Reinforcement Learning for Solving Lunar Lander in Gymnasium

Chester Lee

Department of Computer Science, California State University-Fresno

chester07@mail.fresnostate.edu

Abstract— This study explores the application of Deep Q-Learning to the Lunar Lander environment, a control problem available in the Gymnasium suite. The project investigates how reinforcement learning can train an agent to safely land a spacecraft within a specified zone. This study will describe the domain's dynamics, implement a Deep Q-Network (DQN) model, and optimize it through systematic hyperparameter tuning. The study evaluates the agent's performance using metrics such as rolling mean rewards and success rates, presenting insights into the strengths and limitations of the model. Furthermore, we propose potential enhancements and discuss the future applicability of reinforcement learning in more complex domains.

Keywords— Lunar Lander, Deep Q-Learning, Reinforcement Learning

I. Introduction

Reinforcement learning (RL) is a popular way of solving problems in sequential decision making in dynamic and uncertain environments. The Lunar Lander environment in the Gymnasium provides a rigorous testing ground for RL methods because it involves non-linear dynamics, limited controls, and safety-resources trade-offs. This project focuses on training an RL agent to solve the Lunar Lander problem using Deep Q-Learning (DQN), which is basically Q-Learning in conjunction with deep neural networks.

Our main objectives are to create an extensive evaluation of the Deep Q-Network with respect to the Lunar Lander task, experimenting with the hyperparameter tuning to improve performance, and thorough analysis of the results generated. This paper is organized in sections such that section II will outline the methodology used in this study including experimental design. Section III will demonstrate the procedures taken in running this experiment whereas section IV discusses the results obtained and analysis made on these results. Conclusion and future work are discussed in Sections V and VI.

II. Methodology

Deep Q-Learning is one of the most widely used algorithms for RL-related tasks with discrete action spaces, such as playing games on Atari and balancing CartPole. In this study, we will use this algorithm to attempt to land the spaceship model on the ground. There are also other algorithms that are possible for this lunar lander problem such as Proximal Policy Optimization and Actor-Critic methods, which are approaches that focus on policy optimization. On the other hand, DQN is a value-based RL method that primarily focuses on discrete action space problems.

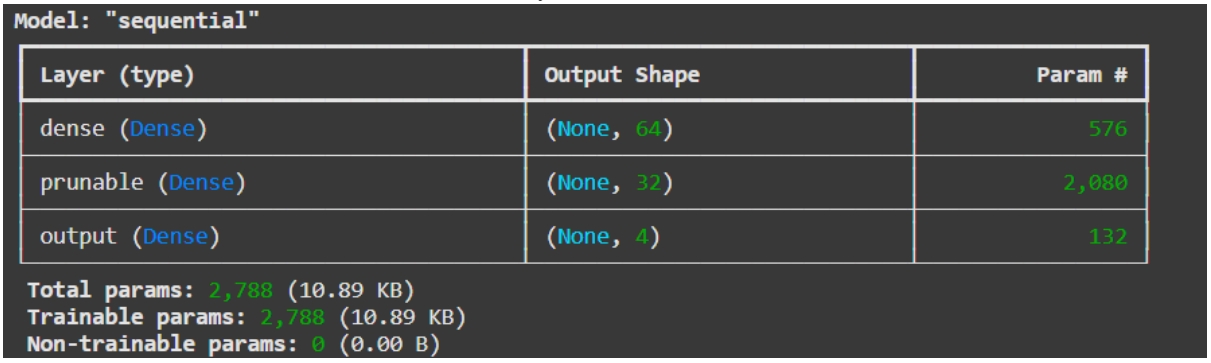
Lunar Lander environment in Gymnasium is a popular RL simulation where the agent needs to land a spaceship safely on a designated landing pad on a 2D plane. In this study, we will work on the base problem without considering changing the parameters of the physics-based dynamics such as gravity, wind, and turbulence. The environment has discrete and continuous versions of action space where the agent can control the main engine and the steering engine. The observation space for this environment is an 8-dimensional vector where it consists of the horizontal position, vertical position, horizontal velocity, vertical velocity, orientation angle, angular velocity and two Boolean indicate whether the two legs of the lander touch the ground. As for the reward structure, the rewards are determined by the distance between the lander and the landing pad, speed of the lander, titled angle, lander legs in contact with the ground, firing the engine, and most importantly, crashing or landing safely.

