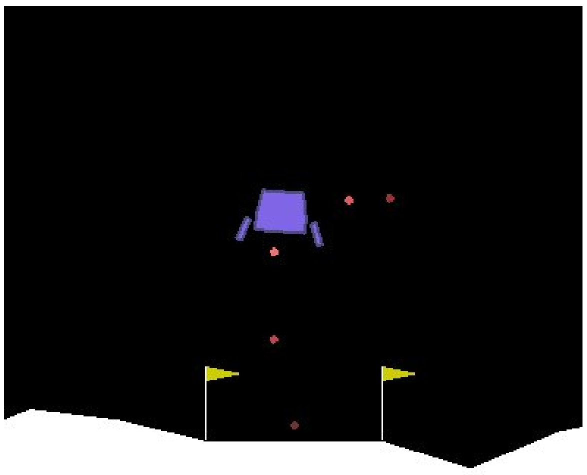
Delft University of Technology

Bio-inspired Intelligence and Learning for Aerospace Applications
AE4350

Driving a Lunar Lander with Deep Q-Learning

By:
Gioele Buriani (5629888)

September 1, 2022



Contents

1 Introduction

1.1 Choice of the project

Among all the topics of the course, reinforcement learning was definitely the one that interested me the most. One of the reasons might be that, even without knowing it, reinforcement learning applications have been the ones that pushed me most towards studying robotics: a legged robot that literally learned how to walk or a computer that learned how to beat the world champion of chess. For these reasons, I had no doubts about choosing this particular field of machine learning for this project.

As for where to apply it, at first, I considered trying to apply it to games I used to play since I was younger, such as 2048. By working on it, however, I realized that, in order to achieve a satisfactory result, I would have had to spend most of my time building the environment, thus disregarding the scope of this course that was more directed to the learning itself. For this reason, I started looking at Open-AI Gym for an interesting pre-built environment where I could concentrate more on the learning part and my attention was caught by the Lunar lander application. I have always had a passion for space applications (I am doing an internship at DLR in Germany and I would like to work at ESA one day), so I thought that this could be a challenging and at the same time fun application for my project.

In the end, therefore, this project will consist of the application of reinforcement learning algorithms for the control of a lunar lander.

1.2 The environment

As mentioned in 1.1, I am going to use the lunar lander environment from Open-AI Gym. More precisely, the environment I will be using is OpenAIs LunarLander-v2 Gym Environment [1].

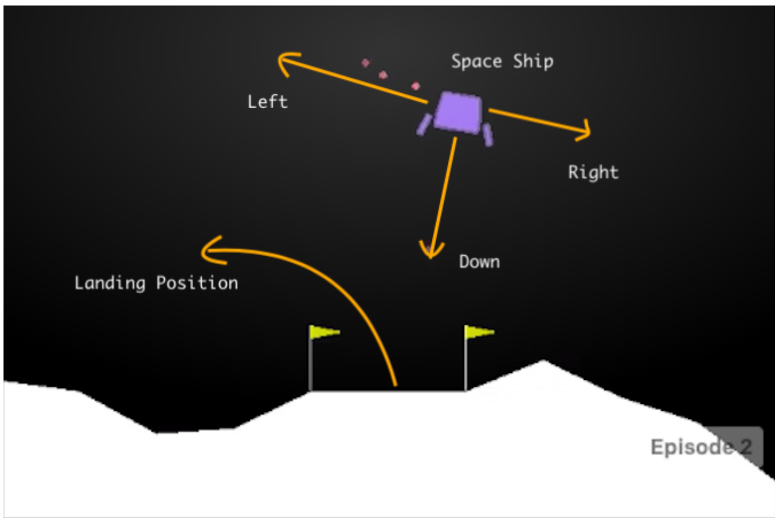


Figure 2: The LunarLander environment

As can be seen from Figure 2, the environment consists of a ground where two flags stand. The two flags signal the landing point and the goal is to apply the correct thrust to the spaceship's engines in order to make it safely land in the landing area.

