CS294-112 Deep Reinforcement Learning HW2: Policy Gradients
due September 30th 2019, 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_fall2019/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:

$$\nabla_\theta J(\theta) = \nabla_\theta \int \pi_\theta(\tau) r(\tau) d\tau \tag{2}$$

$$= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau. \tag{3}$$

$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) r(\tau) \right] \tag{4}$$

$$\tag{5}$$

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

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i) \tag{6}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \left(\sum_{t=0}^{T-1} r(s_{it}, a_{it}) \right). \tag{7}$$

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. To clarify, all files needed to run your code are in this new repository, but if you look through the code, you'll see that it's building on top of the code structure/hierarchy that we laid out in the previous assignment.

You will need to get some portions of code from your homework 1 implementation, and copy them into this new homework 2 repository. These parts are marked in the code with `# TODO: GETTHIS from HW1`. Note that solutions for HW1 will be released on Piazza, so you can ensure that your implementation is correct. The following files have these placeholders that you need to fill in by copying code from HW1:

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

After bringing in the required components from the previous homework, you can then begin to work on the code needed for this assignment. These placeholders are marked with `TODO`, and they can be found in the following files for you to fill out:

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

Similar to the previous homework assignment, the script to run is found inside the scripts directory, and the commands to run are included in the README.

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 Vanilla Policy Gradients

We start by implementing the simplest form of policy gradient algorithms (vanilla policy gradient), as given by equation 9. In this section and the next one, you can ignore blanks in the code that are used if `self.nn_baseline` is set to `true`.

Problem 1

1. Copy code from HW1 to fill in the blanks,
   as indicated by `# TODO: GETTHIS from HW1`
   in the following files:
   `infrastructure/tf_utils.py`
   `infrastructure/utils.py`
   `infrastructure/rl_trainer.py`
2. Read and fill in the `# TODO` blanks in the `train` function in `pg_agent.py`
3. Implement estimating the return by filling in the blanks in the `calculate_q_vals` function (also in `pg_agent.py`). Use the discounted return for the full trajectory:

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

   Note that this means you only need to fill in "Case 1" inside this function (under `if not self.reward_to_go`).
4. Copy code from HW1 to fill in some blanks,
   as indicated by `# TODO: GETTHIS from HW1`
   in `policies/MLP_policy.py`
5. Define the computational graph for the policy (`define_train_op`)
   in `policies/MLP_policy.py`

5 Implementing "reward-to-go"

Problem 2 Implement the "reward-to-go" formulation (Equation 10) for estimating the q value. Here, the returns are estimated as follows:

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

Note that this is "Case 2" inside the `calculate_q_vals` function in `pg_agent.py`).

6 Experiments: Policy Gradient

After you have implemented the code from sections 4 and 5 above, you will run experiments to get a feel for how different settings impact the performance of policy gradient methods.

Problem 3. CartPole: Run multiple experiments with the PG algorithm on the discrete `CartPole-v0` environment, using the following commands:

```
python run_hw2_policy_gradient.py --env_name CartPole-v0 -n 100 -b 1000 -dsa
    --exp_name sb_no_rtg_dsa

python run_hw2_policy_gradient.py --env_name CartPole-v0 -n 100 -b 1000 -rtg -
    dsa --exp_name sb_rtg_dsa

python run_hw2_policy_gradient.py --env_name CartPole-v0 -n 100 -b 1000 -rtg
    --exp_name sb_rtg_na

python run_hw2_policy_gradient.py --env_name CartPole-v0 -n 100 -b 5000 -dsa
    --exp_name lb_no_rtg_dsa

python run_hw2_policy_gradient.py --env_name CartPole-v0 -n 100 -b 5000 -rtg -
    dsa --exp_name lb_rtg_dsa

python  run_hw2_policy_gradient.py --env_name CartPole-v0 -n 100 -b 5000 -rtg
    --exp_name lb_rtg_na
```

What's happening there:

- `-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, `standardize_advantages=True`.
- `-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 (number of hidden layers and the size of the hidden layers—for CartPole, you can use one hidden layer with 32 units), and more. You can change these as well, but keep them **FIXED** between the 6 experiments mentioned above.

A trick which is known to usually boost empirical performance by lowering variance of the estimator is to center advantages and normalize them to have mean of 0 and a standard deviation of 1.

From a theoretical perspective, this does two things:

• Makes use of a constant baseline at all timesteps for all trajectories, which does not change the policy gradient in expectation.

