IFT6163 Assignment 3: Policy Gradients

Simon Chamorro

March 6, 2022

1 Policy Gradients

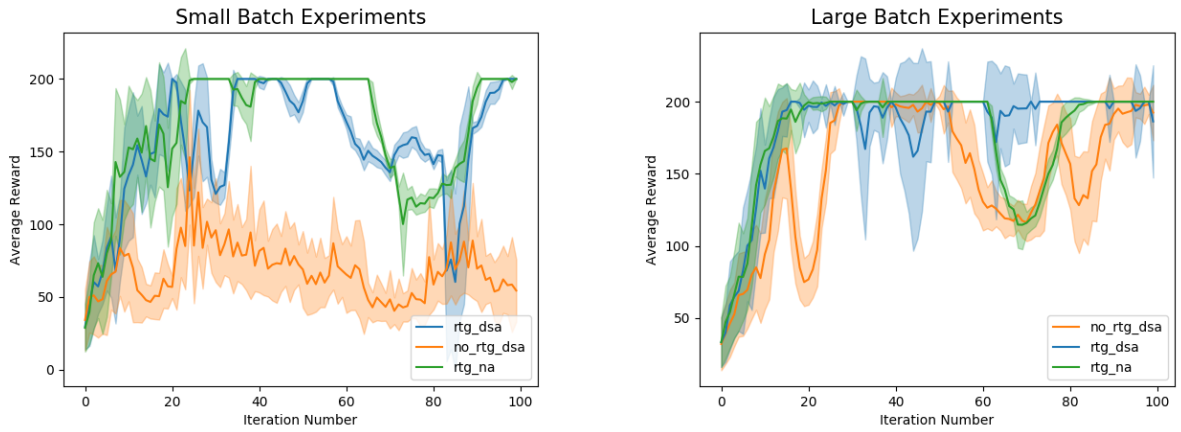
For the experiments in this section, and all the experiments in this report, the default parameters are the following:

- learning rate: 0.005
- number of iterations: 100
- batch size (both train and eval): 1000
- size: 64
- number of layers: 2
- discount: 1.0
- reward to go: false
- don't standardize advantages: false

All tthe experiments can be reproduced by running `scripts/all_exps.sh`, while all the figures can be recreated by running `scripts/make_plots.py`.

Name: Simon Chamorro

1.1 Experiment 1 - Cartpole



(a) Small Batch Size

(b) Large Batch Size

Figure 1: **Policy Gradient Small-Scale Experiments.** Results for Policy Gradient on the CartPole environment using different batch sizes (1000 for figure 1a and 5000 for figure 1b).

Which value estimator has better performance without advantage-standardization: the trajectory-centric one, or the one using reward-to-go?

We can see that when using reward to go, the agent achieves better performance and training is more stable. This is specially true in the small batch size experiments.

Did advantage standardization help?

Advantage standardization did make convergence faster and training more stable. Its effect is not huge, but it is most noticeable in the small scale experiments. We see that the green curve reaches a stable state at 200 faster than the blue one.

Did the batch size make an impact?

The batch size definitely made a big impact. Bigger batch sizes make policy gradient more stable and converge faster, as there is more diverse on-policy data to train from at each iteration.

All the exact commands to run the experiments for this question are provided in the bash script in `scripts/01-policy-gradient-cartpole.sh`.

Name: Simon Chamorro

1.2 Experiment 2 - Inverted Pendulum

After searching over batch sizes ranging from 256 to 2048, we see that there are successful agents reaching a 1000 reward for batch sizes as small as 256. The final configuration chosen is a batch size of 256 and a learning rate of 0.01. Figure 2 shows the performance for different learning rates and a batch size of 256, while figure 3 shows the performance of the final configuration averaged over five different seeds. We see that not every seed achieves a performance of 1000. All the exact commands to run the experiments for this question are provided in the bash script in `scripts/02-policy-gradient-pendulum.sh`. For instance, the command to run one of the seeds used to produce figure 3 would be the following:

```
python run_hw3.py env_name=InvertedPendulum-v2 ep_len=1000
discount=0.9 reward_to_go=true batch_size=256 learning_rate=0.01
exp_name=q2_final_b256_r01_1 rl_alg=reinforce seed=0
```
[1]

