

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,

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

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 $\tau = (s_0, a_0, r_0, s_1 \dots)$ 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

```
python main.py --env_name cartpole --baseline --r_seed 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:

```
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

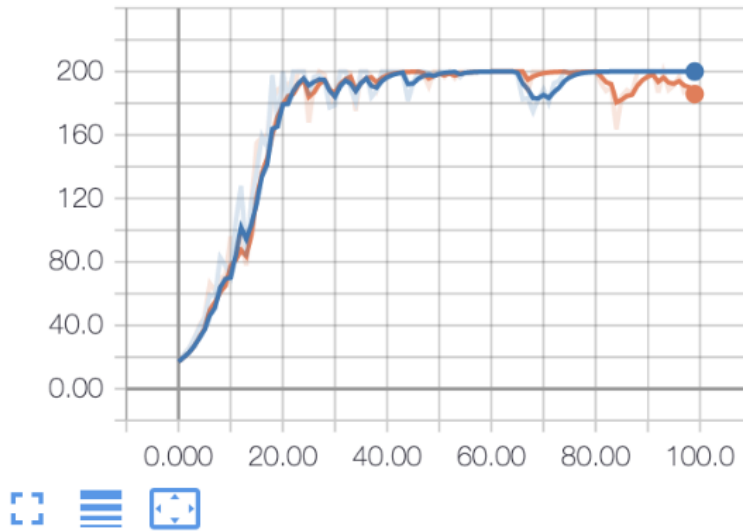
```
python main.py --env_name cartpole --no-baseline --r_seed 15
```

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

Solution Curves that reach the desired performance of 200 at least once are fine.

Avg_Reward

Avg_Reward



Eval_Reward

Eval_Reward

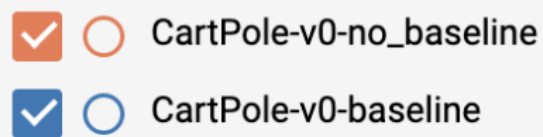
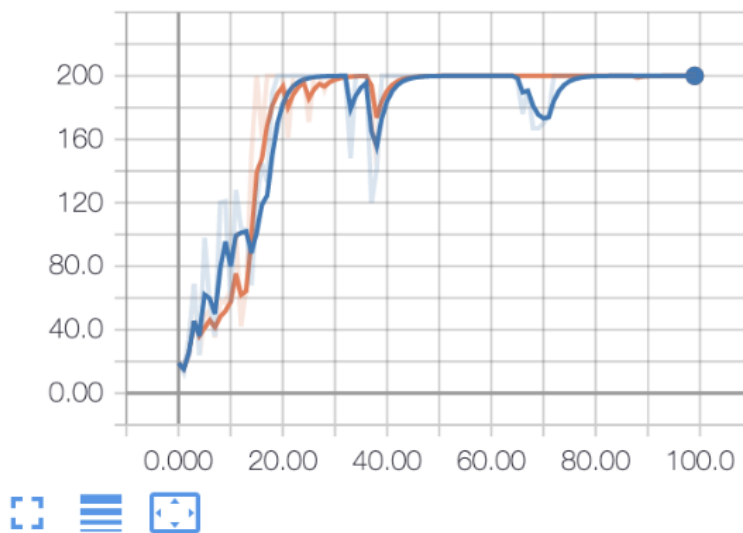


Figure 1: Seed 15 Cartpole. Orange is no baseline, Blue is baseline

- (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

```
python main.py --env_name cartpole --baseline --r_seed 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:

```
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 main.py --env_name cartpole --no-baseline --r_seed 12345456
```

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

Solution Curves that reach the desired performance of 200 at least once are fine.

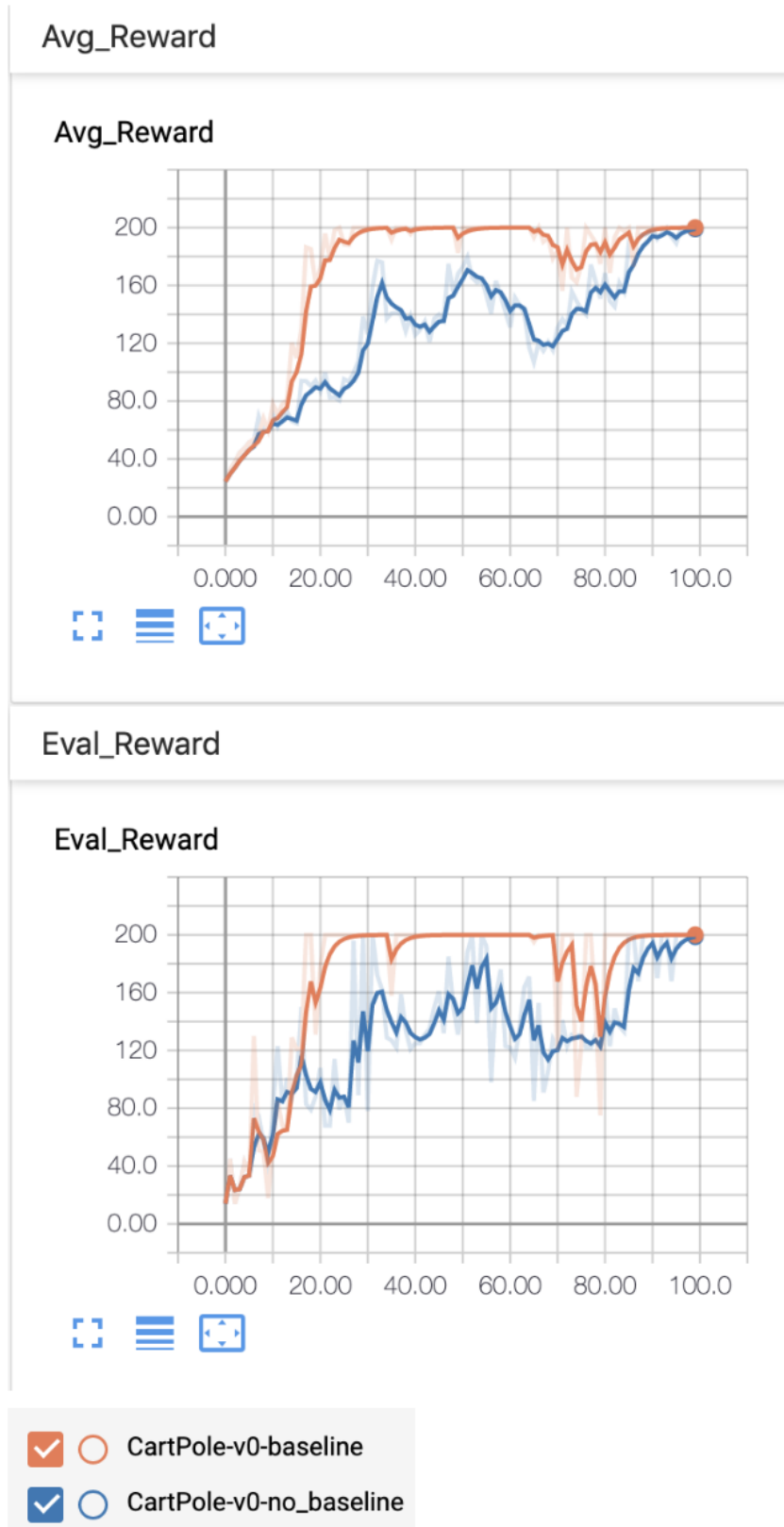


Figure 2: Seed 12345456 Cartpole. Orange is baseline, Blue is non baseline

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

