Reinforcement Learning on Lunar Lander

Sze-Hang, Wong
Georgia Institute of Technology

Abstract

Abstract In this paper, I will discuss various reinforcement learning algorithms to solve the problem of Lunar Lander with continuous actions. The first part of the paper discusses key considerations in model selection and the trade-off between them (section 2). In the second part of the paper, I discuss the implementation of PPO (Proximal Policy Optimization) (section 4), the hyper-parameters tunings as well as key challenges encountered during model training (section 6)

1 Overcooked Environment

Overcooked is a multi-agent environment simulating a cooperative cooking task in a constrained kitchen layout. Each agent receives observations consisting of surrounding grid cells, the total dimension of observation received from the environment is 96 (46 observed by target agent, another 46 observations by other agent and reminind 4 is the position information of target agent), including positions of ingredients (e.g., onions), cooking tools (e.g., pots). The action space includes six discrete commands: stay, up, down, left, right, and interact. Agents must coordinate to pick up ingredients, place them into pots, cook soups, and deliver them to a service window. The environment includes shaped rewards for subtasks such as picking up onions or placing them into pots, and a sparse reward for successful soup delivery. Episodes last a fixed at 400 timesteps. The agents must cooperate to maximize the numner of so

2 Considerations of Algorithm Selection

2.1 On-Policy vs Off-Policy

The advantage of On-Policy learning is training stability by avoiding the dead triad of Reinforcement Learning (Off-Policy, generalization and bootstrapping). The core idea of On-Policy learning is to improve the policy the agent is executing, i.e. behavior and target policy is the same, so the policy is keep updating during learning rendering the old samples useless once the policy is updated. Equation 1 is the typical optimization objective of On-Policy algorithm (Reinforce). the policy π_θ is in the expectation calculation, this show training samples are collect from specific policy at each policy optimization step and this match the definition of On-Policy learning:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot R_t \right] \tag{1}$$

On the other hand, the advantage of Off-Policy learning is the ability to reuse training samples collected from behavior policy to improve target policy through the optimization iterations cycle, so sample efficient is high. However, the learning might fall into the dead triad of reinforcement learning. Equation 2 is the typical loss function of Off-Policy algorithm (DQN), the policy π is not included in the loss function means the training samples are not depended on the target policy being optimized.

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a) \right)^2 \right] \tag{2}$$

2.2 Value-based Semi-Gradient vs Policy Gradient algorithms

Value-based Semi-Gradient algorithm aims to generalize the values of states (or state-action pairs), and an optimal policy is derived from these optimal values. Since value estimation can be performed using data from any policy, Off-Policy learning can be applied with value-based methods to improve sample efficiency.

Policy Gradient approaches improve their policies by learning how adjusting the policy affects overall performance. They optimize the policy in the direction that emphasizes state-action combinations with more positive impact on returns. Because they rely on samples generated by the current policy, policy gradient methods are essentially on-policy learning algorithms.

2.3 Continuous vs Discrete actions Space

Off-policy algorithm like DQN is limited to discrete action spaces, researchers developed algorithms like DDPG (Deep Deterministic Policy Gradient), which extend value-based learning to continuous action spaces. Instead of searching over all possible actions to find $\max_a Q(s, a)$, DDPG learns a deterministic policy $\pi(s)$ that outputs continuous actions directly, aiming to maximize $Q(s, \pi(s))$ during training. Other variation of DDPG are TD3 and SAC. Details of those algorithm can be found in Morales (2020)

On the other hand, all major on-policy policy gradient algorithms (e.g. Reinforce, A2C, PPO) naturally support continuous action spaces because they optimize a parameterized policy, and that policy can be defined over continuous actions spaces using probable distributions. To handle discrete action space, simply replace with categorical distribution.

3 Algorithm selection

I categorize algorithms into two groups: On-Policy (Table 1) Off-Policy (Table 2). The best algorithm from each category will be selected for a final comparison. The quantitative metrics used in this comparison—such as sample efficiency, implementation complexity, training stability, and support for continuous action spaces—have been discussed in the preceding section 2.

