ML_Project

Lorenzo Mastrandrea 1892793

1 Introduction

Make an agent learn how to move and how to win in different environments was the challenge that I try to take in this project. The scope is to show how Reinforcement Learning techniques are useful in this kind of problems.
The same Feedforward neural network is used for solving with the Deep Q-learning algorithm two different environments of Gymnasium, **Taxi** and **Mountain Car**. In particular in the first domain I will make a comparison with respect to the Q-learning approach as it is a small discrete environment, while the second one it will be analyzed only from the point of view of Deep Reinforcement Learning.

2 Taxi

2.1 The environment

There is a 5x5 grid-world in which there are four designated **pick-up and drop-off** locations. The taxi spawns in a random location while the passenger spawns casually in one of the previous special locations. The goal of the taxi is to reach the passenger, pick him up, move him to the destination (that is one of the four special locations) and eventually drop off the passenger. When the passenger reaches the destination, the episode ends.

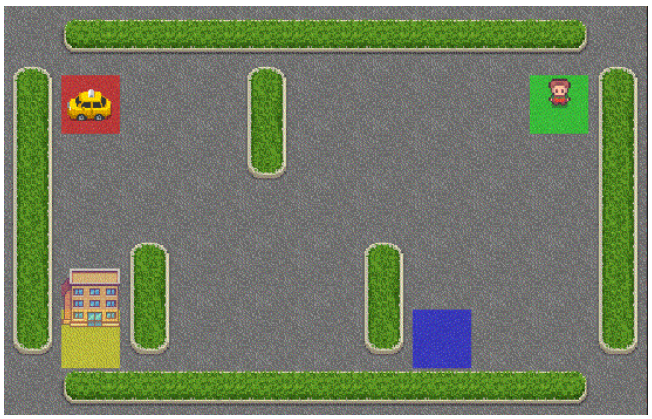
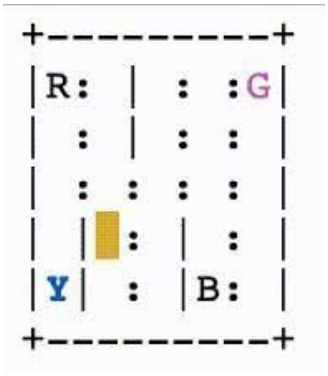


Figure 1: Taxi-environment

The agent, means the taxi, can choose between six actions at every step of the episode:

- 0: Move south (down)
- 1: Move north (up)
- 2: Move east (right)
- 3: Move west (left)
- 4: Pickup passenger
- 5: Drop off passenger



In total there are 500 possible states since there are 25 different locations, 5 possible locations of the passenger (4 special locations + the case when the passenger is in the taxi) and 4 possible destination locations. Destinations on the map are represented with the first letter of the color.
The passenger and the destination locations could be (0: Red), (1: Green), (2: Yellow), (3: Blue), but the passenger has one more location that is (4: In taxi).
The states are integer numbers and are encoded in this way: **((taxi_row * 5 + taxi_col) * 5 + passenger_location) * 4 + destination**.
The episode starts with the player in a random state and for each action that the agent makes he gets back a reward of -1 unless it successfully delivers the passenger getting +20 or -10 if it executes the "pickup" and "drop-off" actions illegally. The episode ends if the taxi drops off the passenger to the destination or if it overcomes the length of the episode whose limit is 200 only if using the time_limit_wrapper.

2.2 Proposed solution

Q-Learning creates a Q-table which associates a Q-value to each state-action pair and so the working agent can "refer to" it in order to maximize its reward in the long run. The Feedforward neural network implemented for well-approximating these Q-values has this architecture:

- the **input layer** is made of **4** nodes, representing the actual state of the game;
- **three hidden layers** of **128 units** each, where each unit takes as input the outputs of the nodes of the previous layer and it applies a linear combination followed by the use of the ReLu activation function;
- the **output layer** consists of **6 neurons** that take as input the outputs of the units of the third hidden layer and it computes only a linear combination giving as result the Q-values associated to each state-action pair, fixing the state.

