Component-Based Design: Reinforcement Learning Algorithms Tailored for the Lunar Lander Gym Environment

Agrima Khanna
Computer Science Department
Vanderbilt University
Nashville, USA
agrima.khanna@vanderbilt.edu

***Abstract*—This study presents a systematic component-based design framework for implementing three distinct Reinforcement Learning (RL) algorithms: Deep Q-Networks (DQN), Deep Deterministic Policy Gradients (DDPG), and Proximal Policy Optimization (PPO). These algorithms are modeled to interface with the OpenAI Gym's Lunar Lander environment, that requires agents to learn control for safe landing. Each RL algorithm is decomposed into modular components, encapsulating specific functionalities such as policy networks, value estimation, and experience replay buffers, etc., enabling a clear delineation of responsibilities and facilitating ease of experimentation. The interoperability of components is designed with the flexibility to not only operate within their respective algorithmic confines but also to allow cross-communication where beneficial. This modularity supports rapid prototyping, testing, and fine-tuning of each algorithm's unique strategies, driving towards the goal of achieving efficient and robust landing maneuvers. The efficacy of this design is evaluated by the performance of the RL agents in learning optimal landing sequences, balancing exploration, and exploitation, and adapting to the intricate dynamics of the Lunar Lander simulation.**

***Index Terms*—Cyber-Physical Systems, Component-Based Design, Reinforcement Learning, Deep Q-Networks (DQN) Deep Deterministic Policy Gradients (DDPG) Proximal Policy Optimization (PPO), Lunar Lander.**

I. Introduction

Reinforcement Learning is a paradigm of learning in artificial intelligence that empowers an agent to make a sequence of decisions by interacting with a complex, uncertain environment to achieve a defined goal. It is distinguished by the agent's ability to learn optimal behavior through trial-and-error interactions with a dynamic environment, guided by rewards or penalties. Deep Reinforcement Learning (DRL) enhances this capability by integrating deep learning techniques, enabling the agent to interpret high-dimensional sensory inputs and make informed decisions.

Among the notable advancements in DRL, Deep Q-Networks (DQN) gained prominence following their success in achieving human-level control in complex environments, as highlighted by Mnih et al. (2015). [4] DQN leverages deep neural networks to approximate the optimal action-value function, which is crucial in value-based methods for reinforcement learning where the goal is to maximize cumulative future rewards.

Further, Deep Deterministic Policy Gradient (DDPG) is an algorithm that stands out as an efficient off-policy method suitable for environments with continuous action spaces, such as robotic control or autonomous vehicles. It combines the benefits of Policy Gradient methods and Q-Learning, enabling direct policy learning while maintaining a separate value function for stability and efficiency. The Soft Actor-Critic (SAC) algorithm, an evolution of DDPG, introduces an entropy maximization term that encourages exploration and robustness in policy development, which is particularly effective in high-dimensional and complex tasks. [3]

Proximal Policy Optimization (PPO) is another significant contribution to the field, representing a stable and effective on-policy approach. It simplifies and improves the trust region policy optimization by using a clipped surrogate objective function, which prevents large policy updates and ensures stable and reliable learning performance. [5]

Modelling reinforcement learning algorithms as components addresses the increasing complexity and scalability demands of these applications. Decomposing algorithms into modular components enhances understanding, iteration, and extension of their capabilities. It also promotes component reuse and system interpretability. Importantly, this approach facilitates increased concurrency within the system, allowing for parallel processing and more efficient execution. [2] Adhering to software engineering best practices, this modular structure can support the continuous evolution of reinforcement learning algorithms.

In this project, I have utilized the Python-based OpenAI Gym module to simulate the Lunar Lander environment, as illustrated in Figures 1 and 2. Figure 1 depicts the lander's initial state upon entering the environment, while Figure 2 demonstrates the lander undergoing various maneuvers. The lander, represented in black in the figures, is tasked with the challenge of making a soft landing on a flat surface, depicted as the gray ground. The lander is equipped with continuous control thrusters that allow it to navigate left, right, and downward to counteract the gravitational pull.

