Using Reinforcement Learning to solve Lunar Lander environment

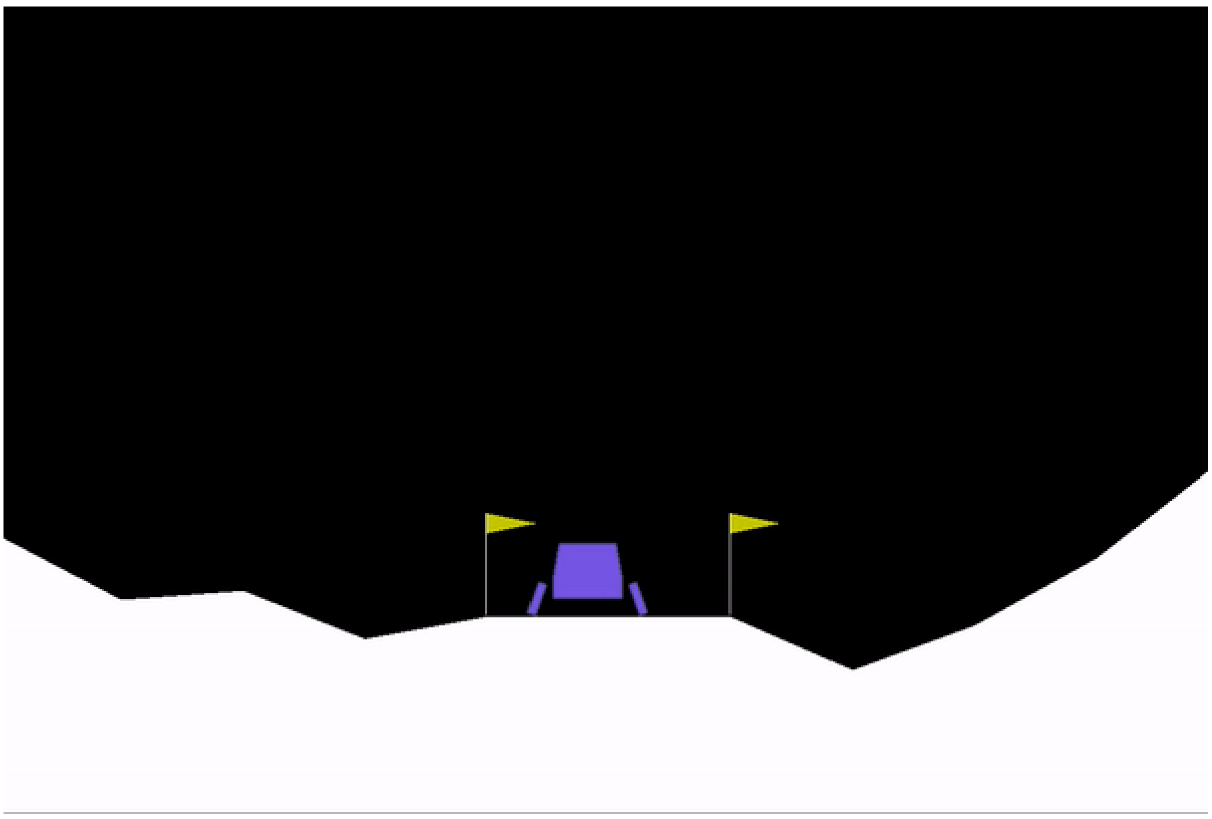
Valerio Spagnoli 1973484
Matteo Ventali 1985026

June 13, 2025

Contents

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

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

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

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

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:

