

# CS294-112 Deep Reinforcement Learning HW2: Policy Gradients due September 20th, 11:59 pm

## 1 Introduction

The goal of this assignment is to experiment with policy gradient and its variants, including variance reduction methods. Your goals will be to set up policy gradient for both continuous and discrete environments, and implement a neural network baseline for variance reduction. You can clone the assignment at <https://github.com/berkeleydeeprlcourse/homework>

Turn in your report and code for the full homework as described in Section 7 by September 20th.

## 2 Code Setup

The only file you need to modify in this homework is `train_pg.py`. The files `logz.py` and `plots.py` are utility files; while you should look at them to understand their functionality, you will not modify them.

The function `train_PG` is used to perform the actual training for policy gradient. The parameters passed into this function specify the algorithm's hyperparameters and environment.

After you fill in the blanks, you should be able to just run `python train_pg.py` with some command line options to perform the experiments. To visualize the results, you can run `python plot.py path/to/logdir`. (Full documentation for the plotter can be found in `plot.py`.)

### 3 Building Networks

Implement the utility function, `build_mlp`, which will build a feedforward neural network with fully connected units. Test it to make sure that it produces outputs of the expected size and shape. **You do not need to include anything in your write-up about this**, it will just make your life easier.

## 4 Implement Policy Gradient

The `train_PG.py` file contains an incomplete implementation of policy gradient, and you will finish the implementation. The file has detailed instructions on which pieces you will write for this section.

### 4.1 Background

#### 4.1.1 Reward to Go

Recall that the policy gradient  $g$  can be expressed as the expectation of a few different expressions. These result in different ways of forming the sample estimate for  $g$ . Here, you will implement two ways, controlled by a flag in the code called `reward-to-go`.

1. Way one: trajectory-centric policy gradients, for which `reward-to-go=False`. Here, we compute

$$\begin{aligned} g_\theta &= E_{\tau \sim \pi_\theta} [\nabla_\theta \log P(\tau|\pi_\theta) R(\tau)] \\ &\approx \frac{1}{|D|} \sum_{\tau \in D} \nabla_\theta \log P(\tau|\pi_\theta) R(\tau) \\ &= \frac{1}{|D|} \sum_{\tau \in D} \left( \sum_{t=0}^T \nabla_\theta \log \pi_\theta(a_t|s_t) \right) R(\tau), \end{aligned}$$

where  $\tau = (s_0, a_0, s_1, \dots)$  is a trajectory,  $D$  is a dataset of trajectories collected on policy  $\pi_\theta$ ,  $\theta$  is the set of parameters for the policy, and  $R(\tau) = \sum_{t=0}^T \gamma^t r_t$  is the discounted sum of rewards along a trajectory.

2. Way two: state/action-centric policy gradients, for which `reward-to-go=True`.

Here, we compute

$$g_{\theta} = \underset{\tau \sim \pi_{\theta}}{E} \left[ \sum_{t=0}^T \gamma^t \nabla_{\theta} \log \pi(a_t | s_t) Q^{\pi}(s_t, a_t) \right]$$

$$\approx \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^T \gamma^t \nabla_{\theta} \log \pi(a_t | s_t) \sum_{t'=t}^T \gamma^{t'-t} r_{t'}.$$

The flag `reward-to-go` refers to the fact that in this case, we push up the probability of picking action  $a_t$  in state  $s_t$  in proportion to the ‘reward-to-go’ from that state-action pair—the sum of rewards achieved by starting in  $s_t$ , taking action  $a_t$ , and then acting according to the current policy forever after.

### 4.1.2 Advantage Normalization

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  $1/\sigma$ , where  $\sigma$  is the standard dev of the empirical advantages.

## 4.2 Instructions

After you have completed the code, you will run experiments to get a feel for how different settings impact the performance of policy gradient methods, and report on your results.