```
python main.py --env_name pendulum --baseline --r_seed 15
```

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

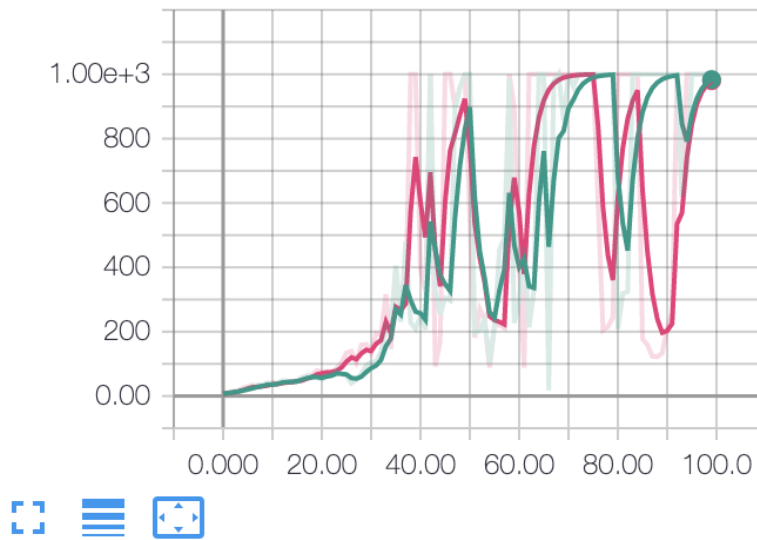
```
python main.py --env_name pendulum --no-baseline --r_seed 15
```

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

Solution Curves that reach the desired performance of 1000 at least once are fine.

Avg_Reward

Avg_Reward



Eval_Reward

Eval_Reward

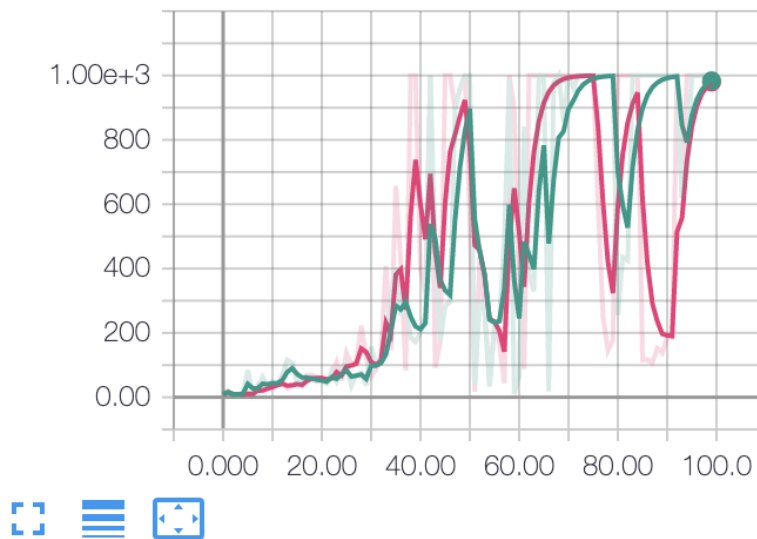


Figure 3: Seed 15 Pendulum. Red is no baseline, Green is baseline

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


```
python main.py --env_name pendulum --baseline --r_seed 8
```

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

