# Machine Learning Report

Davide Fortunato 1936575

September 2024

## 1 Brief overview of Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique that trains agents to make decisions in an environment to maximize cumulative rewards. Unlike supervised learning, which relies on labeled data, reinforcement learning agents learn through trial and error, interacting with the environment and receiving feedback in the form of rewards or punishments. The goal of RL is to find an optimal policy, a function that maps states to actions. Given a state, the policy determines the best action to take to achieve the desired outcome, typically maximizing long-term rewards. Regarding this, two fundamental concepts in RL are:

- Q-function (Q(s, a)): Estimates the expected future reward for taking action a in state s and following an optimal policy thereafter. It measures the long-term value of an action in a given state.

$$\hat{Q}_n(s,a) = (1-\alpha)\hat{Q}_{n-1}(s,a) + \alpha\left[r + \gamma \max_{a'} \hat{Q}_{n-1}(s',a')\right]$$

- Value function (V(s)): Estimates the expected future reward starting from state s and following an optimal policy. It represents the overall worth or utility of a state.

The Q-function and value function are closely related. In fact, the value function can be derived from the Q-function by taking the maximum Q-value over all possible actions in a given state:

$$V(s) = \max_a Q(s,a)$$

By learning the Q-function or value function, RL agents can develop policies that effectively navigate complex environments and achieve their objectives.

## 2 Problem addressed

In this project, I addressed the challenge of implementing a Reinforcement Learning (RL) agent to solve the Mountain Car problem, a classic example of a deterministic Markov Decision Process (MDP). The scenario involves a car that starts at a random position at the bottom of a sinusoidal valley and the agent's task is to control the car by applying forces in either the left or right direction, with the ultimate goal of driving the car up the hill on the right side to reach a designated goal state. There are two versions of the Mountain Car problem available in the Gymnasium library: one featuring a discrete action space and the other a continuous action space. In this implementation, the **discrete version** is utilized, where the agent has a limited set of actions.

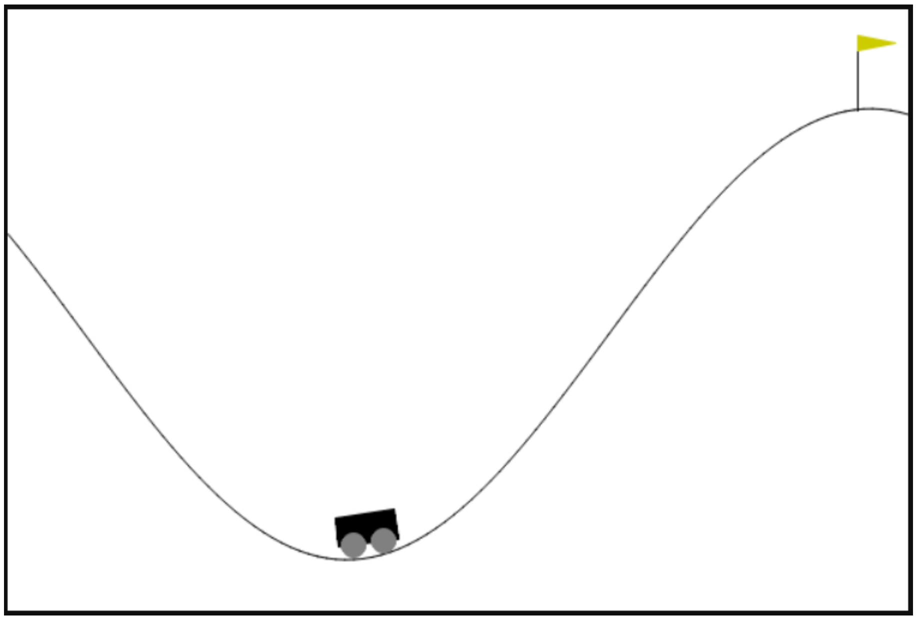

Figure 1: Rendered Gymnasium Mountain Car environment

More specifically, the state of the car is defined by two key variables: its **position** along the x-axis, whose range is [-1.2, 0.6] and its **velocity**, which can take any value in the range [-0.07, 0.07]. The position indicates where the car is located within the valley, while the velocity reflects the speed and direction of its movement. At the beginning, the position of the car is assigned a uniform random value in between [-0.6, -0.4] and the starting velocity of the car is always assigned to 0. At each step, the agent must select one of these 3 actions: accelerate to the left, accelerate to the right, or do not accelerate. Given an action, the mountain car follows the following **transition dynamics**:

$$\begin{aligned}\text{velocity}_{t+1} &= \text{velocity}_t + (\text{action} - 1) \cdot \text{force} - \cos(3 \cdot \text{position}_t) \cdot \text{gravity} \\ \text{position}_{t+1} &= \text{position}_t + \text{velocity}_{t+1}\end{aligned}$$

where force = 0.001 and gravity = 0.0025. The collisions at either end are inelastic with the velocity set to 0 upon collision with the wall. The position is clipped to the range [-1.2, 0.6] and velocity is clipped to the range [-0.07, 0.07]. The goal is to reach the flag placed on top of the right hill as quickly as possible, as such the agent is penalised with a **reward** of -1 for each timestep. The episode ends when either the car's position reaches or exceeds 0.5 (**termination**) or the episode length reaches 200 steps (**truncation**).

