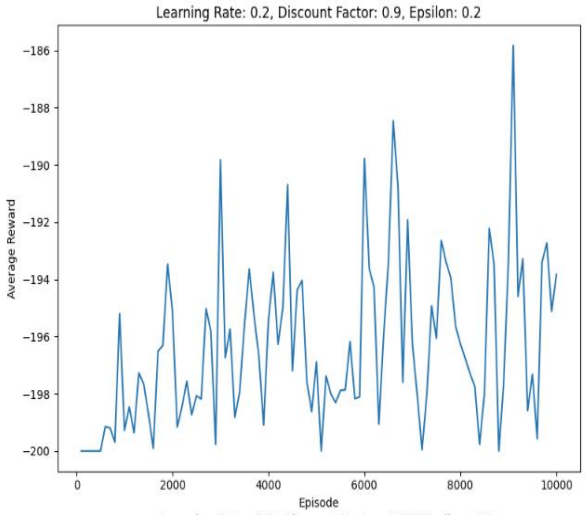

Learning Rate: 0.1, Discount Factor: 0.9, Epsilon: 0.1
Average Reward
Episode

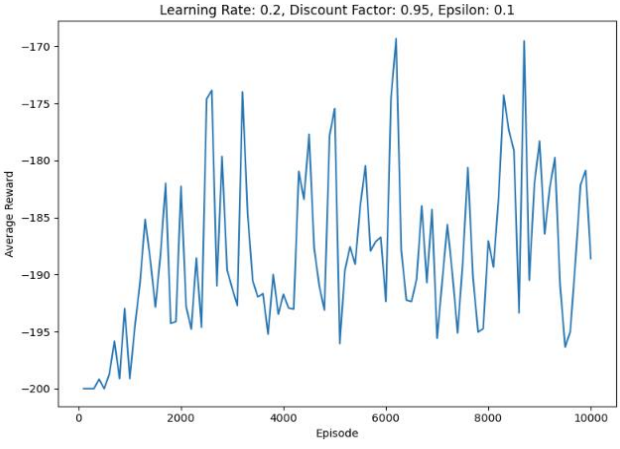

Learning Rate: 0.1, Discount Factor: 0.9, Epsilon: 0.2
Average Reward
Episode

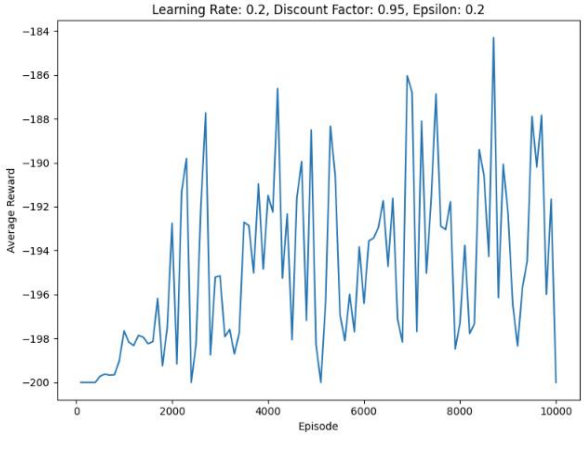

Learning Rate: 0.1, Discount Factor: 0.95, Epsilon: 0.1
Average Reward
Episode


Learning Rate: 0.1, Discount Factor: 0.95, Epsilon: 0.2
Average Reward
Episode


Learning Rate: 0.2, Discount Factor: 0.9, Epsilon: 0.1
Average Reward
Episode


Learning Rate: 0.2, Discount Factor: 0.9, Epsilon: 0.2
Average Reward
Episode


Learning Rate: 0.2, Discount Factor: 0.95, Epsilon: 0.1
Average Reward
Episode


Learning Rate: 0.2, Discount Factor: 0.95, Epsilon: 0.2
Average Reward
Episode

| Learning Rate | Discount Factor | Epsilon | Maximum Average Reward | Episode Number |
|---|---|---|---|---|
| 0.1 | 0.9 | 0.1 | -150.12 | 5800 |
| 0.1 | 0.9 | 0.2 | -170.49 | 4300 |
| 0.1 | 0.95 | 0.1 | -157.29 | 5800 |
| 0.1 | 0.95 | 0.2 | -172.42 | 9300 |
| 0.2 | 0.9 | 0.1 | -165.94 | 9700 |
| 0.2 | 0.9 | 0.2 | -185.83 | 9100 |
| 0.2 | 0.95 | 0.1 | -169.33 | 6200 |
| 0.2 | 0.95 | 0.2 | -184.31 | 8700 |

**Observation**

- The best hyperparameters are (Learning Rate=0.1, Discount Factor0.95, Epsilon=0.1), because when we use these hyperparameters we get the best Average reward that equal -157.29 in episode number equal 5800
- we notice that the worst Average reward is -200, and this happens in episode number (100, 200,300,400) for each combination.

## Result

| Trials | State |
|---|---|
| Trial 1 | Fail to reach Goal |
| Trial 2 | Fail to reach Goal |
| Trial 3 | Reach to goal |
| Trial 4 | Reach to goal (optimization way) |

## Conclusion

We explored the Mountain Car problem using reinforcement learning, we started by understanding the problem concept and motive. Then, in the MountainCar-v0 environment, we implemented Q-learning technique and trained an agent. To promote effective learning, we discovered the importance of keeping a balance between exploration and exploitation. We also looked at how reward shaping, and different update rules affected the agent's learning process.

We gained insights into the agent's performance and the efficiency of different hyperparameter settings by visualizing the training progress and analyzing the cumulative rewards and time steps every episode ,

During the training process, we used animations to visualize the agent's behavior, which helped us understand how the agent interacted with the environment , Factors like as the quality of the reward function, the method chosen, and the complexity of the challenge itself can all have an impact on the agent's performance.