LAB6-VARIANT2 GROUP 12 REPORT SHEET #Ayşe Nur Gümüş- Kaan Baturalp Coşdan

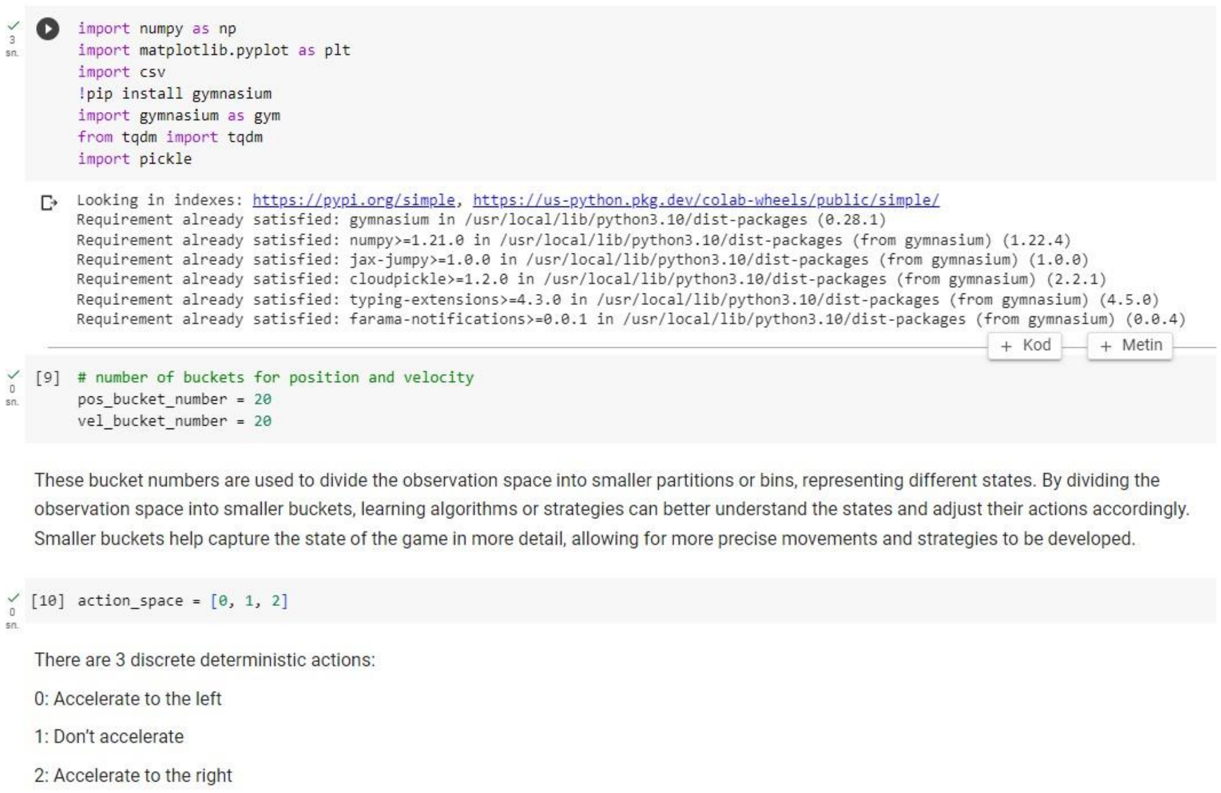
Project Goal: The goal of this project is to implement the Q-Learning algorithm to solve the MountainCar problem in Reinforcement Learning. The objective is to train an agent to control a car in a simulated environment and make it reach the top of a hill by applying the correct actions at the right time.

Scope: The project focuses on using the "MountainCar-v0" environment provided by the OpenAI Gym library. The implementation involves initializing the Q-table, performing the training loop to update the Q-table based on the agent's actions and observed states, and logging training metrics to track the agent's progress. Additionally, inference is conducted to visualize how the agent solves the environment using the updated Q-table

Algorithm:

1. Initialization:
    - Initialize the Q-table with initial values.
    - Set the hyperparameters: learning rate (alpha), discount factor (gamma), exploration rate (epsilon).
2. Training Loop:
    - Iterate over a predetermined number of episodes:
        - Reset the environment and get the initial state.
        - Select the initial action by indexing the Q-table with the initial state.
        - Repeat the following steps until the episode ends or is truncated:
            - Apply the selected action to the environment and get the new state, reward, termination status, and truncation status.
            - Select the new action by indexing the Q-table with the new state (using an epsilon-greedy policy for exploration or selecting the action with the maximum Q-value).
            - Update the Q-table: Q[state, action] = Q[state, action] + alpha * (reward + gamma * max(Q[new_state, :]) - Q[state, action])
            - Update the state: state = new_state
3. Recording Training Metrics:
    - Save the obtained reward at the end of each episode.
    - Record and visualize the metrics to track the progress of the training process.
4. Inference:
    - Visualize how the agent solves the environment using the updated Q-table.

CODE AND EXPLANATION



```python
import numpy as np
import matplotlib.pyplot as plt
import csv
!pip install gymnasium
import gymnasium as gym
from tqdm import tqdm
import pickle
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gymnasium in /usr/local/lib/python3.10/dist-packages (0.28.1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (1.22.4)
Requirement already satisfied: jax-jumpy>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (1.0.0)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (2.2.1)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (4.5.0)
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/python3.10/dist-packages (from gymnasium) (0.0.4)
```

```python
[9]  # number of buckets for position and velocity
     pos_bucket_number = 20
     vel_bucket_number = 20
```

These bucket numbers are used to divide the observation space into smaller partitions or bins, representing different states. By dividing the observation space into smaller buckets, learning algorithms or strategies can better understand the states and adjust their actions accordingly. Smaller buckets help capture the state of the game in more detail, allowing for more precise movements and strategies to be developed.

