Solving Lunar Lander with Double Deep Q-Network and modified reward function

Mărgărit Darius
Github page

Abstract - In this work, the main goal is to improve the reward system of the Lunar Lander game (from Box2D environments) by modifying the functions that grant the reward after each step, in order to obtain a more realistic environment in which the space shuttle constantly reduces its speed and touches the ground slowly and precise, eventually the success rate of the landings being high. As a secondary goal, the setting of hyper-parameters and construction of the neural network, together with the modification of the stop conditions of an episode, results in the efficiency of the algorithm and obtaining a low training time compared to the original environment.

I PROBLEM STATEMENT

The original environment is a classic rocket trajectory optimization problem called LunarLander-v2 and is part of the Python Gymnasium package[1]. An episode starts with the ship descending from the top of the screen with a random initial force applied to its center of mass. At each step, the agent is provided with the current state of the lander which is an 8-dimensional vector of real values, indicating the horizontal and vertical positions, orientation, linear and angular velocities, and the state of each landing leg (left and right). The agent must choose one of the 4 possible actions (0 - do nothing, 1 - fire left orientation engine, 2 - fire main engine, 3 - fire right orientation engine).

The reward system implemented by the authors can be found in the game description. After every step a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode. For each step, the reward:

- 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 is firing.
- is decreased by 0.3 points each frame the main engine is firing.

Only a small increase/decrease is given for how closer/further the lander is to the landing pad, slower/faster the lander is moving and the more the lander is tilted (angle not horizontal). This reward is calculated for 2 consecutive states, the idea being that between 2 states the changes in position and velocity should be as small as possible, resulting in a negligible impact on how slow and accurate the landing is. The episode finishes if the lander crashes (-100 points), the lander gets outside of the viewport (x coordinate is greater than 1, also -100 points) or the lander is not awake (a body which is not awake is a body which does not move and does not collide with any other object), so it landed safely ($+100$ points). The episode can also finish when it hits the maximum episode length (e.g. 1000 steps).

II BACKGROUND

Reinforcement learning[2] is a machine learning training method based on rewarding desired behaviors and punishing undesired ones. In general, a reinforcement learning agent (the entity being trained) is able to perceive and interpret its environment, take actions and learn through trial and error. The agent performs an action in the given environment which leads to the interpreter that will provide the agent with a reward (positive or negative) and the next state.

Q-Learning is an off-policy value-based method that uses a Temporal Difference (TD) approach to train its action-value function[3][4]:

$$Q_{new}(S_t, A_t) \leftarrow Q_{old}(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q_{old}(S_t, A_t)] \quad (1)$$

where $Q_{new}(S_t, A_t)$ is the new Q-value for action A_t at state S_t, α is the learning rate (or step size) that determines the extent to which newly acquired information overcomes old information and γ is the discount factor that determines the importance of future rewards.

Therefore, the Q-function is encoded by a Q-table, a table where each cell corresponds to a state-action pair value.

To balance the trade-off between exploration and exploitation, we use an Epsilon-Greedy strategy which consists of generating a random number, if the number is less than epsilon, the agent takes a random action, otherwise it chooses the action from the table with the higher Q-value. Epsilon decreases from 1 with each episode.

In the case of our problem, while the action space is discrete, the state space is continuous in 8 dimensions. Therefore, creating and updating a Q-table for the environment would not be efficient. It is characteristic of most games to have a complex and continuous observable space with a discrete set of actions, thus a variant is introduced to predict Q-values instead of calculating them and is called Deep Q-Network (DQN)[5].

DQN uses a neural network (as a function approximation) in which the current state is entered as input and the Q-values for possible actions in the environment are returned as output.

An efficiency problem occurs when comparing 2 consecutive states. There is not much difference between them, so the agent cannot learn anything new from both. The agent is also likely to forget what it is like to be in a state it has not seen in a while, which affects learning from catastrophic events that happened before. Therefore, replaying the experience is involved[6]. Several transitions with the form $(s_t, a_t, s_{t+1}, r_t, d)$ will be stored in a replay memory buffer (old ones will be discarded as new ones come in) and during training a small batch of random states will be selected to train the neural network.

However, a single neural network for outputting the Estimated Q-value is not sufficient, as some reference values are needed in order to compute the loss function. A Target Neural Network is introduced (resulting in Double DQN) which will be used to predict Target Q-values for s_{t+1} (the next state, if the episode is not done, $d \neq 1$) from each transition in the chosen batch. Estimated Q-values are computed using the main neural network with s_t and a_t, following that the error of the loss function would be calculated using the two values obtained.

