Implementation of Gym Lunar Lander using DQN and its enhancements

1 Problem Definition

This project seeks to address the Lunar Lander environment using the OpenAI Gym framework. The agent controls the lunar lander in the environment with the goal being to land it between the two flag posts on the surface. This goal can be achieved by controlling three engines (Left, Right and Central Engine) of the lunar lander. The difficulty levels can be altered by controlling the gravity and wind levels of the environment. The Lander starts at top centre with a randomly generated force, and environment gets terminated if the ‘Lander has crashed’, ‘Lander has left the screen’, ‘Lander lands safely’ . The observational space of the lunar lander environment features a continuous state space returning 8-dimensional vector- x and y coordinates, linear velocities in x and y directions, Angle and Angular Velocity, Leg Booleans for left and right touching the ground. The Action Space involves, four discrete actions- Do Nothing and the operations of the three individual engines.

1.1 Reward function

Penalties

- Moving Away from the centre
- -100 for crashing
- -0.03 for running engines.

Rewards

- +10 for leg contacting ground.
- +100 for landing successfully
- +200 for landing on the pad

2 Background

Deep Learning has transformed various fields by allowing computers to learn directly from the data being provided, especially in the tasks involving perception, classification, and prediction. One of the most fascinating applications of Deep Learning lies in Reinforcement Learning, where in an agent interacts with the environment to achieve certain goals heuristically [3]. Reinforcement Learning has shown remarkable success in a plethora of domains including game-playing, robotics, finance, and healthcare.

The integration of Deep Learning techniques in the advancements of Reinforcement learning led to the emergence of Deep-Q Networks (DQN). DQN is a deep neural network model that combines Q-Learning with deep learning architectures. This allows DQN to learn directly from high dimensional sensory input, making it suitable for tasks involving raw visual data, such as playing Atari Games.

Deep learning has been instrumental in enhancing RL algorithms to handle complex tasks with its ability to automatically learn hierarchical representations from raw data. In Reinforcement Learning, an Agent interacts with the environment by taking actions and receiving feedback in the form of rewards. The main objective of an agent is to learn a policy that maximizes the total rewards over time [2].

Generally, Reinforcement Learning algorithm faces challenges in scaling to high-dimensional input spaces or large action spaces. Deep learning helps curb these challenges by providing a framework to efficiently learn the representations from raw sensory inputs. This integration enables RL algorithms to handle complex tasks with high dimensional observations, such as images or sensory data.

2.1 Deep Q-Networks (DQN)

Deep Q-Networks was introduced by DeepMind in 2013 [3]., and it marked a significant milestone in Reinforcement Learning by demonstrating superhuman performance on a variety of Atari 2600 games. DQN employs convolutional neural networks to approximate the Q-Function, which estimates the expected future rewards for each action given a state. DQN learns to make optimal decisions in complex environments by iteratively updating the Q-values based on the observed transitions.

The key components of DQN include **Experience replay** and **Target Networks**

Each experience can contains a tuple Q(S, A, R, S'), wherein S is the current state, A refers to the action taken by the agent, R signifies the Reward earned in that state and S', the next state. However, a program learns from the information in the most recent experience tuple and hence is bound to **make bad decisions** or **get stuck in a loop**.

Experience Replay, or **Replay Buffer**, is a collection of all the experience tuples and acts as a large storage space for the experience tuples. As Jordan Torres defines it – 'The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay'. With the introduction of Experience Replay, the agent randomly selects a bunch of experience tuples from the buffer, this enables the agent to learn from its experience in different time steps and takes the action accordingly.

While training neural networks, its important to keep in mind its stability. The Bellman Equation provides us with the value of Q (S, A) via Q (S', A'). Both these states are just one step apart and hence becomes difficult for the neural network to distinguish between them. Bellman's equation is a concept of reinforcement learning, especially in dynamic programming and value based methods. It expresses the relationship between the state-action pair and the values of its consecutive state-action pairs.

$$Q^*(s,a) \quad = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a \right]$$

When we perform an update of our Neural Network Parameters to make Q (S, A) closer to the required and desired result, we can indirectly alter the value produced for Q(S', A') and the other nearby states making our training unstable. A **Target Network** is used to make the training more stable, in which we keep a copy of the neural network and use it for the Q (S', A') in the Bellmans' equation.

