Project Report (Task 6)

Lunar Lander Problem (Reinforcement Learning)

CS4486 Artificial Intelligence

12.04.2024

Ishaank CHOPRA

SID: 56852511

I. Background

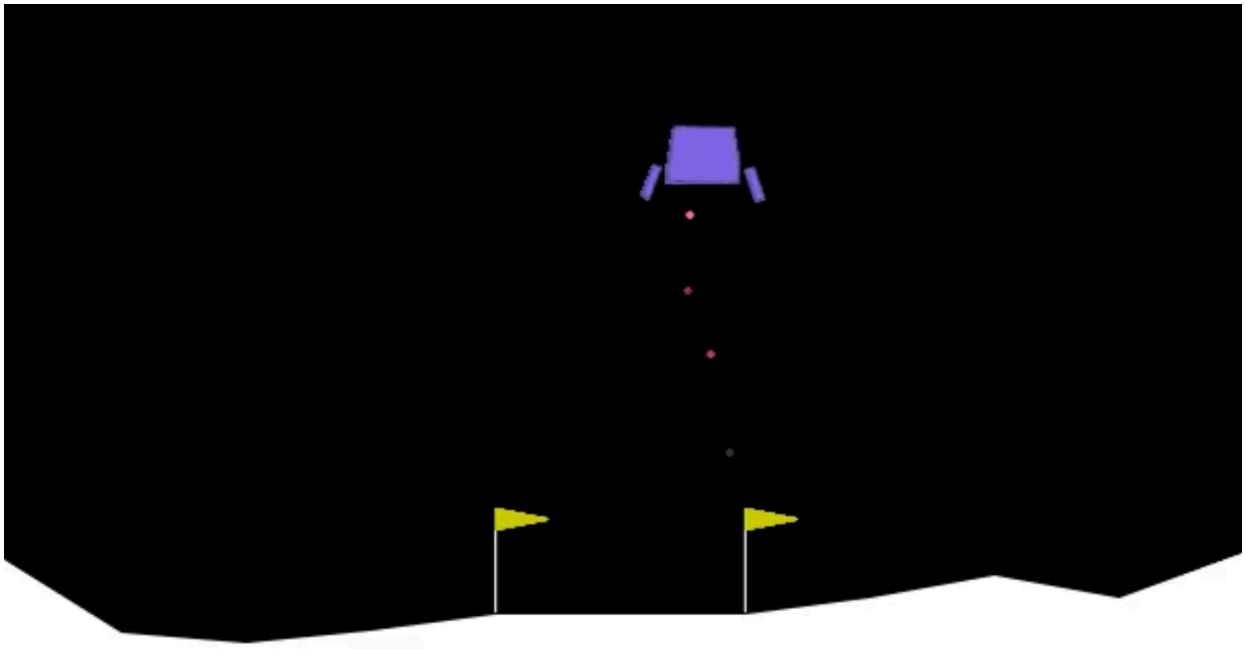
Landing a Lunar Lander: A Challenge of Control and Precision

Imagine a spacecraft, hurtling towards the lunar surface. Its mission: to touch down safely on the designated landing pad. This seemingly simple task becomes a complex dance of control and precision when faced with the realities of space.

LunarLander-v2, developed by OpenAI, is an exciting reinforcement learning environment. The objective of this simulation is to guide a lunar lander spacecraft to a safe landing on a designated pad located on the moon's surface. To achieve this, the player controls the lander by applying thrust to its engines in four different directions: left, right, downward, and upward. The environment adheres to the laws of physics, including gravity and the effects of engine thrust.

The game serves as a rigorous test for an agent's learning capabilities and decision-making skills within a complex and dynamic setting. The lunar lander faces the challenge of navigating through rocky, uneven terrain while avoiding obstacles. Simultaneously, it must carefully manage its descent to reach the designated landing pad. Notably, the lander operates with unlimited fuel, and the game concludes either upon a successful landing or a crash.

The environment is implemented using OpenAI Gym, a toolkit designed for developing and comparing reinforcement learning algorithms. In this game, the state of the lunar lander is described by several variables, including its position, velocity, fuel level, and angle. The agent's actions correspond to the thrust applied to the engines in different directions. For each successful landing, the agent receives a reward, while crashing or excessive thrust usage incurs penalties. Ultimately, the agent's objective is to learn a policy—a mapping from states to actions—that maximizes the expected cumulative reward over time.



The LunarLander-v2 Environment is:

- Fully Observable: All necessary state information is known and observed at every frame.
- Single Agent: There is no competition or cooperation; only one agent is involved.
- Deterministic: There is no stochasticity in the effects of actions or rewards obtained.
- Episodic: Reward is dependent only on the current state and action.
- Discrete Action Space: Only four actions are available: Thrust, Left, Right, and Nothing.
- Static: There is no penalty or state change during action deliberation.
- Finite Horizon: The episode terminates after a successful landing, crash, or a set number of steps.

Mixed Observation Space:

- X Coordinate of Lander
- Y Coordinate of Lander
- X Coordinate of Linear Velocity
- Y Coordinate of Linear Velocity
- Angle Of Lander
- Angular Velocity Of Lander
- Left Leg Is Grounded? (Boolean)
- Right Leg Is Grounded? (Boolean)

Finally, after each step, a reward is granted.
The total reward of an episode is the sum of the rewards for all steps within that episode.

Reward System:

- Moving from the top of the screen to the landing pad and coming to rest: + 100–140 pts
- Lander moves away from landing pad: loses reward same as moving same distance towards the pad
- Landing/Crashing: +/- 100 pts
- Grounding Each Leg: + 10 pts
- Firing Main Engine: -0.3 pts per frame
- Firing side engine: -0.03 pts per frame

