Stability in the Lunar Lander Environment

Dimitri Kachler, Mirette Moawad, Nader Khalil, Alind Gupta, Andrea Zoccante

***Abstract*—This report explores the topic of Stability within two Reinforcement learning algorithms - On-Policy Monte Carlo and Dueling Double Deep-Q Network - applied to the Lunar Lander environment. Stability has utility within Reinforcement Learning in producing robust agents in real-life deployments. We were able to propose stability and robustness claims about the Monte Carlo algorithm, or the lack thereof. The repository can be accessed through this link: https://github.com/Nader-Youhanna/Lunar-Landing**

I. Introduction

This paper explores the concepts of stability and robustness within the domain of reinforcement learning (RL). The significance of stability in reinforcement learning is paramount, especially as RL algorithms are increasingly applied in real-world scenarios. In practical applications, the stability of agents is a crucial concern. Consider the example of a robotic drone designed for package delivery: its primary objective is to ensure the safe and reliable delivery of packages. A scenario where increasing the drone's speed results in frequent loss of direction underscores the failure in achieving a stable and successful agent operation. Moreover, the concept of robustness gains attention when considering that many autonomous controllers, trained in simulated environments, face challenges when deployed in real-world settings. This discrepancy, known as the Reality Gap [12], highlights a substantial decline in performance due to environmental changes, emphasizing the need for RL algorithms to bridge this gap effectively.

The challenge in studying agent stability stems from the absence of one single universal definition or metric for evaluation. Albeit, there certainly do exist quantified metrics that measure specific aspects of reliability [13]. Stability encompasses various facets of an RL solution, necessitating specific focus areas for analysis. Furthermore, the relevance of stability assessments is contingent upon the agent's performance level, as evaluations on underperforming agents may not provide meaningful insights. Drawing on a previous study on stability within machine learning, stability is interpreted as "a low variance," a definition we adopt in our analysis.

Our research methodology involves the utilization of the Lunar Lander environment from Gymnasium [2] as a testbed, applying two distinct algorithmic approaches: a Monte Carlo Method and a Deep-Q Network (DQN) strategy. These algorithms were selected for their differing learning paradigms, offering a comparative perspective on stability. Our training of agents to navigate the Lunar Lander environment culminates in the development of solutions that are subsequently subjected to a series of experiments. These experiments are designed to assess various stability dimensions, including performance, behavioral, and environmental stability, providing comprehensive insights into the stability dynamics of reinforcement learning agents.

II. Background

Reinforcement Learning (RL) is a computational approach to understanding and automating goal-directed learning and decision-making. It is inspired by the way humans learn through trial and error, seeking to replicate this process in agents that interact with a dynamic environment to achieve specific objectives. A key development in RL has been the fusion of deep learning techniques with Q-learning, culminating in Deep Q-Learning (DQL) [11]. DQL employs deep neural networks to approximate the Q-value function, which estimates the value of executing a particular action in a given state. This advancement enables agents, such as the Lunar Lander, to learn optimal policies that determine the best actions for successful landings while maximizing cumulative rewards. The efficacy of DQL methods is underscored in seminal works like "Rainbow: Combining Improvements in Deep Reinforcement Learning" [4] and "Playing Atari with Deep Reinforcement Learning" [3], which have informed our application of both Monte Carlo methods and the Experience Replay Buffer technique.

The DQL update rule is given as below:

$$Q_\omega(s,a) \leftarrow Q_\omega(s,a) + \alpha[r + \gamma \max_{a'} Q_\omega(s',a') - Q_\omega(s,a)] \tag{1}$$

Monte Carlo Method

In the Monte Carlo (MC) method, value functions are estimated and policies are optimized based on average returns from episodes. When updating value estimates based on observed returns [7], MC methods wait until the end of an episode, in contrast to methods that use bootstrapping estimates from subsequent states. This method works especially well for episodic tasks because it makes it possible to assess and refine policies using whole sequences of states, actions, and rewards.

a) Buckets: Within the paradigm of Monte-Carlo methods, we also assume the observation space to take on a discrete representation, to which the Lunar Lander environment is in contradiction as a continuous environment. To render the environment and the agent compatible, we use the method in [5] to discretize the observation state such that state values fall into linearly spaced buckets. For example, the x-coordinate has 8 bins defined on $[-1, 1]$, so an x-value of 0.35 falls in the 6^{th} bin.

