0 if tex.enableprimitives thenΩtex.enableprimitives(Ω'pdf@',Ω'primitive', 'ifprimitive', 'pdfdraftmode','draftmode'Ω)Ωtex.enableprimitives(", 'luaescapestring')ΩendΩ

CS 234: Assignment #3

Due date: February 24, 2020 at 11:59 PM (23:59) PST

These questions require thought, but do not require long answers. Please be as concise as possible.

We encourage students to discuss in groups for assignments. **However, each student must finish the problem set and programming assignment individually, and must turn in her/his assignment.** We ask that you abide by the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work is done by yourself. If you have discussed the problems with others, please include a statement saying who you discussed problems with. Failure to follow these instructions will be reported to the Office of Community Standards. We reserve the right to run a fraud-detection software on your code.

Please review any additional instructions posted on the assignment page at http://cs234.stanford.edu/assignment3. When you are ready to submit, please follow the instructions on the course website.

1 Policy Gradient Methods (50 pts coding + 15 pts writeup)

The goal of this problem 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. The framework for the policy gradient algorithm is setup in main.py, and everything that you need to implement is in the files network-utils.py, policy-network.py and baseline-network.py. The file has detailed instructions for each implementation task, but an overview of key steps in the algorithm is provided here.

1.1 REINFORCE

Recall the policy gradient theorem,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]$$

REINFORCE is a Monte Carlo policy gradient algorithm, so we will be using the sampled returns G_t as unbiased estimates of $Q^{\pi_\theta}(s,a)$. Then the gradient update can be expressed as maximizing the following objective function:

$$J(\theta) = \frac{1}{|D|}\sum_{\tau \in D}\sum_{t=1}^{T} \log(\pi_\theta(a_t|s_t))G_t$$

where D is the set of all trajectories collected by policy π_θ, and $\tau = (s_0, a_0, r_0, s_1 ...)$ is a trajectory.

1.2 Baseline

One difficulty of training with the REINFORCE algorithm is that the Monte Carlo sampled return(s) G_t can have high variance. To reduce variance, we subtract a baseline $b_\phi(s)$ from the estimated returns when computing the policy gradient. A good baseline is the state value function, $V^{\pi_\theta}(s)$, which requires a training update to ϕ to minimize the following mean-squared error loss:

$$L_{MSE} = \frac{1}{|D|}\sum_{\tau \in D}\sum_{t=1}^{T}(b_\phi(s_t) - G_t)^2$$

1.3 Advantage Normalization

After subtracting the baseline, we get the following new objective function:

$$J(\theta) = \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=1}^{T} \log(\pi_\theta(a_t|s_t)) \hat{A}_t$$

where

$$\hat{A}_t = G_t - b_\phi(s_t)$$

A second variance reduction technique is to normalize the computed advantages, $\hat{A}_t$, so that they have mean 0 and standard deviation 1. From a theoretical perspective, we can consider centering the advantages to be simply adjusting the advantages by a constant baseline, which does not change the policy gradient. Likewise, rescaling the advantages effectively changes the learning rate by a factor of $1/\sigma$, where σ is the standard deviation of the empirical advantages.

1.4 Coding Questions (50 pts)

The functions that you need to implement in **network-utils.py**, **policy-network.py** and **baseline-network.py** are enumerated here. Detailed instructions for each function can be found in the comments in each of these files.

- build_mlp in network-utils.py
- add_placeholders_op in policy-network.py
- build_policy_network_op in policy-network.py
- add_loss_op in policy-network.py
- add_optimizer_op in policy-network.py
- get_returns in policy-network.py
- normalize_advantage in policy-network.py
- add_baseline_op in baseline-network.py
- calculate_advantage in baseline-network.py
- update_baseline in baseline-network.py

1.5 Writeup Questions (15 pts)

(a)(i) (2 pts) (CartPole-v0) Test your implementation on the CartPole-v0 environment by running the following command. We are using random seed as 15

With the given configuration file, the average reward should reach 200 within 100 iterations. *NOTE: training may repeatedly converge to 200 and diverge. Your plot does not have to reach 200 and stay there. We only require that you achieve a perfect score of 200 sometime during training.*

Include in your writeup the tensorboard plot for the average reward. Start tensorboard with:

and then navigate to the link it gives you. Click on the "SCALARS" tab to view the average reward graph.

Now, test your implementation on the CartPole-v0 environment without baseline by running

Include the tensorboard plot for the average reward. Do you notice any difference? Explain.

(a)(ii) (2 pts) (CartPole-v0) Test your implementation on the CartPole-v0 environment by running the following command. We are using random seed as 12345456

With the given configuration file, the average reward should reach 200 within 100 iterations. *NOTE: training may repeatedly converge to 200 and diverge. Your plot does not have to reach 200 and stay there. We only require that you achieve a perfect score of 200 sometime during training.*

Include in your writeup the tensorboard plot for the average reward. Start tensorboard with:

and then navigate to the link it gives you. Click on the "SCALARS" tab to view the average reward graph.

Now, test your implementation on the CartPole-v0 environment without baseline by running

Include the tensorboard plot for the average reward. Do you notice any difference? Explain.

(b)(i) (2 pts) (InvertedPendulum-v1) Test your implementation on the InvertedPendulum-v1 environment by running

With the given configuration file, the average reward should reach 1000 within 100 iterations. *NOTE: Again, we only require that you reach 1000 sometime during training.* Include the tensorboard plot for the average reward in your writeup.

Now, test your implementation on the InvertedPendulum-v1 environment without baseline by running for seed 15

Include the tensorboard plot for the average reward. Do you notice any difference? Explain.

