Solving Lunar Lander using DQN and DDQN

Bochen Zhao
Department of Computer Science
University of Bath
Bath, BA2 7AY
bz470@bath.ac.uk

Jungho Park
Department of Computer Science
University of Bath
Bath, BA2 7AY
jp2654@bath.ac.uk

Ravikan Thanapanaphruekkul
Department of Computer Science
University of Bath
Bath, BA2 7AY
rt911@bath.ac.uk

Sarat Smitinont
Department of Computer Science
University of Bath
Bath, BA2 7AY
ss4658@bath.ac.uk

Sarut Sunpawatr
Department of Computer Science
University of Bath
Bath, BA2 7AY
ss4659@bath.ac.uk

1 Problem Definition

A landing mission is a challenging problem in the real world as it requires precise control, navigation, and fuel management under the constraints of gravity and the spacecraft's limited resources. The complexity arises from the need to account for various unpredictable factors, such as gravitational anomalies, surface terrain, and the lander's technical limitations.

This project is to simulate the real-world landing problem by using the Lunar Lander environment provided by the OpenAI gymnasium framework (Klimov, 2023). The goal is to teach the Lunar Lander agent to successfully land on randomly generated surfaces using reinforcement learning (RL) algorithms. The agent will learn to land perfectly between two flags on a randomly generated surface by utilising two side engines and one main engine underneath the agent.

The following sections explain the environment setup for this problem.

State Space

The state space for the Lunar Lander environment is represented as an eight-dimensional vector (Klimov, 2023):

- The agent's position coordinate x & y.
- The agent's linear velocity in x & y direction.
- The agent's rotation angle.
- The agent's angular velocity.
- Two Booleans represents whether both agents' legs are in contact with the ground.

Action Space

There are four discrete actions for the agent in this environment (Klimov, 2023): (0)do nothing, (1)fire left orientation engine, (2)fire main engine, and (3)fire right orientation engine.

Rewards

A reward is granted after every step (Klimov, 2023). 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:

- Proximity Reward: Increases when the lander approaches the landing pad; decreases when distancing.
- Velocity Reward: Increases for slow movement; decreases for high speeds.
- Alignment Penalty: A decreasing reward for changing the angle of the lander away from horizontal.
- Ground Contact Bonus: +10 points for each leg touching the ground.
- Side Engine Penalty: -0.03 points for each frame with side engine activity.
- Main Engine Penalty: -0.3 points for each frame with main engine activity.
- Terminate Reward: Assigns +100 for a safe landing, -100 for a crash.

2 Background

Deep Neural Network in Reinforcement Learning

Deep Q-Network (DQN), Double DQN (DDQN), and Dueling DQN represent evolutionary steps in reinforcement learning, all deriving from the core principle of Q-Learning. Each of those has used, with no exception, a deep neural network to represent the Q-function at the core of assessing the state-action pairing rewards.

DQN

With the previous linear learning models, this could hardly lead to AI systems that exhibit human-level performance across various tasks with a small amount of prior knowledge, thus being able to operate solely under specific conditions or in simpler environments or even using only handcrafted features. To this end, the DQN (Mnih et al., 2015) combines reinforcement learning with deep learning rather ambitiously. This results in utilising a deep convolutional neural network (CNN), instead of a conventional Q-table, to approximate the optimal action-value function, a function that computes the maximum sum of future rewards (discounted over future time steps) per each potential state-action pair.

Experience Replay

DQN introduces experience replay, a method to eliminate the correlations between successive updates and stabilise learning (Lin, 1992). It samples from past transition tuples (state, action, reward, next state) in the replay buffer. These tuples are then randomly sampled to train the network, breaking sequences bound to disrupt learning.

Liu and Zou (2018) builds on this basic Experience Replay by proposing and researching an **adaptive Experience Replay** mechanism. Such adaptivity of memory buffer size would be dynamic with changes in the learning context and ameliorate many issues with static memory size, such as overshooting or slow learning rates.

Prioritised experience replay is another advanced form of experience replay that samples crucial experiences more frequently based on their temporal-difference (TD) error (Schaul et al., 2016). This error indicates the unexpectedness of an experience, with higher errors suggesting greater learning value. By focusing on these significant experiences, this method aims for quicker and more reliable convergence and has proven to enhance learning efficiency in complex settings. However, managing a priority queue requires considerable computational resources and risks overfitting or instability by potentially concentrating too much on a few high-error experiences.

Target Network

The DQN algorithm employs a secondary network, called the target network, that is separated from the Q-value updates so that they can be stabilised in updates. This target network is of the same architecture as the main network but updates its weights with a lower frequency, aimed at anchoring the learning targets.