This environment has several important features that I will now list:

- Fully Observable: all the state information is known at every time frame
- Mixed Observation Space: the state space has 8 values:
  - X direction distance from target site (continuous)
  - Y direction distance from target site (continuous)
  - X direction velocity (continuous)
  - Y direction velocity (continuous)
  - Angle (continuous)
  - Angular velocity (continuous)

  - Left leg ground contact (binary)
  - Right leg ground contact (binary)

- Single Agent: there is only one agent, no cooperation or competition is involved
- Discrete Action Space: the action space only comprehends 4 actions:
  - thrust
  - nothing
  - left
  - right
- Static: there is no penalty or state change during action deliberation
- Episodic: the reward is dependent only on the current state and action
- Deterministic: there is no randomness in the effects of actions or the rewards obtained
- Finite Horizon: the episode ends after a successful event, a failure event or after 1000 time steps

1.3 The algorithm

As mentioned in 1.1, my intention to tackle this problem was through reinforcement learning. Thus, since we were dealing with a single process with a discrete action-space, my first idea was to use one of the most used off-policy algorithms: Q-Learning [2].

However, the state space for this problem is continuous, meaning that I would need to use infinite Q-functions for each state-action pair. In order to solve this problem, there is the need for a function approximator, such as a neural network, in order to estimate the Q-functions. For this reason, using a classic Q-learning algorithm would have been highly inefficient and, therefore, I decided to instead use a more powerful algorithm: Deep Q-Learning [3], or DQN in short.

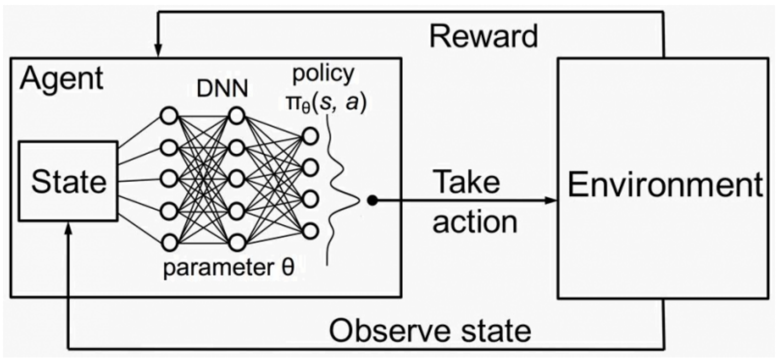


Figure 3: Graphical representation of DQN

Basically, as can be also observed from Figure 3 and Figure 4, the main difference between classic Q-learning and Deep Q-Learning is that the agent in itself contains a neural network that allows assigning to each input state the best possible action following its training. In other words, when the number of states increases a lot, like in our case, it becomes rather unmanageable to keep a separate Q-function for each state-action pair and thus the standard Q-table is substituted by a more functional neural network. While training, based on the reward mechanism, the agent's neural network adjusts its weights in order to progressively select output actions that gain a better reward for each input state.

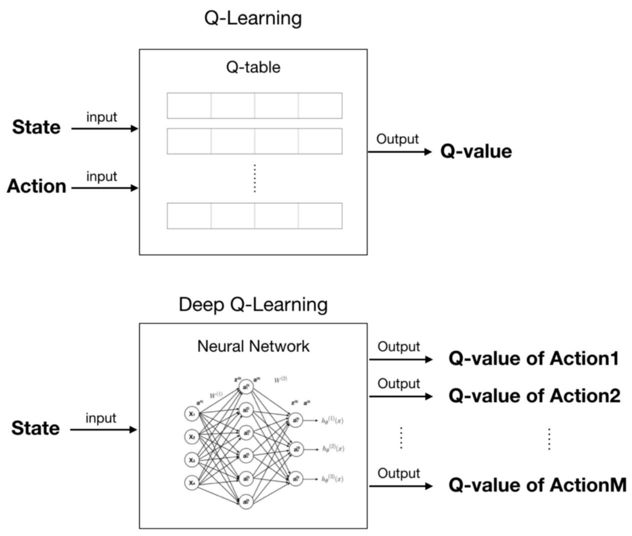


Figure 4: Graphical comparison between Q-Learning and DQN

In particular, as will be further explained in Section 2, this algorithm will also be boosted with the use of a target network to improve stability and an experience replay buffer to improve sample efficiency. The overall result, as expected, will thus show slow convergence but high efficiency.

2 Implementation

2.1 Reward mechanism

After describing the environment's characteristics and the selected algorithm, we need to discuss the reward mechanism. Rewards are one of the main elements behind any reinforcement learning algorithm and thus are fundamental to its success. A high reward will "teach" the network that the selected action is positive, a low reward will have the opposite effect. To be fair, using the OpenAI Gym environment, the reward system is predefined in the environment, but I still think it is worth mentioning.

In general, after each step, a reward is given based on the current state and the action taken. It is worth mentioning the importance of always associating a reward to a state-action pair, and not only to a state itself, in order, for example, to avoid the agent just sitting in a state with a higher reward without taking any action.

