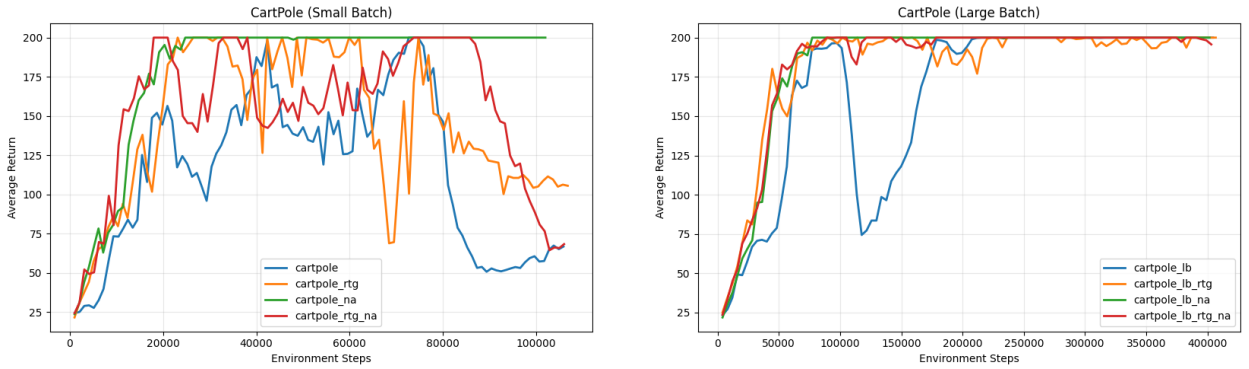
CS 185/285 Assignment 2: Policy Gradients

Seoyeon Choi

Spring 2026

1 Experiment 1: CartPole



(a) Small batch

(b) Large batch

Figure 1: CartPole performance comparison.

1.1 Questions

Which value estimator has better performance without advantage normalization: the trajectory-centric one, or the one using reward-to-go? The reward-to-go performs better than the trajectory-centric when advantage normalization is not used. From Fig. 1a, `cartpole_rtg` is significantly more stable and reaches high return faster than `cartpole`. Also in Fig. 1b, `cartpole_lb_rtg` avoids the large collapse seen in `cartpole_lb` and converges more smoothly.

Between the two value estimators, why do you think one is generally preferred over the other? Reward-to-go is generally preferred over trajectory-centric return. Trajectory-centric return assigns the same total episode return to every timestep in that trajectory. This means early actions get credit for rewards that happen far in the future, even if they had little influence. Instead, reward-to-go removes reward contributions from earlier timesteps and provides a less noisy and more causally relevant signal, reducing variance in policy gradient updates.

Did advantage normalization help? Yes, especially for small batch size. Normalization helps more when variance is high, i.e. in small batch. This can be seen in Fig. 1a, where `cartpole_na` and `cartpole_rtg_na` are more stable.

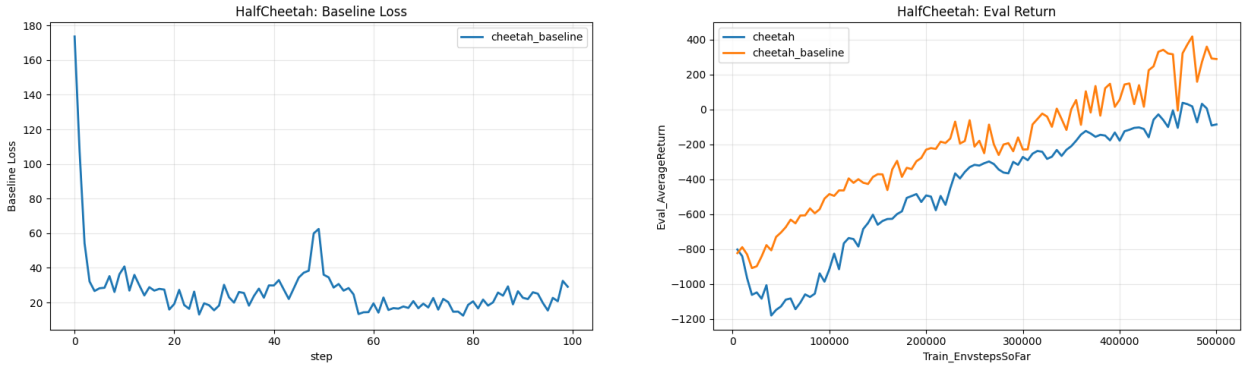
Did the batch size make an impact? Yes, overall, the learning is much smoother in the large batch with fewer collapses, converging much early.

