LunarLander Implementation Concerning DQN PPO and Deep SARSA

Haichuan Wei, Aobo Sun, Cheng Han, Tianyi Hu, Huatai Wei

***Abstract*—This project applies deep reinforcement learning (DRL) to solve the LunarLander-v2 problem in OpenAI Gym. We implement and compare three algorithms: Deep Q-Networks (DQN), Deep SARSA, and Proximal Policy Optimization (PPO). Hyperparameters such as learning rate, exploration rate, and epochs are tuned to optimize performance.**

I. Introduction

Reinforcement Learning (RL) is highly effective for complex decision-making in high-dimensional, dynamic environments. In OpenAI's LunarLander-v2, the agent must navigate an 8-dimensional state space—including continuous variables (position, velocity, angle) and discrete ones (leg contact states)—using four discrete actions. The objective is to achieve an average score of 200 points by safely landing on a target pad.

This state-action space, despite discretization, is too extensive for traditional Q-learning, which relies on maintaining and updating a large Q table for every state-action pair. In high-dimensional or continuous spaces, this becomes impractical. Thus, we turn to **Deep Reinforcement Learning (DRL)** to approximate the state-action value function (Q-value) or policy using neural networks. This paper explores training an agent using **DQN, Deep SARSA, and PPO** methods to address the challenges of LunarLander.

II. Implementation Through Different Algorithms

A. Implementation using Deep Q-Network

1) DQN: The reason that we want to try with DQN algorithm is pretty straightforward——Similar to traditional Q-Learning, which is also a **value-based** algorithm, DQN determines strategies by estimating the Q-function and Q-value (the value of the state-action pair) through a neural network. It belongs to the **off-policy** method, which means that its update does not depend on the current strategy, but can be updated using old data. In terms of neural network selection, DQN uses a policy network and a target network to predict and calculate the Q value respectively to stabilize the training process. It also introduces an experience replay mechanism, which improves data utilization and reduces the correlation between samples by storing past experience in a buffer and randomly sampling from it during training.

2) Dueling DQN: Similar to traditional DQN, Dueling DQN is a value-based algorithm, where strategies are determined by estimating the Q-function. However, it introduces a novel architecture to estimate the value of states and actions more effectively. Dueling DQN modifies the Q-function by separating the value function $V(s)$ and the advantage function $A(s, a)$, which leads to a more efficient learning process.

Dueling DQN uses a shared feature extractor for both state value and advantage estimation, and then combines them to produce the final Q-values using the following formula:

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s, a') \tag{1}$$

In this formula, $V(s)$ denotes the value of the state s, which represents the expected return for being in that state. The term $A(s, a)$ refers to the advantage of taking action a in state s, indicating how much better or worse this action is compared to other possible actions in the same state. Additionally, $\frac{1}{|\mathcal{A}|}\sum_{a'} A(s, a')$ represents the mean advantage over all actions, ensuring that the Q-value is centered around the value of the state.

3) Double DQN: Like traditional DQN, Double DQN is also a value-based and off-policy algorithm. However, it addresses a critical issue in DQN: Q-value overestimation. In traditional DQN, the network sometimes overestimates Q-values, especially when the maximum Q-values for the next states are wrongly chosen by the same network that is being updated. Double DQN reduces this overestimation by using two separate networks: one to select the best action and the other to evaluate its value.

The key update rule for Double DQN is as follows:

$$Q_{\text{target}}(s, a) = r + \gamma \cdot Q_{\text{next}}(s', \arg\max_{a'} Q_{\text{eval}}(s', a')) \tag{2}$$

Here, $Q_{\text{target}}(s, a)$ represents the target Q-value, where r is the immediate reward and γ is the discount factor. The term $Q_{\text{next}}(s', \cdot)$ refers to the target network's Q-value for the next state s', while $Q_{\text{eval}}(s', a')$ denotes the evaluation network's Q-value for s'. Finally, $\arg\max_{a'} Q_{\text{eval}}(s', a')$ identifies the action that maximizes the Q-value in the evaluation network for the next state.

Both Dueling DQN and Double DQN build upon the foundational DQN algorithm but improve it by addressing different challenges: Dueling DQN provides a more efficient way of evaluating states and actions by separating state value and action advantages. Double DQN mitigates Q-value overestimation by decoupling action selection and evaluation between two networks.

These improvements contribute to more stable and effective learning in complex environments.