DQN's success in Atari games is a testament to its ability to learn efficient and effective strategies from raw pixel inputs. By treating each frame of the game as a state, DQN learns to predict Q-values for each action, guiding the agent towards maximizing its score. Through repeated interactions with the environment and updates to its parameters, DQN improves its performance and exhibiting super-human level play in many games.

2.2 DQN Architecture

A traditional DQN, also known as Vanilla DQN, has the following layers [4]

- **Input Layer:** At the bottom of the network, the input layer receives the information about the state from the environment. This input is the raw pixel values of the game screen.

- **Convolutional Layers:** These layers apply convolutional filters to the input, allowing the neural network to learn features from the visual data. Each filter extracts different visual patterns from the input image.
- **Activation Functions:** Activation functions like ReLU (Rectified Linear Units) are typically applied to introduce non-linearity in the network.
- **Flattening Layer:** This layer is responsible for reshaping the output into a single vector, and it serves as an input to the fully connected layers.
- **Fully Connected Layers:** These Layers connect every neuron in one layer to every neuron in the next layer, allowing high-level feature learning and abstraction.
- **Output Layer:** This layer produces the Q-Values, which represent the estimated quality of taking each possible action given the current state.

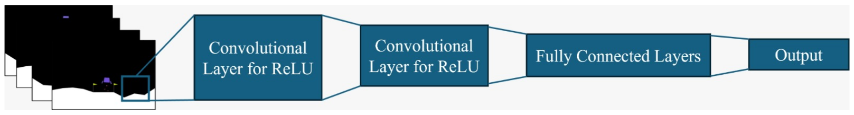

Figure 1: DQN Architecture for Lunar Lander

3 Method

3.1 Duelling DQN

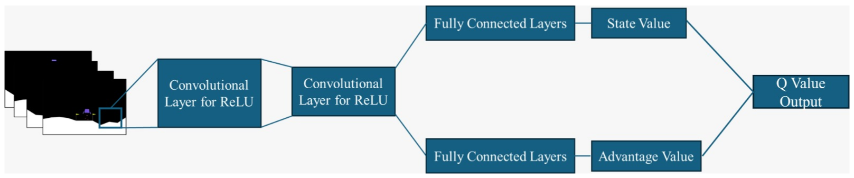

Figure 2: Duelling DQN Architecture

Duelling DQN is an enhancement of the Deep Q-Network (DQN) algorithm to improve the performance and efficiency in Reinforcement Learning tasks. The key feature of Duelling DQN is to separate **the estimation of the value of being in a state and the advantage of taking each action in that state**.

As mentioned in the previous section, In Traditional DQN architectures, the network directly outputs the Q-Values for each action for a state. However, an interesting feature of Dueling DQN is the introduction of two streams within the neural network, splitting this DQN process. These two streams are **the Value Stream** and **the Advantage Stream**.

The value stream estimated the value function V(S), representing the value of being in a particular state regardless of the action taken. While the Advantage stream estimates the Advantage Function A (S, A), representing the additional value of taking each action A in the given state S compared to the average action value in that state. It quantifies how much better or worse each action is relative to the others.

The final Q-Value for each action A in State S is obtained by combining the value and advantage streams using the following equation:

By splitting the value and advantage estimations, duelling DQN can learn more efficiently and generalize better across different actions. This architecture helps in situations where the value of states is stable but advantage of actions varies significantly.

3.2 DDQN and PER

Double DQN (DDQN) and Prioritized Experience Replay (PER) are two enhancements to the standard Deep Q-Network (DQN) algorithm, which is a popular reinforcement learning technique used for training agents to make decisions in environments with discrete action spaces.