	Policy Type	**Action Space**	**Sample Efficiency**	**Implementation Complexity**	**Training Stability**
DQN	off	discrete	High	2 (Q-online & target)	Medium
DDPG	off	continuous	High	4 (Policy-online & target, Q-online & target)	Low
TD3	off	continuous	High	6 (Policy-online & target, Qx2-online & target)	Medium
SAC	off	continuous	High	5 (Policy, Qx2-online & target)	High

Table 1: Comparison of Off-Policy Algorithms

	Policy Type	**Action Space**	**Sample Efficiency**	**Implementation Complexity**	**Training Stability**
Reinforce	on	continuous	Low	1 (Policy)	Low
PPO	on	continuous	High	2 (Policy, Value)	Medium

Table 2: Comparison of On-Policy Algorithms

From the analysis, Proximal Policy Optimization (PPO) was chosen for the Lunar Lander problem because policy gradient algorithms naturally align with optimizing continuous action spaces. The simplicity of its design—particularly the use of a shared neural network for both the policy and value functions—is a definite advantage for debugging and feature enhancements. In contrast, Soft Actor-Critic (SAC) requires training at least two separate neural networks for the critics, adding complexity. Given the limited time and computational resources, PPO offered a more practical and efficient solution for this problem.

4 PPO (Proximal Policy Optimization)

4.1 Reinforce

Equation 3 is the core identity of the REINFORCE algorithm, upon which PPO is also conceptually built. A full proof can be found in Sutton and Barto (1998). It is observed that taking the gradient of the log probability of the policy function π_θ is equivalent to taking the gradient of the value function. Thus, maximizing the value function is equivalent to maximizing the log probabilities of actions under the policy π_θ.

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}[V^{\pi_\theta}(s)] = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot G_t\right] \tag{3}$$

where:

- $\pi_\theta(a_t \mid s_t)$ is the parameterized policy.
- $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ is the return from time t.
- $J(\theta) = \mathbb{E}[V^{\pi_\theta}(s)]$ is the expected return under policy π_θ.

To better understand the intuition behind why optimizing the policy function is equivalent to maximizing the value function, consider Equation 4, which defines the update rule for the policy parameters. This update shows that actions resulting in higher returns G_t contribute more significantly to the gradient and therefore have a larger influence on parameter updates.

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \left(\frac{\nabla_\theta \pi_\theta(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)} \right) \cdot G_t \tag{4}$$

Additionally, the smaller the probability $\pi_\theta(a_t \mid s_t)$ of selecting action a_t under the current policy, the larger the corresponding update magnitude (since it appears in the denominator). This means that rare actions leading to high returns will have a greater impact on learning. This aligns with intuition: discovering a new, unexpectedly successful action should influence the policy more than reinforcing already well-known good actions.

4.2 Reinforce with baseline

Equation 5 is the baseline version of REINFORCE, which incorporates a baseline function to adjust the returns G_t. Mathematically, adding a baseline has no effect on the expected gradient, but it plays a critical role in stabilizing training by helping the agent better distinguish between good and bad actions.

$$\nabla_\theta J(\theta) = \mathbb{E}_t\left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot (G_t - B(s_t))\right] \tag{5}$$

A raw return G_t alone doesn't tell much about the quality of an action—it only reflects the total reward received after that action. However, by comparing the return to a baseline, $B(s_t)$, we gain a better signal to determine the quality of action. This idea forms the foundation of Actor-Critic methods. In an Actor-Critic framework, the actor (policy) selects actions. The critic (value function) evaluates how good the chosen actions were, relative to the expected value of the state. This feedback allows the actor to improve by reinforcing actions that led to better-than-expected outcomes and discouraging those that did not.