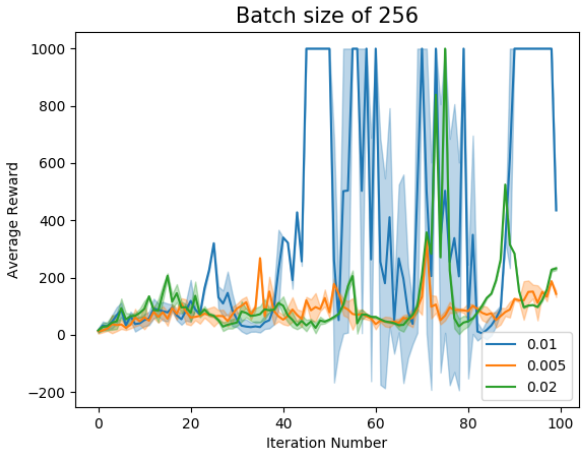


Figure 2: Batch Size of 256

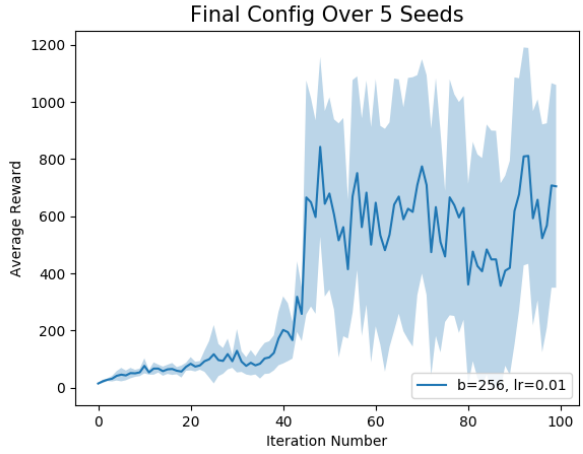


Figure 3: Batch of 256 and learning rate of 0.01 over 5 seeds

Name: Simon Chamorro

2 Policy Gradients with Neural Network Baselines

After implementing neural network baselines for our policy gradient agent, we will test it on the LunarLander and the HalfCheetah environments.

2.1 Experiment 3 - Lunar Lander

For this experiment, the following command was used:

```
python run_hw3.py env_name=LunarLanderContinuous-v2 ep_len=1000
discount=0.99 reward_to_go=true nn_baseline=true batch_size=40000
learning_rate=0.005 exp_name=q3_b40000_r0.005 rl_alg=reinforce
```
[2]

