CS294-112 Deep Reinforcement Learning HW2: Policy Gradients
Pytorch Version

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 experiment with variance reduction tricks, including implementing reward-to-go and neural network baselines.

2 Review

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

$$\pi_\theta(\tau) = p(s_1, a_1, ..., s_T, a_T) = p(s_1)\pi_\theta(a_1|s_1) \prod_{t=2}^{T} p(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_t)$$

and

$$r(\tau) = r(s_1, a_1, ..., s_T, a_T) = \sum_{t=1}^{T} r(s_t, a_t).$$

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

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

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

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

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i) \tag{4}$$

$$= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \left(\sum_{t=1}^{T} r(s_{it}, a_{it}) \right). \tag{5}$$

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 $r(s_t, a_t)$.

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, yielding following the modified objective, where the sum of rewards here is a sample estimate of the Q function, known as the "reward-to-go:"

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

Multiplying a discount factor γ to the rewards can be interpreted as encouraging the agent to focus on rewards closer in the future, which can also be thought of as a means for reducing variance (because there is more variance possible futures further into the future). We saw in lecture that the discount factor can be incorporated in two ways.

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

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

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

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

.

We have seen in lecture that subtracting a baseline that is a constant with respect to τ from the sum of rewards

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[r(\tau) - b \right] \tag{9}$$

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. The value function is trained to approximate the sum of future rewards starting from a particular state:

$$V_\phi^\pi(s_t) \approx \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}\left[r(s_{t'}, a_{t'})|s_t\right], \tag{10}$$

so the approximate policy gradient now looks like this:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{it'}, a_{it'})\right) - V_\phi^\pi\left(s_{it}\right)\right). \tag{11}$$

Problem 1. State-dependent baseline: In lecture we saw that the policy gradient is unbiased if the baseline is a constant with respect to τ (Equation 9). The purpose of this problem is to help convince ourselves that subtracting a state-dependent baseline from the return keeps the policy gradient unbiased. For clarity we will use $p_\theta(\tau)$ instead of $\pi_\theta(\tau)$, although they mean the same thing. Using the law of iterated expectations we will show that the policy gradient is still unbiased if the baseline b is function of a state at a particular timestep of τ (Equation 11). Recall from equation 3 that the policy gradient can be expressed as:

$$\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\nabla_\theta \log p_\theta(\tau) r(\tau)\right].$$

By breaking up $p_\theta(\tau)$ into dynamics and policy terms, we can discard the dynamics terms, which are not functions of θ:

$$\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=1}^{T} r(s_{t'}, a_{t'})\right)\right].$$

When we subtract a state dependent baseline $b(s_t)$ (recall equation 11) we get

$$\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\left(\sum_{t'=1}^{T} r(s_{t'}, a_{t'})\right) - b(s_t)\right)\right].$$

An alternative approach is to look at the entire trajectory and consider a particular timestep $t^* \in [1, T-1]$ (the timestep T case would be very similar to part (a)). Our goal for this problem is to show that

$$\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)\right] = 0.$$

By linearity of expectation we can consider each term in this sum independently, so we can equivalently show that

$$\sum_{t=1}^{T} \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \left(b(s_t)\right)\right] = 0. \tag{12}$$

(a) Using the chain rule, we can express $p_\theta(\tau)$ as a product of the state-action marginal (s_t, a_t) and the probability of the rest of the trajectory conditioned on (s_t, a_t) (which we denote as $(\tau/s_t, a_t|s_t, a_t)$):

$$p_\theta(\tau) = p_\theta(s_t, a_t)p_\theta(\tau/s_t, a_t|s_t, a_t)$$

Please show equation 12 by using the law of iterated expectations, breaking $\mathbb{E}_{\tau\sim p_\theta(\tau)}$ by decoupling the state-action marginal from the rest of the trajectory.

(b) Alternatively, we can consider the structure of the MDP and express $p_\theta(\tau)$ as a product of the trajectory distribution up to s_t (which we denote as $(s_{1:t}, a_{1:t-1})$) and the trajectory distribution after s_t conditioned on the first part (which we denote as $(s_{t+1:T}, a_{t:T}|s_{1:t}, a_{1:t-1})$):

$$p_\theta(\tau) = p_\theta(s_{1:t}, a_{1:t-1})p_\theta(s_{t+1:T}, a_{t:T}|s_{1:t}, a_{1:t-1})$$

(a) Explain why, for the inner expectation, conditioning on $(s_1, a_1, ..., a_{t^*-1}, s_{t^*})$ is equivalent to conditioning only on s_{t^*}.