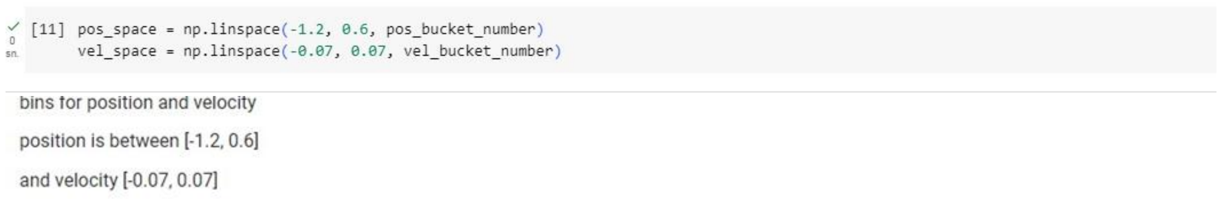
```python
[10] action_space = [0, 1, 2]
```

There are 3 discrete deterministic actions:

0: Accelerate to the left

1: Don't accelerate

2: Accelerate to the right

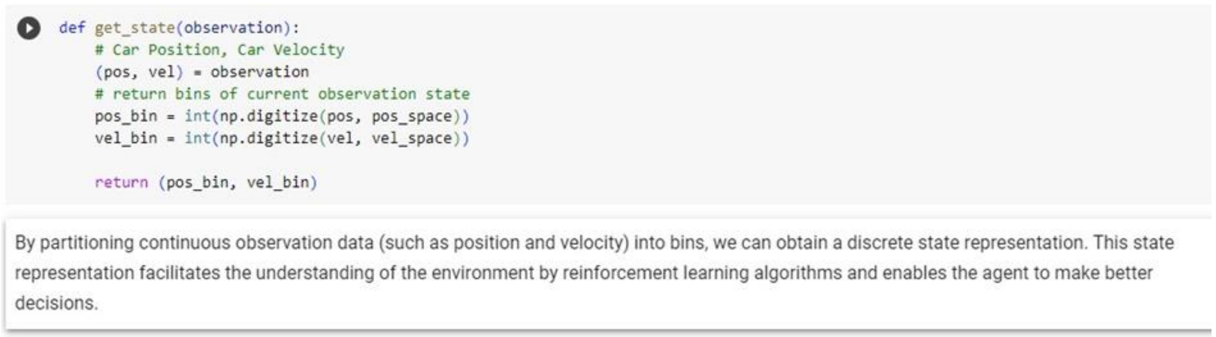


```python
[11] pos_space = np.linspace(-1.2, 0.6, pos_bucket_number)
     vel_space = np.linspace(-0.07, 0.07, vel_bucket_number)
```

bins for position and velocity

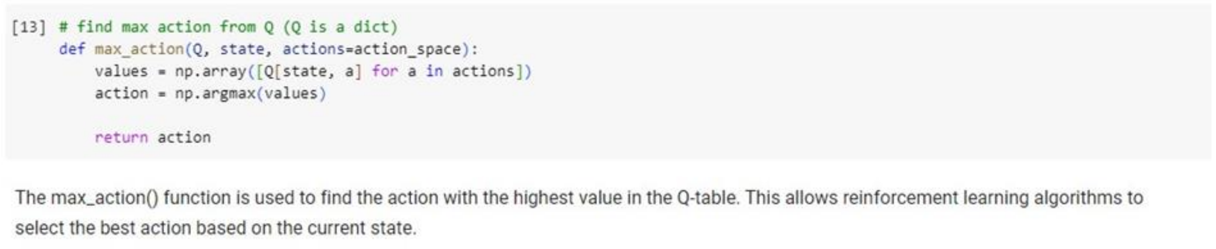
position is between [-1.2, 0.6]

and velocity [-0.07, 0.07]



```python
def get_state(observation):
    # Car Position, Car Velocity
    (pos, vel) = observation
    # return bins of current observation state
    pos_bin = int(np.digitize(pos, pos_space))
    vel_bin = int(np.digitize(vel, vel_space))

    return (pos_bin, vel_bin)
```

By partitioning continuous observation data (such as position and velocity) into bins, we can obtain a discrete state representation. This state representation facilitates the understanding of the environment by reinforcement learning algorithms and enables the agent to make better decisions.



```python
[13] # find max action from Q (Q is a dict)
     def max_action(Q, state, actions=action_space):
         values = np.array([Q[state, a] for a in actions])
         action = np.argmax(values)

         return action
```