Double DQN and Prioritized Experience Replay are two enhancements to the Deep Q Network Algorithm. The Core idea of DDQN is to address the overestimation bias inherent in the Q-Learning update by decoupling action selection from action valuation. Instead of using the same network to select and evaluate actions, DDQN utilizes two separate neural networks- A Policy Network and A Target Network. This results in more stable and accurate Q-Value estimates. Double DQN is similar to the traditional DQN algorithm, but contains two Q-functions .Each Q-function can utilize the value of the second Q-funtion to update the next state. DDQN first finds the actions corresponding to the highest Q-value in the current network, and then places the selected action with the maximum value in the target network to calculate the target Q value.

As mentioned in an earlier section, Experience replay stores Experience (S, A, R, S') tuples to aid an efficient learning for the agent. All the experience might not contribute to learning with the same importance, some experiences may be more relevant than the others. Prioritized Experience Replay (PER) addresses this issue by assigning priorities to the experiences based on their TD error (Temporal-Difference), which indicates the magnitude of the error in the Q-value prediction. Experiences with higher TD errors are given higher priority and are more likely to be sampled during training, allowing the agent to learn more effectively from the past experiences. By combining Double DQN and Prioritized Experience replay, greater improvements in the performance and the stability of Reinforcement Learning algorithms is achieved.

3.3 Distributional DQN

Distributional Network uses a neural network to approximate the distribution of returns for each state-action pair. The output of the network is a set of values that represent the parameters of the distribution. Instead of learning a scalar Q-value for each state-action pair, the distributional DQN learns a probability distribution over possible returns. This distribution is typically represented as a set of quantile values or a categorical distribution over a fixed number of bins. Distributional DQN can capture more information about the uncertainty and variability of the rewards, which can lead to better exploration and decision-making in complex environments.

3.4 NoisyNet DQN

Exploration is a critical aspect of Reinforcement Learning. The agents must discover and learn about the environment to make appropriate decisions. Without the aspect of exploration, agents would have suboptimal strategies and policies. Noisy DQN brings about a controlled stochasticity into parameters of neural networks which are commonly used as function approximators in Reinforcement Learning Agents. This stochasticity allows the agent to explore broader range of actions and states during training.

4 Implementation

4.1 Vanilla DQN

The Vanilla DQN method employs Q-learning to learn a function Q(s, a) that estimates the total rewards achievable from taking action a in state s and following the optimal policy. The optimal policy involves selection of an action that maximizes Q(s, a) for all states s. In this approach, a neural network approximates the Q-function by taking the environment states as input and outputting the Q-value for each action. The network returns a vector of Q-values with a distinct value for each action, given the input state.

To train the network, episodes are played in the environment, and transitions (state, action, reward, next state) are saved in a replay buffer. The weights of the network are updated by sampling batches of transitions from the replay buffer and minimizing the difference between predicted Q-values and target Q-values, which are obtained from the Bellman equation. The replay buffer stores transitions from the past, and the agent learns from them in a non-sequential manner, breaking down the correlation between consecutive transitions to enhance learning stability.

To ensure stable target values for Q-learning, a separate target network is employed, regularly updated with the weights of the primary network to prevent destabilization during training. The algorithm uses an epsilon-greedy exploration method, which enables the agent to make random moves and explore new states. As time progresses, the value of epsilon reduces, and the agent relies more on learned knowledge.

4.2 Double DQN

In Double DQN, two neural network models used are Policy Network and Target Network, wherein the Policy Network is the main network that is updates every iteration and is used for both selecting actions during training and gameplay. The target network weights are periodically updates with the weights from the Policy Network to calculate the target Q-values during the update step. This separation helps stabilize learning.

During each step of an episode, an action is chosen using an epsilon-greedy policy based on the policy network. The experiences are stores in a replay buffer. This buffer is a collection of tuples with data from past episodes. The use of a replay buffer helps in breaking the correlation between the consecutive samples, leading to a more stable and effective learning.