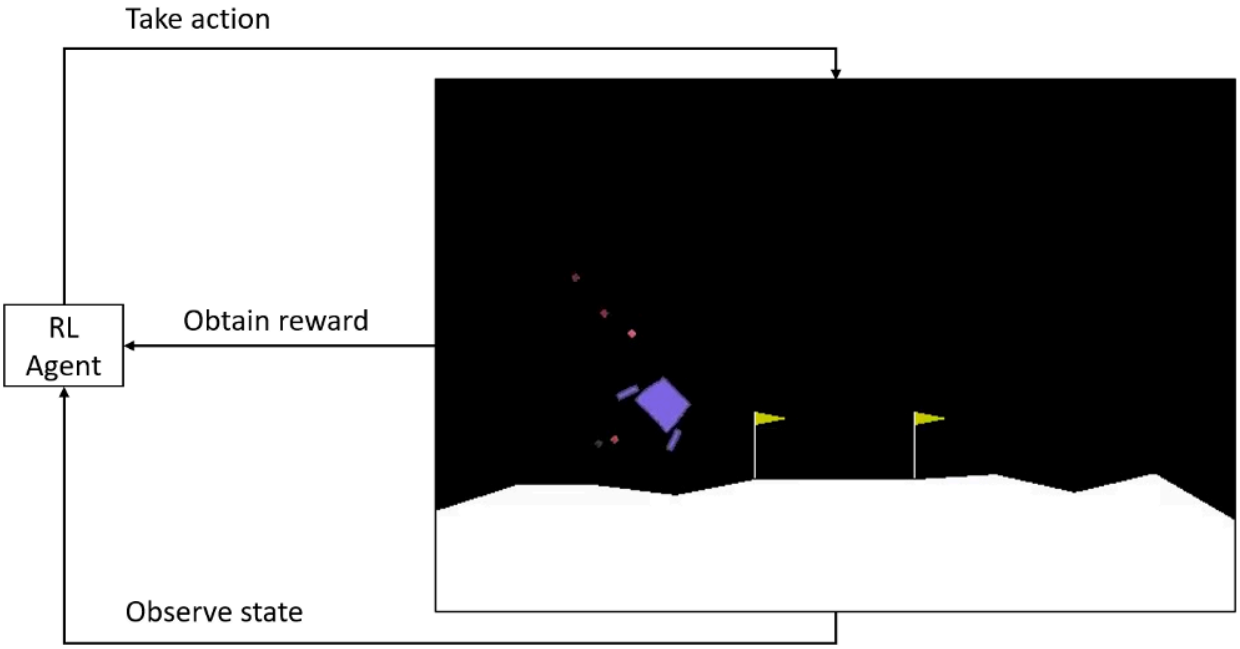
A score of 200 or more upon successful landing is considered a solution.

By successfully training an agent to land the Lunar Lander, I aim to develop an intelligent system capable of handling complex tasks in challenging environments. This project holds significant implications for real-world space exploration and the advancement of autonomous navigation systems. The knowledge and techniques acquired here could lead to more efficient and dependable landing procedures in upcoming space missions, unlocking exciting possibilities for lunar exploration and beyond.

II. Method

I have used **reinforcement learning** for this project. Reinforcement Learning (RL) is a specialized area within machine learning. Its primary focus is training an agent to interact with an environment and learn how to make decisions that maximize a cumulative reward. RL draws inspiration from the way humans learn through trial and error by interacting with their surroundings.

In a standard Reinforcement Learning framework, an AI agent engages with an environment through interaction. The agent observes the current **state** of the environment and, guided by a specific **policy**, selects an **action** to take. The chosen action is then applied to the environment, causing it to progress by a single **step**. As a result, a **reward** is generated, indicating the positive or negative impact of the action within the context of the game. By utilizing this **reward**, the agent aims to adjust its policy to optimize future rewards.



PPO (Proximal Policy Optimization) Algorithm

As I've established, RL provides the foundation for training our lunar lander agent. It equips the agent with the ability to learn from interactions with the environment, adapt its behavior, and optimize its policy. Now, let's zoom in on Proximal Policy Optimization (PPO) — a powerful technique that refines this learning process.

The Proximal Policy Optimization (PPO) algorithm, introduced by the OpenAI team in 2017, quickly became one of the most popular reinforcement learning methods, surpassing other RL approaches at that time. PPO operates by collecting a small batch of experiences from interactions with the environment. Using this batch, it updates its decision-making policy. However, unlike some other methods, PPO discards these experiences after each update. This approach is known as "on-policy learning," where the collected experience samples are only helpful for improving the current policy. In summary, PPO efficiently adapts its policy based on recent experiences to maximize cumulative rewards over time.

The core concept is to keep the new policy reasonably close to the old policy after an update. To achieve this, PPO uses clipping to prevent overly large updates. While this introduces minor training variance and some bias, it ensures smoother training. Importantly, it prevents the agent from taking nonsensical actions that could lead to irrecoverable mistakes.

The mathematical equation is as follows:

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta))\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)]$$

where,

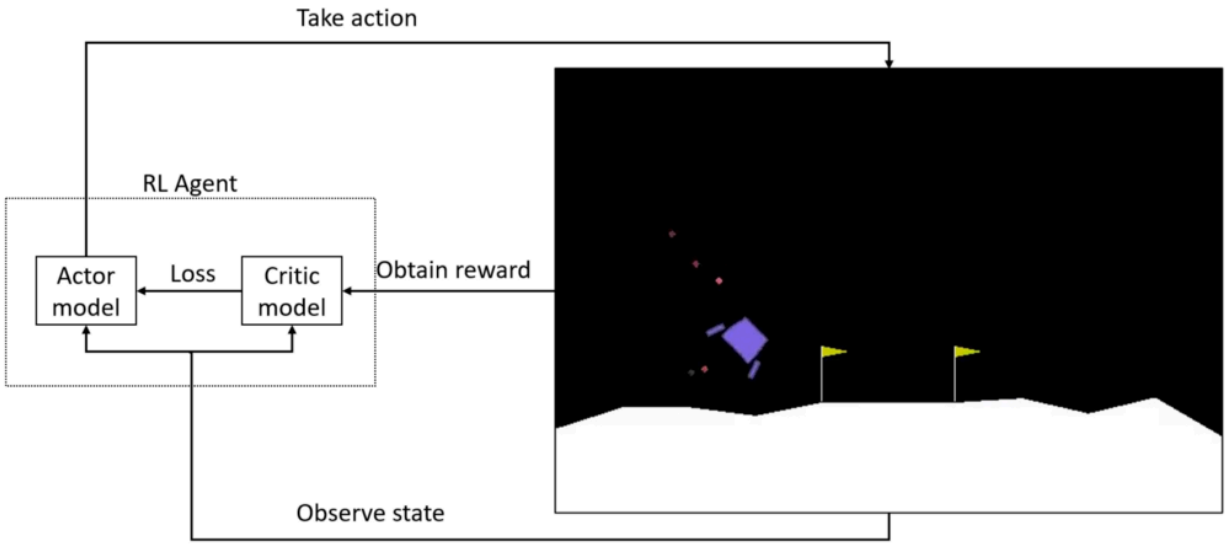
- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

More about PPO in mathematical detail can be found at
https://spinningup.openai.com/en/latest/algorithms/ppo.html

Now, let's dive deeper into how our agent defines and updates its policy.