A. Algorithm

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions from D

$$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1, \\ r_j + \gamma \text{max}_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

    Perform a gradient descent step on $\epsilon = y_j - Q(\phi_j, a_j; \theta)$ with respect to the network parameters θ
    Every C step reset $\hat{Q} = \tau Q + (1 - \tau)\hat{Q}$
  end for
end for

The algorithm is a combination of the original DDQN presented in [5] and the idea of t-Soft Update of Target Network from [7].

B. Network Architecture

Both neural networks (Estimated Q-Network and Target Q-Network) are identical in terms of architecture (all layers are fully-connected), having an input layer that receives 8 real values representing a state of the lander. After this, there are 4 hidden layers with $512, 256, 128$ and 32 neurons each. Finally, the output layer has 4 neurons, each representing the Q-value for the possible actions. As the activation function for the layers is used Rectifier Linear Units (ReLU)[8]. The Huber loss function was used to avoid exploding gradients. [9]For Mean Squared Error (MSE) loss function, if the training sample does not align with current estimates, the gradient terms might explode and alter the network weights substantially leading to undesirable performance. Huber loss function $H(\epsilon) = \sqrt{1 + \epsilon^2} - 1$, where ϵ is the error between prediction and target value, has the advantage of being differentiable and has derivatives bounded between -1 and 1. The optimization used to train the network is Adam[10].

C. Hyper-parameters

As in [9], in order to choose the best combination of parameters, numerous tests are necessary to observe the behavior of the agent in different situations. The problems that appear are not always related to the way of choosing the parameters, but also to the architecture of the neural network. Observing the agent in different tests, the following parameters seemed to us the most suitable for the given problem:

1. *ε-decay* $= 0.99$: It is important for the agent to experience as many situations (state-action pair) as possible and to observe the reward received. The most significant score is given to a proper landing, so at the beginning of the training (first 50 episodes) it will not succeed, but will learn from the mistakes made until then (crashes). If the rate is too low, we risk having an agent who did not manage to land at all, so from that moment on, it will never manage to do that. Having a high rate is also a problem, because the agent ends up using its neural network too late, and from there the training becomes cumbersome and can take several hours. The tests show that the best values are between 0.99 and 0.995.

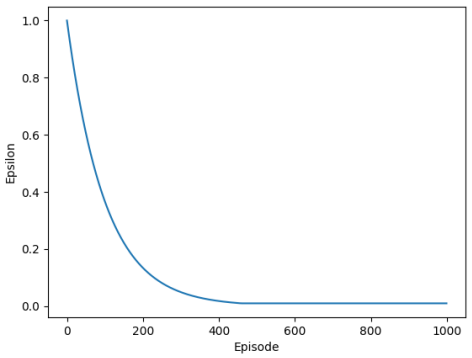

Figure 1: Epsilon reduction during training

2. Learning rate $\alpha = 0.0005$: The ideal values can vary between 0.0005 and 0.001 (the value also suggested in the original research paper for the Adam optimizer[10]) so that the Reward Moving Average over the last 100 episodes does not change the upward trend suddenly during training. A higher value leads to large oscillations of the learning trend, while too low values lead to the stagnation of learning from the beginning of the training.

3. $\tau = 0.001$: Used in the t-soft update for the parameters of the Target Neural Network, τ parameter helps the two networks not become identical quickly, but in a manner suitable for learning, so that a value that is too high would lead to exactly this phenomenon, and a too low would lead to the immediate stagnation of the agent in a situation where he can no longer learn anything.

4. Discount factor $\gamma = 0.999$: It shows the importance of future rewards with a value close to 1, because the last reward is the biggest in this case, being quite important, but the short-term ones should not be omitted either, otherwise the agent might never reach to the reward of the successful completion of the episode.

5. Others: The buffer size is set to $100,000$ and the batch size to 64, these values are set for a good memory / time ratio for the training to perform well. The update frequency of the Target Neural Network is one update every 4 episodes. Trying different values like 3 or 5 caused a longer agent training time.

III MODIFICATION OF REWARDS

Observing how a well-trained agent manages the actions of the lander in order to obtain the greatest possible reward, we noticed that it no longer performs a realistic landing, but one that is as fast and efficient as possible. Thus, if the ship is generated by the game at the beginning in a lousy state, the agent often fails to land. This way of learning is mainly caused by the bad distribution of rewards in each step depending on the state of observation of the ship.

However, in the original implementation of the game there is a function that penalizes major changes that occur between two consecutive states, but this function does not penalize the agent sufficiently. We have come to the conclusion that an agent manages to learn faster if it is penalized enough for the mistakes it makes, than if it is rewarded for everything it does well. Otherwise, by penalizing slightly and offering too high positive rewards, the agent tends to find a way to exploit (farm) a certain reward function.