- α: learning rate;
- r: reward received after taking action a in state s;
- γ: 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 ε-greedy policy. Under this approach, the agent selects a random action with probability ε, 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 -2π to 2π
- 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:

```python
def discretize(obs):
        result = []

        # Interval array
        x_intervals     = [-0.5, 0.5]
        y_intervals     = [-0.1, 0.1, 1.5]
        vx_intervals    = [-7.5, -5, -0.3, -0.1, 0.1, 0.3, 5, 7.5]
        vy_intervals    = [-7.5, -5, -0.3, -0.1, 0.1, 0.3, 5, 7.5]
        theta_intervals = [-1.25663706,  -0.1, 0.1, 1.25663706]
        omega_intervals = [-7.5, -5, -0.1, 0.1, 5, 7.5]

        result.append(np.digitize(obs[0], x_intervals))
        result.append(np.digitize(obs[1], y_intervals))
        result.append(np.digitize(obs[2], vx_intervals))
        result.append(np.digitize(obs[3], vy_intervals))
        result.append(np.digitize(obs[4], theta_intervals))
        result.append(np.digitize(obs[5], omega_intervals))

        # No need to discretize boolean variables
        result.append(int(obs[6]))
        result.append(int(obs[7]))
        return tuple(result)
```

To implement in a efficient way the Q-table we have used the data structure called *defaultdict* as shown in the following snippet of code:

```python
    def tabular_QLearning(self, modality=1): # 0 for random, 1 for eps-greedy policy
        # - when a state is not already in the dictionary, then it is added with a value of
        ↪  (0,0,0,0). This happens the first time
        #   the agent ends up in that state
        self.q_table = defaultdict(lambda: np.zeros(self.env.action_space.n))

                                        . . .

```

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 ε-greedy strategy, showing both the progression of rewards and the variation of ε across episodes (orange line).

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

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

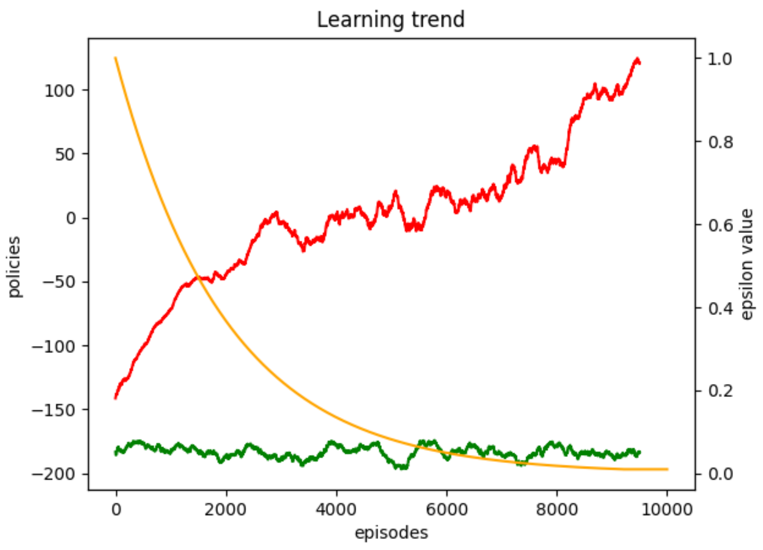


Figure 1: Tabular training process over 10000 episodes

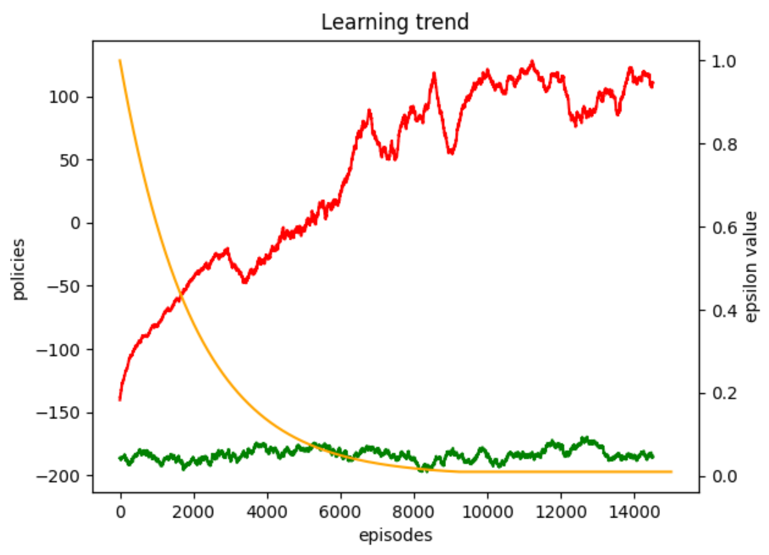


Figure 2: Tabular training process over 15000 episodes

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%.