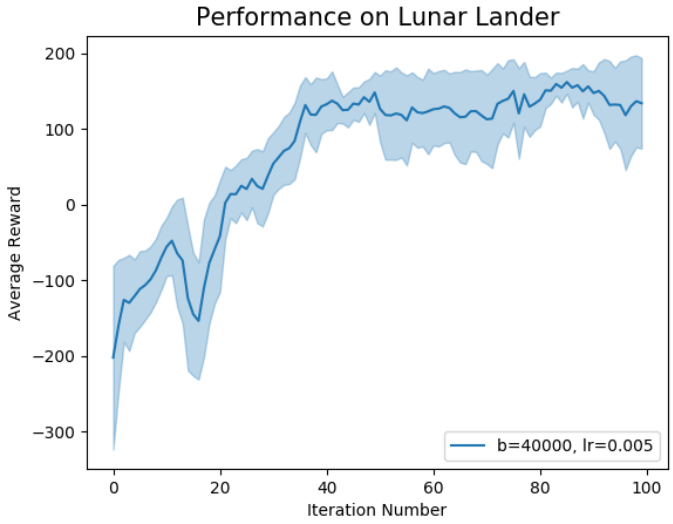


Figure 4: Performance of PG with nn baseline on LunarLander

2.2 Experiment 4 - Half Cheetah

For the Half Cheetah environment, we used an episode length of 150 to speed up training and searched for the best hyperparameters for batch size and learning rate. Figure 5 shows the results. All the exact commands to run both experiments shown above are provided in the bash script in scripts/04-policy-gradient-half-cheetah.sh. For instance, the command for the best hyperparameter combination in figure 5 is:

```
python run_hw3.py env_name=HalfCheetah-v2 ep_len=150
discount=0.95 size=32 reward_to_go=true nn_baseline=true batch_size=50000
learning_rate=0.02 exp_name=q4_search_b50000_r0.02_rtg_nnbaseline
rl_alg=reinforce
```
[3]

Name: Simon Chamorro

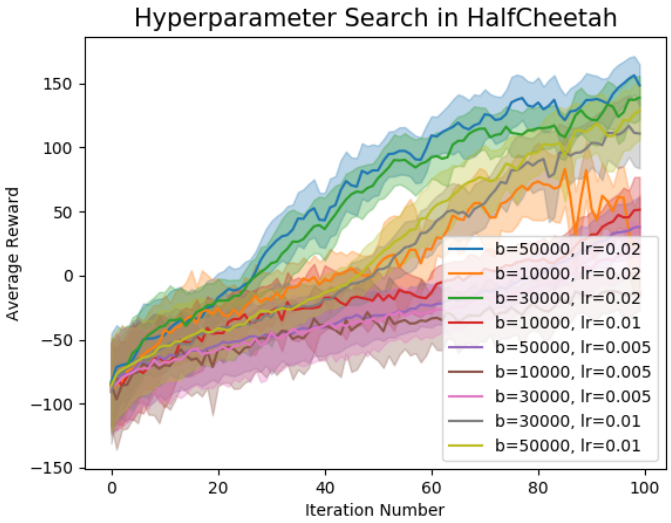


Figure 5: Hyperparameter search on HalfCheetah

As we can see, the best combination is a batch size of 50000 and a learning rate of 0.02, so we used these parameters for the following experiments, where we do an ablation study to see the importance of using the reward-to-go and the neural network baseline. Figure 6 shows these results.