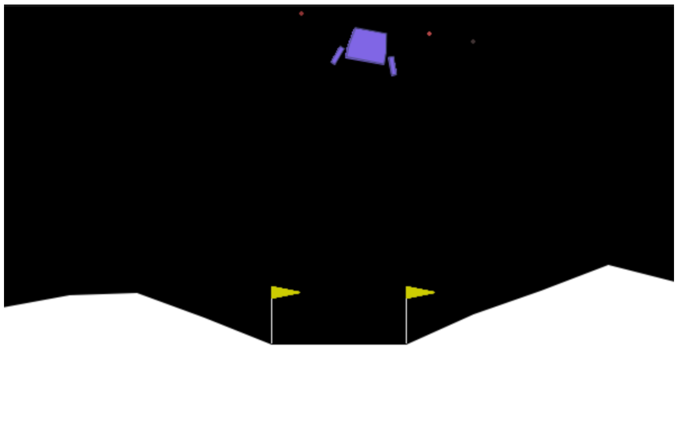


Fig. 1: Initial state

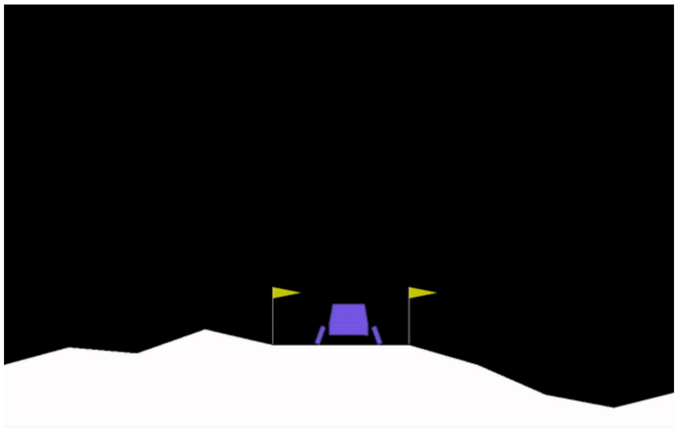


Fig. 2: Successful landing of the agent

The primary objective is to guide the lander to a safe touchdown at a designated landing zone, marked by two flag poles, without running out of fuel or causing a crash. Successful landings are indicated by the lander resting between the flags, maintaining an upright position, and minimizing velocity upon touchdown. Running the code enables the observation of the lander's behavior as it attempts to achieve this goal under a specified policy, which can range from random actions to sophisticated, learned behaviors through various reinforcement learning algorithms.

II. Methodology

A. Lunar Lander Environment

The Lunar Lander environment on is a simulation within the Gymnasium-Box2D environments. It presents a classic rocket trajectory optimization problem, following Pontryagin's maximum principle, which suggests it's optimal to fire the engine at full throttle or turn it off. The environment offers two versions: discrete or continuous. [1]

- Action Space: The lunar lander environment has two variations. One with a discrete action space and the other with a continuous action space.
  - Discrete Action Space: In this mode, there are four specific actions available:
    - Do nothing.
    - Fire the left orientation engine.
    - Fire the main engine.
    - Fire the right orientation engine. These actions are distinct and non-overlapping, meaning the lander can only perform one of these actions at any given time.
  - Continuous Action Space: This mode offers a broader range of actions, represented as a continuous range of values. The actions control the throttle of the engines:
    - The main engine's throttle varies continuously based on the input value, with negative values turning off the engine and positive values controlling the engine power from 50
    - Lateral booster throttle is also continuously controlled. Depending on the input value, either the left or right booster will fire, or none will activate if the value is within a certain mid-range. Like the main engine, the power varies from 50
- Observation Space: The state is an 8-dimensional vector, including the lander's coordinates in x and y, linear velocities in x and y, angle, angular velocity, and two booleans indicating whether each leg is in contact with the ground.
- Rewards: Rewards are given after every step, varying based on the lander's proximity to the landing pad, its speed, tilt angle, and whether the legs are in contact with the ground. Using the side engine and the main engine affects the reward negatively.
- Starting State: The lander begins at the top center of the viewport with a random initial force applied.
- Episode Termination: An episode ends if the lander crashes, leaves the viewport, or is not awake (a state in Box2D where the body doesn't move or collide with other bodies).