B. *Implementation using Deep SARSA*

Deep SARSA is a reinforcement learning method that combines the SARSA algorithm, a policy-based reinforcement learning approach, with deep neural networks. [1] It primarily addresses complex problems with continuous state spaces by estimating state-action values and constructing an optimal policy for a given agent. The SARSA algorithm is a temporal-difference (TD)-based reinforcement learning method that estimates the expected return for the next step based on the current state-action pair. In contrast to Q-learning, SARSA is an on-policy algorithm that updates the Q-values based on the actions and rewards chosen by the current policy. This makes the policy more stable, preventing the issue of delayed updates to target Q-values. SARSA is particularly effective in handling uncertainty and environmental changes, especially in tasks with high exploration requirements.

The update formula for the SARSA algorithm is as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \tag{3}$$

In contrast, the update formula for Deep SARSA is as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}; \theta) - Q(s_t, a_t; \theta) \right) \tag{4}$$

As observed, Deep SARSA extends SARSA by using deep neural networks to approximate the action-value function (Q-values), resulting in the Q-network. Deep SARSA is well-suited to handle more complex environments, particularly those involving continuous and high-dimensional state spaces. For example, in the LunarLander project, the agent must make decisions across multiple stages to safely land the spacecraft, which involves continuous states such as position, velocity, and rotation angles, along with a high-dimensional state space.

C. *Implementation using Proximal Policy Optimization*

The Proximal Policy Optimization method was proposed by OpenAI in 2017. [2] Unlike the previously mentioned methods such as DQN, Proximal Policy Optimization is an on-policy update method based on the gradient of the policy, which means that the agent used to interact with the environment is the same, and each time the parameters are updated, sampling needs to be performed again. Compared with other algorithms, the advantage of PPO is that it limits the update range of the policy by introducing the clip operation, which avoids excessive policy fluctuations and significantly improves training stability and sample utilization. Specifically, the loss function to be optimized by the PPO algorithm is as follows:

$$L(\theta) = \mathbb{E}\left[\min\left(r(\theta) \cdot \hat{A}, \text{clip}\left(r(\theta), 1-\epsilon, 1+\epsilon \right) \cdot \hat{A} \right) \right] \tag{5}$$

The objective function is characterized by its ability to control changes in the corresponding policy based on the policy ratio shown below:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)} \tag{6}$$

And then to limit the magnitude of policy updates using the $\text{clip}(\cdot)$ method to avoid excessive policy shifts. In addition, a generalized advantage function (GAE) $\hat{A}$ is used to replace the traditional reward function to more accurately evaluate the actual effect of each action during the update [3], thereby improving the stability of learning. Specifically, the GAE can be calculated using the following formula:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{7}$$

Therefore, in the specific training practice of this LunarLander environment, since the PPO method is also implemented based on the Actor-Critic framework, we will define two computational networks in the specific implementation, and define the loss function with the clip method, as well as the calculation with the GAE advantage function to implement the corresponding PPO algorithm, and verify the ability of the algorithm through training and plotting.

III. Exprimental Results

A. *DQN*

1) General Training Situation: In this experiment, we compared four reinforcement learning algorithms: DQN, Double DQN, Dueling DQN, and Double-Dueling DQN. We focused on two aspects of performance: the training episodes required to reach an average reward of 200 and the episodes when epsilon converges to 0.

The image presents the reward for each training episode across four reinforcement learning methods. The methods compared include DQN, Double DQN, Dueling DQN, and Double-Dueling DQN. The plot tracks how the reward changes over the course of training for each method, as shown in Figure 1.

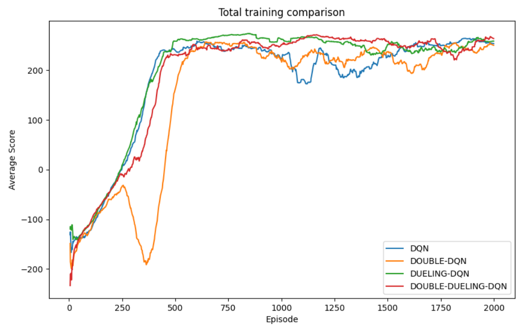
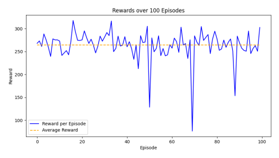
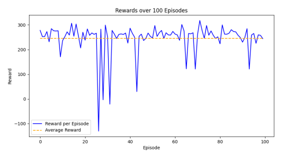


