INF581 Final Project report – Lunar Lander RL agent

Ayham Olleik
ayham.olleik@polytechnique.edu
Laurene Pascal
laurene.pascal@polytechnique.edu

Khalig Aghakarimov
khalig.aghakarimov@polytechnique.edu
Markus Chardonnet
markus.chardonnet@polytechnique.edu

Abstract

Reinforcement Learning is an area of Machine Learning concerned with how intelligent agents should take actions in an environment in order to maximize the notion of cumulative reward. Nowadays, it has widely spread applications in different fields such as robotics, game industry, autonomous self-driving, etc. The purpose of this project is to apply some aspects behind RL on the so-called "Lunar Lander" problem.

Contents

1. Introduction: Environment

The topic we have chosen is about the landing of a ship on the moon whose environment set up can be found in AI gym open source library. To be more precise, the name of the environment is LunarLander-v2 that you can read more about clicking **here**. One can see that the environment has an observation space of 8 different parameters:

1. x coordinate of lunar lander position (numerical)
2. y coordinate of lunar lander position (numerical)
3. horizontal component of velocity (numerical)
4. vertical component of velocity (numerical)
5. angle of lander with respect to the ground (numerical)
6. angular velocity (numerical)
7. whether leg1 touches the ground (boolean)
8. whether leg2 touches the ground (boolean)

Besides the action space of the environment consists of 4 actions, which involves the firing of the *right engine*, *left engine*, *main engine* as well as *doing nothing*. We thought it could be interesting in terms of an implementation since it involves multiple parameters to be handled so that the landing takes place properly. Apart from the landing without any crush, we should also ensure that the ship lands exactly between the 2 flags as described in Figure 1.

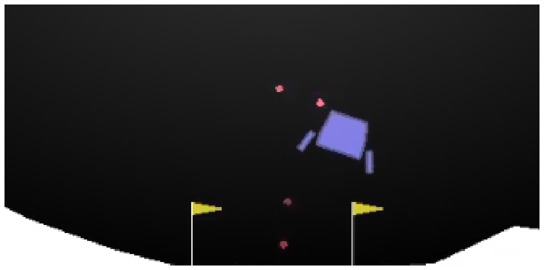
Figure 1. LunarLander-v2

2. Background

In order to understand the following report, one must have in hand some notions on Reinforcement Learning as it is the main topic. The objective is to learn how an agent, placed in some environment should behave in order to achieve a certain goal. At each step of the process, the agent is initially in some state of the considered space. It performs an action and observes two things: it gets a certain reward and moves to a new state. We assume that the performed action directly influences the next state and that the reward depends on: a) the current state, b) the next state, and c) action. Thus, the goal of the agent is to make the right actions to maximise future rewards.

3. Motivation

The intention of this section of the paper is to show the need to use an RL agent in our model to be able to behave properly to reach our goal. At first, we decided to write a non-learning algorithm that reads the observation and try to infer the proper action in a smart, but simplistic way. Algorithm 1 illustrates our non-RL agent. As you can see, in case of a height not enough to set the ship to land on the right place, we focus on lifting the ship as much as possible to be able to move it back to the right track. After this step is guaranteed, we then focus primarily on fixing the angle of the ship by doing random pick between the main engine and one of the left or right engines based on the sign of the lander angle. Finally, we focus on fixing the x coordinate of the ship. This idea looked nice and did work in some cases, but these general cases were not guaranteed to work in all scenarios. An example of that is the hard part when the ship is close to the ground, but not between the two flagged area. Thus, the challenge is to move it up in the direction of the landing spot while preserving the angle. This became our motivation to use smartness and train an RL on this environment.

4. The Agent

4.1. Agent based on a dummy non-learning algorithm

This is the non-RL agent we described in the Algorithm 1 to show the motivation of the need of RL in solving this problem.

4.2. Agent based on a Q-table

Q-Table method is one of the simplest RL methods. It records every action an agent took from the respective state and the reward it received. Through training and updating the Q table, the agent chooses the reward-maximizing actions. In this method the size of table is clearly linked to the size of the possible states multiplied by the possible actions that can be done. In our current problem we have 4 actions, but our state space is continuous and thus it is impossible to use this method with this setup. We then decided to transform this continuous state space to a discrete one by dividing the range space of the values into small chunks (example dividing the x, y coordinate from continuous between 0 and 1 to 0.1, 0.2, 0.3 ...1. That is possible since most of the values we encountered are within a certain range of -4 to 4 approximately. After running our algorithm for several times we did not see an improvement in our reward score. In addition, we faced some value problems with the velocity since it was sometimes in a very low scale that is hard to quantify and make use of. After this method, we decided to go and investigate the RL with policy gradient function, because it accepts a continuous state space.