1.2 Commands used

Below are the commands used to train the eight experiments shown in Fig. 1.

```
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 1000 \
--exp_name cartpole
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 1000 \
-rtg --exp_name cartpole_rtg
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 1000 \
-na --exp_name cartpole_na
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 1000 \
-rtg -na --exp_name cartpole_rtg_na
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 4000 \
--exp_name cartpole_lb
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 4000 \
-rtg --exp_name cartpole_lb_rtg
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 4000 \
-na --exp_name cartpole_lb_na
uv run src/scripts/run.py --env_name CartPole-v0 -n 100 -b 4000 \
-rtg -na --exp_name cartpole_lb_rtg_na
```

2 Experiment 2: HalfCheetah



(a) Learning curve for baseline loss

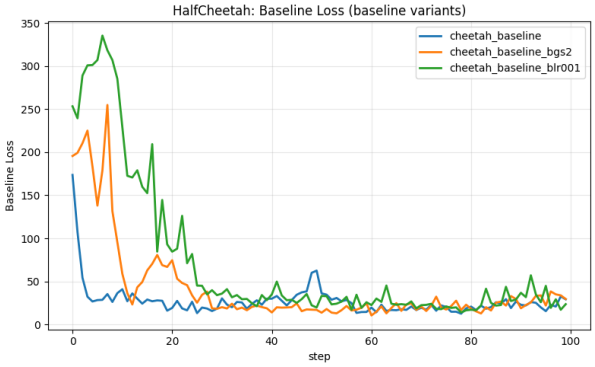
(b) Learning curve for the eval return

Figure 2: HalfCheetah performance comparison: No baseline vs Baseline

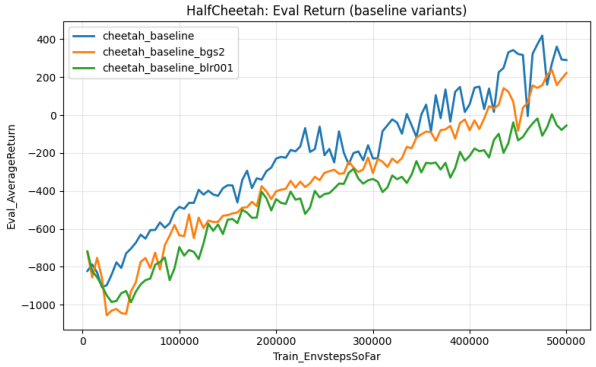
2.1 Discussion

No baseline vs Baseline Adding a learned value function baseline significantly improves performance shown in Fig. 2. The baseline reduces variance in the policy gradient by subtracting an estimate of the value function from the returns.

Variants of baseline methods: how did it affect (a) baseline loss curve and (b) the performance of the policy? As shown in Fig. 3a, all of the variants' baseline loss curve eventually stabilize, but the default baseline shows the most stable and fast converging curve. The eval return curve in Fig. 3b shows similar results where the default baseline achieves the best final return.



(a) Learning curve for baseline loss



(b) Learning curve for the eval return

Figure 3: HalfCheetah performance comparison: Variants of baseline methods. `cheetah_baseline_bgs2` denotes decreased number of baseline gradient steps as 2 and `cheetah_baseline_blr001` denotes run with decreased baseline learning rate as 0.001

Small learning rate slows value function adaptation, and small batch size increases variance in value updates, leading to slow convergence and lower return.

2.2 Commands used

Below are the commands used to train the experiments shown in Fig. 2 and Fig. 3.

```
# No baseline
uv run src/scripts/run.py --env_name HalfCheetah-v4 -n 100 -b 5000 -eb 3000 -rtg \
--discount 0.95 -lr 0.01 --exp_name cheetah
# Baseline
uv run src/scripts/run.py --env_name HalfCheetah-v4 -n 100 -b 5000 -eb 3000 -rtg \
--discount 0.95 -lr 0.01 --use_baseline -blr 0.01 -bgs 5 --exp_name cheetah_baseline
# Baseline with decreased number of baseline gradient steps (-bgs)
uv run src/scripts/run.py --env_name HalfCheetah-v4 -n 100 -b 5000 -eb 3000 -rtg \
--discount 0.95 -lr 0.01 --use_baseline -blr 0.01 -bgs 2 --exp_name
    cheetah_baseline_bgs2
# Baseline with decrease baseline learning rate (-blr)
uv run src/scripts/run.py --env_name HalfCheetah-v4 -n 100 -b 5000 -eb 3000 -rtg \
--discount 0.95 -lr 0.01 --use_baseline -blr 0.001 -bgs 5 --exp_name
    cheetah_baseline_blr001
```

3 Experiment 3: LunarLander

How did λ affect task performance? The choice of λ significantly affects performance. Smaller λ values (especially $\lambda = 0$) perform poorly, exhibiting high bias due to heavy bootstrapping. As λ increases toward 1, performance improves substantially. λ values in the range 0.98–1 achieve the best returns.