Fig. 1: Total training comparison chart

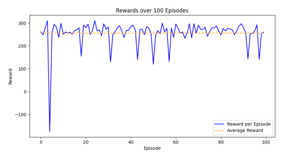
The table below compares the performance of four reinforcement learning methods: DQN, Double DQN, Dueling DQN, and Double-Dueling DQN. It shows the number of episodes required for each method to achieve two important milestones: (1) an average reward of 200 and (2) the



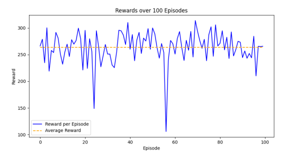
(a) Subfigure 1 DQN



(b) Subfigure 2 Double DQN



(c) Subfigure 3 Dueling DQN



(d) Subfigure 4 Double Dueling DQN

Fig. 2: 100 consecutive episodes tests

convergence of epsilon to approximately 0, which indicates the transition from exploration to exploitation in the training process.

TABLE I: Episode corresponding to reward ≥ 200 and epsilon ≈ 0 for DQN variants

	Reward ≥ 200	Epsilon ≈ 0
DQN	398	920
Double DQN	523	915
Dueling DQN	413	917
Double-Dueling DQN	432	920

In the performance of different algorithms, due to the lack of additional structures, DQN can quickly find effective strategies at the early stage of training and reach the reward value of 200 at the earliest. In contrast, Double DQN introduces the target network to reduce overestimation bias of Q values, which increases stability, but is slower to train, requiring more episodes to reach the same reward value. Dueling DQN and Double-Dueling DQN further improve stability by separating value and advantage functions, and are suitable for handling high-dimensional complex environments, although their convergence rate is slow. Especially in Dueling DQN, the curve convergence is smoother and the fluctuation is less, because its structural optimization makes the model pay more attention to the value of the learning state, reduce the dependence on specific action selection, and thus be more stable in complex environment, avoiding excessive fluctuation of Q value estimation.

2) *General Test Situation:* In Figure 2, the four images present the reward per episode over 100 consecutive episodes using the trained agent across four different reinforcement learning methods: DQN, Double DQN, Dueling DQN, and Double-Dueling DQN. These plots illustrate the performance of each method over time, showing the fluctuations and trends in reward as the agent progresses through the training episodes.

we observe that the average reward achieved by the agents trained with the four methods—DQN, Double DQN, Dueling DQN, and Double-Dueling DQN—hovers around 260. This consistency indicates similar levels of performance across the methods in terms of the reward metric.

3) *Hyperparameter regulation:* Figure 3 shows the average reward performance of Dueling DQN algorithm under different epsilon values (i.e., exploration rate), in order to observe the influence of different epsilon values on the model training process. We conducted the experiment under the following conditions: epsilon = 0.9 (training to 1500 episodes), epsilon = 0.8 (training to 900 episodes), and epsilon = 0.99 (training to 1050 episodes). With these Settings, we are able to analyze the effects of different exploration-exploitation strategies on convergence rates and reward fluctuations.

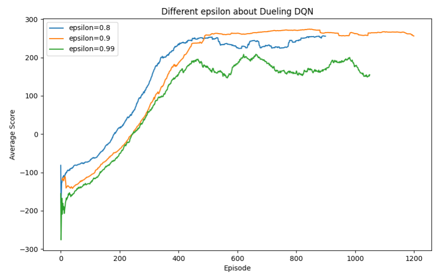


Fig. 3: Different epsilon about Dueling DQN

It can be seen from the figure 4 that the size of the epsilon value has a significant impact on the convergence speed and stability of the model. When epsilon = 0.9, the model rapidly converges around 300 to 400 episodes, and the reward value fluctuates little, showing good stability and balanced exploration-utilization effect. epsilon = 0.8 is more stable, but the convergence rate is slower, and the final reward value is slightly lower, indicating that the reduction of exploration leads to the reduction of training efficiency. However, the setting of epsilon = 0.99 causes the model to fluctuate greatly during the whole training process, indicating that too high exploration rate interferes with the stability and it is difficult to achieve effective convergence.

B. *Deep SARSA*

1) *General Training Situation:* In our experiments with the Deep SARSA algorithm, we focused on two key metrics: the number of training episodes required to reach an average reward of 200 and the sustained average reward after extended training. The following figure tracks the progression of Deep SARSA's average reward over the course of training.