B. Interface Descriptions

This section outlines the main components constituting the system. The Trainer interacts with the gym environment during the training phase and outputs the observation to the agent and the tester interacts with and renders the environment during the testing phase. The trainer and tester have the same functionality in all three interfaces.

- DQN Interface The Deep Q-Network (DQN) addresses decision-making problems in the Lunar Lander environment with discrete action space. At its core, the DQN component integrates a deep neural network with a reinforcement learning framework, enabling the approximation of the optimal action-value function. This component is structured to efficiently process high-dimensional state inputs, making informed decisions to maximize cumulative rewards. The key components include:

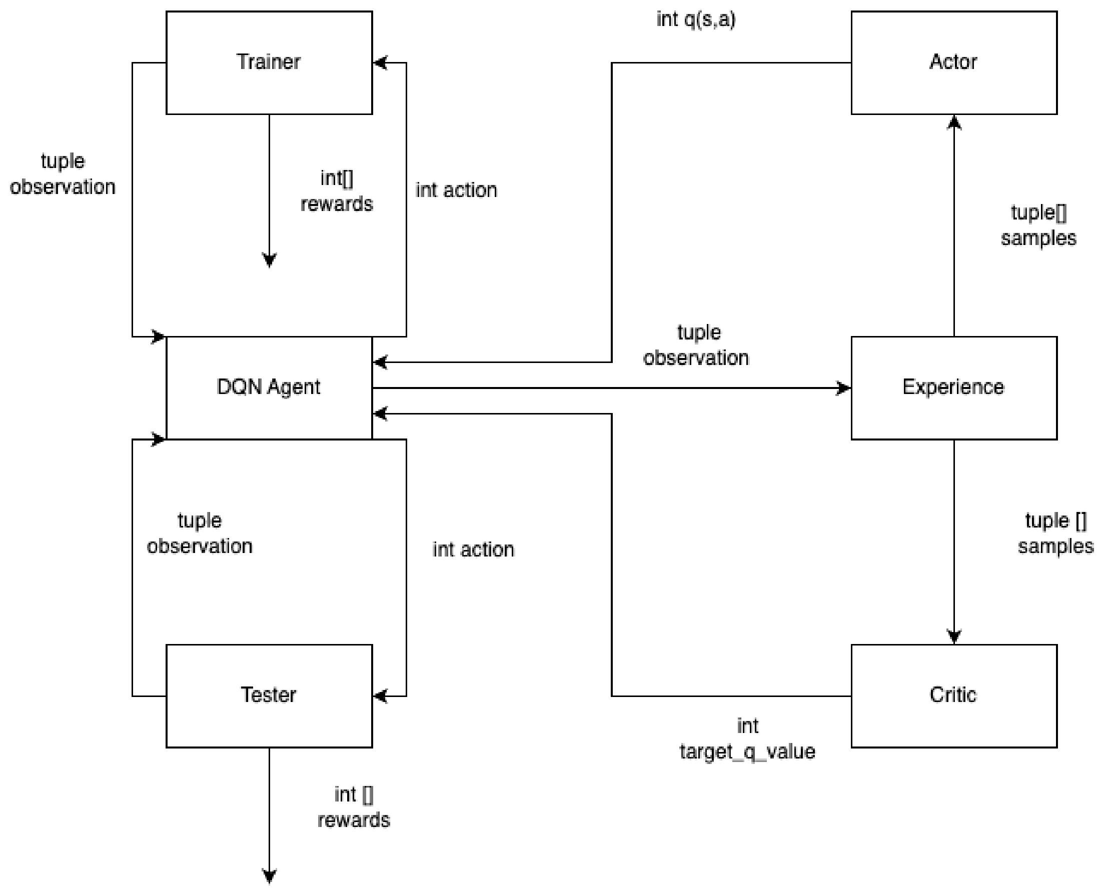


Fig. 3: Component diagram for the DQN interface

– DQN Agent: The agent facilitates flow of data between the Trainer/Tester components with the Q-Network, Experience component, and the Target Network. It is also used for loss computation.
– Q-Network: The network is used to map Q-values to state action pairs.
  ∗ Input Layer: The network begins with an input layer, which has a number of neurons equal to the state size. This layer takes the state of the environment as its input.
  ∗ First Hidden Layer: Following the input layer, there is a hidden layer with 64 neurons. This layer processes the input state.
  ∗ Second Hidden Layer: After the first hidden layer, there is a second hidden layer which also has 64 neurons. This further transforms the representation of the state input.