We remind you the update rule for the Monte-Carlo method:

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{N(s,a)} \sum (G_t - Q(s,a)) \tag{2}$$

EXPERIENCE REPLAY BUFFER

In this method, the agent's transitions are stored in a memory buffer from which random mini-batches are sampled in order to update the network which stabilizes significantly the learning process. the Experience Replay buffer tackles the problem of correlated experiences and non-stationary distributions. In addition, the Experience Replay buffer allows for complicated prioritization schemes that skew learning updates in favor of more informative transitions.

DUELING DOUBLE DQN

To address the overestimation bias inherent in the original DQN's Q-value approximation, Double Q-learning employs a separate target network to decouple the selection of actions from their evaluation, thereby providing a more stable and accurate estimate of action values. The Dueling Network Architecture further refines this approach by splitting the neural network into two distinct pathways: one to assess the state value function and another to compute the advantage of each action, allowing for a more nuanced understanding of action value that is independent of the state's value. Therfore, DDDQN offers a more precise, efficient, and reliable method for learning optimal policies.

NOISY NET DQN

Noisy Net DQN [16] introduces a creative approach to exploration by integrating parametric noise directly into the network's architecture. This technique, developed to improve the conventional DQN, embeds exploration at the core of the agent's decision-making process by adding trainable noise to the weights of the network layers.

NOTATION

Firstly, we refer to a state as s and an action as a. Additionally, we use the concept of a Q-function in both DQN and Monte Carlo. For Monte Carlo, we have a Q-table, noted by $Q(s, a)$, while DQN uses a Q-network noted by $Q_\omega(s, a)$. Then, $\hat{Q}_\omega(s, a)$ defines the prediction Q-network. Furthermore, s' defines the new state after action a is taken. Then, τ notes a trajectory being a list of subsequent action-state pairs. Finally, σ and μ define the standard deviation and the mean respectively.

III. METHODOLOGY/APPROACH

LUNAR LANDER ENVIRONMENT

The Lunar Lander's goal is to successfully land a spacecraft on the moon. The agent controls the lander's engines to decelerate and navigate to land down on a specific landing site.

a) Our Code Base: During this project, we have worked with some code files that are relevant to this report. They can be accessed via the following GitHub [10] link: Link to GitHub

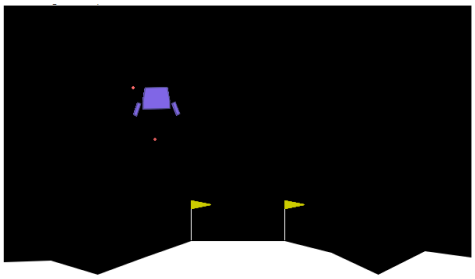

Fig. 1. Lunar Lander environment in action

A. State Space

The state space of the Lunar Lander environment [2] is continuous and has several parameters that capture the current status of the lander:

- Horizontal Position: The x-coordinate of the lander, indicating its position relative to the landing site.
- Vertical Position: The y-coordinate of the lander, representing its altitude above the moon's surface.
- Horizontal Velocity: The rate of change of the lander's horizontal position.
- Vertical Velocity: The rate of change of the lander's vertical position.
- Angle: The orientation of the lander relative to the vertical axis.
- Angular Velocity: The rate of change of the lander's angle.
- Left Leg Contact: A binary indicator signifying whether the left leg of the lander has made contact with the ground.
- Right Leg Contact: A binary indicator signifying whether the right leg of the lander has made contact with the ground.

B. Action Space

The action space in the Lunar Lander environment can be either discrete or continuous, depending on the specific variant of the environment:

- **Discrete Action Space:** In this variant, the agent chooses from a set of discrete actions at each timestep. These actions are:
  1) Fire the main engine.
  2) Fire the left orientation engine.
  3) Fire the right orientation engine.
  4) Do nothing.
- **Continuous Action Space:** The continuous variant allows for a finer and more accurate control over the lander's engines as actions are real-valued vectors indicating the strength and direction of the thrust applied by the lander's engines.