The max_action() function is used to find the action with the highest value in the Q-table. This allows reinforcement learning algorithms to select the best action based on the current state.

```python
# This function is a tool used to track and visualize the performance of a reinforcement learning algorithm.
def plot(scores, epsilons, num_of_episodes):
    fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(20, 12), sharex=True)

    ax1.plot(scores, color='blue')
    ax1.set_xlabel('Episode', fontsize=16)
    ax1.set_ylabel('Score', color='blue', fontsize=16)
    ax1.tick_params(axis='y', labelcolor='blue', labelsize=14)
    ax1.set_ylim(-1000, 0)
    ax1.set_yticks(range(-1000, 1, 100))
    ax1.set_xlim(0, num_of_episodes)
    ax1.set_xticks(np.arange(0, num_of_episodes, 100))
    ax1.grid(alpha=0.4)

    ax2.plot(epsilons, color='red', linewidth=3)
    ax2.set_ylabel('Epsilon', color='red', fontsize=16)
    ax2.tick_params(axis='y', labelcolor='red', labelsize=14)
    ax2.set_ylim(0, 1)
    ax2.grid(alpha=0.4)

    mean_scores = [np.mean(scores[max(0, i - 99):(i + 1)]) for i in range(len(scores))]

    ax3 = ax1.twinx()
    ax3.plot(mean_scores, color='green', linewidth=3)
    ax3.set_ylabel('Mean score (last 100 episodes)', color='green', fontsize=16)
    ax3.tick_params(axis='y', labelcolor='green', labelsize=14)
    ax3.set_ylim(-1000, 0)
    ax3.set_yticks(range(-1000, 1, 100))
    ax3.spines['right'].set_visible(False)
    ax3.spines['left'].set_visible(False)
    ax3.yaxis.set_label_position('right')
    ax3.yaxis.tick_right()

    plt.title('num_of_episodes = ' + str(num_of_episodes), fontsize=20, color='green')

    plt.tight_layout()
    plt.savefig('scores_mcc.png')
    plt.show()
```

```python
[16] if __name__ == '__main__':
        num_of_episodes = 5000 #the total number of episodes (iterations) the agent will run during the learning process.
        num_of_steps_per_episode = 1000 #represents the maximum number of steps the agent can take within each episode. If the agent doesn't reach the goal or terminate before reaching this step limit, the episode ends
        alpha = 0.1  # is the learning rate parameter used in the Q-learning algorithm. It determines the weight given to the new information obtained from each learning step.
        gamma = 0.9  # is the discount factor parameter used in the Q-learning algorithm. It determines the importance of future rewards compared to immediate rewards. A higher value of gamma gives more weight to future rewards.
        eps = 1.  # the exploration parameter or epsilon, represents the probability of an agent taking a random action
```

```python
[15] env = gym.make('MountainCar-v0', max_episode_steps=num_of_steps_per_episode)
```

In the Mountain Car environment, a car tries to climb a hill but doesn't have enough power to do so on its own. The car needs to apply the gas at the right time and control the acceleration correctly to reach the top of the hill. The environment is defined by the car's current position, velocity, and three possible actions to choose from (left, stay, right).

This line creates the environment and assigns it to the env variable, making it ready to perform actions and interact with the environment.

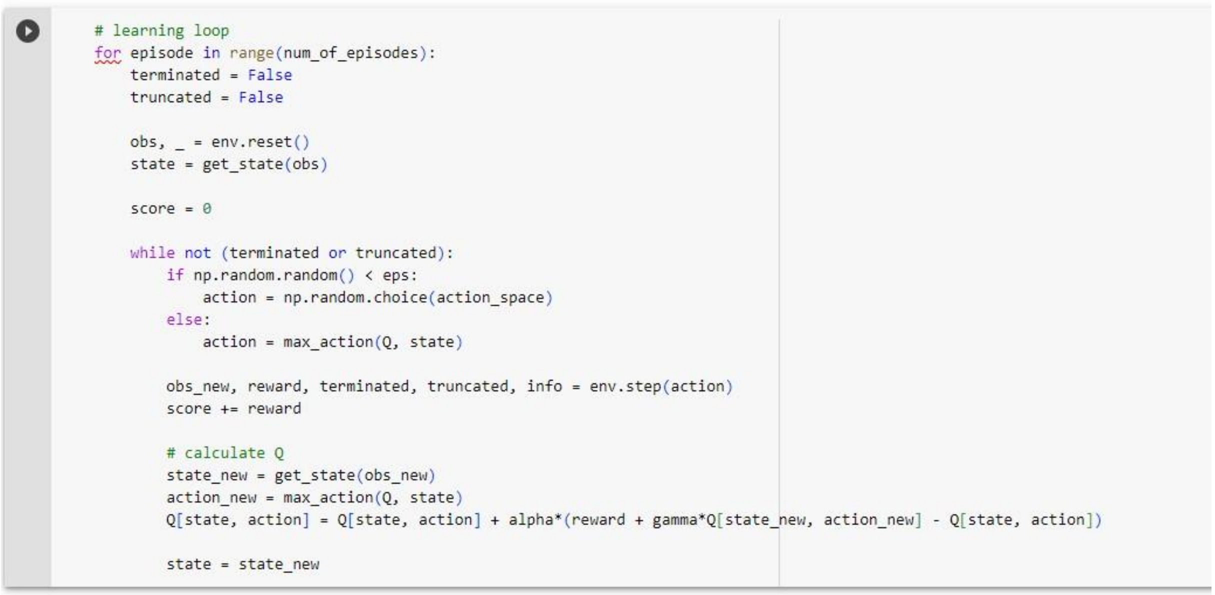
```python
[18]    states = list()
        Q = {}

        scores = np.zeros(num_of_episodes)
        epsilons = np.zeros(num_of_episodes)

        for position in range(pos_bucket_numer + 1):
            for velocity in range(vel_bucket_number + 1):
                states.append((position, velocity))

        for state in states:
            for action in action_space:
                Q[state, action] = 0
```

This loop initializes the Q-values in the dictionary Q with a value of 0 for each state-action pair.

This code snippet prepares an empty dictionary Q to store the Q-values and creates the list of possible states in the environment along with their initial values.

```python
[21]  # create progress bar
     progress_bar = tqdm(total=num_of_episodes, desc='Learning')
```

This progress bar is a tool used to track the progress of the learning process in the reinforcement learning algorithm.



