Assignment 2: Policy Gradients

Due September 28, 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 startercode can be found at

`https://github.com/berkeleydeeprlcourse/homework_fall2020/tree/master/hw2`

2 Review

2.1 Policy gradient

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

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

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:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \int \pi_\theta(\tau) r(\tau) d\tau & (2)\\ &= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau. & (3)\\ &= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) r(\tau)] & (4)\\ & & (5) \end{aligned}$$

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

$$\begin{aligned} \nabla_\theta J(\theta) &\approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i) & (6)\\ &= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \left(\sum_{t=0}^{T-1} r(s_{it}, a_{it}) \right). & (7) \end{aligned}$$

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 $r(s_t, a_t)$.

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."

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\sum_{t'=t}^{T-1} r(s_{it'}, a_{it'}) \right). \tag{8}$$

2.2.2 Discounting

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). We saw in lecture that the discount factor can be incorporated in two ways, as shown below.

The first way applies the discount on the rewards from full trajectory:

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

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

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

.

2.2.3 Baseline

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

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[r(\tau) - b \right]. \tag{11}$$

This leaves the policy gradient unbiased because

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

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:

$$V_\phi^\pi(s_t) \approx \sum_{t'=t}^{T-1} \mathbb{E}_{\pi_\theta} \left[r(s_{t'}, a_{t'}) | s_t \right], \tag{12}$$

so the approximate policy gradient now looks like this:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) \right) - V_\phi^\pi \left(s_{it} \right) \right). \tag{13}$$

3 Overview of Implementation

3.1 Files

To implement policy gradients, we will be building up the code that we started in homework 1. All files needed to run your code are in the `hw2` folder, but there will be some blanks you will fill with your solutions from homework 1. These locations are marked with `# TODO: get this from hw1` and are found in the following files:

- `infrastructure/rl_trainer.py`
- `infrastructure/utils.py`
- `policies/MLP_policy.py`

After bringing in the required components from the previous homework, you can begin work on the new policy gradient code. These placeholders are marked with `TODO`, located in the following files:

- `agents/pg_agent.py`
- `policies/MLP_policy.py`

The script to run the experiments is found in `scripts/run_hw2.py` (for the local option) or `scripts/run_hw2.ipynb` (for the Colab option).

3.2 Overview

As in the previous homework, the main training loop is implemented in `infrastructure/rl_trainer.py`.

The policy gradient algorithm uses the following 3 steps:

1. **Sample trajectories** by generating rollouts under your current policy.
2. **Estimate returns and compute advantages**. This is executed in the `train` function of **pg_agent.py**
3. **Train/Update parameters**. The computational graph for the policy and the baseline, as well as the update functions, are implemented in `policies/MLP_policy.py`.

4 Implementing Policy Gradients

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 9):

$$r(\tau_i) = \sum_{t'=0}^{T-1} \gamma^{t'} r(s_{it'}, a_{it'}). \tag{14}$$

The second ("Case 2") uses the "reward-to-go" formulation from Equation 10:

$$r(\tau_i) = \sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}). \tag{15}$$

Note that these differ only by the starting point of the summation.

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 `nn_baseline` is `True`; we will return to baselines in Section 6. (These sections are in `MLPPolicyPG:update` and `PGAgent:estimate_advantage`.)

5 Small-Scale Experiments

After you have implemented all non-baseline code from Section 4, you will run two small-scale experiments to get a feel for how different settings impact the performance of policy gradient methods.

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

```
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 \
    -dsa --exp_name q1_sb_no_rtg_dsa

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 \
    -rtg -dsa --exp_name q1_sb_rtg_dsa

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 \
    -rtg --exp_name q1_sb_rtg_na

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 5000 \
    -dsa --exp_name q1_lb_no_rtg_dsa

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 5000 \
    -rtg -dsa --exp_name q1_lb_rtg_dsa

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 5000 \
    -rtg --exp_name q1_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).
- `-dsa` : Flag: if present, sets `standardize_advantages` to False. Otherwise, by default, standardizes advantages to have a mean of zero and standard deviation of one.
- `-rtg` : Flag: if present, sets `reward_to_go=True`. Otherwise, `reward_to_go=False` by default.
- `--exp_name` : Name for experiment, which goes into the name for the data logging directory.

Various other command line arguments will allow you to set batch size, learning rate, network architecture, and more. You can change these as well, but keep them fixed between the 6 experiments mentioned above.

Deliverables for report:

- Create two graphs:
  - In the first graph, compare the learning curves (average return at each iteration) for the experiments prefixed with `q1_sb_`. (The small batch experiments.)
  - In the second graph, compare the learning curves for the experiments prefixed with `q1_lb_`. (The large batch experiments.)
- Answer the following questions briefly:
  - Which value estimator has better performance without advantage-standardization: the trajectory-centric one, or the one using reward-to-go?
  - Did advantage standardization help?
  - Did the batch size make an impact?

- Provide the exact command line configurations (or `#@params` settings in Colab) you used to run your experiments, including any parameters changed from their defaults.

What to Expect:

- The best configuration of CartPole in both the large and small batch cases should converge to a maximum score of 200.

Experiment 2 (InvertedPendulum). Run experiments on the `InvertedPendulum-v2` continuous control environment as follows:

```
python cs285/scripts/run_hw2.py --env_name InvertedPendulum-v2 \
    --ep_len 1000 --discount 0.9 -n 100 -l 2 -s 64 -b <b*> -lr <r*> -rtg \
    --exp_name q2_b<b*>_r<r*>
```

where your task is to find the smallest batch size `b*` and largest learning rate `r*` that gets to optimum (maximum score of 1000) in less than 100 iterations. The policy performance may fluctuate around 1000; this is fine. The precision of `b*` and `r*` need only be one significant digit.

Deliverables:

- Given the `b*` and `r*` you found, provide a learning curve where the policy gets to optimum (maximum score of 1000) in less than 100 iterations. (This may be for a single random seed, or averaged over multiple.)
- Provide the exact command line configurations you used to run your experiments.

6 Implementing Neural Network Baselines

You will now implement a value function as a state-dependent neural network baseline. This will require filling in the remaining `TODO` sections skipped in Section 4. In particular:

- This neural network will be trained in the `update` method of `MLPPolicyPG` 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 $\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'})\right) - V_\phi^\pi(s_{it})$.

7 More Complex Experiments

Note: The following tasks take quite a bit of time to train. Please start early! For all remaining experiments, use the reward-to-go estimator.

Experiment 3 (LunarLander). You will now use your policy gradient implementation to learn a controller for `LunarLanderContinuous-v2`. The purpose of this problem is to test and help you debug your baseline implementation from Section 6.

Run the following command:

```
python cs285/scripts/run_hw2.py \
    --env_name LunarLanderContinuous-v2 --ep_len 1000
    --discount 0.99 -n 100 -l 2 -s 64 -b 40000 -lr 0.005 \
    --reward_to_go --nn_baseline --exp_name q3_b40000_r0.005
```

Deliverables:

- Plot a learning curve for the above command. You should expect to achieve an average return of around 180 by the end of training.

Experiment 4 (HalfCheetah). You will be using your policy gradient implementation to learn a controller for the `HalfCheetah-v2` benchmark environment with an episode length of 150. This is shorter than the default episode length (1000), which speeds up training significantly. Search over batch sizes `b` $\in [10000, 30000, 50000]$ and learning rates `r` $\in [0.005, 0.01, 0.02]$ to replace `<b>` and `<r>` below.

```
python cs285/scripts/run_hw2.py --env_name HalfCheetah-v2 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b <b> -lr <r> -rtg --nn_baseline \
    --exp_name q4_search_b<b>_lr<r>_rtg_nnbaseline
```

Deliverables:

- Provide a single plot with the learning curves for the HalfCheetah experiments that you tried. Describe in words how the batch size and learning rate affected task performance.

Once you've found optimal values `b*` and `r*`, use them to run the following commands (replace the terms in angle brackets):

```
python cs285/scripts/run_hw2.py --env_name HalfCheetah-v2 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b <b*> -lr <r*> \
    --exp_name q4_b<b*>_r<r*>

python cs285/scripts/run_hw2.py --env_name HalfCheetah-v2 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b <b*> -lr <r*> -rtg \
    --exp_name q4_b<b*>_r<r*>_rtg

python cs285/scripts/run_hw2.py --env_name HalfCheetah-v2 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b <b*> -lr <r*> --nn_baseline \
    --exp_name q4_b<b*>_r<r*>_nnbaseline

python cs285/scripts/run_hw2.py --env_name HalfCheetah-v2 --ep_len 150 \
    --discount 0.95 -n 100 -l 2 -s 32 -b <b*> -lr <r*> -rtg --nn_baseline \
    --exp_name q4_b<b*>_r<r*>_rtg_nnbaseline
```

Deliverables: Provide a single plot with the learning curves for these four runs. The run with both reward-to-go and the baseline should achieve an average score close to 200.

8 Bonus!

Choose any (or all) of the following:

- A serious bottleneck in the learning, for more complex environments, is the sample collection time. In `infrastructure/rl_trainer.py`, we only collect trajectories in a single thread, but this process can be fully parallelized across threads to get a useful speedup. Implement the parallelization and report on the difference in training time.
- Implement GAE-λ for advantage estimation.[1] Run experiments in a MuJoCo gym environment to explore whether this speeds up training. (`Walker2d-v2` may be good for this.)
- In PG, we collect a batch of data, estimate a single gradient, and then discard the data and move on. Can we potentially accelerate PG by taking multiple gradient descent steps with the same batch of data? Explore this option and report on your results. Set up a fair comparison between single-step PG and multi-step PG on at least one MuJoCo gym environment.

9 Submission

[1] `https://arxiv.org/abs/1506.02438`

9.1 Submitting the PDF

Your report should be a document containing

(a) All graphs and answers to short explanation questions requested for Experiments 1-4.

(b) All command-line expressions you used to run your experiments.

(c) (Optionally) Your bonus results (command-line expressions, graphs, and a few sentences that comment on your findings).

9.2 Submitting the code and experiment runs

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

- A folder named `run_logs` with all the experiment runs from this assignment. These folders can be copied directly from the `cs285/data` folder. **Do not change the names originally assigned to the folders, as specified by `exp_name` in the instructions.** Video logging is disabled by default in the code, but if you turned it on for debugging, you need to run those again with `--video_log_freq -1`, or else the file size will be too large for submission.
- The `cs285` folder with all the `.py` files, with the same names and directory structure as the original homework repository (excluding the `cs285/data` folder). Also include any special instructions we need to run in order to produce each of your figures or tables in the form of a README file.

As an example, the unzipped version of your submission should result in the following file structure. **Make sure that the submit.zip file is below 15MB.**

```
submit.zip
├── run_logs
│   ├── q1_lb_rtg_na_CartPole-v0_12-09-2019_17-53-4
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   ├── q3_b40000_r0.005_LunarLanderContinuous-v2_12-09-2019_00-17-58
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   └── ...
├── cs285
│   ├── agents
│   │   ├── bc_agent.py
│   │   └── ...
│   ├── policies
│   │   └── ...
│   └── ...
├── README.md
└── ...
```

9.3 Turning it in

Turn in your assignment by the deadline on Gradescope. Uploade the zip file with your code to **HW2 Code**, and upload the PDF of your report to **HW2**.