Two approaches for solving the Lunar Lander environment by Gymnasium

Cristian Andrei Mai Mihai

September 2024

1 Introduction

The following report was written as a discussion for my Machine Learning project in my Computer Engineering course. It analyzes the use of Tabular Q-learning and Deep Q-Network (DQN) reinforcement learning algorithms to solve the Lunar Lander environment created with Gymnasium[1]. The approaches consist of a training phase, where the agent gradually increases its knowledge of the environment and its performance, and a test phase, where the agent demonstrates how much it has learned and its ability to solve the environment.

2 Problem formulation

The Lunar Lander environment is part of the Box2D environments, and it consists of a lunar module that must land on the surface of the moon in the safest possible way. During various episodes[2], it was empirically observed that the module's initial state changes, as does the landscape of the environment. However, the landing spot is always set at the center of the generated lunar surface, and these characteristics are determined by the seed of the environment. From this information, it's clear that performing a gentle landing is not easy, and learning strategies must find a solution based on unknown a priori conditions. But how does the environment work in practice? Well, as mentioned before, it is part of the Box2D environments, so it is expected to have a two-dimensional limited domain in which it's possible to work. The following are the main components of the Lunar Lander environment:

[1] At this link, you can find the Lunar Lander environment: link

[2] An episode refers to the series of actions our agent performs from the initial state to a termination state

Action space:

- 0: do nothing
- 1: fire left orientation engine
- 2: fire main engine
- 3: fire right orientation engine

Observation Space:

- X coordinate: [-1.5, 1.5]
- Y coordinate: [-1.5, 1.5]
- X linear velocity: [-5.0, 5.0]
- Y linear velocity: [-5.0, 5.0]
- Angle: [-3.1415927, 3.1415927]
- Angular velocity: [-5.0, 5.0]
- left leg contact with the ground: [0, 1] (boolean)
- right leg contact with the ground: [0,1] (boolean)

Rewards:

- is increased/decreased the closer/further the lander is to the landing pad
- is increased/decreased the slower/faster the lander is moving
- is decreased the more the lander is tilted (angle not horizontal)
- is increased by 10 points for each leg that is in contact with the ground
- is decreased by 0.03 points each frame a side engine if firing
- is decreased by 0.3 points each frame the main engine is firing
- the episode receive an additional reward of -100 or +100 points for crashing or landing safely respectively

Starting State: The lander starts at the top center of the viewport with a random initial force applied to its center of mass.

Episode Termination: The condition for which the episode finishes are essentially three:

1. the lander crashes (the lander body gets in contact with the moon)

2. the lander gets outside of the viewport (x coordinate is greater than 1 or, of course, lower than -1)

3. the lander is not awake, or better it does not move for a certain amount of frames

Based on this structure, an episode is considered successful if it's cumulative reward is greater or equal than 200 points.

Some details about the environment:

- first of all the states are continuous! An important detail that will have a great influence in the following approaches
- the actions are instead discrete, making it easier to choose an action during the training
- the rewards are mostly decrementing, creating a situation where the agent gets punished a lot for bad behaviour, without having great positive rewards in the opposite case.

These factors and their consequences will be further analyzed when I will discuss about the approaches.

3 Solutions

As mentioned in the introduction, in this chapter I will describe the implementation and results of the two algorithms.

3.1 Tabular Q-learning

Before going into the details of the implementation, let's talk about what **Tabular Q-learning** is. The main idea is simple: train over a certain number of episodes, and for each step of the various episodes, save the (state, action) pair and the reward given by the environment in a table. These tuples will then be used by the agent to improve its performance, or rather, to discover the **optimal policy** that must be followed. The optimal policy is a function that, given a state, returns the best possible action for that state; its formulation is the following:

$$\pi^*(s) = \arg\max_a Q^*(s, a) \tag{1}$$

3.1.1 Architecture

Now that we know what we are looking for, I will explain the details of my implementation of Tabular Q-learning.
First of all, it is important to consider that the reward that we store in the table

cannot be the same as the one we receive from the environment, but must be modified using the following equation:

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

This equation states that the value we are looking for depends on the previous value we had for the pair (state, action), but also on the future best reward, giving the agent an understanding of the possible future events when it chooses one action over another. In fact, this concept is very important: the best action now may lead to a worse situation later, compared to a slightly worse action in the moment that may, instead, lead the agent to a preferable state.
In this equation, there are also two parameters that I have not discussed yet: α and γ. The first one is called the **learning rate** and determines how much importance to place on the reward of the next state compared to the previous known Q-value. The second one is the **discount factor** and controls how much influence the next state's Q-value will have in a cascading effect; in fact, the next state's Q-value will be influenced by its own next state, and so on.
So, now we know how to compute the Q-value that must be stored in the look-up table, but how do we choose the action to execute in each state? Intuitively, one might say that the action to execute must be the best-known action in the table, but this approach can lead the agent to a sub-optimal policy, preventing it from solving the environment. Therefore, a different strategy must be adopted: in this paper, the strategy we will focus on is the **ϵ-greedy strategy**. The main idea is the following: in the initial phase, when the agent still does not know much about the environment, it should choose more random actions than "best actions" (where a best action is the one with the highest Q-value for the pair (state, action) in the look-up table). In the later stage, the agent should do the opposite. This way, we explore a greater variety of states initially, and later we exploit the information accumulated to perform better on the best-known path. The implementation of this strategy consists of choosing an initial value for ϵ and slowly decreasing it during training. To choose an action, the agent will draw a random number and check if it is greater or less than the value of ϵ: if greater, it performs the "best action"; otherwise, it performs a random action from the possible ones. In the test phase, ϵ will be set to zero, so the agent always chooses the best action.
In the "Problem Formulation" chapter, I discussed important details of the Lunar Lander environment, particularly the presence of **continuous states**. The table created by the Tabular Q-learning algorithm must be read and updated at every step of every episode, so a very large table would be unfeasible, which is exactly the problem posed by having continuous states. Therefore, we must first discretize the state space in a more manageable way. There are different ways to achieve this discretization, but the method I chose is **Tile Coding**.
But how does Tile Coding work? The idea is as follows: create several grids over the state space, and when a value falls within a tile (a cell of the grid), convert that value into the index number of the corresponding tile. The tiling must be done for every dimension of the state, and the number of tiles is a hyperparam-

eter, which I will discuss later. For the last two values of the observation, which range between [0, 1], it is crucial not to divide them into tiles, but instead treat them as boolean values.
As mentioned, Tile Coding uses multiple grids. Without these, one would not achieve much difference compared to simpler methods like rounding the state values. The number of grids is also a hyperparameter, and the offsets between the grids are generated randomly (these offsets are necessary to prevent overlapping of the tiles). Using multiple grids is essential to improve generalization and avoid aliasing, where different states might be grouped together as if they were the same.
One important detail to consider is the update rule: earlier, I mentioned that we often choose an action based on its Q-value for a given state. However, we are no longer storing the state directly; instead, we store a tuple composed of three values: the index of the grid, the converted state values, and the action with its corresponding Q-value. As such, in the update rule, to compute the Q-value for a given state-action pair, it is necessary to sum all the Q-values (one for each grid) for which the state corresponds to the converted value. The Tile Coding mechanism in the algorithm is managed by the TileCoder class.
Earlier, I mentioned hyperparameters without explaining what they are: **hyperparameters** are parameters whose values can significantly influence the final result. Therefore, these parameters need to be fine-tuned. Typically, this involves trying out different combinations to find the one that best fits the algorithm.
In my case, there are several hyperparameters, and I will now present the values I used for each of them:

- α: 0.2 / 10 (the division for 10 it's necessary because of the use of 10 grids in the tiling)
- γ: 0.99
- number of episodes: 75000
- ϵ: 1.00
- minimum ϵ 0.01
- ϵ decay: 0.9999
- number of tilings (or grids): 10
- dimensions of the tiles: [18, 18, 18, 18, 18, 18, 2, 2]

3.1.2 Experiments results

After discussing the architecture of the algorithm, let's now take a look at some numbers. I will present the results obtained from the training and testing phases, using the hyperparameters mentioned earlier.
First, the time needed for training: **2420.73 seconds**. While this might seem

like a large number, it's actually quite reasonable. Although the execution of a single episode is almost instantaneous, the process must be repeated for a large number of episodes. Additionally, it's important to note that the time required for each episode increases as the training progresses. This is because the size of the look-up table grows almost linearly with the total number of steps performed by the agent.
Now, let's examine the results of the training.