## 3 Solution adopted

In order to solve the environment I applied two distinct approaches to Q-learning: Tabular Q-learning and Deep Q-Network (DQN) in its simplest form. The difference between these two approaches lies in the way they want to get the optimal Q-Function: in the tabular approach, we mantain a table with up to as many table entries as the number of (state,action) pairs and the algorithm proceeds as follows:

**Algorithm 1** Tabular Q-learning Algorithm

```
1: Initialize Q-values arbitrarily: $Q(s,a) \forall s \in S, a \in A$ (often initialized to 0)
2: Set learning rate $\alpha \in (0,1]$ and discount factor $\gamma \in [0,1)$
3: for each episode do
4:     Initialize the starting state $s$
5:     while not terminal do
6:         Choose action $a$ from state $s$ using an $\epsilon$-greedy policy:
               $a = \begin{cases} \text{random action with probability } \epsilon \\ \arg\max_a Q(s,a) \text{ with probability } 1-\epsilon \end{cases}$
7:         Take action $a$, observe reward $r$ and next state $s'$
8:         Update the table entry Q(s,a) using the update rule:
               $Q(s,a) \leftarrow Q(s,a) + \alpha \left[ r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$
9:         Set the current state $s \leftarrow s'$
10:    end while
11: end for
```

On the other hand, the DQN approach uses a single neural network to approximate the Q-function. This means that the network learns to predict Q-values for each action by minimizing the difference between predicted and target Q-values. A key feature of DQN is the experience replay buffer, which stores past interactions with the environment in the form of transitions (state, action, reward, and next state). Instead of learning from consecutive experiences, which can be highly correlated, the agent randomly samples mini-batches from the buffer. This random sampling breaks the temporal correlation between experiences, allowing the network to learn more efficiently and reducing the risk of instability during training.

**Algorithm 2** Deep Q-Network (DQN) with Experience Replay

```
1: Initialize Q-network with random weights $\theta$
2: Initialize target Q-network with weights $\theta^- = \theta$
3: Initialize experience replay buffer $D$ (capacity $N$)
4: Set learning rate $\alpha \in (0,1]$ and discount factor $\gamma \in [0,1)$
5: for each episode do
6:     Initialize the starting state $s$
7:     for each step in the episode do
8:         Choose action $a$ from state $s$ using an $\epsilon$-greedy policy:
               $$a = \begin{cases} \text{random action with probability } \epsilon \\ \arg\max_a Q(s,a;\theta) \text{ with probability } 1-\epsilon \end{cases}$$
9:         Take action $a$, observe reward $r$ and next state $s'$
10:        Store the transition $(s,a,r,s')$ in the experience replay buffer $D$
11:        Sample a mini-batch of transitions $(s_j,a_j,r_j,s'_j)$ from $D$
12:        Set the target for each transition:
               $$y_j = \begin{cases} r_j \quad \text{if } s'_j \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s'_j,a';\theta^-) \quad \text{otherwise} \end{cases}$$
13:        Update the Q-network weights by minimizing the loss
14:        Set the current state $s \leftarrow s'$
15:    end for
16: end for
```

## 4 Experimental results

In this section, I will detail the various attempts made to solve the environment using both the Q-table and DQN approaches. Before diving into the specifics, let's first define the key hyperparameters that characterize each approach:

- **Bins**: Discrete intervals used to simplify continuous state spaces, allowing the algorithm to work with manageable state representations in Q-learning.
- **Gamma ($\gamma$)**: The discount factor, which determines the importance of future rewards. A value close to 1 emphasizes long-term rewards, while a value near 0 focuses on immediate rewards.
- **Learning Rate ($\alpha$)**: Controls how much new information overrides the old in updating the Q-values. A higher learning rate makes the model adapt faster but might lead to instability.

- **Episodes**: The number of complete game rounds or trials the agent goes through during training. Each episode starts from an initial state and continues until the environment reaches a terminal state.
- **Epsilon** ($\epsilon$): The exploration rate in the $\epsilon$-greedy policy, which dictates how often the agent explores new actions rather than exploiting known ones.
- **Epsilon Decay Rate**: The rate at which $\epsilon$ decreases over time, encouraging the agent to explore less and exploit more as it becomes more confident in its policy.
- **Batch Size**: In DQN, this refers to the number of experiences (state-action-reward-next state transitions) sampled from memory to update the network in a single training step.

## 4.1 Q-Table

### 4.1.1 First attempt

The first try was run with the following hyperparameters:

- **Gamma** ($\gamma$) = 0.99
- **Learning Rate** ($\alpha$) = 0.005
- **Episodes** = 10000
- **Bins** = 20
- **Epsilon Decay Rate** = 0.99986

