CS 6955
Project 5

By: Tyler Trotter

Part 1a: Report on what you see and what it means. Is the agent learning? What happens to it's performance over time? Is it monotonically improving? Briefly discuss if what you observe makes sense.

Using simplest formulation of policy gradient, the agent appears to be learning, though the loss spikes tremendously the episode length likewise increases (which is the measure of how long it stayed afloat). That said, the learning isn't great, lasting only a few seconds. It certainly isn't learning monotonically, as you can tell by the last number being less than the previous (48th and 49th epochs). This is because of the high variance is the basic policy gradient algorithm. Below is the table you should reference:

Epoch	Loss	Return	Ep_Len
0	20.298	22.249	22.249
1	22.929	25.892	25.892
2	24.558	28.770	28.770
3	26.159	30.804	30.804
4	30.333	36.362	36.362
⋮	⋮	⋮	⋮
44	226.834	340.200	340.200
45	221.227	320.706	320.706
46	213.571	326.500	326.500
47	218.290	321.562	321.562
48	204.340	295.059	295.059
49	182.761	279.368	279.368

Table 1: Training results over 50 epochs

Part 1b: What do you notice qualitatively about how its policy changes over time? Include one screenshot of your rendered policy in your report to show that your visualization is working.

Over time, it is clear that the agent is learning to better balance the stick on the cart. At the beginning, it is pretty lackluster and falls easily. Towards the end of the training, however, it can stay afloat for much longer – nearly the whole time. Below is a screenshot of my agent learning a policy.

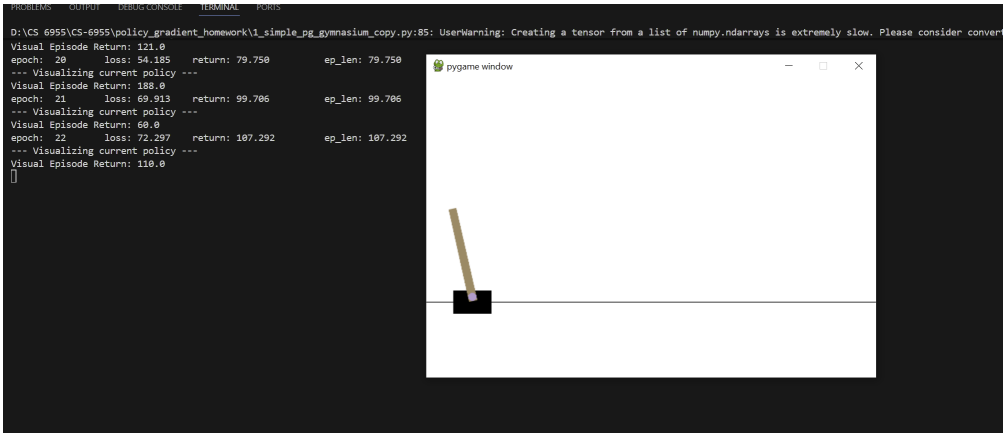

Figure 1: 1b Screenshot

Part 2: Run your new code and compare the performance with your older version. Run each method 5 times (make sure you add code to save the data that is currently just being printed to the screen) and include a plot of the two methods' learning curve (average return vs. timesteps). Briefly discuss what your experiment shows. Do you notice any differences? Is one approach better? Why or why not?

It is obvious that the reward to go is objectively better. The experiment demonstrates as much. The primary difference is the superior performance of the reward-to-go approach because it is more flexible at adapting to incoming rewards and the Reward-to-go approach is superior because it addresses the credit assignment problem. By only considering rewards obtained after an action is taken, we remove the 'noise' of rewards that occurred in the past, which the current action could not have possibly influenced. This significantly reduces variance in the gradient estimate

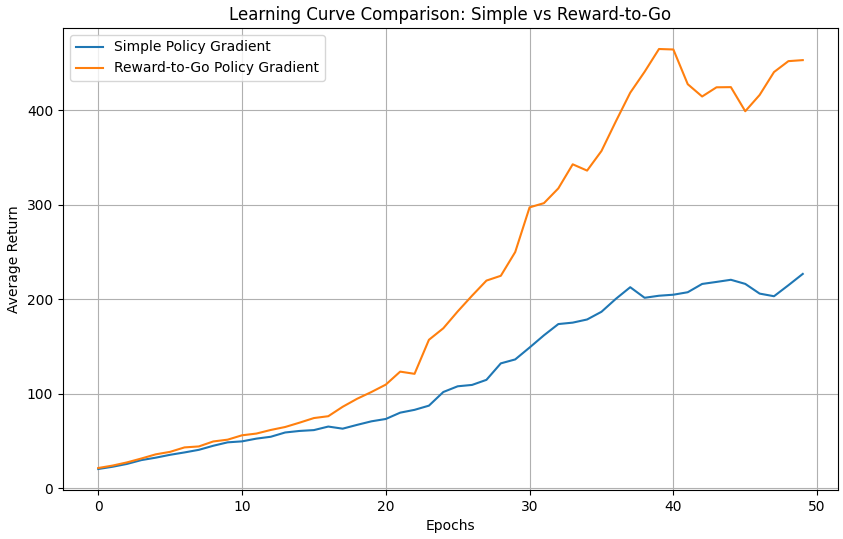

Figure 2: Comparison between Simple Policy Gradient and Reward-To-Go Policy Gradient

Part 3: Report on what environment you chose and the results of using your code.

I chose InvertedPendulum-v4 as my primary continuous environment because it provides a clear baseline for testing the transition from discrete distributions to continuous Gaussian distributions. I learned that while the logic remains similar to the discrete case, the policy now outputs a mean and standard deviation for a Gaussian distribution, which allows the agent to fine-tune the exact amount of torque applied to the pendulum.