Figure 1: Average, max and minimum rewards over 500 episodes

In Figure 1, we can observe three lines: the maximum value in orange, the average value in blue, and the minimum value in green, all tracked over 500 episodes during training. The red vertical lines indicate the changing values of ϵ over the course of the episodes.
But what can we infer from this graph? Each line provides important insights into the training process and its progression. For instance, the orange line stabilizes around a maximum of 300 points after only 20,000 episodes, while the green minimum line fails to stabilize and seems to never converge to the desirable threshold of 200 points. Reaching this threshold would indicate that, in the worst case, the agent can still solve the environment.
The interpretation of this information is as follows: the agent cannot consistently exceed 300 points, no matter how much additional training is done, and at the same time, it struggles to find a universally effective solution for the environment. The cause of this behavior lies in the reward system of the Lunar Lander environment, which, as I mentioned earlier, imposes very punishing penalties. This system tends to create scenarios where the agent can either succeed dramatically or fail spectacularly. This is confirmed by the wide range of potential rewards: 300 points as the upper bound and almost -600 points as the lower bound. A high average score indicates many successful episodes with only a few failures. However, this is not always the case, there are episodes

where the agent may not fully complete its task but still manages to land, albeit imperfectly (for instance, landing too quickly or off-target).
The most interesting part of the graph, however, is the blue average line. This line reflects the agent's overall behavior across 500 episodes, not just its performance in a single episode, where a single bad action can result in a crash. The average line provides a more general view of the agent's performance and more clearly shows whether the agent is learning to solve the environment. After initially converging to a value higher than 200 points, the maximum line loses its significance, while the minimum line fails to capture the agent's general behavior accurately.
When analyzing the evolution of the average line over the episodes, it's clear that it grows steadily over time. It starts at around -200 points and stabilizes around 200 points after about 35,000 episodes. This is precisely what we want to see: it shows that the agent is learning and improving, gradually converging to a near-optimal policy.
Now, a reasonable question might be: why not stop the training after the average appears to stabilize around 200 points, as it seems the agent can't improve beyond this? The answer is that this assumption is not entirely true. I allowed the agent to train for almost twice the number of episodes needed to seemingly reach the optimal policy in order to "cement" its knowledge of the system. This decision is supported by empirical observations, as the agent performed significantly better when trained for a longer period.
The red vertical lines in the graph do not provide direct insights into the agent's performance but help to illustrate the impact of the ϵ-greedy strategy. When the agent is exploring more, it is more likely to take suboptimal actions, which can result in lower rewards. Toward the end of the training, especially when the value of ϵ reaches 0.05, the agent almost exclusively chooses the best actions, leading to more consistent and improved results.

Now, let's examine what happens during the test phase, where the agent no longer updates the look-up table and focuses solely on exploiting its knowledge by executing only the best actions.

Figure 2: Rewards of the 100 episodes in the test

The success rate is 90%, with only 1 episode resulting in a negative reward. This indicates that the agent performs well in most cases. However, there are still some episodes where the agent cannot achieve a safe landing and instead results in a more forceful landing. Nonetheless, the agent now fails significantly less often compared to when it was allowed to choose random actions.

3.2 DQN

As before, I will now introduce the second approach for solving the Lunar Lander environment: **Deep Q-Learning (DQN)**. This algorithm is similar to Tabular Q-learning, but instead of using a table to represent the optimal policy, it aims to reconstruct the policy using a **Neural Network**.