During Training, small batches of experiences are sampled randomly from the replay buffer. This random sampling helps prevent cycles in updates that can arise from the sequential nature of the episode data.

For Double Q-Leaning Update, For each sample of the small batch:

- Calculating the next state actions using the policy network. This step involves passing the next state actions using the policy network. This step involves passing the next states through the policy network and selecting the action with the highest Q-value.
- For the actions selected by the policy network the target network provides the estimated Q-values. These are used to compute the expected Q-value
- Compute the Target Q-Value

The Loss is calculated as the mean squared error between the current Q-values and the targets calculated in the previous step. To minimize this loss, backpropagation is used.

Periodically, the weights from the Policy Network are copied to the Target Network. This step is crucial to stabilize the targets during learning.

DDQN is an enhancement upon standard DQN by decoupling action selection from its evaluation, thus mitigating the overestimation issues of Deep Q-Network. While DQN uses the same network for both action selection and value evaluation, leading to potential overestimation biases, DDQN employs separate networks for these tasks. By selecting actions using one network and evaluating their values using another, DDQN reduces overestimations, resulting in more stable learning and better policy development. This modification enhances performance and stability in complex decision-making environments.

4.3 Double DQN with PER

Our solution combines Double Deep Q-Learning Networks and Prioritized Experience Replay to create an efficient reinforcement learning agent that can solve the LunarLander-v2 environment. The Double DQN introduces the policy network and target network structures to reduce the overestimation of Q-value prediction. We use PER to prioritize important experiences during training, which is done by using a priority queue of transitions based on their temporal difference error. The SumTree and PrioritizedReplayBuffer classes provide an efficient way to store and retrieve the prioritized transitions.

During optimization, a prioritized batch of transitions is sampled, and the importance sampling weights are offset for the bias. The training loop selects actions using an epsilon-greedy strategy and stores transitions with calculated priorities. The policy network is optimized periodically, and the target network is synchronized. Finally, the agent is evaluated by testing it for several episodes with rendering on at the end of training.

In a standard Double DQN, experiences are held in a replay buffer and sampled uniformly. This suggests the replay buffer may contain redundant or less informative transitions that may act to slow learning. By contrast, Double DQN with PER prioritizes experience on the basis of their TD error, with a view to sampling those transitions more likely to yield the most benefit to learning. It turns out to be faster learning, and it makes sure high-impact experiences are focused on. The procedure, however, introduces a sampling bias, and for that reason, importance-sampling weights are used during training to correct the loss function so that it becomes invariant to the frequency of sampling. The net outcome is a more efficient and focused process of training that allows the agent to learn faster and more effectively, mitigating the flaws of uniform experience sampling.

4.4 Duelling DQN with PER

Dueling DQN uses a novel neural network architecture to decouple the estimation of state values from the estimation of action advantages. This makes it easier for an agent to distinguish between the inherent value of a state and the relative benefit between taking different actions, allowing it to learn more stably.

During training, a batch of experiences with high priority is sampled from the replay buffer. The agent calculates target Q-values using the Double DQN approach, through which action selection and Q-value estimation are performed by different networks. Contrast of its estimates with targets gives loss of the network, and importance sampling weights counter prioritization bias, it updates the parameters of the network, and the priorities in the replay buffer based on new TD-errors. It will run training for a few episodes, where the agent acts according to an epsilon-greedy policy, and the target network synchronizes the policy network occasionally. By combining Dueling DQN and PER, the agent can focus on valuable experiences while keeping value estimation in good condition, leading to better learning of the policy.

The integration of two recently proposed techniques in reinforcement learning, Dueling Deep Q-Networks with Prioritized Experience Replay, can improve both the stability and learning efficiency of traditional DQN variants. Dueling DQN breaks down the conventional Q-function into an additive set of value and advantage functions in a forward manner. The value function determines the value of being in some state, and the advantage is the assessment of benefit for each action possible from being in that state. This allows further separation of the ideas in the network, differentiating between the general values of states and the advantages of the actions one might take. PER optimizes the experience replay mechanism by prioritizing experiences based on their temporal difference error, ensuring that the agent learns from transitions with the most learning potential. By doing so, it helps the agent separate the two concepts of state values and action advantages better. At the same time, it prioritizes critical experiences during training to improve the learning performance and stability over standard DQNs.

