Diverse Reinforcement Learning Approaches for Tackling the Classic Rocket Trajectory Problem

Elie Mulamba
emulamba@aimsammi.org

Ndeye Ngone Gueye
nngueye@aimsammi.org

Solafa Fadlallah
sfadlallah@aimsammi.org

1 Problem statement

The main objective of this project is to address the lunar lander challenge within the OpenAI gym framework[3] by employing reinforcement learning techniques. In this imitated environment, we reproduced a setting where a spacecraft must carefully land at a selected spot under the influence of low gravity. The primary goal of this task is to guide the agent in achieving a smooth and fuel-efficient landing on the landing pad. While the state space closely mirrors continuous physics, the available actions are discrete in nature.

2 Model

2.1 Framework

The lunar lander challenge is tackled within the framework of "gym," a toolkit created by OpenAI[3] that obliges the intention of developing and bench-marking reinforcement learning algorithms. Gym presents support for a wide range of learning environments, spanning from Atari games to robotics applications. In our specific setup, we utilized a simulator known as "Box2D" and the corresponding environment is referred to as "LunarLander-v2"[1].

2.2 State and observations state

2.2.1 Starting state

In every episode, the rocket starts from a consistent position, precisely at the top center of the view-port. The variable aspect here is that, initially, a random force is exerted on the rocket's center of mass.

2.2.2 Observation space

The observation space provides information regarding different characteristics of the lander. To be more specific, it encompasses eight state variables that are connected to the state space and they are: the lander's horizontal position (x-coordinate), the lander's vertical position (y-coordinate), horizontal velocity (v_x), vertical velocity (v_y), orientation in space (θ), angular velocity (v_θ), whether the left leg is touching the ground (Boolean) and whether the right leg is touching the ground (Boolean). The landing pad is identified by the coordinates (0, 0) for x and y position.

[1]The code is written using **Jax** and **Haiku** frameworks. It can be found at https://github.com/Solafa11/RL_PROJECT1.git.

2.3 Actions

There are 4 actions within this environment:

- 0: Taking no action
- 1: Activating the left orientation engine
- 2: Activating the main engine
- 3: Activating the right orientation engine

2.4 Rewards

Launching a proper reward system significantly impacts the agent's performance[6]. The agent's objectives include maintaining a stable mid-air posture and expeditiously reaching the landing pad. In this particular scenario, there is a fixed landing platform at coordinates (0,0) which remains constant. There is no limit to the amount of fuel available. The overall reward for successfully guiding the lander from the upper portion of the screen to the landing platform ranges from 100 to 140 points, contingent upon the precise landing position on the platform. If the lander deviates from its course towards the landing platform, it incurs a penalty equivalent to the reward it would have gained by moving closer to the platform. An episode concludes if the lander experiences a collision or comes to a complete stop, as indicated by a binary flag from the environment. In such instances, the lander incurs an additional -100 points for a collision or gains +100 points for a safe landing. Furthermore, each time a leg touches the ground, it earns the lander +10 points, but using the main engine results in a penalty of -0.3 points on each occasion. Activating the side engine incurs a penalty of -0.03 points per frame.
Goal: Success in this challenge is defined as maintaining an average score of 200 points or greater over a span of 200 consecutive landing attempts.

2.5 Episode termination

Every episode commences from the specified initial state and continues until the terminal state is triggered. The terminal state can occur under any of the following conditions:

- The rocket ventures beyond the visible area (extends too far to the left or to the right).
- The rocket collides with the moon's surface (fails to land on the platform and hits the lower boundary).
- A state of rest is attained, meaning the rocket is not in motion and has not encountered a collision. This encompasses instances where the rocket has successfully landed on the landing platform.