• Rescales the learning rate by a factor of 1/σ, where σ is the standard dev of the empirical advantages

Deliverables for report:

- Create two graphs:
  - In the first graph, compare the learning curves (average return at each iteration) for the experiments prefixed with `sb_`. (The small batch experiments.)
  - In the second graph, compare the learning curves for the experiments prefixed with `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 you used to run your experiments. (To verify your chosen values for the other parameters, such as learning rate, architecture, and so on.)

What to Expect:

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

Problem 4. InvertedPendulum: Run experiments in `InvertedPendulum-v2` continuous control environment as follows:

```
python run_hw2_policy_gradient.py --env_name  InvertedPendulum-v2 --ep_len
    1000 --discount 0.9 -n 100 -l 2 -s 64 -b <b*> -lr <r*> -rtg --exp_name
    ip_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.

7 Implementing Neural Network Baseline

For the rest of the assignment we will use "reward-to-go."

Problem 5. We will now implement a value function as a state-dependent neural network baseline. Note that there is nothing to submit for this problem, but subsequent sections will require this code.

1. In `MLP_policy.py` implement V_ϕ^π, a neural network that predicts the expected return conditioned on a state. Also implement the loss function to train this network and its update operation `self.baseline_op`.
2. In `Agent.estimate_advantage` in `pg_agent.py`, use the neural network to predict the expected state-conditioned return, standardize it to match the statistics of the current batch of "reward-to-go", and subtract this value from the "reward-to-go" to yield an estimate of the advantage. Follow the hints and guidelines in the code. This implements $\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'})\right) - V_\phi^\pi(s_{it})$.
3. In `MLPPolicyPG.update`, update the parameters of the the neural network baseline by using the Tensorflow session to call `self.baseline_op`. "Rescale" the target values for the neural network baseline to have a mean of zero and a standard deviation of one. You should now have completed all `# TODO` entries in this file.

8 Experiments: More Complex Tasks

Note: The following tasks take quite a bit of time to train. Please start early!

Problem 6: LunarLander For this problem, you will use your policy gradient implementation to solve `LunarLanderContinuous-v2`. Use an episode length of 1000. Note that depending on how you installed gym, you may need to run `pip install 'gym[all]'` in order to use this environment. The purpose of this problem is to test and help you debug your baseline implementation from Section 7 above.

Run the following command:

```
python run_hw2_policy_gradient.py --env_name LunarLanderContinuous-v2 --ep_len
    1000 --discount 0.99 -n 100 -l 2 -s 64 -b 40000 -lr 0.005 -rtg       --
   nn_baseline --exp_name ll_b40000_r0.005}
```

Deliverables:

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

Problem 7: HalfCheetah For this problem, you will use your policy gradient implementation to solve `HalfCheetah-v2`. Use an episode length of 150, which is shorter than the default of 1000 for HalfCheetah (which would speed up your 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 run_hw2_policy_gradient.py --env_name HalfCheetah-v2 --ep_len 150 --
    discount 0.95 -n 100 -l 2 -s 32 -b <b> -lr <r> --video_log_freq -1 --
    reward_to_go --nn_baseline --exp_name hc_b<b>_lr<r>_nnbaseline
```

Deliverables:

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

Once you've found suitable values of `b` and `r` among those choices (let's call them `b*` and `r*`), use `b*` and `r*` and run the following commands (remember to replace the terms in the angle brackets):

```
python run_hw2_policy_gradient.py --env_name HalfCheetah-v2 --ep_len 150 --
    discount 0.95 -n 100 -l 2 -s 32 -b <b*> -lr <r*> --exp_name hc_b<b*>_r<r*>

python run_hw2_policy_gradient.py --env_name HalfCheetah-v2 --ep_len 150 --
    discount 0.95 -n 100 -l 2 -s 32 -b <b*> -lr <r*> -rtg --exp_name hc_b<b*>
    _r<r*>

python run_hw2_policy_gradient.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
    hc_b<b*>_r<r*>

python run_hw2_policy_gradient.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 hc_b<b*>_r<r*>
```

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.

9 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-v1` 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.

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

10 Submission

10.1 Submitting the PDF

Your report should be a document containing

(a) All graphs and answers to short explanation questions requested in Problems 3, 4, 6, and 7.