The scoring system and the ending of an episode can be improved so that the landing is more realistic and the training time of the agent shorter. At the same time, these improvements lead to lower fuel consumption and better agent performance. These conditions are checked at each step in an episode:

- -2 points if the angle of the ship exceeds 17 degrees starting from the Oy axis in both directions
- -1 point if the absolute horizontal speed exceeds 0.3 units of measure and -0.75 points if the vertical speed exceeds 0.5 units of measure or the ship starts to ascend
- -0.1 points if the ship exceeds 0.2 measurement units on the Ox axis (both sides), $+0.05$ otherwise
- -0.25 points if the ship is up to 0.5 units above the ground and the speed is greater than 0.15 units (if the ship ascend, the reward also decreases), otherwise $+0.1$ and in addition $+0.05$ if the ship is up to 0.1 units of measurement from the ground
- -0.05 if the two conditions above do not simultaneously offer the positive reward, $+0.15$ otherwise (an additional condition for the ship to reach a favorable position with a reduced speed)
- -300 points if the ship crashes
- $+200$ points if both legs touch the ground between the flags and the landing speed is below 0.15 units of measurement
- $+50$ points if the ship is not close to any flag, but is between them, -50 otherwise

These additional rewards induce the agent a behavior favorable to a realistic landing (in which the ship slows down as it approaches the landing surface) which implicitly leads to a higher rate of successful landings, since the agent will not rush to land to obtain the last reward. Moreover, firing the main engine is now sanctioned with -0.5 points, because summing up all the positive rewards that the agent can receive, they do not exceed 0.5, thus we exclude the possibility that the agent will take advantage of a certain state of the ship in which to farm rewards.

Regarding the end of an episode, we noticed that in the default environment, the condition for ending the episode where the lander is not awake (as mentioned in Chapter I), makes the episodes last longer, forcing the agent to take unnecessary additional steps, which also recursively affects the entire learning process, filling the replay buffer with redundant states. An unsuccessful landing (outside the flags) did not ended an episode, but the agent tries from that position to reach between the flags, although it has not learned to take off (to activate the main engine so that it rises from the ground), thus using the side engines in vain, until the maximum number of steps is reached. The new condition implemented leads an episode to be immediately ended every time the ship touches the ground for the first time with both legs, regardless of the landing zone. Running two agents trained with the same algorithm on the default environment and the modified one for 2000 episodes, on average, the steps required to end an episode decreased by 49.27%:

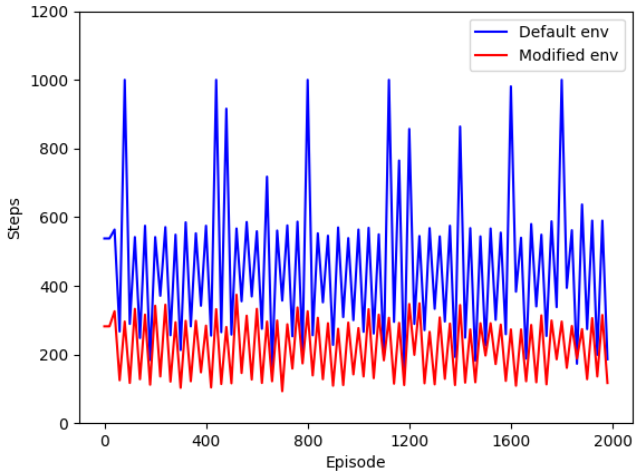

Figure 2: Steps per episode

Also, if we consider for the firing of the main engine, 1 unit of measure on the scale of fuel consumption; for the firing of the secondary engines (left and right) we consider 0.1 units of measure and do the same experiment with 2000 episodes, on average, the fuel consumption is reduced in the modified environment by 16.81%:

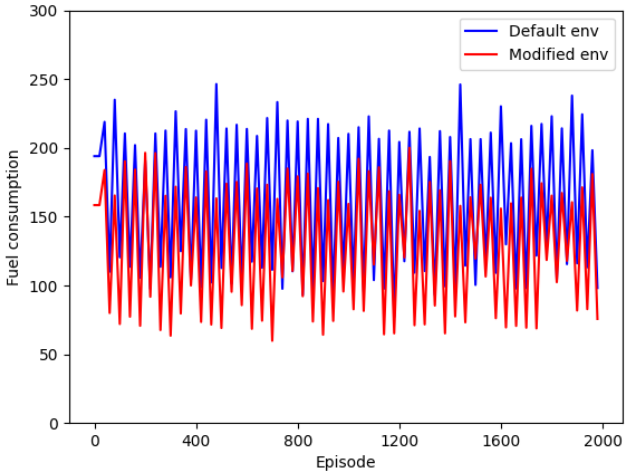

Figure 3: Fuel consumption per episode

IV RESULTS

Testing different algorithms on the two environments (the default one and the one with the modified reward), the differences are major in terms of success rate (surface landings) and time spent for training. After 50,000 episodes played with the two agents, trained with the DDQN presented previously, in the default environment the accuracy was 99.3%, the agent being trained for 30 minutes (1000 episodes) on the GPU in Google Colaboratory. In comparison, the agent trained in the modified environment had an accuracy of 99.6%, the training requiring 7 minutes (600 episodes) on the GPU in Google Colaboratory. Besides, taking into account the shortcomings of the default environment, the accuracy of the first agent is not given by the precise landings (between the flags), many being outside them (something that is not desirable in real life). The second agent does not encounter this problem, because it receives its landing reward only if it landed between the flags. Also, this can be seen by analyzing the amount of spikes in Figure 4.

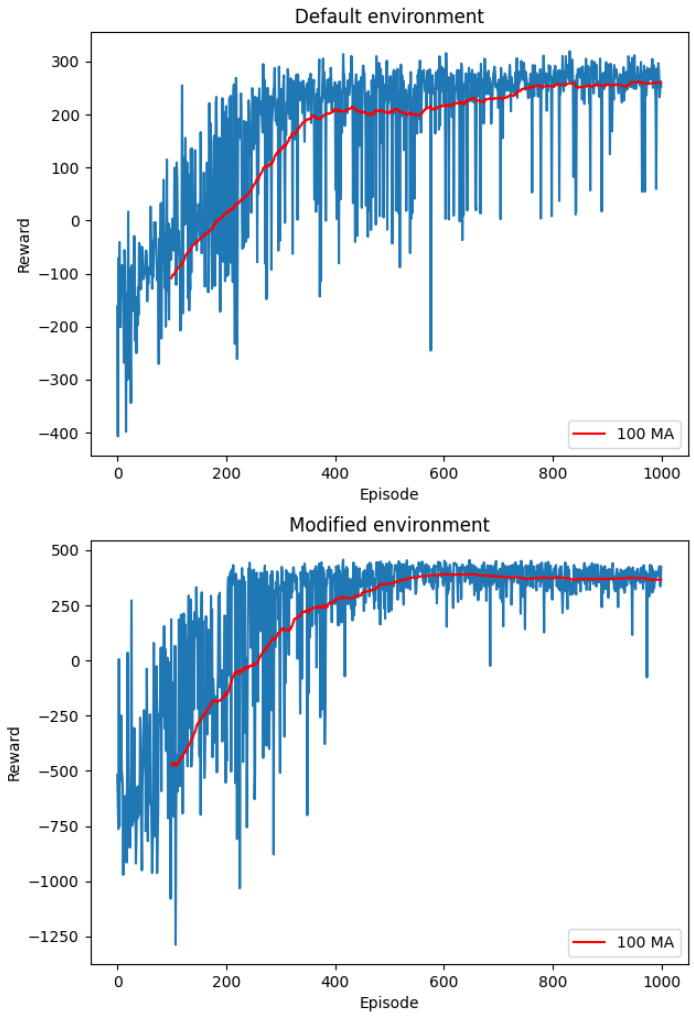

Figure 4: Rewards in both environments during training

V DISCUSSION

If we analyze the game's environment in more detail, it can be observed that it has several variables that can be modified, thus the game becomes even closer to reality. There are some options added by the creators that allow to modify the gravity (depending on the gravitational acceleration of the environment in which the ship is located), the possibility of wind in the environment acting with a certain force on the lander (or with different forces during landing), turbulence, etc.

In addition, at the beginning of the game a force chosen randomly from a normal distribution within a certain interval is applied to the center of the ship (which causes the ship to have a position that needs to be fixed in order to land successfully), which is why the range of the normal distribution can vary so that the game becomes more difficult. There are some aspects that could lead the agent to an accuracy close to 100%, these will be presented in the subsection Future work.

A. Problems encountered

The main problem encountered was the time. At the beginning of this research, the algorithms used for training were not efficient at all, thus a single training epoch lasted several hours and the result was not always the desired one. The choice of hyper-parameters and finding a good architecture for the neural network were also long-term processes, requiring a suitable combination in order to obtain in a short time a performing agent.