III. Experiment

The objective of this experiment is to train a DQN agent with Lunar Lander simulation. This project utilizes multiple Python tools such as Keras, Gymnasium, Matplotlib, etc. The overall experiment contains two sections, training the agent, testing the agent, and repeating with different hyperparameters.

A. Defining and Training the Model

Figure 1 demonstrates the summary of the model, which includes two dense layers and an output layer. We decided to fix the nodes to 64 and 32 for the dense layers.



Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	576
prunable (Dense)	(None, 32)	2,080
output (Dense)	(None, 4)	132

Total params: 2,788 (10.89 KB)
Trainable params: 2,788 (10.89 KB)
Non-trainable params: 0 (0.00 B)

Figure 1. Model Summary

B. Testing the Model

After finishing the training process, we loaded our results and displayed them in tables, graphs, and bar charts. We also ran a trial run with the agent and generated a visual gif to display the results.

IV. Result, Analysis and Limitations

For the results, we ran the agent with different hyperparameters such as the decaying factor, number of dense layers, minimal epsilon, episodes timed out steps, and buffer size. In total, we ran the agent three times and the results were tabulated.

<u>First Run</u>

Starting epsilon: 1

Decaying factor: 0.99995

Number of prunable dense layers: 64

Minimal epsilon: 0

Episodes timed out steps: 500

Buffer size: Unlimited

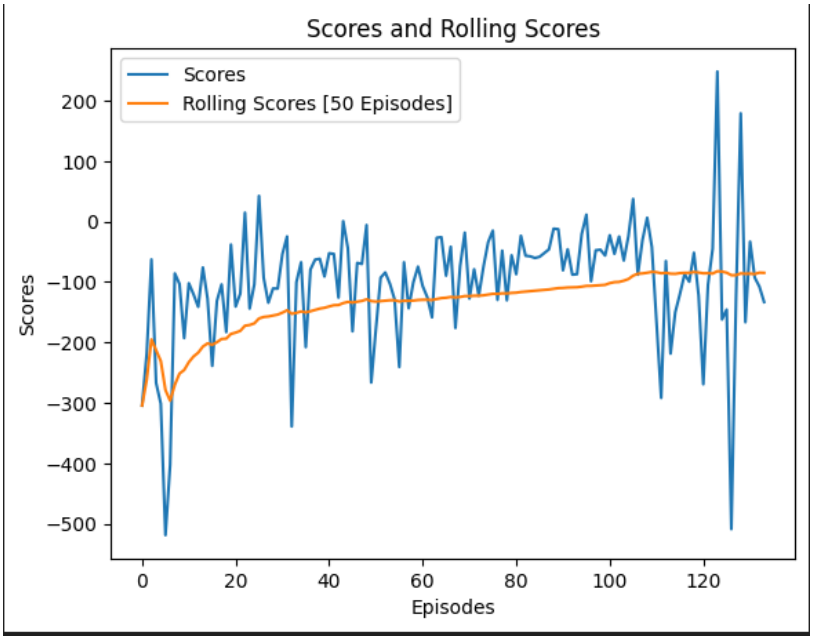
```
Successes: 2 Crashes: 132 Timeouts: 0
```

Top Scores

	Steps	Reward	Score	R50_Score	Done	Crashed
123	479	100	248.397809	-82.170884	True	False
128	449	100	179.142294	-85.816941	True	False
25	87	-100	42.487970	-160.416607	False	True
105	104	-100	37.468598	-89.540548	False	True
22	119	-100	14.771590	-172.400643	False	True

Bottom Scores

	Steps	Reward	Score	R50_Score	Done	Crashed
0	83	-100	-304.530655	-304.530655	False	True
32	114	-100	-339.177207	-152.662985	False	True
6	96	-100	-402.508642	-296.515302	False	True
126	88	-100	-509.105748	-88.792171	False	True
5	97	-100	-519.078237	-278.849746	False	True



In this run, the agent ran 134 episodes with only 2 successful episodes. The weird number of episodes is due to the simulation crash caused by memory issues. As the buffer size for past episodes is unlimited, the memory usage increases linearly causing issues. We can observe that the rolling score is increasing slowly but halts after 100 episodes. It might be caused by system instability when a memory leak occurs.