4.3. Agent based on a parametric policy

Parametric Policy : In order to fit the environment more adequately, we decided to develop an agent based on a parametric policy. As the space of state is naturally modeled in a continuous way, such a policy is more suitable than a policy based on a Q-table, built for discrete state spaces. In this case, the policy will be a function based on a general model and a parameter [1].

The first question we can ask ourselves is the ability of the parametric policy to approach an optimal policy. Indeed, the best policy in our parametric model could be different, and worse than the optimal policy.

A common disadvantage of this method is when the state space has a large dimension, but it should not be the case for the 8-dimensional space of our environment. We note π_θ the policy depending on the parameter θ.

Configuration : As there are four possible actions and the state of space has eight dimensions, we will therefore have 8 input nodes (one for each dimension) and 4 output nodes. The next explained simple architecture did give quite optimistic results (**Section 5**) and thus we decided to stick with it. Our best model for parametric policy is:

- **Input layer:** 4 Linear nodes
- **Single hidden layer:** 10 ReLU nodes
- **Output layer:** 4 linear nodes with SOFTMAX

We also use a batch size of $batch = 10$, $\alpha = 0.0001$ and a discount factor $\gamma = 0.95$ for previous rewards.

Learning : Learning is performed using Reinforce Policy Gradient Theorem Algorithm. The idea is to apply a Gradient Ascent algorithm on the expected total rewards following the parametric policy in order to maximise this expectation. We call this expectation J:

$$\nabla_\theta J(\theta) = \underset{\pi}{E}[G_t \nabla_\theta \ln \pi_\theta(a|s)] \tag{1}$$

Performing gradient ascent is possible as soon as J is differentiable with respect to the policy parameter. Thanks to the Gradient Policy algorithm, we get a nice formulation of J's gradient and we can observe that J is differentiable as soon as the policy is (this is the case for our choice).

The expectation is approximated by a Monte Carlo method on sampled trajectories. The trajectories are sampled following the the trained policy. Thus, it is an on-policy algorithm which means that the sampled trajectories are generated using the policy that we are optimizing.

4.4. Agents based on Deep Q-learning

As the previous methods did not give very promising results, we decided to try agents based on Deep Q-Learning. The methods based on Deep Q-learning are more efficient though they take a lot more time to be trained depending on the architecture. Therefore, it is relevant to use it here, where the gradient policy and Q-learning don't seem to be enough for our problem. [2]

Description: The method of Deep Q-Learning is based on Q-Learning and neural networks. The use of a neural network is to better optimize the objective function. This method is very similar to a simple Q-Learning : in Q-Learning, we update a Q-table through the equation of Bellman describes below :

$$Q(S_t, A_t) = (1-\alpha)Q(S_t, A_t) + \alpha(R_t + \gamma \max_a Q(S_{t+1}, a)) \tag{2}$$

For a neural network, we use the same function, but in order to update the weights of our network, and not the Q-table itself.

Configuration: In order to achieve satisfactory results for our task, we decided to be inspired by the neural network architectures available online that solve similar problems. We focused on two architectures that we found online. Our first architecture consists of:

- an input layer of 8 nodes due to the dimension of the observation space as described in the Introduction
- the first hidden layer of 512 nodes
- the second hidden layer of 256 nodes
- an output later of 4 nodes because of the possible number of actions.

You can have a look at the architecture visually in *Figure 2*. For the training, we are considering a batch size $n = 64$, a learning rate $\eta = 0.001$ and a discount factor $\gamma = 0.99$. The ReLU is used for the 2 hidden layers whereas for the output layer, a simple linear activation function is applied. Besides, the metric that we are using is the MSE (Mean Squared Error) together with the Adam Optimizer that handles the update procedure of weights.

