Machine Learning Project

Riccardo Gobbato - Matricola 1918319

September 19, 2023

1 Introduction

This paper delves into an investigation that involves the comparison of two distinct reinforcement learning methods to tackle the gymnasium environment MountainCar. The first approach employs a Q-table to store Q values, which are essential for determining the best policy. Meanwhile, the second approach uses a deep neural network to approximate the Q-function.

2 MountainCar properties

According to the official documentation provided by the Gym platform, the Mountain Car Markov Decision Process (MDP) is a deterministic environment featuring a car placed somewhat unpredictably at the base of a curved valley. In this environment, the car's only available actions are to accelerate either to the left or right. The primary objective of this MDP is to strategically maneuver the car by applying accelerations to reach a specific goal state located atop the right hill.

The observations provided by the environment are represented as a two-dimensional ndarray with dimensions (2,). These observations include the car's current position along the x-axis, constrained within the range of -1.2 to 0.6, as well as its velocity, which falls within the range of -0.07 to 0.07.

Regarding the action space, there are three discrete and deterministic actions that an agent can take within this environment:

- Action 0: Accelerate the car to the left.
- Action 1: maintain the current speed.
- Action 2: Accelerate the car to the right.

In this environment, the agent faces a penalty of -1 for every time step it takes, and episodes come to an end under two conditions: either when the car successfully reaches the designated goal position or when more than 200 time steps have elapsed, resulting in a reward of -200. It's crucial to note that the agent's reward is primarily influenced by how quickly it can complete an episode.

3 Q-Table Implementation

3.1 Q-Table introduction

In reinforcement learning, a Q-table (short for Quality table or Action-Value table) is a data structure used to store and manage the estimated values of taking specific actions in specific states of an environment. Q-tables are commonly associated with tabular reinforcement learning algorithms like Q-learning. The fundamental parts are:

1. States: the environment is divided into discrete states. These states represent the different situations or configurations the agent can be in.
2. Actions: for each state, there are a set of possible actions that the agent can take.

3. Q-Values: the Q-table contains Q-values, which are estimates of the expected cumulative rewards an agent can achieve by taking a specific action in a specific state.

4. Initialization: the Q-table is often initialized with arbitrary values or set to zeros.

5. Updating Q-Values: during the learning process, the agent interacts with the environment, takes actions, and receives rewards. Based on the rewards received and the transitions between states, the Q-values in the table are updated using techniques such as the Q-learning update rule:

   $Q(s,a) = Q(s,a) \cdot (1-\alpha) + \alpha \cdot [R + y \cdot max(Q(s',a'))]$

   $\alpha \rightarrow$ the learning rate, controlling the step size of updates.
   $R \rightarrow$ the immediate reward received after taking action a in state s.
   $y \rightarrow$ the discount factor, which represents the agent's preference for immediate rewards over future rewards.
   $s' \rightarrow$ the next state after taking action a in state s.
   $a' \rightarrow$ the action chosen in the next state s'.

6. Exploration vs. Exploitation: the agent balances exploration (trying random actions to discover better ones) and exploitation (choosing actions that are believed to be the best according to the current Q-values) as it updates the Q-table.

Q-tables are effective for small sized state and action spaces. However, they become impractical for large or continuous state and action spaces. In such cases, deep neural networks are often used to approximate Q-values, leading to algorithms like Deep Q-Learning.

3.2 Code for table class

```
%pip install gymnasium[classic-control]
%pip install tensorflow
%pip install matplotlib

import gymnasium as gym
from gymnasium import wrappers
from collections import deque
from collections import defaultdict
import random
import os
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

There is first the part of code with the import and download of the needed library.

```
class table_class:
    def __init__(self, learning_rate, epsilon, env):

        self.env = env
        self.epsilon = epsilon
        self.learning_rate = learning_rate
        self.size = (env.observation_space.high -
        env.observation_space.low) / 50
        self.q_value_table = np.zeros((50,50,3))

        # state [position speed]
        # self.env.observation_space.low:[-1.2  -0.07]
        # self.env.observation_space.high:[0.6  0.07]
        # self.size:[0.036  0.0028]
```

```
def decresing_epsilon(self):
    if self.epsilon > 0.5 :
      self.epsilon = 90 * self.epsilon / 100
    else :
      self.epsilon = 98 * self.epsilon / 100

    # to have always a bit of randomness
    if self.epsilon < 0.001 :
      self.epsilon = 0.005


def decresing_learning_rate(self):
    self.learning_rate = 98 * self.learning_rate / 100


def action(self, state):
    if random.random() > self.epsilon:
        return self.best_action(state[0],state[1])
    else:
         return np.random.choice(3)


def best_action(self, state_position, state_speed):
    state_discrete_position =
    ((state_position - self.env.observation_space.low[0])
    / self.size[0]).astype(np.int64)

    state_discrete_speed =
    ((state_speed - self.env.observation_space.low[1])
    / self.size[1]).astype(np.int64)

    return np.argmax(self.q_value_table[state_discrete_position]
    [state_discrete_speed])
    # take the index of max q value (so the best action)


def discretization(self, state, next_state):
    state_discrete_position =
    ((state[0] - self.env.observation_space.low[0])
    / self.size[0]).astype(np.int64)

    state_discrete_speed =
    ((state[1] - self.env.observation_space.low[1])
    / self.size[1]).astype(np.int64)

    next_state_discrete_position =
    ((next_state[0] - self.env.observation_space.low[0])
    / self.size[0]).astype(np.int64)

    next_state_discrete_speed =
    ((next_state[1] - self.env.observation_space.low[1])
    / self.size[1]).astype(np.int64)

    return state_discrete_position, state_discrete_speed,
    next_state_discrete_position, next_state_discrete_speed
```