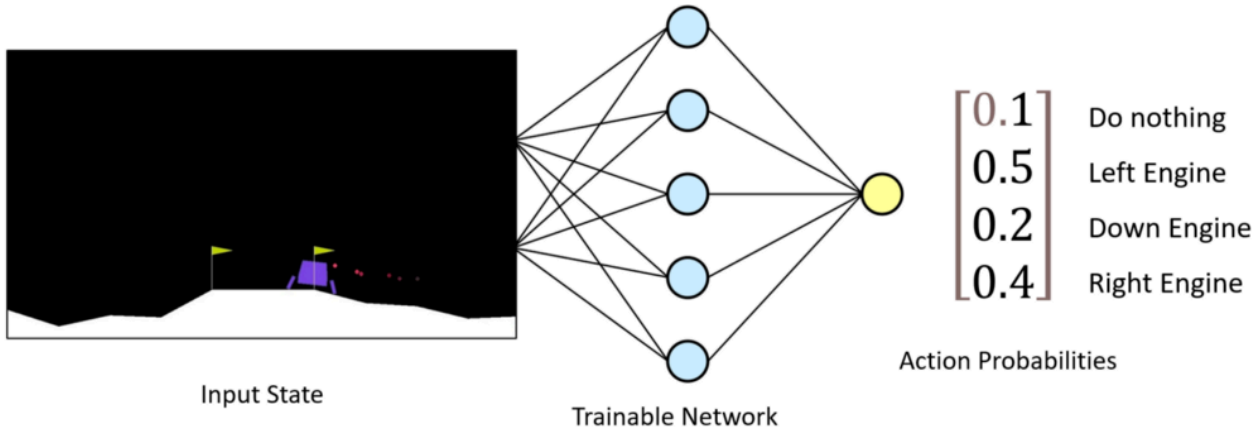
The Actor-Critic model's structure:

PPO utilizes the Actor-Critic approach for the agent, which relies on two models: the Actor and the Critic



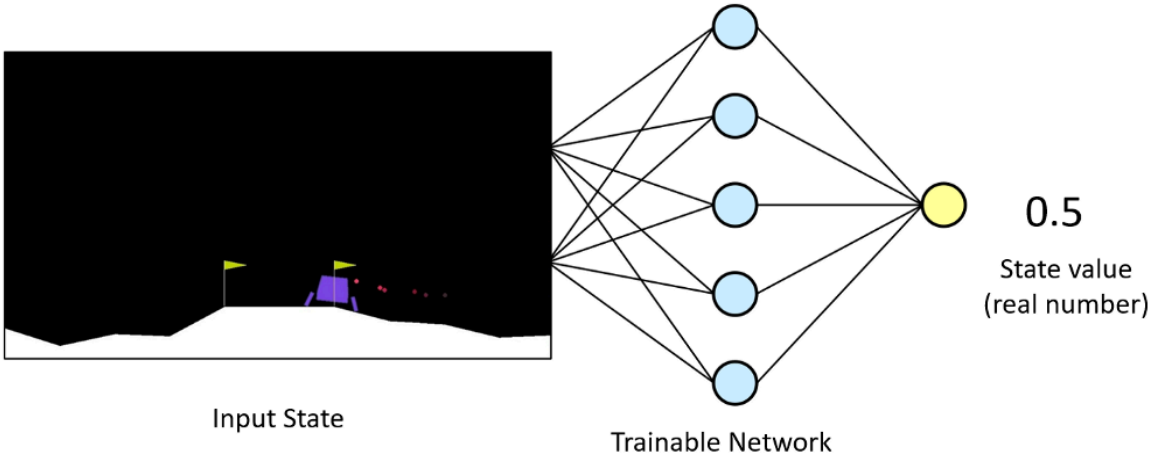
The Actor model:

In LunarLander-v2 environment, the Actor model plays a crucial role in learning which action to take based on the observed state of the environment. Specifically, it takes an eight-value list representing the current state of the rocket (such as position, velocity, and other relevant variables) as input. The output of the Actor model determines which engine to fire, guiding the lunar lander's actions during the game.



The Critic model:

When we send the action predicted by the Actor to the environment, we closely observe the game's outcome. Positive outcomes, such as successful spaceship landings, result in positive rewards from the environment. Conversely, if the spaceship falls or unfavorable events occur, we receive negative rewards. These rewards play a crucial role in training our Critic model, which evaluates the quality of our actions based on their impact on the game.



The Critic model plays a crucial role in reinforcement learning. Its primary task is to evaluate whether the action taken by the Actor improved the environment's state. The Critic provides feedback by outputting an actual number—a Q-value—that rates the action taken in the previous state. By comparing this rating from the Critic, the Actor can assess its current policy and decide how to enhance itself for better decision-making.

The structure of the Critic neural network closely resembles that of the Actor. The primary distinction lies in the final layer of the Critic, which outputs an actual numerical value. Consequently, the activation function used is Linear (not Softmax), as we do not require a probability distribution similar to the Actor.

A crucial step in the PPO algorithm involves running through this entire loop with the two models for a fixed number of steps, known as PPO steps. During these steps, we interact with the environment, collecting states, actions, rewards, and other relevant information. However, before training, we must process these rewards to ensure that our model can effectively learn from them.

Now, let us take a look at my code:

1) Importing the necessary libraries for the implementation of the reinforcement learning algorithm and the training of the neural network models.

```
#Importing the necessary libraries
import matplotlib.pyplot as plt
import numpy as np
import pickle
import gym

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as func
import torch.distributions as dist
```

- matplotlib.pyplot: A library for creating visualizations and plots.
- numpy: A library for numerical computing that provides support for arrays and mathematical operations.
- pickle: A module for object serialization, used for saving and loading Python objects.
- gym: A library that provides a collection of environments for reinforcement learning tasks, including the Lunar Lander environment used in this project.
- torch: The PyTorch library, which is a popular deep learning framework.
- torch.nn: A module in PyTorch that provides classes and functions for defining neural networks.
- torch.optim: A module that provides various optimization algorithms for training neural networks.
- torch.nn.functional: A module that contains various functions for implementing neural network operations.
- torch.distributions: A module that provides classes for probability distributions used in reinforcement learning algorithms.

