CS 5955/6955 Advanced Artificial Intelligence Spring 2025

Homework 5: Policy Gradient
Student: Simón Pedro González
UID: 1528314

1 Basic Policy Gradient Algorithm

A As mentioned in the tutorial, the “loss” value depends on both the collected data and the current parameters, and it does not directly measure performance. Therefore, we should pay attention to the average returns for each epoch (Fig. 1). We can see that the agent is learning, but the performance is not monotonically improving, there are some dips. This is because (a) batch gradient ascent does not guarantee improvement at each step, (b) the policy is stochastic and may sample poor actions and (c) PG is on-policy, so temporary policy degradation can produce a dip in performance in some epochs.

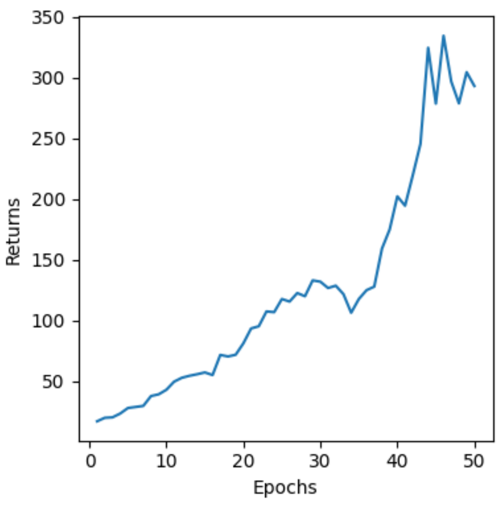
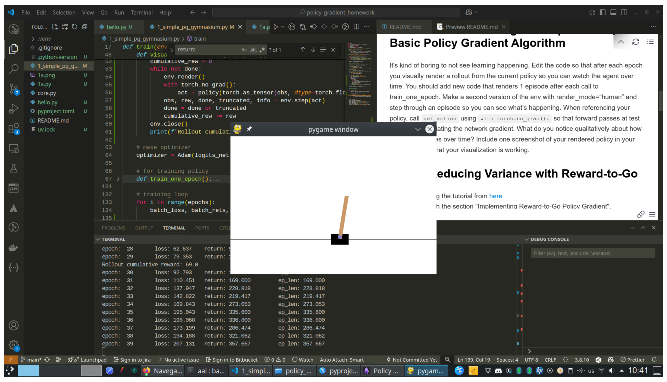


Figure 1: Basic policy gradient rewards per epoch.

B Qualitatively I can see that the agent is learning to balance the pole better, which is evident from the duration of the episode: the agent lasts longer without terminating the episode due to going out of bounds or the pole becoming too tilted. In Fig. 2, we can see the render.



```
def train(env
    def visua
            cumulative_rew = 0
            while not done:
                env.render()
                with torch.no_grad():
                    act = policy(torch.as_tensor(obs, dtype=torch.flo
                obs, rew, done, truncated, info = env.step(act)
                done = done or truncated
                cumulative_rew += rew
            env.close()
            print(f'Rollout cumulat

    # make optimizer
    optimizer = Adam(logits_net

    # for training policy
    def train_one_epoch(): ...

    # training loop
    for i in range(epochs):
        batch_loss, batch_rets,
```

```
epoch:  28      loss: 62.637    return:
epoch:  29      loss: 79.353    return:
Rollout cumulative reward: 69.0
epoch:  30      loss: 92.793    return:
epoch:  31      loss: 110.451   return: 169.000      ep_len: 169.000
epoch:  32      loss: 137.947   return: 228.818      ep_len: 228.818
epoch:  33      loss: 142.822   return: 219.417      ep_len: 219.417
epoch:  34      loss: 169.843   return: 273.053      ep_len: 273.053
epoch:  35      loss: 195.043   return: 335.600      ep_len: 335.600
epoch:  36      loss: 198.068   return: 336.800      ep_len: 336.800
epoch:  37      loss: 173.199   return: 286.474      ep_len: 286.474
epoch:  38      loss: 194.188   return: 321.062      ep_len: 321.062
epoch:  39      loss: 207.131   return: 357.667      ep_len: 357.667
```

Basic Policy Gradient Algorithm

It's kind of boring to not see learning happening. Edit the code so that after each epoch you visually render a rollout from the current policy so you can watch the agent over time. You should add new code that renders 1 episode after each call to train_one_epoch. Make a second version of the env with render_mode="human" and step through an episode so you can see what's happening. When referencing your policy, call `get_action` using `with torch.no_grad():` so that forward passes at test ating the network gradient. What do you notice qualitatively about how es over time? Include one screenshot of your rendered policy in your at your visualization is working.

educing Variance with Reward-to-Go

g the tutorial from here
h the section "Implementing Reward-to-Go Policy Gradient".