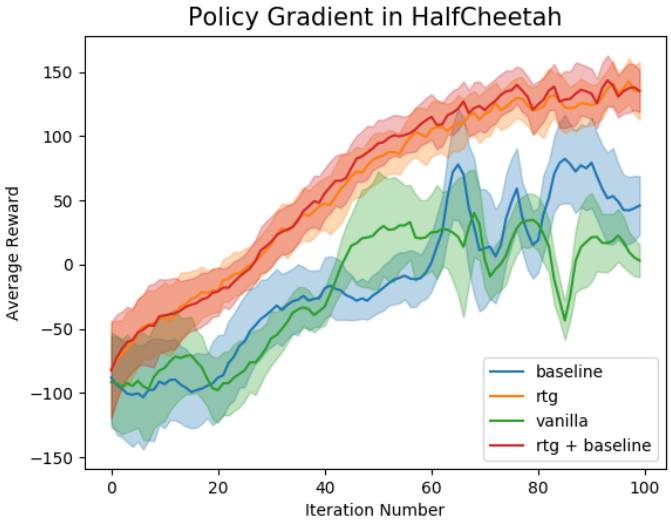


Figure 6: Performance on HalfCheetah

Name: Simon Chamorro

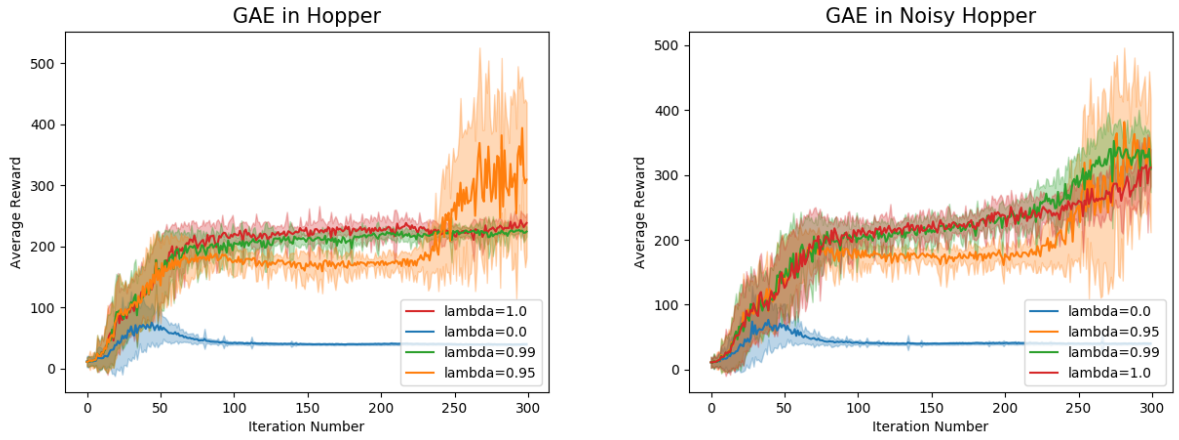
3 Generalized Advantage Estimation

We test our implementation of $GAE - \lambda$ in the Hopper environment, first without any noise, then by adding noise to the the actions of our agent. The commands to run all experiments are provided in the file `scripts/05-pg-gae-hopper.sh`. For example:

```
python run_hw3.py env_name=Hopper-v2 ep_len=1000
discount=0.99 n_iter=300 size=32 reward_to_go=true nn_baseline=true
batch_size=2000 learning_rate=0.001 rl_alg=reinforce gae_lambda=0.95
exp_name=q5_noisy_b2000_r0.001_lambda0.95 action_noise_std=0.5
```

[4]

3.1 Experiment 5 - Hopper and Noisy Hopper



(a) Normal (b) Noisy

Figure 7: **GAE experiments in the Hopper environment.** Results for Policy Gradient with GAE on the Hopper environment. We see that GAE outperforms the vanilla PG agent (equivalent to $\lambda = 1.0$). The difference in performance is most notable in the environment without noise.

4 Actor-Critic

We evaluate our Actor-Critic implementation in various environments. First, in CartPole to do a sanity check and choose the best hyperparameters. Then, we evaluate our agent in both InvertedPendulum and HalfCheetah.

4.1 Experiment 6 - CartPole

We notice that the best configuration is to have 10 target updates and 10 gradient steps per target update. These parameters reach a maximum performance in CartPole fastest. The commands to run this experiment are provided in the bash script `scripts/06-actor-critic-cartpole.sh`.