4.3 Returns calculation - GAE

The choice of return calculation directly affects the bias–variance trade-off. As discussed in temporal-difference learning (Sutton and Barto (1998)), using the Monte Carlo return TD(1) results in high variance due to reliance on complete trajectories. But using the one-step value estimate TD(0) introduces high bias, since it bootstraps from current value estimates. To strike a balance between bias and variance, PPO adopts Generalized Advantage Estimation (GAE), which is conceptually similar to TD(λ). However, unlike TD(λ), which smooths over full returns, GAE operates on the temporal-difference residuals δ_t rather than raw rewards across future time steps. The GAE formula is:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{6}$$

where the TD residual δ_t is defined as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{7}$$

where:

- r_t = reward at time step t
- γ = discount factor
- $V(s_t)$ = value estimate at state s_t
- $V(s_{t+1})$ = value estimate at state s_{t+1}

4.4 Policy Ratio Clipping

Proximal Policy Optimization (PPO) introduces ratio clipping to limit how much the updated policy is allowed to deviate from the old policy during training. To achieve this, PPO uses importance sampling to rewrite the policy gradient from REINFORCE using samples drawn from the old policy $\pi_{\theta_{\text{old}}}$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{\text{old}}}} \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{8}$$

To simplify notation, PPO defines the policy ratio:

$$r_t(\theta) := \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

PPO applies a clipping mechanism to this ratio to stabilize training. Without clipping, extremely large or small ratios—caused by outlier samples or numerical instability—can lead to large, destabilizing updates that derail convergence. Clipping $r_t(\theta)$ within a small interval around 1 (typically $[1-\epsilon, 1+\epsilon]$) ensures that the updated policy does not deviate too much from the old one.

The clipped objective function is defined as:

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t \left[\min \Big(r_t(\theta)\hat{A}_t,\ \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t \Big) \right] \tag{9}$$

This objective discourages large policy updates in directions that would overly increase or decrease the policy probability, helping to maintain stable learning while still encouraging improvement.

5 Implementation

5.1 Module Design

Follow the Githash published at this paper for the full source code. Following are the major functions implemented in the PPO Python implementation:

Policy_Value_Model()	a Neural Network 512x256x256 - shared value-policy network
optimize_step()	This is where PPO optimization takes place. Each rollout (4096 timesteps) is fed into this function and processed over 5 passes (epochs). The rollout is divided into mini-batches of 64 timesteps, resulting in a total of 320 updates per rollout (calculated as 4096x5/64)
calculate_GAE()	GAE generating function to facilitate debugging and hyperparameter tuning (γ and λ)
main training loop	this is the entry point where sampling will be collected according to the latest policy. Each rollout contains 4096 time steps with a variable number of episodes.

6 Hyper-parameters Tuning

Several important hyper-parameters were researched before final training: Gamma, Clipping Parameter, Lambda

6.0.1 Lambda (λ)

Lambda (λ) controls the balance between bias and variance in estimating the advantage function using Generalized Advantage Estimation (GAE). Lower values of λ focus more on immediate rewards, which reduces variance but increases bias, while higher values look further ahead, reducing bias but increasing variance. I tested three values of λ: 0.1, 0.5, and 0.9. As shown in Figure 5.1, the algorithms converge fastest when $\lambda = 0.5$. When λ was too low (0.1), the estimates were too biased towards short-term rewards, which made training less stable and sometimes led to poor policies. On the other hand, when λ was high (0.9), the variance in the estimates was too large, causing noisy updates and slower learning.

Choosing $\lambda = 0.5$ hit a good balance: it reduced variance enough to keep training stable while still considering longer-term rewards well. This led to better convergence and higher rewards overall. Based on this, I used $\lambda = 0.5$ for the final training setup.