Each episode is created by various time steps: the total reward of the episode is the sum of all the steps within the episode. In general, the closer the lander gets to the landing point, the higher the reward. For example, if the lander finds itself at the landing site with zero speed it has accumulated a total of between 100 and 140 reward points. Also, the agent receives an extra 100 for landing or -100 for crashing. on top of that, grounding one of the legs is worth an extra 10 points (20 points for both legs). On the other side, thrusting the main engine creates a penalty of -0.3 points. Overall, a solution is found when 200 points are reached within the episode (the lander landed correctly at the landing site without thrusting the engine too much).

From a more formal point of view we can define the environment E can be defined by functional mapping. In particular, when the environment E is in a state s (this will be represented with the symbol E_s) and receives an action a, it will produce a reward r, a next state s' and an episode termination flag d. The mapping can thus be expressed as:

$$E_s(a) \longrightarrow (r, s', d) \tag{1}$$

In other words, when an action is applied to an environment in a certain state, this will create a reward, a change in state in the environment (or lack thereof) and will either terminate or not the episode.

This set of variables will be used throughout the project as the experience tuple $< s, a, r, s', d >$.

The relationship that returns an action for a given state is called the policy $\pi(s) \longrightarrow a$ and is what we want our agent to learn.

We can then better formalize the reward system and define the cumulative value of an episode as:

$$V^\pi(s) = E_s(\pi(s))_r + \begin{cases} 0 & \text{if } E_s(\pi(s))_d \\ V^\pi(E_s(\pi(s))_{s'} & \text{otherwise} \end{cases} \tag{2}$$

This means that the total value will be the reward of the current step plus zero if the episode is terminated or plus the rewards of the next steps otherwise.

2.2 Q function

For the algorithm itself, I followed a similar approach to the one proposed by Mnih et al. when using DQN to play Atari games [4].

Basically, I first reformulated the problem as a Markov Decision Process, in order to be able to solve it using Bellman's equations, so that:

$$V^\pi(s) = \max_a(R(s,a) + \gamma V^\pi(s')) \tag{3}$$

where $E_s(a)$ was rewritten as s', V is the value function, R is the reward function and γ is the discount factor.

At this point it is possible to define the Q-form of the three main equations:

$$V^\pi(s) = \max_a Q^\pi(s,a) \tag{4}$$

$$Q^\pi(s,a) = R(s,a) + \gamma V^\pi(s') \tag{5}$$

$$\pi(s) = \arg\max_a Q^\pi(s,a) \tag{6}$$

It is worth noticing that, since the Q values are approximated by a neural network of parameters θ (that will be further discussed in 2.3) the expression of $Q^\pi(s,a)$ needs to be changed in $Q^\pi(s,a,\theta)$.

Now we can calculate the difference between the predicted Q value and the sum of the reward and the target Q value, represented by the next state's best Q value. The expression for this difference, called error, will be:

$$\text{error} = Q(s,a,\theta) - (r + \gamma \max_{a'} Q(s', a', \gamma^-) \tag{7}$$

At this point, we can calculate the losses for the neural network using Mean Squared Error as:

$$L(\theta) = E_{(s,a,r,s')}[(r + \gamma \max_{a'} Q(s', a', \theta^-) - Q(s, a, \theta))^2] \tag{8}$$

where θ^- is a set of wights belonging to a target network that will be further discussed in 2.4.2.

Now we have a parameterized Q function of which we can calculate the loss based on a set of weights and parameters.

2.3 Neural network architecture

As previously mentioned in 1.3, since the state-space is continuous, it is impossible to find a single Q-function for each state-action pair. In order to cope with this problem, it is necessary to introduce a function approximator that allows one to find a single discrete action for each state configuration. This function approximator takes the form of a neural network.

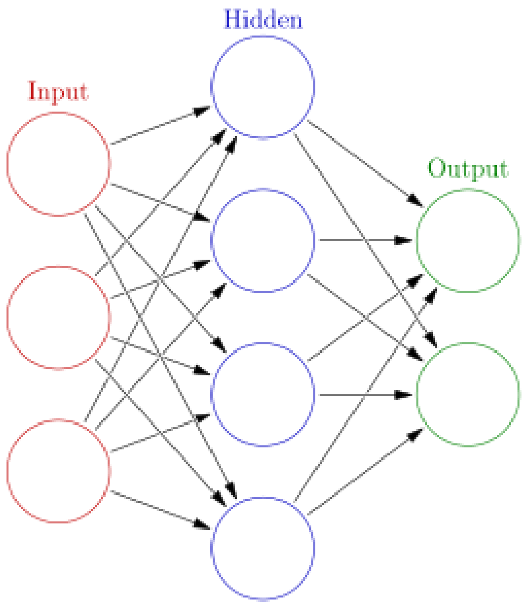


Figure 5: Graphical representation of a neural network

To create the neural network I simply used the base class for all neural network modules provided by PyTorch [5].