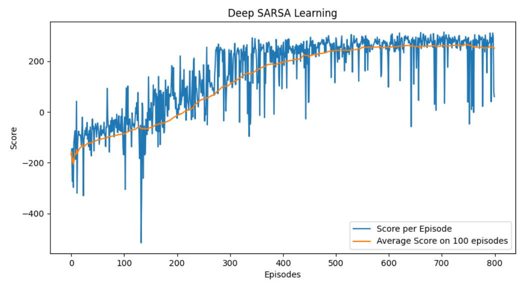


Fig. 4: Score after 800 Training Episodes with Deep SARSA

Through iterative tuning of Deep SARSA's hyperparameters, including epsilon decay rate and learning rate, we observed significant advantages in training efficiency and convergence speed. Specifically, after parameter optimization, the model was able to reach an average reward of 200 within approximately 400 episodes, demonstrating effective control and task performance in the LunarLander environment. Continuing training to around 800 episodes led to further improvement, with the average reward stabilizing around 270, indicating that the model achieved consistent control at a high level while handling the complex and dynamic state space intrinsic to the LunarLander task. This outcome shows that the optimized Deep SARSA algorithm exhibits strong adaptability and robustness, making it well-suited for high-dimensional continuous control tasks.

2) General Test Situation: After the training, a landing test was also performed 100 times in a row using the saved weight file after training. The corresponding results of the agent's score for each landing and the average score are shown below:

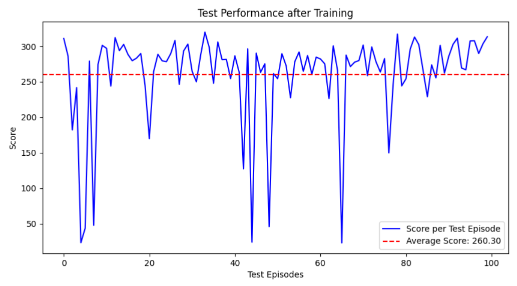


Fig. 5: Score of 100 Consecutive Episodes Tests

Despite obtaining lower scores several times, it can be seen that the vast majority of the scores are above 200, with an average score of around 260, which fully demonstrates the good performance of the training model obtained using this algorithm.

3) Hyperparameter Regulation: Learning rate is a key factor affecting model training. We chose three different learning rates to test the model: Learning rate = 0.01, 0.001, and 0.0001, to understand how learning rate affects convergence speed, stability, and score. The average score result is shown in the following figure:

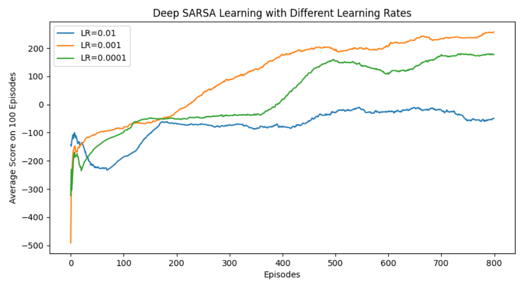


Fig. 6: Different Learning Rate about Deep SARSA

From the graph, it can be seen that different learning rates have a significant impact on the performance of the Deep SARSA algorithm in the LunarLander task. When the learning rate is 0.01, the model initially converges the fastest, but due to the large update amplitude each time, the strategy is unstable and the average score is low, ultimately hovering around -100. When the learning rate is 0.0001, the model update pace is conservative. Although the convergence is stable, due to slow adjustment, the average score after 800 rounds is only around 100, which fails to reach the optimal level. When the learning rate is 0.001, a balance is found between speed and stability, which is clearly the best performance among all experiments. The learning rate affects the convergence speed and final stability of the model by determining the stride size of each parameter update. Therefore, a moderate learning rate ensures both the stability of policy optimization and convergence speed, which can effectively help the model achieve higher average scores.

C. Proximal Policy Optimization

1) General Training Situation: Firstly to be shown is a graph showing the average score obtained during the training process of the PPO algorithm will be displayed. In the figure below, the abscissa is the episode used for training, and the ordinate is the average score obtained during the current training. The following figure shows the training results after training with the PPO algorithm:

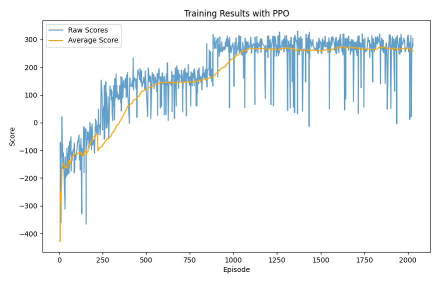


Fig. 7: Training Results with PPO

The overall training results show that the PPO algorithm excels in convergence speed and stability. It quickly increases the average score in the early stages, showcasing strong exploratory ability, and after reaching the target score of around 200, it maintains minimal score fluctuations, indicating good stability. Specifically, by approximately 500 iterations, the average score reaches the 200 mark, demonstrating PPO's capability to rapidly identify effective strategies. As training continues, the model's score stabilizes around 200, confirming that PPO's "clipping" mechanism effectively prevents excessive updates, keeping policy adjustments within a reasonable range. By around 700 iterations, the score volatility significantly decreases, consistently staying above the target. Compared to other policy optimization algorithms, PPO's fast convergence and stability make it especially suitable for tasks requiring high score thresholds, while also offering generalizability and scalability.