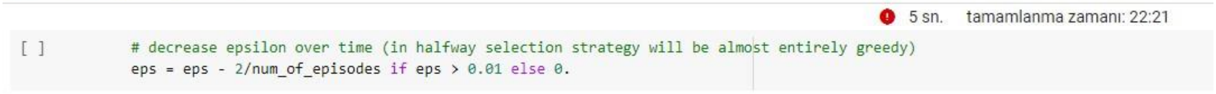
```python
# learning loop
for episode in range(num_of_episodes):
    terminated = False
    truncated = False

    obs, _ = env.reset()
    state = get_state(obs)

    score = 0

    while not (terminated or truncated):
        if np.random.random() < eps:
            action = np.random.choice(action_space)
        else:
            action = max_action(Q, state)

        obs_new, reward, terminated, truncated, info = env.step(action)
        score += reward

        # calculate Q
        state_new = get_state(obs_new)
        action_new = max_action(Q, state)
        Q[state, action] = Q[state, action] + alpha*(reward + gamma*Q[state_new, action_new] - Q[state, action])

        state = state_new
```

The terminated variable represents the condition where the car has reached the goal and the episode is completed successfully. If the car reaches the top of the hill and accomplishes the task, the terminated value becomes True.

The truncated variable is used when the episode has a maximum step limit (num_of_steps_per_episode). If the car fails to reach the goal or satisfy other termination conditions within the maximum step limit, the truncated value becomes True. In this case, the episode ends, but it is not considered a successful completion.

Q[state_new, action_new] - Q[state, action]

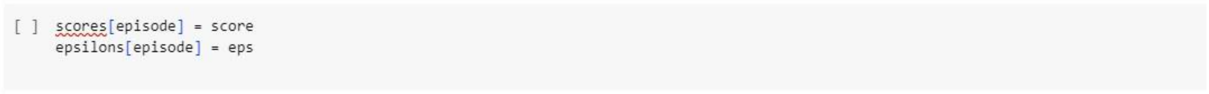
we had calculated the error



5 sn.   tamamlanma zamanı: 22:21

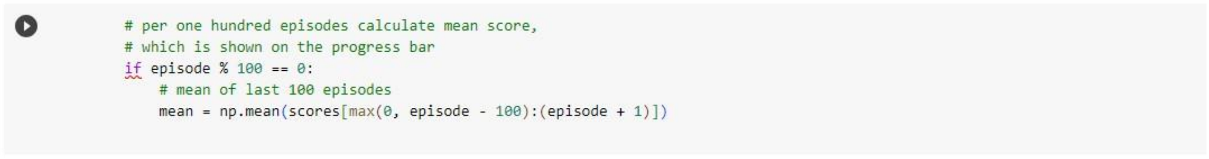
```python
[ ]     # decrease epsilon over time (in halfway selection strategy will be almost entirely greedy)
        eps = eps - 2/num_of_episodes if eps > 0.01 else 0.
```

It decreases the value of epsilon over time. This allows the agent to focus more on making accurate predictions by reducing the exploration rate. Here, the epsilon value is decreased by 2/num_of_episodes in each iteration. However, if the epsilon value is less than 0.01, it is not decreased further and the minimum value of 0.01 is used.

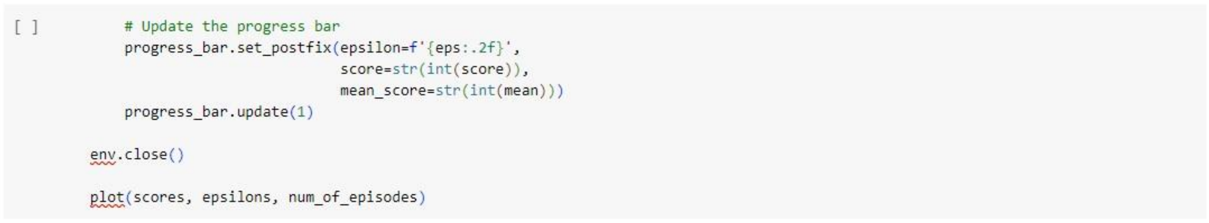


```python
[ ] scores[episode] = score
    epsilons[episode] = eps
```

It saves the score and epsilon values obtained for each episode in the 'scores' and 'epsilons' arrays, respectively. These arrays can be used to track the progress of the learning process and visualize the results.



```python
    # per one hundred episodes calculate mean score,
    # which is shown on the progress bar
    if episode % 100 == 0:
        # mean of last 100 episodes
        mean = np.mean(scores[max(0, episode - 100):(episode + 1)])
```

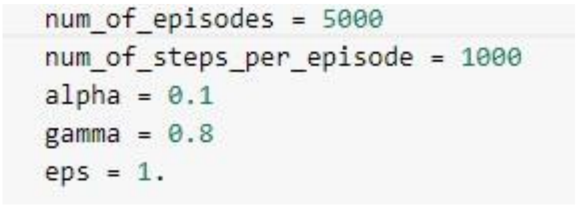


```python
[ ]
    # Update the progress bar
    progress_bar.set_postfix(epsilon=f'{eps:.2f}',
                             score=str(int(score)),
                             mean_score=str(int(mean)))
    progress_bar.update(1)

env.close()

plot(scores, epsilons, num_of_episodes)
```