4.2 Hyperparameters tuning

To obtain the optimal hyperparameters, a tuning process was carried out. This involved experimenting with different configurations to evaluate their impact on performance. The plot below illustrates the results of three distinct configurations, highlighting how their performance evolved over time and helping identify the most effective setup.

The configurations have been executed over 15000 episodes with the following parameters:

- Green curve
  - $\gamma = 0.9$
  - $\varepsilon\ decay = 0.99$
- Blue curve
  - $\gamma = 0.9$
  - $\varepsilon\ decay = 0.999$

- Red curve
  - $\gamma = 0.99$
  - ε *decay* $= 0.9995$

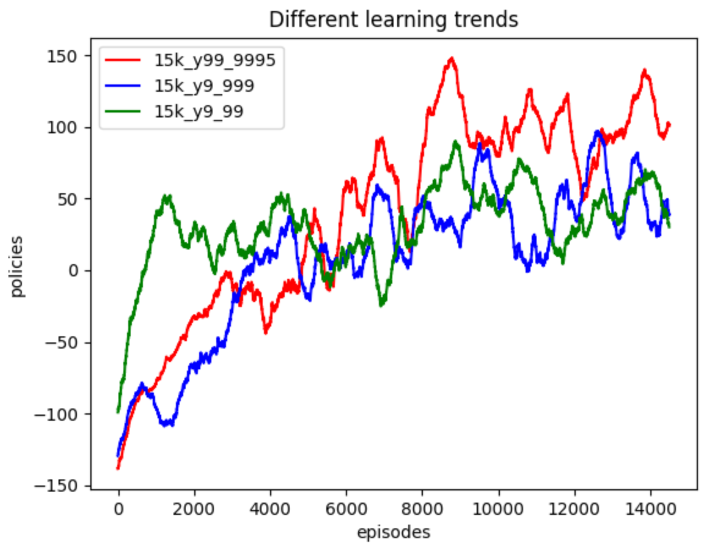


Figure 3: Comparison between three different configuration

The graph shows three different trends based on varying values of gamma and ε-decay. We can see that a fast convergence does not always lead to an optimal policy. In some cases, the agent may quickly settle on suboptimal behaviors due to insufficient exploration or premature exploitation. In fact, although the green line converges almost instantly, the resulting trend stabilizes at relatively low values, indicating suboptimal performance. By increasing the value of both gamma and ε-decay, learning becomes slower and more stable, allowing the agent to gradually refine its policy. Only after a sufficient number of episodes does the agent begin to consolidate an optimal policy, demonstrating the importance of careful hyperparameter tuning for long-term performance.

Given these considerations, the configuration represented by the red curve is the most effective.

4.2.1 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.