What does $\lambda = 0$ correspond to? What about $\lambda = 1$? $\lambda = 0$ corresponds to pure one-step TD learning, while $\lambda = 1$ corresponds to Monte Carlo returns. Since LunarLander reward is sparse and delayed, successful landing depends on long-horizon credit assignment. Therefore, larger λ values perform better with reduced bias in advantage estimates outweighs the increase in variance.

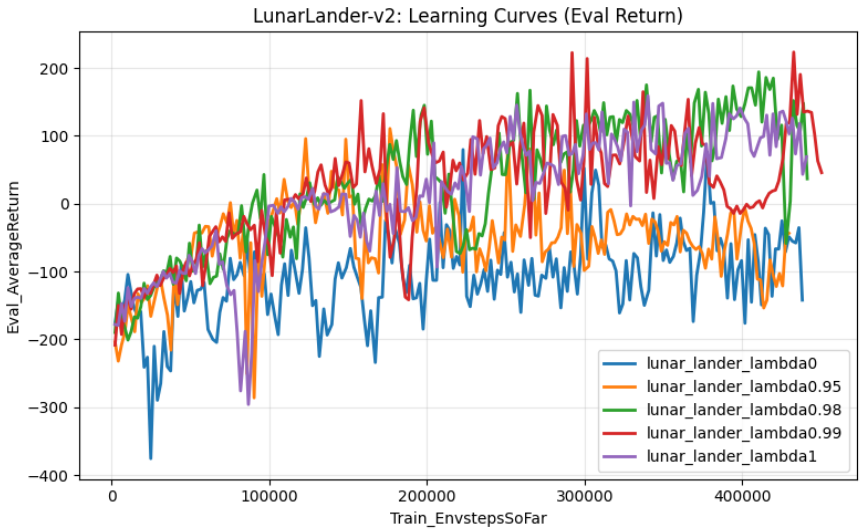


Figure 4: LunarLander performance comparison.

3.1 Commands used

Below is the bash script used to run the experiments shown in Fig. 4.

```
LAMBDA_VALUES=(0 0.95 0.98 0.99 1)

for l in "${LAMBDA_VALUES[@]}"; do
  uv run src/scripts/run.py --env_name LunarLander-v2 --ep_len 1000 --discount 0.99 \
    -n 200 -b 2000 -eb 2000 -l 3 -s 128 -lr 0.001 --use_reward_to_go --use_baseline \
    --gae_lambda "$l" --exp_name "lunar_lander_lambda${l}"
done
```

4 Experiment 4: InvertedPendulum

Table 1: Hyperparameter configurations for InvertedPendulum. All runs use reward-to-go, baseline, and advantage normalization unless otherwise noted. The best set of hyperparameter is outlined as bold.

Exp Name	n	b	λ	γ	lr	blr	bgs	Net (L×H)
pendulum_gae097_b1000_lr3e3	100	1000	0.97	0.99	0.003	0.001	5	2×64
pendulum_b2000_lr2e3_gae095	50	2000	0.95	0.99	0.002	0.001	5	2×64
pendulum_b800_lr3e3_gae098	**125**	**800**	**0.98**	**0.99**	**0.003**	**0.001**	**5**	**2×64**
pendulum_conservative_lr1e3	100	1000	0.97	0.995	0.001	0.001	10	2×32

4.1 Tuning hyperparameters

Table 1 shows the hyperparameters of the candidates that were tested. These candidates were tested to balance fast learning and stability. The main pattern across the runs is using return-to-go, a baseline, and advantage normalization because these usually reduce gradient variance and