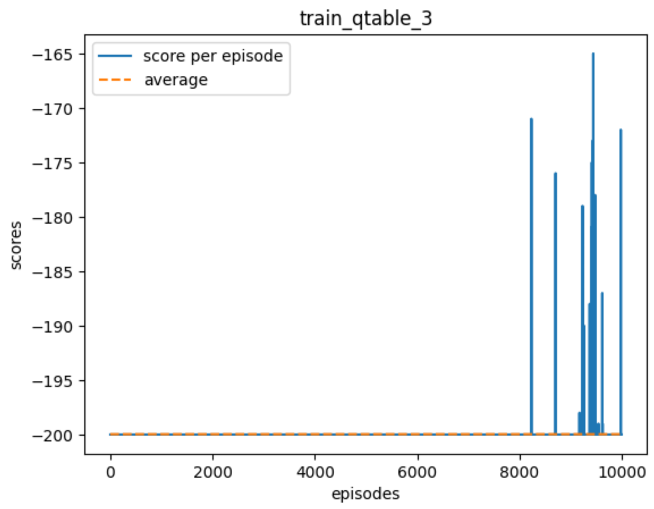

Figure 2: First attempt train scores Q-Table

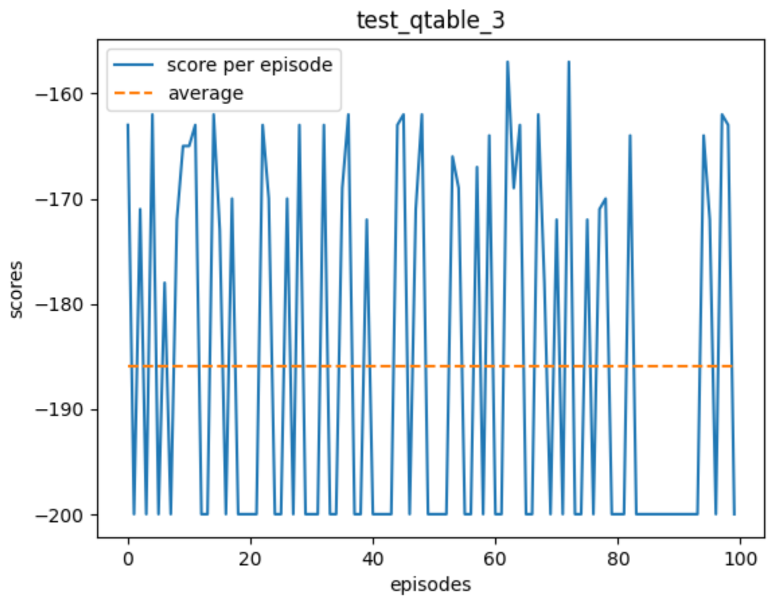

Figure 3: First attempt test scores Q-Table

The initial attempt did not solve the environment, as the car successfully reached the top of the mountain in only 42 out of 100 test episodes. The main issue with this approach was the insufficient number of episodes, which limited the agent's ability to explore and optimize the Q-Table. Recall that an episode is considered successful when the score is less than 200, meaning the agent reaches the top in fewer than 200 steps. That's why one of the adjustments in the hyperparameters in the next attempt was incrceasing the number of episodes.

#### 4.1.2 Second attempt

- **Gamma** ($\gamma$) = 0.95
- **Learning Rate** ($\alpha$) = 0.01
- **Episodes** = 50000
- **Bins** = 30
- **Epsilon Decay Rate** = 0.999976