2) Defining two neural network models: the Actor and Critic models. These models are commonly used in reinforcement learning algorithms to enable an agent to learn and improve its decision-making capabilities.

```
#Actor Neural Network model
class Actor(nn.Module):
    def __init__(self, i_dim, o_dim, h_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(i_dim, h_dim),
            nn.Dropout(p=0.16),
            nn.PReLU(),
            nn.Linear(h_dim, o_dim),
            nn.Softmax(dim=-1)
        )

    def forward(self, x):
        return self.net(x)

#Critic Neural Network model
class Critic(nn.Module):
    def __init__(self, i_dim, h_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(i_dim, h_dim),
            nn.Dropout(p=0.16),
            nn.PReLU(),
            nn.Linear(h_dim, 1)
        )

    def forward(self, x):
        return self.net(x)
```

- The Actor model is responsible for mapping the input state to a probability distribution over possible actions. It uses a sequential network architecture with linear layers to transform the input state. Dropout regularization is applied to prevent overfitting, and a parametric rectified linear unit (PReLU) activation function introduces non-linearity. The final linear layer is followed by a softmax activation function, which ensures that the output probabilities sum up to 1, representing the probabilities of different actions.
- On the other hand, the Critic model estimates the expected return or value of a given state. It also employs a sequential network architecture with linear layers. Dropout regularization is applied to enhance generalization, and a PReLU activation function introduces non-linearity to the network.
- The forward method in each model defines the forward pass of the network, where the input state is passed through the network layers to obtain the corresponding outputs.

3) Setting up the training environment for the Lunar Lander task using Gym's make function with the 'LunarLander-v2' environment.

```
Train_envir = gym.make('LunarLander-v2')
I_DIM = Train_envir.observation_space.shape[0]
H_DIM = 128
O_DIM = Train_envir.action_space.n

Actor_ = Actor(I_DIM, O_DIM, H_DIM)
Opt_Actor = optim.Adam(Actor_.parameters())
Critic_ = Critic(I_DIM, H_DIM)
Opt_Critic = optim.Adam(Critic_.parameters())
```

- The observation space shape of the environment is obtained and stored in I_DIM. The hidden dimension is set to 128 (H_DIM), and the output dimension is determined by the number of possible actions in the environment (O_DIM).
- An instance of the Actor model is created, passing the input dimension, output dimension, and hidden dimension as arguments. The Actor_ instance represents the neural network model responsible for mapping the spacecraft's state to action probabilities.
- The Adam optimizer (Opt_Actor) is initialized to optimize the parameters of the Actor_ model during training.
- Similarly, an instance of the Critic model is created, passing the input dimension and hidden dimension as arguments. The Critic_ instance estimates the expected return or value of a given state.
- The Adam optimizer (Opt_Critic) is initialized to optimize the parameters of the Critic_ model during training.
- The learning rate used by the Adam optimizers is 0.001 by default.

4) Including several hyperparameters used in a reinforcement learning setting.

```
MAX_EPISODES = 2000
DISCOUNT_FACTOR = 0.99
EPSILON_DECAY = 0.3
PPO_STEPS = 7
REWARD_THRESHOLD = 200
```

- MAX_EPISODES: Determines the maximum number of episodes or iterations for training the agent. An episode typically represents a complete interaction between the agent and the environment.
- DISCOUNT_FACTOR: Represents the discount factor used to discount future rewards in reinforcement learning algorithms. It determines the importance of immediate rewards compared to future rewards. A value of 0.99 indicates that future rewards are considered with a slight discount.
- EPSILON_DECAY: Controls the rate at which the exploration-exploitation trade-off changes during training. It determines the decay rate of the exploration factor (epsilon) used in epsilon-greedy policies. A higher value leads to a slower decay, meaning the agent explores the environment for a longer period before shifting towards exploitation.
- PPO_STEPS: Defines the number of steps or iterations performed in each update of the PPO algorithm. A higher value allows for a more thorough exploration of the policy space but may require more computational resources.
- REWARD_THRESHOLD: Sets a threshold value for the reward. It represents the desired reward level that the agent aims to achieve in the environment. Once the agent surpasses this threshold, it is considered to have successfully solved the task.

5) Implementing a training loop for a reinforcement learning agent using the PPO algorithm.

```
Rewards, Policy_Losses, Value_Losses, Mean_Rewards = [], [], [], []

for epi in range(1, MAX_EPISODES + 1):
    States, Actions, Log_prob_actions, Values, rewards = [], [], [], [], []
    Done = False
    epi_reward = 0
    state, step = Train_envir.reset()

    while not Done:
        state = torch.FloatTensor(state).unsqueeze(0)
        States.append(state)

        Pred_action = Actor_(state)
        distribution = dist.Categorical(Pred_action)
        Action = distribution.sample()
        Log_prob_action = distribution.log_prob(Action)

        Pred_Value = Critic_(state)

        state, reward, terminated, trunked, step = Train_envir.step(Action.item())
        if terminated or trunked:
            Done = True
        else:
            Done = False

        Actions.append(Action)
        Log_prob_actions.append(Log_prob_action)
        Values.append(Pred_Value)
        rewards.append(reward)

        epi_reward += reward

    Returns, Advantage = [], 0
    for r in reversed(rewards):
        Advantage = r + Advantage * DISCOUNT_FACTOR
        Returns.insert(0, Advantage)
    Returns = torch.tensor(Returns)
    Returns = (Returns - Returns.mean()) / Returns.std()

    Values = torch.cat(Values).squeeze(-1)
    advantages = Returns - Values
    advantages = (advantages - advantages.mean()) / advantages.std()
```

```
States = torch.cat(States)
Actions = torch.cat(Actions)
Log_prob_actions = torch.cat(Log_prob_actions).detach()
advantages = advantages.detach()

for step in range(PPO_STEPS):

    Pred_action = Actor_(States)
    distribution = dist.Categorical(Pred_action)
    new_log_prob_actions = distribution.log_prob(Actions)
    policy_ratio = (new_log_prob_actions - Log_prob_actions).exp()

    policy_loss1 = policy_ratio * advantages
    policy_loss2 = torch.clamp(policy_ratio, min = 1.0 - EPSILON_DECAY,
                               max = 1.0 + EPSILON_DECAY) * advantages
    policy_loss = -torch.min(policy_loss1, policy_loss2).sum()
    Pred_Value = Critic_(States).squeeze(-1)
    value_loss = func.smooth_l1_loss(Returns, Pred_Value).sum()

    Opt_Actor.zero_grad()
    Opt_Critic.zero_grad()
    policy_loss.backward()
    value_loss.backward()
    Opt_Actor.step()
    Opt_Critic.step()

Rewards.append(epi_reward)
Policy_Losses.append(policy_loss.item())
Value_Losses.append(value_loss.item())

if len(Rewards) >= 100:
    Latest_100_Mean = sum(Rewards[-100:]) / 100
    Mean_Rewards.append(Latest_100_Mean)
    if epi % 10 == 0:
        print(f'Episode {epi:3}')
        print(f'Reward =  {epi_reward}, Mean of last 100 episodes = {Latest_100_Mean}')

    if Latest_100_Mean > REWARD_THRESHOLD:
        print(f"Stopped Training: Mean of last 100 episode rewards
              has exceeded 200 ({Latest_100_Mean})!")
        break
```

```
torch.save(Actor_.state_dict(), 'trained_actor_model.pth')

with open('Rewards.pkl', 'wb') as file:
    pickle.dump(Rewards, file)
with open('Policy_Losses.pkl', 'wb') as file:
    pickle.dump(Policy_Losses, file)
with open('Value_Losses.pkl', 'wb') as file:
    pickle.dump(Value_Losses, file)
with open('Mean_Rewards.pkl', 'wb') as file:
    pickle.dump(Mean_Rewards, file)
```