```
def update_table(self, state, next_state, action, reward):

    state_discrete_position, state_discrete_speed,
    next_state_discrete_position, next_state_discrete_speed =
    self.discretization(state, next_state)

    next_best_value = np.max(self.q_value_table
    [next_state_discrete_position][next_state_discrete_speed])
    * (not terminate)
    # take max q_value between the 3 actions

    # 0.95 discount factor
    self.q_value_table[state_discrete_position]
    [state_discrete_speed][action] = ( (1 - self.learning_rate) *
    self.q_value_table[state_discrete_position]
    [state_discrete_speed][action] + self.learning_rate *
    ( reward + 0.95 * next_best_value ) )
```

There are different part in this class:

- Initialization of learning rate, epsilon and env.
- Best Action: this method takes a state as input and returns an action based on the agent's current Q-values and exploration strategy, the continuous state is first discretized to find the corresponding Q-values in the Q-table, the action with the highest Q-value is selected as the policy action.
- Action: given a state, this method selects an action based on the exploitation-exploration strategy. With probability epsilon, a random action is chosen to explore the environment, and with probability 1 - epsilon, the action is selected using the policy defined by the Q-values.
- Update Table: this method is used to update the Q-values based on observed state. It takes the current observation, the chosen action, the received reward, a termination flag, and the next observation as input. The method first discretizes the observations to find the corresponding Q-values. The size used for the discretization is 50. I have initially used high numbers around 1000 but they were not appropriate.
- Decresing Epsilon.
- Decresing Learning Rate.

3.3 Code for Training

```
episodes = 150000
env = gym.make('MountainCar-v0')
env = wrappers.RecordEpisodeStatistics(env, deque_size=episodes)

reward_single_episode = deque(maxlen=100)
learning_rate = 0.1
epsilon = 0.7
win_episode = 0
episode_result=[]
reward_result=[]

if not os.path.exists("table/"):
  os.makedirs("table/")
```

```
table = table_class(learning_rate, epsilon, env)

for episode in range(1,episodes):
    state, _ = env.reset()
    step = 1
    terminate, truncate = False, False

    while not terminate and not truncate:

        action = table.action(state)
        next_state, reward, terminate, truncate, _ = env.step(action)
        table.update_table(state, next_state, action, reward)

        if terminate or truncate :

            reward_single_episode.append(step * -1)

            average_reward = sum(reward_single_episode) /
            len(reward_single_episode)

            if episode % 10 == 0:
              episode_result.append(episode)
              reward_result.append(average_reward)

            if episode % 100 == 0:
                print(f"Episode {episode},
---------------e {table.epsilon:.4f},
---------------average_reward {average_reward:.2f},
---------------position {state[0]:.4f}, speed {state[1]:.4f},
---------------learning_rate {table.learning_rate:.4f}")
            break

        state = next_state
        step+=1

    if (episode % 400 == 0 and episode > 0):
        table.decresing_learning_rate()

    if (episode % 200 == 0 and episode > 0):
        table.decresing_epsilon()

    if (episode % 1000 == 0 and episode > 0):

        np.save(f'./table/{episode}-q.npy', table.q_value_table)

        plt.plot(episode_result, reward_result)
        plt.xlabel('Episodes')
        plt.ylabel('Average Reward')
        plt.show()
```

About the code:

- Initialization of the parameters.
- The environment is wrapped to record episode statistics.
- Training Loop: for each episode, the environment is reset; within each episode, there is a loop that continues until the episode is either terminated or truncated. In each step:

- The agent selects an action based on the policy.
- The selected action is executed in the environment.
- The agent's Q-values are updated.
- The episode continues until it is truncated or terminated.

- Decresing epsilon and learning rate are called.

3.4 Code for Testing

```
env = gym.Env
episodes = 1000
load_file = "table5"
total_reward, win_episode = [], 0

#env = gym.make('MountainCar-v0', render_mode = "human")
env = gym.make('MountainCar-v0')

table = table_class(learning_rate, epsilon, env)
table.q_value_table = np.load(f'./table/{episode}.npy')

for episode in range(episodes):

    state, _ = env.reset()
    terminate, truncate, episode_reward = False, False, 0.0

    while not terminate and not truncate:
      action = table.best_action(state[0], state[1])

      next_state, reward, terminate, truncate, _ = env.step(action)
      episode_reward += reward
      state = next_state

      if next_state[0] >= 0.5:
        win_episode += 1

    average_reward.append(episode_reward)

    if episode % 10 == 0:
      print(f"Episode {episode}, Reward {episode_reward:.2f}")


mean = sum(average_reward) / len(average_reward)
accuracy = win_episode / episodes