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

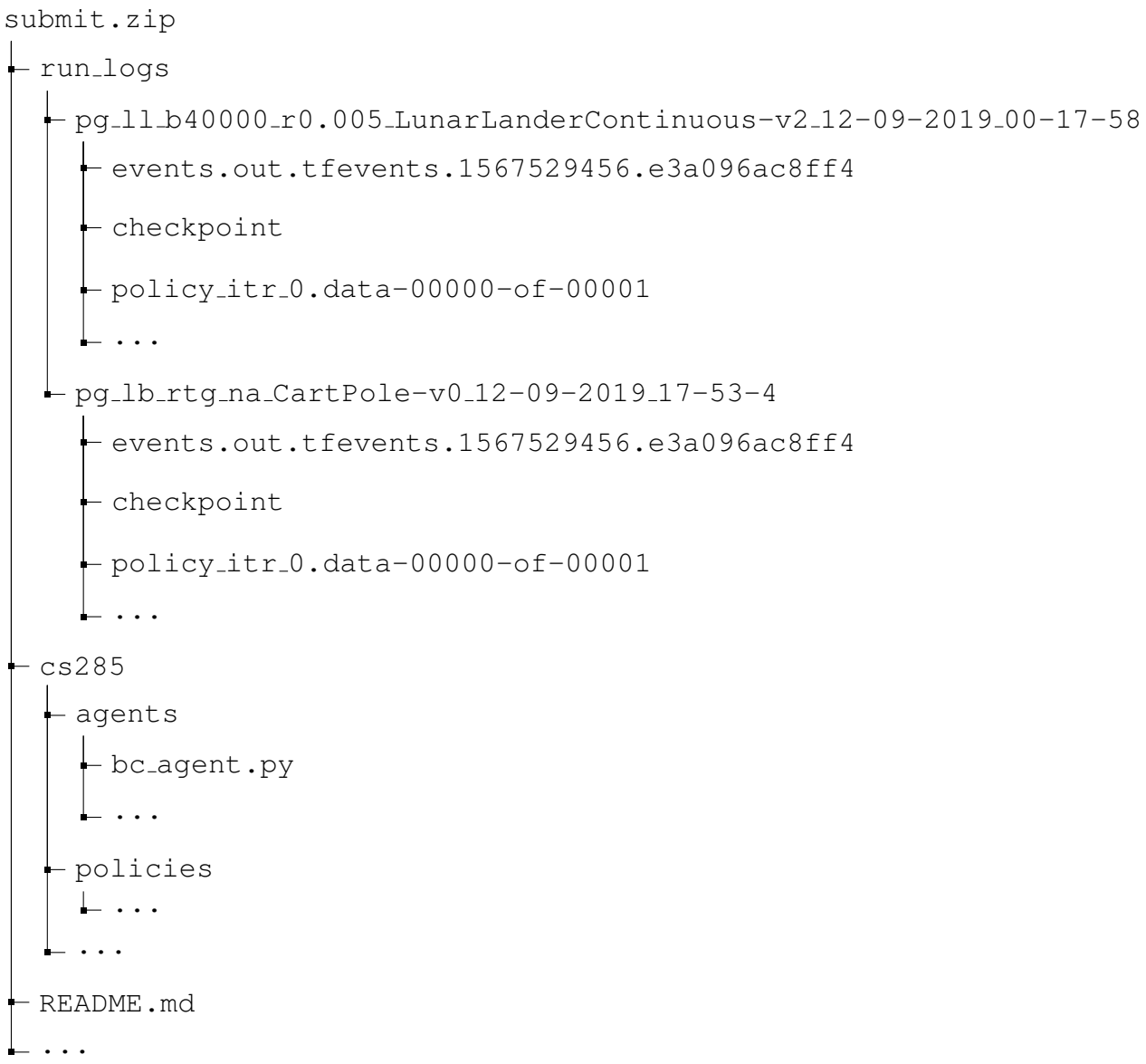
(c) (Optionally) Your bonus results (command-line expressions, graphs, and a few sentences that comment on your findings).

10.2 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: 6 runs from problem 3, your best run from problem 4, 1 run from problem 6, and the 5 runs from problem 7. 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. Also, 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 (e.g. "run python myassignment.py -sec2q1" to generate the result for Section 2 Question 1) in the form of a README file. Note that unlike the previous assignment, we will now be requiring this assignment's plotting to be done in a python script, such that running a single script like this can generate the plot.

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.**



10.3 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**.