- Initialization: Four empty lists are created: Rewards, Policy_Losses, Value_Losses, and Mean_Rewards. These lists will store the rewards obtained in each episode, policy losses, value losses, and the mean rewards over the last 100 episodes, respectively.
- Training Loop: The code enters a loop that iterates over the specified number of episodes (MAX_EPISODES). In each episode, the environment is reset, and the initial state and step are obtained.
- Episode Loop: Inside the episode loop, the agent interacts with the environment until the episode is done. The state is converted to a PyTorch tensor and stored in the States list. The actor model (Actor_) is used to predict the action probabilities (Pred_action) based on the current state. The distribution of actions is obtained using dist.Categorical, and an action is sampled from this distribution. The log probability of the selected action (Log_prob_action) is also stored.
- Environment Step: The selected action is applied to the environment using Train_envir.step(Action.item()). The resulting next state, reward, termination flag, and step are obtained. If the episode is terminated or truncated (a predefined step limit is reached), the Done flag is set to True; otherwise, it is set to False.
- Logging: The selected action, log probability, predicted value, and reward are appended to their respective lists (Actions, Log_prob_actions, Values, rewards). The episode reward (epi_reward) is updated.
- Advantage Calculation: After the episode is completed, advantages and returns are calculated. The advantages are computed by subtracting the predicted values from the discounted returns. Both advantages and returns are normalized by subtracting the mean and dividing by the standard deviation.

- PPO Update: A loop is performed PPO_STEPS times to update the actor and critic models using the PPO algorithm. In each step, the actor model predicts new action probabilities (Pred_action) for the collected states. The log probabilities of the actions are computed, and the policy ratio is calculated. The PPO loss terms (policy_loss1 and policy_loss2) are calculated based on the policy ratio and advantages. The critic model predicts values for the states, and the value loss is computed using the smoothed L1 loss.
- Backpropagation and Optimization: The gradients of the policy and value losses are computed and used to update the actor and critic models, respectively, through their respective optimizers (Opt_Actor and Opt_Critic).
- Logging and Termination Check: The episode reward, policy loss, and value loss are appended to their respective lists (Rewards, Policy_Losses, Value_Losses). If the length of the rewards list is greater than or equal to 100, the mean reward over the last 100 episodes is computed and appended to the Mean_Rewards list. If the mean reward exceeds the specified threshold (REWARD_THRESHOLD), the training is stopped, indicating that the agent has solved the task.
- Saving Results: Finally, the trained actor model's state dictionary is saved to a file named 'trained_actor_model3.pth'. The rewards, policy losses, value losses, and mean rewards lists are pickled and saved to separate files for later analysis.

6) Running the agent with the loaded trained actor model on a test environment and evaluating its performance

```
import os
os.environ["IMAGEIO_FFMPEG_EXE"] = r"/Users/ishaankchopra/Downloads/ffmpeg"
from gym.wrappers.monitoring.video_recorder import VideoRecorder

Actor_ = Actor(I_DIM, O_DIM, H_DIM)
Actor_.load_state_dict(torch.load('trained_actor_model.pth'))
Actor_.eval()

Test_envir = gym.make('LunarLander-v2',render_mode="rgb_array")

vid = VideoRecorder(Test_envir, path=f"LunarLander-v2.mp4")

TEST_EPISODES = 10
for epi in range(1, TEST_EPISODES + 1):
    state, step = Test_envir.reset()
    Done = False
    epi_reward = 0

    while not Done:

        Test_envir.render()
        vid.capture_frame()

        state = torch.FloatTensor(state).unsqueeze(0)
        with torch.no_grad():
            Action_probability = Actor_(state)
        distri = dist.Categorical(Action_probability)
        Action = distri.sample()

        state, reward, terminated, trunked, step = Test_envir.step(Action.item())
        if terminated or trunked:
            Done = True
        else:
            Done = False
        epi_reward += reward

    print(f'Test Episode {epi}:')
    print(f'Total Reward = {epi_reward}')

vid.close()
Test_envir.close()
```

- Load the Trained Actor Model: Initialize an Actor_ object and load the trained weights from the file 'trained_actor_model3.pth' using torch.load(). Set the model to evaluation mode using Actor_.eval().
- Initialize the Test Environment: Create the test environment using gym.make() with the environment name 'LunarLander-v2' and render mode set to "rgb_array".
- Video Recording Initialization: Set up video recording using gym.wrappers.monitoring.video_recorder.VideoRecorder(). Pass the test environment (Test_envir) as an argument and specify the output path for the video.
- Set Test Episode Count: Define the number of test episodes with the variable TEST_EPISODES.
- Episode Loop: Inside the episode loop, reset the environment and initialize variables. Interact with the environment until the episode is done.
- Render and Capture Video: Render the test environment using Test_envir.render(). Capture the current frame and record it in the video using vid.capture_frame().
- Obtain Action from Actor Model: Convert the state to a PyTorch tensor and pass it through the actor model (Actor_) to obtain the action probabilities. Sample an action from the probability distribution.
- Environment Step: Apply the selected action to the test environment using Test_envir.step(Action.item()). Obtain the next state, reward, termination status, and step count.
- Update Episode Reward: Update the episode reward by adding the reward obtained from the environment.
- Print Test Results: After each test episode, print the episode number and the total reward obtained.
- Close Video Recording and the Environment: Finalize the video recording by calling vid.close(). Close the test environment using Test_envir.close().