```
python main.py --env_name pendulum --no-baseline --r_seed 8
```

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

Solution Curves that reach the desired performance of 1000 at least once are fine. Ideally, baseline should be more consistent and less variance here.

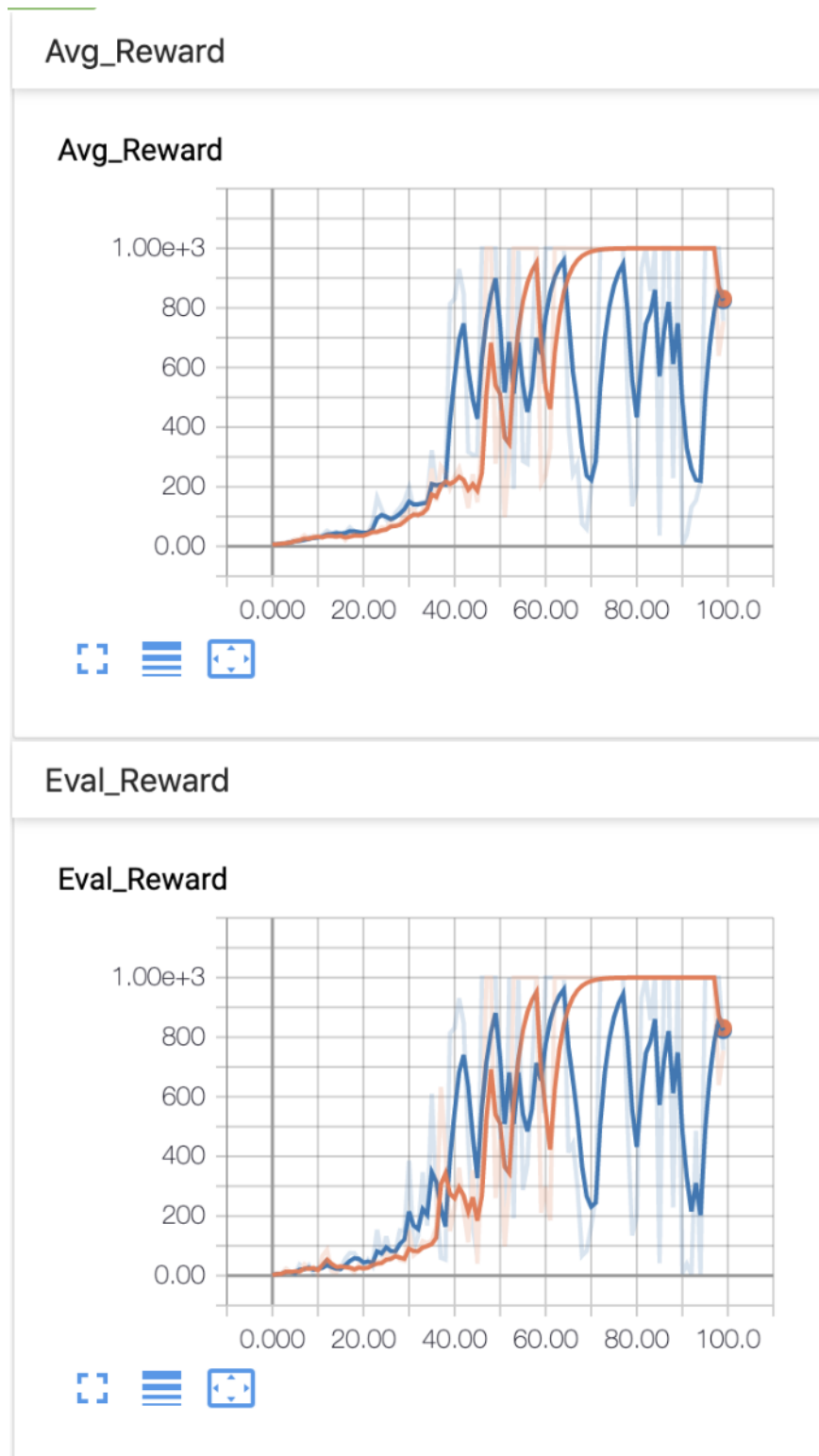


Figure 4: Seed 8 Pendulum. Orange is baseline, Blue is non baseline

- (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

```
python main.py --env_name cheetah --baseline --r_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.

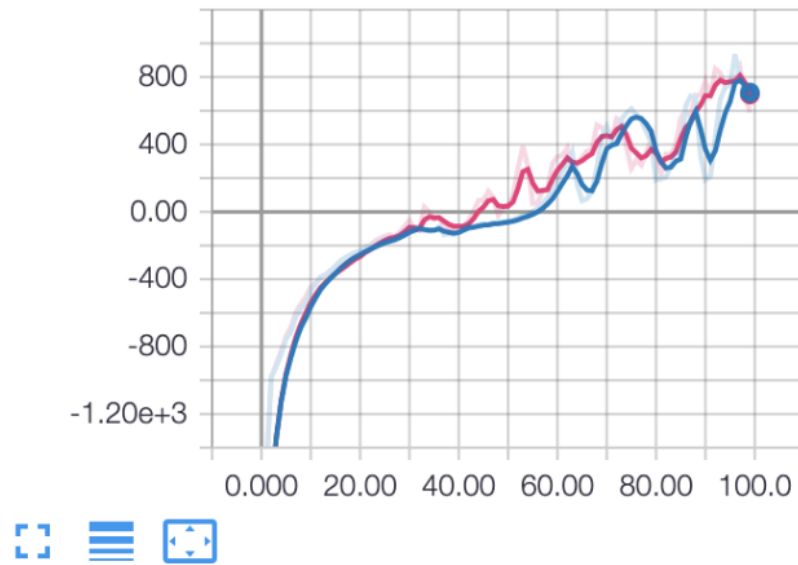
Now, test your implementation on the HalfCheetah-v1 environment without baseline by running for seed 123

```
python main.py --env_name cheetah --no-baseline --r_seed 123
```

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

Solution Any curve that reaches above desired performance of 200 with baseline is good. Should ideally observe higher performance with baseline.

