REPORT

EXPERIMENT-1

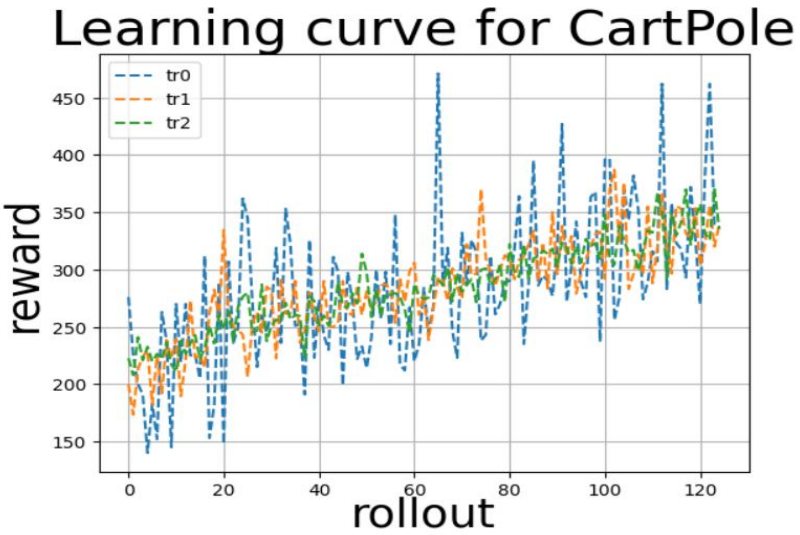
1. **Take snapshots from your code that show your implementation of eq 6, 7, 8 and include in the report.**

```
def estimate_loss_function(self, trajectory):
    loss = list()
    # print(trajectory)
    for t_idx in range(self.params['n_trajectory_per_rollout']):
        if self.params['reward_to_go']:
            returns = apply_discount(trajectory['reward'][t_idx])
        else:
            returns = apply_reward_to_go(trajectory['reward'][t_idx])

        log_prob = sum(trajectory['log_prob'][t_idx]) / self.params['n_trajectory_per_rollout']
        advantage = sum(returns)
        loss.append(log_prob * advantage)
    loss = torch.stack(loss).mean()

    return loss
```

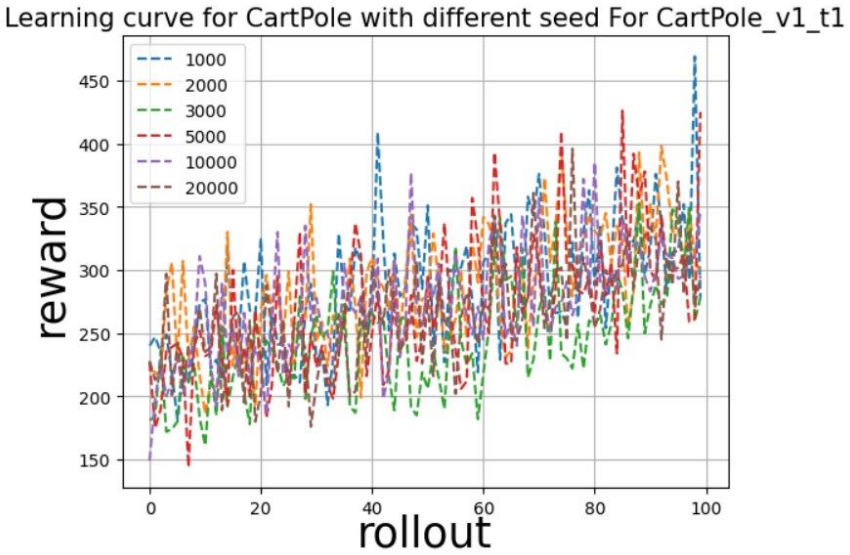
2. **Create a graph that compares the learning curve from the trail above. Label the curves as t0,t1,t2.**