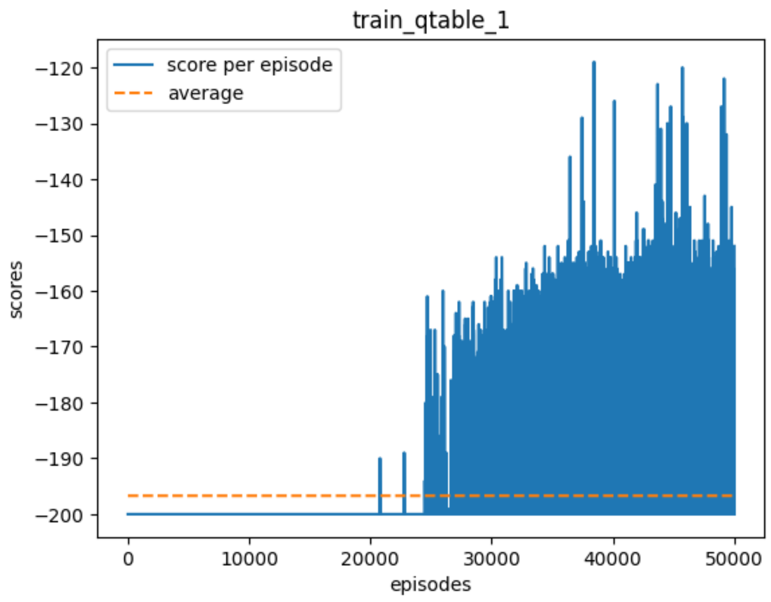

Figure 4: Second attempt train scores Q-Table

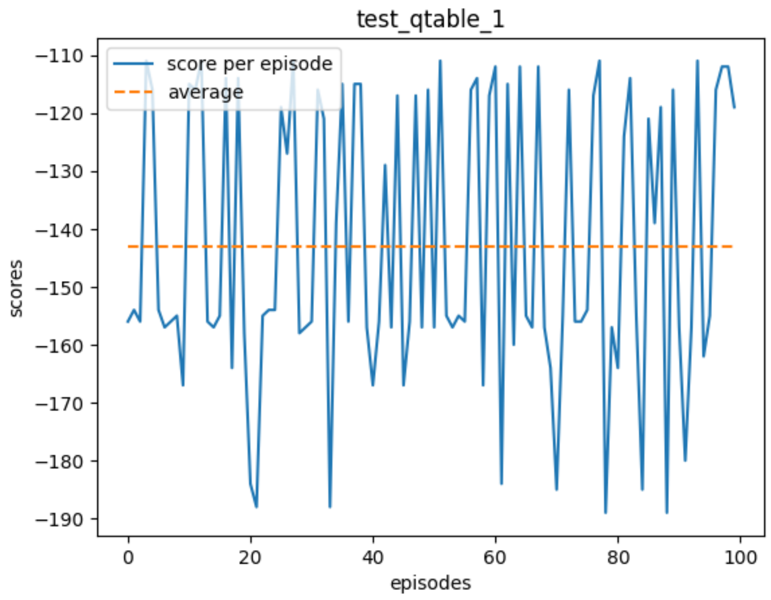

Figure 5: Second attempt test scores Q-Table

In this attempt, we can observe that from the 20000th episode onward, the agent consistently starts reaching the top of the mountain. This indicates that it took approximately 20,000 episodes of exploration to sufficiently optimize the Q-Table. Furthermore, the agent demonstrates a strong understanding of how to solve the task, as it achieves a score of less than 200 in all 100 test episodes, indicating successful completion within the step limit. But there is still room for a little improvement, let's see why.

### 4.1.3 Third attempt

- **Gamma** ($\gamma$) = 0.99
- **Learning Rate** ($\alpha$) = 0.005
- **Episodes** = 50000
- **Bins** = 20
- **Epsilon Decay Rate** = 0.99986

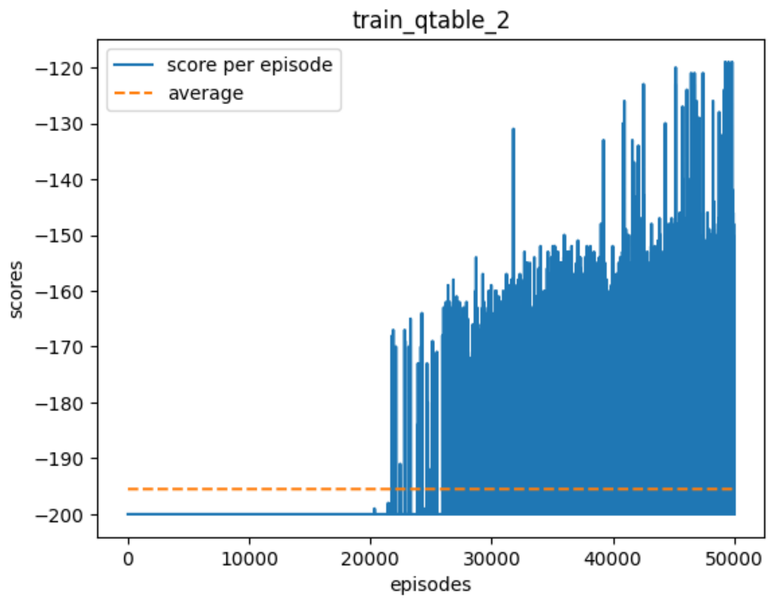

Figure 6: Third attempt train scores Q-Table

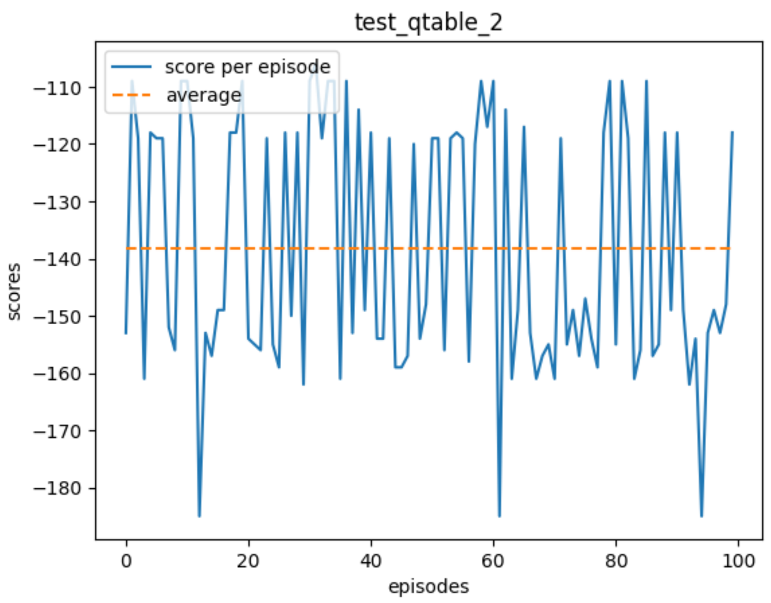

Figure 7: Third attempt test scores Q-Table

Here we have the same hyperparameters of the first try, except for the number of episodes which I kept equal to 50000 in order to let the agent optimize the q-table by performing more exploration actions. In this case we can see that the average score in the test episodes is better with respect to the second attempt since the agent performs an average score of bit more than -140, always mantaining 100 successes out of 100 trials

## 4.2 Deep-Q-Network

### 4.2.1 First attempt

The first attempt was, in some way, trying to simulate the same idea of the q-table, i.e. making a huge amount of episodes and hoping that network weights will be optimized but unfortunately it didn't work out. In fact, without adjusting the reward and with a single hidden layer, neither after 30000 episodes the agent was able to make even just a success in the test episodes. In particular let's see the hyperparameters:

- **Gamma** ($\gamma$) = 0.95
- **Learning Rate** ($\alpha$) = 0.01
- **Episodes** = 28000
- **Batch Size** = 32

- **Epsilon Decay Rate** = 0.999957
- **Hidden layers** = 1

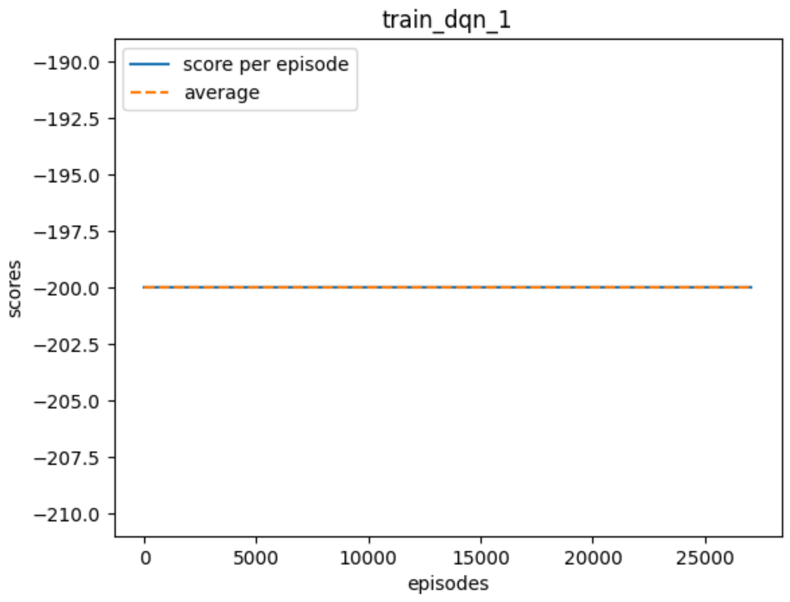

Figure 8: First attempt train scores DQN

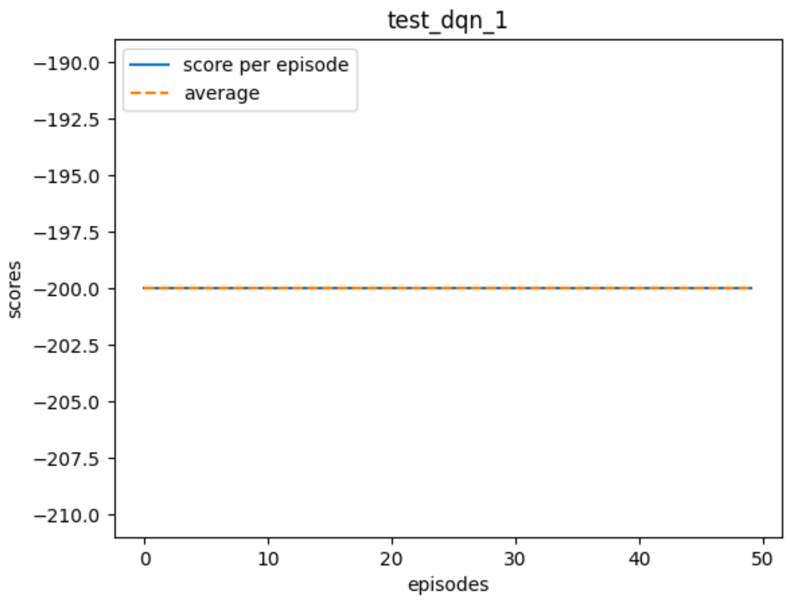

Figure 9: First attempt test scores DQN

Unfortunately, the approach without reward adjustment did not work in my case, or it might have worked but would have required a significantly larger number of episodes, resulting in a much longer training time. Let's see in the next section how I have dealt with this problem.

### 4.2.2 Second attempt

Instead of assigning just a reward of -1 for each step, I tried different ways of adjusting the reward (for instance, one way was assigning a positive reward for those actions that lead to a velocity increase or decrease) and the best one was the following:

```
if new_state[0][0] - state[0][0] > 0 and action == 2:
    reward = reward + 3
if new_state[0][0] - state[0][0] < 0 and action == 0:
    reward = reward + 3
if terminated:
    reward = 50
```

where `new_state[0][0] - state[0][0]` compares the car's position between two consecutive time steps, indicating whether the car is moving right (positive value) or left (negative value) and action equal to 0 or 2 refers respectedly to move left or move right. In this way I gave a little reward if the car keeps moving according to its momentum and finally, when the car reaches the top, I assign a big reward equal to 50.

- **Gamma** ($\gamma$) = 0.99
- **Learning Rate** ($\alpha$) = 0.005
- **Episodes** = 2500
- **Batch Size** = 32
- **Epsilon Decay Rate** = 0.99954
- **Hidden layers** = 1