3.2.1 Architecture

There are some common aspects in the architecture of the two implementations, so I will briefly mention them. We still use the same update formula (though with slight modifications) and the same ϵ-greedy strategy.
At the same time, there are various new components in the DQN algorithm that should be explained. First, the neural network is structured as follows:

- An **input layer** composed of neurons equal in number to the variables in the observation space (in our case, 8).
- A **hidden** layer composed of 24 neurons.

- An **output** layer composed of neurons equal in number to the actions in the action space (in our case, 4).

The choice of these numbers of neurons was influenced both by the nature of the environment and by tuning the hyperparameters. The dimension of the input layer must match the number of variables in the observation space because these are the features that the neural network needs to examine for making predictions. The output layer must have the same dimension as the action space because we want to output the Q-values for each action and select the one with the highest value. The dimension of the hidden layer, as well as the number of hidden layers, is a hyperparameter. The number of hidden layers was not tuned and was kept to one to avoid excessive complexity in the neural network. The number of neurons in the single hidden layer was chosen through empirical experimentation.
Between the layers, the activation function used is ReLU, while no activation function is applied after the output layer, as we want the outputs to be linear. The ReLU function is defined as follows:

$$\text{ReLU}(x) = \max(0, x)$$

As with the previous approach, the agent has various hyperparameters, and I will now list them along with the values I used for each. I will explain the use of some of these hyperparameters, which have not been discussed yet, later on:

- Dimension of the memory buffer: 10000
- γ: 0.99
- Learning rate: 0.001
- α: 0.1
- ϵ: 1.00
- minimum ϵ: 0.01
- ϵ decay: 0.995
- number of episodes: 1000
- size of the batch: 32
- frequency of model update: 4

The first of these values introduces a new concept: the **Replay Buffer**. It is a type of queue that stores tuples, somewhat similar to the look-up table in the previous approach, but with fundamental differences. The values stored in the Replay Buffer include the state, the action performed, the given reward, the next state, and the notion of done (whether the episode has terminated or not). The usage of this "table" differs from that in Tabular Q-learning: instead of

using it to choose an action to execute, we use it to replay past experiences. This process helps the model or agent break the temporal correlation of the states, avoid learning from continuous states, and stabilize the learning process. Here's how the agent operates with the Replay Buffer: it performs an action in a given state by either choosing randomly or selecting the best action based on the state input to the neural network, saves the resulting tuple, and then performs a "replay." During replay, a batch of samples from the Replay Buffer is used. Targets are computed using the update formula (using the reward itself if done is true, otherwise using the formula), and these values are passed as input to the model. To speed up the process, states are passed together as a matrix and targets as a vector, rather than one by one. The optimizer, which adjusts the weights of the nodes to minimize the loss function, then resets its gradients, the model performs predictions on the given states, the loss is calculated using a certain criterion, and backward propagation is performed. Finally, the optimizer executes a step to adjust the model's weights. The **optimizer** chosen is Adam, and the learning rate is one of the hyperparameters. The **criterion** used is Mean Squared Error (MSE). Replay is conducted only if the number of experiences in the buffer is greater than or equal to the batch size.
After executing the replay, the algorithm continues with the new state and repeats the process.
A final detail about the architecture of the DQN is the use of two models: one that is updated continuously and one that is updated every 4 episodes. This approach helps stabilize the learning process. Since the neural network aims to approximate a function that changes during training, using the same model for both prediction and target computation may lead to instability. To address this, the targets during replay are computed by the target model rather than the standard model, ensuring more stable learning and preventing the issue of chasing a moving target.

