# CS 234: Assignment #3
# Winter 2019

**Due date: 2/27 11:59 PM PST**

We encourage students to discuss in groups for assignments. We ask that you abide by the university Honor Code and that of the Computer Science department. 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 refer to the course website, Academic Collaboration and Misconduct section for details about collaboration policy.

Please review any additional instructions posted on the assignment page. When you are ready to submit, 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 vanilla policy gradient algorithm is setup in the starter code `pg.py`, and everything that you need to implement is in this file. The file has detailed instructions for each implementation task, but an overview of key steps in the algorithm is provided here. For this assignment you need to have MuJoCo installed, please follow the installation guide.

### REINFORCE

Recall the vanilla 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 $\pi_\theta$, and $\tau = (s_0, a_0, r_0, s_1..., s_T)$ is a trajectory.

### Baseline

One difficulty of training with the REINFORCE algorithm is that the monte-carlo estimated return $G_t$ 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 parametrized by $\phi$, $b_\phi(s) = V^{\pi_\theta}(s)$, which requires a training update to $\phi$ to minimize the following mean-squared error loss:

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

## 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 $\sigma$ is the standard deviation of the empirical advantages.

### 1.1 Coding Questions (50 pts)

The functions that you need to implement in `pg.py` are enumerated here. Detailed instructions for each function can be found in the comments in `pg.py`. We strongly encourage you to look at `pg.py` and understand the code structure first.

- `build_mlp`
- `add_placeholders_op`
- `build_policy_network_op`
- `add_loss_op`
- `add_optimizer_op`
- `add_baseline_op`
- `get_returns`
- `calculate_advantage`
- `update_baseline`

### 1.2 Writeup Questions (15 pts)

(a) (4 pts) (CartPole-v0) Test your implementation on the CartPole-v0 environment by running

```
python pg.py --env_name cartpole --baseline
```

With the given configuration file `config.py`, 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:

```
tensorboard --logdir=results
```

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

```
python pg.py --env_name cartpole --no-baseline
```

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

(b) (4 pts) (InvertedPendulum-v1) Test your implementation on the InvertedPendulum-v1 environment by running

```
python pg.py --env_name pendulum --baseline
```

With the given configuration file `config.py`, 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

```
python pg.py --env_name pendulum --no-baseline
```

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

(c) (7 pts) (HalfCheetah-v1) Test your implementation on the HalfCheetah-v1 environment with $\gamma = 0.9$ by running

```
python pg.py --env_name cheetah --baseline
```

With the given configuration file `config.py`, the average reward should reach 200 within 100 iterations. *NOTE: There is some variance in training. You can run multiple times and report the best results or average. We have provided our results (average reward) averaged over 6 different random seed in figure 1* Include the tensorboard plot for the average reward in your writeup.

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

```
python pg.py --env_name cheetah --no-baseline
```

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

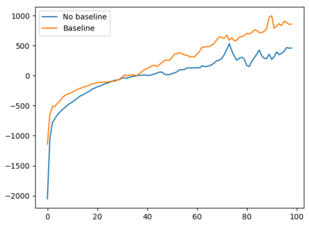

Figure 1: Half Cheetah, averaged over 6 runs

## 2 Best Arm Identification in Multiarmed Bandit (35pts)

In this problem we focus on the Bandit setting with rewards bounded in $[0,1]$. A Bandit problem instance is defined as an MDP with just one state and action set $\mathcal{A}$. Since there is only one state, a "policy" consists of the choice of a single action: there are exactly $A = |\mathcal{A}|$ different deterministic policies. Your goal is to design a simple algorithm to identify a near-optimal arm with high probability.
Imagine we have $n$ samples of a random variable $x$, $\{x_1, \ldots, x_n\}$. We recall Hoeffding's inequality below, where $\overline{x}$ is the expected value of a random variable $x$, $\widehat{x} = \frac{1}{n}\sum_{i=1}^n x_i$ is the sample mean (under the assumption that the random variables are in the interval [0,1]), $n$ is the number of samples and $\delta > 0$ is a scalar:

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

Assuming that the rewards are bounded in $[0,1]$, we propose this simple strategy: allocate an identical number of samples $n_1 = n_2 = ... = n_A = n_{des}$ to every action, compute the average reward (empirical payout) of each arm $\widehat{r}_{a_1}, \ldots, \widehat{r}_{a_A}$ and return the action with the highest empirical payout $\arg\max_a \widehat{r}_a$. The purpose of this exercise is to study the number of samples required to output an arm that is at least $\epsilon$-optimal with high probability. Intuitively, as $n_{des}$ increases the empirical 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 defining a *good event*. Under this *good event*, the empirical payout of each arm is not too far from its expected value. Starting from Hoeffding inequality with $n_{des}$ samples allocated to every action show that:

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

In other words, the *bad event* is that at least one arm has an empirical mean that differs significantly from its expected value and this has probability at most $A\delta$.

(b) (20 pts) After pulling each arm (action) $n_{des}$ times our algorithm returns the arm with the highest empirical payout:

$$a^\dagger = argmax_a \widehat{r}_a$$

Notice that $a^\dagger$ is a random variable. Define $a^\star$ as the optimal arm (that yields the highest average reward $a^\star = argmax_a \overline{r}_a$). Suppose that we want our algorithm to return at least an $\epsilon$ optimal arm with probability $1 - \delta'$, as follows:

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

How many samples are needed to ensure this? Express your result as a function of the number of actions $A$, the required precision $\epsilon$ and the failure probability $\delta'$.