Double Deep Q-learning

In traditional Deep Q-learning, using the max operator in the Bellman equation for selecting and evaluating actions tends to favour overestimated values. This preference can negatively distort the policy. Double Deep Q-learning was originally developed for tabular environments. It was adapted for deep learning by applying function approximation (van Hasselt et al., 2016). It modifies the DQN algorithm by integrating a secondary network that decouples the selection of actions from their evaluation, which in DQN depended on the same value estimates and often resulted in over-estimations.

Dueling DQN

Whereas Double Deep Q-learning follows the strategy of dividing the action selection and evaluation, Dueling Deep Q-learning changes the architecture by dividing the fully connected layers near the output into two streams (Wang et al., 2016). The first stream refers to the Value Function Stream (V) and estimates the intrinsic value of the state independent of any action. Another stream computes the relative advantage of actions at the state compared to the policy's default or average action. Finally, those two streams are combined and summed up to estimate the Q value for every action. This change in architecture allows the network to identify useful states and simultaneously differentiate the effects of every individual action in those states.

To summarise, DDQN and Dueling DQN offer unique improvements compared to the standard DQN model. DDQN adjusts the update rule to decrease overestimation, stabilising and making the learning algorithm more accurate. On the other hand, Dueling DQN permits the network to value the state independently and the advantages of the different actions, hence improving the network's representational capabilities for effective learning and policy evaluation.

3 Method

In this section, five agents, including baseline DQN, DQN with PER, DDQN with PER, Dueling DQN with PER, and Dueling DDQN with PER, are implemented for the experiment. To make this comparison fair and reproducible, the seed of PyTorch and the Numpy environment is fixed across all models. Moreover, all models are trained for 1,750 episodes and optimised using the Adam optimiser.

We chose DQN because it shows several advantages compared to traditional models such as Q-learning or SARSA for complex environments like Lunar Lander. Q-learning and SARSA are based on the look-up tables for storing computationally expensive state-action values. DQN uses neural networks for approximate Q-values, allowing it to generalise to unseen states based on similarities to previously encountered states, Additionally, DQN will compute the whole action-value for the given state in a single forward pass. This can accelerate the action-value computation significantly in environments with many actions.

DQN & DQN with Prioritised Experience Replay (PER)

The DQN architecture was built using three fully connected layers with 256 neurons and employing the ReLU activation functions. This decision was based on an experimental evaluation to match the medium complexity of the Lunar Lander environment.

We applied Huber loss instead of squared loss to provide a stable update. It is a kind of error that penalises less for high loss values, leading to more conservative training updates. It is similar to MSE for small errors, as in the equation:

$$L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{1}$$

During training, we applied epsilon greedy policy with exponential decay of epsilon, allowing for deeper and more active exploring when knowledge is limited early on in the agent's process. Consequently, as the agent's learning and state-action values stabilise, the exponential decay shifts the model's focus towards efficiently exploiting any previously unknown state-action space, leaving Epsilon to handle any unpredictable situations successfully.

We applied Prioritised Experience Replay (PER) because the traditional Experience Replay considers every experience equally, not focusing on the most informative experiences. This method leads to slow learning. To cope with this scenario, we incorporated a stochastic sampling strategy considering

greedy prioritisation and uniform random sampling(van Hasselt et al., 2016). This method ensures a monotonically increasing probability of sampling transitions based on their priority while always allowing a chance for the lowest-priority transitions to be replayed (2). However, Prioritised replay introduces bias by altering this distribution, affecting the accuracy of converged estimates. This issue can be addressed with importance-sampling (IS) weights to correct the bias(3)(van Hasselt et al., 2016). We also used SumTree instead of simple arrays or lists that require costly linear scans for operation because SumTree offers efficient and proportional sampling and quick priorities updates with logarithmic time complexity, as the search space will be reduced by half in each step. As such, it reduces the difficulty level for large datasets.

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2}$$

$$w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta \tag{3}$$

Double Deep Q-Network (DDQN) with Prioritised Experience Replay (PER)

The DDQN also improves on the DQN by tackling the overestimation error of the Q-values update mechanism of the DQN. In DDQN, a critical modification is the separation of action selection and value estimation, accomplished by employing two distinct neural networks: the policy and target networks. The task of the policy network is to determine the action that yields the highest Q-value in the state s', and the objective of the target network is to estimate the value of taking that action by using a slightly old set of weights which is illustrated by (4).

$$y = r + \gamma Q(s', \arg\max_a Q(s', a; \theta); \theta^-) \tag{4}$$

This method, therefore, drastically reduces the risk of value overestimating by setting the parameters so that the optimal action is selected and evaluated, and these are not performed with the same set of parameters, thus building more reliable learning and more robust policies.

