Tarun Sharma

ts5098

Introduction

This homework is designed to follow up on the lecture about policy gradient algorithms. For this assignment, you will need to know about the basics of the policy gradient algorithm we talked about in class, specifically REINFORCE and PPO. If you have not already, we propose you brush up on the lecture notes.

You are ALLOWED to discuss this homework with your classmates at the level of general solution strategies or tips. However, any work that you submit must be entirely your own, which includes your own unique code and your own unique written answers.

You are NOT ALLOWED to use any large language models (LLMs) or other AI-assisted tools.

If you find errors in the homework assignment or have public questions, please post in the Campuswire “#questions-hw” channel or ask during instructor/TA office hours.

Points

Q1	REINFORCE plots	10 pts
Q2	Understanding REINFORCE	5 pts
Q3	PPO implementation & plots	10 pts
Q4	Understanding PPO vs. REINFORCE	5 pts
		30 pts total

Code folder

Find the folder with the provided code in the following google drive folder: https://drive.google.com/file/d/1cEQZFOdoJPv64xW5zQweAnYA2vQsP2VF/view?usp=sharing Download all the files in the same directory, and run the `run.py` file to run your code. You will have to complete the TODOs in `ppo/ppo.py` to complete this homework.

Thanks to Eric Yu for this code.

Environment

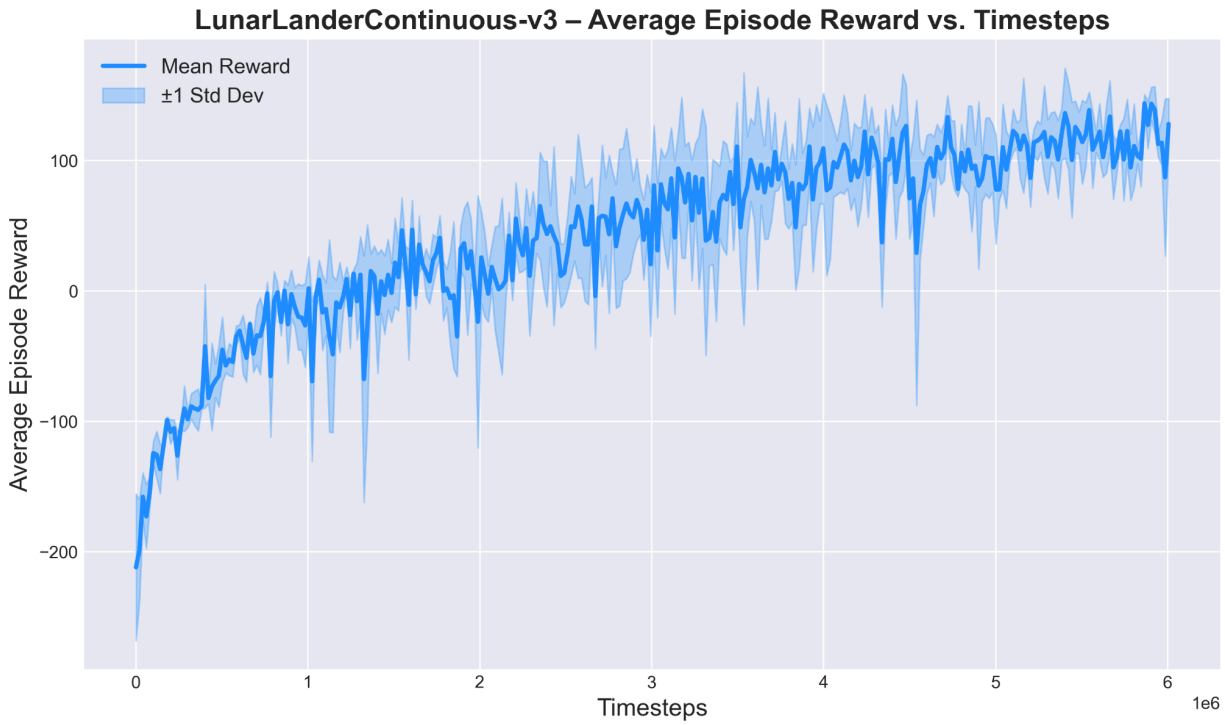
The environment we will use in this homework is built upon the Pong, Space Invaders, and Breakout environment from OpenAI gym Atari environments (https://gym.openai.com/envs/#atari). In this homework, we will attempt to learn these agents from state observations. We already have a working implementation of REINFORCE in the code folder, which you can run as `python run.py --alg reinforce --seed 42 --env <env_name>`.

