

# **Using Reinforcement Learning to solve Lunar Lander environment**

Valerio Spagnoli 1973484

Matteo Ventali 1985026

June 12, 2025

# Contents

1	Introduction . . . . .	2
2	Basics of Q-Learning . . . . .	3
2.1	Tabular Q-learning . . . . .	4
2.2	DQN . . . . .	5
3	Lunar Lander environment . . . . .	6
3.1	Observation space . . . . .	6
3.2	Action space . . . . .	6
3.3	Reward system . . . . .	6
4	Experiments . . . . .	8
4.1	Lunar Lander with tabular Q-learning . . . . .	8
4.1.1	Training results . . . . .	9
4.1.2	Running results . . . . .	10
4.2	Lunar Lander with DQN . . . . .	12
4.2.1	Training results . . . . .	13
4.2.2	Running results . . . . .	13
4.3	DQN vs Tabular Q-learning . . . . .	13
5	Conclusion . . . . .	14

# 1 Introduction

This project aims to address the Lunar Lander environment by applying a reinforcement learning technique called Q-learning. We will explore two different approaches: tabular Q-learning, which is suitable for environments with small and discrete state spaces, and Deep Q-Networks (DQN), a more advanced method that combines Q-learning with deep neural networks, making it capable of handling larger and continuous state spaces.

The goal is to compare these two techniques in terms of learning performance, stability, and efficiency when applied to the Lunar Lander task, which involves controlling a lander to safely reach the ground between two designated flags under the influence of gravity and physics-based dynamics.

## 2 Basics of Q-Learning

Q-Learning is a reinforcement learning algorithm used to learn the optimal policy

$$\pi^*(s) : S \rightarrow A$$

for an agent interacting with an environment. It is based on learning a Q-function

$$Q(s, a) : S \times A \rightarrow \mathbb{R}$$

which estimates the expected cumulative (discounted) reward of taking an action  $a$  in a state  $s$  and then following the optimal policy thereafter.

The critical part of the algorithm is how to maintain and update the Q-function. In particular, at every iteration, the algorithm updates its value with one of the following rules:

- **deterministic rule:**

$$Q^{new}(s, a) \leftarrow r + \gamma \cdot \max_{a' \in A} Q(s', a')$$

- **non deterministic rule:**

$$Q^{new}(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma \cdot \max_{a' \in A} Q(s', a') - Q(s, a)]$$

In the previous equations appear these terms:

- $\alpha$ : learning rate;
- $r$ : reward received after taking action  $a$  in state  $s$ ;
- $\gamma$ : discount factor for future rewards;
- $s'$  and  $a'$ : respectively, the next state and the next action.

---

**Algorithm 1** Q-learning Algorithm

---

```
1: Initialize  $Q(s, a)$  arbitrarily for all states  $s$  and actions  $a$ 
2: for each episode do
3:   Initialize state  $s$ 
4:   while episode not terminated do
5:     Choose action  $a$  from  $s$  using a strategy
6:     Execute action  $a$ , observe reward  $r$  and next state  $s'$ 
7:     Update  $Q(s, a)$ :
8:   end while
9: end for
```

---

The next action strategy must balance:

- **exploration**: taking random actions to explore the environment and gather information;
- **exploitation**: selecting actions according to the policy learned so far to maximize rewards.

A commonly used exploration strategy in reinforcement learning is the  $\varepsilon$ -greedy policy. Under this approach, the agent selects a random action with probability  $\varepsilon$ , and chooses the action that maximizes the current Q-value estimate with probability  $1 - \varepsilon$ . This allows the agent to balance exploration and exploitation. To reduce the exploration as training progresses, epsilon decays exponentially over time using a fixed decay rate.