  ∗ Output Layer: The final layer of the network is the output layer with a number of neurons equal to the size of the action space (which is 4 in this case). This layer produces the Q-value estimates for each possible action in the given state.
  ∗ ReLU Activation Functions: For both hidden layers, the ReLU (Rectified Linear Unit) activation function is applied. This non-linear activation function allows the network to capture complex patterns in the data.
– Experience Replay Buffer: A mechanism to store and randomly sample previous experiences, thus breaking the correlation between consecutive learning steps and enhancing learning stability.
– Target Network: An additional network used for stabilizing the training process by providing a fixed target for Q-value updates. The architecture is the same as that of the Q-Network.
– Trainer and Tester

- DDPG Interface The Deep Deterministic Policy Gradient (DDPG) component offers a robust solution for environments with continuous action spaces. DDPG utilizes an actor-critic approach, where the actor network proposes actions given the current state, and the critic evaluates these actions. The component features include:
  – DDPG Agent: The agent facilitates flow of data (observed state and action to be taken) between the Trainer/Tester components with the Actor Network, Critic Network, and the Experience component.
  – Actor Network: Determines the best action to take in a given state. The Actor Network is structured as

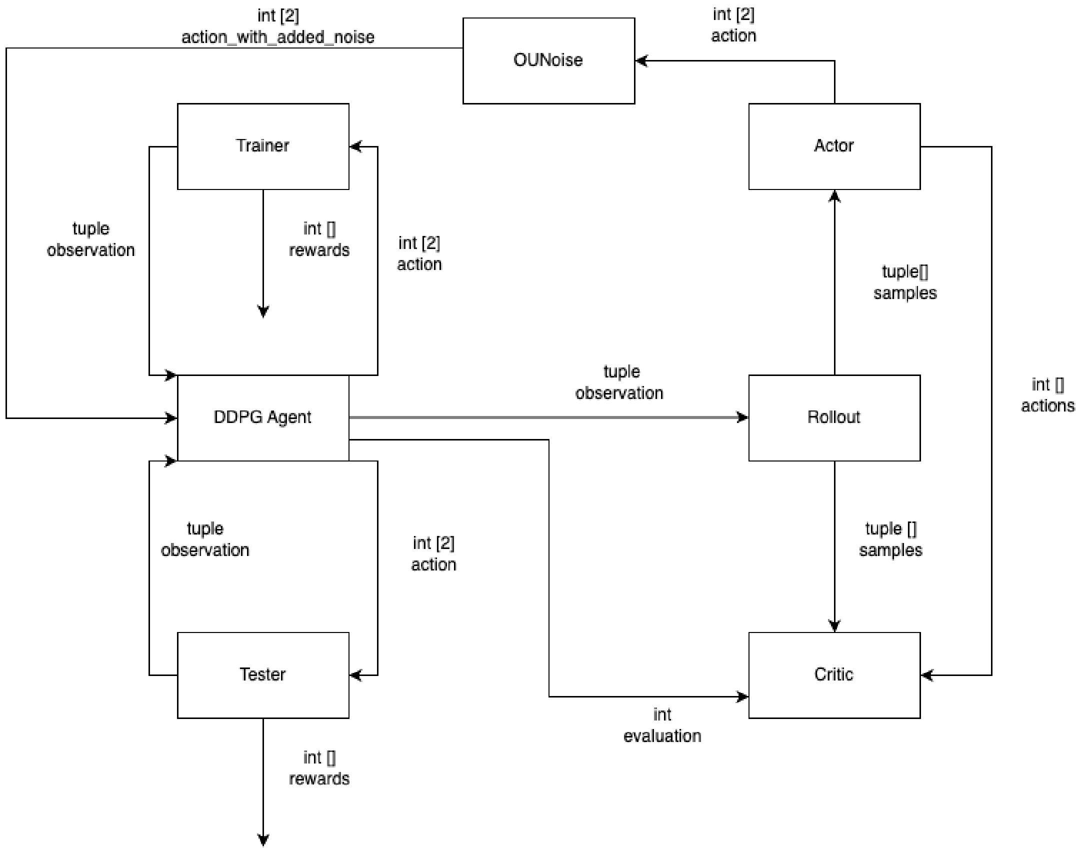


Fig. 4: Component diagram for the DDPG interface

follows:

 * Input Layer: This layer has neurons corresponding to the environment's state dimension, which is 8 neurons.
 * First Hidden Layer: Contains 400 neurons, receiving and processing the input state data.
 * Second Hidden Layer: Comprises 300 neurons, providing additional analysis of the state information.
 * Output Layer: Has neurons equal to the action space dimension (4 in this case), which dictate the action probabilities.
 * ReLU Activation Functions: These are utilized post both hidden layers for non-linear transformation of data.
 * Layer Normalization: Implemented after each hidden layer to ensure consistent learning and data scaling.

– Critic Network: Evaluates the chosen action and state pair, estimating their value. The architecture of the Critic Network is designed as follows:

 * Input Layer: Composed of neurons equal to the state dimension, which in this case is 8.
 * First Hidden Layer: Contains 400 neurons, processing the input from the state.
 * Second Hidden Layer: Has 300 neurons, further analyzing the information from the first layer.
 * Action Value Layer: Maps the number of actions, equal to the action space dimension, to the second hidden layer's dimensions.
 * Output Layer: Consists of a single neuron that outputs the estimated Q-value for a given state-action pair.
 * ReLU Activation Functions: Applied in both hidden layers to introduce non-linearity.
 * Layer Normalization: Used after each of the two hidden layers to stabilize the learning process.

– Ornstein-Uhlenbeck Process: A noise generation process for exploration, enhancing the agent's ability to discover new strategies.
– Replay Buffer: Same as the one in DQN.
– Trainer and Tester

• PPO Interface Proximal Policy Optimization (PPO) is particularly adept at handling problems where careful balance between exploration and exploitation is essential. The PPO component is characterized by:

  – PPO Agent: The agent facilitates flow of data between the Trainer/Tester components with the Actor Network, Critic Network, and the Rollout Buffer component.