1. Run the PG algorithm in the discrete `CartPole-v0` environment from the command line as follows:

```
python train_pg.py CartPole-v0 -n 100 -b 1000 -e 5 -dna --exp_name
sb_no_rtg_dna
python train_pg.py CartPole-v0 -n 100 -b 1000 -e 5 -rtg -dna --exp_name
sb_rtg_dna
python train_pg.py CartPole-v0 -n 100 -b 1000 -e 5 -rtg --exp_name
sb_rtg_na
python train_pg.py CartPole-v0 -n 100 -b 5000 -e 5 -dna --exp_name
lb_no_rtg_dna
python train_pg.py CartPole-v0 -n 100 -b 5000 -e 5 -rtg -dna --exp_name
lb_rtg_dna
```



```
python train_pg.py CartPole-v0 -n 100 -b 5000 -e 5 -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).
- `-e` : Number of experiments to run with the same configuration. Each experiment will start with a different randomly initialized policy, and have a different stream of random numbers.
- `-dna` : Flag: if present, sets `normalize_advantages` to `False`. Otherwise, by default, `normalize_advantages=True`.
- `-rtg` : Flag: if present, sets `reward_to_go=True`. Otherwise, by default, `reward_to_go=False`.
- `--exp_name` : Name for experiment, which goes into the name for the data 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.

### Deliverables for report:

- Graph the results of your experiments **using the `plot.py` file we provide**. 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 gradient estimator has better performance without advantage-centering—the trajectory-centric one, or the one using **reward-to-go**?
  - Did advantage centering help? 
  - Describe what you expected from the math—do the empirical results match the theory?
  - Did the batch size make an impact? 

- Provide the exact command line configurations you used to run your experiments. (To verify batch size, learning rate, architecture, and so on.)

#### **What to Expect:**

- CartPole converges to a maximum score of 200.
2. Run experiments in the `InvertedPendulum-v1` continuous control environment and find hyperparameter settings (network architecture, learning rate, batch size, reward-to-go, advantage centering, etc.) that allow you to solve the task. Try to find the smallest possible batch size that succeeds.

Note: Which gradient estimator should you use, based on your experiments in the previous section?

#### **Deliverables:**

- 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.) (Also, your policy performance may fluctuate around 1000—this is fine.)
- Provide the exact command line configurations you used to run your experiments. If you made any extreme choices (unusually high learning rate, weirdly deep network), justify them briefly.

## **5 Implement Neural Network Baselines**

In this section, you will implement a state-dependent neural network baseline function. The `train_PG.py` file has instructions for what parts of the code you need to modify.

After you have completed the code, run the following experiments. Make sure to run over multiple random seeds:

1. For the inverted pendulum task, compare the learning curve with both the neural network baseline function and advantage normalization to the learning curve without the neural network baseline but with advantage normalization.

## **6 HalfCheetah**

For this section, you will use your policy gradient implementation to solve a much more challenging task: `HalfCheetah-v1`. From the command line, run:

```
python train_pg.py HalfCheetah-v1 -ep 150 --discount 0.9 (other settings)
```

where (other settings) is replaced with any settings of your choosing. The `-ep 150` setting makes the episode length 150, which is shorter than the default of 1000 for HalfCheetah and speeds up your training significantly.

1. Find any settings which result in the agent attaining an average score of 150 or more at the end of 100 iterations, and provide a learning curve.

This may take a while ( $\sim 20$ -30 minutes) to train.

## 7 Bonus

Choose any (or all) of the following:

- A serious bottleneck in the learning, for more complex environments, is the sample collection time. In `train_PG.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- $\lambda$  for advantage estimation.<sup>1</sup> 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.

## 8 Submission

Your report should be a document containing 1) all graphs requested in sections 4, 5, and 6, and 2) the answers to all short ‘explanation’ questions in sections 4, and 3) all command line expressions you used to run your experiments.

You should also turn in your modified `train_pg.py` file. If your code includes additional files, provide a zip file including your `train_pg.py` and all other files needed to run your code, along with any special instructions needed to exactly duplicate your results.

Turn this in by September 20th 11:59pm by emailing your report and code to `berkeleydeeprlcourse@gmail.com`, with subject line “Deep RL Assignment 2”.

---

<sup>1</sup><https://arxiv.org/abs/1506.02438>