print(f"\n\nAverage Reward: {mean:.2f}, Accuracy {accuracy:.2f}\n")
```

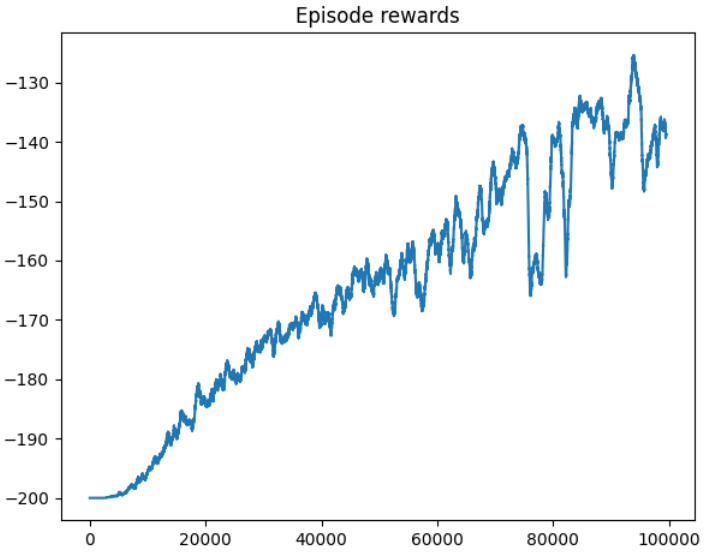
- Initialization of variables.
- Load table.
- Implementation of the loop: the agent selects an action based on its policy. The selected action is executed. The script counts the number of episodes won, based on the condition next state position >= 0.5. This condition may indicate that the agent has successfully completed an episode.
- It calculates average reward and accuracy.

4 Q-Table Results

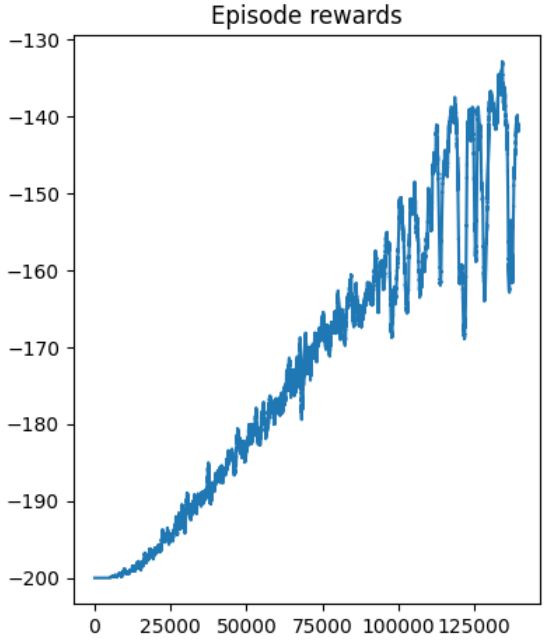
4.1 Performance Evaluation on Q-Table

I did a lot of testing before getting results. In particular I have been blocked by the implementation of the discretization: initially I have thought that it was enough to truncate (transform to integer) the position and the speed; after I have understand that I had to create a more complex method. I have also used a lot of time to find a correct size. These two things are the main problems that I have encountered during the implementation of the Q-Table. In fact, the implementation of the update rule, the code for training and testing was pretty similar to the one seen during lessons.

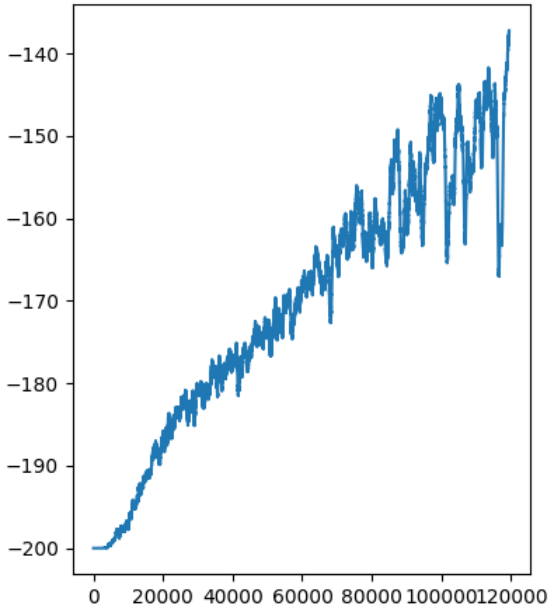
I have started setting a dynamic epsilon and a fixed learning rate.
With this parameters I have obtained an Average reward of -132.41 and an Accuracy of 1.00.



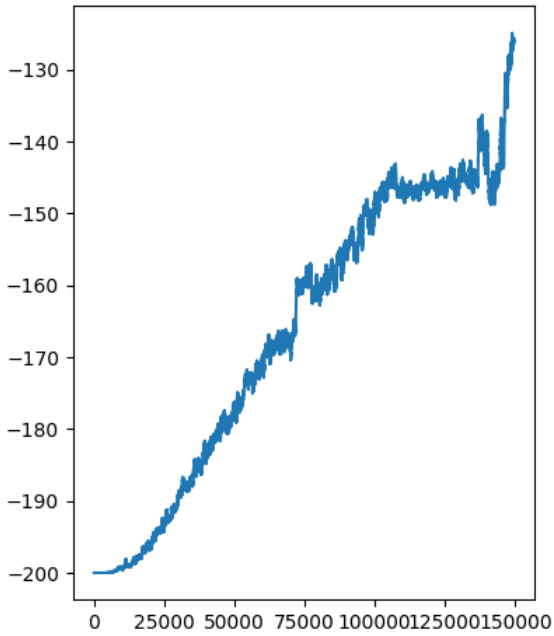
I have tried to change the max final value of the epsilon, and in particular max final epsilon is equal to 0.01, and I have run the system for over 150000 episodes. Here there is an high instability in the training results.
With this parameters I have obtained an Average reward of -130.35 and an Accuracy of 0.99.



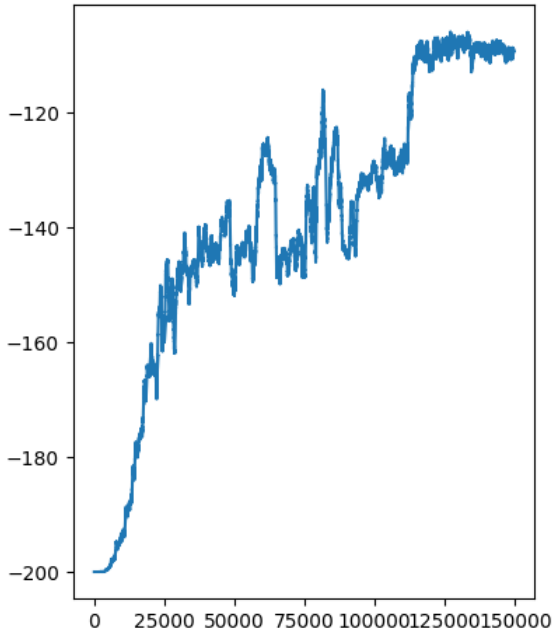
I have changed the final epsilon to 0.05, and I have run again the system for over 150000 episodes. There is again an high instability in the training results, so I have thought to modify again the learning rate and the epsilon.
With this parameters I have obtained an Average reward of -124.35 and an Accuracy of 1.00.



I have changed the final epsilon to 0.001 and the initial epsilon to 0.6. I have also changed the code implementing a dynamic learning rate. Here the stability is increased.
With this parameters I have obtained an Average reward of -122.73 and an Accuracy of 1.00.



I have also used again the dynamic learning rate and the dynamic epsilon. This is the best training that I have done.
With this parameters I have obtained an Average reward of -109.48 and an Accuracy of 1.00.



Initally, I have tried to change the gamma (discount factor). This has not brought particular improvements and so for the above results the discount factor was fixed to 0.95 .

4.2 Discussion on the Q-Table Results

Q-tables are more suitable for environments with finite state spaces, where they can quickly converge to an optimal policy. I have found some problems during the implementation:

- Discretization of States: as explained before.
- Tuning: To achieve satisfactory results, I had tried different values for learning rate, epsilon, size, number of episodes.

5 Deep Q-Learning Implementation

5.1 Deep Q-Learning introduction

Deep Q-Learning is a reinforcement learning technique that combines the principles of Q-Learning with deep neural networks to handle complex and high-dimensional state spaces. It's particularly effective in scenarios where the state and action spaces are too large for traditional tabular Q-Learning methods to be practical. These are the most important parts:

1. The Q-value: it represents the expected cumulative future rewards an agent can obtain by taking that action and following the optimal policy.
2. Deep Neural Network: instead of using a traditional Q-table to store Q-values for each state-action pair, a deep neural network is employed to approximate the Q-function. The neural network takes the current state as input and produces Q-values for all possible actions as output.
3. Buffer Replay: to stabilize training and improve sample efficiency, this technique involves storing the agent's experiences (state, action, reward, next state) in a replay buffer and sampling mini-batches from this buffer to train the neural network. This reduces the correlation between consecutive experiences and helps prevent the network from overfitting to recent experiences.
4. Exploration vs Exploitation (as in Q-Table): it employs an epsilon strategy for exploration. With probability epsilon, the agent selects a random action to explore, and with probability 1 - epsilon, it selects the action with the highest estimated Q-value.

5. Training: The network is trained iteratively by sampling experiences from the replay buffer, calculating the loss between the predicted Q-values and target Q-values, and then updating the neural network's weights.

5.2 Code for NN class

In this part there is the use of the Replay Buffer. The replay buffer stores a collection of states to break the correlation of the last episodes in the data and enable more stable and efficient training of RL agents: you do not train the network only with states of the last episodes. It is a container for storing and sampling experiences collected.

