# Cyber Physical Systems
# Project: Landing a Lunar Module with DQN

Michel El Saliby[1]
Alberto Trevisan[2]

[1]Master in Computer and Electronics Engineering, University of Trieste
[2]Master in Systems and Electrical Engineering, University of Trieste

Fall 2023 Course

## 1 Introduction

Reinforcment learning (RL) is a model-free approach used to synthesize a controller in which an agent learns to make decisions by performing actions and receiving rewards. Deep Q-network (DQN) combines Q-learning with deep neural networks (NNs) to handle large state spaces effectively.

This project presents the implementation of a DQN algorithm and its robustness verification using the Moonlight framework [3]. The primary objective of the controller is to accurately land a 2D lunar module in a specified location. The environment utilized for this purpose is LunarLander-v2, provided by the Python library Gym [1].

## 2 Problem Statement

The 2D LunarLander-v2 environment involves controlling a spacecraft to land on a designated platform, Figure 1. The 8-dimensional state space consists of the $x$ and $y$ positions,

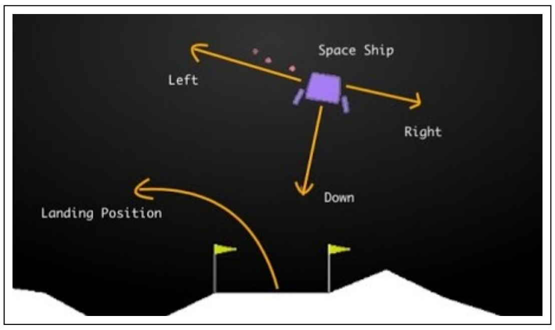

Figure 1: Visual representation of the LunarLander-v2 environment.

$x$ and $y$ velocities, angle with respect to the $y$ axis, and angular velocity of the lander, along with two indicators of contact between the two legs and the ground. The action space includes four discrete actions: do nothing, fire left orientation engine, fire right orientation engine, and fire the main engine, which provide propulsion directed upwards.

The primary goal is to design an agent capable to decide a policy to control the lander starting from different initial conditions, ensuring a precise, safe, and efficient landing. We ask the controller to manage the landing procedure starting from a certain height, so what happens before the moment that height is reached for the first time is not considered in the remainder of this project.

The specifications that must be met to ensure that the lander maintains safe and controlled behavior throughout its descent and landing process are described here in a user-friendly way: **(i)** The horizontal position of the lander must always remain within a certain range; **(ii)** If the lander is at or below a specific altitude, its angle must remain within acceptable limits to avoid excessive tilting; **(iii)** If the lander is at or below a specific altitude, its vertical speed must be below a threshold to ensure a soft landing; **(iv)** When the lander is near the ground, its horizontal position must be within a tighter range to ensure it lands on the designated platform; **(v)** The lander has to land eventually. From a technical point of view, it must reach a position where its altitude is below a certain threshold, indicating a successful landing. Actually, the latter specification could have been formulated using the contact sensors present in the state vector. However, we have found that the LunarLander-v2 library sometimes fails to update the values of these two states.

Finally, we want the spacecraft to land correctly starting from a certain range of values for each state, which are defined for the $x$ and $y$ positions, the $x$ and $y$ velocities and the angle. The other three states are always initialized to 0.

## 3 Assessment

To avoid ambiguity and to facilitate the verification process, we formulate the specifications using signal temporal logic (STL) [5]. The user-friendly specifications proposed in Section 2 are then formalized:

**(i) General Horizontal Position:**

$$\varphi_1 = \mathcal{G}\left(|x| < c_{\Phi_1}\right)$$

**(ii) Rotation:**

$$\varphi_2 = \mathcal{G}\left[(y \leq y_{\Phi_2}) \rightarrow (|\theta| < c_{\Phi_2})\right]$$

**(iii) Vertical Speed:**

$$\varphi_3 = \mathcal{G}\left[(y \leq y_{\Phi_3}) \rightarrow (|v_y| < c_{\Phi_3})\right]$$

**(iv) Landing Zone Horizontal Position:**

$$\varphi_4 = \mathcal{G}\left[(y \leq y_{\Phi_4}) \rightarrow (|x| < c_{\Phi_4})\right]$$

**(v) Landing Completion:**

$$\varphi_5 = \mathcal{F}\left(\mathcal{G}\left(y \leq c_{\Phi_5}\right)\right)$$

Alternatively, this specification can be written as:

$$\varphi_5 = (\text{True})\,\mathcal{U}(\mathcal{G}(y \leq c_{\Phi_5}))$$

Although the two formulas should be equivalent, we have found that the first one doesn't work in the Moonlight framework (raises `Java Exception`), so we chose to use the second one.