Second Run

Starting epsilon: 1

Decaying factor: 0.9997

Number of prunable dense layers: 64

Minimal epsilon: 0.05

Episodes timed out steps: 500

Buffer size: 5000 steps

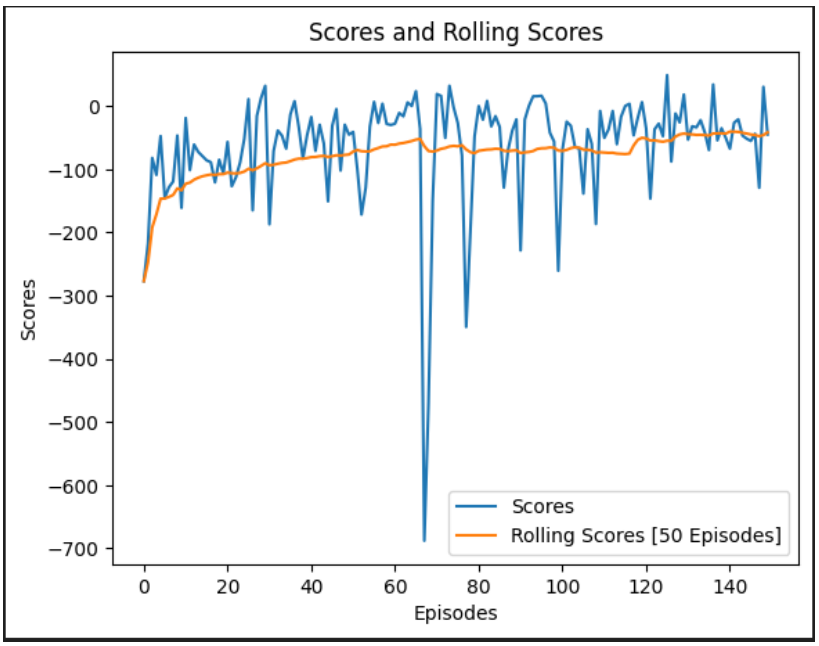
```
Successes: 0 Crashes: 49 Timeouts: 101
```

Top Scores

	Steps	Reward	Score	R50_Score	Done	Crashed
125	500	-2.229802	48.833068	-54.925984	False	False
136	500	-0.007980	34.075273	-43.473062	False	False
29	500	-0.697557	32.023964	-90.547481	False	False
73	500	-1.195849	32.001282	-63.998005	False	False
148	500	-1.135161	30.339317	-45.962958	False	False

Bottom Scores

	Steps	Reward	Score	R50_Score	Done	Crashed
99	213	-100.0	-260.641999	-70.362943	False	True
0	67	-100.0	-276.993687	-276.993687	False	True
77	162	-100.0	-349.575400	-68.713358	False	True
68	270	-100.0	-470.600712	-71.174481	False	True
67	172	-100.0	-687.867967	-63.462219	False	True



In this run, we introduced the buffer size to avoid the previous issue and changed the minimal epsilon and decaying factor. We wished the results could converge much earlier with changed epsilon as the program will still have increasing memory usage even after buffer introduction. The change in epsilon does not have an obvious effect on the results. The agent ran for 150 episodes and did not succeed in any of the runs.

Third Run

Starting epsilon: 1

Decaying factor: 0.99995

Number of prunable dense layers: 32

Minimal epsilon: 0.1

Episodes timed out steps: 1000

Buffer size: 10000 steps

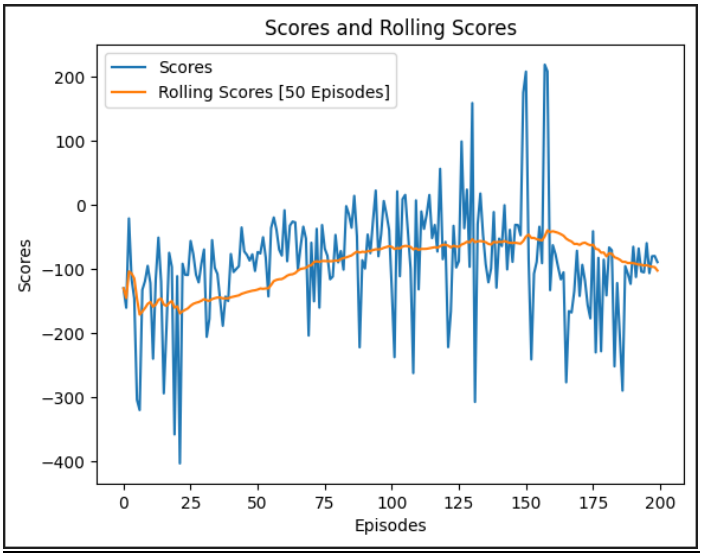
```
Successes: 5 Crashes: 133 Timeouts: 62
```