```
%pip install gymnasium[classic-control]
%pip install tensorflow

import numpy as np
import tensorflow as tf
import gymnasium as gym
import os
import random
import matplotlib.pyplot as plt
from gym import wrappers
from keras import regularizers
from keras.optimizers import Adam
from keras.models import Sequential
from keras.models import load_model
from keras.layers import Dense
from collections import deque

class nnq:
    def __init__(self, env, learning_rate, epsilon):

        self.epsilon = epsilon
        self.learning_rate = learning_rate
        self.train_net = self.network()

    def network(self):

        model = Sequential() #input_shape = 2 -> position and speed

        model.add(Dense(24, input_shape=env.observation_space.shape,
        activation='relu',kernel_initializer='he_uniform'))
        model.add(Dense(36, activation='relu',
        kernel_initializer='he_uniform'))
        model.add(Dense(24, activation='relu',
        kernel_initializer='he_uniform'))
        model.add(Dense(3, activation='linear',
        kernel_initializer='he_uniform'))

        opt = tf.optimizers.Adam(learning_rate=tf.keras.optimizers.schedules.
        ExponentialDecay(self.learning_rate, decay_steps=300,
        decay_rate=0.96, staircase=True))
        model.compile(optimizer=opt, loss='mse', metrics=["mse"])

        return model
```

```
def best_action(self,state):
    best_action_to_do = self.train_net(np.atleast_2d(state))
    return np.argmax(best_action_to_do[0], axis=0)


def action(self, state):
    if random.random() > self.epsilon :
        return self.best_action(state)
    else:
         return np.random.choice(3)

def replay_buffer(self, buffer, batch):

    buffer_batch = random.sample(buffer, batch)

    state = np.array([i[0] for i in buffer_batch])
    action = np.array([i[1] for i in buffer_batch])
    next_state = np.array([i[2] for i in buffer_batch])
    reward = np.array([i[3] for i in buffer_batch])
    terminate = np.array([i[4] for i in buffer_batch])

    current = self.net(state) #actual rewards
    #target = np.zeros((state.shape[0],3))
    target = np.copy(current)

    next_q = self.net(next_state) #future rewards
    max_next_q = np.amax(next_q, axis=1)
    #max reward in future reward (after three actions)

    for e in range(state.shape[0]):
        target[e][action[e]] = reward[e] +
        0.99 * (1 - terminate[e]) * max_next_q[e]

    self.net.fit(x=state, y=target, epochs=1,verbose=0)
    self.learning = self.net.optimizer.learning_rate.numpy()


def decresing_epsilon(self):
    if self.epsilon > 0.3 :
      self.epsilon = 95 * self.epsilon / 100
    else :
      self.epsilon = 995 * self.epsilon / 1000

