Assignment 2: Policy Gradients

Due February 25, 11:59 pm

1 Introduction

The goal of this assignment is to experiment with policy gradient and its variants, including variance reduction tricks such as implementing reward-to-go and neural network baselines.

The starter code for this assignment can be found at

https://github.com/berkeleydeeprlcourse/homework_spring2026/tree/main/hw2

Each individual training run in this assignment may take from a few seconds to a few (up to ten) minutes on a modern laptop CPU. We generally recommend using laptop CPUs rather than cloud GPUs for this assignment, as the environments and models are small enough and often run faster on CPUs. You can also save your cloud credits for later assignments that require heavier computation.

2 Review

2.1 Policy Gradient

Recall that the reinforcement learning objective is to learn a θ^* that maximizes the objective function:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}\big[r(\tau)\big] \tag{1}$$

where each rollout τ is of length H, as follows:

$$\begin{aligned}\pi_\theta(\tau) &= p(s_0, a_0, \ldots, s_{H-1}, a_{H-1}) \\ &= p(s_0)\,\pi_\theta(a_0 \mid s_0) \prod_{t=1}^{H-1} p(s_t \mid s_{t-1}, a_{t-1})\,\pi_\theta(a_t \mid s_t),\end{aligned} \tag{2}$$

and

$$r(\tau) = r(s_0, a_0, \ldots, s_{H-1}, a_{H-1}) = \sum_{t=0}^{H-1} r(s_t, a_t). \tag{3}$$

The policy gradient approach is to directly take the gradient of this objective:

$$\nabla_\theta J(\theta) = \nabla_\theta \int \pi_\theta(\tau)\, r(\tau)\, d\tau \tag{4}$$

$$= \int \pi_\theta(\tau)\, \nabla_\theta \log \pi_\theta(\tau)\, r(\tau)\, d\tau \tag{5}$$

$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau)\, r(\tau)]\,. \tag{6}$$

In practice, the expectation over trajectories τ can be approximated from a batch of N sampled trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau^i)\, r(\tau^i) \tag{7}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) \right) \left(\sum_{t=0}^{H-1} r(s_t^i, a_t^i) \right). \tag{8}$$

Here we see that the policy π_θ is a probability distribution over the action space, conditioned on the state. In the agent-environment loop, the agent samples an action a_t from $\pi_\theta(\cdot \mid s_t)$ and the environment responds with a reward $r(s_t, a_t)$.

2.2 Variance Reduction

2.2.1 Reward-To-Go

One way to reduce the variance of the policy gradient is to exploit causality: the notion that the policy cannot affect rewards in the past. This yields the following modified objective, where the sum of rewards here does not include the rewards achieved prior to the time step at which the policy is being queried. This sum of rewards is a sample estimate of the Q function, and is referred to as the "reward-to-go."

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) \left(\sum_{t'=t}^{H-1} r(s_{t'}^i, a_{t'}^i) \right). \tag{9}$$

2.2.2 Discounting

Multiplying a discount factor γ to the rewards can be interpreted as encouraging the agent to focus more on rewards that are closer in time, and less on rewards that are further in the future. This can also be thought of as a means for reducing variance (because there is more variance possible when considering futures that are further into the future).

In practice, the discount factor can be incorporated in two ways. The first way applies the discount on the rewards from the full trajectory:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) \left(\sum_{t'=0}^{H-1} \gamma^{t'} r(s_{t'}^i, a_{t'}^i) \right), \tag{10}$$

and the second way applies the discount on the "reward-to-go":

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) \left(\sum_{t'=t}^{H-1} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) \right). \tag{11}$$

While these two formulations look similar, they have different effects.

In the first formulation, discounting is applied in an *absolute* sense. Each reward is weighted simply according to how far it occurs in the full trajectory. As a result, the policy mostly focuses on earlier parts of the trajectory and gradually care less about states that are visited later.