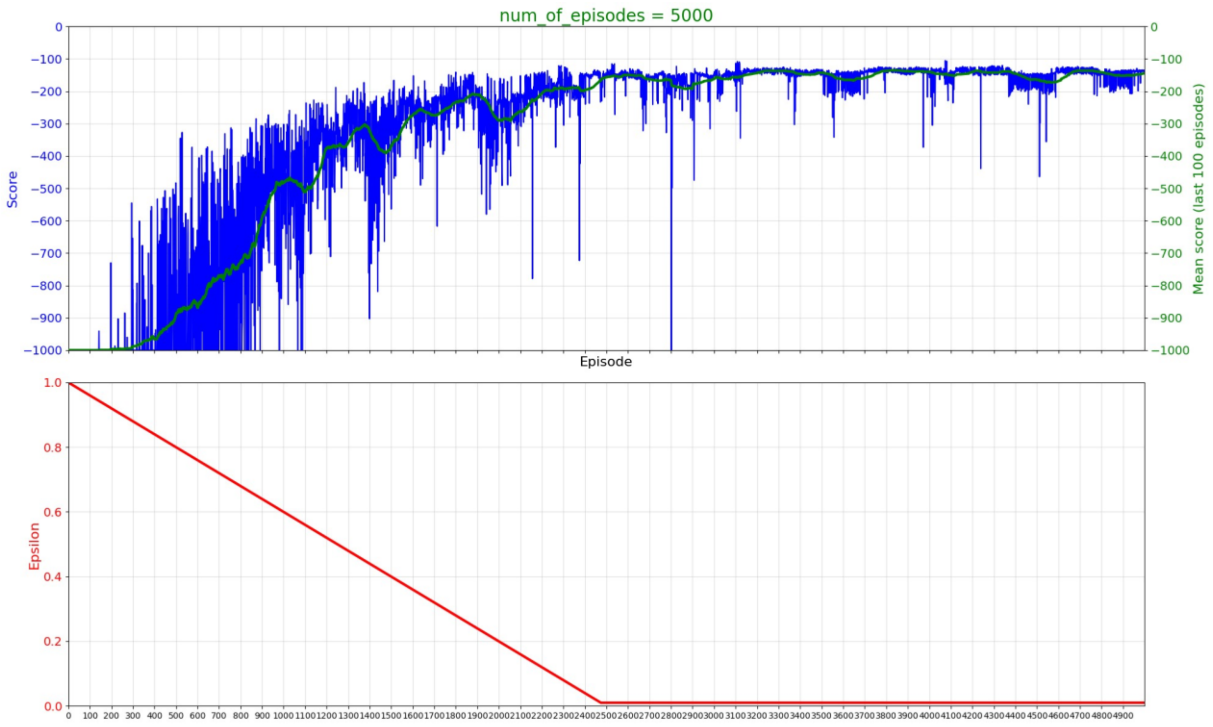
It updates and advances the progress bar.

After completing the learning loop, it closes the environment.

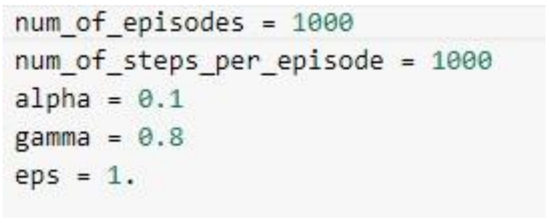
RESULTS AND VISUALIZATION



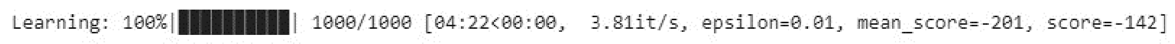
```
num_of_episodes = 5000
num_of_steps_per_episode = 1000
alpha = 0.1
gamma = 0.8
eps = 1.
```



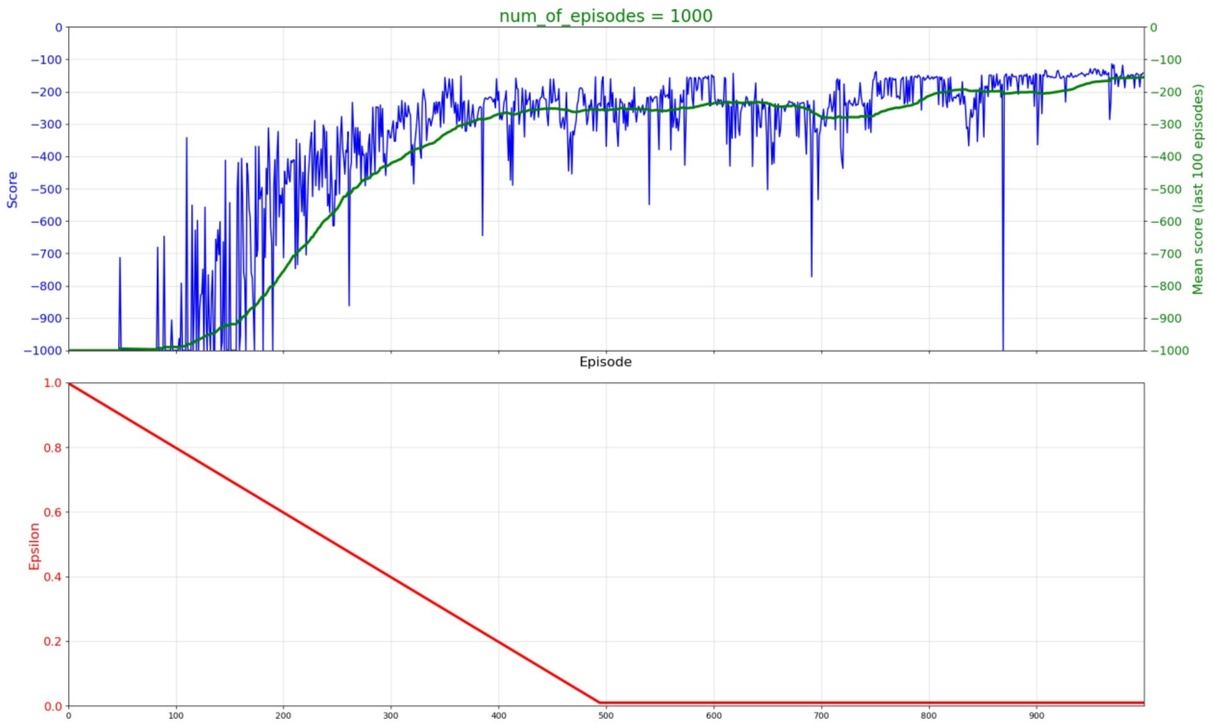
Initially, it started with a very large number of episodes and as it can be seen from the graph, after 2500 iterations became equal to epsilon 0. In fact, it was observed that we did not need such a large value. Moreover, the converge time was very long, as it can be understood from it/s.



```
num_of_episodes = 1000
num_of_steps_per_episode = 1000
alpha = 0.1
gamma = 0.8
eps = 1.
```