## 2.1 Tabular Q-learning

In the tabular version of Q-learning, the action-value function  $Q(s, a)$  is represented explicitly as a table, where each entry corresponds to a specific state-action pair. During training, this table is updated iteratively using the Q-learning update rule. Tabular Q-learning is effective in environments with small, discrete state and action spaces, where it is feasible to store and update the Q-values for all possible combinations. Its main advantages are simplicity, ease of implementation, and theoretical convergence guarantees. However, this approach does not scale to environments with large or continuous state spaces, as the size of the Q-table grows rapidly and becomes impractical. In such cases, approximating the Q-function with a neural network (DQN) or other approaches becomes necessary.

## 2.2 DQN

In Deep Q-Learning (DQN), the Q-function is approximated using a neural network instead of a lookup table, making it suitable for environments with large or continuous state spaces. Unlike traditional supervised learning, where the dataset is fixed and provided in advance, the training data in DQN is generated incrementally through interaction with the environment. Specifically, during training, the agent builds its dataset in the following way:

- at each time step, the agent observes a transition  $(s, a, r, s', done)$  by interacting with the environment;
- this transition is stored in a memory called *replay buffer*;
- periodically, after some interaction steps, a random mini-batch of transitions is sampled from the buffer to train the Q-network.

To apply the Q-learning update rule, the same network is used both to predict the current  $Q(s, a)$  and to estimate the target value  $Q^{new}(s, a)$ . The power of this approach is the ability of the agent to generalize across similar states and learn effective policies in complex environments.

### 3 Lunar Lander environment

This environment is a classic rocket trajectory optimization problem. The environment is based on a 2D physics simulation powered by Box2D, and the lander must navigate a gravitational field.

#### 3.1 Observation space

The **observation space** consists of 8 continuous variables representing:

- The lander’s  $x$  and  $y$  positions, ranging from  $-2.5$  to  $2.5$
- The lander’s velocity components, ranging from  $-10$  to  $10$
- The lander’s angular velocity, ranging from  $-10$  to  $10$
- The lander’s angle, ranging from  $-2\pi$  to  $2\pi$
- Two binary flags that represent whether each leg is in contact with the ground or not.

The landing pad is always at coordinates  $(0, 0)$ . Given the continuity of the observation space, a discretization approach was adopted for the implementation of tabular Q-learning.

#### 3.2 Action space

The environment provides a discrete **action space**, composed by 4 main actions, encoded by 4 integers that range from 0 to 3.

- 0: do nothing
- 1: fire left orientation engine (pushing right)
- 2: fire main engine (pushing upward)
- 3: fire right orientation engine (pushing left)

#### 3.3 Reward system

The agent receives a reward based on the quality of the landing:

A successful landing near the center earns positive rewards. Crashes or going out of bounds result in large negative rewards. Fuel usage penalizes the agent slightly,

encouraging efficiency. Leg contact with the ground provides small bonuses. The episode terminates when the lander comes to rest (landed or crashed), making the problem episodic.

A final reward  $> 200$  is considered a success.



## 4 Experiments

### 4.1 Lunar Lander with tabular Q-learning

To apply tabular Q-learning to the Lunar Lander environment, it is necessary to adapt the algorithm to handle the continuous state space provided by the simulator. Since tabular methods require a finite set of discrete states, the continuous observation space—which includes position, velocity, angle, and leg contact information—must be discretized. This typically involves dividing each dimension of the state vector into a fixed number of bins, effectively mapping continuous observations to a limited set of discrete states. This discretization step is crucial, as it enables the use of a Q-table to store and update action-value estimates for each state-action pair. However, it also introduces a trade-off between precision and tractability: finer discretization improves state resolution but increases the size of the Q-table, potentially making learning slower and more memory-intensive.

