Assignment 2: Policy Gradient

Andrew ID: `bharathh`
Collaborators: `jlessing, krrishj`
NOTE: Please do **NOT** change the sizes of the answer blocks or plots.

5 Small-Scale Experiments

5.1 Experiment 1 (Cartpole) – [5 points total]

5.1.1 Configurations

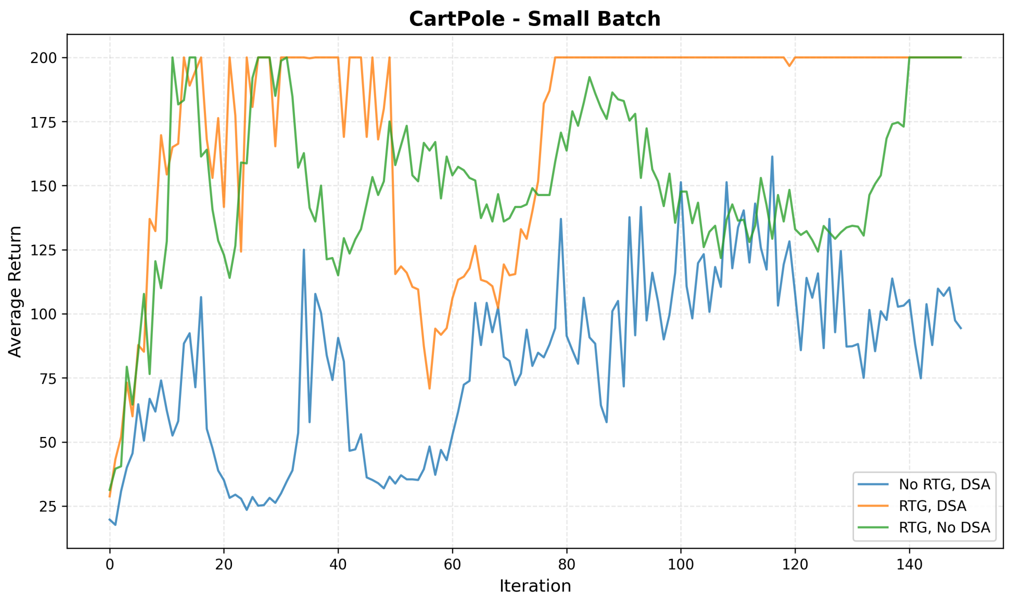
Q5.1.1

```
python rob831/scripts/run_hw2.py --env_name CartPole-v0 -n 150 -b 1500 \
    -dsa --exp_name q1_sb_no_rtg_dsa

python rob831/scripts/run_hw2.py --env_name CartPole-v0 -n 150 -b 1500 \
    -rtg -dsa --exp_name q1_sb_rtg_dsa

python rob831/scripts/run_hw2.py --env_name CartPole-v0 -n 150 -b 1500 \
    -rtg --exp_name q1_sb_rtg_na

python rob831/scripts/run_hw2.py --env_name CartPole-v0 -n 150 -b 6000 \
    -dsa --exp_name q1_lb_no_rtg_dsa

python rob831/scripts/run_hw2.py --env_name CartPole-v0 -n 150 -b 6000 \
    -rtg -dsa --exp_name q1_lb_rtg_dsa

python rob831/scripts/run_hw2.py --env_name CartPole-v0 -n 150 -b 6000 \
    -rtg --exp_name q1_lb_rtg_na
```