III. Experiments

I conducted a series of experiments to evaluate the performance of the trained model. As the LunarLander-v2 environment is considered to be a rather preliminary task to solve, the rewards, average rewards, and episodes needed to solve it are the key metrics to take into account. The evaluation process involved the following steps:

1) **Testing the Model**: As seen in the last part of the code (Method 6)), I tested the model by running it for 10 episodes in the LunarLander-v2 environment and capturing the video of the simulation. During each episode, the model interacted with the environment and selected actions based on its learned policy. I observed that the model consistently achieved rewards greater than 200 in all episodes, indicating its proficiency in safely navigating and landing the lunar lander at the target

   Here is the output of my test:

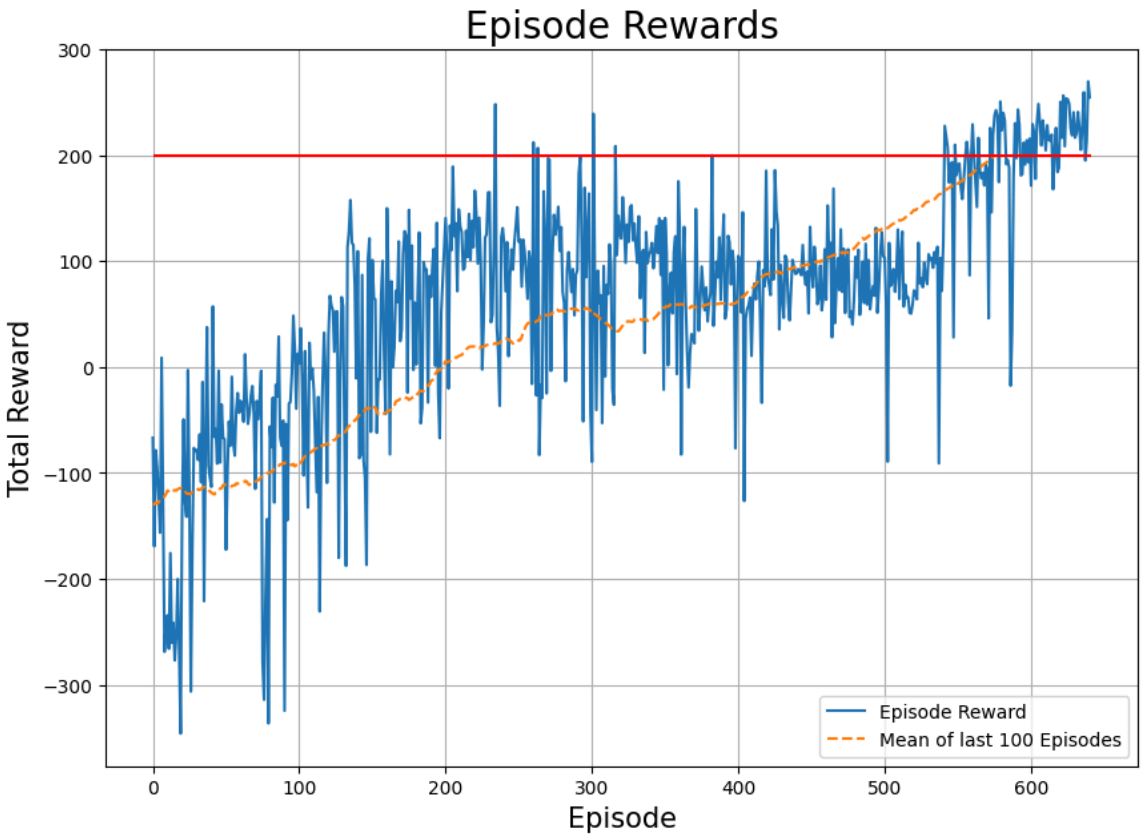
```
Test Episode 1:
Total Reward = 267.30814472835607
Test Episode 2:
Total Reward = 242.71596712062657
Test Episode 3:
Total Reward = 220.7320547550753
Test Episode 4:
Total Reward = 232.9768500509414
Test Episode 5:
Total Reward = 244.83975732384047
Test Episode 6:
Total Reward = 244.66665154173165
Test Episode 7:
Total Reward = 232.00510084643835
Test Episode 8:
Total Reward = 230.99215636264782
Test Episode 9:
Total Reward = 215.41148015144609
t:  51%|█████     | 2432/4803 [00:15<00:02, 857.57it/s, now=None]
Test Episode 10:
Total Reward = 222.55191713839767
Moviepy - Building video LunarLander-v2.mp4.
Moviepy - Writing video LunarLander-v2.mp4

t:  51%|█████     | 2432/4803 [00:20<00:02, 857.57it/s, now=None]
Moviepy - Done !
Moviepy - video ready LunarLander-v2.mp4
t:  51%|█████     | 2432/4803 [00:22<00:02, 857.57it/s, now=None]
```

   The link for my video output: [LunarLander-v2.mp4]

2) **Analysis of Episode Rewards**: To gain insights into the rewards obtained during the training episodes, I plotted the episode rewards over time. I visualized the rewards using a line plot. Additionally, I calculated the mean rewards of the last 100 episodes and presented them as a dashed line on the graph. This analysis allowed me to observe the trend of episode rewards and confirm that the model consistently surpassed the predefined reward threshold of 200 at the last episodes of training. These results demonstrate the effectiveness of the model in successfully solving the LunarLander-v2 task.