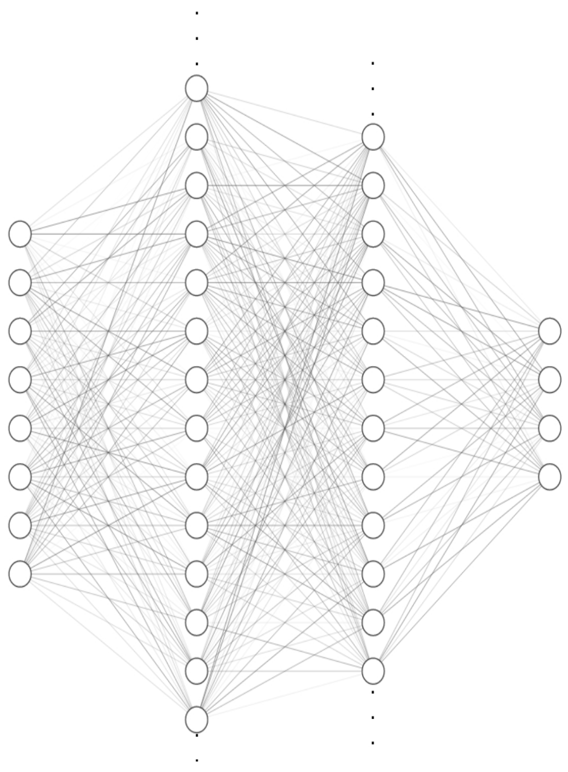

Figure 2. Architecture of the DQN model

Our second architecture was also found in the literature. It consists of :

- an input layer of 8 nodes due to the dimension of the observation space as described in the Introduction
- the first hidden layer of 32 nodes

- the second hidden layer of 32 nodes
- an output later of 4 nodes because of the possible number of actions.

This network is smaller than the previous one, and therefore can be trained with more episodes. We used a learning rate $\eta = 0.001$, ReLu for the two hidden layers and a linear function for the output layer. We used the metric MAE (Mean Absolute Error) and the Adam Optimizer. Moreover, a horizon of 1000 episodes was given for the ship.

4.5. Agent based on Self Adaptive Evolution Strategies (SAES)

The (1+1)SAES is following the exact code explained in the class lecture. The algorithm starts with the objective function f, initial solution x and the self-adaptation learning rate τ and loops over each generation and repeats the following:

- mutation of σ (current individual strategy) : $\sigma' \leftarrow \sigma\, e^{\tau \mathcal{N}(0,1)}$
- mutation of $\boldsymbol{x}$ (current solution) : $\boldsymbol{x}' \leftarrow \boldsymbol{x} + \sigma' \mathcal{N}(0,1)$
- eval $f(\boldsymbol{x}')$
- survivor selection $\boldsymbol{x} \leftarrow \boldsymbol{x}'$ and $\sigma \leftarrow \sigma'$ if $f(\boldsymbol{x}') \leq f(\boldsymbol{x})$

5. Results

For policy gradient method, our algorithm is based on a simple neural network architecture with a single hidden layer as described in **Section 4.3**. The algorithm also trains quickly compared to other methods tried with deeper neural networks. Our model was trained for around 1000 episodes which took around 10 min on a normal CPU machine. The max average result reached was around 130 average reward over a 32 experiment window.
Secondly, we trained our first Deep Q-learning model (which seemed to give better results for our problem) on a CPU for 400 episodes. The architecture mentioned in the **Section 4.4** took around 10-12 hours to finish the whole training procedure. The second Deep Q-learning model was trained on a CPU for 70.000 episodes, and took about ten minutes to train.
Finally, we decided to compare our models to the Optimization method, so-called (1+1)Self-Adaptive Evolution Strategies (SAES) that was developed during the lab session of the course. You can find some numerical results about the time and score performance of the training and testing of all models in *Table 1* and *Table 2*.
Looking at the tables, it becomes obvious that SAES is the best compared to the other techniques, both in terms of the score and time complexity. The first DQN model gives satisfactory results as well, although it takes much more time to train on a CPU machine as it involves a huge neural network. The second DQN model also gave good results for a short training time, but showed some instability when being trained: indeed, the training did not always provide the results we obtained, the reward could sometimes remain negative. Thirdly, Policy Gradient is surprisingly fast to improve the average reward (similar to SAES), but unfortunately reached a local maximum around 100-120 even after 4000 episodes as shown in *Figure 3*.

Model	**Training score**	**Validation score**	**Training time**
Policy Gradient	43	130	10 mins
DeepQL 1	205.881	205	734 mins
DeepQL 2	210	210	10 mins
SAES	260	234.07	6 mins

Table 1. Training summary

Model	**Episodes**	**Average Reward**	**Time (in seconds)**
Dummy Agent	10	-90	0.8
Policy Gradient	10	32	2.9
DeepQL 1	10	205.144	67.61
DeepQL 2	10	210	13
SAES	10	229	3.5
Dummy Agent	100	-92	6.4
Policy Gradient	100	43	16
DeepQL 1	100	205.248	653
DeepQL 2	100	209	96
SAES	100	234.07	31.7

Table 2. Testing summary

Below you can see the graphical results that illustrate the training process for each model, more precisely the average scores obtained for each episode.