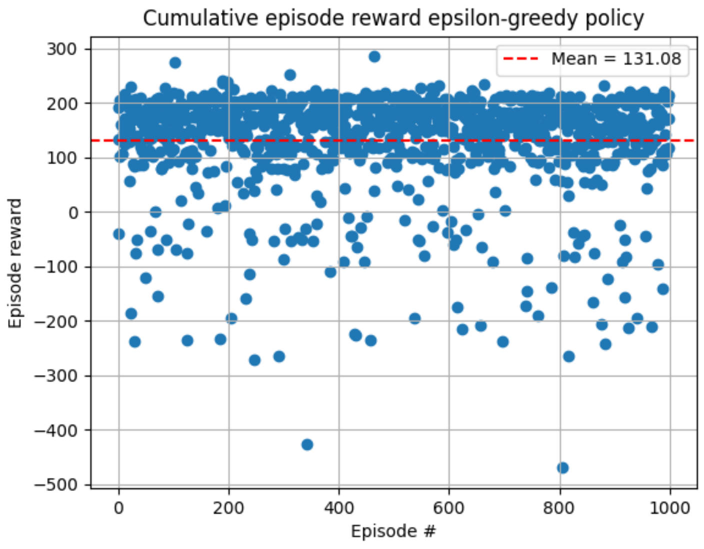


Figure 4: Scatter plot of policy learned by using ε-greedy strategy

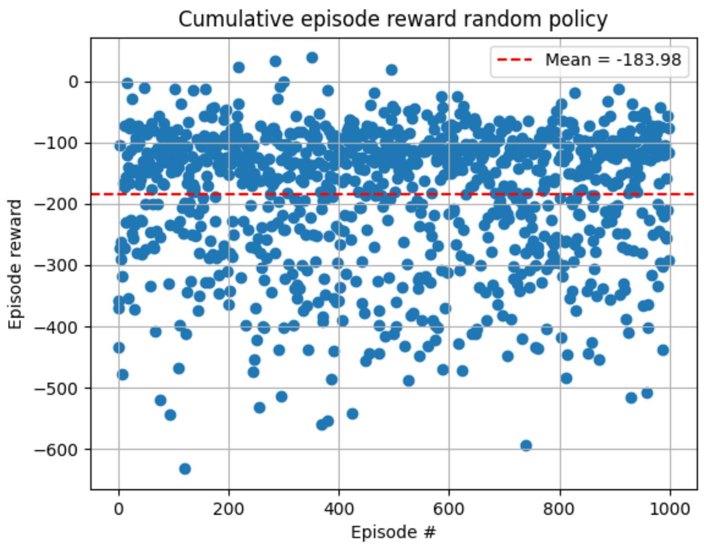


Figure 5: Scatter plot using the random policy

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.

4.3 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:

```python
def __init__(self, state_dim, num_actions, device="cpu"):
    self.device = device
    self.model = nn.Sequential(
        nn.Linear(state_dim, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, num_actions)
    ).to(self.device)
    self.loss_fn = nn.MSELoss()
    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:

```python
def _prepareBatch(self, batch, q_network: DQN):
    training_set = [] # Result of preparing the data

    # Preparing the batch
    for t in batch: # (s, a, r, s', done)
        # Q(s,a) and Q(s',a) forall action a
        q_values_s = q_network.predict_qValue(t[0])[0]
        q_values_ns = q_network.predict_qValue(t[3])[0]

        # Q(s,a) = Q(s,a) + alpha[r + y*max_a'{Q(s',a')} - Q(s,a)]
        # Updating only in corrispondence of the action
        action = int(t[1])
        row = (t[0], q_values_s)
        if self.update_modality == 0:
            row[1][action] = t[2] + self.gamma * np.max(q_values_ns) * (1 - int(t[4])) #det
        else:
            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

        training_set.append(row)