3 Background

The lunar lander environment presents a challenge due to its large, continuous state space. This restricts the choice of suitable algorithms to those that do not assume perfect knowledge of the environment. Dynamic programming, for instance, is not feasible here because it cannot efficiently estimate the state-value function. On the other hand, Monte Carlo methods do not require perfect knowledge of the environment and learn from the agent's experiences, making them a better fit for large state spaces. However, one drawback of Monte Carlo methods is their slow convergence since they require a complete episode for learning and need to store the action-value function and all returns in memory, potentially consuming a lot of space.
Temporal-difference methods, another family of algorithms, overcome the slow convergence issue of Monte Carlo methods. They can learn at each step by using bootstrapping while still relying on the agent's experiences. Q-learning is an example of such an algorithm, and it is an off-policy method, allowing it to learn the optimal policy, not just the one it's following.
However, all these methods share a common characteristic: they assume finite state and action spaces, allowing for a tabular representation of the Q-value estimate. To extend their applicability to environments with a large number of states, function approximation for the state-value function becomes essential. DeepMind, for instance, used neural networks as function approximators to tackle

Atari games, effectively removing the constraint of state space size since there's no need to maintain a Q-table in memory.

4 Approaches

The decision was made to implement two distinct algorithms for training our agents. This choice was made to facilitate a comparative analysis of their effectiveness in addressing this specific problem. The two chosen algorithms are: DQN [2] and DDQN [4] algorithms.

4.1 Deep-Q Networks (DQN)

DQN introduced by[2], marked a significant advancement by combining traditional reinforcement learning techniques, specifically Q-Learning as proposed by[1], with deep neural networks. This innovation was groundbreaking as it enabled agents to approximate Q-values within previously challenging large state/action spaces.
In the DQN approach, observations from the environment are input into a neural network, and training is conducted through gradient descent with the objective of minimizing a chosen loss function, typically Mean Squared Error. The algorithm shares similarities with Q-learning but incorporates several key enhancements such as a replay buffer, a target network to stabilize the learning process and function approximation to address large state-action spaces.
Learning in the DQN is primarily performed with the prediction network, and periodically, the target network is synchronized. Additionally, a replay buffer is utilized to store all encountered transitions in the environment.
For a given state s, the DQN produces a vector containing action values $Q(s, \cdot; \theta)$ where θ is the network parameters. The target network, characterized by its parameters θ^-, is identical to the online network, with the only difference being that its parameters are replicated every τ steps from the online network so that $\theta_t^- = \theta_t$.The target used by DQN is given by[4]:

$$Y_t^{DQN} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-) \tag{1}$$

In the case of DQN, a single neural network serves two crucial purposes: selecting the agent's actions and assessing the target Q-values. However, this setup can introduce what is known as a "maximization bias," where the agent implicitly relies on overestimated values when estimating the maximum value. [5] provided an example of how this bias can lead to the selection of incorrect actions by the agent.

4.2 Double Deep-Q Networks (DDQN)

To tackle the maximization bias challenge, a developed edition of DQN, known as Double Deep Q-Network (DDQN), incorporates a separate neural network for renewing target values. This additional neural network has the same design and architecture as the one used for action selection, and it is episodically updated with the weights from the action-selection network. By diminishing the frequency of updates to the target network, the target values remain more stable. Subsequently, this enhancement to the architecture makes the action-selection network more effectual, leading to quicker convergence and more robust learning by the agent.
The two separate networks in the DDQN can be employed to mitigate instability arising from off-policy learning, function approximation and bootstrapping occurring simultaneously in the DQN - this problem is referred to as the "deadly triad[7]".
The target used by DDQN is given by[4]:

$$Y_t^{DDQN} = R_{t+1} + \gamma Q(\max_a Q(S_{t+1}, \arg\max Q(S_{t+1}, a; \theta_t), \theta_t^-)) \tag{2}$$

5 Results and discussion

Diverse neural network designs, both larger and smaller, were experimented with dissimilar training batch sizes for the DDQN. Remarkably, the larger neural network did not drastically impact the overall training period for the same number of episodes, while utilizing a smaller batch size fast-tracked the training process, as point out in the table 1. The motive behind performing these initial experiments

Table 1: Tests of different batch and network sizes

Network - No. batch	Time(min)
64 layer - 64	534
64 layer - 256	562
64/256 layer - 64	576
64/256 layer - 256	609

Table 2: hyper-parameters settings

Hyper-parameter	Value
batch size	256
replay size	500,000
discount factor γ	0.9
$\epsilon(start)$	1.0
$\epsilon(end)$	0.01
learning rate	0.0005

on various architecture sizes was to recognize models that are fit for more broad testing including parameter modifications. It's worth noting that the neural network size had slight influences on both training and testing durations. As a result, only a casual examination on these sizes was performed. The main goal of the project was not so much to determine the perfect node/layer combination but rather to assess how parameter values impact the overall architecture and affect the learning process in the lunar lander environment.