In the second formulation, discounting is applied in an *relative* sense (with respect to the current time step t). This uniformly encourages the policy to prefer actions that lead to more immediate rewards, regardless of when the state occurs in the trajectory.

In practice, we almost always use the second formulation (do you see why?).

2.2.3 Baseline

Another variance reduction method is to subtract a baseline (a constant with respect to τ) from the sum of rewards:

$$\nabla_\theta J(\theta) = \nabla_\theta \, \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \big[r(\tau) - b \big]. \tag{12}$$

This leaves the policy gradient unbiased because

$$\nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[b] = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \big[\nabla_\theta \log \pi_\theta(\tau) \cdot b \big] = 0.$$

In this assignment, we implement a value function V_φ^π which acts as a state-dependent baseline. This value function is trained to approximate the sum of future rewards starting from a particular state:

$$V_\varphi^\pi(s_t) \approx \sum_{t'=t}^{H-1} \mathbb{E}_{\pi_\theta} \big[\gamma^{t'-t} r(s_{t'}, a_{t'}) \mid s_t \big], \tag{13}$$

so the approximate policy gradient now looks like:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) \left(\sum_{t'=t}^{H-1} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i) - V_\varphi^\pi(s_t^i) \right). \tag{14}$$

2.2.4 Generalized Advantage Estimation

The quantity

$$\left(\sum_{t'=t}^{H-1} \gamma^{t'-t} r(s_{t'}, a_{t'})\right) - V_\varphi^\pi(s_t)$$

can be interpreted as an estimate of the advantage function:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t), \tag{15}$$

where $Q^\pi(s_t, a_t)$ is estimated using Monte Carlo returns and $V^\pi(s_t)$ is estimated using the learned value function V_φ^π.

We can further reduce variance by using V_φ^π in place of Monte Carlo returns to estimate the advantage as:

$$A^\pi(s_t, a_t) \approx \delta_t = r(s_t, a_t) + \gamma V_\varphi^\pi(s_{t+1}) - V_\varphi^\pi(s_t), \tag{16}$$

with the edge case $\delta_{H-1} = r(s_{H-1}, a_{H-1}) - V_\varphi^\pi(s_{H-1})$. However, this comes at the cost of introducing bias to our policy gradient estimate, due to modeling errors in V_φ^π. We can instead use a combination of n-step Monte Carlo returns and V_φ^π to estimate the advantage:

$$A_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) + \gamma^n V_\varphi^\pi(s_{t+n+1}) - V_\varphi^\pi(s_t). \tag{17}$$

Increasing n incorporates the Monte Carlo returns more heavily in the advantage estimate, which lowers bias and increases variance, while decreasing n does the opposite. Note that $n = H - t - 1$ recovers the unbiased (higher-variance) Monte Carlo advantage estimate, while $n = 0$ recovers the lower-variance but higher-bias estimate δ_t.

We can combine multiple n-step advantage estimates as an exponentially weighted sum, which is known as generalized advantage estimation (GAE). Let $\lambda \in [0, 1]$. Then:

$$A_{\text{GAE}}^\pi(s_t, a_t) = \frac{1-\lambda}{1-\lambda^{H-t-1}} \sum_{n=1}^{H-t-1} \lambda^{n-1} A_n^\pi(s_t, a_t), \tag{18}$$

where $\frac{1-\lambda}{1-\lambda^{H-t-1}}$ is a normalizing constant. Note that a higher λ emphasizes advantage estimates with higher values of n, and a lower λ does the opposite. Thus, λ serves as a control for the bias-variance tradeoff, where increasing λ decreases bias and increases variance. In the infinite-horizon case ($H = \infty$), one can show:

$$A_{\text{GAE}}^\pi(s_t, a_t) = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} A_n^\pi(s_t, a_t) \tag{19}$$

$$= \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'}. \tag{20}$$

where we have omitted the derivation for brevity (see the GAE paper for details: `https://arxiv.org/pdf/1506.02438.pdf`).