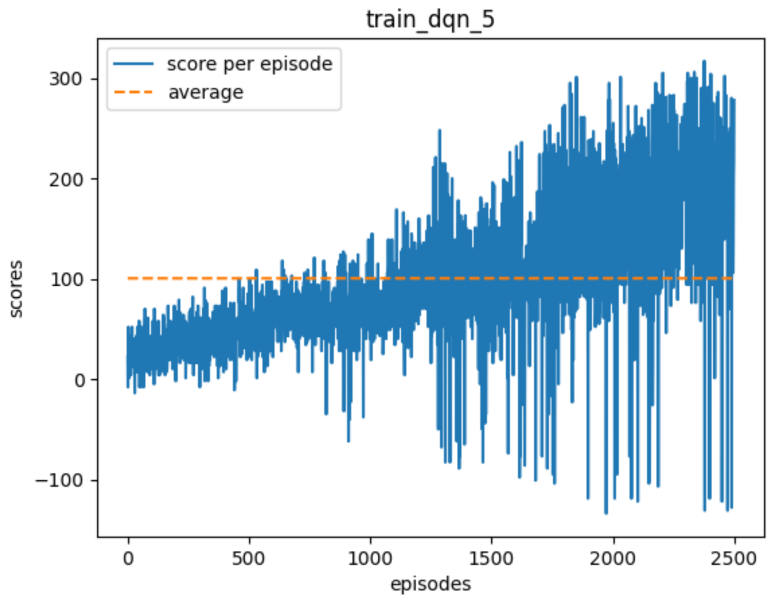

Figure 10: Second attempt train scores DQN

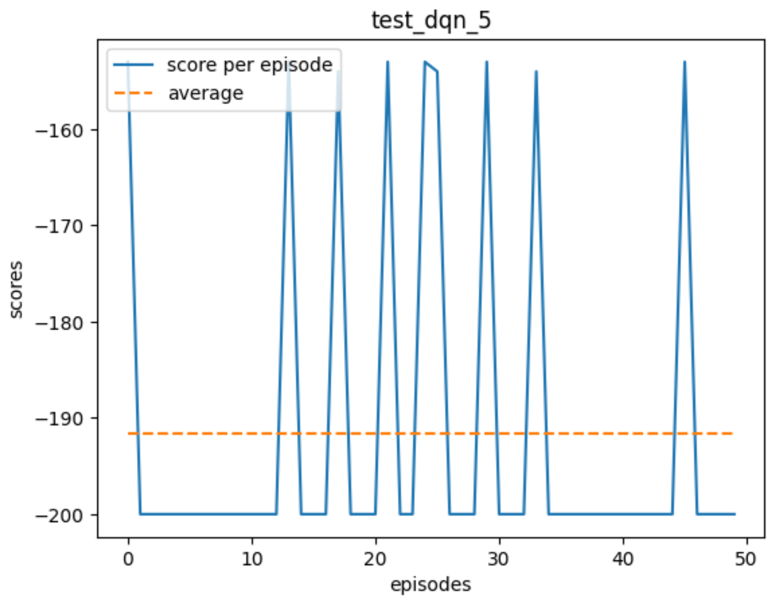

Figure 11: Second attempt test scores DQN

The reward adjustment has partially improved the agent's performance, as it now successfully reaches the top of the mountain in 9 test episodes. In the next subsection, we will examine the impact of adding a new hidden layer and Chapter 5 will provide a more detailed overview of the neural network's final architecture.

### 4.2.3 Third attempt

- **Gamma** ($\gamma$) = 0.99
- **Learning Rate** ($\alpha$) = 0.005
- **Episodes** = 2500
- **Batch Size** = 32
- **Epsilon Decay Rate** = 0.99954
- **Hidden layers** = 2