In our project, the discretization was manually designed to strike a balance between state granularity and computational feasibility. Instead of relying on automatic binning or predefined wrappers, we defined a set of thresholds for each dimension of the observation vector based on empirical intuition and experimentation. This handcrafted approach allowed us to capture the most relevant features of the environment. The following Python function implements this custom discretization:

```
1 def discretize(obs):
2     result = []
3
4     # Interval array
5     x_intervals = [-0.5, 0.5]
6     y_intervals = [-0.1, 0.1, 1.5]
7     vx_intervals = [-7.5, -5, -0.3, -0.1, 0.1, 0.3, 5, 7.5]
8     vy_intervals = [-7.5, -5, -0.3, -0.1, 0.1, 0.3, 5, 7.5]
9     theta_intervals = [-1.25663706, -0.1, 0.1, 1.25663706]
10    omega_intervals = [-7.5, -5, -0.1, 0.1, 5, 7.5]
11
12    result.append(np.digitize(obs[0], x_intervals))
13    result.append(np.digitize(obs[1], y_intervals))
14    result.append(np.digitize(obs[2], vx_intervals))
15    result.append(np.digitize(obs[3], vy_intervals))
16    result.append(np.digitize(obs[4], theta_intervals))
17    result.append(np.digitize(obs[5], omega_intervals))
18
19    # No need to discretize boolean variables
```

```

20     result.append(int(obs[6]))
21     result.append(int(obs[7]))
22     return tuple(result)

```

#### 4.1.1 Training results

The following section presents the results obtained during the training phase. These experimental outcomes illustrate how the performance of the learned policy evolves over time, highlighting learning progress and convergence properties. Key metrics such as cumulative reward expressed as a moving average and training stability are used to measure the effectiveness of the tuned hyper-parameters. The graph includes a comparison between a baseline run using a purely random policy (green line) and the training run (red line) using the  $\varepsilon$ -greedy strategy, showing both the progression of rewards and the variation of  $\varepsilon$  across episodes (orange line).

The following results have been obtained using following hyper-parameters:

- $\alpha = 0.1$
- $\gamma = 0.99$
- $\varepsilon \text{ decay} = 0.9995$
- $\text{start } \varepsilon = 1.0$  ,  $\text{min } \varepsilon = 0.01$