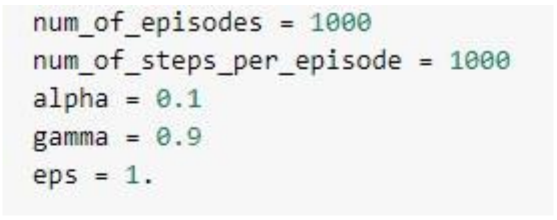


```
Learning: 100%|██████████| 1000/1000 [04:22<00:00,  3.81it/s, epsilon=0.01, mean_score=-201, score=-142]
```

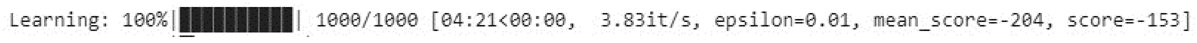


We reduced the number of episodes, and as a result, the score was worse. However, the processing speed was faster when obtaining the results. This was an expected outcome, as decreasing the number of episodes generally leads to faster execution.

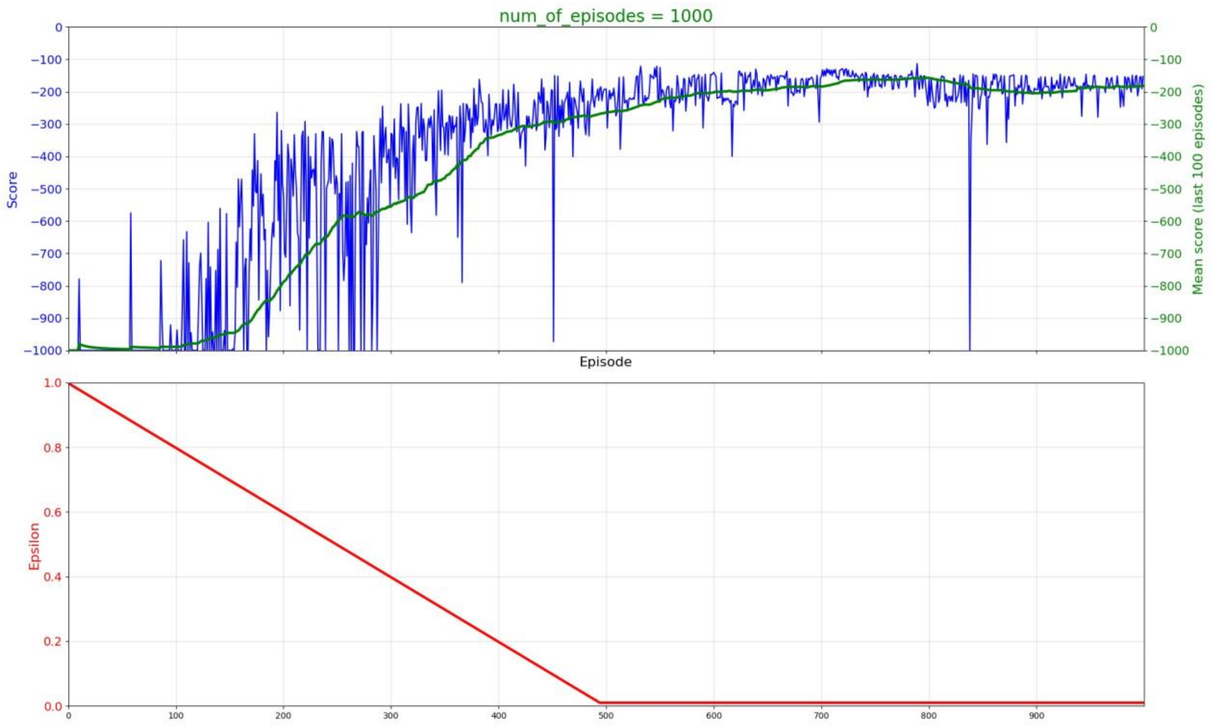
Despite the lower score, the advantage of reducing the number of episodes is the improved performance speed. Therefore, for this project, we decided to keep the number of episodes at 1000.



```
num_of_episodes = 1000
num_of_steps_per_episode = 1000
alpha = 0.1
gamma = 0.9
eps = 1.
```

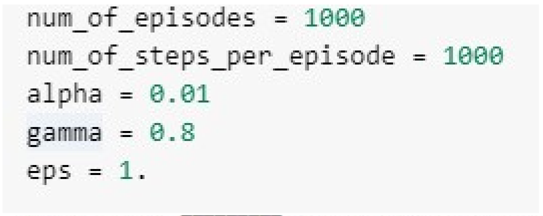


```
Learning: 100%|██████████| 1000/1000 [04:21<00:00,  3.83it/s, epsilon=0.01, mean_score=-204, score=-153]
```

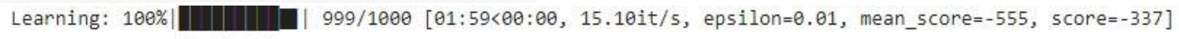


Increasing the value of gamma in the Q-learning algorithm leads to giving more importance to long-term rewards. A higher gamma value means that the agent places more emphasis on future rewards when updating its Q-values. This can result in the agent making decisions that prioritize higher cumulative rewards rather than immediate gains.