The Lunar Lander environment, especially in its continuous state space variant needs sophisticated RL techniques capable of dealing with the complexity of continuous control tasks, therefore for this phase, we will apply two methods for the discrete variant.

The reward for moving from the top of the screen to the landing pad and coming to rest is about $100 - 140$ points. If the lander moves away from the landing pad, it loses reward. If the lander crashes, it receives an additional -100 points. If it comes to rest, it receives an additional $+100$ points. Each leg with ground contact is $+10$ points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. The solved problem is 200 points.

Our methodology included using two different approaches:

C. On-Policy First-Visit Monte Carlo Control

The Monte Carlo algorithm that we use as a default method, to be improved upon later, is the On-Policy First-Visit Monte Carlo Control algorithm, as shown in (Fig.2), and using its code implementation from [5]. This algorithm is On-Policy, meaning that the behavioral q-table it uses to generate state-action pairs, is the same as its q-table it uses to represent the optimal policy..

Essentially, this algorithm generates an episode, according to its behavior policy, and returns the ordered list of all its $(state, action, reward)$ tuples until the termination state. The learning step then proceeds by retroactively traversing the trajectory τ, at each step updating the Q-table by adding the difference between the q-value, $Q(s, a)$ and the discounted reward, G, normalized by the amount of visits to the state, $C(s, a)$: $Q(s, a) \leftarrow Q(s, a) + \frac{G - Q(s,a)}{C(s,a)}$.

a) Epsilon-Greedy: In order to have a guarantee on covering the entire search space, one must enact an ϵ-soft policy [1], for which we commonly choose ϵ-greedy.

D. Prioritized Replay Buffer

Prioritized Replay Buffer is a technique used to enhance the training of a Deep Q-Network with *Experience Replay*.

Experience Replay, pioneered by DeepMind [3], involves storing and randomly sampling past experiences from the agent's interactions with the environment to break the temporal correlation in the data, improving the stability of learning. The replay buffer stores tuples of the form ***(s, a, r, s')*** in a data structure known as replay buffer. During the learning process, instead of learning from the most recent experience, the agent randomly selects a batch of experiences from the replay buffer. his stochastic sampling approach breaks the correlation between successive experiences, enhancing the overall stability and robustness of the learning process

Our approach, as outlined in the *Rainbow Paper* [4], does not treat all experiences equally. Instead, each experience is assigned a priority based on a certain criterion. The basic idea is to prioritize the replay of experiences that are more informative or challenging for the learning agent. This prioritization is typically determined by the magnitude of the temporal difference (TD) error, which represents the difference between the predicted and actual rewards in a learning update.

As such, we implemented a priority queue data structure. When we push back a sample in the queue, it is inserted in a position based on the TD error, which is computed in the training function. For example, samples with high TD error are inserted towards the front of the queue as they represent experiences that are more informative for the agent or from which there is something important to learn.

The TD error is computed as the difference between the predicted value $\hat{Q}_\omega(s, a)$ of a state-action pair (s, a) and the target value $Q(s, a)$ given by the Bellman equation

E. Dueling Double DQN

Estimating the highest value by using the maximum over-estimated values introduces a maximization bias in traditional DQN, leading to overestimated action values, unstable training, and suboptimal policies. To mitigate this, two separate Q-value estimators are employed, each updating the other, enabling unbiased estimates of action Q-values using the opposite estimator.

We adopted the Dueling Double Deep Q-Network (DDDQN) approach, integrating Dueling Network Architectures with Double Deep Q-Networks (DDQN) to correct action value overestimation and improve state-value learning in reinforcement tasks. DDDQN enhances learning efficiency and policy generalization. It does this by distinguishing between (a) state value estimation, assessing the quality of a state, and (b) action advantage estimation, evaluating the benefit of each action within a state, thus allowing for more generalized policy learning without relying on the advantage stream alone.