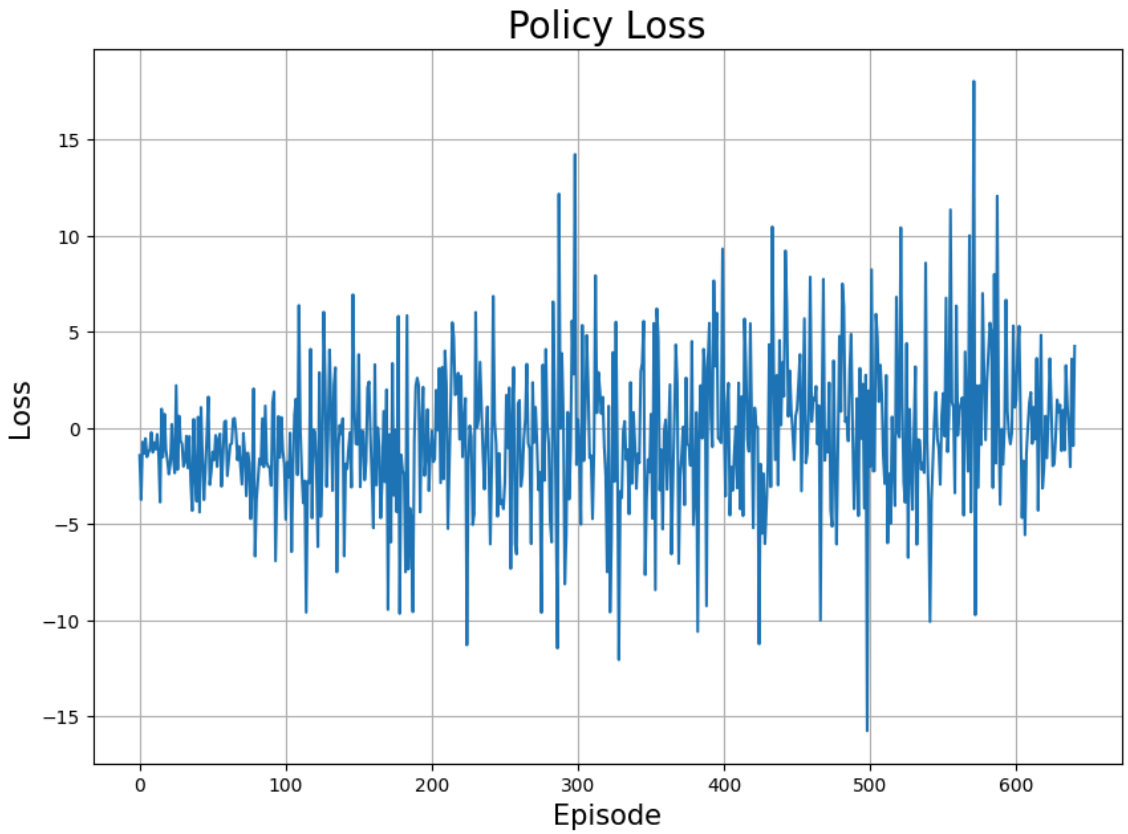
   Here is the plot:



   As we see from the graph, the model was able to achieve and maintain a reward level that surpasses the predefined reward threshold of 200 at around episode 640 / 2000.

3) **Analysis of Policy Loss**: I also examined the policy loss throughout the training process. The policy loss reflects the disparity between the model's predicted actions and the desired actions. I plotted the loss values as a function of the episode number. This analysis provided valuable insights into the convergence of the model's policy predictions.

   Here is the plot:

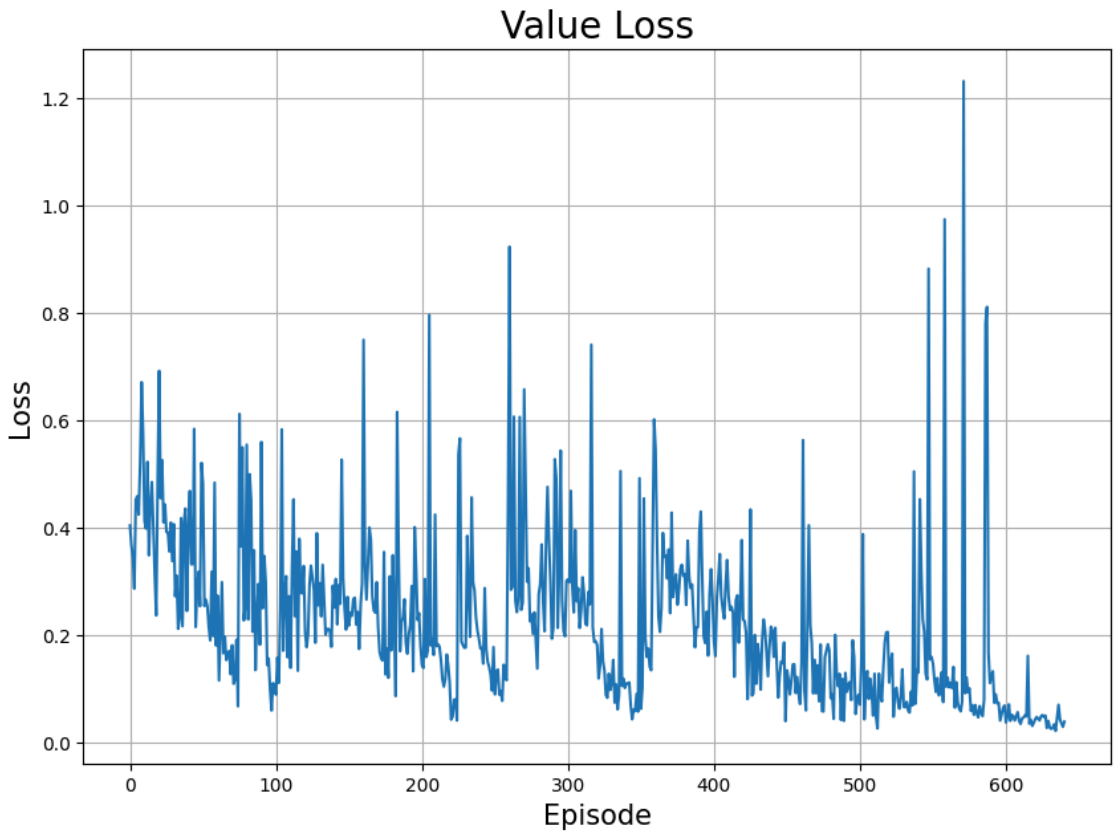


   A reduced policy loss indicates that the model's predicted actions are approaching the desired actions. This means the model is improving its decision-making skills and choosing the best actions to increase rewards in the LunarLander-v2 setting.

   The policy loss declines towards the end of the plot, showing that the model is consistently enhancing its policy to match the desired actions during training. This enhancement in policy directly leads to the model's capability to consistently attain high rewards and effectively maneuver and land the lunar lander.

4) **Analysis of Value Loss**: In addition, I investigated the value loss, which measures the discrepancy between the model's predicted state values and the true state values. I plotted the value loss history against the episode number. This analysis allowed me to assess the convergence of the value function and determine the model's progress in approximating the true state values.