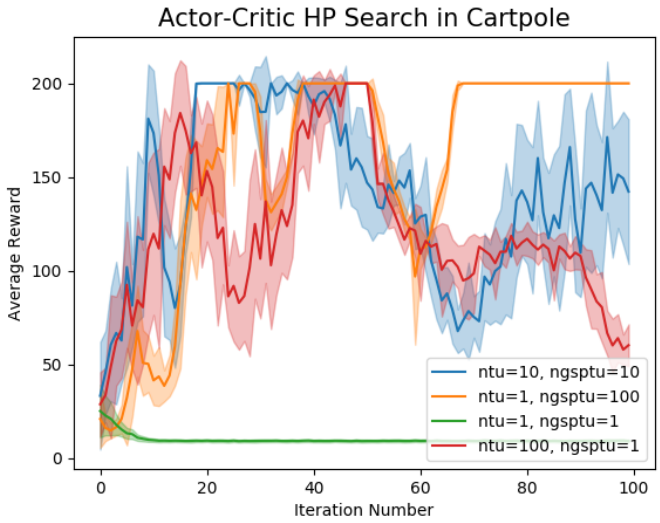
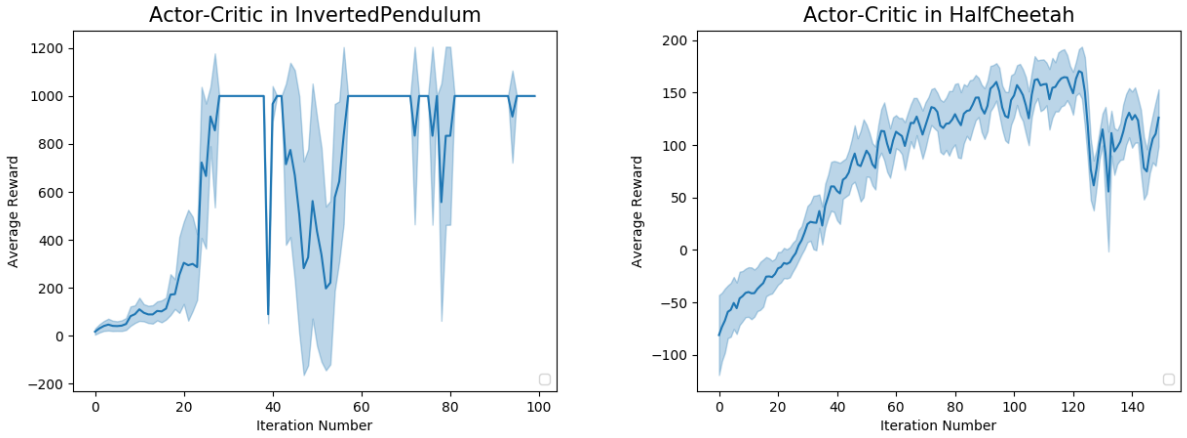


Figure 8: Hyperparameter search on CartPole

4.2 Experiment 7 - Difficult Environments

For this experiment, we use the best parameters from experiment 6: ntu=10 and ngsptu=10. The commands are in the file `scripts/07-actor-critic-difficult.sh`.



(a) InvertedPendulum

(b) HalfCheetah

Figure 9: **Actor-Critic experiments in InvertedPendulum and HalfCheetah.**

Name: Simon Chamorro

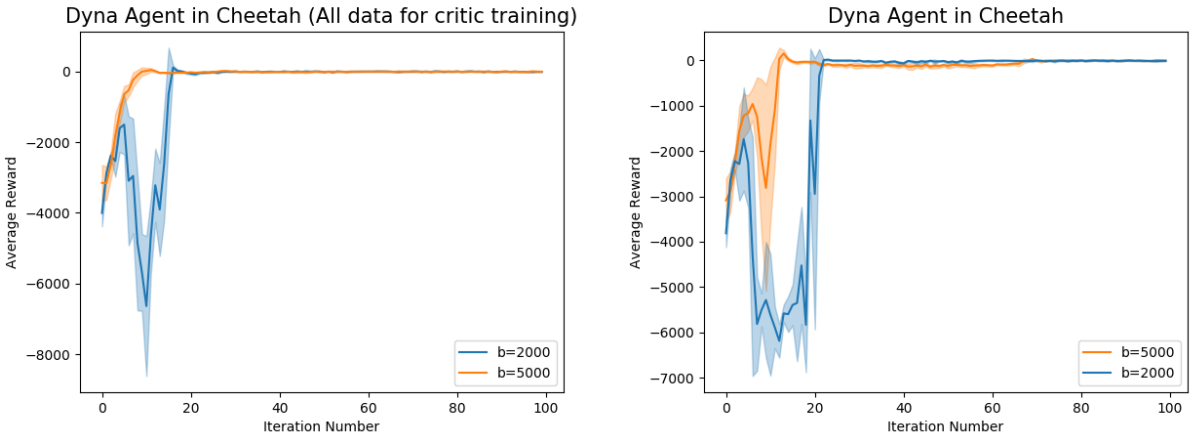
5 Dyna

Using our code from the actor-critic agent, the policy gradient agent, and the model-based code from the previous assignment, we implement a version of Dyna. We test our agent on the cheetah environment by running the commands in the file `scripts/08-dyna.sh`. Here is an example:

```
python run_hw3.py env_name=cheetah-ift6163-v0 size=32 learning_rate=0.01
discount=0.95 rl_alg=dyna train_critic_with_new_data=false
exp_name=q8_cheetah_n500_arch1x32_b5000 batch_size=5000
```
[5]