2) General Test Situation: After the training, a landing test was also performed 100 times in a row using the saved weight file after training. The corresponding results of the agent's score for each landing and the average score are shown below:

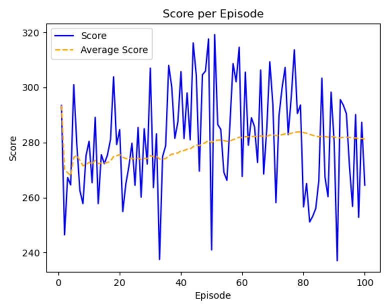


Fig. 8: PPO Algorithm Test Score Result

From the above test results, it can be seen that despite individual failed episodes, the score of the trained agent can be maintained above 200 in the vast majority of episodes, with an average score of 280, which reflects the effectiveness and excellence of the PPO algorithm trained in the LunarLander environment.

3) Hyperparameter Regulation——Clipping Rate: Due to space constraints, the study of PPO hyperparameter adjustment has chosen to display the most representative clip rate indicator. In the following result graph, blue indicates the result when the clip value is 0.20, orange indicates the result when it is 0.05, and green indicates the result when it is 0.35. As can be seen from the result graph, when the clip value is low, the agent learns more slowly but the overall change is more stable, which reflects the fact that a smaller clip value limits the amplitude of policy updates and the training process is more stable. But an extremely low clip rate might also indicates that the algorithm would easily been trapped in local optimum. In this figure, the training process can't even be converged when clip rate is 0.05. When the clip value is high, the agent learns faster , however the learning process is more volatile in the first few episodes. But higher clip rate can solve the problem and eventually get converged at about 1000 episodes.

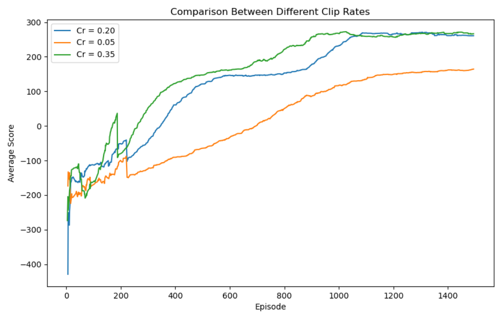


Fig. 9: Comparison Between Different Clipping Rates

IV. Conclusion

In this project, we explored multiple reinforcement learning algorithms for solving the LunarLander problem. The chosen methods—DQN, Deep SARSA, and PPO—each align with specific aspects of the task. DQN is particularly suited to LunarLander due to its ability to efficiently apply Q-learning in a discrete action space, making it well-suited for environments where identifying optimal policies based on a finite set of actions is critical. Deep SARSA was selected for its sensitivity to the current policy's risk, enabling a balance between exploration and exploitation that can adaptively optimize performance in stochastic environments. PPO emerged as a strong candidate due to its stable policy optimization and constrained update steps, which allow for efficient learning with reliable convergence, making it highly effective in enhancing both training stability and efficiency.

Our choice of these algorithms was motivated by their respective strengths in handling discrete action spaces, risk management, and stable policy optimization. During the hyperparameter experimentation, we prioritized parameters that directly influence model stability and exploration strategies, though time constraints limited the full scope of optimization. However, further tuning of parameters—such as learning rate, discount factors, and exploration decay—could potentially yield improved results, pushing the model's performance to its limits.

Several challenges were encountered, including balancing exploration with convergence speed, which was most evident in Deep SARSA due to its tendency to overly focus on recent policy adjustments. Future work could involve extensive hyperparameter tuning and adapting these algorithms to different environments to assess generalization capabilities. This could reveal further insights into the model's robustness across various contexts.

References

[1] D. Zhao, H. Wang, K. Shao, and Y. Zhu, "Deep reinforcement learning with experience replay based on sarsa," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–6.
[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. [Online]. Available: https://arxiv.org/abs/1707.06347
[3] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018. [Online]. Available: https://arxiv.org/abs/1506.02438