Among the two tested approaches, the DDQN with a neural network architecture consisting of two hidden layers, each with 256 neurons and employing the RELU activation function, demonstrated the best performance. This DDQN model used a batch size of 256 and a replay memory of 500,000 proved to be adequate for storing a sufficient amount of historical steps to train the mode. It was trained for up to 10,000 episodes regardless whether the trial was successful.
With such settings, it has been noticed that the convergence -the average score of the last 200 episodes surpassed 200 points- has been approximately approached for the DDQN. When using identical sets of hyper-parameters, the standard DQN underperformed the DDQN.
The values of the hyper-parameters chosen are based on empirical observations. These values are given in the table 2. The same settings of the hyper-parameters has been used for both DQN and DDQN for comparison purposes.

In the subsequent section, a graphical representation illustrating the average rewards and average loss for both the DQN and DDQN models is presented for analysis and reference.

5.1 DQN

The average return in the training phase of the DQN reached 200 almost after episode 1600 as it is shown in figure 1.

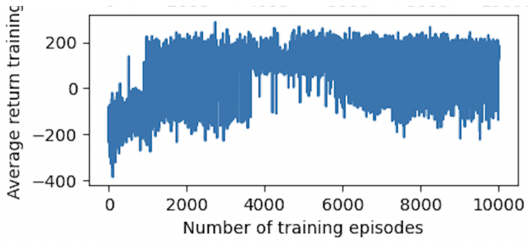

Figure 1: Average training return for DQN

The average return during the evaluation phase spotted after to commence positive trend after 8000 episodes. However, it is noteworthy that this value remained drastically distant from the required threshold of 200. This divergence suggests that the DQR model is exhibiting sub-optimal performance within the Lunar Lander environment, given the current configuration of hyper-parameters as shown in figure 2.

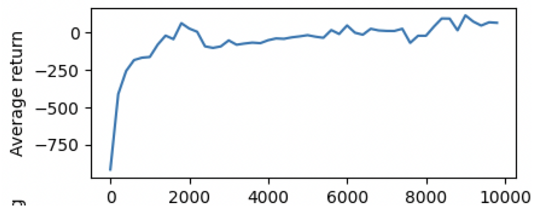

Figure 2: Average evaluation return for DQN

The average loss of the DQN is presented in figure 3 below.

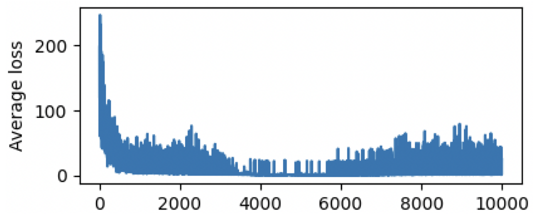

Figure 3: Average loss for DQN

5.2 DDQN

The mean return noted during both the training 4 and evaluation 5 phases outstandingly exceeds that achieved by the DQN model employing identical hyper-parameter settings.

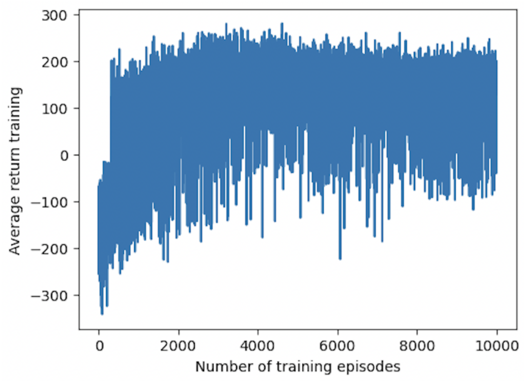

Figure 4: Average training return for DDQN

Furthermore, the average loss 3 incurred by the DDQN model is substantially lower compared to that of the DQN. These findings provide compelling evidence of the superior performance of DDQN, illustrating a remarkable degree of convergence toward the desired threshold.