5 Results and Discussion

Cumulative Rewards per Episode:

Vanilla DQN: We saw a steady rise of improvement with less fluctuation towards the end. **DDQN:** Presents certain fluctuations but is consistently performing and also reaching to the optimal policy better than the Vanilla DQN

DDQN with PER: Similar oscillations to DDQN, although spikes are much more pronounced. This indicates that the agent was able to learn from more critical experiences.

Dueling DQN with PER: Also, this shows some variance in the difference and provides a little bit more of a fair reward with a few spikes here and there, suggesting that the integration of the dueling architecture with PER may increase the ability of the agent to evaluate state.

Mean Average Loss Per Episode:

Vanilla DQN: Has higher initial losses that go down slowly, which indicates slower convergence.

DDQN: The second network reduces the overestimation error.

DDQN with PER: Has large initial decreases and then stabilizes to be very low, which shows the system is learning effectively from the prioritized experiences. Dueling DQN with PER: Similar to the DDQN with PER, but stabilizes faster. Thereby, dueling architecture potentially gives more stable value estimation.

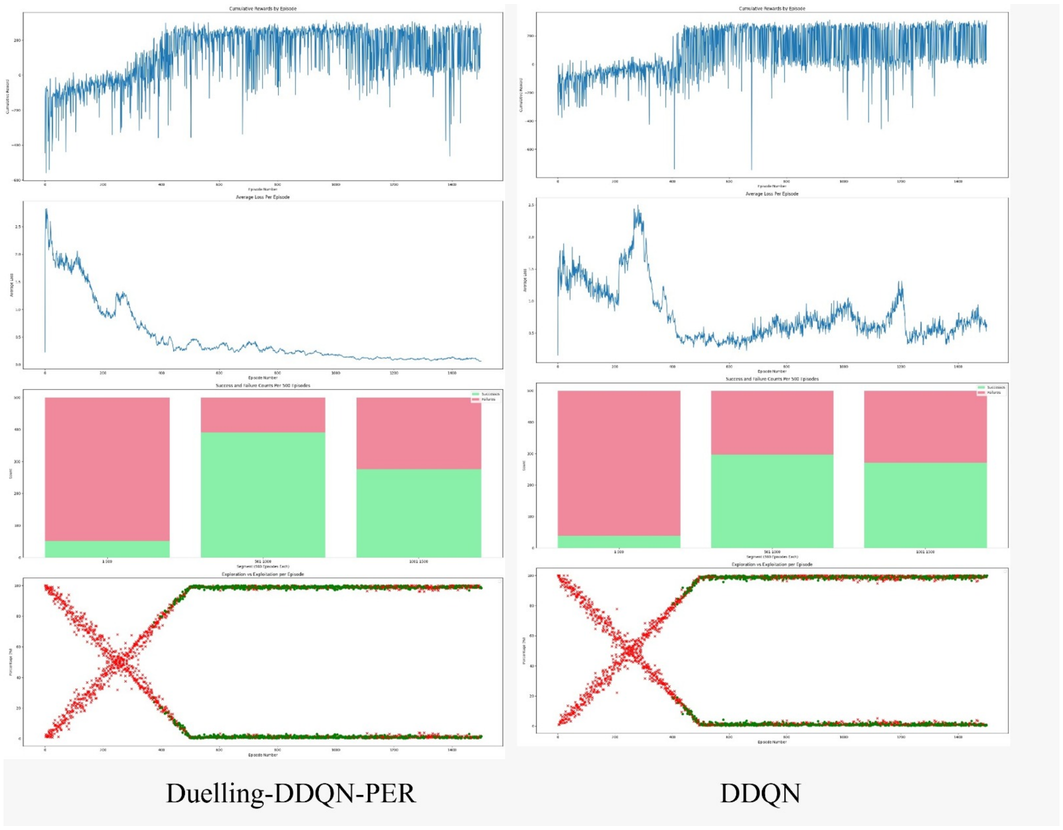

Figure 3: Graphical Representation of Duelling DDQN-PER and DDQN

Success and Failure Number per Segment of Episodes

Vanilla DQN: Shows some increase in average success rate over time, but still keeps failing a lot in the later episodes.