Figure 2: Render of Basic policy gradient on Cart-Pole.

2 Reducing Variance with Reward-to-Go

In Fig. 3, we can see the average return over five runs of Vanilla Policy Gradient (PG) and Reward-to-Go Policy Gradient (R2G-PG). R2G-PG not only learns faster, but also has lower variance across runs. As explained in the tutorial and in class, R2G-PG excludes rewards received before each action, so actions are reinforced based only on future rewards, which are the ones they are responsible for. This accurate credit assignment leads to lower variance in the gradient estimates and more stable learning.

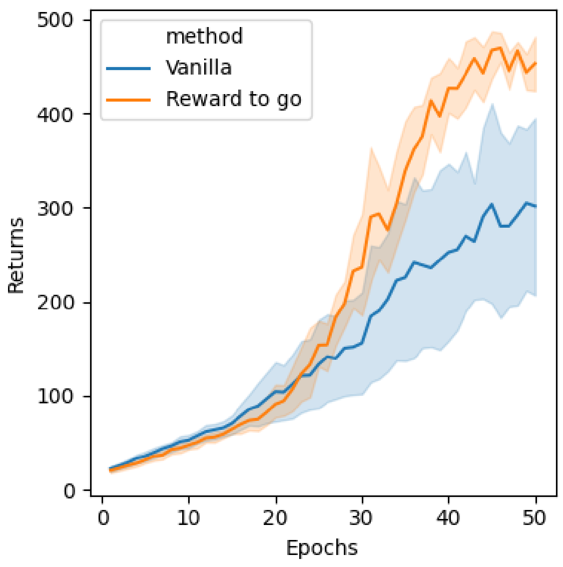
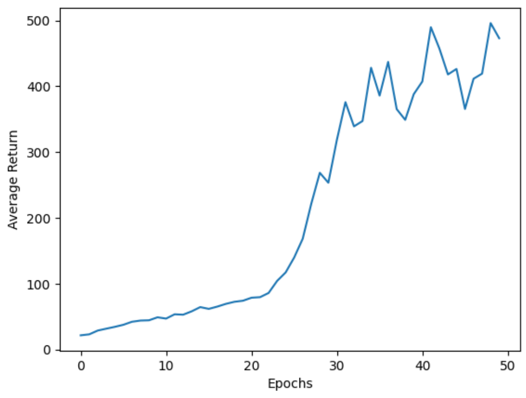


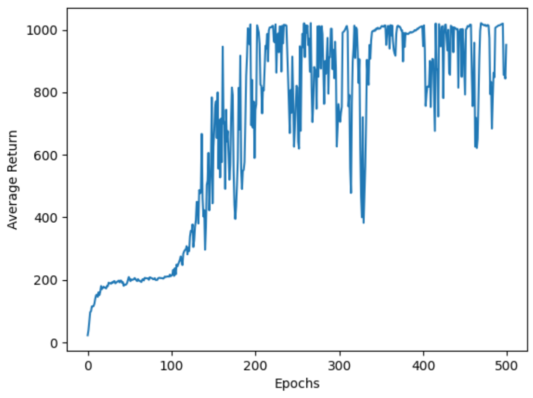
Figure 3: Returns per epoch for Vanilla PG and R2G-PG for 5 experiments each in the Cart-Pole environment.

3 Continuous Action

I implemented and tested the general-purpose R2G-PG algorithm and was able to replicate the results from the last section on the CartPole environment (Fig. 4a). For the continuous case, I tested the R2G-PG algorithm in the Hopper environment (Fig. 4b). Learning occurred; however, the policy appears to be stuck around 1000 return. The reward in the Hopper environment consists of a per-timestep "healthy" reward, a "forward" reward, and a "control cost". It seems the agent only learned to maintain balance, gathering all possible "healthy" rewards during the episode, but without making forward progress. Tweaking the hyperparameters did not help much in this case.