    return training_set
```

4.3.1 Training results

During the training phase of the Deep Q-Network (DQN), experiments were conducted over a total of 8000 episodes to evaluate the effectiveness of two different update strategies: one based on a *deterministic rule* and the other on a *non-deterministic rule*. The goal of this comparison was to analyze the impact of the two update mechanisms on the learning process and the final performance of the agent. This setup was used to investigate how the network behaves under softer, more gradual updates (using non-deterministic formula with $\alpha = 0.1$) versus more aggressive ones. In both cases, the training parameters and simulation environment were kept constant to ensure a fair and consistent comparison.

Hyperparameters were kept the same as the tabular version. Furthermore, the learning rate of the optimizer was fixed to 0.001 and the replay buffer size is fixed to 200 000 records.

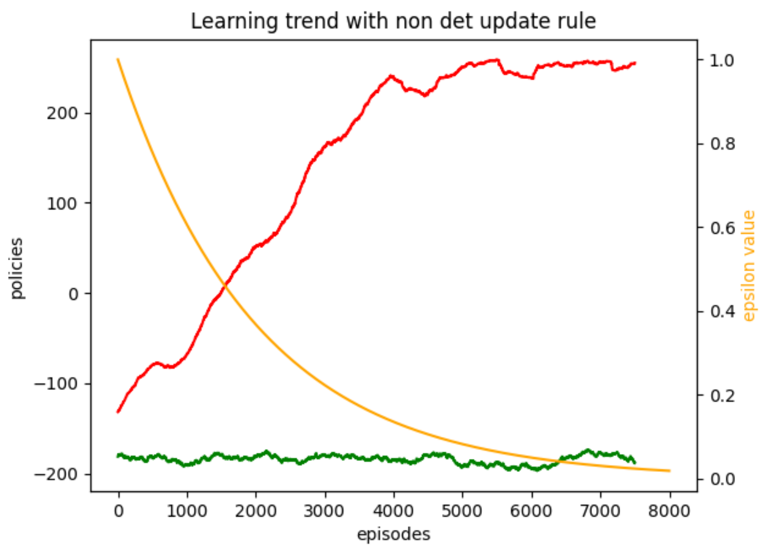


Figure 6: Tabular training process over 8000 episodes with non-deterministic rule

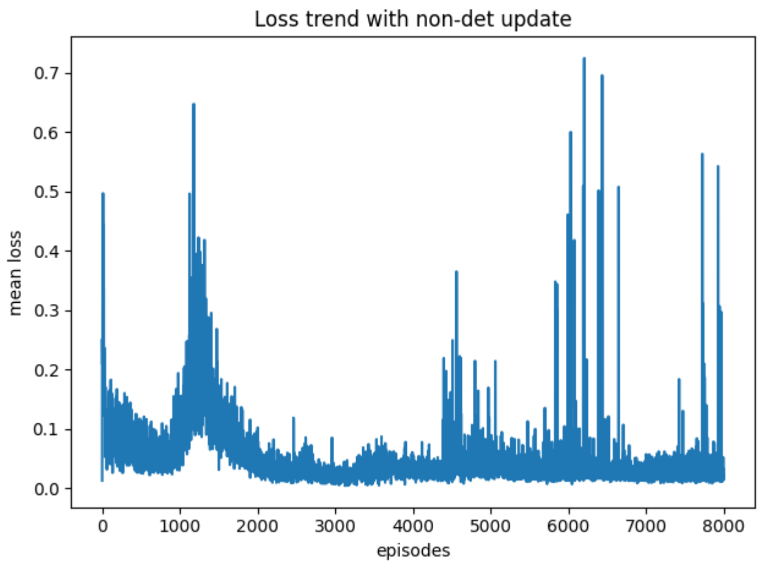


Figure 7: Average loss per episode using non-deterministic rule

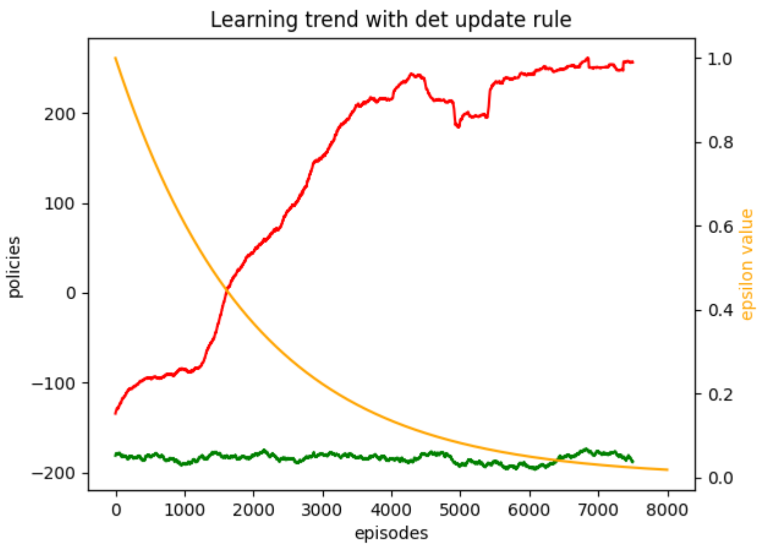


Figure 8: Tabular training process over 8000 episodes with deterministic rule

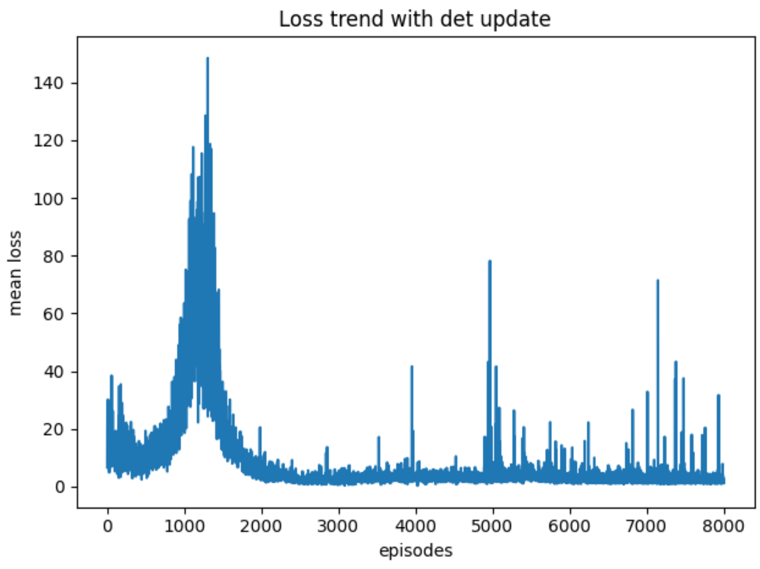


Figure 9: Average loss per episode using deterministic rule

The two graphs show how a Deep Q-Network learns using two different rules: one deterministic and one non-deterministic. Both versions are able to learn well and reach a stable reward of about 250, with similar speed and performance. However, the loss curves are very different. This is due by the usage of a non-deterministic approach, with α, which lead to a soft update to the target, making loss values ranging in different intervals. Still, both versions show an initial peak in the loss followed by stabilization, which shows that the network is learning correctly before reaching a steady state.