DDQN: More successes are shown for later parts, and the learning efficiency is better.

DDQN with PER: The success rate is rather enhanced in the latter half, which clearly shows the impact of learning crucial experiences.

Dueling DQN with PER: High success rates were reached quite early, due to the more effective state value and action advantage estimation.

Exploration vs. Exploitation In general, trends from both graphs point to a reduction in exploration (red crosses) and an increase in exploitation (green dots) as training proceeds. Dueling DQN with PER gives a balanced situation regarding exploitation and exploration, since it rapidly shifts into exploitation. This might mean a higher level of confidence in deciding.

General Comparison:

Stability and Convergence: The Dueling DQN with PER appears to have the most stable and efficient learning curve while also learning the optimal policy fastest, it also has considerably lower and stable loss rates, with relatively higher success rates. **Learning from Critical Experiences:** In contrast, the PER integrated models—that is, DDQN with PER and Dueling DQN with PER—had noticeably higher returns, suggesting sensitivity in benefit for prioritized replay with the increased criticality of past experiences.

Policy Efficiency: DDQN is clearly better than Vanilla DQN at dealing with overestimations of the action values; Dueling DQN shows the best overall policy by which is due to the state values and action advantages getting estimated separately It is also worth noticing that it starts to overfit toward the end of training.

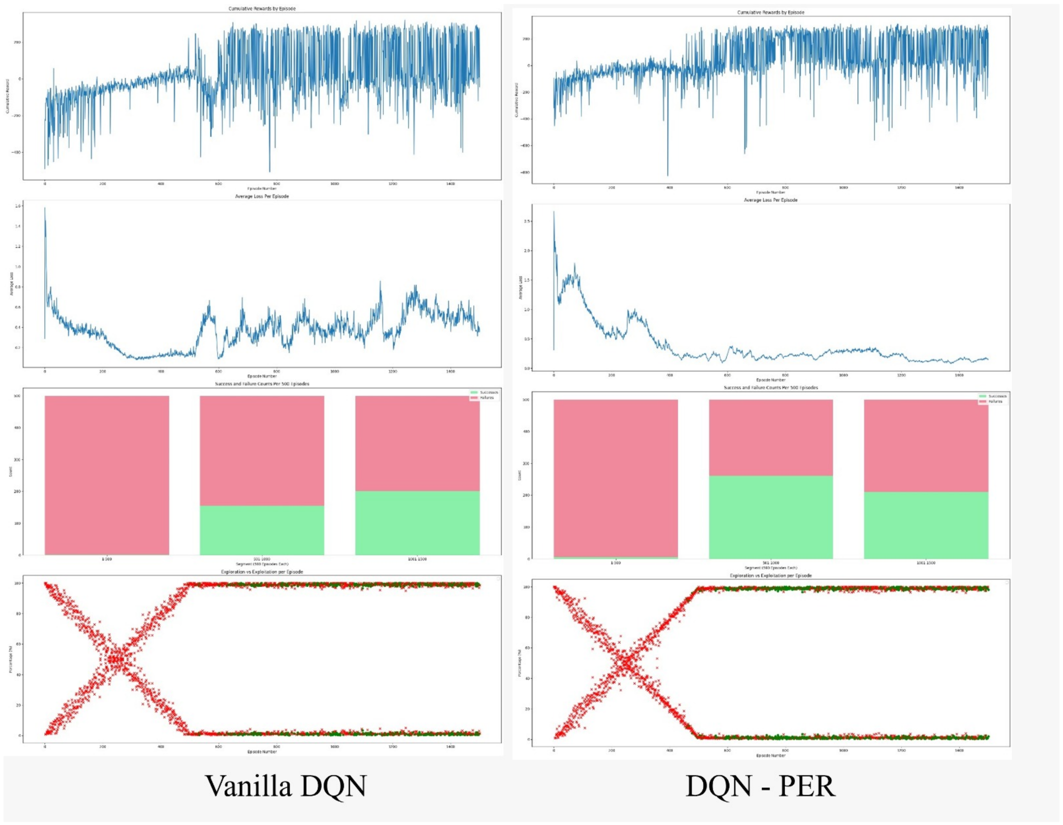

Figure 4: Graphical Representation of Vanilla DQN and DQN - PER

Various algorithms were implemented to optimize the gameplay of Lunar Lander in order to maximize the rewards. Although some algorithms proved successful and maximized the reward, a few other algorithms proved unsuccessful.