(b) Please show equation 12 by using the law of iterated expectations, breaking $\mathbb{E}_{\tau\sim p_\theta(\tau)}$ by decoupling trajectory up to s_t from the trajectory after s_t.

Since the policy gradient with respect to θ can be decoupled as a summation of terms over timesteps $t \in [1, T]$, because we have shown that the policy gradient is unbiased for each of these terms, the entire policy gradient is also unbiased with respect to a vector of state-dependent baselines over the timesteps: $[b(s_1), b(s_2), ...b(s_T)]$.

3 Code Setup

3.1 Files

The starter code is available here. The only file you need to modify in this homework is `train_pg_f18.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. For the Lunar Lander task, use the provided `lunar_lander.py` file instead of `gym/envs/box2d/lunar_lander.py`. After you fill in the appropriate methods, you should be able to just run `python train_pg_f18.py` with some command line options to perform the experiments. To visualize the results, you can run `python plot.py path/to/logdir`.

3.2 Overview

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.

The `Agent` class contains methods that define the neural networks, sample trajectories, estimate returns, and update the parameters of the policy.

At a high level, the dataflow of the code is structured like this:

1. *Define neural network components* from `torch.nn` in Pytorch.
2. *Build the forward pass function* for your neural network model by using the components you just defined.

Then we will repeat Steps 3 through 5 for N iterations:

3. *Sample trajectories* by executing the functions that samples an action given an observation from the environment. Collect the states, actions, and rewards as numpy variables.
4. *Estimate returns* in numpy (estimated Q values, baseline predictions, advantages).
5. *Update parameters* by executing the functions that updates the parameters given what you computed in Step 4.

4 Building Neural Networks

Problem 2. Neural networks: We will now begin to implement a neural network that parametrizes π_θ.

(a) Implement the utility function, `build_mlp`, which will build a feedforward neural network with fully connected units (Hint: use `torch.nn.Linear`). 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.

(b) Next, implement the functions used for forward pass. At this point, you only need to implement the parts with the "Problem 2" header.

(i) Define the model components in `PolicyNet.define_model_components`. You should define the parameters of your model here, which will be tracked by `torch.autograd` later. They can be any instance of `torch.nn.Module` or `torch.nn.Parameter`.

(ii) Define the method `PolicyNet.forward`: This defines forward pass for our policy network. It outputs the parameters of a distribution $\pi_\theta(a|s)$. In this homework, when the distribution is over discrete actions these parameters will be the logits of a categorical distribution, and when the distribution is over continuous actions these parameters will be the mean and the log standard deviation of a multivariate Gaussian distribution.

(iii) Define the method `Agent.sample_action`: This receives an observation and produces an action that sampled from $\pi_\theta(a|s)$. This method will be called in `Agent.sample_trajectory`.

(iv) Define the method `Agent.get_log_prob`: Given an action that the agent took in the environment, this computes the log probability of that action under $\pi_\theta(a|s)$. This will be used in the loss function.

5 Implement Policy Gradient

5.1 Implementing the policy gradient loop

Problem 3. Policy Gradient: Recall from lecture that an RL algorithm can viewed as consisting of three parts, which are reflected in the training loop of `train_PG`:

1. `Agent.sample_trajectories`: Generate samples (e.g. run the policy to collect trajectories consisting of state transitions (s, a, s', r))
2. `Agent.estimate_return`: Estimate the return (e.g. sum together discounted rewards from the trajectories, or learn a model that predicts expected total future discounted reward)
3. `Agent.update_parameters`: Improve the policy (e.g. update the parameters of the policy with policy gradient)

In our implementation, for clarity we will update the parameters of the value function baseline also in the third step (`Agent.update_parameters`), rather than in the second step (as was described in lecture). You only need to implement the parts with the "Problem 3" header.

(a) **Sample trajectories:** In `Agent.sample_trajectories`, use the method `Agent.sample_action` which you just defined in "Problem 2" to sample an action given an observation from the environment.

(b) **Estimate return:** We will now implement $r(\tau)$ from Equation 1. Please implement the method `Agent.sum_of_rewards`, which will return a sample estimate of the discounted return, for both the full-trajectory (Equation 7) case, where

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

and for the "reward-to-go" case (Equation 8) where

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

In `Agent.estimate_return`, normalize the advantages to have a mean of zero and a standard deviation of one. This is a trick for reducing variance.