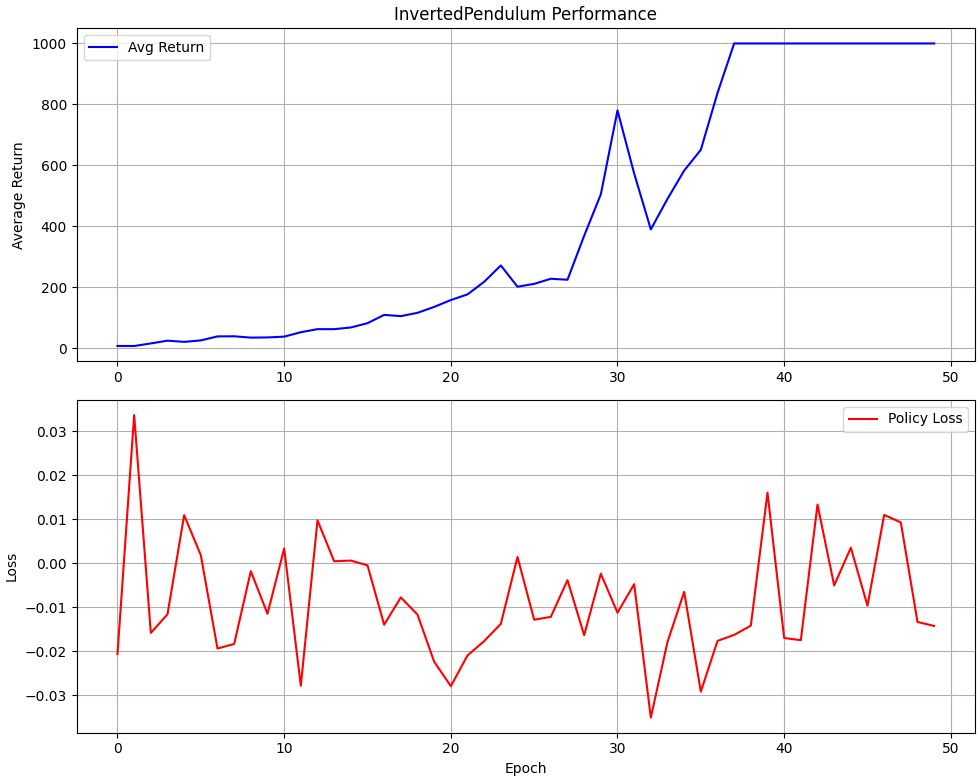

Figure 3: Results on Updated Gymnasium

Extra Credit A: Test out your implementation on both CartPole and a continuous action environment of your choice. Report and discuss how your policy performs. How do your results compare to performance without a baseline?

As seen in the results for the continuous Pendulum task, when compared to previous methods, the critic's loss steadily decreased as it learned the environment's values. This provided a stable floor for the actor, allowing the policy to converge on a smooth balancing strategy in fewer timesteps than the reward-to-go version. To interpret this, the agent's movements shifted from frantic, high-frequency adjustments to calm, subtle corrections–an improved policy.

Epoch: 1	Pi-Loss: -0.0456	AvgReturn: 11.04
Epoch: 2	Pi-Loss: -0.0567	AvgReturn: 11.88
Epoch: 3	Pi-Loss: -0.0489	AvgReturn: 12.50
Epoch: 4	Pi-Loss: -0.0464	AvgReturn: 14.18
Epoch: 5	Pi-Loss: -0.0164	AvgReturn: 16.07
Epoch: 6	Pi-Loss: -0.0424	AvgReturn: 17.87
Epoch: 7	Pi-Loss: -0.0222	AvgReturn: 19.28
Epoch: 8	Pi-Loss: -0.0320	AvgReturn: 21.64
Epoch: 9	Pi-Loss: -0.0225	AvgReturn: 23.05
Epoch: 10	Pi-Loss: -0.0109	AvgReturn: 25.95
⋮		⋮
Epoch: 40	Pi-Loss: -0.0113	AvgReturn: 135.00
Epoch: 41	Pi-Loss: -0.0109	AvgReturn: 128.97
Epoch: 42	Pi-Loss: -0.0103	AvgReturn: 130.59
Epoch: 43	Pi-Loss: -0.0121	AvgReturn: 140.08
Epoch: 44	Pi-Loss: -0.0170	AvgReturn: 126.40
Epoch: 45	Pi-Loss: -0.0016	AvgReturn: 132.84
Epoch: 46	Pi-Loss: -0.0122	AvgReturn: 125.15
Epoch: 47	Pi-Loss: 0.0165	AvgReturn: 136.08
Epoch: 48	Pi-Loss: -0.0069	AvgReturn: 148.09
Epoch: 49	Pi-Loss: -0.0004	AvgReturn: 140.35
Epoch: 50	Pi-Loss: -0.0017	AvgReturn: 143.22

Extra Credit B:
PPO builds upon the advantage actor critic framework by introducing a clipped surrogate objective, as they call it. While the basic policy gradient is sensitive to step size and can suffer from huge performance collapses if a single update moves the policy into a bad region of the parameter space, PPO guarantees stability by limiting how much the policy can change in one update.

Additionally, PPO tracks the ratio between the new policy and the old. If this ratio moves too far from 1, the objective is clipped, removing the incentive for the optimizer to push the update further. This allows for multiple epochs of updates on the same batch of data, which would cause standard policy gradient to overfit and diverge, resulting in significantly improved sample efficiency and more better convergence.