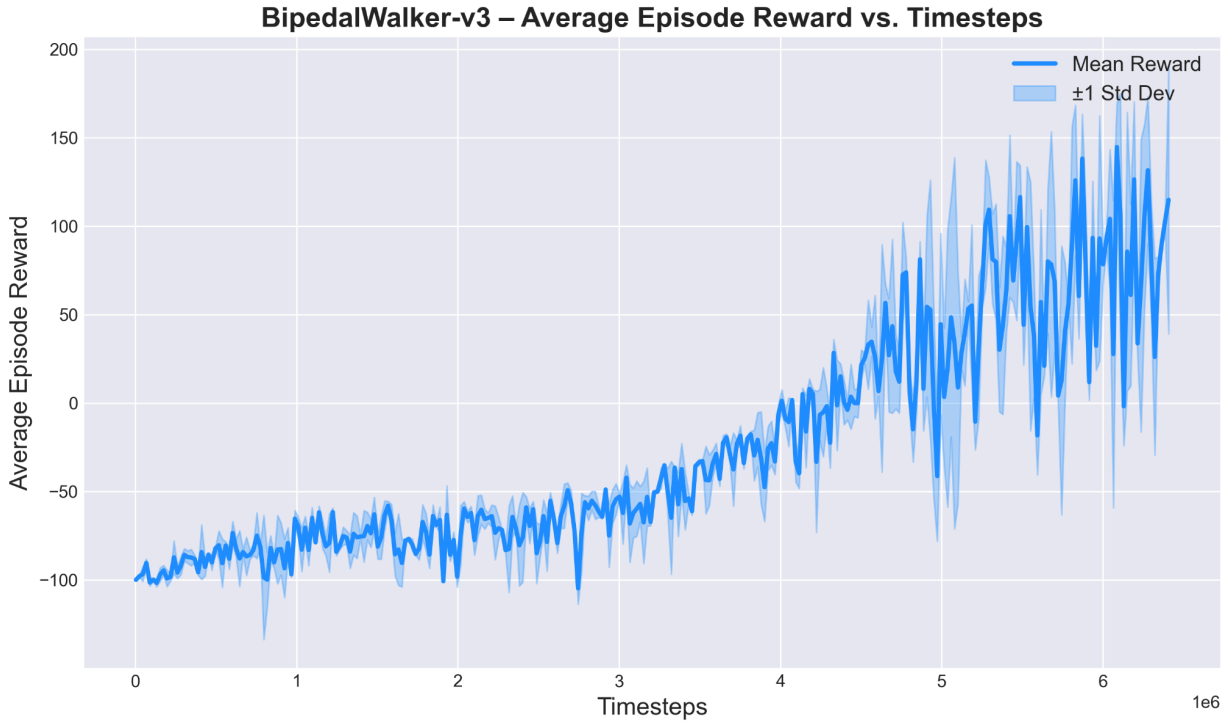
Deliverables:

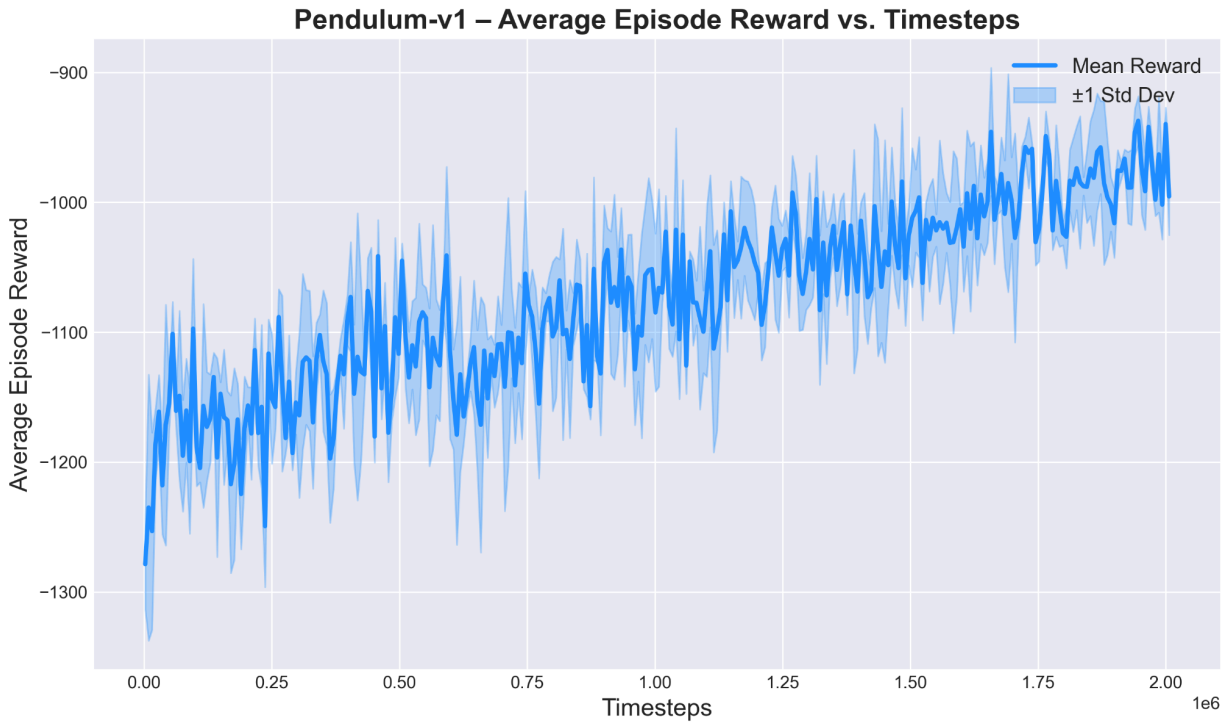
- Within PDF write-up, any written answers and plots.
- Within the compressed ZIP folder, the code files you changed (e.g. `ppo.py, run.py`). Please name your submission ZIP folder as *<net_ID>_assignment4.zip*

Questions

1. In the code folder, you will find already available code for running REINFORCE. Run this code on the following environments: Pendulum-v1, BipedalWalker-v3, and LunarLanderContinuous-v2. It is okay if REINFORCE does not perform as well in these environments, though the return should generally go up slowly. Generate the plot of average episodic return over training environment steps for these 3 environments over three different seeds, and create three plots that show the average performance of REINFORCE on each environment.


LunarLanderContinuous-v3 – Average Episode Reward vs. Timesteps
Mean Reward
±1 Std Dev
Average Episode Reward
100
0
−100
−200
0
1
2
3
4
5
6
Timesteps
1e6


BipedalWalker-v3 – Average Episode Reward vs. Timesteps
Mean Reward
±1 Std Dev
Average Episode Reward
200
150
100
50
0
−50
−100
0
1
2
3
4
5
6
Timesteps
1e6