I chose a very simple architecture in order not to over-complicate the problem and it appeared to be enough for a satisfactory result. I decided to use only one hidden layer (the blue one in Figure 12) made of 64 neurons, which is one of the standard architectures to start from. Obviously, I also needed an input layer (the red one in Figure 12) and an output layer (the green one in Figure 12). The number of input and output neurons was obviously fixed: the neural network is fed in a state (of dimension 8) and needs to give out an action (belonging to an action space of dimension 4). Therefore, the neural network will have 8 input neurons, a hidden layer of 64 neurons and 4 output neurons.

Overall, as an activation function I used ReLU since, once again, is the standard one and proved to work well. Also, I chose to use Adam optimizer for the same reason.

2.4 Algorithm improvements

After describing the differences between standard Q-learning and Deep Q-Learning in terms of Q function approximation by means of a neural network, the rest of the procedure is the same as for standard Q-learning.

I implemented a standard ϵ-greedy strategy, thus the agent acts on the environment with either a random action or following the current policy (more on this matter in 2.5.3). Each action triggers a reward and a change of state. The algorithm keeps acting on the state while at the same time learning which actions generated good rewards and which did not in specific states. The best state-action pairs are "remembered" by the algorithm in the form of neural network weights through a training procedure. In the end, the neural network is able to determine for each state which is the best action to take: we created the best policy.

This, however, is simple and straightforward when described with words, but is harder to set up in practice. In fact, the presence of a neural network within the algorithm introduces some problems to it related, such as instability and lack of convergence to an optimal value function. Because of the high correlation between actions and states, in some cases, the network weights can oscillate or even diverge.

For this reason, I implemented two techniques that are widely used to take care of these problems: experience replay [6] and target network [7].

2.4.1 Experience replay

One of the main requirements when training a neural network for a supervised learning problem is that the training data is Independent and Identically Distributed, IID in short. This is due to the fact that we want the network to be able to classify each input regardless of the correlation it might have with the rest of the data. On the contrary, the correlation biases the network that, therefore, performs much more poorly when given uncorrelated data.

In the DQN case, the experience tuples fed to the network are highly correlated since they are detected in a sequential fashion. For this reason, we need to find a way to be able to use these experiences, but in a random order so that we break the data correlation.

A solution to this problem is using an experience replay buffer: we save all our previous experiences and, instead of using the latest ones (that will appear in sequential order) to train the network, we sample a random batch of experiences from the buffer and train the network with those. Basically, experience tuples (S, A, R, S') get saved into the buffer (of fixed size) as the interaction with the environment takes place by at the same time pushing much older experiences out of the buffer. In this way, we end up with relatively new experiences, but of such a number out of which a small batch can be sampled at random. In my implementation, for example, I used a buffer of size 10^5 and a mini-batch size of 64.

The act of storing past experiences in a buffer and randomly sampling small batches of them to train the neural network is called experience replay.

2.4.2 Target network

In standard Q-Learning we usually update a guess with a guess: when dealing with neural networks, once again, this can potentially lead to harmful correlations. The Bellman equation gives us the value of $Q(s, a)$ thanks to $Q(s, a)$. However, by definition, the two states s and s have only one step between them. This makes them very similar, and its very hard for a Neural Network to distinguish between them. During training, when we update the Neural Networks parameters to make Q(s, a) closer to the desired result, we can indirectly modify the value produced for Q(s, a) and other states nearby. This can make the training very unstable.

To make training more stable, there is another strategy, called target network. This strategy consists in keeping a copy of the neural network and using it for the Q(s, a) value in the Bellman equation. In this way, the predicted Q values of this second Q-network, called the target network, are used to back-propagate through and train the main Q-network. It is important to highlight that the target networks parameters are not trained, but they are periodically synchronized with the parameters of the main Q-network.

All in all, it has been proven that using the target networks Q values to train the main Q-network will improve the overall stability of the training.