Dueling Deep Q-Network (Dueling DQN) and Dueling Double Deep Q-Network (Dueling DDQN) with Prioritised Experience Replay (PER)

On top of DQN and DDQN, we further implemented the Dueling network into the previous two agents: Dueling DQN and Dueling DDQN. The network can converge faster by independently learning the value of a state via the Value Stream and the relative advantage of each action via the Advantage Stream. For instance, in situations where multiple actions in a state have little impact, the agent can still learn the value of the state through the Value Stream, while traditional DQN and DDQN agents may struggle to differentiate these actions when their values are similar. Following the state-value function in (5) ensures that only action with a higher value than the average is advantageous.

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|A|}\sum_{a'} A(s, a'; \theta, \alpha)\right) \tag{5}$$

Our implemented architecture starts with a shared fully connected layer with 256 neurons connected to the input layer, then separates into two paths. Both Value and Advantage streams contain one fully connected layer with 256 neurons and one output layer with 1 and 4 neurons for the Value and Advantage stream, respectively. All fully connected layers are equipped with ReLU activation functions to introduce non-linearity.

4 Results

Figure 1 illustrates the agents' performance. The episodes and the rewards achieved by the agents per episode are plotted on the x-axis and y-axis, respectively. To provide a more stable and fair learning curve (which is to make sure the agent does not get a high reward by luck), the value (reward) achieved by each agent is averaged over its previous 100 episodes, meaning it is more difficult for each agent to achieve a high score. The red-dashed reference line indicates when the score is over 200, meaning that the agent is considered to solve the environment successfully.

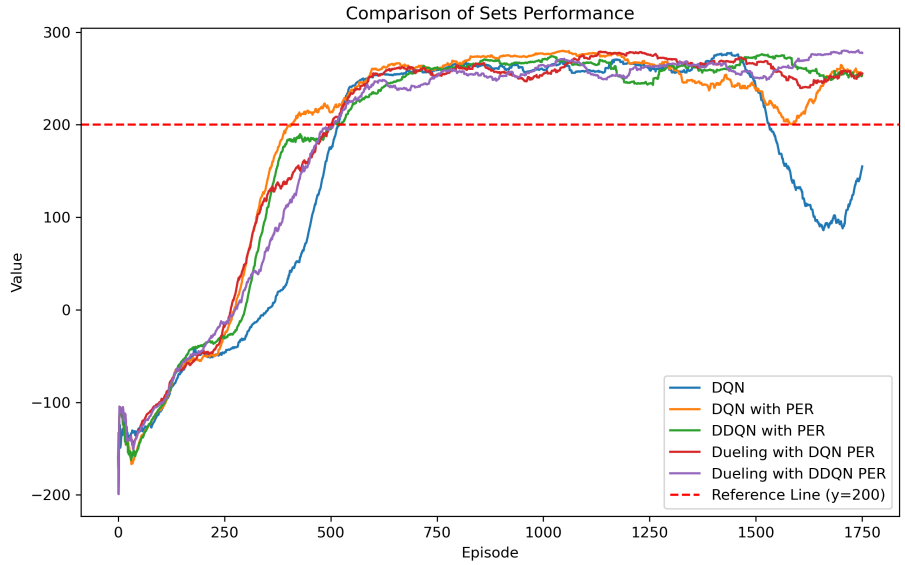

Figure 1: Learning curves for different agents.

To further show the importance of the PER and other variants, we provide a baseline performance of DQN, against which enhancements like PER, Double DQN (DDQN), and Dueling architectures can be evaluated. To make a fair comparison, we used the same set of hyperparameters across all algorithms, which can be found in Table 2. The figure demonstrates that while the basic DQN model establishes a foundational learning curve, incorporating PER enhances learning efficiency and performance robustness. This is evident as the curves associated with PER variants, such as DQN PER and DDQN PER, ascend more swiftly and maintain higher scores than the base DQN model.

Interestingly, all the variants followed a similar learning curve. DQN with PER was the first model to succeed at 406 episodes. As the agents continued to learn, DQN and DQN PER experienced overfitting, and their performance significantly dropped in later episodes around 1500. After reaching the solved state, DDQN PER, Dueling DQN PER and Dueling DDQN PER maintained a more stable learning curve than the other agents, indicating they could sample from more successful events in their replay buffer. Moreover, the Dueling agents managed to achieve the highest average rewards during 1750 episodes of training, which shows that during the long run, the Dueling architectures' unique separation of state value and action advantages, combined with Prioritised Experience Replay, enhances stability and improves the estimation of action values. This architecture allows for a more accurate assessment of the significance of each state and action, leading to more effective learning and prevention of overfitting.