(c) **Update parameters:** In `Agent.update_parameters` implement a loss function (which can use the result from `Agent.get_log_prob`) to whose gradient is

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

Then, set the optimizer (we use `torch.optim.Adam` in this case) in the right way and perform gradient decent to update the parameters of the policy.

5.2 Experiments

After you have implemented the code, we will run experiments to get a feel for how different settings impact the performance of policy gradient methods.

Problem 4. CartPole: Run the PG algorithm in the discrete `CartPole-v0` environment from the command line as follows:

```
python train_pg_f18.py CartPole-v0 -n 100 -b 1000 -e 3 -dna --exp_name
    sb_no_rtg_dna
python train_pg_f18.py CartPole-v0 -n 100 -b 1000 -e 3 -rtg -dna --exp_name
    sb_rtg_dna
python train_pg_f18.py CartPole-v0 -n 100 -b 1000 -e 3 -rtg --exp_name
    sb_rtg_na
python train_pg_f18.py CartPole-v0 -n 100 -b 5000 -e 3 -dna --exp_name
    lb_no_rtg_dna
python train_pg_f18.py CartPole-v0 -n 100 -b 5000 -e 3 -rtg -dna --exp_name
    lb_rtg_dna
python train_pg_f18.py CartPole-v0 -n 100 -b 5000 -e 3 -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, `reward_to_go=False` by default.

- `--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?
  - 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:

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

Problem 5. InvertedPendulum: Run experiments in `InvertedPendulum-v2` continuous control environment as follows:

```
python train_pg_f18.py InvertedPendulum-v2 -ep 1000 --discount 0.9 -n 100 -e 3
    -l 2 -s 64 -b <b*> -lr <r*> -rtg --exp_name hc_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 Implement Neural Network Baselines

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

Problem 6. Neural network baseline: We will now implement a value function as a state-dependent neural network baseline. The sections in the code are marked by "Problem 6."

(a) In `Agent.__init__` implement V_ϕ^π, a neural network that predicts the expected return conditioned on a state.

(b) In `Agent.compute_advantage`, use the neural network to predict the expected state-conditioned return (call `self.value_net`), normalize 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. This implements

$$\left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{it'}, a_{it'})\right) - V_\phi^\pi\left(s_{it}\right)$$

.

(c) In `Agent.update_parameters`, implement the loss function to train this network. "Rescale" the target values for the neural network baseline to have a mean of zero and a standard deviation of one.

7 More Complex Tasks

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

Problem 7: LunarLander For this problem, you will use your policy gradient implementation to solve `LunarLanderContinuous-v2`. Use an episode length of 1000. The purpose of this problem is to help you debug your baseline implementation. Run the following command:

```
python train_pg_f18.py LunarLanderContinuous-v2 -ep 1000 --discount 0.99 -n
    100 -e 3 -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 8: 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 train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.9 -n 100 -e 3 -l 2
    -s 32 -b <b> -lr <r> -rtg --nn_baseline --exp_name hc_b<b>_r<r>
```

Deliverables:

- How did the batch size and learning rate affect the 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 train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100 -e 3
      -l 2 -s 32 -b <b*> -lr <r*> --exp_name hc_b<b*>_r<r*>
  python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100 -e 3
      -l 2 -s 32 -b <b*> -lr <r*> -rtg --exp_name hc_b<b*>_r<r*>
  python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100 -e 3
      -l 2 -s 32 -b <b*> -lr <r*> --nn_baseline --exp_name hc_b<b*>_r<r*>
  python train_pg_f18.py HalfCheetah-v2 -ep 150 --discount 0.95 -n 100 -e 3
      -l 2 -s 32 -b <b*> -lr <r*> -rtg --nn_baseline --exp_name hc_b<b*>_r<
     r*>
  ```

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

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 train_pg_f18.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

9 Submission

Your report should be a document containing

(a) Your mathematical response (written in LaTeX) for Problem 1.

(b) All graphs requested in Problems 4, 5, 7, and 8.

(c) Answers to short explanation questions in section 5 and 7.

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

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

Please also submit your modified `train_pg_f18.py` file. If your code includes additional files, provide a zip file including your `train_pg_f18.py` and all other files needed to run your code. Please include a `README.md` with instructions needed to exactly duplicate your results (including command-line expressions).