Deep Reinforcement Learning (Fall 2025)
Assignment 3

October 19th, 2025

This assignment should be complete it by yourself or in a group of 2. It will due at 23:59pm Nov 2nd, 2025. You should submit the reports, code and running logs on *Canvas*.

Instructions

In this assignment, you will implement Policy Gradient and its variants, including variance reduction tricks such as implementing reward-to-go and neural network baselines.

You will use the same environment as in the second assignment, Cartpole-v1, to conduct the experiments.

Recall that the objective of Policy Gradient is to learn a θ^* that maximizes the objective function:

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

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

$$\pi_\theta(\tau) = p(s_0, a_0, ..., s_{T-1}, a_{T-1}) = p(s_0)\pi_\theta(a_0|s_0) \prod_{t=1}^{T-1} p(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_t)$$

and

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

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

$$\nabla_\theta J(\theta) = \nabla_\theta \int \pi_\theta(\tau) r(\tau) d\tau$$

$$= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau$$

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

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

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

$$= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \left(\sum_{t=0}^{T-1} r(s_{it}, a_{it}) \right). \tag{2}$$

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 $\pi_\theta(\cdot|s_t)$ and the environment responds with a reward $r(s_t, a_t)$.

0.1 Variance Reduction

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."

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\sum_{t'=t}^{T-1} r(s_{it'}, a_{it'}) \right). \tag{3}$$

Discount. Multiplying a discount factor γ to the rewards can be interpreted as encouraging the agent to focus more on the rewards that are closer in time, and less on the 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).

The discount can be applied to the full trajectory:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \left(\sum_{t'=0}^{T-1} \gamma^{t'} r(s_{it'}, a_{it'}) \right). \tag{4}$$

or on the "reward-to-go:"

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\sum_{t'=t}^{T-1} \gamma^{t'} r(s_{it'}, a_{it'}) \right). \tag{5}$$

In practice, the term $\gamma^{t'}$ in Equation 5 is close to 0 and will make the optimization process ignore the policy learning at the large time step t'. Therefore, we usually adopt another surrogate to approximate the original form:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) \right). \tag{6}$$

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

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[r(\tau) - b]. \tag{7}$$

This leaves the policy gradient unbiased because

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

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

$$V_\phi^{\pi_\theta}(s_t) \approx \sum_{t'=t}^{T-1} \mathbb{E}_{\pi_\theta}[\gamma^{t'-t} r(s_{t'}, a_{t'})|s_t], \tag{8}$$

so the approximate policy gradient now looks like this:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) \right) - V_\phi^\pi(s_{it}) \right). \tag{9}$$

1 Problem 1

1.1 Policy Gradient

In this section you will implement Policy Gradient with two different return estimators within `src/pg_agent.py`.

* The first is to use the discounted cumulative return of the full trajectory, which corresponds to the "vanilla" form of the Policy Gradient.
  You should implement the function `_discounted_return()` in `src/pg_agent.py`, which computes the discounted cumulative return for a single episode.

* The second is to use "reward-to-go" formulation, which exploit causality (the notion that the policy cannot affect rewards in the past) and does not include the rewards achieved prior to the timestep at which the policy is being queried.
  You should implement the function `_discounted_reward_to_go()` in `src/pg_agent.py`, which computes the reward-to-go for a single episode.

Then you will utilize the above two functions and implement the function `_calculate_q_vals()` in `src/pg_agent.py`, which computes the returns for a batch of episodes. There are two cases in the function to complete.

You need to implement the function `get_action()` and `forward()` in `src/policies.py` then.

1.2 Experiments

Implement the function `sample_trajectory()` in `src/utils.py` to use the policy to collect online data for training.

Then you need to implement the function `update()` for `MLPPolicyPG` in `src/policies.py` to update the policy.

Implement the condition `if self.normalize_advantages` in the function in `_estimate_advantage()` in `src/pg_agent.py`. Normalize the advantages to have a mean of zero and a standard deviation of one within the batch. (In this section, the advantages will simply be the computed returns.)