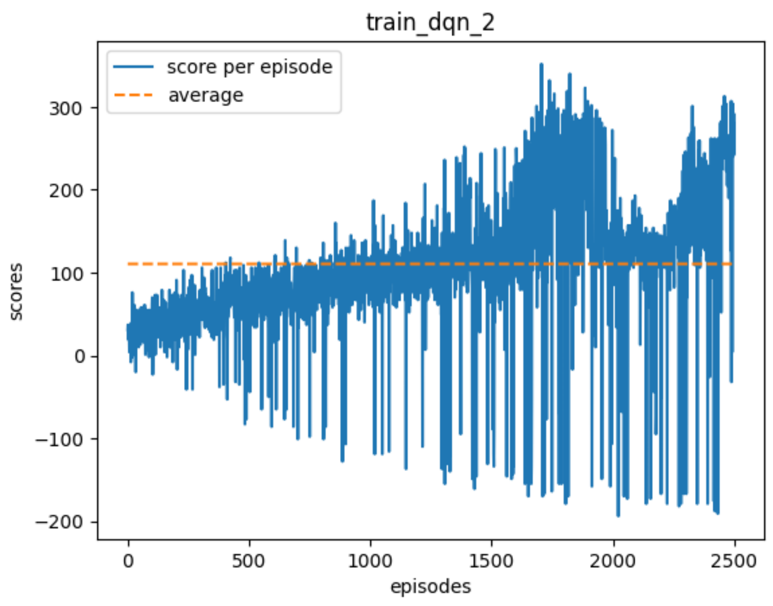

Figure 12: Third attempt train scores DQN

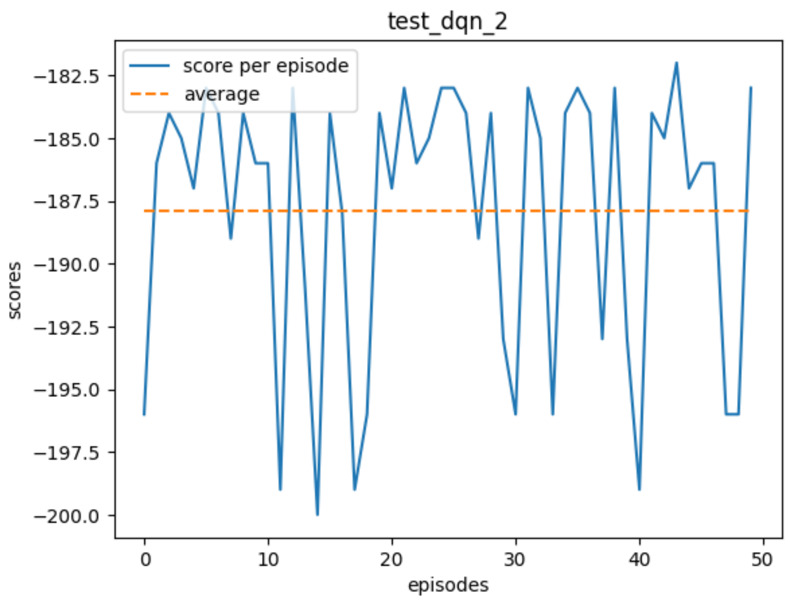

Figure 13: Third attempt test scores DQN

The addition of a second hidden layer not only slightly improved the average score in the 2500 train episodes, but more importantly it let the agent reach the top of the mountain in 49 out of the 50 test episodes.

## 5 Architecture of the implementation

### 5.1 Q-Table

The first important aspect to consider is how I implemented the Q-Table. To do this, I used the Python library NumPy, which is a powerful tool for numerical computing. It provides support for working with large arrays and matrices, along with a variety of mathematical functions to operate on these structures efficiently. In this case, I used the following instruction to initialize the Q-Table:

```
 q_table = np.random.uniform(low=0, high=0,
                    size=(DISCRETE_OS_SIZE + [env.action_space.n]))
```

The instruction uses NumPy's random.uniform function to initialize the Q-Table with random values. However, since both the low and high parameters are set to 0, the Q-Table will be filled with zeros. The size argument defines the shape of the Q-Table. It is set to (DISCRETE_OS_SIZE + [env.action_space.n]), where DISCRETE_OS_SIZE represents the discretized observation space, and env.action_space.n corresponds to the number of possible actions in the environment. This ensures that the Q-Table has an entry for every possible state-action pair in the environment.

Another important aspect to consider is the method used to discretize the state at each step of the algorithm described in Chapter 3. This process is crucial because it transforms the continuous state space into a finite, manageable set of discrete states.

```
num_bins = 20
discrete_space_size = [num_bins] * len(env.observation_space.high)
bin_size = (env.observation_space.high - env.observation_space.low) /
                                        discrete_space_size
def get_discrete_state(continuous_state):
    discrete_state = (continuous_state - env.observation_space.low) /
                                            bin_size
    return tuple(discrete_state.astype(np.int32))
```

This code converts continuous states from the environment into discrete states by dividing the continuous observation space into a fixed number of bins and determining which bin each dimension of the state belongs to. This discretized representation allows the Q-learning algorithm to work in an otherwise continuous environment.

Other key components of the architecture are the training and testing loop, which by the way I've already discussed in detail in Chapter 3 and therefore won't elaborate on here. Additionally, to track the agent's performance over time and generate the charts included in this report, I used a module called ScoreLogger. This class is responsible for logging the agent's scores during both the training and testing phases, allowing for performance analysis and visual representation of the learning progress.

## 5.2 Deep-Q-Network

The first aspect I want to highlight is the architecture of the neural network that enabled successful performance in the environment using the Deep Q-Network (DQN) approach. The architecture consists of a four-layer neural network, designed as follows:

1. **Input layer**: The network's input layer has 2 neurons, corresponding to the two state variables of the agent: position and velocity. These inputs represent the current state of the environment.
2. **Hidden layers**
    - **First hidden layer**: This layer contains 24 neurons with ReLU (Rectified Linear Unit) activation. This layer helps the network learn intermediate features of the state by applying non-linearity.
    - **Second hidden layer**: Comprising 48 neurons and also using ReLU activation, this layer builds on the features extracted by the first hidden layer, allowing the network to capture more complex patterns and interactions between state variables.
3. **Output layer**: The final layer has 3 neurons, each representing a possible action that the agent can take: move left, stay still, or move right. This layer uses a linear activation function, outputting the Q-values for each action, which the network uses to make decisions.

This architecture was implemented using the Keras framework, which facilitated the development and training of the neural network. Keras provides an intuitive and efficient interface for building and compiling deep learning models, making it straightforward to define a multi-layered architecture, compile it with the Adam optimizer, and incorporate mean squared error as the loss function to guide the model's learning process. Let's take a look at the code part where the model is defined.

```
def build_model(self):
    model = Sequential()
    model.add(Dense(24, input_dim=self.state_size,
            activation='relu'))
    model.add(Dense(48, activation='relu'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mean_squared_error',
        optimizer=Adam(learning_rate=self.learning_rate))
    return model
```