Next there is a comparison between the two learning trends.

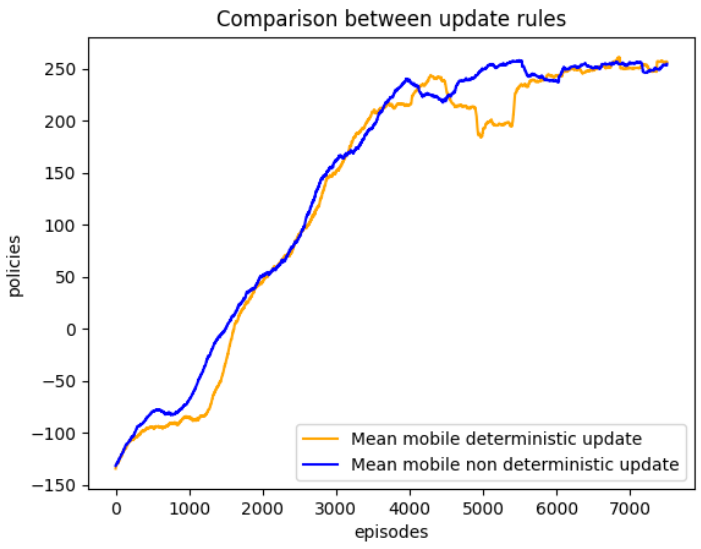


Figure 10: Average loss per episode using deterministic rule

Based on the graph, it is evident that the learning curves of both the deterministic and non-deterministic update rules exhibit very similar trends over the course of the episodes.

This similarity suggests a high degree of interchangeability between the two approaches, as both lead to comparable performance in terms of policy improvement.

4.3.2 Running results

To evaluate the performance of the learned policies, we measured the average reward obtained over 1000 episodes for each update rule. The results are presented in the form of a scatter plot, allowing a direct visual comparison between the deterministic and non-deterministic approaches.

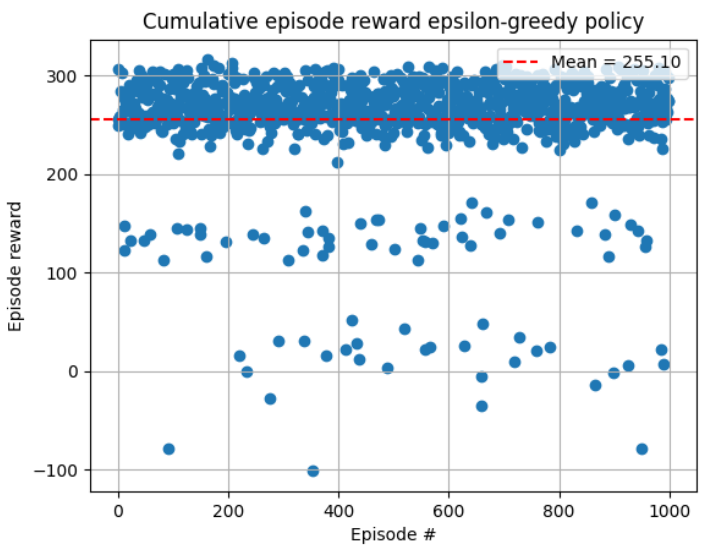


Figure 11: Scatter plot using policy learned with non-deterministic update rule

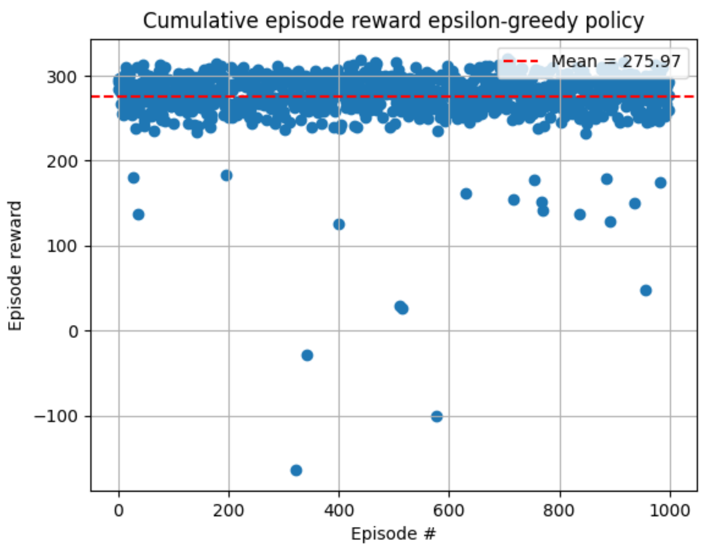


Figure 12: Scatter plot using policy learned with deterministic update rule

In both cases, the policies demonstrate excellent performance, achieving high average rewards over the 1000 evaluation episodes. However, in the deterministic update case, we observe a slightly higher average reward overall, along with a greater concentration of values in the reward range between 250 and 300.

4.4 DQN vs Tabular Q-learning

In the following graphs we present a comparison between the tabular and DQN approaches presenting side-by-side the learning trend graphs shown in the previous sections.

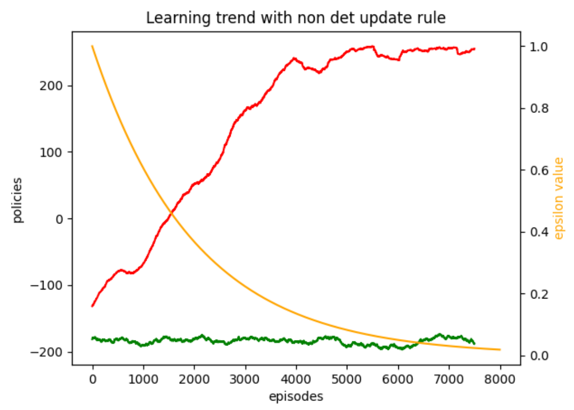


Figure 13: Learning trend using DQN with non deterministic rule update