You should now be able to run the following command lines:

```
python run.py --env_name CartPole-v1 -n 200 -b 1000 --exp_name cartpole
python run.py --env_name CartPole-v1 -n 200 -b 1000 -rtg --exp_name cartpole_rtg
python run.py --env_name CartPole-v1 -n 200 -b 1000 -na --exp_name cartpole_na
python run.py --env_name CartPole-v1 -n 200 -b 1000 -rtg -na --exp_name cartpole_rtg_na
python run.py --env_name CartPole-v1 -n 200 -b 4000 --exp_name cartpole_lb
python run.py --env_name CartPole-v1 -n 200 -b 4000 -rtg --exp_name cartpole_lb_rtg
python run.py --env_name CartPole-v1 -n 200 -b 4000 -na --exp_name cartpole_lb_na
python run.py --env_name CartPole-v1 -n 200 -b 4000 -rtg -na --exp_name cartpole_lb_rtg_na
```

Here,

- -n: The number of iterations
- -b: Batch size, which is the state-action pairs sampled while acting according to the current policy at each iteration.
- -rtg: A Flag. If present, compute reward-to-go. Otherwise, compute total return.
- -na: A Flag. If present, normalize the returns in a batch to have a mean of zero and standard deviation of one within a batch.

- –exp_name: Name of the experiment, which goes into the name for the logging directory.

You could also change other command line arguments as you like. You can see the arguments in `run.py`.

1.3 Deliveries

1.3.1

Create two graphs:

- In the first graph, compare the learning curves (average return vs. number of environment steps) for the experiments running with batch size of 1000. (The small batch experiments.) (15 pts)

– In the second graph, compare the learning curves for the experiments running with batch size of 4000. (The large batch experiments.) (15 pts)

Note that the x-axis should be number of environment steps, not number of policy gradient iterations.

1.3.2

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? Why? (10 pts)

– Did advantage normalization help? (10 pts)

– Did the batch size make an impact? (10 pts)

Provide the exact command line configurations you used to run your experiments, including any parameters changed from their defaults.

The best configuration in both the small and large batch size cases should converge to a maximum score of 500.

2 Implement a Neural Network Baseline

2.1 Neural Network Baseline

In this section, you will implement a value function as a state-dependent neural network baseline. Then you could compute the advantages using the returns computed in the last section and the baseline.

Implement the function `forward()` in `src/critics.py`, as well as the function `_estimate_advantage(` in `src/pg_agent.py`.

Then implement the function `update()` in `src/critics.py`.

Implement the function `update()` in `src/pg_agent.py`. We will train the baseline network for multiple gradient steps for each policy update, determined by the parameter `baseline_gradient_steps`.

2.2 Experiments

You should now be able to run the following command lines:

```
python run.py --env_name CartPole-v1 -n 100 -b 5000 -rtg --exp_name cartpole_rtg_no_baseline
python run.py --env_name CartPole-v1 -n 100 -b 5000 -rtg -na --use_baseline --exp_name cartpole_na_rtg_baseline
python run.py --env_name CartPole-v1 -n 100 -b 5000 -rtg --exp_name cartpole_rtg_no_baseline
python run.py --env_name CartPole-v1 -n 100 -b 5000 -rtg -na --use_baseline --exp_name cartpole_na_rtg_baseline
```

Here,

- -n: The number of iterations
- -b: Batch size, which is the state-action pairs sampled while acting according to the current policy at each iteration.
- -rtg: A Flag. If present, compute reward-to-go. Otherwise, compute total return.
- -na: A Flag. If present, normalize the returns in a batch to have a mean of zero and standard deviation of one within a batch.
- –exp_name: Name of the experiment, which goes into the name for the logging directory.

You could also change other command line arguments as you like.

2.3 Deliveries

- Plot a learning curve for the baseline loss. (5 pts)
- Plot a learning curve for the evaluation return. You should expect to converge to the maximum reward of 500. (15 pts)
- Run another experiment with a decreased number of baseline gradient steps (-bgs in command line) and/or baseline learning rate (-blr in command line). How does this affect (a) the baseline learning curve and (b) the performance of the policy? (15 pts)
- How does the command line argument -na influence the performance? Why is that the case? (5 pts)

2.4 Bonus (20 pts)

You could also run your code on the environment `HalfCheetah`. Simply change the env_name to `HalfCheetah-v4`: `--env_name HalfCheetah-v4`.

You should do the same thing as you have done for Cartpole-v1. You should expect to achieve an average return over 300 for the baselined version in this environment.