In the finite-horizon case, we can write:

$$A_{\text{GAE}}^\pi(s_t, a_t) = \sum_{t'=t}^{H-1} (\gamma\lambda)^{t'-t} \delta_{t'}, \tag{21}$$

which serves as a way we can efficiently implement the generalized advantage estimator, since we can recursively compute:

$$A_{\text{GAE}}^\pi(s_t, a_t) = \delta_t + \gamma\lambda A_{\text{GAE}}^\pi(s_{t+1}, a_{t+1}). \tag{22}$$

2.3 Advantage Normalization

Another common trick in policy gradient methods is to normalize the advantages to have mean 0 and standard deviation 1 within a batch. While this is technically biased (because the normalization depends on the batch of trajectories that we sampled; to see this, think about the extreme case where we use a batch of size 1, in which case the normalized advantage is always 0), it is often an effective heuristic for improving performance in practice. Also, the bias becomes negligible when the batch size is large enough.

Concretely, advantage normalization transforms the advantage estimates in a batch from

$$A^\pi(s_t, a_t) \tag{23}$$

to

$$\frac{A^\pi(s_t, a_t) - \mu}{\sigma + \varepsilon}, \tag{24}$$

where μ and σ are the mean and standard deviation of the advantage estimates within a batch of trajectories, and ε is a small constant (e.g., $\varepsilon = 10^{-8}$) added to the denominator for numerical stability.

3 Vanilla Policy Gradients

3.1 Implementation

You will be implementing two different return estimators within `pg_agent.py`. The first ("Case 1" within `_calculate_q_vals`) uses the discounted cumulative return of the full trajectory and corresponds to the "vanilla" form of the policy gradient (Equation (10)):

$$r(\tau^i) = \sum_{t'=0}^{H-1} \gamma^{t'} r(s^i_{t'}, a^i_{t'}). \tag{25}$$

The second ("Case 2") uses the "reward-to-go" formulation from Equation (11):

$$r(\tau^i) = \sum_{t'=t}^{H-1} \gamma^{t'-t} r(s^i_{t'}, a^i_{t'}). \tag{26}$$

Implement these return estimators as well as the remaining sections marked `TODO` in the code. For the small-scale experiments, you may skip those sections that are run only if `use_baseline` is `True`; we return to baselines in Section 4.

3.2 Experiments

Experiment 1 (CartPole). Run multiple experiments with the PG algorithm on the discrete `CartPole-v0` environment, using the following commands:

```
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
```

What's happening here:

- `-n`: Number of iterations.
- `-b`: Batch size (number of state-action pairs sampled while acting according to the current policy at each iteration).

- `-rtg`: If present, sets `reward_to_go=True`. Otherwise, `reward_to_go=False` by default.
- `-na`: If present, sets `normalize_advantages` to True, normalizing the advantages to have mean 0 and standard deviation 1 within a batch.
- `--exp_name`: name for the experiment, used in the data logging directory name.

Various other command line arguments allow you to set batch size, learning rate, network architecture, and more.

Deliverables:

- Submit the logs of all eight experiments above. The best configuration of `CartPole` in both large and small batch cases should converge to a maximum return of 200.
- Create two plots:
  - In the first plot, compare the learning curves (average return vs. number of environment steps) for the experiments prefixed with `cartpole` without `_lb`. (The small batch experiments.)
  - In the second plot, compare the learning curves for the experiments prefixed with `cartpole_lb`. (The large batch experiments.)

  For all plots, the x-axis should be the number of environment steps, logged as `Train_EnvstepsSoFar` (*not* number of policy gradient iterations).
- Answer the following questions briefly:
  - Which value estimator has better performance without advantage normalization: the trajectory-centric one, or the one using reward-to-go?
  - Between the two value estimators, why do you think one is generally preferred over the other?
  - Did advantage normalization help?
  - Did the batch size make an impact?
- Provide the exact command line configurations you used to run your experiments, including any parameters changed from defaults.