5.1.2 Plots

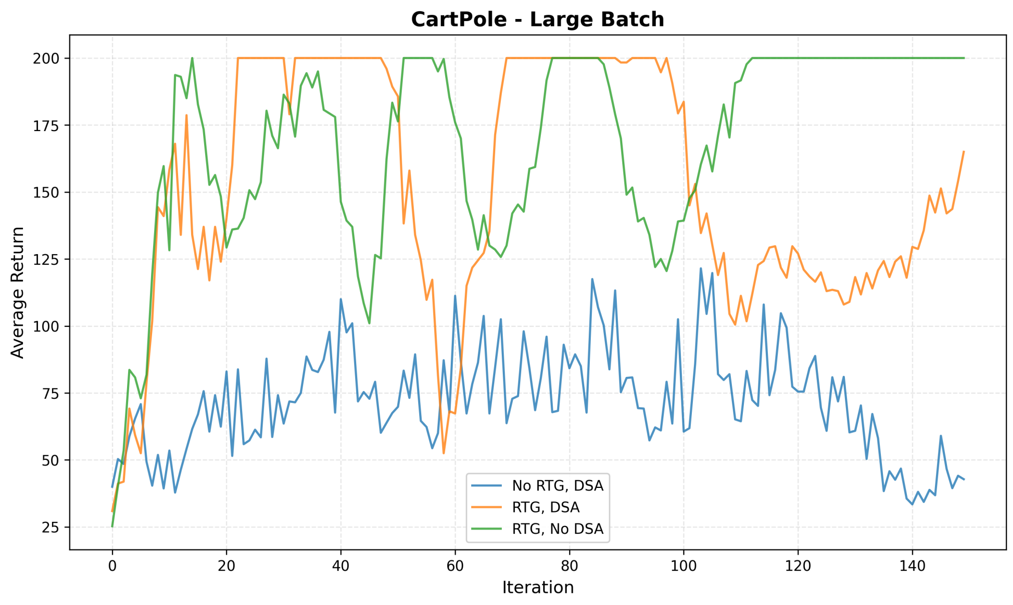
5.1.2.1 Small batch – [1 points]

Q5.1.2.1



5.1.2.2 Large batch – [1 points]

Q5.1.2.2



5.1.3 Analysis

5.1.3.1 Value estimator – [1 points]

Q5.1.3.1

In both small and large batch size cases, the reward-to-go value estimator has better performance than the trajectory-centric (blue curve) one.

5.1.3.2 Advantage standardization – [1 points]

Q5.1.3.2

In both cases, the average return with or without advantage standardization reaches the maximum value of 200. However, the one without advantage standardization (orange curve) has larger fluctuations than the the one with (green curve), thus indicating that that advantage standardization helped in reducing variance.

5.1.3.3 Batch size – [1 points]

Q5.1.3.3

In terms of average return there does not seem to be much difference between the batch sizes, given that RTG and DSA flags are same. However, the small batch size curves have many more fluctuations than the larger curves, thus indicating that larger batch size helped in reducing variance.

5.2 Experiment 2 (InvertedPendulum) – [4 points total]

5.2.1 Configurations – [1.5 points]

Q5.2.1

```
python rob831/scripts/run_hw2.py --env_name InvertedPendulum-v4 \
    --ep_len 1000 --discount 0.92 -n 100 -l 2 -s 64 -b 200 -lr 0.05 -rtg \
    --exp_name q2_b200_r0.05
```

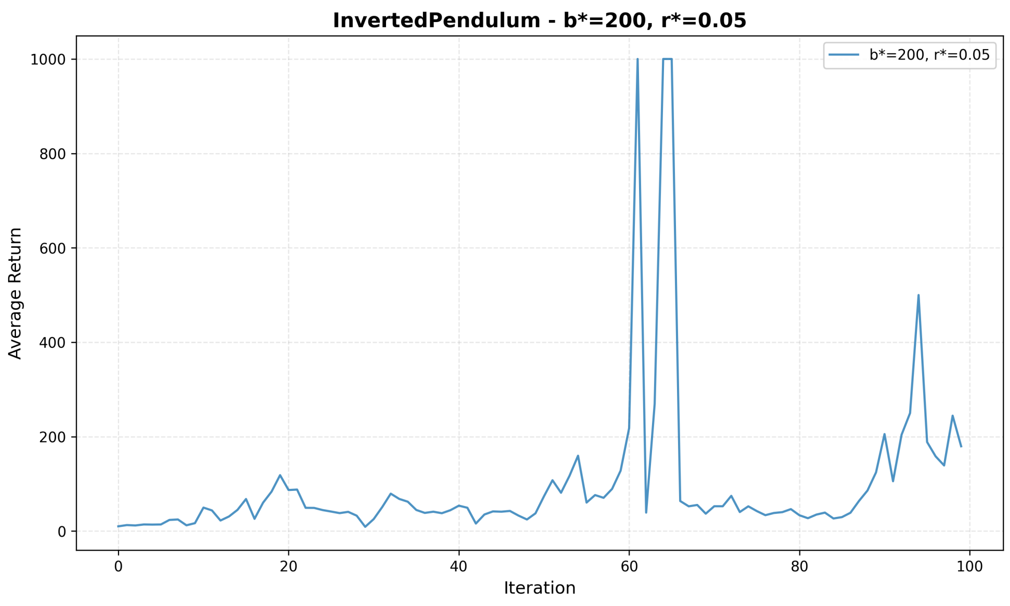
5.2.2 smallest b* and largest r* (same run) – [1.5 points]

Q5.2.2

b* = 200, r* = 0.05
To find the above values, I ran 15+ experiments with various different batch size and learning rate combinations. I started with b = 50, then tried a number of r's, then b = 60 and so on until I found a combination that touched an eval return of the maximum possible 1000. I then chose the maximum possible learning rate at this batch size that still reached 1000.