Avg_Reward



Eval_Reward

Eval_Reward

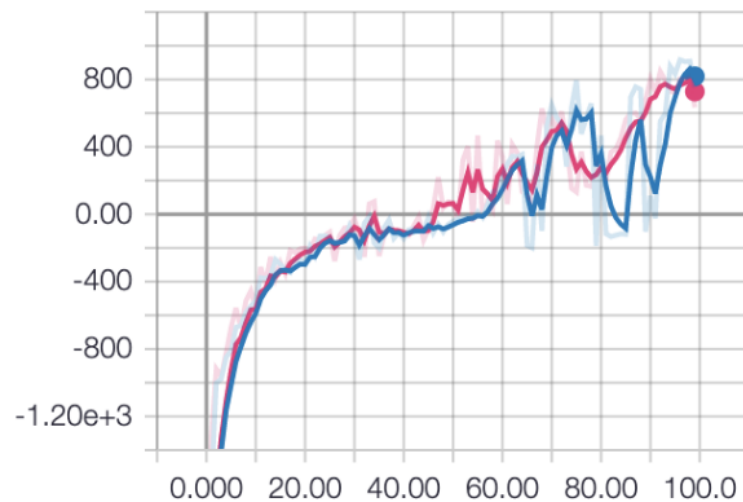


Figure 5: Seed 123 half cheetah. Blue is baseline, Red no baseline

- (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

```
python main.py --env_name cheetah --baseline --r_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.

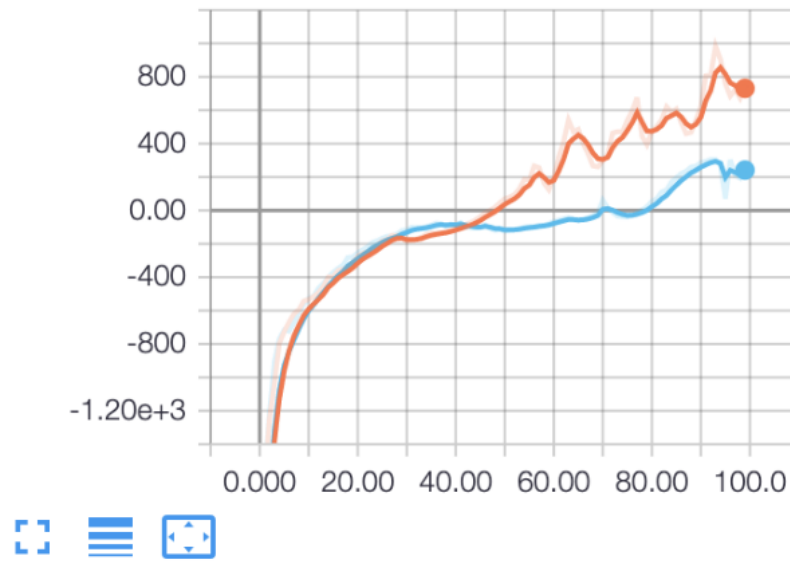
Now, test your implementation on the HalfCheetah-v1 environment without baseline by running for seed 15

```
python main.py --env_name cheetah --no-baseline --r_seed 15
```

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

Solution Any curve that reaches above desired performance of 200 with baseline is good. Should ideally observe higher performance with baseline.

Avg_Reward



Eval_Reward

Eval_Reward

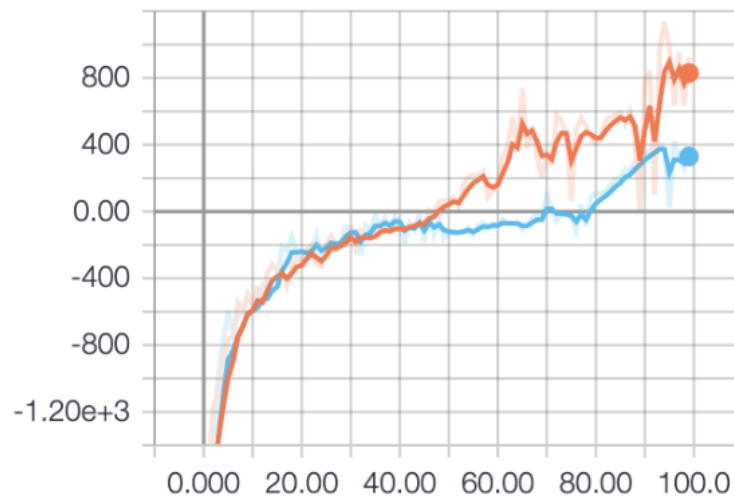


Figure 6: Seed 15 half cheetah. Orange is baseline, Blue no baseline

- (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).