2. Based on your plots, why do you think REINFORCE suffers in these environments?

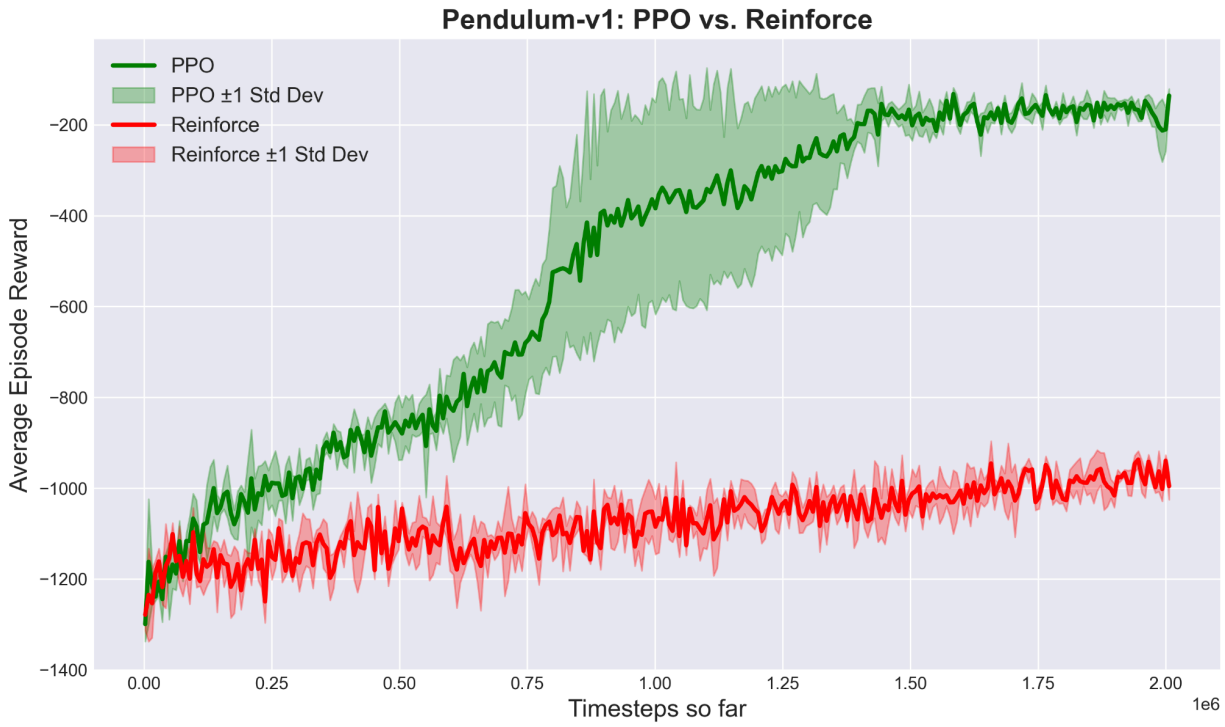
   A big reason REINFORCE tends to struggle in these environments is that it uses a plain Monte Carlo approach without a baseline. This causes its gradient estimates to have a lot of noise—especially in longer or more complex tasks like BipedalWalker-v3 or LunarLanderContinuous-v3, where rewards can come late or vary a lot from episode to episode. Because REINFORCE doesn't subtract a value function estimate (or advantage) from the rewards, each update it makes can be "swingy" and less stable. You can see that in the plots, where the returns take a while to climb and fluctuate a lot. Essentially, in these continuous-action, higher-dimensional environments, the variability of the updates slows down learning and makes it harder to converge on a good policy compared to methods like PPO, which incorporate additional tricks (like clipping and baselines) to reduce that variance.