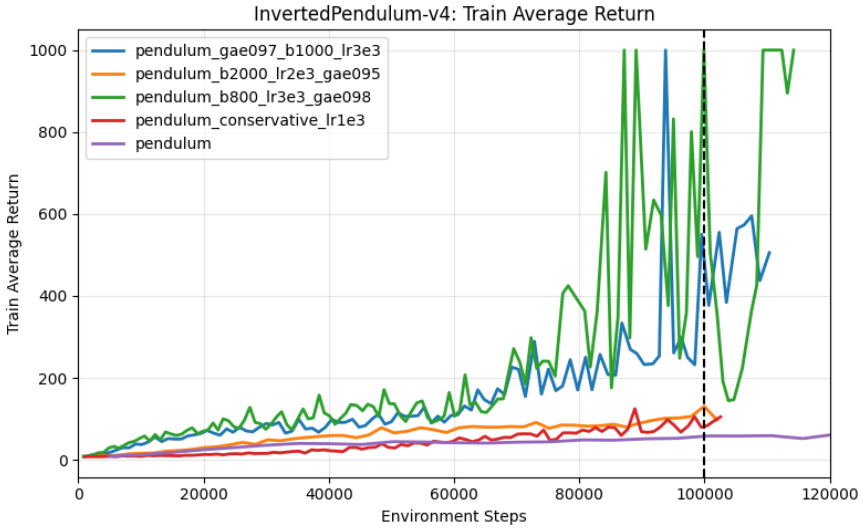


Figure 5: InvertedPendulum performance comparison.

improve sample efficiency. I also included GAE in most runs with lambda values around 0.95 to 0.98, since that range often gives a good bias-variance tradeoff. I varied batch size (800, 1000, 2000) and adjusted n_iter so each configuration stays relatively within the 100K environment step (n * b ¡= 100000), testing the tradeoff between noisier but more frequent updates (smaller batches) and smoother but less frequent updates (larger batches). The learning rates (0.001 to 0.003) and baseline settings (blr=0.001, bgs=5 or 10) were chosen to cover both a reasonably aggressive regime and a more conservative/stable regime, while network sizes (32 or 64, 2 layers) are kept modest because this task typically does not need large models.

The best-performing configuration uses **batch size 800, $\lambda = 0.98$, and learning rate 0.003**, reaching return 1000 within 100K steps. Smaller batch sizes enable more frequent updates, accelerating learning despite slightly higher variance. Larger batches and conservative learning rates slow convergence significantly. λ values close to 1 provide the best bias–variance tradeoff. Overall, update frequency and sufficiently large learning rates were the dominant factors for fast convergence in this task.

4.2 Commands used

Below is the command that yielded the best result shown in Fig. 5 (green line).

```
uv run src/scripts/run.py --env_name InvertedPendulum-v4 -n 125 -b 800 -eb 1000 \
  -rtg --use_baseline -na --gae_lambda 0.98 \
  --discount 0.99 -lr 0.003 -blr 0.001 -bgs 5 \
  -l 2 -s 64 --exp_name pendulum_b800_lr3e3_gae098
```

Below are the entire command used to experiment Fig. 5.

```
# Default
uv run src/scripts/run.py --env_name InvertedPendulum-v4 -n 100 -b 5000 -eb 1000 \
--exp_name pendulum
```

```
# First try
uv run src/scripts/run.py --env_name InvertedPendulum-v4 -n 100 -b 1000 -eb 1000 \
  -rtg --use_baseline -na --gae_lambda 0.97 \
  --discount 0.99 -lr 0.003 -blr 0.001 -bgs 5 \
  -l 2 -s 64 --exp_name pendulum_gae097_b1000_lr3e3

# Lower-variance batch
uv run src/scripts/run.py --env_name InvertedPendulum-v4 -n 50 -b 2000 -eb 1000 \
  -rtg --use_baseline -na --gae_lambda 0.95 \
  --discount 0.99 -lr 0.002 -blr 0.001 -bgs 5 \
  -l 2 -s 64 --exp_name pendulum_b2000_lr2e3_gae095

# Smaller batch, faster updates
uv run src/scripts/run.py --env_name InvertedPendulum-v4 -n 125 -b 800 -eb 1000 \
  -rtg --use_baseline -na --gae_lambda 0.98 \
  --discount 0.99 -lr 0.003 -blr 0.001 -bgs 5 \
  -l 2 -s 64 --exp_name pendulum_b800_lr3e3_gae098

# Conservative learning rate
uv run src/scripts/run.py --env_name InvertedPendulum-v4 -n 100 -b 1000 -eb 1000 \
  -rtg --use_baseline -na --gae_lambda 0.97 \
  --discount 0.995 -lr 0.001 -blr 0.001 -bgs 10 \
  -l 2 -s 32 --exp_name pendulum_conservative_lr1e3
```