(a) Cart-Pole environment



(b) Hopper environment

Figure 4: General purpose R2G-PG implementation performance in two environments.

4 Generalized Advantage Estimation

I implemented GAE starting from the R2G-PG algorithm. The first significant change is the use of the value network to estimate the TD-errors $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, followed by the use of the `discount_cumsum` function to compute the advantage values A_t for each episode. The second change involves computing the policy loss using A_t, and finally training the value network with the rewards-to-go as targets using the `torch.nn.MSELoss` function, for 80–100 iterations. After tweaking the hyperparameters I compared the performance of GAE and the previous algorithms in the CartPole (Fig. 5) and Hopper (Fig. 6) environments. GAE outperformed Vanilla PG and R2G-PG. In the Hopper environment, GAE shows a slower start, but also a more stable improvement. By the 500th epoch, the agent started to learn how to move forward, occasionally reaching near-2000 reward in some episodes.

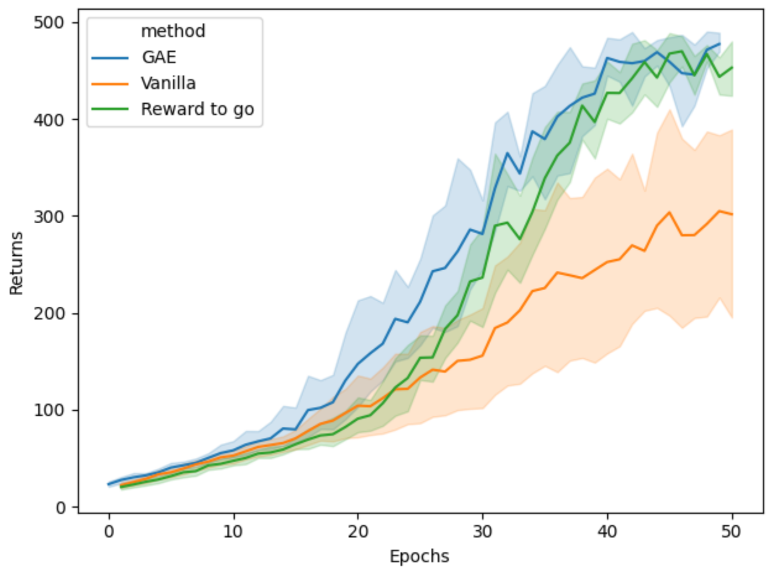


Figure 5: GAE, R2G-PG and Vanilla PG performance for five experiments each in the Cart-Pole environment.

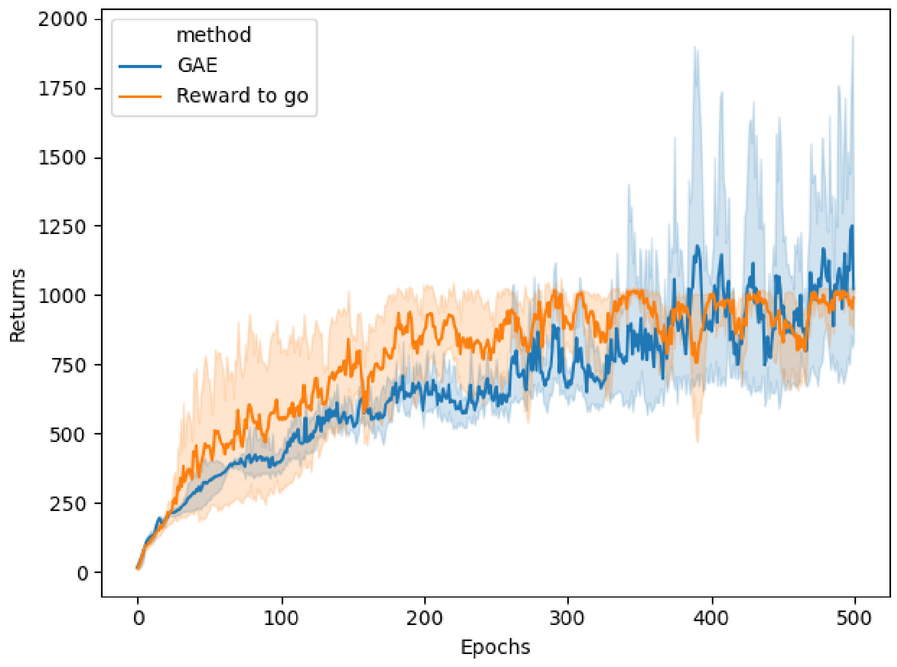


Figure 6: GAE and R2G-PG performance for five experiments each in the Hopper environment.