All the layers are fully connected and BackPropagation is used for updating the weights of the network with Mean Squared Error as Loss function and 'Adam' as optimizer.
Experience Replay is a technique used during the training process because instead of providing the network with sequential experiences as they occur in the environment, we fit into it a fixed **batch** of random samples taken from a buffer that stores for each step **the current state, the action taken, the received reward, the next state**. A reason for doing this is because consecutive samples are correlated to each other and this will lead to inefficient learning, so breaking this correlation helps to improve the performance.
The training process can be resumed in these few steps:

1. through the **epsilon-greedy strategy** an action is chosen and executed in the environment;
2. then the current state, the action, the reward and the next state are stored in the **replay memory**;
3. from this buffer we take a batch of samples;
4. for each sample we compute the **Q-values** related to the action actually taken and also the **Q-target values** that are results of the **Bellman Equation**;
5. compute the **loss** between these two values;
6. update the network through **Gradient Descent**.

2.3 Hyper-Parameters

At the beginning the agent knows nothing about the environment and each step leads to a negative reward unless it drops off the passenger to destination, almost impossible, so it's important to make a lot of random actions initially for exploring most of the states.

```
EPS_MAX = 1
EPS_MIN = 0.01
EPS_DECAY = 0.99997
```

Epsilon starts with a high value and then decreases slowly at every step in order to have time to explore many states. The formula for decreasing is simply.

```
epsilon = epsilon * EPS_DECAY
if epsilon < EPS_MIN:
    epsilon = EPS_MIN
```

All other hyper-parameters are fruit of many tries for having better results:

```
LR = 1e-3
GAMMA = 0.99
BATCH SIZE = 128
MAX REPLAY MEMORY SIZE = 100000
```

2.4 Training results

The training process lasted for 10000 episodes and the plot below shows the performance over time of the agent.

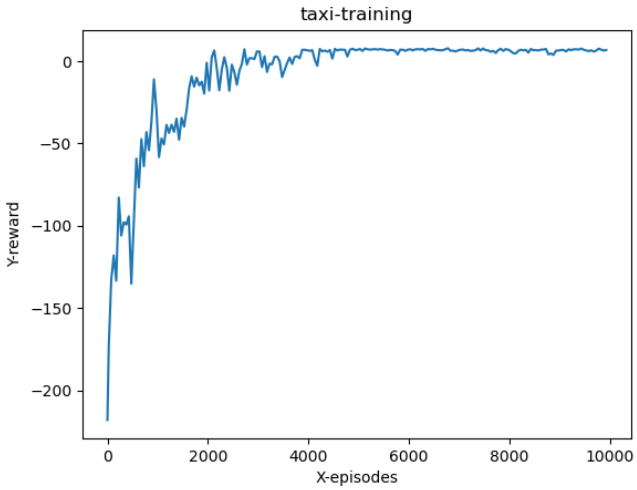


Figure 2: Taxi-training

As the chart shows, during the first episodes the agent obtains worse scores because it has not yet the knowledge of what to do in order to obtain higher rewards. As the time increases, the performance grows thanks to the exploitation of the epsilon greedy strategy that allows to choose better actions.
Now that we have a trained model we can use it for testing if the agent has learnt how to get the best reward. For doing so we will compare the test of the model with a random test on 1000 episodes.