Moreover, there is a further technique that I decided to apply in this case which is called soft update. This basically consists of the fact that the target network does not get updated all at once but by frequent small updates based on a parameter τ. In [8] the author propose an algorithm called DPG in which they used a value of $\tau = 0.001$. The target network then gets updated as:

$$\theta^- = \theta \times \tau + \theta^- \times (1 - \tau) \tag{9}$$

Since the update is small and thus has to be frequent, I triggered the soft update whenever the agent learns.

2.5 Hyper-parameters tuning

In 2.3 and 2.4.2 we already briefly mentioned the choice of hyper-parameters for the neural network and the soft update. There are, however, some hyper-parameters related to the Q-learning algorithm that needs to be explained.

2.5.1 Learning rate α

The learning rate α in standard Q-learning represents the rate of change of the Q-functions. A high learning rate means that the Q-functions change rapidly, but could change too much and miss the optimal values. On the other hand, a low learning rate leads to a higher probability of finding the optima but takes more time to learn. The possible values are in the range $\alpha \in (0, 1]$ and a standard good value is around $\alpha = 0.5$.

In our case, however, the Q-table is substituted by a neural network, which means the learning procedure is completely different. In fact, the learning rate now signifies the rate of change of the weights of the neural network. Even though the concept is the same, the actual implementation of the values changes, and with it the possible values.

Indeed, I started assigning to learning a value of $\alpha = 0.5$. I soon realized that this value was way too high and the neural network learning diverged. I tried to gradually lower the value until I found a fitting value of $\alpha = 0.0005$ which actually allowed the network to converge.

2.5.2 Discount factor γ

The discount factor γ is another hyper-parameter that needs to be considered when dealing with reinforcement learning. In fact, the learning goal to find the optimal policy is to maximize what is known as discounted return, which is defined as:

$$R^{\pi}(x_0) = \sum_{k=0}^{\infty} \gamma^k r_{k+1} \tag{10}$$

In fact, when choosing an action, in most cases we are not interested in taking simply the one that yields the biggest initial reward, but we also want to consider the future steps. For example, we do not want the lander to charge the landing station at full speed: even though in the first steps it will get closer thus receiving higher rewards, it will most likely crash in the end, ending in a failed attempt.

For this reason, it is important to choose a fitting value for the learning rate as $\alpha \in [0, 1)$, where $\alpha = 0$ means there is no interest in the future. The role of γ is therefore to induce a sort of "pseudo-horizon" for optimization while at the same time encoding an increasing uncertainty about the future. This has been proven to help the convergence of the algorithm.

Since I wanted to take into great consideration the future step for a problem like this one, I decided to choose a standard value for the discount factor as $\gamma = 0.99$.

2.5.3 Exploration probability ϵ

As mentioned in 2.4, for this project I adopted the widely used ϵ-greedy strategy. The last hyper-parameter to be tuned pertains exactly to this strategy and it is the exploration probability ϵ.

Basically, when choosing the action to be performed on the environment, the agent has two possibilities: exploration or exploitation. The exploration consists in choosing a random action to try out something new and *explore* some new possibilities. On the other hand, the more normal procedure is to follow what at the moment is thought to be the best possible action, in other words, we *exploit* the current policy. Both actions are important: without exploration, the agent would only follow the first policy, most likely missing all the better ones, without exploitation the agent would just keep acting randomly without following any policy. For this reason, there needs to be a good balance between the frequency of choice of the two different actions. Here is what ϵ is used for: whenever it has to pick an action, the agent will have a probability of ϵ of exploring (random action) and a probability of $1 - \epsilon$ of exploiting (follow policy). In a more formal way:

$$a_k = \begin{cases} \arg\max_{a^*} Q(s_k, a^*) & \text{with probability } (1 - \epsilon) \\ \text{random} & \text{with probability } \epsilon \end{cases} \tag{11}$$

Usually, we want some exploration, but we do not want it to happen too often otherwise it will take much more time to reach the whole optimal policy. For this reason, a typical value for the exploration probability is $\epsilon = 0.1$. However, ideally, we would like to start with higher exploration in order to immediately open as many "roads" as possible, while later in the training we would like to focus more on exploitation in order to optimize the best policy. For this reason, a new technique to manage ϵ has been created: the diminishing schedule. The idea is that ϵ starts with a very high value of $\epsilon = 1$, but then decreases until it reaches a minimum value of $\epsilon = 0.01$. Obviously, the episode might terminate (and often does) before ϵ reaches its minimum value. The expression of epsilon per episode is:

$$\epsilon_{k+1} = d\epsilon_k \tag{12}$$

where d is a variable called epsilon decay whose value is $d = 0.995$.