5.1 Experiment 8 - Cheetah

Figure 10 show the results for two different settings. First, in figure 10a, we use all the new data generated by the world model to train the policy as well as the critic. In the second experiment (figure 10b), the new data is used only to train the actor, whereas the critic is trained only with the original data. We see that using all the data stabilizes the training and is a better option overall.



(a) Critic trained on all data

(b) Critic trained on original data

Figure 10: **Dyna Experiments in Cheetah.**

Name: Simon Chamorro

6 Bonus - Multi-step Policy Gradient

In this section, we explore the possibility of doing multiple policy gradient steps with the same batch of data to accelerate training. To do so, we will first compare the performance of experiment 3 from section 2.1. The results are shown in figure 11. We can see that the performance seems to improve quickly at the beginning, but it collapses after around 30 iterations. To avoid this, we try to lower the learning rate to 0.001 instead and 0.005. We can see that increasing the step number to 2 does improve performance, but after increasing it to 3, it collapses again. We can conclude that multi-step policy gradient could help accelerate training, but also makes it more unstable because the data that we use is no longer on-policy. This addition to policy gradient would require a more thorough hyperparameter search.

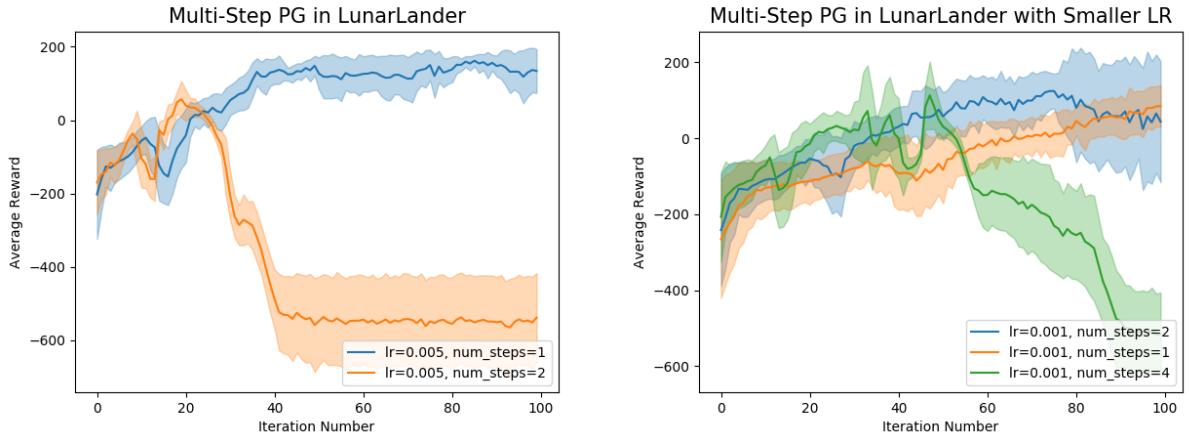


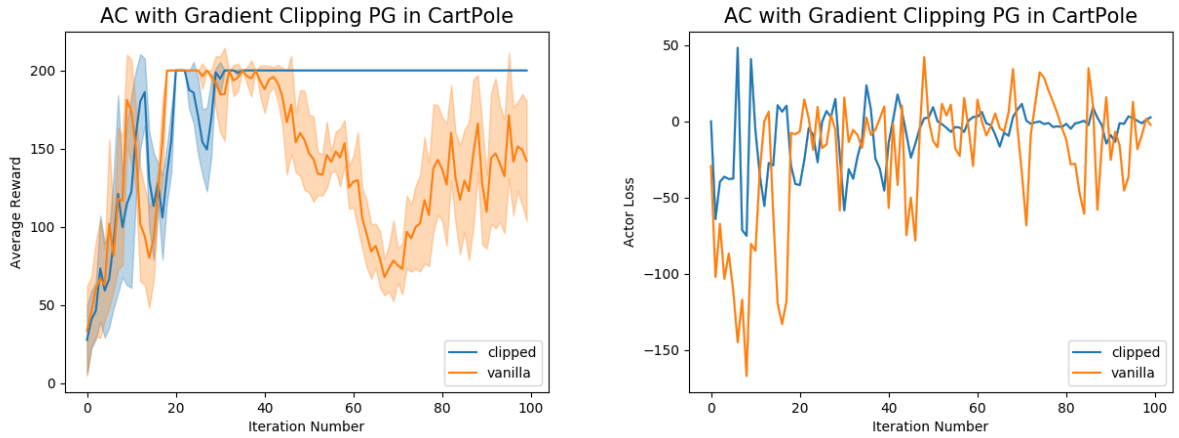
Figure 11: **Multi-Step PG experiments in LunarLander**.

All the experiments presented in this section can be reproduced by running `scripts/09-multi-step-pg.sh`.

Name: Simon Chamorro

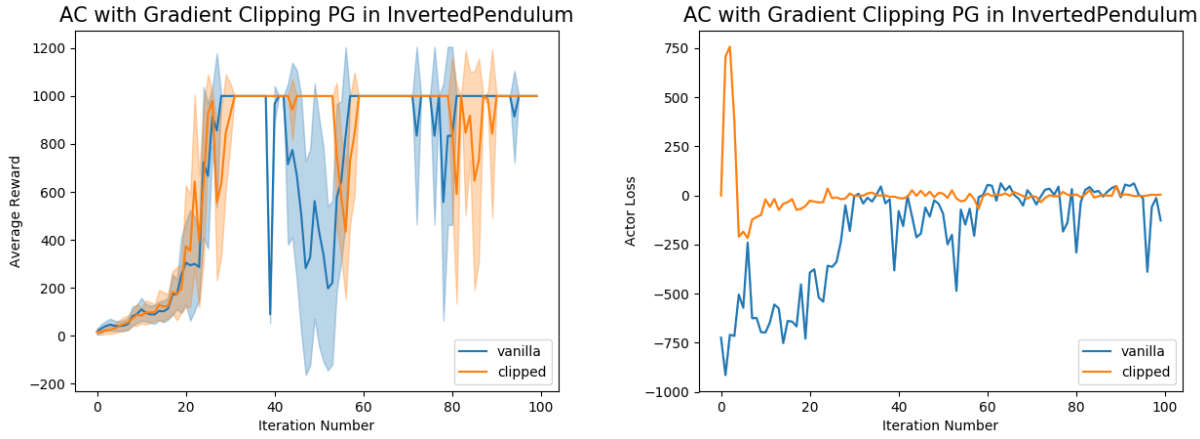
7 Bonus - Actor-Critic with PPO Clipping

Another improvement that was explored for actor-critic was the addition of the PPO clipping term. The code for this experiment can be found in `ift6163/policies/MLP_policy.py` where the update method of the actor-critic policy is modified to clip the ratio between the new policy and the old. We use the default $\epsilon = 0.2$ parameter from the PPO paper for all experiments. We evaluate the effect of gradient clipping on two environements, first CartPole, then InvertedPendulum. The results are shown in figures 12 and 13 respectively. We can see that this clipping improves performance and specially stability. We notice that it also stabilizes the actor loss throughout training by limiting the policy change at each step. All the experiments presented in this section can be reproduced by running `scripts/10-ac-ppo-clip.sh`.