```
python main.py --env_name cheetah --no-baseline --r_seed 12345456
```

```
python main.py --env_name cheetah --baseline --r_seed 12345456
```

```
python plot.py --env HalfCheetah-v1 -d results
```

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

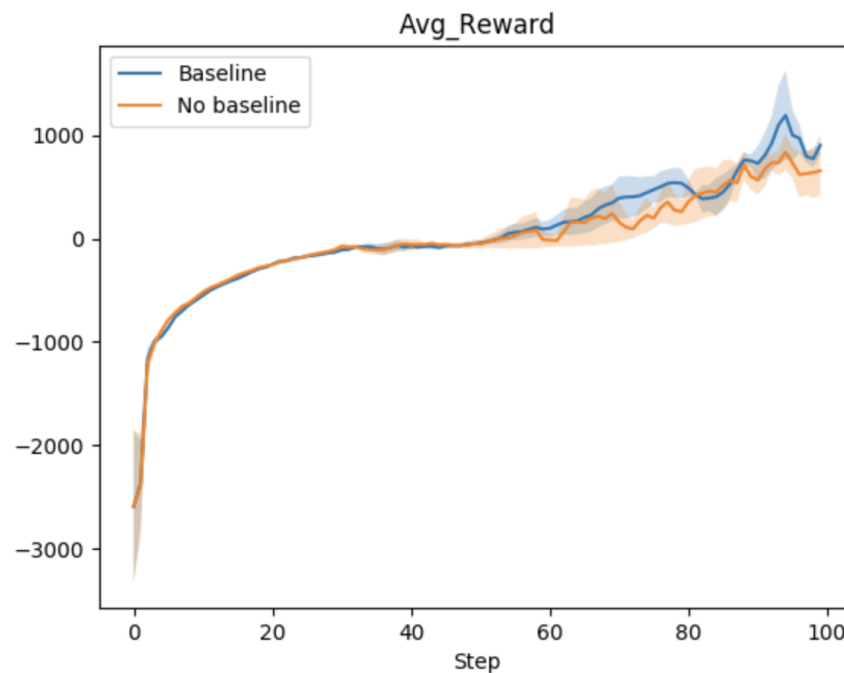


Figure 7: Avg Plot for Cheetah. Blue is baseline, Orange is no baseline

Solution Performance across seeds:

- With all the seeds, at least in Cheetah environment, we observe the agent able to learn a decent policy that gives the desired rewards.
- Each seed has slightly different performance since Policy Gradient methods converge to local minima.

Baseline vs no baseline:

- In some seeds, we observe baseline does significantly better than no baseline but in some of them the performance may be similar. The primary reason for this is this the large amount of stochasticity involved (coming from both the environment and the policy); even if one method is better on average, it may happen not to be better for a particular seed¹. Since baseline is doing some extra set of operations, even if the seed is same across baseline and no baseline, the performance will be different in those two cases.
- Typically, you will observe baseline has lesser variance, but this may not be observed in all cases.

Average Plot: Any plot where baseline exceeds desired performance (200) is good.

¹See <https://arxiv.org/abs/1709.06560>

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.