5.2.3 Plot – [1 points]

Q5.2.3



7 More Complex Experiments

7.1 Experiment 3 (LunarLander) – [1 points total]

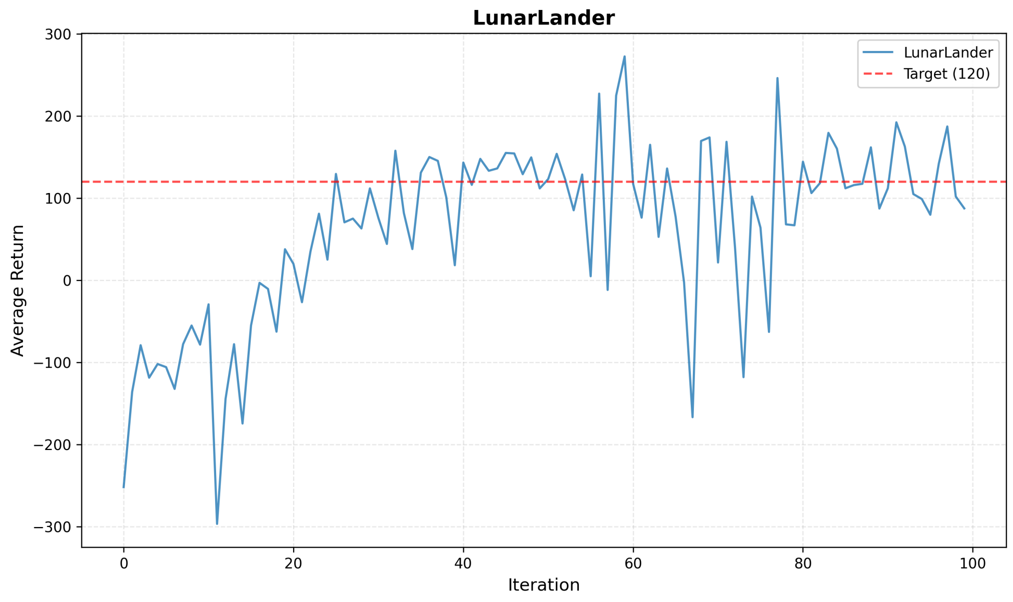
7.1.1 Configurations

Q7.1.1

```
python rob831/scripts/run_hw2.py \
    --env_name LunarLanderContinuous-v4 --ep_len 1000
    --discount 0.99 -n 100 -l 2 -s 64 -b 10000 -lr 0.005 \
    --reward_to_go --nn_baseline --exp_name q3_b10000_r0.005
```