Additionally, we incorporated NoisyNet-DDQN in place of the standard DDQN. While the latter uses an online and a target Q-network with epsilon-greedy policies for exploration and exploitation, balancing exploration rates can be challenging. NoisyNet-DDQN introduces parameterized noise into the Q-networks' layers, promoting inherent exploration. This noise variates Q-values for identical states in different passes, enabling exploration without external policies like epsilon-greedy. Moreover, as the network adjusts the noise level over time, it autonomously decides when to explore based on environmental states, potentially enhancing learning efficiency by reducing exploration in familiar states and increasing it in novel ones.

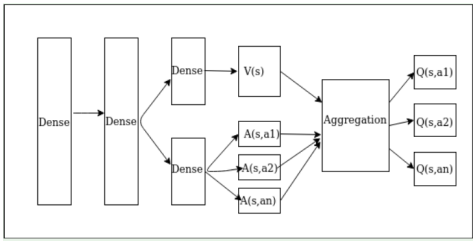

Fig. 2. A typical DDDQN architecture

F. Experiments

In order to evaluate the efficacy of these methods from the perspective of Stability, we have devised 4 experiments to analyse the implications quantitatively.

1) Training Performance: At the most basic level, we should understand whether our agents are capable of solving the Lunar Lander environment or not. If we see that an agent performs poorly, this can indicate that it is training improperly,

and hence impacts the credibility of the results relating to stability. For the Monte Carlo agent we take 4000 episodes, while the Prioritized DQN is allowed 200 episodes. And then we calculate its mean, μ, and the standard deviation, σ.

2) Bucket Sizes: It was pre-experimentally observed that different bucket sizes, could fails sometimes catastrophically. This would indicate that the method for Monte Carlo is non-robust depending on its setup. We plot the training performance of various sizess in order to see if there are any that fail, with multiple trials of n=3.

3) Introduction of Wind and Turbulence: We can study the agent under varying conditions. Hence, we introduce Wind Power and Turbulence to the Lunar Lander Environment to introduce some external effect. Wind is a chaotic force directly applied to the Lander [2], scaled by the Wind Power. Turbulence then sets a cap on how large the angular force can be. We try the following combinations:

- Wind Power : [0,5,10,20,30]
- Turbulence : [0.0, 0.5, 1.0, 2.0, 3.0]

Furthermore, we can collect these combinations within a matrix to see whether any patters arise.

4) Delayed Rewards: One of the problems with applying Reinforcement Learning to real-life applications is that rewards are not always guaranteed to come immediately upon taking an action [9]. As such, it is of interest to observe how agents are able to deal with the uncertainty in the current value of a state. Our experiment follows by withholding the reward for a given interval, e.g. the agent receives a zero for its reward for 20 consecutive steps. Then, it is given all the aggregated reward at once in order to learn. We then plot how effective these agents are after training by studying their average final score.

IV. Results and Discussion

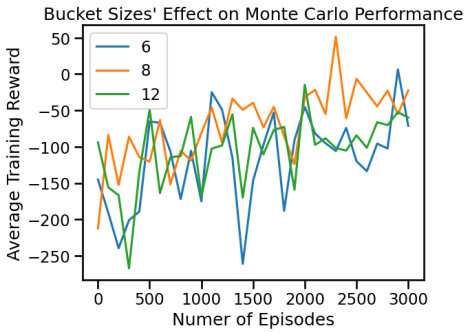

Fig. 3. Despite large changes in the discretization, the performance remains comparable

1) Training Performance: As can be seen in the table below, the Monte Carlo method has a decent final score, hinting at convergence. And this is also corroborated by visual evidence in the form of a video. However, the DQN algorithm (in Appendix) does not seem to converge to a good solution. Therefore, the DQN algorithm has low credibility, while the Monte Carlo method is potentially fit for quanititative claims.

Agent	μ	σ
MC	-35.3	31.1
DQN	-200	N/A

2) Bucket Sizes: Interestingly, the Bucket Sizes seem to have no discernable effect on the training performance in (Fig. 4), and this goes contrary to our hypothesis.