Furthermore, to ensure that the learned controller meets the requirements, we perform a falsification procedure. Falsification is often treated as an optimization problem where the objective is to minimize the robustness of the specifications, searching for conditions under which the system fails to meet the specified properties. We have implemented a falsification procedure based on a simple evolutionary algorithm:

1. Generate a set of $n_{\text{pop}}$ initial states where each state is uniformly sampled within the given bounds.

2. Run a simulation for each initial state using the learned DQN policy and record the resulting trajectories.

3. Compute the robustness values of each trajectory against the specified properties. The robustness values of each STL formula are then normalized based on the $c_{\Phi_i}$ value of that formula. Then, the fitness of a trajectory is defined as the minimum among the robustness values given by the different STL formulas (equivalent to the logical AND between all the formulas).

4. Save the initial conditions of the individuals with a negative fitness, as they falsify the specifications. Even if this is the case, continue with the falsification procedure, as it may provide more insight into the specific condition.

5. Generate the next generation by iteratively selecting the best $n_{\text{parents}}$ individuals from the current population. From each parent, generate $n_{\text{offspring}}$ individuals by adding to each of the 5 tunable states a Gaussian noise with zero mean and variance proportional to the size of the required range of initial values of that state. If the mutation results in a new state that does not lie within the specified bounds, the mutation procedure is repeated a maximum of $\max_{\text{retries}}$ times. If $\max_{\text{retries}}$ is exceeded, a completely new state is generated as in the first point. Thus, we have a new generation of $n_{\text{parents}} \times n_{\text{offspring}} = n_{\text{pop}}$ individuals.

**6**. The falsification procedure stops when $\max_{\text{evaluations}}$ individuals have been evaluated.

We note that we could have defined the fitness function as the robustness of a single STL formula, thus having a number of minimization problems equal to the number of formulas. This approach would have provided more insight into the robustness of each specification; however, in the context of this project, we preferred to use the minimum robustness approach because it required less computational time.

## 4 Controller

There are several reasons why we opted to use DQN as the controller. Firstly, it effectively handles continuous states, making it suitable for environments with complex and varied inputs. Moreover, it learns directly from experiences stored in a replay buffer, enhancing its adaptability and robustness. Lastly, its capability to manage uncertainties and disturbances in environments ensures reliable performance across dynamic condition.

The key points of the DQN algorithm that we implemented were inspired by [6]. Here is a brief overview of the key ideas:

- **Approximation of Q-function using a NN**: a NN takes the current states in input and generates the q-values for the number of actions.
- **Experience replay**: to stabilize training and improve convergence, DQN uses a replay buffer to store past experiences. During network updates, experiences are randomly sampled from the replay buffer to form training batches.
- **Target networks**: to further stabilize the training, two NNs are used: one main network (online) to predict the Q-values, and one secondary network (target) to calculate the Q-target.
- **NN weight update**: The NN weights $\theta$ are updated using stochastic gradient descent to minimize the loss-function. In the DQN the loss-function is the mean square error between the predicted Q-function and the Q-target.

  $$MSE = \mathbb{E}\left[(y_t - \mathrm{Q}(s_t, a_t; \theta))^2\right]$$

  where $y_t$ is the Q-target for the state $s_t$ and the action $a_t$; while $\mathrm{Q}(s_t, a_t; \theta)$ is the predicted Q-value for the state $s_t$ and the action $a_t$

The reward function is crucial for DQN as it guides the agent's learning by providing feedback on the quality of its actions. We defined a reward function based on three parts:

- The current state of the lander is used to compute a dense reward based on several metrics. With this dense reward $r_i$ and the previous dense reward $r_{i-1}$ we compute a potential-based reward.
- Actions that reduce fuel consumption receive a slight decrease in reward, which stabilizes the tendency of the learned policy to land correctly and avoids unnecessary oscillatory behavior. Furthermore, consuming less fuel is a highly desirable property, although it is not explicitly required by the specifications.
- When the lander completes its task, the reward is adjusted. If the lander lands without both legs making contact, it receives a significant negative reward, otherwise, it receives a positive reward.

As for the NN, we decided to use small ones, with the online and the target networks having two fully connected inner layers, each with 64 neurons. After some trial and error, we found this to be a good balance to avoid overfitting. The NNs were implemented using the Keras Python library [2], which initializes the NN weights using the Xavier normalized initialization method [4].