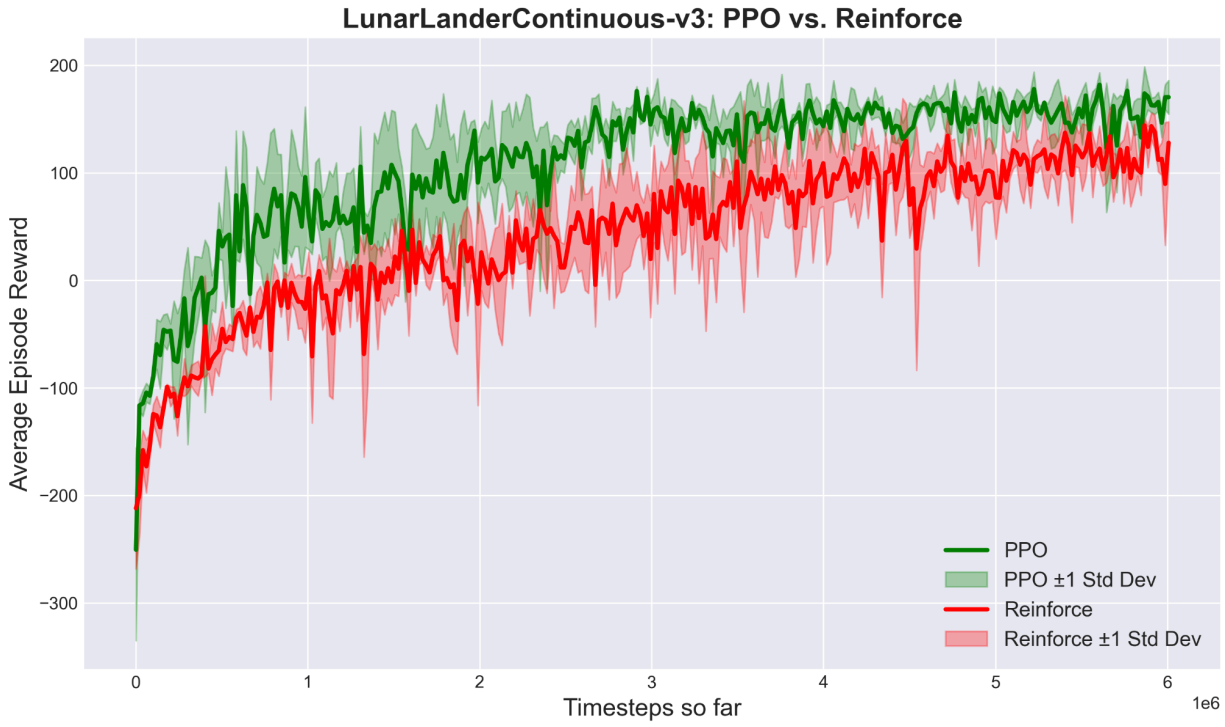
3. Now, complete the PPO code found in `ppo/ppo.py`. You will find a few different TODOs for you. Follow the original PPO pseudocode if you need to. Once again, use the previous three environments and three different seeds to plot your training rewards. **Clearly show the comparison between REINFORCE and PPO in your plots.**