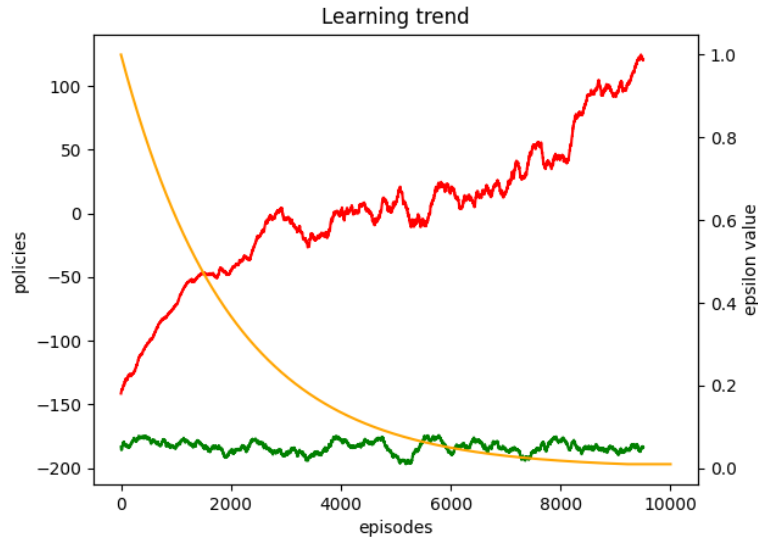


Figure 1: Tabular training process over 10000 episodes

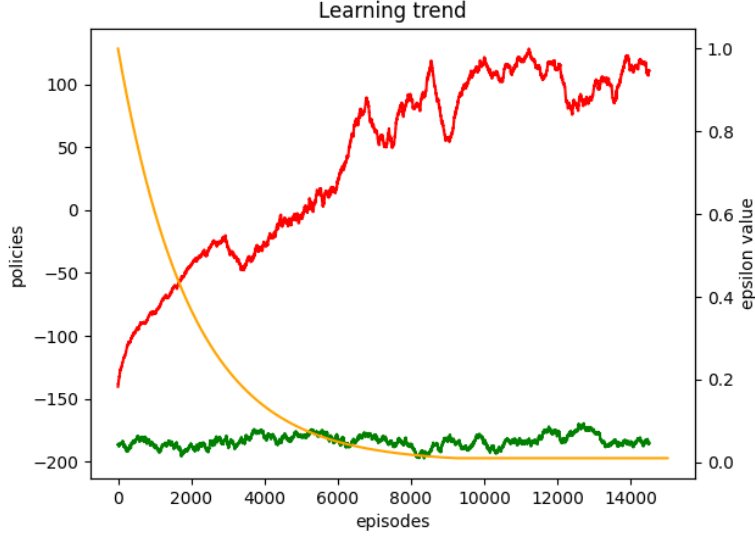


Figure 2: Tabular training process over 15000 episodes

In the first graph, we observe a steadily increasing trend in the cumulative episode reward, indicating that the agent is learning over time. However, the reward curve has not yet reached a plateau, suggesting that the learning process has not fully saturated. To address this limitation, the number of training episodes was extended from 10,000 to 15,000. The resulting trend, shown in the second graph, clearly demonstrates that the agent was able to further consolidate its policy, benefiting from a longer exploitation phase that allowed more stable and consistent improvement. This is supported by the fact that the plateau phase in the second graph is reached when epsilon falls within the range  $[0.01, 0.05]$ , meaning the agent performs random exploration with a probability between 1% and 5%, and follows the learned policy with a high certainty ranging from 95% to 99%.

#### 4.1.2 Running results

The following section presents the results obtained by running the learned policy after 15,000 training episodes. In this evaluation phase, the agent interacts with the environment for 1,000 episodes using the final policy without further learning. For each episode, the cumulative reward is recorded, and the results are visualized in a scatter plot.