Top Scores

	Steps	Reward	Score	R50_Score	Done	Crashed
157	648	100.0	218.338840	-49.541282	True	False
158	559	100.0	207.969341	-40.124839	True	False
150	692	100.0	207.450301	-49.874924	True	False
149	483	100.0	174.728553	-56.678971	True	False
130	751	100.0	158.376213	-53.982476	True	False

Bottom Scores

	Steps	Reward	Score	R50_Score	Done	Crashed
5	97	-100.0	-304.473562	-146.771911	False	True
131	716	-100.0	-307.866565	-58.689870	False	True
6	97	-100.0	-320.489939	-171.588772	False	True
19	121	-100.0	-358.521407	-160.910899	False	True
21	85	-100.0	-403.799483	-169.714589	False	True



In this run, we changed back the epsilon value and increased the minimal epsilon as we tried running a longer experiment and the agent had more time to explore the simulation. We also reduce the dense layer in order to reduce the runtime. Lastly, we changed the buffer size and episode timed-out steps because we noticed most of the previous episodes resulted in a timeout where the agent chose to hover the lander instead of landing it. The agent ran for 200 episodes and had 5 successful attempts. We can observe that the rolling score is increasing slowly but decreases after 160 episodes. This issue might caused by the small buffer size where it can only store 10

episodes of data in the case when the episode timed out. The successful attempts that are close together also prove the issue with a small buffer size.

Overall, after reviewing the gif for the results, we noticed that the trained agent chooses the hover the lander instead of landing it. This phenomenon is caused by the large penalty if the lander crashes during landing. Even though the agent could earn +100 points when landed successfully, it chose to avoid the crash penalty of -100 points. Buffer size is also an important factor as it determines how many previous experiences the agent can access. The limitations and challenges we faced were to balance the resources between buffer size and memory usage. The run time for the training is also troublesome as 200 episodes take at least 4 hours to complete.

V. Conclusion

In conclusion, our project confirms the potential of Deep Q-Learning in solving the Lunar Lander problem, albeit with limitations that require further exploration. The results show that while the agent can learn to achieve some level of performance, there are significant challenges in tuning hyperparameters to optimize for stability, convergence, and efficiency. Memory management and buffer size emerged as critical factors impacting the success and runtime of training. Additionally, the agent's tendency to prioritize safety by avoiding crashes highlights the influence of the reward structure on the learning process. The findings underscore the importance of systematic experimentation with hyperparameters and model architectures to achieve better performance. Although our approach was constrained by computational resources, the outcomes provide a foundation for refining the application of Deep Q-Learning in this domain.

VI. Future Work

Future efforts to improve this project could focus on several areas. Firstly, incorporating more advanced reinforcement learning algorithms, such as Proximal Policy Optimization (PPO) or Double DQN, might address some of the challenges associated with exploration-exploitation trade-offs and stability. These algorithms could enhance the agent's capacity to generalize and adapt effectively.

Secondly, modifying the reward structure could encourage more proactive behaviors from the agent, such as attempting to land rather than merely hovering. Introducing intermediate rewards for approaching the landing pad and penalizing excessive hovering could shift the agent's strategy.

Another critical improvement involves optimizing computational resources. Techniques like experience replay prioritization, more efficient memory management strategies, or distributed training could reduce the runtime and alleviate memory issues. Furthermore, utilizing cloud computing or GPU acceleration might significantly decrease training times and enable more extensive experimentation.

Finally, expanding the problem to include additional complexities such as wind dynamics, varying gravity, or moving landing pads would provide a more rigorous testbed for the agent and evaluate its adaptability to real-world scenarios. The insights gained from these enhancements could contribute to applying reinforcement learning to more complex and dynamic control problems.