    # to have always a bit of randomness
    if self.epsilon < 0.001 :
      self.epsilon = 0.009
```

About this code:

- Initialization of env, epsilon and learning rate.
- Neural Network Model:
  - Input layer with 24 units and ReLU activation
  - Hidden layer with 36 units and ReLU activation.
  - Another hidden layer with 24 units and ReLU activation.
  - Output layer with as many units as there are actions in the environment (3 in this case) and a linear activation function.

  - The model is compiled with the Adam optimizer and mean squared error (MSE) loss function.

- Best Action: this method takes a state as input and returns the action with the highest Q-value according to the agent's neural network.
- Action: this method implements Exploration-Exploitation selection.
- Sample batch: after adding a state to the buffer (in the training code), there is a sampling of the batch. This random sampling helps to break any temporal correlations in the data. The code stores the components of each sampled experience. The components are converted in np.array to allow the use in the network.
- Training: this method is used to train the agent. It takes a batch of experiences as input, including states, next states, actions, rewards, and termination flags. It computes the target Q-values using the Q-learning update rule and then fits the Q-network to minimize the mean squared error between current Q-values and target Q-values.
- Decresing Epsilon: this method is responsible for reducing the exploration parameter epsilon over time.

5.3 Code for Training

```
episodes = 3500
learning_rate = 0.01
epsilon = 1.0
batch = 64

reward_single_episode = deque(maxlen=100)
win_episode = 0
episode_result=[]
reward_result=[]

if not os.path.exists("nn/"):
  os.makedirs("nn/")

env = gym.make('MountainCar-v0')
nnq_class = nnq(env, learning_rate, epsilon)

#print(env.action_space) ->
Discrete(3)
#print(env.observation_space) ->
Box([-1.2  -0.07], [0.6  0.07], (2,), float32)
#print(env.observation_space.shape) ->
(2,)

buffer = deque(maxlen=10000)


for episode in range(1, episodes):
  state, _ = env.reset()
  terminate, truncate = False, False

  for step in range(1, 201): #200 step to win
    action = nnq_class.action(state)
    next_state, reward, terminate, truncate, _ = env.step(action)

    buffer.append((state, next_state, reward, action, terminate))
```

```
    state = next_state

    if step % 20 == 0 :
      if len(buffer) > batch :
          nnq_class.replay_buffer(buffer, batch)

    if terminate or truncate:

        reward_single_episode.append(step * -1)
        average_reward = sum(reward_single_episode)
        / len(reward_single_episode)
        episode_result.append(episode)
        reward_result.append(average_reward)

        print(f"Episode: {episode} -
        Average Reward: {average_reward:.4f}   -----
        epsilon(randomness): {nnq_class.epsilon:.4f} -
        learning rate: {nnq_class.learning_rate:.4f}")

        break

if episode % 200 == 0 :
  np.save(f'./table/{episode}-q.npy', nnq_class.net)

  plt.plot(episode_result, reward_result)
  plt.xlabel('Episodes')
  plt.ylabel('Average Reward')
  plt.ylim(-200, None)
  plt.show()

if episode == 1200:
  batch = 32
if episode == 2500:
  batch = 16

if episode % 5 == 0 :
  nnq_class.decresing_epsilon()
```

About the code:

- Initialization of the parameters.
- Replay Buffer Initialization: a Replay Buffer object is created. This buffer will store experiences to be used during training.
- Episode Loop: the code enters a loop that iterates over a predefined number of episodes (3500 in this case). For each episode, the environment is reset.
- Step Loop: in each episode, there is a step loop that runs for a maximum of 200 steps (a common limit for the environment). The agent selects an action. The agent takes the selected action, observes the next state, reward, and termination status from the environment. The experience is added to the replay buffer.
- The agent starts training only when there are enough experiences in the buffer.
- The agent method is called with a batch of experiences from the buffer.
- Monitoring and Plotting data and graph.
- Dynamic Batch Size and Decresing Epsilon.

5.4 Code for Testing

```
env = gym.make('MountainCar-v0')
load_file = "200"
average_reward, win_episode = [], 0
learning_rate=0.01
epsilon= 1.0
episodes = 20

nnq_class = nnq(env, learning_rate, epsilon)

nnq_class.net = np.load(f'./table/{episode}.npy')


for episode in range(episodes):

    state, _ = env.reset()
    terminate, truncate, episode_reward = False, False, 0.0

    while not terminate and not truncate:
      action = nnq_class.best_action(state)

      next_state, reward, terminate, truncate, _ =
      env.step(action)

      episode_reward += reward
      state = next_state

      if next_state[0] >= 0.5:
        win_episode += 1

    average_reward.append(episode_reward)
    print(f"Episode: {episode} -
    Episode reward: {episode_reward:.2f}")

mean = sum(average_reward) / len(average_reward)
accuracy = win_episode / episodes