(a) Reward (b) Actor Loss

Figure 12: **Gradient Clipping in Cart Pole.**



(a) Rewards (b) Actor Loss

Figure 13: **Gradient Clipping in Inverted Pendulum.**

Name: Simon Chamorro

8 Bonus - Parallelized trajectory collection

Finally, we study the time improvement that is possible by collecting data in parallel threads. The experiments are all conducted in CartPole. Figure 14 presents the results of all training runs simply as a sanity check of the parallel implementation to confirm that we obtain similar performances, since this should not be affected by the parallelization. Then, figure 15 shows the time improvement results. We notice that using six threads instead of one yields a data collection time decrease of 73%, dropping from 1.64 to 0.44 seconds. This results in a 34% training time improvement, dropping from 317 to 208 seconds. All the experiments presented in this section can be reproduced by running `scripts/11-parallelized-data-collection.sh`. Notice that the current implementation does not work when using a GPU, so the `no_gpu=true` parameter is important.

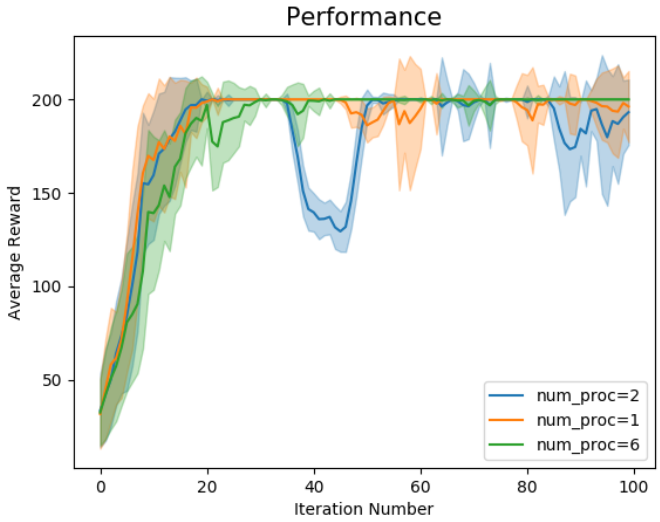


Figure 14: **Parallel Data Collection Results.**

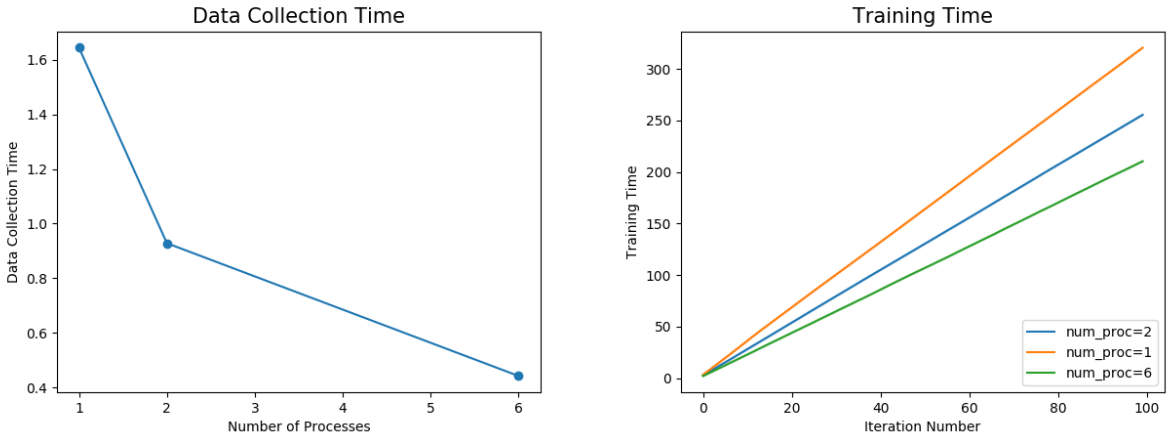


Figure 15: **Parallel Data Collection Time.**