Table 1: Comparisons of all the agents

Agent	**Episode when solved**	**Time taken to solve (seconds)**	**Maximum average reward (.3dp)**
DQN	518	2360.26	277.752
DQN PER	406	1748.52	279.992
DDQN PER	507	2467.93	276.220
Dueling DQN PER	501	3030.06	279.059
Dueling DDQN PER	499	3344.92	280.441

We have further tuned the DDQN PER agent with a higher exponential epsilon decay factor, namely DDQN PER with excessive epsilon decay. The learning curve is illustrated by Figure 2. With a higher decay rate, the DDQN PER agent reached an average reward of over 200 under 300 episodes. Notably, with excessive epsilon decay, the performance dropped in later episodes, becoming very unstable

because it spent less time exploring. Although the standard DDQN PER solved the environment at 507 episodes, it kept a consistent learning curve throughout the training.

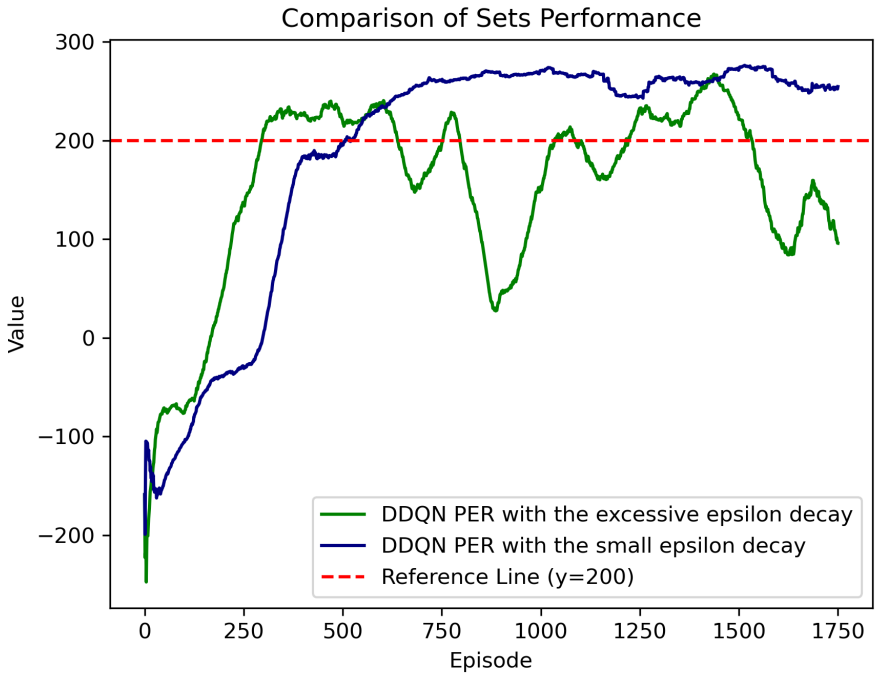

Figure 2: Learning curves for DDQN PER with different epsilon decay.

5 Discussion

All of the agents we implemented were able to solve the Lunar Lander environment as they have all achieved an average reward of over 200. In addition, we have presented a comparison using a standard DQN agent as a baseline to show the performance improvement using Prioritised Experience Replay and the Dueling architecture.

We found that when solving a not-so-complex game environment like Lunar Lander, the DQN PER agent successfully solved the environment quicker than other agents. DQN achieves a quick rise to 200 points primarily due to its strategy of making rapid, aggressive estimations on the best moves, leveraging early, bold guesses that sometimes overestimate action values. This approach initially accelerates learning but can lead to unsustainable strategies as the agent may not maintain accuracy with these high estimations over time. The inherent instability in DQN becomes apparent as it faces new challenges in later games, such as getting unstable in later training episodes. The early, potentially inaccurate estimations lead to a fragile strategy that can crumble under unexpected conditions.

In contrast, Dueling DQN and DDQN demonstrate a steadier learning process and performance. Their structured approach to evaluating the advantages of actions and the overall state values provides a more balanced learning trajectory. However, in a relatively simple environment like Lunar Lander, the sophisticated capabilities of the Dueling architecture do not manifest a distinct advantage over DDQN. This similarity in performance suggests that the enhanced features of the Dueling architecture, designed to finely distinguish between state values and the advantages of particular actions, might not be fully leveraged since Wang et al. (2016) suggested that this architecture is more advantageous than the deep Q-network architecture in case the number of actions is large.