3 Results

After describing the procedure that I followed to create this project, it is time to see what results came out of it.

3.1 Epsilon diminishing schedule

This brief section is only here to demonstrate the working of the procedure described in 2.5.3. As previously explained, one of the most efficient ways to manage the exploration-exploitation trade-off is to use an ϵ-greedy policy with ϵ on a diminishing schedule.

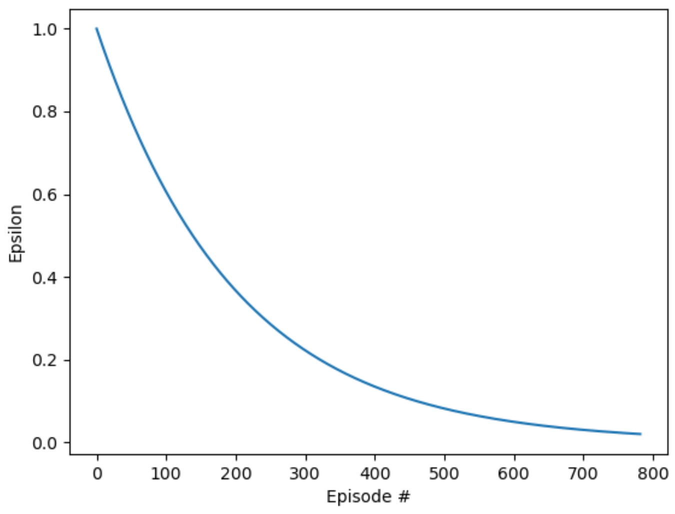


Figure 6: Value of ϵ decreasing with each episode

As can be seen from Figure 6, the value of ϵ starts at $\epsilon = 1$ and then decreases with inverse proportionality (of factor $d = 0.995$) for each episode, until it reaches a value very close to zero.

3.2 Score per episode

Now we finally get to what is probably the most significant result for this project: how much the algorithm actually learns and improves for each episode.

First I need to make a remark regarding the randomness of this algorithm. In fact, both the neural network and the Q-learning algorithm have a random part.

For the neural network, the randomness is related to the initialization of the weights: the initial weights of the neural network get initialized with random values, before starting to get updated at each iteration. This can create quite significant changes in terms of convergence times: if the weights have a "lucky" initialization, the network might converge much faster to an optimal value. Obviously, if the initial values are "unlucky" the opposite behaviour occurs.

For the Q-learning algorithm, the random part was already explained in 2.5.3. As was explained, when exploring, the agent chooses an action randomly. Once again, it might choose some "lucky" actions that immediately improve the policy, but might also fail to do that for several iterations.

For these reasons, the same algorithm might have quite different performances based on the "luck" of the trial. For this reason, it is important to test the consistency of the algorithm in several different trials. More on this in 3.3.

In order to obtain some repeatability of the performance despite this randomness, it is possible to set the random seed of the trial to a certain number. This means that the choices keep being random, but the same random choices are taken at all trials. This is important when modifying the algorithm in order to improve it, but, once again, needs to be tested for generalization in the end.

For all my trials I kept a random seed of 42.

After this premise, we can now see the improvements in the score per episode in Figure ??.

As can be seen, in the figure there are four different graphs, even though they all represent the same metric. The first is the actual score per episode, but, unfortunately, is quite messy and therefore makes it hard to really be able to notice the improvements of the agent.

For this reason, I decided to filter out the data in order to have a better visualization. To do so I applied a moving average filter to the data of various windows: the windows are 10, 20 and 50 elements. This filter averages out close data in order to reduce the noise and the result can be appreciated in the second, third and fourth graphs. It is important to understand that the initial data remains the same, this is only a way to better visualize the improvement.