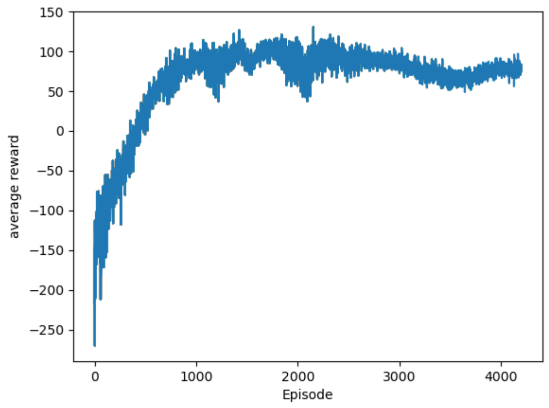

Figure 3. Policy Gradient

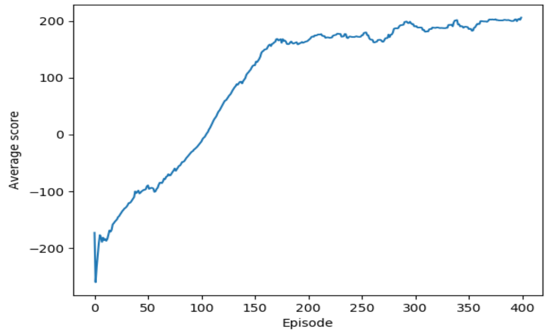

Figure 4. DeepQL 1

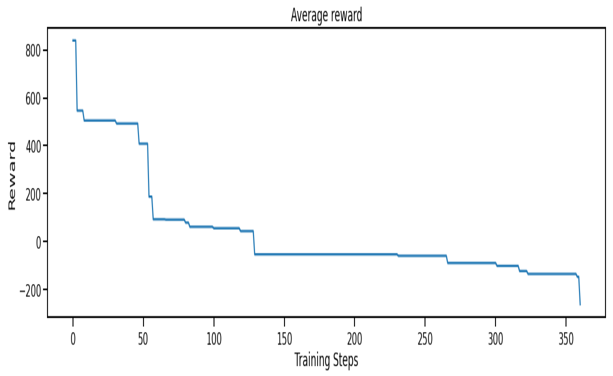

Figure 5. SAES

6. Noticed Problems

- Policy Gradient stopped at a local maximum of 112 average reward although we tried different hyper-parameters.
- Adding layers or nodes for Policy Gradient did not increase the result.
- The first DQN architecture is huge and takes time to render the environment and predict the action even for a loaded model (on CPU).
- The second DQN architecture can be unstable when training.
- Lack of GPU did put us behind since our first working DQN model took almost half a day to finalize the training process.

7. Code

Our code is available as a public repository on this GitHub link.

Algorithm 1: Non-RL Agent

```
function Agent_{non_RL}(observation):
    $angle_{lowthreshold} \leftarrow -2$
    $angle_{highthreshold} \leftarrow 2$
    $X_{highthreshold} \leftarrow 0.1$
    $Y_{highthreshold} \leftarrow 0.6$
    $X_{lowthreshold} \leftarrow -0.15$
    $X_{mediumthreshold} \leftarrow -0.1$
    $factor \leftarrow 57$
    $angle \leftarrow angle_{lander} * factor$
    $rotate_{right} \leftarrow$ [main_engine, left_engine]
    $rotate_{left} \leftarrow$ [right_engine, main_engine]
    $y \leftarrow$ y position of lander
    $x \leftarrow$ x position of lander
    if $y < Y_{highthreshold}$ and
     ($x < X_{lowthreshold}$ or
     $x > X_{highthreshold}$) then
        return main_engine_fire
    if $angle > angle_{highthreshold}$ then
        return $random(rotate_{right})$
    if $angle < angle_{lowthreshold}$ then
        return $random(rotate_{left})$;
    if $x < X_{mediumthreshold}$ then
        return left engine fire;
    if $x > X_{highthreshold}$ then
        return right engine fire;
    else
        return do nothing;
```

References

[1] R. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Adv. Neural Inf. Process. Syst*, vol. 12, 02 2000.

[2] "Deep reinforcement learning: An overview."