4 Using a Neural Network Baseline

4.1 Implementation

You will now implement a value function as a state-dependent neural network baseline. This will require filling in some `TODO` sections skipped in Section 3. In particular:

- This neural network will be trained in the `update` method of `PGAgent` along with the policy gradient update.
- In `pg_agent.py:_estimate_advantage`, the predictions of this network will be subtracted from the reward-to-go to yield an estimate of the advantage. This implements $\left(\sum_{t'=t}^{H-1} \gamma^{t'-t} r(s^i_{t'}, a^i_{t'})\right) - V^\pi_\varphi(s^i_t)$.
- We will train the baseline network for *multiple* gradient steps for each policy update, determined by the parameter `baseline_gradient_steps`.

4.2 Experiments

Experiment 2 (HalfCheetah). Use your baselined policy gradient implementation to learn a controller for `HalfCheetah-v4`.

Run the following commands:

```
# 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
```

You might notice that we omitted `-na` (normalize advantages). That's because in reality, advantage normalization is a very powerful trick, and eliminates the need for a baseline most of the simple environments that we test in.

Deliverables:

- Submit the logs of the two experiments above. The baselined version should achieve an average return above 300 at the end of training. The performance may vary with different random seeds, and it is possible that you might get a bad run that does not achieve this even with a correct implementation. However, it should not take more than 2–3 runs to pass this threshold. If it does, you might want to revisit your implementation.
- Create two plots:
  - A learning curve for the baseline loss.
  - A learning curve for the eval return.
- Run another experiment with a decreased number of baseline gradient steps (`-bgs`) or baseline learning rate (`-blr`). How does this affect (a) the baseline learning curve and (b) the performance of the policy?
- Provide the exact command line configurations you used to run your experiments, including any parameters changed from defaults.

Optional:

- Add `-na` back to see how much it improves things. Also, set `video_log_freq` to `10`, and see some videos of your HalfCheetah walking along in your WandB!

5 Implementing Generalized Advantage Estimation

You will now use the value function you previously implemented to implement a simplified version of GAE-λ. This will require filling in the remaining `TODO` section in `pg_agent.py:_estimate_advantage`.

Experiment 3 (LunarLander-v2). Use your implementation of policy gradient with generalized advantage estimation to learn a controller for a version of `LunarLander-v2` with noisy actions. Search over $\lambda \in [0, 0.95, 0.98, 0.99, 1]$ to replace `<lambda>` below. **Do not** change any of the other hyperparameters (e.g., batch size, learning rate).

```
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 <lambda> --exp_name lunar_lander_lambda<lambda>
```

Deliverables:

- Submit the logs of the five experiments above with $\lambda \in \{0, 0.95, 0.98, 0.99, 1\}$. The best run should achieve an average return above 150 at least once during training. Results may have some variance and you might need to give it another try to get a good run.
- Provide a single plot with the learning curves for the LunarLander-v2 experiments that you tried. Describe in words how λ affected task performance.
- Consider the parameter λ. What does $\lambda = 0$ correspond to? What about $\lambda = 1$? Relate this to the task performance in `LunarLander-v2` in one or two sentences.
- Provide the exact command line configurations you used to run your experiments, including any parameters changed from defaults.

6 Hyperparameters and Sample Efficiency

One criticism of policy gradient methods is that they tend to be very *sample inefficient*. Examining your results for the previous problems, you'll notice that your algorithm takes hundreds of thousands or millions of steps in the environment before it is able to learn a good policy.