7.1.2 Plot – [1 points]

Q7.1.2



7.2 Experiment 4 (HalfCheetah) – [1 points]

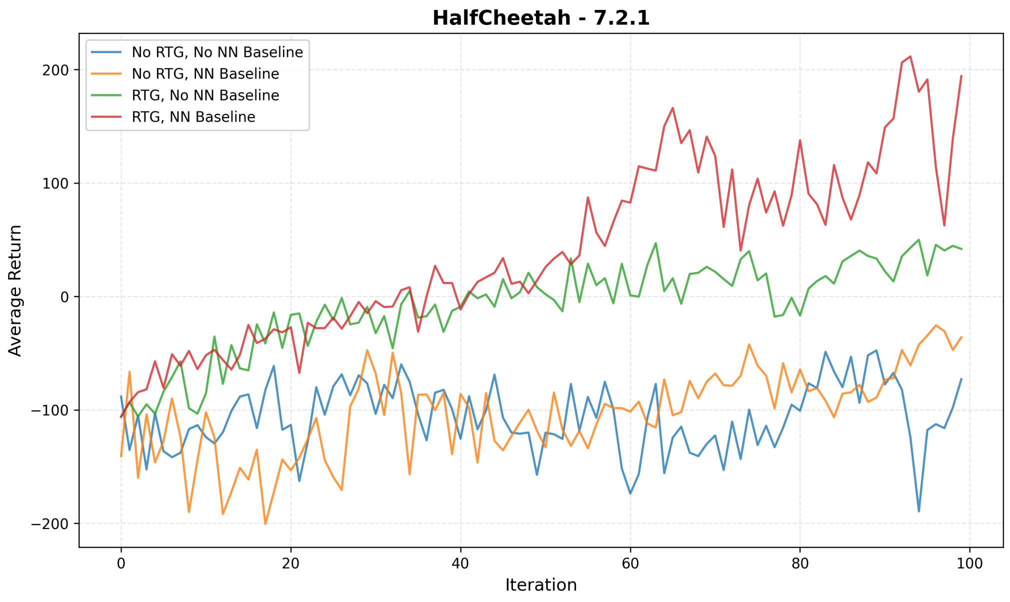
7.2.1 Configurations

Q7.2.1

```
python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 10000 -lr 0.02 \
    --exp_name q4_search_b10000_lr0.02
python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 10000 -lr 0.02 -rtg \
    --exp_name q4_search_b10000_lr0.02_rtg
python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 10000 -lr 0.02 --nn_baseline \
    --exp_name q4_search_b10000_lr0.02_nnbaseline
python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 10000 -lr 0.02 -rtg --nn_baseline \
    --exp_name q4_search_b10000_lr0.02_rtg_nnbaseline
```

7.2.2 Plot – [1 points]

Q7.2.2



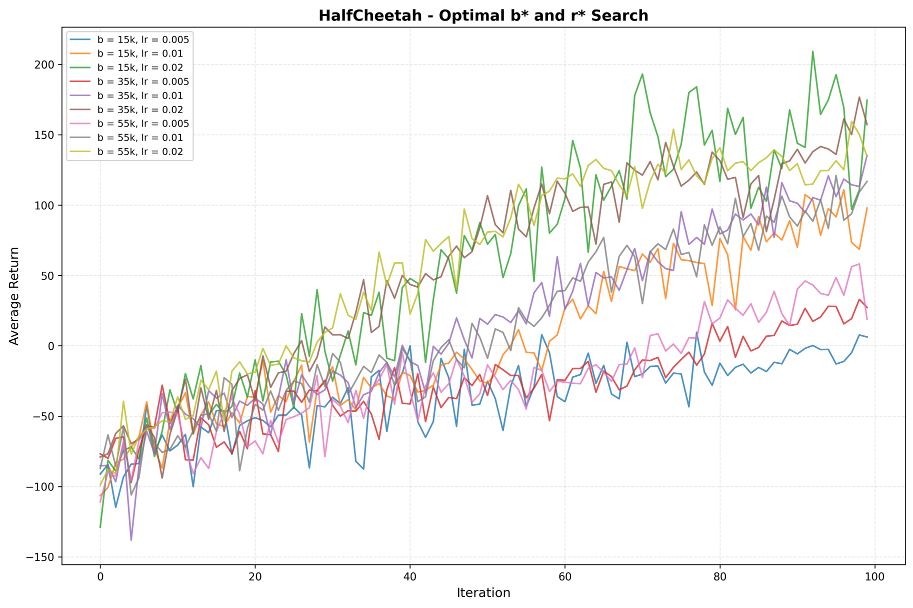
7.2.3 Optimal b* and r* – [0.5 points]

Q7.2.3

b* = 15000, r* = 0.02
From the plot in 7.2.4 it is clear that r* = 0.02.
However, b* is less clear as all 3 curves (green, olve and brown) follow each other closely. 15k (green) has the highest variance, however I chose this as my b*, since it touched the highest average return (>200) among all three.

7.2.4 Plot – [0.5 points]

Q7.2.4



7.2.5 Describe how b* and r* affect task performance – [0.5 points]

Q7.2.5

The plot shows that higher learning rates improve average return and task performance, with lr = 0.02 achieving the best results across batch sizes.
Batch size effects are less clear: for lr = 0.05, larger batches improve returns late in training, but for lr = 0.005 or 0.002, the curves are similar.
However, we can say that smaller batch sizes show greater variance, with curves (e.g., orange vs. purple/grey, green vs. olive/brown) fluctuating more than those of larger batches.