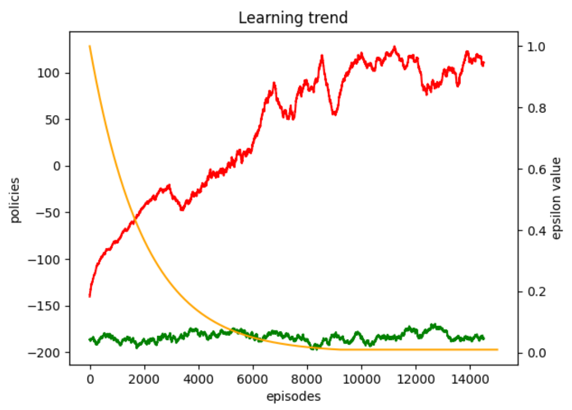


Figure 14: Learning trend using tabular approach

While the DQN approach demonstrates superior overall performance and faster convergence, it is important to note that the results of the tabular method are strongly influenced by the choice of discretization strategy. A well-designed discretization can significantly enhance the performance of tabular Q-learning, as it determines how effectively the continuous state space is represented.

Despite its lower overall reward trend, the average reward achieved by the tabular agent still indicates that the lander successfully performs safe landings in the majority of episodes. This is a very promising result, especially considering the simplicity of the tabular approach. It shows that, with careful design, even classical reinforcement learning methods can yield solid and reliable policies in challenging environments.

5 Conclusion

The goal of this experiment was to solve the Lunar Lander environment using two different reinforcement learning approaches: tabular Q-learning and Deep Q-Networks. The results demonstrate that both methods are capable of learning effective policies, achieving successful landings in the majority of episodes. However, the DQN approach yields superior performance overall, thanks to its ability to generalize optimal actions to previously unseen states. This generalization capability allows the neural network to make more informed decisions in complex and continuous state spaces, leading to more consistent and higher rewards compared to the tabular method.

6 Appendix

We are going to show the implementation of the learning algorithm in both approaches in order to highlight the differences in updating the Q-function $Q(s, a)$, as explained in the previous sections.