   Here is the plot:

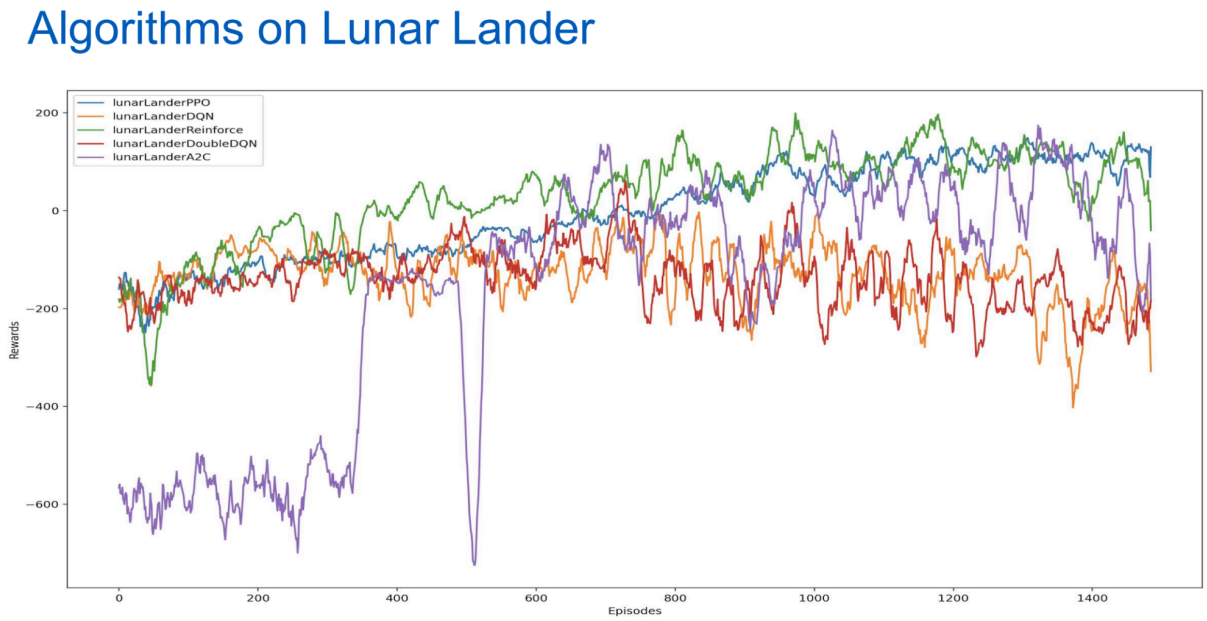


   It can be inferred that there is a decreasing trend in value loss, indicating that the model is progressively converging towards more accurate value estimations. This convergence is a positive indication of the model's learning progress and its ability to understand the rewards and outcomes associated with different states.

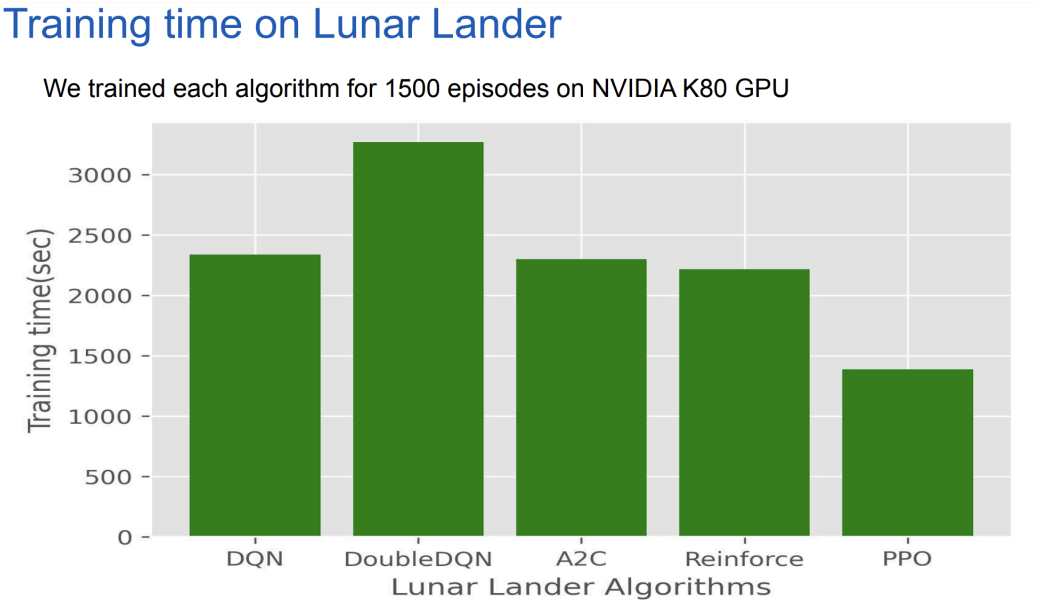
   By minimizing the value loss, the model is improving in its ability to make accurate predictions and comprehend the dynamics of the environment. Enhancing this is essential for making well-informed decisions and attaining improved outcomes in the LunarLander-v2 mission.

Comparison with other algorithms:

I also researched and compared where the PPO Algorithm stands to other existing algorithms in regard to the performance and efficiency of the models.



As we see, the PPO algorithm has done one of the best jobs among other algorithms. It has achieved a mean reward above 200, all while having a good trajectory without much oscillation, unlike other algorithms like Reinforce and A2C.



We also see that the PPO algorithm is the most efficient algorithm because it takes the least amount of time as the complexity of the environment increases.

Reference: Vamsi Krishna, V., & Yarram, S. (2020). Comparison of reinforcement learning algorithms. Retrieved from https://cse.buffalo.edu/~avereshc/rl_fall20/Comparison_of_RL_Algorithms_vvelivel_sudhirya.pdf

Conclusion:

In conclusion, the results achieved by the PPO Actor-Critic model in the LunarLander-v2 environment are quite impressive. The model has consistently demonstrated proficiency in navigating the environment and achieving accurate landings of the lunar lander. This accomplishment reflects the model's ability to learn and adapt to the complexities of the task.

Moreover, the model has consistently surpassed the specified benchmark of obtaining rewards exceeding 200. This showcases the model's capacity to efficiently increase rewards and make ideal choices in order to reach positive results. The model has proven its effectiveness in achieving the task goals by consistently exceeding the reward threshold.

The model's success is evidence of the rigorous training and fine-tuning process it underwent. Through iterative learning, the model assimilated knowledge from past experiences and refined its strategies to improve its performance in the LunarLander-v2 environment.