About the neural network architecture, after several tests, the best performance was obtained using a top-down triangular architecture, with the base immediately after the input layer (512-256-128-32 neurons). Values may vary, performance being similar, therefore reducing/increasing training time.

Setting up the reward system was also a challenge, because initially we had to intuit what values the rewards would have for certain behaviors of the lander. A too sharp decrease of the reward can sometimes lead to an impasse in training, so to saturation in a stage where the agent does not perform well at all. But an unbalanced positive reward that is not given proportionally to the importance of the state of the space shuttle can lead to the modification of the game concept, so to a situation where we do not get the desired behavior.

B. Personal experience

As a personal experience, during this project I discovered important libraries in Python such as NumPy, Pandas, Matplotlib, etc. and more importantly the PyTorch deep learning library with which I learned to work. Besides these, I initially analyzed libraries such as TensorFlow and Stable-Baseline 3, but I did not choose them in this project for efficiency reasons.

I also learned what it is and how to work with a neural network, how such a network works using backpropagation to fine-tune the weights, what a loss function is and what impact it has on the training method, etc. I discovered many algorithms used in Reinforcement Learning with which I experimented on different games to learn exactly how they work.

The research work carried out and the solving of many problems encountered during the development of this project, have improved my skills of searching for information on the Internet, but also of overcoming obstacles, in order to finish successfully a desired project.

C. Future work

The project can be expanded in the future with more changes in the area of rewards, but also through the simulation of a more realistic landing. Thus, one of the ideas would be to change the distance from the ground at which the position and velocity of the space vehicle is initialized. A greater distance means more time available so that the agent can influence the ship's trajectory better in exceptional situations, where the initial position is less favorable for a correct landing. We consider that such an approach would substantially increase the percentage of successful landings (even if this change is made only on a practical level and not visually in the game). Ultimately, this modification can be considered as a realistic one, because in reality the trajectory of the ship starts to change at a significant distance from the ground.

In order to use the variables mentioned at the beginning of chapter V Discussion, namely: turbulence, wind intensity, gravity, the agent needs more training episodes to be able to go through more situations involving these new variables, and an approach using Prioritized Experience Replay[11] is more appropriate in this situation, also using a larger replay memory.

At the same time, a Proximal Policy Optimization(PPO) [12] algorithm could have a better performance on this type of problem, but also an approach based on the Dueling DQN[13] algorithm can be explored and compared with the current results.

VI CONCLUSION

In this project, we successfully managed to bring a significant improvement in the rewards system, while providing justifications for the changes made. The project still has potential on the research and discovery side, there are improvements that we would have implemented if we had more time and resources available, but they were presented in this work as ideas, requiring research. On a personal level, this project opened my path to artificial intelligence and machine learning.

REFERENCES

[1] LunarLander-v2, `https://gymnasium.farama.org/environments/box2d/lunar_lander/`, (accessed: 26.08.2023).

[2] L. P. Kaelbling, M. L. Littman, A. W. Moore, *Reinforcement Learning: A Survey* **1996**, DOI `10.48550/arXiv.cs/9605103`.

[3] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, London, **2020**.

[4] Q-Learning, `https://huggingface.co/learn/deep-rl-course/unit2/q-learning`, (accessed: 26.08.2023).

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, *Human-level control through deep reinforcement learning* **2015**, DOI 10.1038/nature14236.

[6] L.-J. Lin, *Self-improving reactive agents based on reinforcement learning planning and teaching* **1992**, DOI 10.1007/BF00992699.

[7] T. Kobayashi, W. E. L. Ilboudo, *t-soft update of target network for deep reinforcement learning* **2021**, DOI 10.48550/arXiv.2008.10861.

[8] A. F. Agarap, *Deep Learning using Rectified Linear Units (ReLU)* **2018**, DOI 10.48550/arXiv.1803.08375.

[9] Solving Lunar Lander with Double Dueling Deep Q-Network and PyTorch, https://drawar.github.io/blog/2019/05/12/lunar-lander-dqn.html, (accessed: 28.08.2023).

[10] D. P. Kingma, J. Ba, *Adam: A Method for Stochastic Optimization* **2017**, DOI 10.48550/arXiv.1412.6980.

[11] T. Schaul, J. Quan, I. Antonoglou, D. Silver, *Prioritized Experience Replay* **2015**, DOI 10.48550/arXiv.1511.05952.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, *Proximal Policy Optimization Algorithms* **2017**, DOI 10.48550/arXiv.1707.06347.

[13] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, N. de Freitas, *Dueling Network Architectures for Deep Reinforcement Learning* **2015**, DOI 10.48550/arXiv.1511.06581.