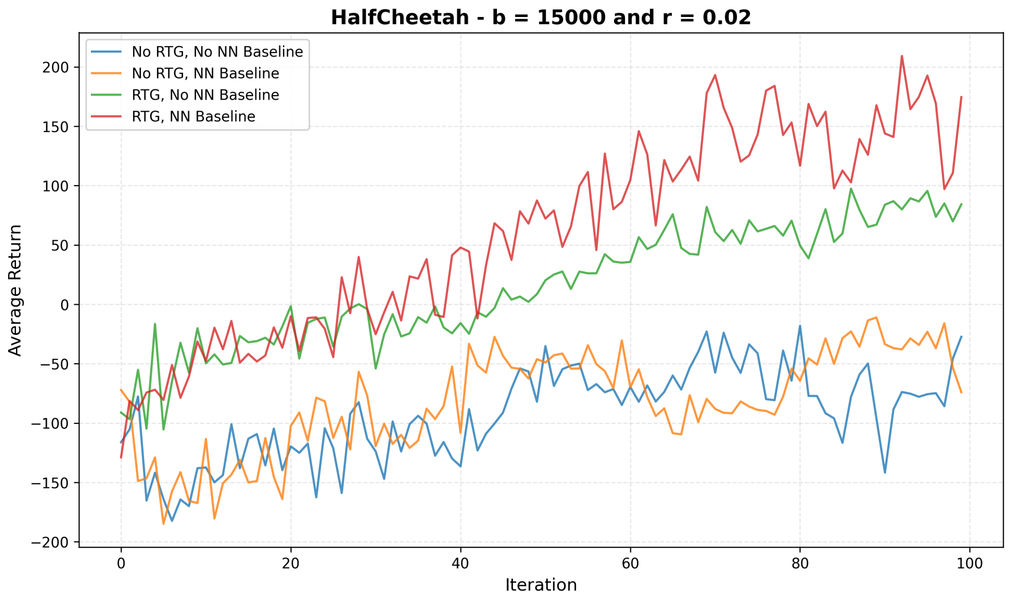
7.2.6 Configurations with optimal b* and r* – [0.5 points]

Q7.2.6

```
python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 15000 -lr 0.02 \
    --exp_name q4_b15000_r0.02

python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 15000 -lr 0.02 -rtg \
    --exp_name q4_b15000_r0.02_rtg

python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 15000 -lr 0.02 --nn_baseline \
    --exp_name q4_b15000_r0.02_nnbaseline

python rob831/scripts/run_hw2.py --env_name HalfCheetah-v4 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b 15000 -lr 0.02 -rtg --nn_baseline \
    --exp_name q4_b15000_r0.02_rtg_nnbaseline
```