While implementing Distributional DQN, the following was noted- The network architecture plays a crucial role in capturing the complex dynamics of the Lunar Lander environment. If the neural network architecture, including the number of hidden layers, units per layer and activation functions are not appropriately designed, the model may struggle to learn an effective policy. Several different combinations of hidden layers were tried and the best result achieved was for 3 hidden layers of 64 nodes. It was also noted that increasing the nodes resulted in a decrease in total rewards.

Secondly, Hyperparameter tuning is also an essential factor for optimizing the training process. Parameters such as learning rate (LR), discount factor, epsilon decay, and target network update rate (TAU) needs to be carefully adjusted to ensure efficiency in the learning process. Despite attempting several different values for LR and TAU, the maximum reward received was -50.

Gradient Clipping can be a crucial technique to prevent exploding gradients during training. By applying gradient clipping to the gradients during backpropagation the magnitude of the gradients can be limited, preventing them from becoming too large, and in turn destabilizing the training process. However, the rewards were not maximized upon implementing Noisy DQN either.

Several Factors could have contributed to the poor suboptimal performance of Noisy DQN in its implementation in the Lunar Lander Environment. The injection of noise into the networks' parameters leads to heightened exploration during the initial stages of training. While exploration is crucial for discovering effective policies, excessive exploration might result in suboptimal actions and negative rewards. As a result, the agent may struggle to learn a successful policy, leading to fluctuating or even declining rewards during training.

The parameters of Noisy DQN, which introduces stochastic noise into the network during training, may not have been appropriately tuned. If the noise parameters are set too high or too low, it can

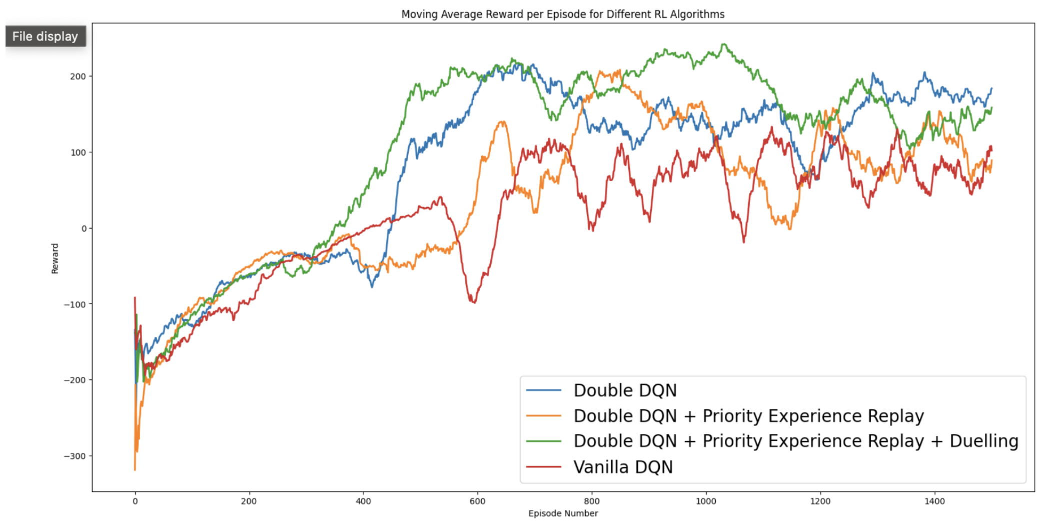

Figure 5: Graphical comparison of all implemented algorithms

disrupt the learning process by either exploring or exploiting the environment excessively. Despite attempts at several combinations of noise parameters, the agent continued to explore excessively.