print(f"\n\nAverage Reward: {mean:.2f},
Accuracy {accuracy:.2f}\n")
```

About this code:

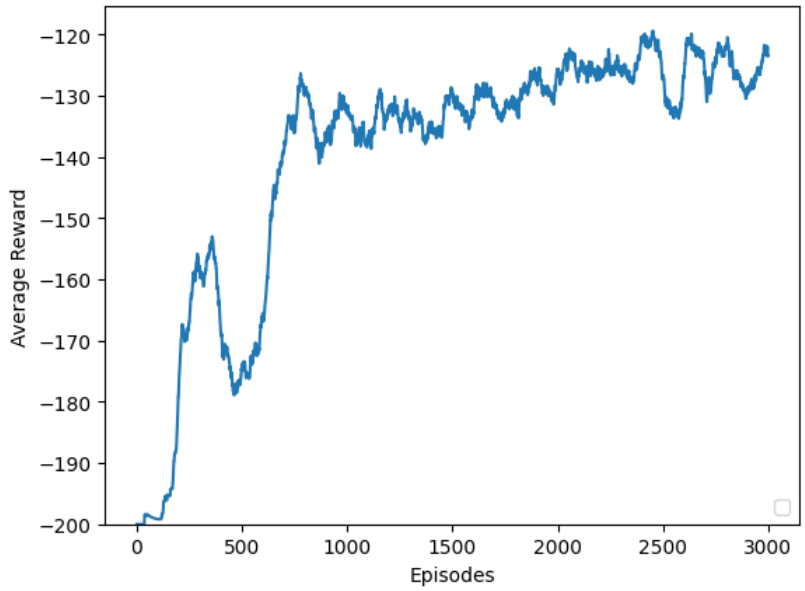
- Initialization of parameters and library.
- The code enters a loop that runs for some iterations. In each iteration, it performs the following steps:
  - Reset the environment.
  - In a while loop, it takes actions in the environment until the episode ends. It collects rewards and updates the state based on the agent's policy.
  - If the agent's next position reaches 0.5, it increments the number of episodes won.
  - After all episodes are completed, the code calculates average reward and accuracy.

6 Deep Q-Learning Results

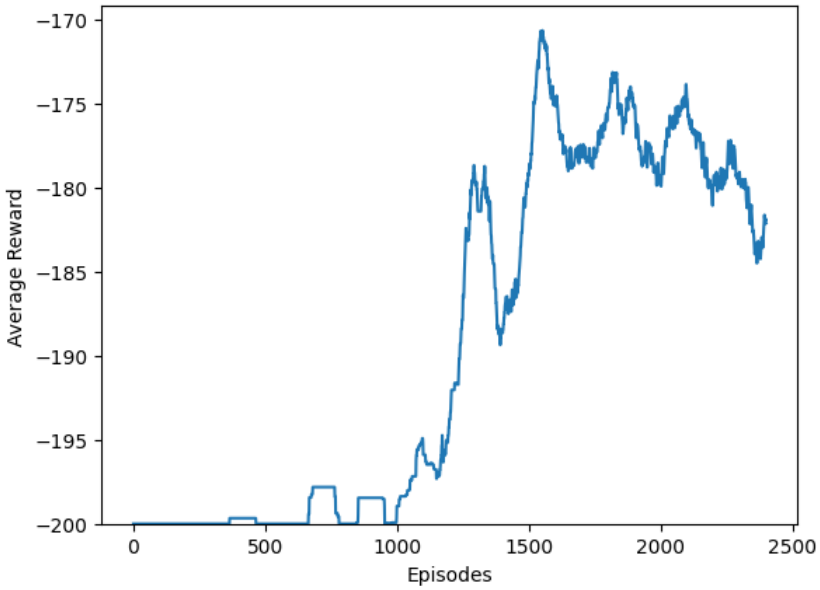
6.1 Performance Evaluation on Deep Q-Learning

Using the structure explained in the previous chapter, I have trained the network several times to find the best values for the parameters. I report below all the interesting traials that I have done.
I have used 3000/3500 episodes to train the network, while I have done the test with 500 episodes.
I have created a function to decrease the epsilon(randomness) and I have always used this dynamic parameter for all the traials, with small changes.

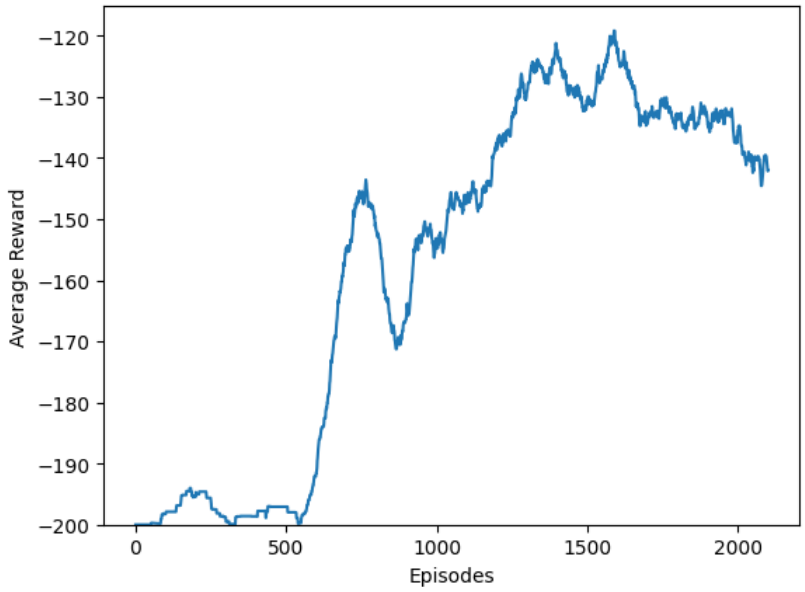
I have started setting the batch to 32 as min and 40000 as max number of steps, with a learning rate of 0.001 decrementing it with Adam. The netowrk created was characterized by 4 layers: 18, 36, 18 and 3 nodes.
With this parameters I have obtained an Average reward of -124.2366 and an Accuracy of 0.93. The training was good and it is possible to see it in the graph below.



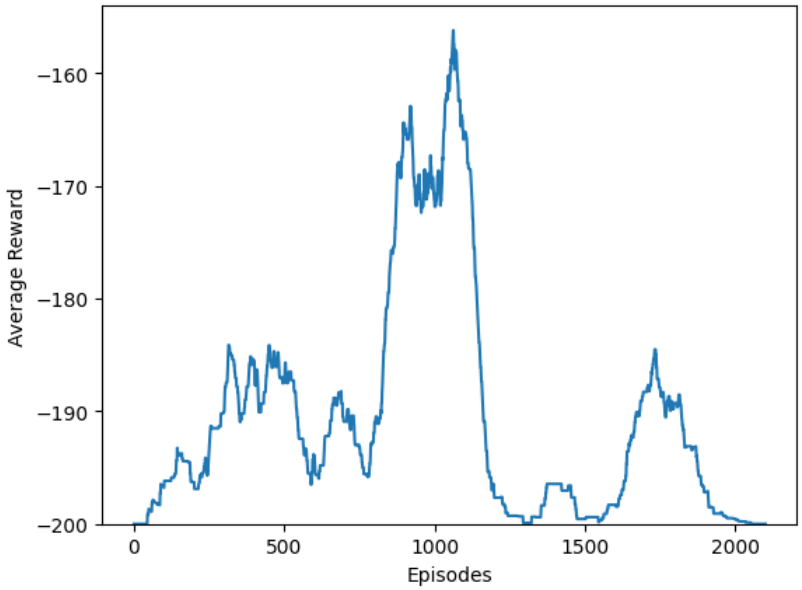
I have continued with the same numbers for the batch, but now with a fixed learning rate of 0.001. The network created was characterized by 3 layers: 48, 36 and 3 nodes.