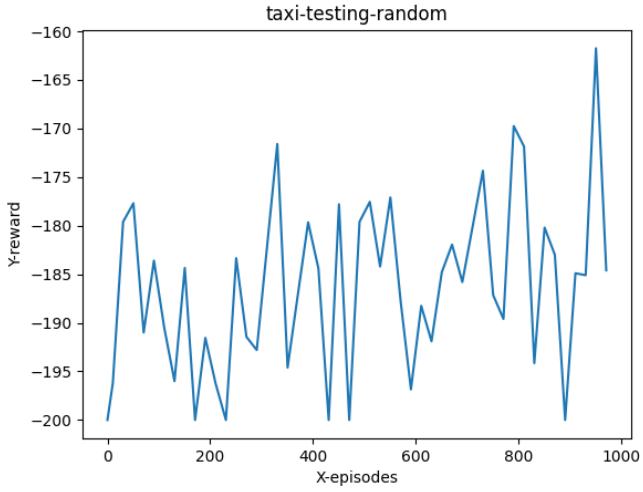


Figure 3: Random policy testing

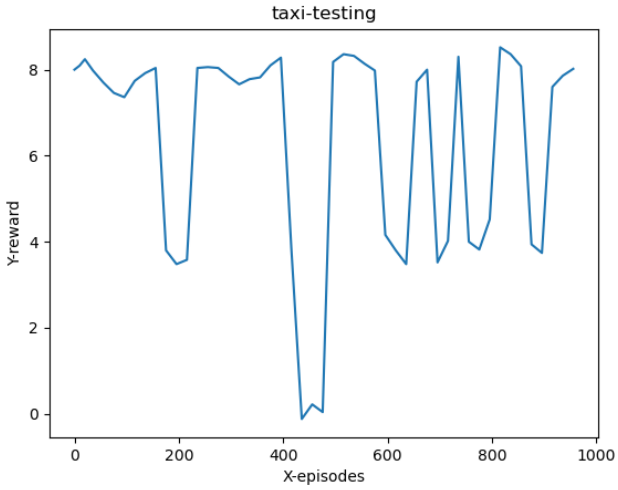
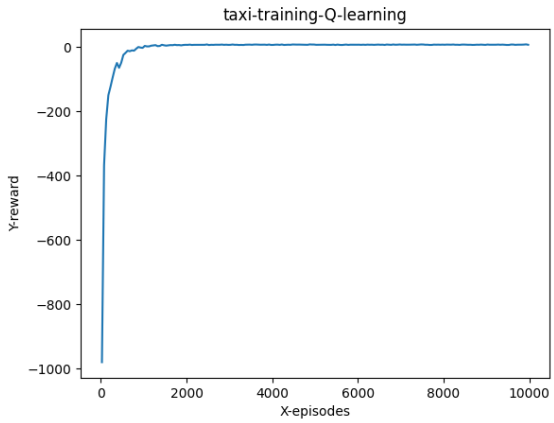
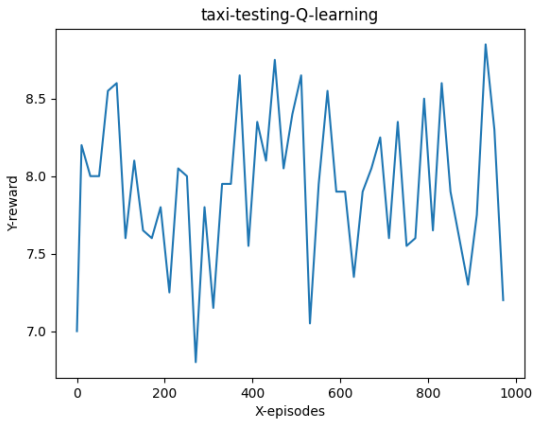


Figure 4: Trained model testing

In the first case the agent moves casually and on average he manages to make -180 points and maybe the passenger never arrives to destination, while in the second case with the trained model he gets on average 7-8 points. This score has sense because before dropping off the passenger to destination, the taxi needs to reach him, pick him up and move him to the final location.
The taxi environment is discrete and it is more suitable for the **Q-learning algorithm**, the approach is more or less the same with the same parameters, but instead of using the neural network there is the **Q-table**, a look-up table, of size 500 x 6 keys because to each state it can be associated one of the six actions. The value associated to each entry is updated according to this training rule:
$Q(s,a) = reward + \gamma * Q^*(s^{next}, a^{next})$
These are the results of the training and testing.



(a) Q-learning training



(b) Q-learning testing

The first plot shows how faster the Q-learning is with respect to the Deep Q-learning algorithm, but although this, the average reward is almost the same 7-8 points.

3 Mountain Car

3.1 The environment

The Mountain Car is a deterministic Markov Decision Problem of the Classic Control environments of Gymnasium. It consists of a car placed stochastically at the bottom of a sinusoidal valley. The goal of the agent is to accelerate in a smart way in order to reach the goal state on the top of the right hill.