5 Proximal Policy Optimization

After checking the suggested PPO implementation, the changes required to adapt my GAE algorithm to PPO were minimal. First, I collected the $\log \pi(a_t|s_t)$ for each interaction. At the end of each episode, the policy was optimized for 80 iterations using the clipped objective discussed in class:

$$- \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a),\ \ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right)$$

The policy ratio was calculated using the log trick: $\exp\{\log \pi_\theta(a|s) - \log \pi_{\theta_k}(a|s)\}$. Policy optimization also included a KL divergence early stopping criterion based on the change in $\log \pi_\theta(a|s)$. The rest of the code remains unchanged from the GAE implementation. I ran my PPO implementation on the Cart-Pole (Fig. 7) and Hopper (Fig. 8) environments. In the Cart-Pole environment, performance improved dramatically, reaching the maximum return almost immediately. In the Hopper environment, PPO was the only algorithm that consistently achieved forward motion within 500 epochs. For comparison, I uploaded videos of R2G-PG (`https://drive.google.com/file/d/1DMRpUNTNTj8u7ei1PyjMcJvwYn22zUxH/view?usp=sharing`), and PPO (`https://drive.google.com/file/d/1UMe_T_3m-b1PLfdPQG82PE9czT41mtvh/view?usp=sharing`) in the Hopper environment, with 500 training epochs. We can see that R2G-PG is stuck collecting the "healthy" rewards, while PPO consistently jumps forward.

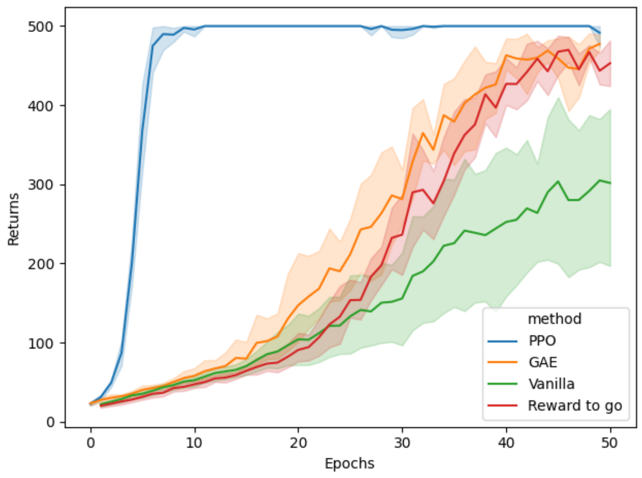


Figure 7: PPO performance compared to the other algorithms in the Cart-Pole environment for 5 experiments each.

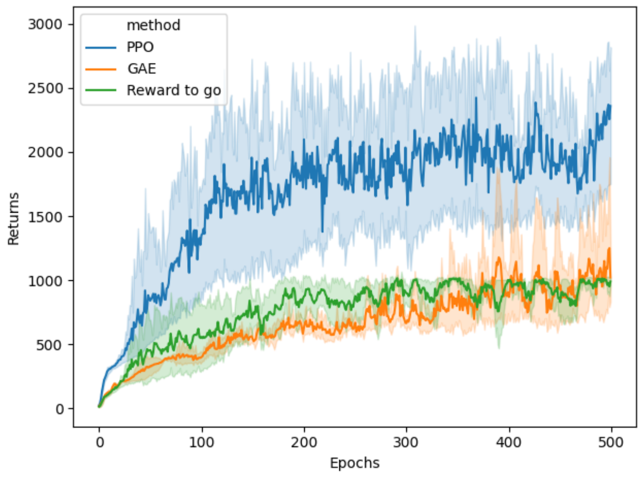


Figure 8: PPO performance compared to the other algorithms in the Hopper environment for 5 experiments each.