With this parameters I have obtained an Average reward of -170.2366 and an Accuracy of 0.63. The network created was not the proper one for the system, moreover the fixed learning rate creates a problem of overfitting after the episode 1500.



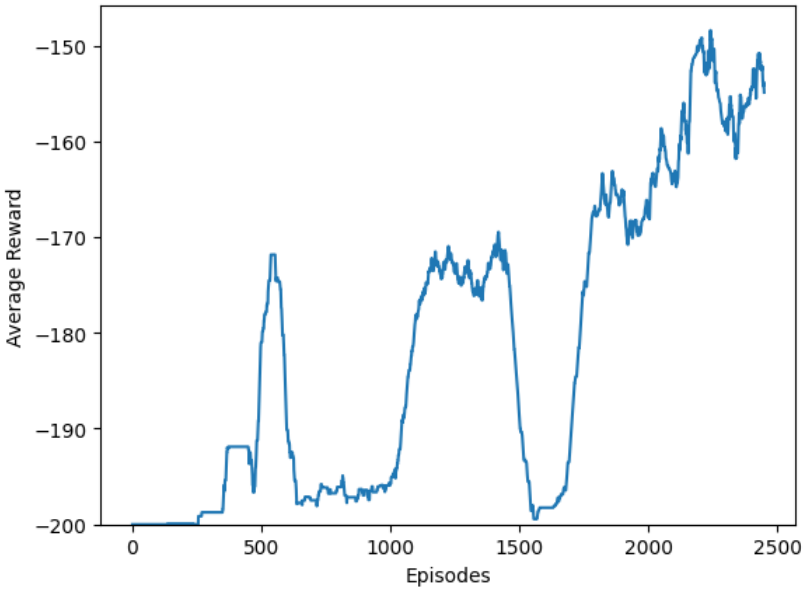
I have continued with the same numbers for the batch, returning with a descending learning rate, now initially equal to 0.01. The network created was characterized by 4 layers: 18, 36, 18 and 3 nodes. With this parameters I have obtained an Average reward of -123.80 and an Accuracy of 1.0. With this traial, I have understood that I have to continue to use a dynamic learning rate.



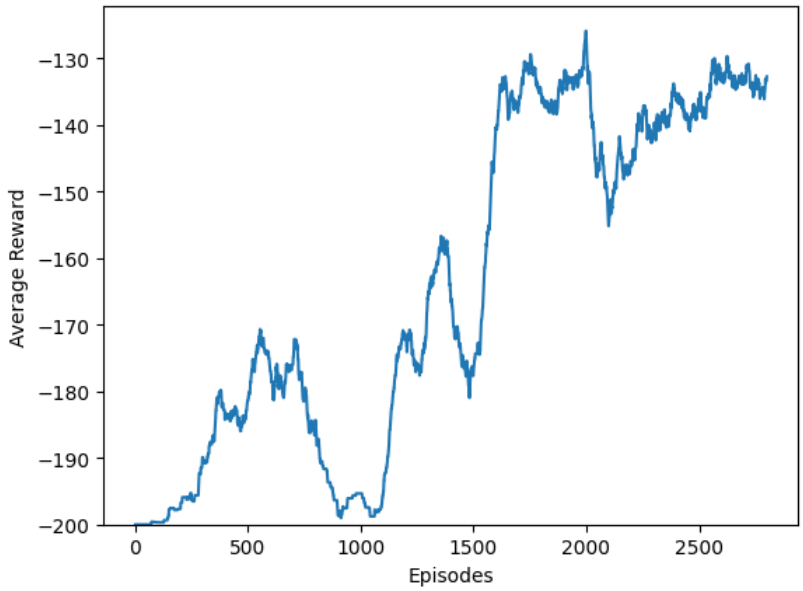
I have same batch and same learning rate. I have changed the number of the nodes of the 4 layers: 24, 24, 24 and 3 nodes.
I have obtained an Average reward of -155.61 and an Accuracy of 1.0. With this trial I have found that I can have a perfect accuracy with a not fantastic average reward. So the problem is solved, but it could be solved better. Moreover the training suffered from overfitting.



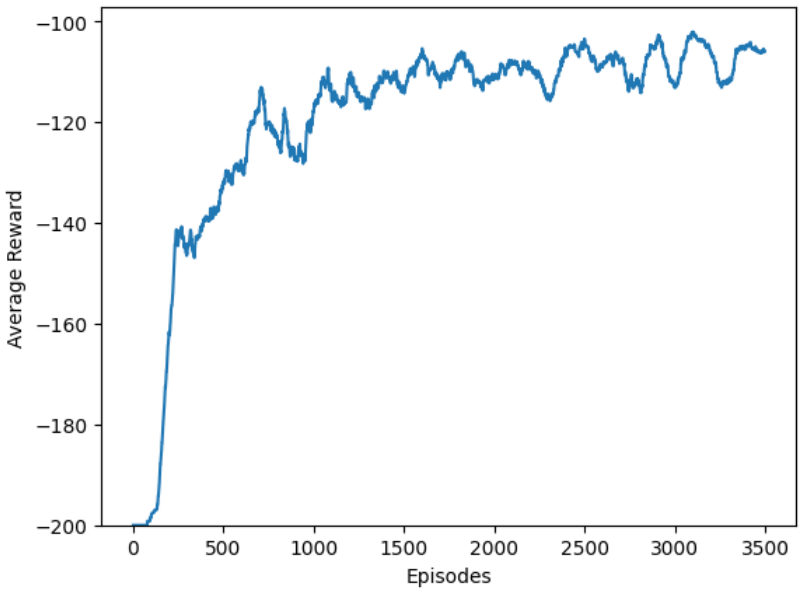
I have continued to operate on the network changing the 4 layers: 36, 18, 9 and 3 nodes. I have created these network with the idea of decrementing the nodes at each layer because the last layer is fixed to 3 nodes (number of the possible actions in this environment).
I have obtained an Average reward of -157.21 and an Accuracy of 1.0. With this trial I have found that I can have a perfect accuracy with a not fantastic average reward.



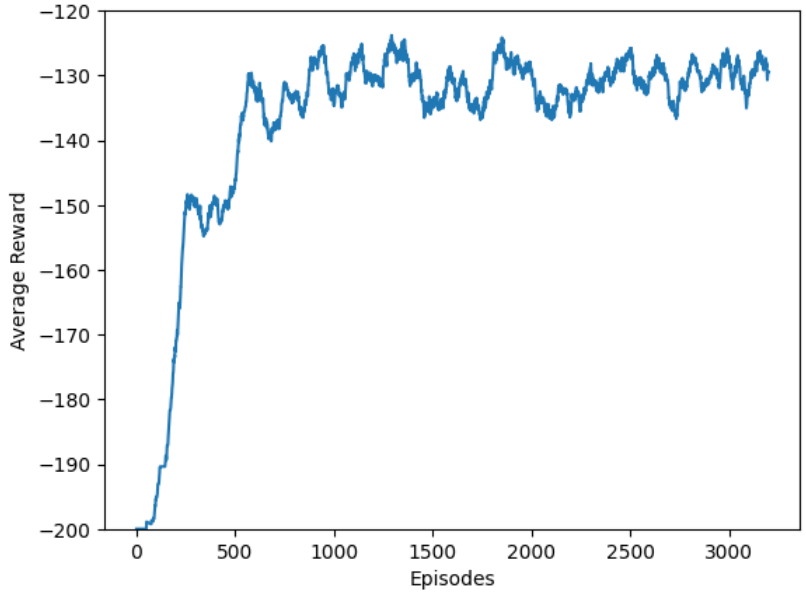
Again, I have tried with another network: 36, 18, 18 and 3. Here I have found an an Average reward of -129.82 and an Accuracy of 0.91.



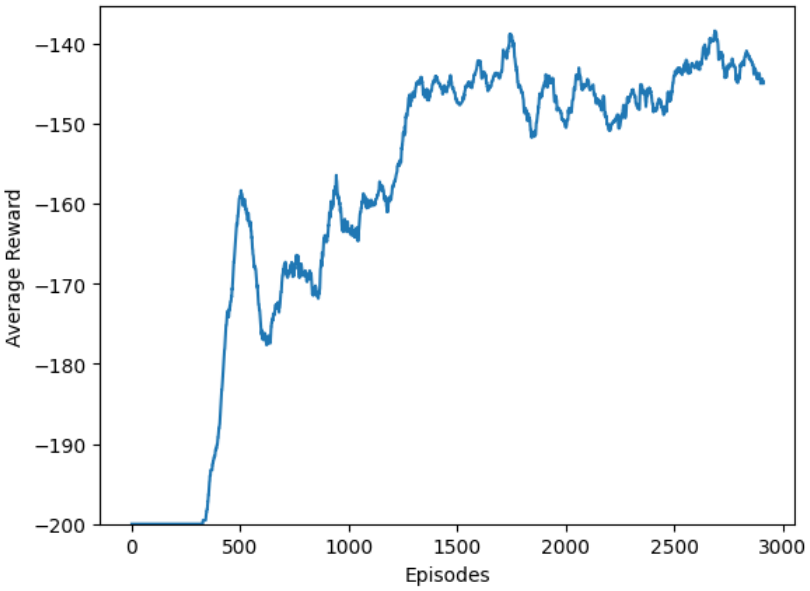