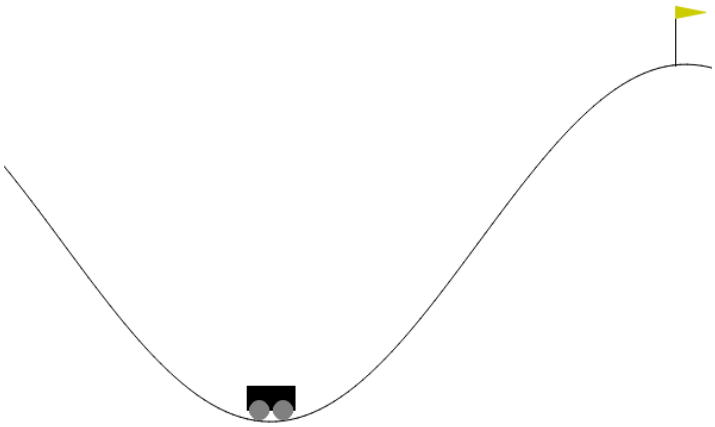


Figure 5: Mountain Car environment

The current state is represented by a vector of size 2:

- the **first element** indicates the **position** of the car along the x-axis. Its value ranges between **-1.2** and **0.6**.
- the **second element** is the **velocity** of the car whose value can be **at least -0.07 and at most 0.07.**

The agent can choose between 3 actions:

- 0: Accelerate to the left;
- 1: Don't accelerate
- 2: Accelerate to the right

Chosen the action, the mountain car follows this transition dynamics:
$velocity_{t+1} = velocity_t + (action - 1) * force - cos(3 * position_t) * gravity$
$position_{t+1} = position_t + velocity_{t+1}$
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.
For each action the agent is penalized with a reward of -1, so the goal is to reach the flag placed on top of the right hill as quickly as possible.
At the beginning of the episode the value of the position of the car belongs to this interval [-0.6 , -0.4] while the starting velocity of the car is always assigned to 0.
The termination of the episode can happen for two reasons:

1. the car reaches a position greater or equal to 0.5, meaning that it reaches the top of the right hill;
2. the car overcomes the limit of the episode which is 200.

3.2 Proposed solution

Differently to the taxi environment, the Mountain Car domain has a continuous observation space, so it is not possible to use Q-learning.
The same Feedforward neural network proposed before is used in order to approximate the Q-values related to each state-action pair and getting the optimal policy.
The only changes in the structure of the neural network are the input layer and the output layer, resulting in this architecture:

- the **input layer** contains **2 neurons** as the shape of the current observation;
- **3 hidden layers** of **128 nodes** each;
- the **output layer** with **3 units**, representing, given the state, the Q-values associated to each state-action pair.

The characteristics of the network were already described in the previous environment and also in this case the Replay Experience technique has been useful because it breaks the correlation between consecutive samples.

3.3 Hyper-Parameters

The more challenging thing to do in this environment is to reach in a random way the top of the right hill because if the car goes right, but it is too slow it cannot reach the flag and it goes back to the valley. It was important to set up the epsilon value to a high value for then decreasing very slowly at the end of every episode. These are the values of the other parameters, tuned in the best way possible for having better results:

```
EPS_MAX = 1
EPS_MIN = 0.01
EPS_DECAY = 0.99997
LR = 1e-3
GAMMA = 0.99
BATCH SIZE = 128
MAX REPLAY MEMORY SIZE = 10000
```

3.4 Training results

The model was trained for 10000 episodes and the figure shows how the agent will act over time.