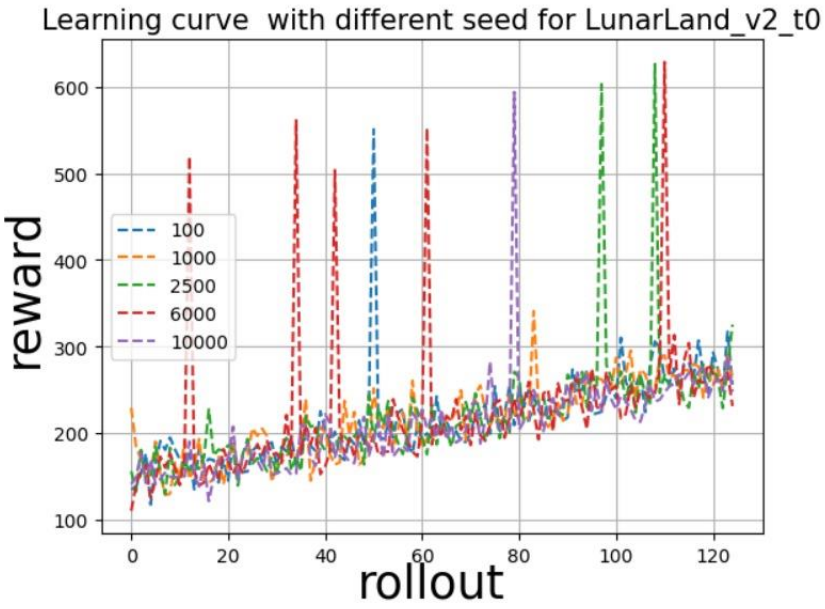


3. **Answer the following questions.**
   a) **Which version of PG is best performing (among return-based, reward-to-go-based and discount-reward-based)?**
   b) **BONUS: If u run the same trails multiple times (with different random seeds) and then plot the learning curves together, it should give a visual estimation of average-trajectory rewards variance (higher spread means higher variance and vice-versa). Can u generate such plots (one for each version) and comment on the resulting variance (which one has higher variance or lower variance)?**

   3 a): The reward-to-go-based policy gradient tends to perform better in environments with sparse rewards.
   In reward-to-go-based policy gradient, the gradient of the policy is estimated based on the sum of rewards obtained from the current time-step until the end of the episode. This approach encourages the agent to focus on actions that lead to a high cumulative reward, which can be particularly effective in environments where rewards are sparse and delayed.

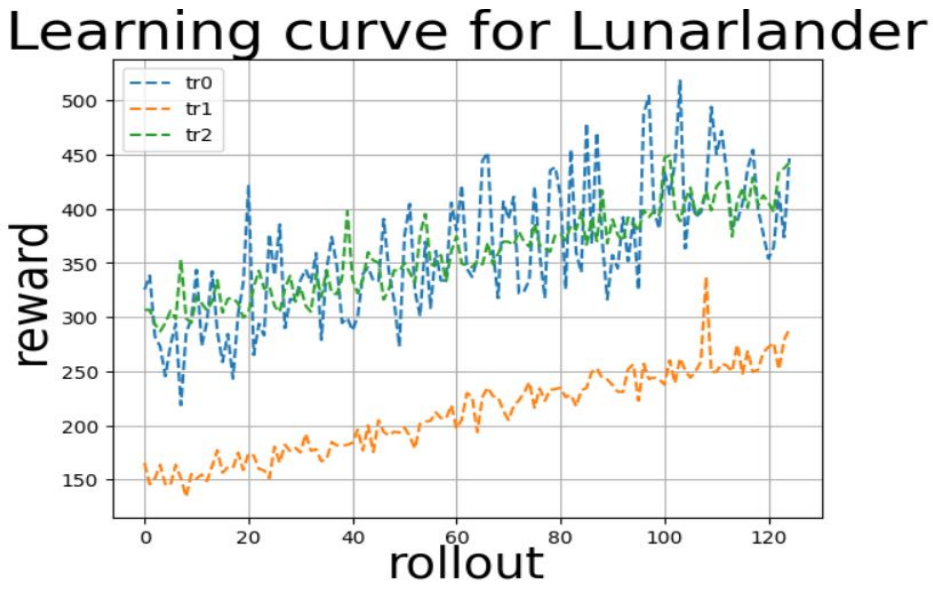
   3 b):





EXPERIMENT-2

1. **Create a graph that compares the learning curves from the three trials above. Label the curves as t0,t1,t2.**



2 a): How does the number of trajectories per rollout effect learning performance? How can you justify this relationship?

The number of trajectories per rollout can have a significant impact on the learning performance in Lunar Lander. Generally, increasing the number of trajectories per rollout can lead to more stable and efficient learning, allowing the algorithm to explore the state space more thoroughly and converge to a better policy.

By increasing the number of trajectories per rollout, we effectively increase the number of samples used to estimate this gradient, which can reduce the variance of the estimates and improve the accuracy of the gradient.

GITHUB:
https://github.com/Sreemourya710/project.git

STUDENT NAME: SreeMourya Ravirala
STUDENT ID: 1002029130