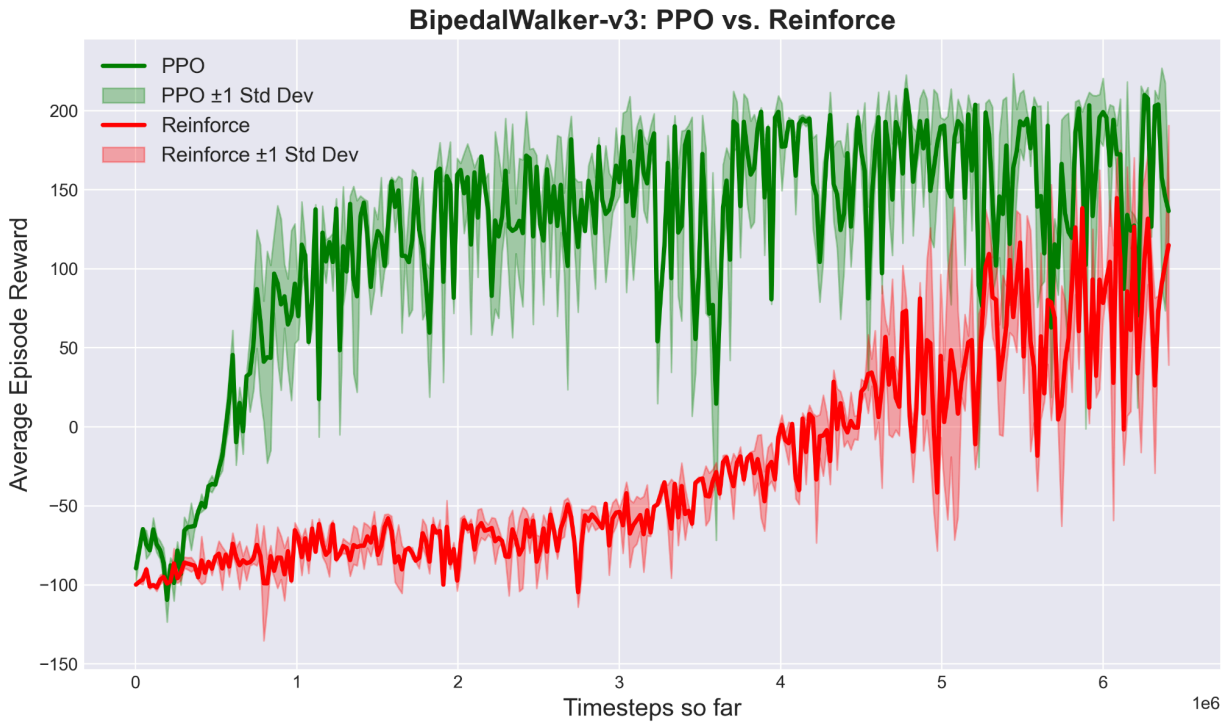
   Your expected mean performance should be AT LEAST:
   Pendulum: -400
   BipedalWalker: 125

LunarLanderContinuous: 100

Submit your writeup, along with your `ppo.py` file as your submission. If you change any of the hyperparameters, include `run.py` as well.







4. How does the performance of PPO and REINFORCE compare in these environments? Does one outperform the other? What factors do you think contribute to this difference?

In my experiments, **PPO consistently outperformed REINFORCE** on Pendulum-v1, BipedalWalker-v3, and LunarLanderContinuous-v3. The performance gap became especially clear in the more complex tasks (BipedalWalker-v3 and LunarLanderContinuous-v3), where REINFORCE tended to learn more slowly and with higher variance compared to PPO. Here are some key reasons that help explain why this happens:

1. **Variance Reduction through a Critic**
   One major advantage of PPO over plain REINFORCE is that PPO learns a separate critic (value function). This critic provides a baseline, which helps to reduce the variance in gradient estimates. In contrast, REINFORCE only uses the returns (i.e., sum of rewards) to estimate gradients, so it can be more prone to large fluctuations during training.

2. **Clipped Surrogate Objective**
   PPO uses a clipped objective function that prevents excessively large updates. This clipping mechanism stabilizes policy updates so the agent does not move too far from its old policy in a single step. REINFORCE has no such mechanism; it simply performs

gradient ascent on the sampled returns, so its updates can be more aggressive and less stable.

3. **Sample Efficiency**
   Because PPO reuses the collected data more effectively (via multiple epochs of updates on the same batch) and stabilizes those updates with clipping, it typically requires fewer total samples to achieve better performance. REINFORCE, on the other hand, uses each trajectory just once for its gradient update, making it more sample-inefficient.

4. **Hyperparameter Sensitivity**
   Both algorithms need careful hyperparameter tuning, but REINFORCE can be more sensitive to factors like the learning rate and batch size. When those are not well tuned, its performance suffers. PPO still benefits from good hyperparameters but often remains more robust across different setups because of its built-in checks (clipping, advantage normalization, etc.).

PPO generally outperforms REINFORCE because it uses a critic to reduce variance, uses a clipped objective to keep updates stable, and makes more efficient use of collected data. REINFORCE, being a simpler algorithm, can sometimes reach decent performance given enough training time and well-chosen hyperparameters, but in practice, it tends to lag behind PPO on these environments in both learning speed and final performance.