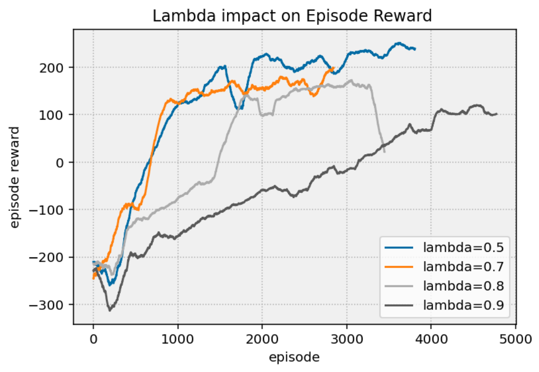

Fig 5.1

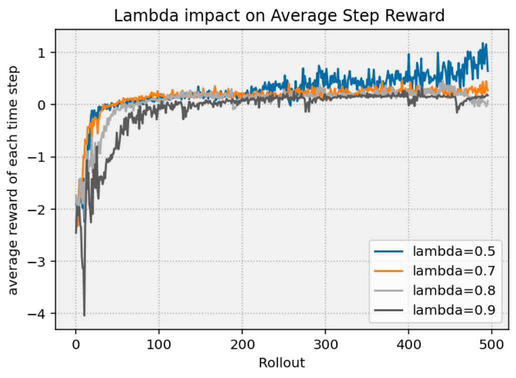

Fig 5.2

6.0.2 Gamma (γ)

In the Lunar Lander problem, a large reward is given when the lander successfully touches down on the landing pad. This suggests that using a high discount factor (γ) is beneficial, as it encourages the algorithm to focus on long-term outcomes. However, since the landing pad occupies only a small portion of the state space, reaching it may require a large number of episodes. Without sufficient exploration, the agent might never encounter this high reward. On the other hand, a low γ could cause the model to overemphasize immediate rewards, making it unlikely for the lander to complete a successful landing within the time limit. Given these competing considerations, I conducted experiments with various values of γ. As shown in Figure 5.3, the algorithm performed best when γ=0.99. Although γ=0.98 led to quicker initial success, it resulted in greater volatility, as evidenced by a significant performance drop around the 3500th training episode.

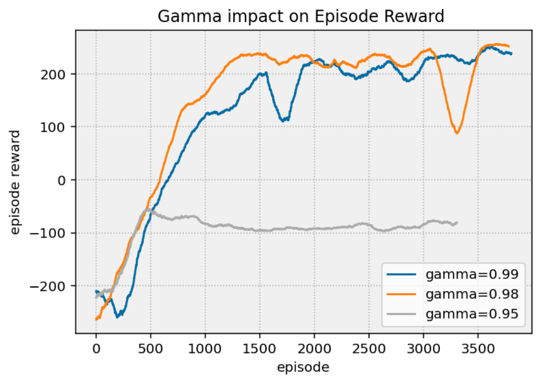

Fig 5.3

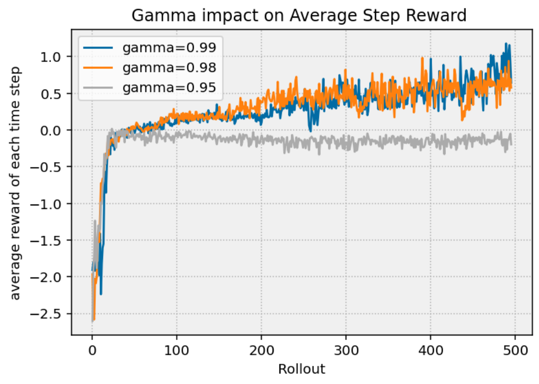

Fig 5.4

6.0.3 Clipping Parameter

A lower clipping parameter in PPO acts as a stronger regularizer, making training more stable by constraining how much the policy is allowed to change between updates. However, if the clipping parameter is set too low, several negative effects may arise. The most notable issue is slower training, as the magnitude of the gradient updates is restricted. More seriously, an overly tight clipping range can lead to underfitting. I tested three clipping parameters: 0.1, 0.15, and 0.2. As shown in Figure 5.5, all three settings reached the performance target at roughly the same time. However, the most stable training was achieved with the lowest clipping parameter of 0.1. This suggests that a lower clipping parameter did not hinder learning or cause underfitting, and in fact contributed to more stable training with competitive performance.