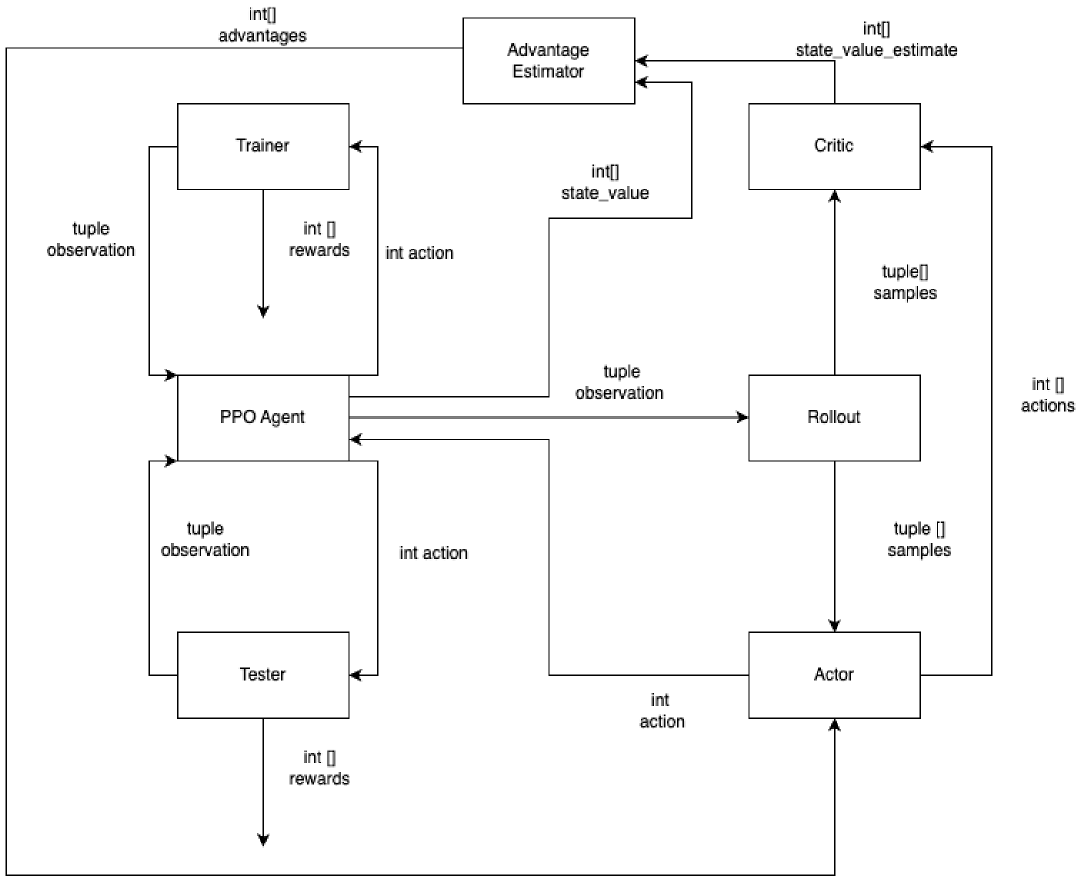
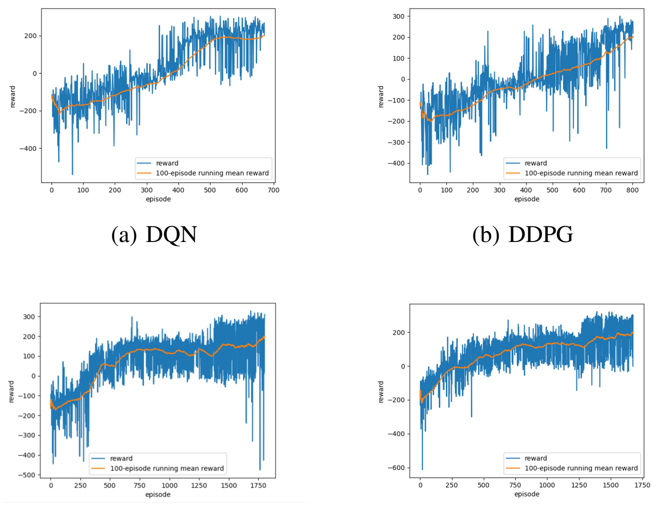


Fig. 5: Component diagram for the PPO interface

- Actor Network: The Actor Network in the PPO framework is structured as follows:
  - Input Layer: This layer has neurons corresponding to the environment's state dimension, which is 8 neurons.
  - First Hidden Layer: Contains 64 neurons, receiving and processing the input state data.
  - Second Hidden Layer: Comprises 64 neurons, providing additional analysis of the state information.
  - Output Layer: Has neurons equal to the action space dimension (4 in this case), which dictate the action probabilities.
  - Tanh Activation Functions: The tanh (hyperbolic tangent) function is used for activation in each hidden layer
- Critic Network: The Critic Network in the PPO framework is structured as follows:
  - Input Layer: Composed of neurons equal to the state dimension, which in this case is 8.
  - First Hidden Layer: Contains 64 neurons, processing the input from the state.
  - Second Hidden Layer: Has 64 neurons, further analyzing the information from the first layer.
  - Action Value Layer: Maps the number of actions, equal to the action space dimension, to the second hidden layer's dimensions.
  - Output Layer: Consists of a single neuron that outputs the estimated Q-value for a given state-action pair.
  - Tanh Activation Functions: The tanh function is used in both hidden layers of the critic network as well.