The convergence properties of Noisy DQN differs from those of traditional DQN algorithms. The stochastic nature of the parameter updates in the Noisy DQN can introduce additional variability into the learning process potentially prolonging the time it takes for the agent to converge to an optimal policy. Consequently, fluctuations in rewards and slower convergence may be expected during the training compared to conventional DQN approaches.

N- Step DQN was implemented in the environment for different values of N. While implementation, The values of N were chosen to be 3, 7 and 10. Contrary to our expectations of receiving better results and higher rewards than Vanilla DQN, N-Step DQN proved to show highly fluctuating rewards and sub-optimal policies. The improvements would have been possible by iteratively trying different combinations of factor, and prioritization exponent and also by increasing the value of N-steps upto 20 or more.

The delay in receiving positive rewards can make it challenging for the agent to estimate the long-term value of actions using n-step returns accurately, leading to slower convergence and potentially suboptimal policies which might have been the reason for the poor performance of the algorithm and training the agent due to incorrect combinations of hyperparameters.

6 Future Work

While the current implementation of Deep Q-Network and its variants have shown positive results in the Lunar Lander Environment, there is still a significant room for improvement by using advanced reinforcement learning techniques. Using Proximal Policy Optimization in the environment would demonstrate reliable performance and better exploration capabilities in comparison to other policy gradient methods. This could potentially improve the sample efficiency, exploration, and the overall performance in tackling the complexities of the Lunar Lander task [5].

Combination of various DQN enhancements like Double DQN, Prioritized Experience Replay, Duelling DQN and Distributional DQN into a single Rainbow DQN algorithm has collaborative effects of these techniques could lead to improved stability, efficiency, and the overall performance [1].

Hierarchical Reinforcement Learning approach could be used to enable the agent to learn and breakdown the complex lunar lander tasks into subtasks, exhibiting better generalization and improved performance in complex scenarios [6].

7 Personal Experience

Our team learned many valuable lessons during our exploration of various Reinforcement Learning (RL) algorithms while working with Lunar Lander simulation. We discovered that traditional tabular methods are not effective in handling complex, high-dimensional environments because state spaces in realistic scenarios, like Lunar Lander, are too large for tabular methods to manage. We realized that approximation methods like Deep Q-Networks (DQN) are essential in such cases.

Our hands-on experiments showed us the importance of more sophisticated techniques like Double DQN, Priority Experience Replay, and Dueling architectures. These modifications improved learning stability and significantly boosted the performance of our agents, as seen in our collected data.

Our journey through different gym environments, which included simpler ones like Cart Pole and more complex settings like Acrobot and Lunar Lander, helped us appreciate the challenges of controlling and balancing in dynamically changing conditions - a crucial aspect of robotics. This practical insight sparked our interest in the potential applications of RL in robotics. We envision these algorithms leading to advancements in autonomous robots that can learn and adapt with unprecedented efficiency.

In conclusion, our experience with implementing RL has been transformative. It has provided us with a deeper understanding of its theoretical underpinnings and a clear vision of its practical applications. The intersection of RL with fields like robotics and trading is fascinating, and it promises a frontier brimming with opportunities for innovation and advancement.

References

[1] 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. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[2] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[3] 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.

[4] CloudThat Resources. Deep reinforcement learning algorithm : Deep q-networks, October 05, 2023.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[6] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *International conference on machine learning*, pages 3540–3549. PMLR, 2017.