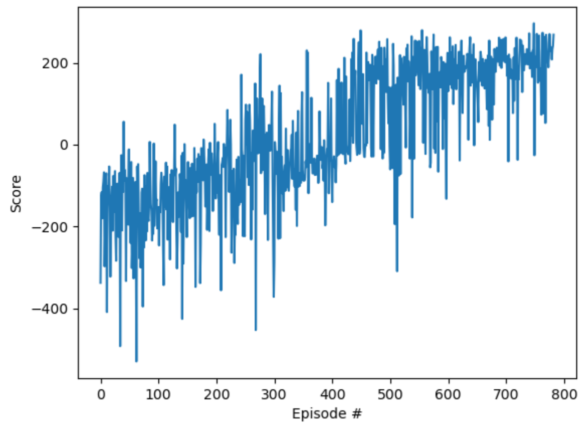


Figure 7: Score per episode

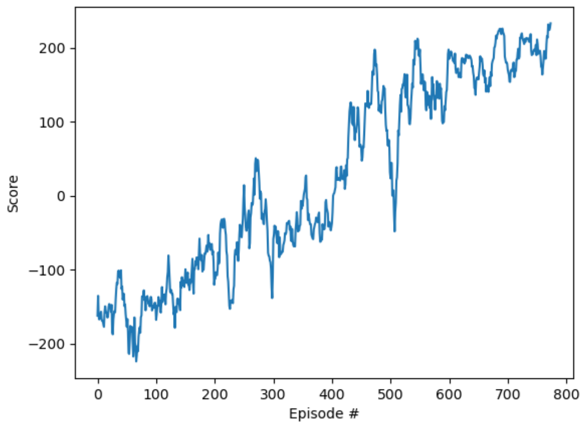


Figure 8: Score per episode (m.a = 10)

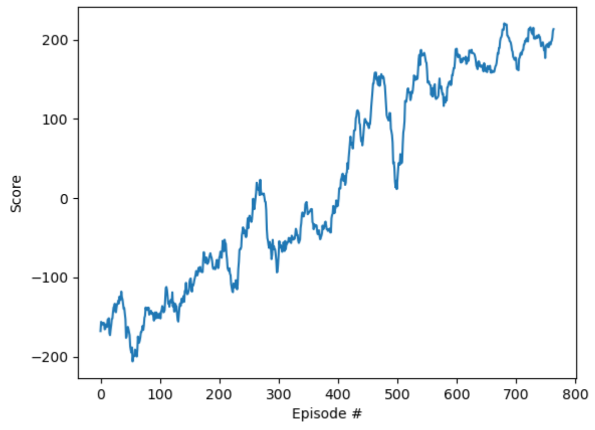


Figure 9: Score per episode (m.a = 20)

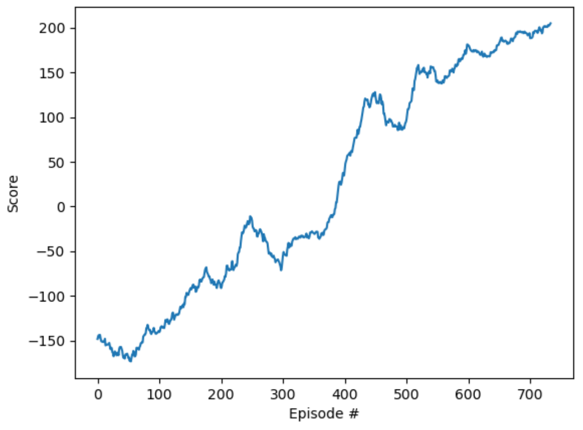


Figure 10: Score per episode (m.a = 50)

Overall, it can be seen that the results improve significantly throughout the episodes, despite some irregularities.

In the end, the agent manages to successfully land the Lunar Lander (thus receiving a final score of at least 200) after 683 episodes.

3.3 Overall consistency

As discussed in 3.2, it is important to test the network for various random seeds to verify how efficient and consistent the algorithm is.

For this reason, I decided to run the test with 10 different random seeds and record the number of episodes necessary for a successful trial (if successful). The results can be seen in Figure 11.

As the graph shows, all 10 trials succeeded, even though 3 of them were more "unlucky" and risked failing. The global consistency though is very high with a 100% of successful attempts. Overall the average number of necessary episodes per trial was 729.

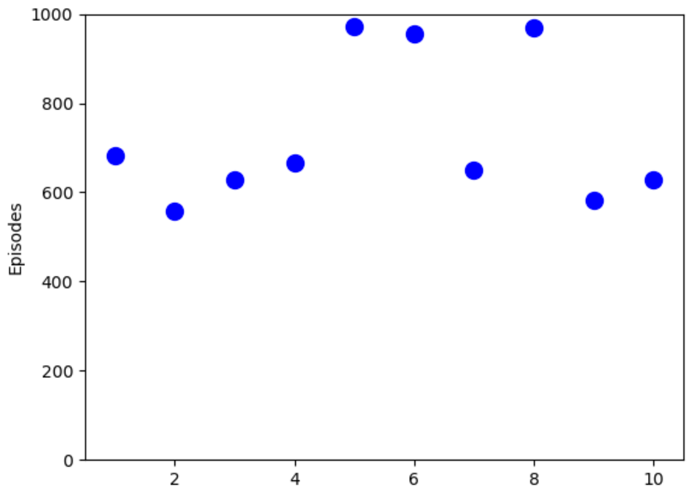


Figure 11: Number of episodes per trial with different random seeds

4 Conclusion

4.1 Future work

After describing the results of this project, it would be interesting to explore some possibilities for future work.