3.2.2 Experiments results

As with the previous algorithm, we will examine the results of the experiments conducted on it, starting with the time required for training: **4253.05 seconds**. This is nearly double the time compared to the other solution, despite requiring significantly fewer episodes. The reason for this extended duration is due to the use of the "replay" function and the complexity of the operations performed by the neural network. Specifically, the agent must replay 32 previous experiences at each step, and the computations involved in the neural network are far more complex than simply retrieving a tuple from a dictionary.
Let's now review the results of the training:

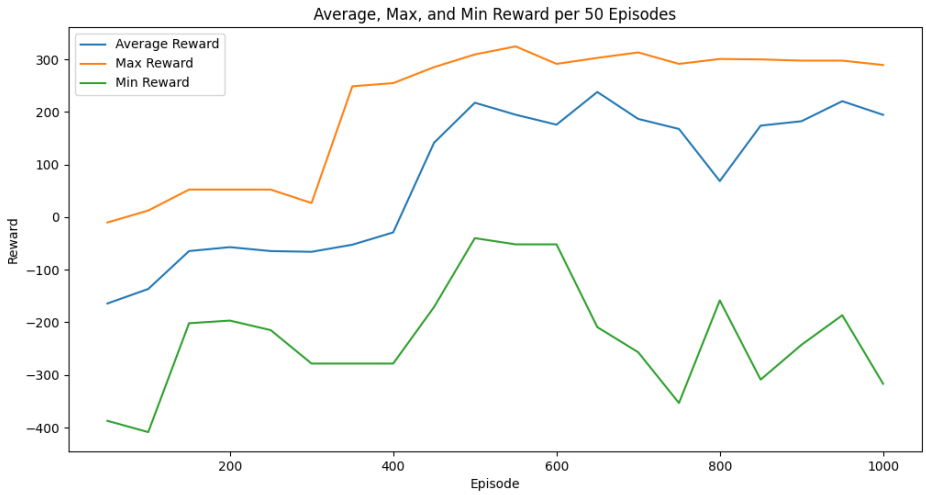

Figure 3: Average, max and minimum rewards over 50 episodes

The graphs presented in Figure 3 resemble those in Figure 1, with maximum reward shown in orange, average reward in blue, and minimum reward in green (data are collected over 50 episodes each). As before, the orange line does not surpass the value of 300 points, confirming that this value can be considered an upper bound. The green line also fails to stabilize. However, the average line increases over the number of episodes, reaching a value of 200 points after only 500 episodes, and converges at this value. As with the previous approach, training must be conducted for more episodes than those required to achieve an average of 200 points, to confirm the result multiple times. Nevertheless, the average line's upward trend indicates that the agent is learning and improving its performance over time.

The graph in figure 4 shows instead the rewards over the 100 episodes of testing.

Figure 4: Rewards of the 100 episodes in the test

Let's start by noting that the success percentage in the DQN approach is 87%, with only 2 episodes performing poorly (scoring lower than 0). As with the previous approach, the agent has learned how to act effectively in most scenarios within the environment but has not yet found a policy that can be considered optimal. However, its performance has improved compared to the cases where the lander executed only random actions.

4 Conclusions

Let's conclude this report by analyzing and comparing the two solutions described.
Firstly, both algorithms perform well in the Lunar Lander environment, achieving success rates of 90% and 87%, respectively, and more importantly, average rewards over the test phase greater than 200 points. However, neither approach could find a universally good solution, which could be attributed to several factors, including:

- The number of training episodes for both methods might have been too low, and the agent could have needed more time to better understand the environment.
- The general tuning of the hyperparameters may have been insufficient, with potentially unexplored combinations of values that could lead to better results. This is particularly relevant for the dimensions of the tiles in

the Tabular Q-learning and the structure of the neural network in the DQN.

It is particularly notable that the Tabular Q-learning performed better than the DQN, which might be unexpected given that DQN generally offers better generalization and directly uses the state information without discretizing it, as done in the Tabular Q-learning approach. This result is likely due to less extensive experimentation with the DQN algorithm; the training time for the DQN was significantly greater than for the Tabular Q-learning.
Another interesting comparison metric is the training time itself. In this case, Tabular Q-learning is faster, but this could be influenced by the hardware used. The first algorithm runs exclusively on the CPU, while the second utilizes the GPU as well. The quality of these hardware components can significantly impact the final results.
Finally, while the Tabular Q-learning provided better results in this instance, it is likely that with further hyperparameter tuning, the performance of the DQN could improve substantially.