Answer:

With baseline, the result is more stable and continously increse reward. This is because baseline is reducing variance for our estimator.

(b)(ii) (2 pts) (InvertedPendulum-v1) Test your implementation on the InvertedPendulum-v1 environment by running

With the given configuration file, the average reward should reach 1000 within 100 iterations. *NOTE: Again, we only require that you reach 1000 sometime during training.* Include the tensorboard plot for the average reward in your writeup.

Now, test your implementation on the InvertedPendulum-v1 environment without baseline by running for seed 8

Include the tensorboard plot for the average reward. Do you notice any difference? Explain.

(c)(i) (2 pts) (HalfCheetah-v1) Test your implementation on the HalfCheetah-v1 environment with $\gamma = 0.9$ by running the following command for seed 123

With the given configuration file, the average reward should reach 200 within 100 iterations. *NOTE: Again, we only require that you reach 200 sometime during training. There is some variance in training. You can run multiple times and report the best results.* Include the tensorboard plot for the average reward in your writeup.

Now, test your implementation on the HalfCheetah-v1 environment without baseline by running for seed 123

Include the tensorboard plot for the average reward. Do you notice any difference? Explain.

(c)(ii) (2 pts) (HalfCheetah-v1) Test your implementation on the HalfCheetah-v1 environment with $\gamma = 0.9$ by running the following command for seed 15

With the given configuration file, the average reward should reach 200 within 100 iterations. *NOTE: Again, we only require that you reach 200 sometime during training. There is some variance in training. You can run multiple times and report the best results.* Include the tensorboard plot for the average reward in your writeup.

Now, test your implementation on the HalfCheetah-v1 environment without baseline by running for seed 15

Include the tensorboard plot for the average reward. Do you notice any difference? Explain.

(c)(iii) (3 pts) How do the results differ across seeds? Describe briefly comparing the performance you get for different seeds for the 3 environments. Run the following commands to get the average performance across 3 runs (2 of the previous runs + 1 another run below).

Please comment on how the averaged performance for HalfCheetah environment baseline vs. no baseline case.

1.6 Testing

We have provided some basic tests in the root directory of the starter code to test your implementation. You can run the following command to run the tests and then check if you have the right implementation for individual functions.

2 Best Arm Identification in Multi-armed Bandit (35pts)

In this problem we focus on the bandit setting with rewards bounded in $[0, 1]$. A bandit problem can be considered as a finite-horizon MDP with just one state ($|\mathcal{S}| = 1$) and horizon 1: each episode consists of taking a single action and observing a reward. In the bandit setting – unlike in standard RL – there are no delayed rewards, and the action taken does affect the distribution of future states.

Actions are also referred to as "arms" in a bandit setting[1]. Here we consider a multi-armed bandit, meaning that $1 < |\mathcal{A}| < \infty$. Since there is only one state, a policy is simply a distribution over actions. There are exactly $|\mathcal{A}|$ different deterministic policies. Your goal is to design a simple algorithm to identify a near-optimal arm with high probability.

We recall Hoeffding's inequality: if $X_1, \ldots, X_n$ are i.i.d. random variables satisfying $0 \le X_i \le 1$ with probability 1 for all i, $\overline{X} = \mathbb{E}[X_1] = \cdots = \mathbb{E}[X_n]$ is the expected value of the random variables, and $\widehat{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the sample mean, then for any $\delta > 0$ we have

$$\Pr\left(|\widehat{X} - \overline{X}| > \sqrt{\frac{\log(2/\delta)}{2n}}\right) < \delta. \tag{1}$$

Assuming that the rewards are bounded in $[0, 1]$, we propose this simple strategy: pull each arm n_e times, and return the action with the highest average payout $\widehat{r}_a$. The purpose of this exercise is to study the number of samples required to output an arm that is at least ϵ-optimal with high probability. Intuitively, as n_e increases the empirical average of the payout $\widehat{r}_a$ converges to its expected value $\overline{r}_a$ for every action a, and so choosing the arm with the highest empirical payout $\widehat{r}_a$ corresponds to approximately choosing the arm with the highest expected payout $\overline{r}_a$.

(a) (15 pts) We start by bounding the probability of the "bad event" in which the empirical mean of some arm differs significantly from its expected return. Starting from Hoeffding's inequality with n_e samples allocated to every action, show that:

$$\Pr\left(\exists a \in \mathcal{A} \quad \text{s.t.} \quad |\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right) < |\mathcal{A}|\delta. \tag{2}$$

Note that, depending on your derivation, you may come up with a tighter upper bound than $|\mathcal{A}|\delta$. This is also acceptable (as long as you argue that your bound is tighter), but showing the inequality above is sufficient.

(b) (20 pts) After pulling each arm (action) n_e times our algorithm returns the arm with the highest empirical mean:

$$a^\dagger = \arg\max_a \widehat{r}_a \tag{3}$$

Notice that $a^\dagger$ is a random variable. Let $a^\star = \arg\max_a \overline{r}_a$ be the true optimal arm. Suppose that we want our algorithm to return at least an ϵ-optimal arm with probability at least $1 - \delta'$, as follows:

$$\Pr\left(\overline{r}_{a^\dagger} \ge \overline{r}_{a^\star} - \epsilon\right) \ge 1 - \delta'. \tag{4}$$

How many samples are needed to ensure this? Express your result as a function of the number of actions, the required precision ϵ and the failure probability δ'.

[1] The name "bandit" comes from slot machines, which are sometimes called "one-armed bandits". Here we imagine a slot machine which has several arms one might pull, each with a different (random) payout.