6 Future work

It would be noteworthy to examine and assess the convergence rate of deep reinforcement learning approaches when contrasted to more basic methods found in the temporal difference algorithm family, such as SARSA or Dyna-Q. A further exciting aspect is to perform a more inclusive hyper-parameter optimization trials to identify the optimal neural network architecture. For instance, judging whether a convolutional network or a dueling network yields a cut above performance could be valuable.

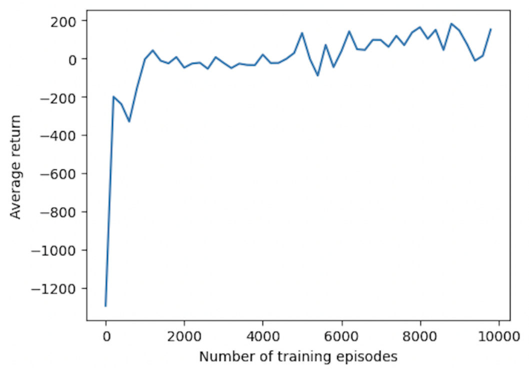

Figure 5: Average evaluation return for DDQN

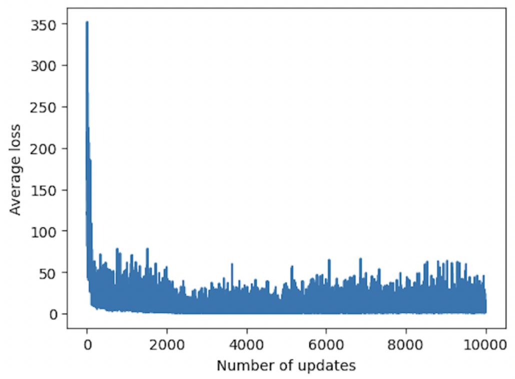

Figure 6: Average loss for DDQN

7 Personal experience

Conducting a complete examination of potential parameter values was unfortunately proved to be challenging due to the significant amount of time required to train and evaluate each model.
One of the most puzzling aspects of this project lies in the interior of the time frame needed to train and evaluate multiple models. Due to the available time frame constraints, we were only able to perform a limited number of hyper-parameter tests. This drawback could potentially affect the robustness of our experiment. It was both captivating and, at times, provoking to fine-tune hyper-parameters that would return pleasing results without significantly extending the experiments in terms of time and computational resources.
Assessing all possible combinations of hyper-parameters might have produced finest solution -hypothetically- but this may not have been perceptible.

7.1 JAX vs PyTorch

In the quest of solving the Lunar Lander problem, it is worth noting that for the algorithms coded in PyTorch framework, the training process exhibited astonishing efficiency, ending within a mere 20 minutes. This efficiency led to accomplishing pleasing results within 1500 episodes only. On the other hand, the utilization of the Jax and Haiku framework for the same model presented a distinct scenario. Training within this framework extended well beyond the 10-hour, emphasizing the considerably higher computational demands inherent to Jax and Haiku. This considerable discrepancy in training duration underlines a critical trade-off when selecting a framework: PyTorch's swiftness in achieving fair performance with moderate computational resources versus Jax and Haiku's computational intensity, which may be necessary for achieving peak performance but significantly requires time-consuming training.[2].

[2]A visual illustration of the results obtained using PyTorch frame is found in the repository

References

[1] Christopher JCH Watkins and Peter Dayan. “Q-learning”. In: *Machine learning* 8 (1992), pp. 279–292.

[2] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[3] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[4] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[6] 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.

[7] Shangtong Zhang, Hengshuai Yao, and Shimon Whiteson. “Breaking the deadly triad with a target network”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 12621–12631.