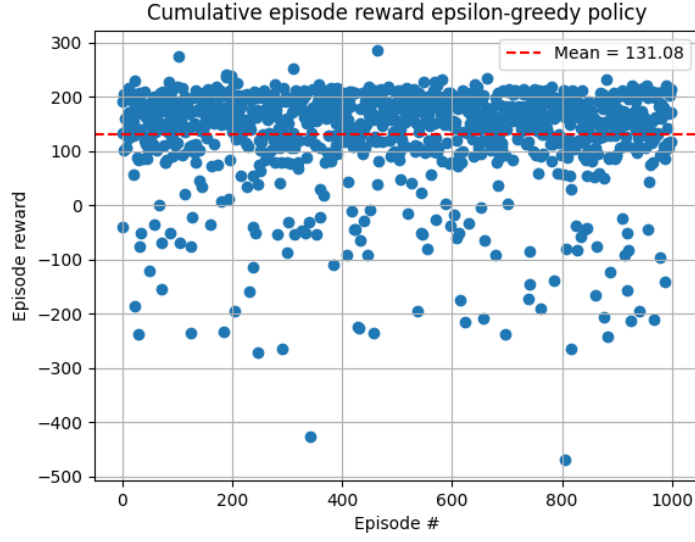


Figure 3: Scatter plot of policy learned by using  $\epsilon$ -greedy strategy

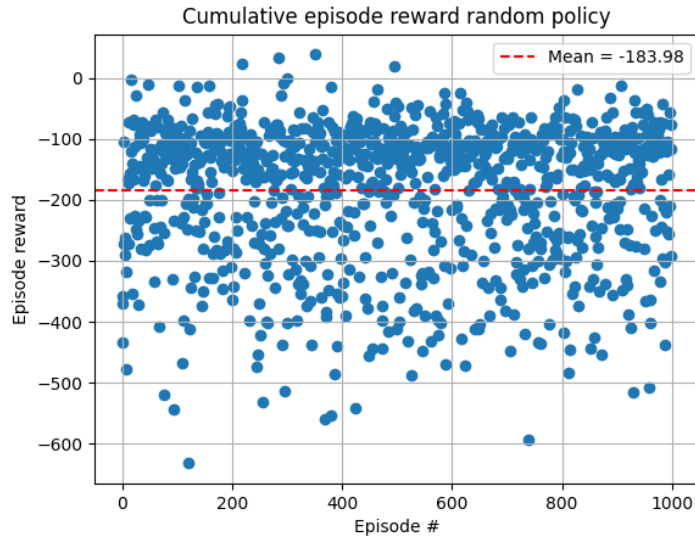


Figure 4: Scatter plot using the random policy

As shown in the first graph, the average reward per episode is approximately 131.08. This suggests that, in most cases, the agent is able to land safely, although not necessarily within the designated landing area marked by the flags (achieving such precision would typically result in a reward greater than 200). Additionally, we observe

the presence of several outlier episodes where the reward drops drastically into negative values. This phenomenon significantly lowers the overall average reward and can be attributed to the discretization of the state space. In the adopted discretization scheme, each feature is divided with higher resolution around zero (i.e., the mean value of each interval is centered closer to 0). However, during the evaluation run, the agent may encounter states that were not visited during training. This mismatch introduces an element of randomness in the action selection, leading to unstable or suboptimal behavior in some cases.

## 4.2 Lunar Lander with DQN

In our DQN implementation for Lunar Lander, the Q-network is structured as a simple feedforward neural network designed to approximate the action-value function  $Q(s, a)$ . The network consists of two hidden layers, each equipped with a *ReLU* activation function, and an output layer that produces a Q-value for each possible action. Specifically, the input layer takes in input the continuous state vector from the environment (8 values), while the output layer has one unit per discrete action available in Lunar Lander (4 actions). In our project, the following Python code (using PyTorch library) defines the structure of the network:

```

1 def __init__(self, state_dim, num_actions, device="cpu"):
2     self.device = device
3     self.model = nn.Sequential(
4         nn.Linear(state_dim, 64),
5         nn.ReLU(),
6         nn.Linear(64, 64),
7         nn.ReLU(),
8         nn.Linear(64, num_actions)
9     ).to(self.device)
10    self.loss_fn = nn.MSELoss()
11    self.optimizer = optim.Adam(self.model.parameters(), lr=0.001)

```

As we already said, during training, the agent samples mini-batches of transitions from the replay buffer to update the Q-network. The following Python code demonstrates how these batches are prepared from stored experiences:

```

1 def _prepareBatch(self, batch, q_network: DQN):
2     training_set = [] # Result of preparing the data
3
4     # Preparing the batch
5     for t in batch: # (s, a, r, s', done)
6         # Q(s,a) and Q(s',a) for all action a

```

```

7         q_values_s = q_network.predict_qValue(t[0])[0]
8         q_values_ns = q_network.predict_qValue(t[3])[0]
9
10        #  $Q(s,a) = Q(s,a) + \alpha[r + y \cdot \max_{a'} \{Q(s',a')\} - Q(s,a)]$ 
11        # Updating only in correspondence of the action
12        action = int(t[1])
13        row = (t[0], q_values_s)
14        if self.update_modality == 0:
15            row[1][action] = t[2] + self.gamma * np.max(q_values_ns) * (1
↪ - int(t[4])) #det
16        else:
17            row[1][action] = q_values_s[action] + self.alpha * (t[2] +
↪ self.gamma * np.max(q_values_ns) * (1 - int(t[4])) - q_values_s[action
↪ ]) #nodet
18
19        training_set.append(row)
20
21    return training_set

```

#### 4.2.1 Training results

#### 4.2.2 Running results

### 4.3 DQN vs Tabular Q-learning

## 5 Conclusion