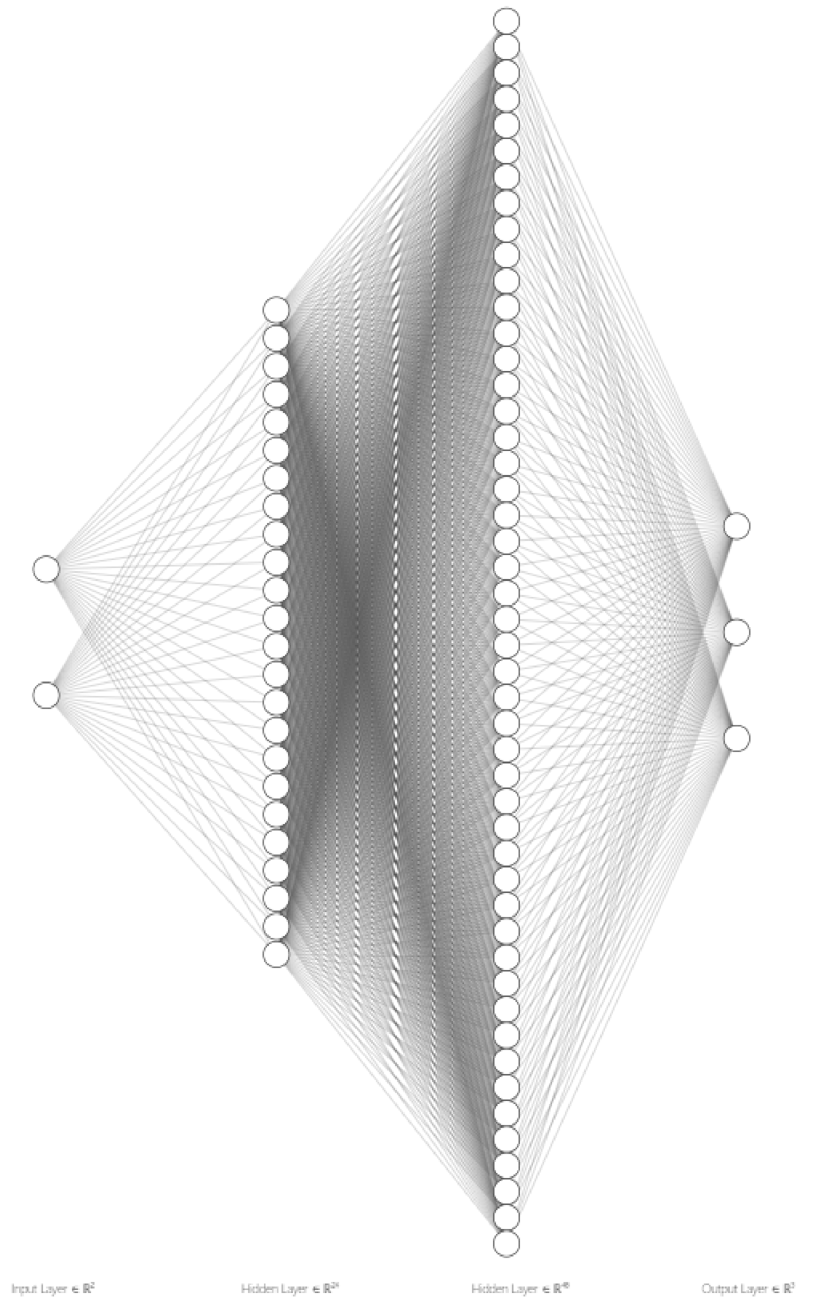
Input Layer $\in \mathbb{R}^2$
Hidden Layer $\in \mathbb{R}^{24}$
Hidden Layer $\in \mathbb{R}^{48}$
Output Layer $\in \mathbb{R}^3$

Figure 14: Neural network visualization

With respect to the Q-Table approach, a new component of the implementation is the replay buffer. In reinforcement learning, a replay buffer (also known as an experience buffer) is a memory storage that holds the agent's experiences during training. These experiences typically are stored as tuples (state, action, reward, new_state, done). Instead of using each experience immediately, the agent can later sample a batch of experiences randomly from this buffer to update the Q-values. This mechanism helps the agent break the correlation between sequential experiences and improves the efficiency of learning by reusing past experiences. Let's see the code related to this part:

```
def replay(self, episode):
    minibatch = random.sample(self.replay_buffer, self.batch_size)
    for new_state, reward, terminated, state, action in minibatch:
        target = reward
        if not terminated:
            target += self.gamma * np.max(self.model.predict(new_state)[0])
        target_function = self.model.predict(state)
        target_function[0][action] = target
        self.model.fit(state, target_function, epochs=1, verbose=0)
    if self.epsilon > self.min_eps:
        self.epsilon *= self.eps_decay
```

This method is invoked during the training loop once the replay buffer has accumulated experiences equal to the hyperparameter *batch_size*. After that, it is called at the end of each episode to update the agent's policy using a randomly sampled minibatch from the buffer. I reported here just the different components of this architecture compared to the Q-Table approach, so for every detail about the code it is possible to delve it at https://github.com/davidef24/Mountain-Car-solution-Machine-Learning-Course

## 6 Conclusions

In conclusion, although it took considerable time and effort to fully grasp how to solve the Mountain Car environment using the DQN approach—particularly due to initial challenges with using only one hidden layer and a lack of effective reward adjustments—I am ultimately satisfied with the solution I developed. More importantly, this project has provided invaluable learning experiences, from fine-tuning network architectures to understanding the significance of exploration strategies and reward shaping in reinforcement learning. The insights gained during this project have deepened my understanding of both the theoretical and practical aspects of reinforcement learning.