3) Introduction of Wind and Turbulence: As expected, the stronger the winds are, both wind power and turbulence, the worse the agent does on average, as seen by the diagonal of colors in (Fig. 6), where the means, μ, are higher in the top left. However, interestingly the variance seems relatively unaffected by wind forces, as the colors in (Fig. 5) are relatively uniform. This seems to suggest that variance is not necessarily tied to performance.

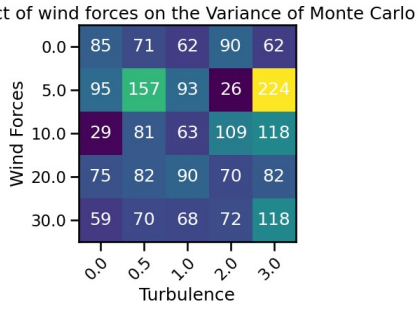

Fig. 4. Variance is left relatively uniform by the wind forces

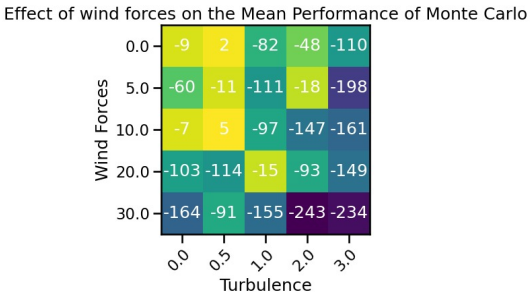

Fig. 5. A diagonal pattern emerges where extra wind forces degrades performance

4) Delayed Rewards: The data seems to suggest that as we increase the intervals between delaying rewards, the agent performs much worse. We have also supplied the standard deviations as the error bars to help quantify how much these values vary, and wether there truly is a pattern or random noise.

5) Discussion: It would appear that we have not trained the DDDQN agent properly such that it we can credibly draw conclusions about its stability. Then, we can conclude our findings as a conglomeration of statements regarding the indication of stability phenomena for the Monte Carlo method with the Lunar Lander environment. Firstly, our findings suggest that Monte Carlo suffers greatly under delayed rewards. It would be interesting to consider whether this is uniquely true of Monte Carlo methods, or if other methods are more robust to it. Furthermore, we do not seem to find that Monte Carlo

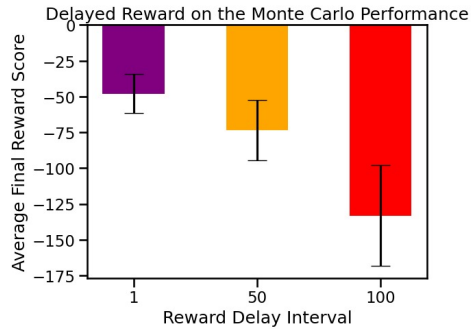

Fig. 6. The bigger the delay is between the rewards, the worse the performance is

is affected by its bucket sizes. Perhaps it is also the case that some trials fail catastrophically, but are not seen in the average. Nonetheless, on average it does not seem that any size is especially prone to this.

V. Conclusion

We have demonstrated the existences of stability and robustness phenomena, namely within the Delayed Reward of our Monte Carlo method. On the other hand, we have also learned that some patterns that might appear to exist on an observational level, might be subject to bias and not present in the quantitative analysis. In regards to training the DQN, we could apply our implementation on a more simple environment in order to discren parameter issues against technical ones. Finally, in order to continue our work, we could compare the Delayed Reward analysis for other methods such as Monte Carlo Policy Gradient, i.e., REINFORCE [11]. Alternatively, we could also explore multi-task reinforcement learning [15], where the agent learns to solve multiple tasks simultaneously, could improve the generalization and robustness of the learned policies.