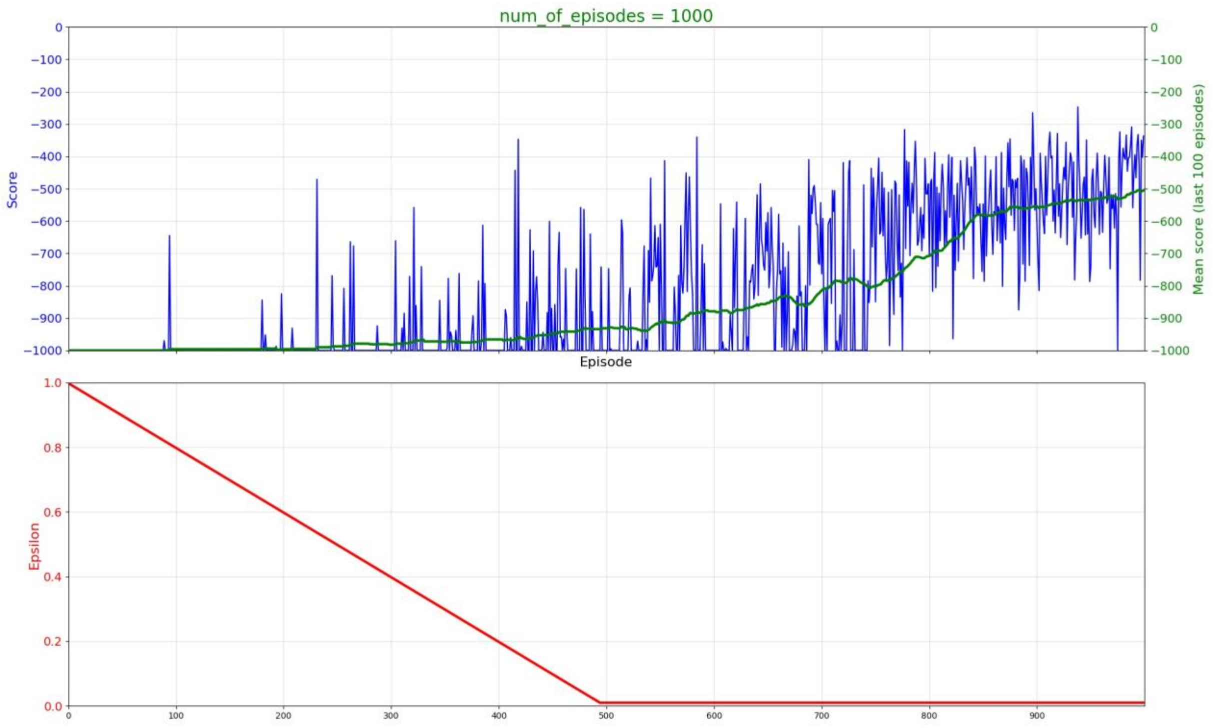
However, the results show that increasing gamma can lead to a decrease in score and ultimately lead to failure.



```
num_of_episodes = 1000
num_of_steps_per_episode = 1000
alpha = 0.01
gamma = 0.8
eps = 1.
```



```
Learning: 100%|██████████| 999/1000 [01:59<00:00, 15.10it/s, epsilon=0.01, mean_score=-555, score=-337]
```