## 5 Experimental setting

To experiment with the solution that we have designed, we have tried to choose reasonable values for the parameters described so far, but we will not justify them as this would be beyond the scope of this project. Furthermore, we will omit the hardware specifications of our setup as we are not interested in the execution time.

Starting from the environment, the arena shown in Figure 1 as a reference has a height of $\pm 1.5$ and a width of $\pm 1.5$. We do not specify the units of measurement, as they are not straightforward and they are not relevant for our purposes (see the Gym documentation for more details). We have distinguished the ranges of initial conditions required by specifications from those used during training, as shown in Table 1. This is common in RL because it enhances the subset of states that the agent encounters during training.

| State | Requirements | Training |
|---|---|---|
| $x$ | $[-0.05, 0.05]$ | $[-0.10, 0.10]$ |
| $y$ | $[1.39, 1.41]$ | $[1.39, 1.41]$ |
| $v_x$ | $[-0.50, 0.50]$ | $[-1.00, 1.00]$ |
| $v_y$ | $[-0.30, 0.00]$ | $[-0.70, 0.70]$ |
| $\theta$ | $[-0.10, 0.10]$ | $[-0.40, 0.40]$ |

Table 1: Admissible ranges for initial state values, both those imposed by requirements and those relaxed for training.

For the parameters of the reward function described in Section 4 see the function `get_reward()` in the file `DQN.py`, link in Section 8. While the training parameters were: a batch size of 64, a soft update parameter ($\tau$) of 0.001, an epsilon decay rate of 0.6, a minimum epsilon value of 0.01 and a learning rate of 0.001. The training runs for a maximum of 5 000 episodes, with each episode allowing up to 1 000 steps (the sampling rate is that given by Gym), using a discount factor ($\gamma$) of 0.995 and network updates every 4 steps.

For the STL formulas described in Section 3, the parameters were: for general horizontal position $c_{\Phi_1} = 0.80$, for rotation $y_{\Phi_2} = 0.40$ and $c_{\Phi_2} = 0.60$. The vertical speed parameters were $y_{\Phi_3} = 0.40$ and $c_{\Phi_3} = 1.00$. The horizontal position of the landing zone had $y_{\Phi_4} = 0.10$ and $c_{\Phi_4} = 0.25$. Finally, the landing completion parameter was $c_{\Phi_5} = 0.10$.

Finally, the parameters for the evolutionary algorithm implemented for the falsification were: $n_{\text{parents}} = 10$, $n_{\text{offspring}} = 2$, the proportional factor of the mutation variance was 0.05, $\max_{\text{retries}} = 100$, and $\max_{\text{evaluations}} = 2000$. Furthermore, the falsification procedure was repeated 5 times to increase the probability of falsifying the specifications.

## 6 Results

The solution that we have designed demonstrated significant convergence and stability properties in the average reward behavior during training, as can be seen in Figure 2.

Furthermore, throughout the falsification process, the vast majority of simulations led to successful landings, with only a few exceptions. Specifically, only 6 falsifying initial states were identified among the 10 000 simulations conducted. Figure 3 shows the initial conditions of the individuals with the lowest fitness of each generation for all 5 falsification runs. Among them, only one instance deviated due to exceeding the theta constraints, while the others were characterized by a falsification of the $x$ landing position. In all the latter cases, although the spacecraft initially landed within the specified range, it began to slide down a hill immediately afterwards due to the specific shape of the terrains, all similar to the one shown in Figure 4. So the problem seems to be the specific shape of the terrain, which is also supported by the fact that in Figure 3 there seems to be no pattern in the initial state of the falsifying trajectories. In particular, when the landing zone is on a high peak,

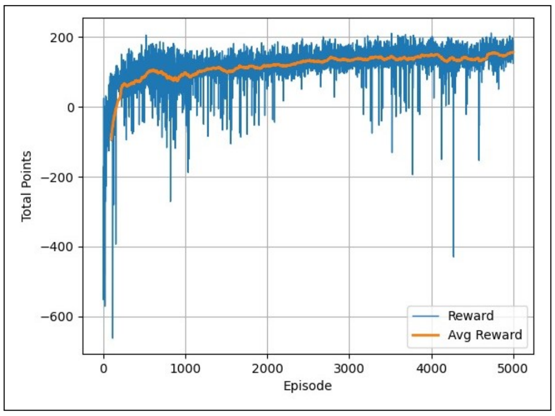

Figure 2: Cumulative discount reward on 5000 episodes.