References

[1] Sutton and Barto. Reinforcement Learning, *MIT Press*, 2020.
[2] "Lunar Lander - Gym Documentation," Gymlibrary.dev, 2022. https://www.gymlibrary.dev/environments/box2d/lunar_lander/ (accessed Mar. 11, 2024). https://www.gymlibrary.dev/environments/box2d/lunar_lander/
[3] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," arXiv.org, 2024. https://arxiv.org/abs/1312.5602 (accessed Mar. 11, 2024). https://arxiv.org/abs/1312.5602
[4] M. Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning," arXiv.org, 2017. https://arxiv.org/abs/1710.02298 (accessed Mar. 11, 2024). https://arxiv.org/abs/1710.02298
[5] omargup, "GitHub - omargup/Lunar-Lander: Tabular Monte Carlo, Sarsa, Q-Learning and Expected Sarsa to solve OpenAI GYM Lunar Lander," GitHub, 2019. (accessed Mar. 10, 2024). https://github.com/omargup/Lunar-Lander
[6] V. Saxena, Burak Guldogan, and Doumitrou Daniil Nimara, "On-policy and off-policy Reinforcement Learning: Key features and differences," Ericsson.com, Dec. 13, 2023. (accessed Mar. 10, 2024). https://www.ericsson.com/en/blog/2023/12/online-and-offline-reinforcement-learning-what--they-and-how-do-they-compare
[7] Read. Lecture V - Reinforcement Learning I. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.
[8] "Lunar Lander - Deep Reinforcement Learning, Noise Robustness, and Quantization," DataSciFinalProj, 2020. (accessed Mar. 11, 2024). https://xusophia.github.io/DataSciFinalProj/
[9] B. Han, Z. Ren, Z. Wu, Y. Zhou, and J. Peng, "Off-Policy Reinforcement Learning with Delayed Rewards," arXiv.org, 2021. https://arxiv.org/abs/2106.11854 (accessed Mar. 28, 2024).
[10] "Build software better, together," GitHub, 2024. https://github.com/ (accessed Mar. 28, 2024).
[11] Read. Lecture VI - Reinforcement Learning I. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2024.
[12] "Closing the Simulation-to-Reality Gap for Deep Robotic Learning," Research.google, Oct. 31, 2017. https://blog.research.google/2017/10/closing-simulation-to-reality-gap-for.html (accessed Mar. 28, 2024).
[13] S. Chan, S. Fishman, J. Canny, A. Korattikara, and S. Guadarrama, "MEASURING THE RELIABILITY OF REINFORCEMENT LEARNING ALGORITHMS." Available: https://openreview.net/pdf?id=SJlpYJBKvH
[14] W. Fedus et al., "Revisiting Fundamentals of Experience Replay." Available: https://arxiv.org/pdf/2007.06700.pdf
[15] Yu Zhang, Qiang Yang, An overview of multi-task learning, National Science Review, Volume 5, Issue 1, January 2018, Pages 30–43, https://doi.org/10.1093/nsr/nwx105
[16] "Papers with Code - NoisyNet-DQN Explained," Paperswithcode.com, 2020. https://paperswithcode.com/method/noisynet-dqn (accessed Mar. 28, 2024).
[17] W. Cheng, X. Liu, X. Wang and G. Nie, "Task Offloading and Resource Allocation for Industrial Internet of Things: A Double-Dueling Deep Q-Network Approach," in IEEE Access, vol. 10, pp. 103111-103120, 2022, doi: 10.1109/ACCESS.2022.3210248. keywords: Deep learning;Servers;Industrial Internet of Things;Resource management;Computational modeling;Energy consumption;Delays;Edge computing;Mobile computing;Mobile edge computing (MEC);task offloading;deep reinforcement learning;Industrial Internet of Things (IIoT),

APPENDIX

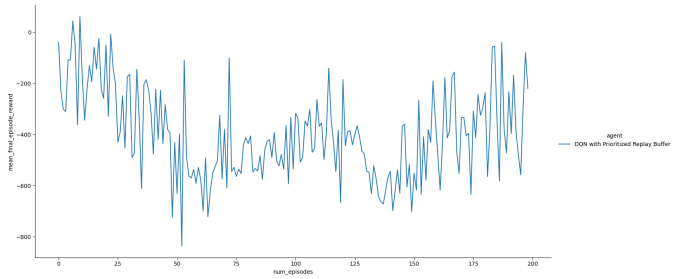

Fig. 7. Spread