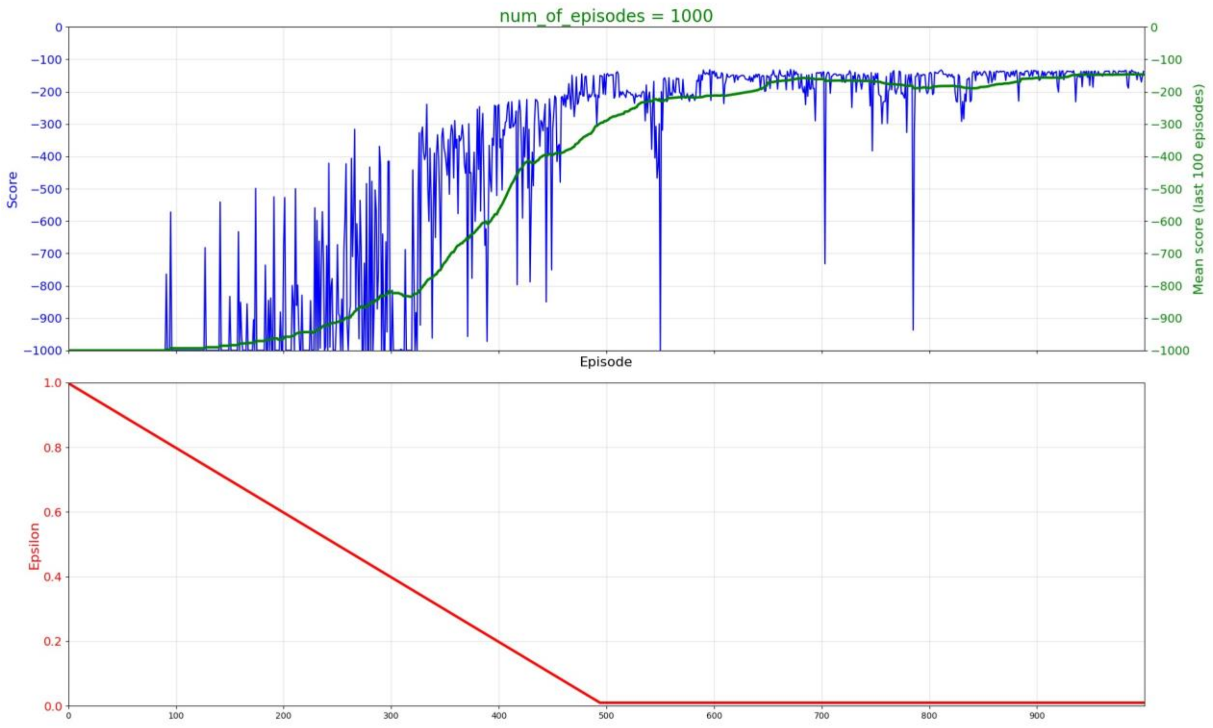


The decrease in alpha results in a slower learning rate in the Q-learning algorithm (measured in it/s). A smaller alpha value means that the agent assigns less weight to updating the Q-values. This allows the agent to learn in a more stable manner by assimilating new information at a slower pace.

However, reducing alpha too much can lead to a slower learning process and the agent may miss important information. As a result, the score and outcomes of the agent may deteriorate.

Additionally, it is interesting but when we set the alpha value to 0.5 below, the results improved significantly

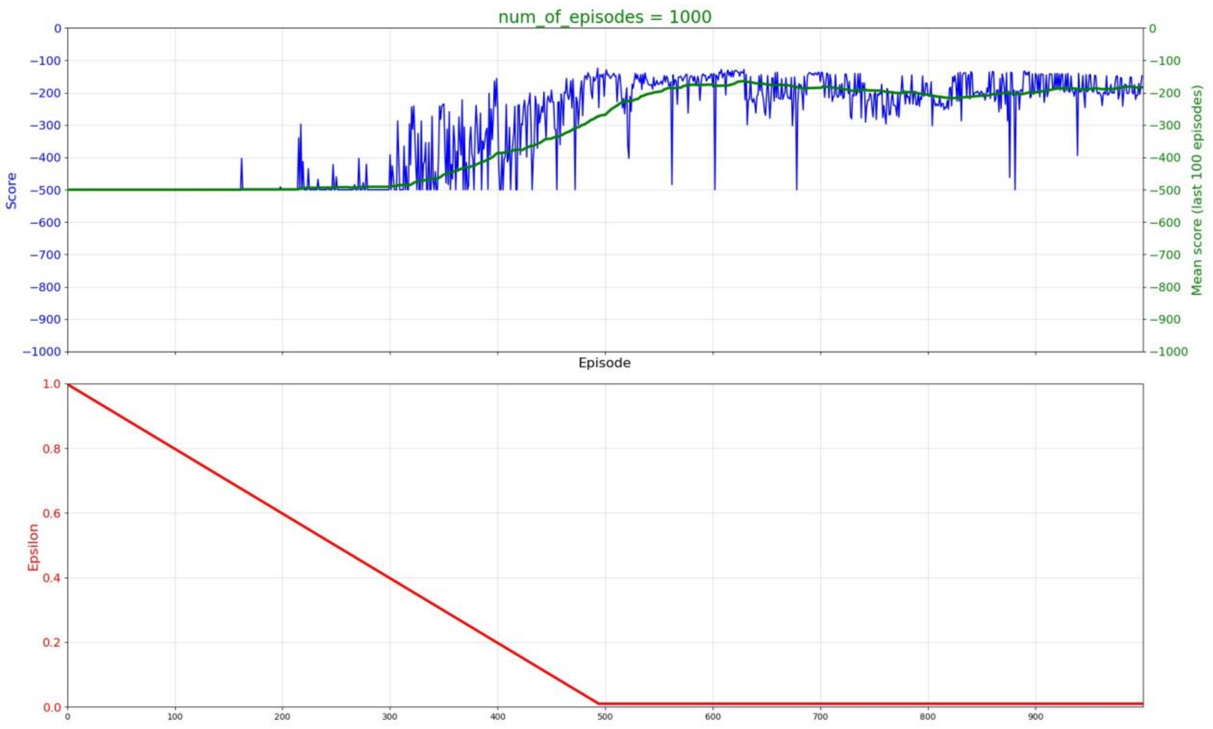
```
Learning: 100%|██████████| 999/1000 [00:55<00:00, 49.79it/s, epsilon=0.01, mean_score=-156, score=-137]
```



Indeed, it appears that when we decrease or increase the values of certain parameters, it is not possible to make a general statement that applies universally. Different parameter values may lead to different outcomes, and there is no one-size-fits-all solution. The optimal parameter values can vary depending on the specific problem or environment being addressed. It is essential to experiment with different parameter settings and carefully observe the results to determine the most suitable configuration for achieving the desired outcomes.

```
Learning: 100%|██████████| 999/1000 [00:54<00:00, 30.02it/s, epsilon=0.01, mean_score=-199, score=-148]
```

```
num_of_episodes = 1000
num_of_steps_per_episode = 1000
alpha = 0.1
gamma = 0.9
eps = 1.
```



Reducing the number of num_of_steps_per_episode reduces the number of steps in each segment. In this case, the agent tries to reach the target by taking fewer steps. It increased the processing speed, but the score still came up with a lower value.

Many attempts were made, but the desired success could not be achieved in achieving a positive score and being successful. Perhaps it would be more useful to use more precise parameters. However, we had the opportunity to observe that the general information for our future projects did not always give the same results.