```
python run_basic_tests.py
```


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 setting². 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, \dots, X_n are i.i.d. random variables satisfying $0 \leq X_i \leq 1$ with probability 1 for all i , $\bar{X} = \mathbb{E}[X_1] = \dots = \mathbb{E}[X_n]$ is the expected value of the random variables, and $\hat{X} = \frac{1}{n} \sum_{i=1}^n X_i$ is the sample mean, then for any $\delta > 0$ we have

$$\Pr \left(|\hat{X} - \bar{X}| > \sqrt{\frac{\log(2/\delta)}{2n}} \right) < \delta. \quad (1)$$

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 \hat{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 \hat{r}_a converges to its expected value \bar{r}_a for every action a , and so choosing the arm with the highest empirical payout \hat{r}_a corresponds to approximately choosing the arm with the highest expected payout \bar{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:

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

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.

Solution By union bound:

$$\begin{aligned} \Pr \left(\exists a \in \mathcal{A} \text{ s.t. } |\hat{r}_a - \bar{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}} \right) &= \Pr \left(\bigcup_{a \in \mathcal{A}} \left\{ |\hat{r}_a - \bar{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}} \right\} \right) \\ &\leq \sum_{a \in \mathcal{A}} \underbrace{\Pr \left(|\hat{r}_a - \bar{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}} \right)}_{< \delta} \\ &< |\mathcal{A}|\delta \end{aligned}$$

as claimed.

²The 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.

Alternatively,

$$\begin{aligned}
 \Pr \left(\exists a \in \mathcal{A} \quad \text{s.t.} \quad |\hat{r}_a - \bar{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}} \right) &= 1 - \Pr \left(\forall a \in \mathcal{A}, \quad |\hat{r}_a - \bar{r}_a| \leq \sqrt{\frac{\log(2/\delta)}{2n_e}} \right) \\
 &= 1 - \prod_{a \in \mathcal{A}} \underbrace{\Pr \left(|\hat{r}_a - \bar{r}_a| \leq \sqrt{\frac{\log(2/\delta)}{2n_e}} \right)}_{\geq 1-\delta} \\
 &\leq 1 - (1 - \delta)^{|\mathcal{A}|}
 \end{aligned}$$

This is indeed tighter than (2); Bernoulli's inequality tells us that

$$(1 - \delta)^{|\mathcal{A}|} \geq 1 - |\mathcal{A}|\delta$$

as long as $\delta \leq 2$.

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

$$a^\dagger = \arg \max_a \hat{r}_a \quad (3)$$

Notice that a^\dagger is a random variable. Let $a^* = \arg \max_a \bar{r}_a$ 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:

$$\Pr \left(\bar{r}_{a^\dagger} \geq \bar{r}_{a^*} - \epsilon \right) \geq 1 - \delta'. \quad (4)$$

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' .

Solution Observe that if every action's empirical mean is within t of its true expected value, i.e.

$$\forall a \in \mathcal{A}, \quad |\hat{r}_a - \bar{r}_a| \leq t$$

then the choice $a^\dagger = \arg \max_a \hat{r}_a$ ensures

$$\bar{r}_{a^\dagger} \geq \hat{r}_{a^\dagger} - t \geq \hat{r}_{a^*} - t \geq \bar{r}_{a^*} - 2t$$

The first and last inequalities follow from the condition above, whereas the middle follows from the definition of a^\dagger (since $\hat{r}_{a^\dagger} \geq \hat{r}_a$ for every a). Taking $t = \frac{\epsilon}{2}$ suffices for ϵ -optimality. So, we want

$$\Pr \left(\forall a \in \mathcal{A}, \quad |\hat{r}_a - \bar{r}_a| \leq \frac{\epsilon}{2} \right) \geq 1 - \delta'$$

i.e.

$$\Pr \left(\exists a \in \mathcal{A} \quad \text{s.t.} \quad |\hat{r}_a - \bar{r}_a| > \frac{\epsilon}{2} \right) < \delta'$$

Applying the result from part (a) with $\delta = \delta'/|\mathcal{A}|$, we see it suffices to have

$$\sqrt{\frac{\log(2|\mathcal{A}|/\delta')}{2n_e}} \leq \frac{\epsilon}{2}$$

Solving for n_e gives

$$n_e \geq \frac{2 \log(2|\mathcal{A}|/\delta')}{\epsilon^2}$$

The total number of samples required is n_e times the number of arms:

$$\frac{2|\mathcal{A}| \log(2|\mathcal{A}|/\delta')}{\epsilon^2}$$