While DQN PER outscores other models in early achievement within simpler environments, its susceptibility to instability underlines the importance of continued exploration and adaptive learning

strategies. However, we did manage to tune the DDQN PER with excessive epsilon decay to allow the agent to solve the environment more quickly than all other models. The consistent performance of Dueling DQN and DDQN highlights their potential in more complex scenarios, where their nuanced understanding of action and state values could provide significant benefits.

6 Future Work

Rainbow DQN

We found the algorithm integrating six advanced modifications into the original Deep Q Network algorithm. This algorithm is called Rainbow DQN and was introduced by Hessel et al. (2017). Since the architecture involves all the important enhancements we have tried and proved to be efficient, this is expected to improve our model. It reduces the overestimation bias by using a Double DQN and prioritises experience replay so that the agent learns much more from limited data. Not only does it use a DDQN, but it also enhances state value learning independently of action taken by the Dueling network on top of it.

While the original DQN updates the Q-values with the next immediate reward, Multi-step Learning in Rainbow DQN uses returns from n-step transitions, allowing for a more informed target and faster convergence.

Distributional DQN models the distribution of possible returns rather than trying to approximate the average total return, as the traditional DQN does (Niruti et al., 2023). This is typically achieved by predicting categorical outcomes for the return, modelling the distribution, and rendering a much richer signal in training.

Finally, Noisy Nets replace some or all of the linear layers in your network with noisy layers. These layers add parametric noise to their weights, which makes the policy's exploration pattern more sophisticated and adaptive based on the network's learning state.

7 Personal Experience

The Lunar Lander environment provides some challenges along with lots of experiences with a mixture of different algorithms, which have equipped us with knowledge and potential improvements of the models. Having the free choice of different algorithms to explore and solve the environment, we enjoyed the time tackling the challenges we faced when we implemented each algorithm. However, one of the problems we faced during the project was that training each agent takes a significant amount of time. This means that whether it is a minor modification to the hyperparameters for each agent or some change in the architecture, it will result in a lot of waiting time. In our cases, trying to fine-tune all the agents is not an easy task to do. Furthermore, setting up the Lunar Lander environment on our local environment takes quite an effort, as we experience some errors when installing all the required packages and dependencies for the environment to work properly.

On the other hand, we are proud that we can implement many algorithms and compare each performance to a fair standard. Surprisingly, it is not always the case that the most complex algorithm can solve particular problems better or faster. Overall, all of our team members could attend every meeting and finish their part of the work on time, making efficient and clear communication. In addition, we were able to not only finish individual parts of the work, but our team members were also able to help out each other's work. We gained valuable communication and teamwork skills from the project and insights into many state-of-the-art algorithms and their potential to solve reinforcement learning problems.

References

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017. URL https://arxiv.org/abs/1710.02298.

Oleg Klimov. Gymnasium documentation, 2023. URL https://gymnasium.farama.org/environments/box2d/lunar_lander.html.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992. URL http://www.incompleteideas.net/lin-92.pdf.

Ruishan Liu and James Zou. The effects of memory replay in reinforcement learning. In *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 478–485, Monticello, IL, USA, October 2018. IEEE. URL https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8636075.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. URL https://www.nature.com/articles/nature14236.

Dhoble Niruti, Gu Jianchen, Gunda Indhu, Kudari Shreyas, Lee Isaac, Xu Sophia, and Yang Kory. Lunar lander - deep reinforcement learning, noise robustness, and quantization, 2023. URL https://xusophia.github.io/DataSciFinalProj/. Accessed: 2023-05-07.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations (ICLR)*. Google DeepMind, 2016. URL https://arxiv.org/abs/1511.05952.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Association for the Advancement of Artificial Intelligence*, 2016. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/viewPaper/12389.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2016. URL https://arxiv.org/abs/1511.06581.

Appendices

Appendix A: Links to video presentation and agent performance

Click here to access the video presentation
Click here to access the agent performance
Click here to access the code

Appendix B: Hyperparameters for all agents trained

Table 2: List of hyperparameters and values

Hyperparameters	**Value**
Batch Size	128
Replay Buffer Size	200000
Learning Rate	0.0001
Episode Number	2000
Target Update	4
γ	0.99
ϵ_{start}	1.0
ϵ_{final}	0.01
ϵ_{decay}	200000
excessive ϵ_{decay}	30000
α	0.8
β	0.5