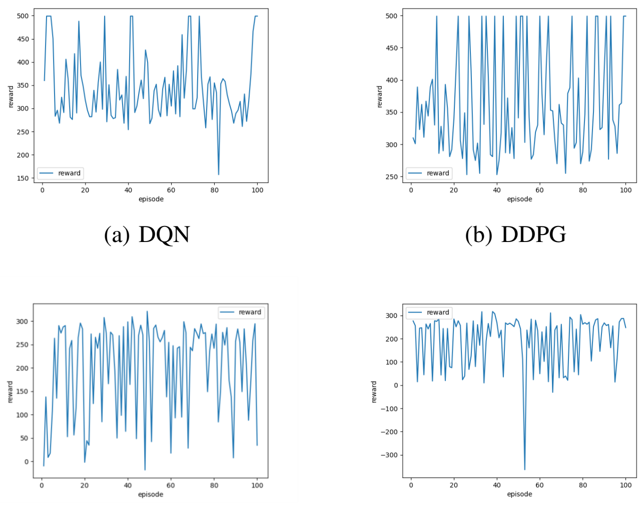
- Advantage Estimator: The Advantage Estimator in PPO calculates the advantage of taking specific actions in a given state, which is essential for updating the policy. It typically involves computing the difference between the actual return and the value estimate provided by the critic network.
- Clipped Objective Function: The Clipped Objective Function in PPO ensures that the updates to the policy are not too large, which helps maintain training stability. It works by clipping the ratio of the new policy probability to the old policy probability, thus bounding the updates within a specific range.
- Rollout Buffer: The Rollout Buffer in PPO stores trajectories experienced by the agent in the environment. It keeps track of states, actions, rewards, and other necessary information like action probabilities,



(a) DQN (b) DDPG

(c) PPO (discrete) (d) PPO (continuous)

Fig. 6: Training results for each algorithm.



(a) DQN (b) DDPG

(c) PPO (discrete) (d) PPO (continuous)

Fig. 7: Testing results for each algorithm

value estimates, and done flags for each step in the episode.

– Trainer and Tester

III. Results and Analysis

A. Training Results

Fig. 6 presents a summary of the training outcomes. The plots display the reward per episode along with the running mean over 100 episodes. Each graph concludes once the 100-episode mean nears a value of 200.

B. Testing Results

Fig. 7 presents a summary of the testing outcomes. The average rewards for all algorithms are depicted over 100 test episodes.

C. Safety Requirements

I have defined the following safety requirements for the systems.

- The agent must always avoid actions that would lead to a crash. This includes maintaining a safe altitude until landing and avoiding abrupt or dangerous maneuvers.
- The agent's actions should always keep the lander within designated operational areas, avoiding straying too far off course.
- All data exchanges between system components (Actor Network, Critic Network, Replay Buffer, etc.) must maintain integrity and adhere to expected formats. This ensures that each component accurately interprets and processes the data it receives.
- The Agent component cannot interact with the tester component before training is completed.

D. Liveness Requirements

I have defined the following liveness requirement for the systems.

- The trainer component must meet the termination criteria of an average reward of 200 over the last 100 trajectories.

E. Analysis

Even though I have outlined safety and liveness requirements for the Lunar Lander environment in this report, it is crucial to acknowledge that while all four systems meet the liveness requirement this does not inherently guarantee safety. This lack of inherent safety arises from several factors related to the characteristics of reinforcement learning (RL).

The reinforcement learning approaches employed ensure the algorithm learns optimal policies based on experiences. However, one of the fundamental limitations of RL is the inability to guarantee performance in states that the algorithm has not previously encountered. An example of this limitation was observed during the testing phase of the PPO (continuous) algorithm. In one particular episode, the agent incurred a reward of negative 300, a result likely stemming from the agent encountering a scenario it had not experienced before. This unpredictability in novel situations can pose safety risks, especially in an environment as dynamic and potentially hazardous as Lunar Lander.

Reinforcement learning algorithms are also notably sensitive to hyperparameters, such as learning rates and optimizers. Variations in these parameters can drastically alter the learning experience and the resulting policy effectiveness. Such sensitivity can lead to inconsistent performance, further complicating the assurance of safety.

The random initialization of the neural network weights adds another layer of unpredictability. This randomness affects the initial learning trajectory, potentially leading to varied learning outcomes and experiences. As a result, the initial phase of training can significantly influence the safety and effectiveness of the final policy.

Given these factors, it is important to recognize that while we strive for safety, the system can only be as safe as the range and diversity of experiences it has encountered during training. In essence, the system is optimized to make safe and effective decisions within the scope of its training experiences. Situations outside of this scope may lead to suboptimal or unsafe responses, as exemplified by the unexpected episode in the PPO algorithm's testing phase.

References

[1] Lunar lander: Gym documentation. https://www.gymlibrary.dev/environments/box2d/lunar_lander/.

[2] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, April 2015.

[3] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *ArXiv:1812.05905*, 2019.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

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