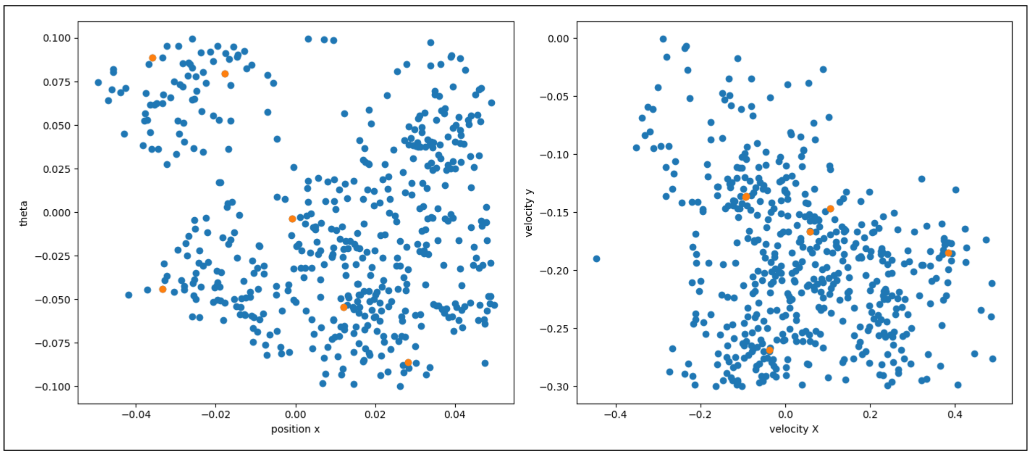

Figure 3: Initial conditions of the individuals with the lowest fitness of each generation for all 5 falsification runs. For clarity, we present the initial conditions in two plots, one for the $x$ position and angle, and one for the velocities. We omit the $y$ position as it does not provide any interesting insights. Orange dots indicate initial conditions that lead to falsification.

the entire body –including both legs– of the spacecraft should be within the landing zone to avoid slippage. However, up to now we have always considered a landing to be successful if the center of the spacecraft was within the specified area. This explains the problem with the landing zones on high peaks. It's also important to note that we were unable to modify the terrain using the simulator, which generates a random shape at the start of each simulation. We have tried to overcome the problems caused by the particular terrain shape, we tried increasing the reward for the $x$ landing position. While this adjustment aimed to improve landing accuracy, it introduced instability into the agent, encouraging abrupt responses that led to oscillatory behavior. This result highlights the delicate balance required when optimizing the reward function to ensure both position accuracy and overall landing stability.

## 7 Conclusions and Future Work

The model has shown excellent convergence and stability. Our reward function proved to be highly effective, quickly and accurately capturing the task requirements. This facilitated an efficient learning process, allowing the model to perform the landing with minimal adjust-

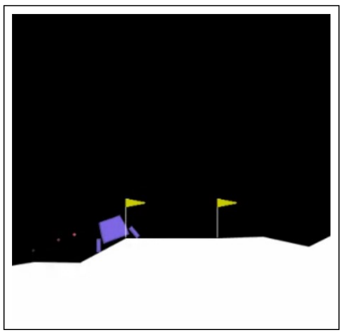
Figure 4: Specific ground shape

ments. The controller appears robust across the specified ranges of initial conditions but exhibits limitations with certain terrain shapes. For the learned model, our analysis suggests that it would be preferable to select a landing zone at the bottom of a valley rather than on a peak to avoid issues related to slipping and subsequent departure from the landing area. This consideration can help mitigate instability problems and improve landing success rates.

Future work could focus on scenarios where landing zones on high peak cannot be avoided. It might be helpful to force the agent to attempt problematic trajectories by incorporating falsification during training. Specifically, the trajectories that lead to falsification during the training procedure could be added to the training memory. This approach could improve its performance on the most challenging paths.

## 8 Reproducibility

Code, trained model and other materials model are available in the folder accessible through the following link: `https://github.com/elsalibymichel/CPS-Project_LunarModule.git`. The folder contains a `README` file explaining its contents.

## References

[1] *Gym documentation: https://gymnasium.farama.org.*

[2] *Keras documentation: https://keras.io/guides.*

[3] *Moonlight documentation: https://github.com/MoonLightSuite/MoonLight/wiki.*

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feed-forward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[5] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[6] Eduardo F. Morales, Rafael Murrieta-Cid, Israel Becerra, and Marco A. Esquivel-Basaldua. A survey on deep learning and deep reinforcement learning in robotics with a tutorial on deep reinforcement learning. *Intelligent Service Robotics*, 14:773 – 805, 2021.