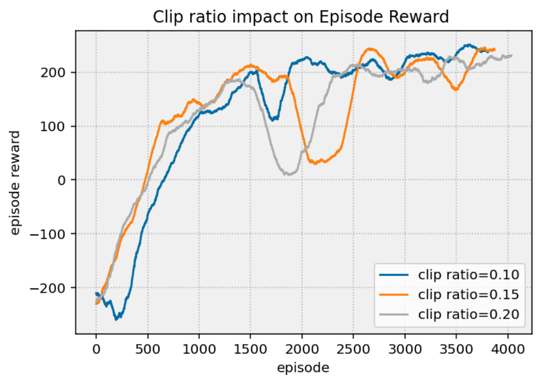

Fig 5.5

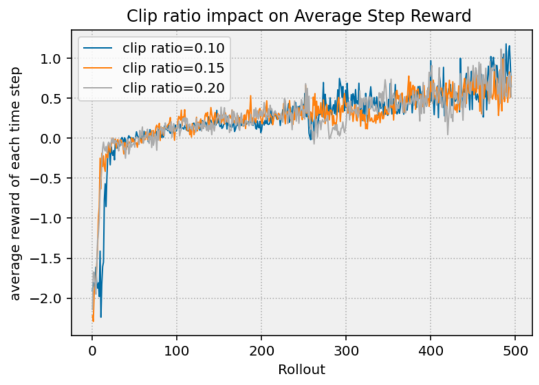

Fig 5.6

7 Major Training Obstacle - the Learning Rate

During initial experiments, I observed that the training process would occasionally collapse after approaching the target reward threshold of 200. This phenomenon is illustrated in Figure 5.7. After restoring a checkpoint just before the collapse, I tested several mitigation strategies: applying gradient clipping, lowering the learning rate, and decreasing the PPO clipping range. Among these, only lowering the learning rate effectively resolved the issue. The model continued to improve consistently after the adjustment. This suggests the collapse was caused by the loss function oscillating around a minimum due to the large learning rate. In such cases, if an outlier sample is encountered, it can produce a large gradient that pulls the model far away from the optimum, leading to instability or collapse. Although the model sometimes recovered from these collapses, it became highly unstable. Biased or corrupted samples could dominate the gradients temporarily, making the model require a long time to recover through new experiences. To address this, I adopted a linear learning rate decay schedule in the final training setup, which gradually reduces the learning rate over time to balance learning speed and stability.

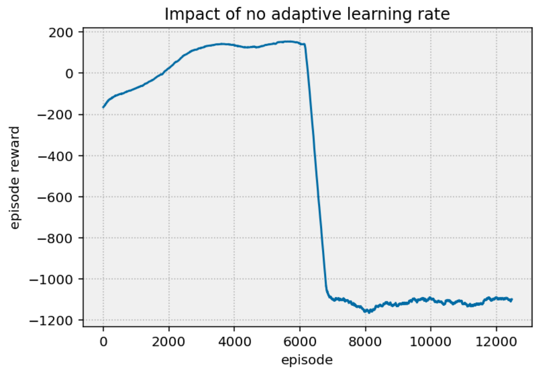

Fig 5.7

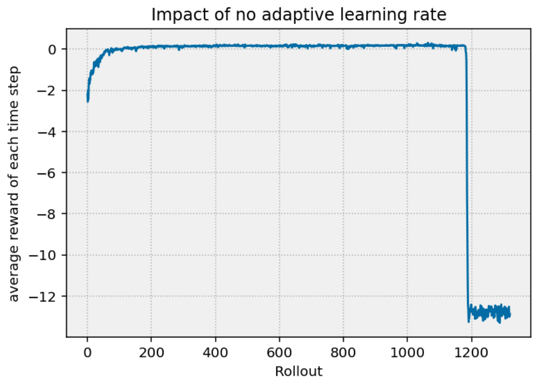

Fig 5.8

8 Solving the Lunar Lander

8.1 Result Presentation

Below are the main Hyper-parameters and training batches arrange of the final PPO model training, detailed discussion of choice of Hyperparameter was discussed at section 6:

Hyper-parameters	
Lambda (λ)	0.5
Gamma (γ)	0.99
PPO Clipping Parameter	0.1

Table 3: First Table

Training Batch Arrangements	
Rollouts	4000
Time-steps sampled per Rollout	4096
Training Epoch per rollout	5
Mini batch size	64

Table 4: Second Table

Figure 6.1 shows the testing results based on 100 consecutive landing episodes. The average reward across these episodes is **254.54**, which is significantly higher than the success threshold of 200, confirming that the Lunar Lander problem has been successfully solved. Figure 6.2 shows the evolution of episode rewards during training. Initially, the average reward is around -200. As training progresses, the reward reaches 200 around the 1000th episode. Training continues to stabilize, ultimately achieving the performance target around the 3500th episode.

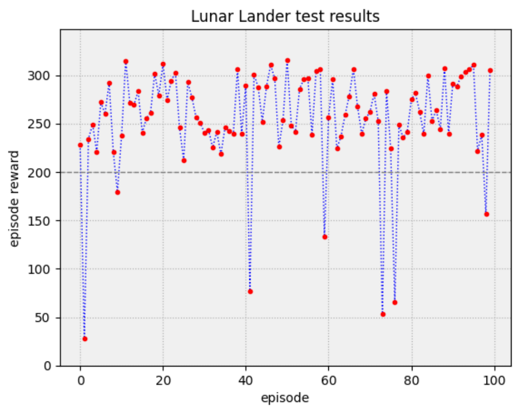

Fig 6.1

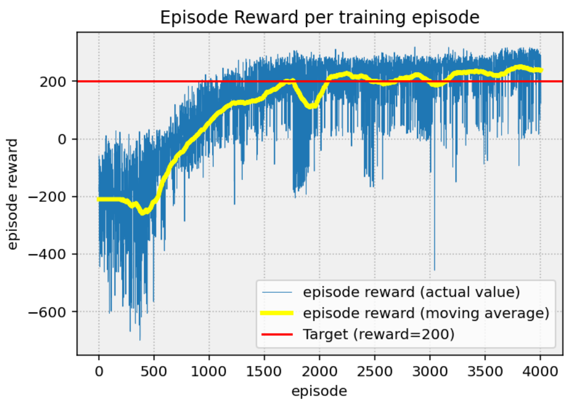

Fig 6.2

8.2 Result Discussion - What Worked and What Didn't

I began with a simple model to keep the project manageable within the given timeframe. Since I had already decided to use Policy Gradient algorithms, I started with REINFORCE. After a quick implementation, I found that convergence was extremely slow and the loss function fluctuated widely. I then upgraded the algorithm to A2C, implementing Generalized Advantage Estimation (GAE) to calculate the returns. This improved convergence somewhat, with the model converging in around 8,000 episodes, which is unreasonably long given the scale of this environment. Moreover, the testing results were inconsistent—only about 60% of the trials met the success criteria.

Finally, I upgraded the model to PPO by adding a gradient clipping component and tuning hyper-parameters as described in Section 6. The model converged quickly in around 3000 episodes and produced excellent results, exceeding the success criteria by a large margin.

9 Retrospect - What I Would Do Differently If I Tried Again

If I had more time to rerun the project, I would explore Value-based Off-Policy algorithms. I would aim to combine Off-Policy and On-Policy models using a voting or averaging mechanism. I believe such a combined model could achieve strong exploration and exploitation capabilities without being limited by the traditional exploration–exploitation trade-off.

References

Miguel Morales. *Deep Reinforcement Learning*. Manning Publication Co., Shelter Island, NY 11964, 2020.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.