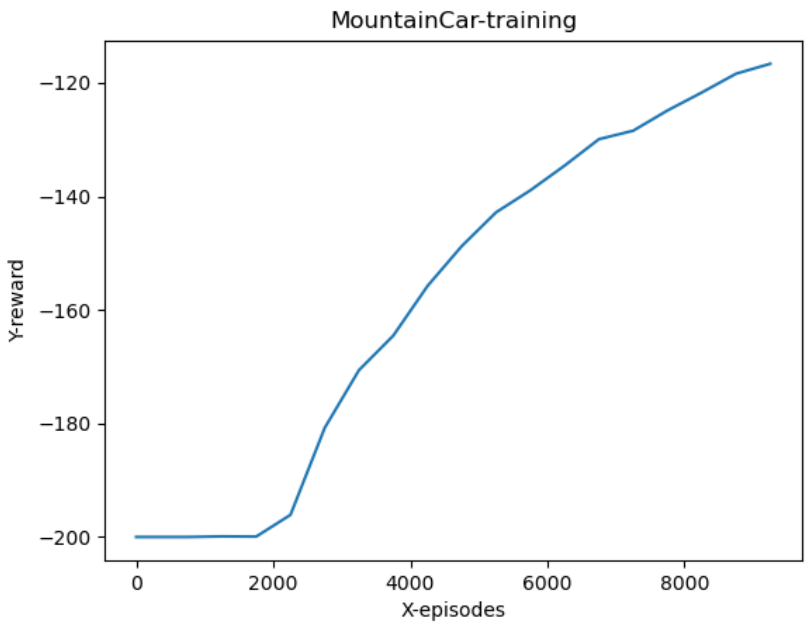


Figure 6: Mountain Car-training

For the first 2000 episodes the agent always loses because it is very rare that it can reach the flag in the first tries; then when it once reach the flag, it will improve the technique getting better rewards.

Now that we have a trained model we can compare it with respect to a random policy in 1000 episodes of testing and see the results.

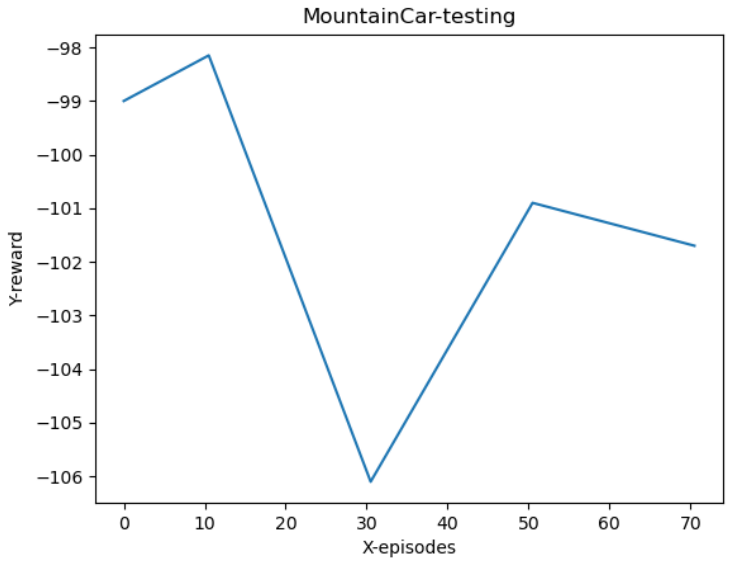


Figure 7: Trained model testing

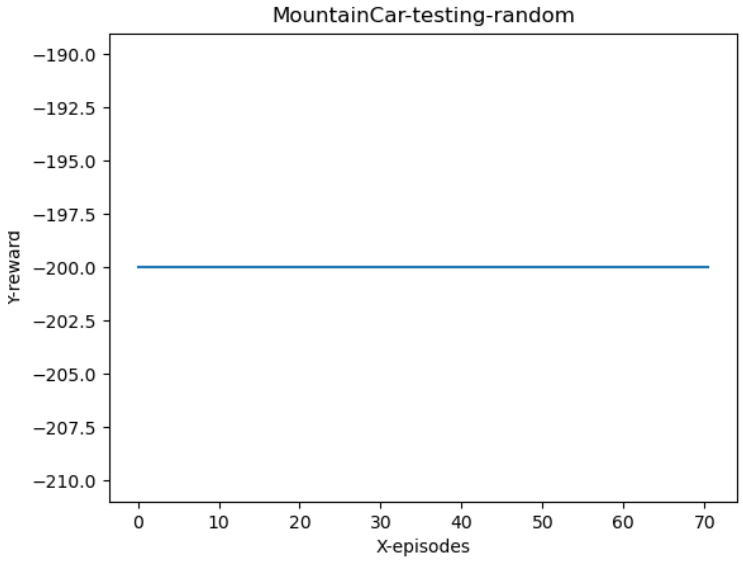


Figure 8: Random policy testing

If the agent plays in a random way will lose at every game, in fact on average it gets -200 of reward, while if it follows the policy of the trained model, it will obtain higher scores on avg -100 points. This is because initially the car needs to go right, for then reaching the left wall, in this way it will have a good approach for catching the flag.