One possible promising line of work would be solving the same problem with different reinforcement learning algorithms and then comparing the results to determine the pros and cons of each of them. For example in this video [9] the author uses Proximal Policy Optimization (PPO), another reinforcement learning algorithm, to solve the Lunar Lander environment. In this other example [10], instead, the authors initially try to apply the SARSA algorithm by first discretizing the states in order to have a finite number of Q-functions and avoid using a neural network.

Apart from solving the same exact problem, it would also be interesting to try and solve a slightly more complex one: the Lunar Lander Continuous environment. The difference between this new environment and the one I used is that the action space, and thus the control, is not discrete anymore but continuous. This makes the whole learning and controlling slightly more complex and surely represents an interesting challenge in which to put to use the knowledge we acquired during this first project.

4.2 Final words

In conclusion, we showed how a non-trivial problem such as the landing of a Lunar Lander (in a simplified environment, obviously) can be solved thanks to reinforcement learning. In particular, we showed how standard reinforcement learning techniques are not enough to deal with this kind of situation and thus it is necessary to lean on deep reinforcement learning techniques, such as deep Q-learning.

With this technique, we obtained satisfactory results with good efficiency and extremely high repeatability.

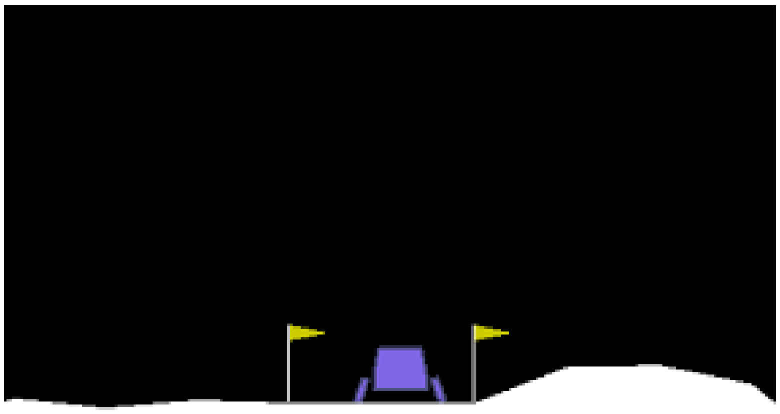


Figure 12: Lunar Lander at the landing site

In the end, we can say that thanks to DQN the spaceship finally landed!

References

[1] https://www.gymlibrary.ml/envs/LunarLander-v2/.

[2] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8.3 (1992), pp. 279–292.

[3] Jianqing Fan et al. “A Theoretical Analysis of Deep Q-Learning”. In: *Proceedings of the 2nd Conference on Learning for Dynamics and Control*. Ed. by Alexandre M. Bayen et al. Vol. 120. Proceedings of Machine Learning Research. PMLR, Oct. 2020, pp. 486–489. URL: https://proceedings.mlr.press/v120/yang20a.html.

[4] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[5] *Module*. https://pytorch.org/docs/stable/generated/torch.nn.Module.html.

[6] Jieliang Luo and Hui Li. “Dynamic Experience Replay”. In: *Proceedings of the Conference on Robot Learning*. Ed. by Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura. Vol. 100. Proceedings of Machine Learning Research. PMLR, 30 Oct–01 Nov 2020, pp. 1191–1200. URL: https://proceedings.mlr.press/v100/luo20a.html.

[7] Shangtong Zhang, Hengshuai Yao, and Shimon Whiteson. “Breaking the Deadly Triad with a Target Network”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 12621–12631. URL: https://proceedings.mlr.press/v139/zhang21y.html.

[8] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. https://arxiv.org/abs/1509.02971. 2015. DOI: 10.48550/ARXIV.1509.02971.

[9] *Introduction to proximal policy optimization tutorial with Openai Gym Environment*. https://www.youtube.com/watch?v=oOmcGQXJRXM&amp;t=2s, journal=YouTube, publisher=YouTube, year=2020.

[10] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. “Solving the lunar lander problem under uncertainty using reinforcement learning”. In: *2020 SoutheastCon*. Vol. 2. IEEE. 2020, pp. 1–8.