While improving sample inefficiency is in general an open challenge, we can do a lot better by just picking better hyperparameters! During training, we have to choose many hyperparameters and settings:

1. Discount factor
2. Network size
3. Batch size (super small batch sizes may have high variance, but very large batch sizes waste a lot of samples because you must recollect the entire batch for every gradient step)
4. Learning rate
5. Whether to use return-to-go
6. Whether to normalize advantages
7. Whether to use GAE, and if we do, what value of λ to use

Experiment 4 (InvertedPendulum). Consider the default hyperparameters below:

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

Your task is to tune hyperparameters so that your implementation reaches maximum performance (return of 1000) in fewer **environment steps** than this default setting (note: this is **not** the same as minimizing number of policy gradient iterations: one policy gradient iteration corresponds to `batch_size` environment steps). Concretely, try to find hyperparameters that reach a return of 1000 within 100K environment steps! Feel free to change any of the hyperparameters mentioned above (or any other hyperparameters you think are relevant). You can set `exp_name` as you like, but it must start with `pendulum` for us to identify it.

Deliverables:

- Submit the log of the run with the best performance. The best run should reach an average return of 1000 at least once within 100K environment steps. Again, results may vary with different random seeds. (**Note:** For this reason, in actual RL research, it is generally *very* important to report results averaged over multiple random seeds to ensure statistical significance and avoid cherry-picking! However, for this assignment, we only require you to submit your best run to reduce computational burden.)
- Provide the best set of hyperparameters on `InvertedPendulum-v4` and the exact command line configuration. Briefly discuss which hyperparameters mattered in your tuning process.
- Show learning curves for the average returns with your hyperparameters and with the default settings, with environment steps on the x-axis.

7 Submitting the Code and Experiment Runs

In order to turn in your code and experiment logs, create a directory that contains the following:

- A directory named `exp` with all the experiment runs from this assignment. We identify the experiment runs by their prefixes (up to the "`_sd`" part in the directory name), and they should exactly match the tree structure below. For the `InvertedPendulum` experiment, we check the prefix up to `pendulum`, as mentioned in the previous section.
- The `src` folder with all the `.py` files, with the same names and directory structure as the original homework repository.

The unzipped version of your submission should have the following file structure. **Make sure that the `exp` directory has the exact structure as shown below and that the submit.zip file is below 100MB.** Do **not** nest the `exp` and `src` directories within another parent directory.

```
submit.zip
├── exp
│   ├── CartPole-v0_cartpole_lb_na_sd1_20260101_012345
│   │   ├── agent.pt
│   │   ├── flags.json
│   │   ├── log.csv
│   │   └── log.pkl
│   ├── CartPole-v0_cartpole_lb_rtg_na_sd1_20260101_012345
│   │   └── ...
│   ├── CartPole-v0_cartpole_lb_rtg_sd1_20260101_012345
│   ├── CartPole-v0_cartpole_lb_sd1_20260101_012345
│   ├── CartPole-v0_cartpole_na_sd1_20260101_012345
│   ├── CartPole-v0_cartpole_rtg_na_sd1_20260101_012345
│   ├── CartPole-v0_cartpole_rtg_sd1_20260101_012345
│   ├── CartPole-v0_cartpole_sd1_20260101_012345
│   ├── HalfCheetah-v4_cheetah_baseline_sd1_20260101_012345
│   ├── HalfCheetah-v4_cheetah_sd1_20260101_012345
│   ├── InvertedPendulum-v4_pendulum_sd1_20260101_012345
│   ├── LunarLander-v2_lunar_lander_lambda0_sd1_20260101_012345
│   ├── LunarLander-v2_lunar_lander_lambda0.95_sd1_20260101_012345
│   ├── LunarLander-v2_lunar_lander_lambda0.98_sd1_20260101_012345
│   ├── LunarLander-v2_lunar_lander_lambda0.99_sd1_20260101_012345
│   └── LunarLander-v2_lunar_lander_lambda1_sd1_20260101_012345
├── src
│   ├── agents
│   │   ├── ...
│   │   ├── pg_agent.py
│   │   └── ...
│   └── ...
├── pyproject.toml
├── uv.lock
└── README.md
```

Turn in your assignment on Gradescope. Upload the zip file with your code and log files to **HW2 Code**, and upload the PDF of your report to **HW2 Report**.