In the end I have tried another network changing also the the batch to 64 episodes as minimun. The network is: 24, 36, 24 and 3 nodes. I have thought that this would have led to a greater instability to the training; so, after some traials, I have dynamically decreased the min number of episodes for the batch starting from 128 to 16.
I have obtained an Average reward of -103.201 and an Accuracy of 0.999. This is the best training that I have done.
This was impressive and so I have tried to use the same approach also with the other network proposed before, but with poor results.



Using the decreasing batch, I have tried 4 layers with 30, 36, 30 and 3 nodes.
I have obtained an Average reward of -130.53 and an Accuracy of 0.98.
I have tried several times but the results are not good enough.



I have tried to use again a learning rate of 0.001 with the best network founded.
I have obtained an Average reward of -145.56 and an Accuracy of 0.95.



6.2 Discussion on the Deep Q-Learning Results

In the previous section, I encountered several challenges, that I have solved traial by traial:

- Network Architecture: The primary challenge in using a neural network approach was selecting the most suitable network architecture for the specific environment. This required several traials. I conducted multiple training sessions, each using about 3000 episodes.
- Learning Rate: I have started using a dynamic learning rate (I have learned this during my bachelor). I have tried (only for research purposes) the fixed one.
- Batch Size Considerations: I explored different batch sizes during training. It is needed to have a small batch to reduced the risk of over correlation in the learning process.

In summary, neural networks may require longer training times but are effective with fewer episodes, leading to a model with greater generalization capabilities.