```
    def DQN_Learning(self, modality=1, update_modality=1):
        self.eps = 1.0
        self.update_modality = update_modality

        q_network = DQN(8, self.env.action_space.n)

        memory = ReplayBuffer(self.memory_capacity)

        s, _ = self.env.reset()

        total_rewards = []
        eps_per_episode = []
        episodes_loss = []

        for n_episode in range(self.max_episodes):
            episode_reward = 0
            episode_loss = []
            done = False
            n_steps = 0
            while not done:
                a = self._next_action(modality, s, q_network)

                ns, reward, terminated, truncated, _ = self.env.step(a)
                n_steps+=1
                episode_reward += reward
                done = terminated or truncated

                memory.add(s, a, reward, ns, done)

                if ( len(memory) > self.batch_dimension and n_steps \% self.update_every == 0
                ↪  ):
                    batch = memory.sample(self.batch_dimension)
                    training_set = self._prepareBatch(batch, q_network)
                    loss = q_network.train(training_set)
                    episode_loss.append(loss)

                if not done:
                    s = ns

            episodes_loss.append(np.mean(episode_loss))
            eps_per_episode.append(self.eps)
            self._espilon_update()
            s, _ = self.env.reset()
            total_rewards.append(episode_reward)
```

```python
def tabular_QLearning(self, modality=1): # 0 for random, 1 for eps-greedy policy

    self.q_table = defaultdict(lambda: np.zeros(self.env.action_space.n))
    self.eps = 1.0

    s, _ = self.env.reset()
    s = discretize(s)

    total_rewards = []
    eps_per_episode = []

    for n_episode in range(self.max_episodes):

        terminated = truncated = False
        episode_reward = 0

        while not (terminated or truncated): # Single episode
            a = self._next_action(s, modality)

            ns, reward, terminated, truncated, _ = self.env.step(a)
            episode_reward += reward
            ns = discretize(ns)

            self.q_table[s][a] += self.alpha * (reward + self.gamma *
            ↪  max(self.q_table[ns]) - self.q_table[s][a])

            if not (terminated or truncated):
                s = ns

        eps_per_episode.append(self.eps)
        self._espilon_update()
        s, _ = self.env.reset()
        s = discretize(s)
        total_rewards.append(episode_reward)
```