7.2.7 Plot for four runs with optimal b* and r* – [0.5 points]

Q7.2.7



8 Implementing Generalized Advantage Estimation

8.1 Experiment 5 (Hopper) – [4 points]

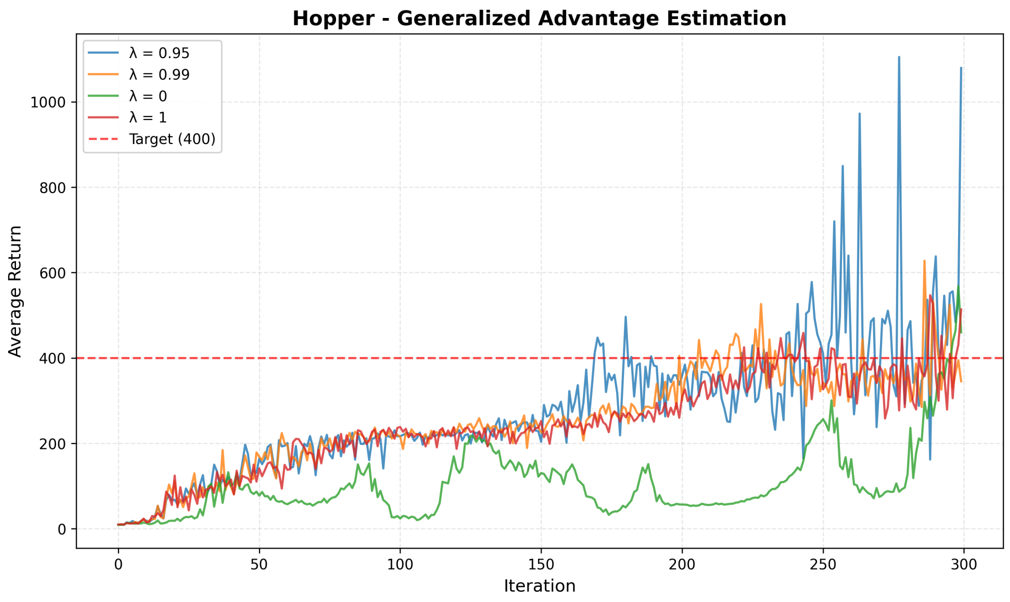
8.1.1 Configurations

Q8.1.1

```
# λ ∈ [0, 0.95, 0.99, 1]
python rob831/scripts/run_hw2.py \
    --env_name Hopper-v4 --ep_len 1000
    --discount 0.99 -n 300 -l 2 -s 32 -b 2000 -lr 0.001 \
    --reward_to_go --nn_baseline --action_noise_std 0.5 --gae_lambda <λ> \
    --exp_name q5_b2000_r0.001_lambda<λ>
```

8.1.2 Plot – [2 points]

Q8.1.2



8.1.3 Describe how λ affects task performance – [2 points]

Q8.1.3

From the plot we can see that, the $\lambda = 0$ curve has low variance, however has the most bias and performs with the least returns. The $\lambda = 0.95$ curve balances bias and variance and has the highest returns amongst the four. The other two curves perform similar to each other.

9 More Bonus!

9.1 Parallelization – [1.5 points]

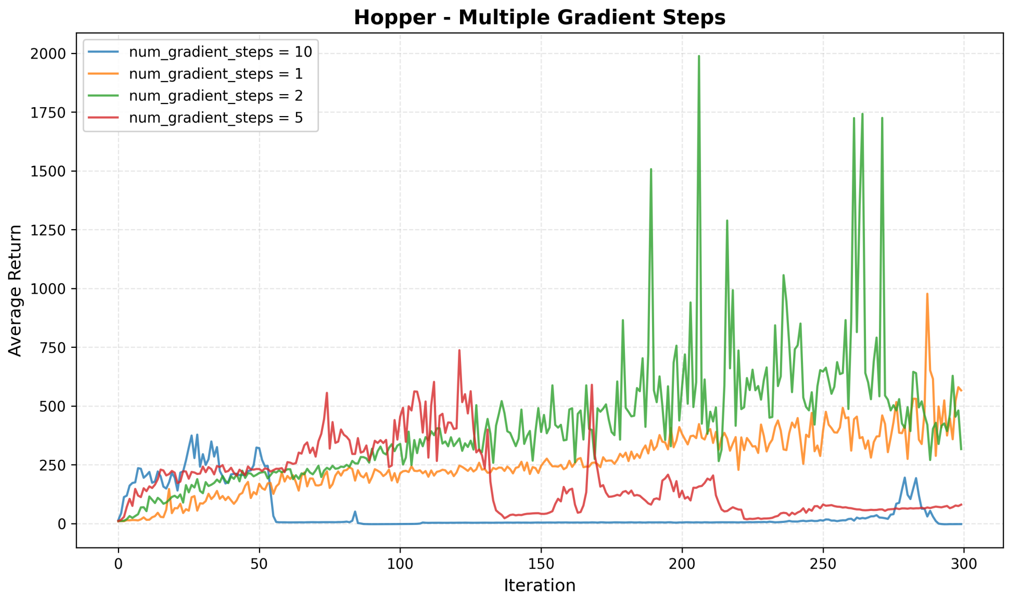
Q9.1

Difference in training time:

```
python rob831/scripts/run_hw2.py \
```

9.2 Multiple gradient steps – [1 points]

Q9.2



```
python rob831/scripts/run_hw2.py --env_name Hopper-v4 --ep_len 1000 \
    --discount 0.99 -n 300 -l 2 -s 32 -b 2000 -lr 0.001 \
    --reward_to_go --nn_baseline --action_noise_std 0.5 \
    --num_gradient_steps <num_steps> --exp_name q9_bonus_step_<num_steps>

Note: Same environment and parameters used as in 8.1.1. Additionally a new param added - num_gradient_steps. When this
↪  is 1, it is same as the original policy gradient method without multiple gradient steps.
```

From the plot we can say that, while 2 gradient steps shows significant improvement over the baseline single step, both 5 and 10 gradient steps perform worse than even the single step method. Thus we can say that, taking more than one gradient step helps because it uses the collected data better improving learning speed. However, too many steps (like 5 or 10) pushes the policy too far from the data it came from, causing unstable updates.