ECE/SIOC 228 Machine Learning for Physical Applications: Assignment 2 (Spring 2022)

Instructor: Yuanyuan Shi
Teaching Assistants:
Srinivas Rao Daru, sdaru@ucsd.edu
Tawaana Homavazir, thomavaz@ucsd.edu
Rohin Garg, rgarg@ucsd.edu
Rishabh Jangir, rjangir@ucsd.edu

*** Deadline is May 27th, 11:59PM**

Submission format:

- For the Code: please submit all the code files as a zip on **Gradescope**. Only submit the python files that you had to edit, you do not need to add the images or plots to the submission.
- For the written parts: explanations and graphs and figures showing the results and performance of your implementation you may either use Latex or a Jupyter Notebook with markdown + Latex. You need to attach the plots generated by `plot.py` in this document (latex pdf or notebook pdf), as well as give a very brief explanation about what the plot shows and how is it different from your expectations.
- In the end, comment on whether or not you observe improved performance using a baseline. Read the complete assignment for a better understanding of what is expected, and ask any doubts regarding the submission format or the code or any other issue you might face on piazza.
- This assignment should not take more than a week to complete, but it is recommended that you start early with the implementation ans ask your doubts early on.

PART II: Programming Assignment

1 Problem 1: Policy Gradient (100 Points)

Experiment with policy gradient and its variants for both continuous and discrete environments. The starter code is setup in `main.py`, and everything that you need to implement is in the files `network_utils.py, policy.py, policy_gradient.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.

Review of the REINFORCE Algorithm

Policy Gradient: recall the policy gradient theorem,

$$\begin{aligned}\nabla_\theta J(\theta) &= E_\tau[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G(\tau)] \\ &\approx \frac{1}{|D|} \sum_{i=1}^{|D|} \sum_{t=1}^{T} \nabla_\theta \log\big(\pi_\theta(a_t^i|s_t^i)\big)\big(\sum_{t=1}^{T} \gamma^{t-1} r(s_t^i, a_t^i)\big)\end{aligned}$$

where τ denotes the trajectory, $G(\tau)$ is the return of trajectory τ. The second line uses $|D|$ sampled trajectories for estimating the expectation, where $r(s_t^i, a_t^i)$ is the immediate return for taking action a_t^i at state s_t^i, and $\sum_{t=1}^{T} \gamma^{t-1} r(s_t^i, a_t^i)$ being the sampled return of trajectory i.

REINFORCE: Recall that the policy at time t' does not affect the reward at time t when $t < t'$. Thus, the REINFORCE estimator can be expressed as the gradient of the following objective function:

$$\nabla_\theta J(\theta) \approx \frac{1}{|D|} \sum_{i=1}^{|D|} \sum_{t=1}^{T} \nabla_\theta \log\big(\pi_\theta(a_t^i|s_t^i)\big) \underbrace{\left(\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)\right)}_{\text{return:} G_t^i}, \tag{1}$$

where D is the set of all trajectories. If different trajectories have different length H_i (with T being the maximum trajectory length, i.e., $H_i \leq T, \forall i$), we can re-write (1) as follows,

$$\nabla_\theta J(\theta) \approx \frac{1}{|D|} \sum_{i=1}^{|D|} \sum_{t=1}^{H_i} \nabla_\theta \log\big(\pi_\theta(a_t^i|s_t^i)\big) \underbrace{\left(\sum_{t'=t}^{H_i} \gamma^{t'-t} r(s_{t'}^i, a_{t'}^i)\right)}_{\text{return:} G_t^i}, \tag{2}$$

where D is the set of all trajectories collected by policy π_θ, $\tau^i = (s_0^i, a_0^i, r_0^i, s_1^i, \cdot, s_{H_i}^i, a_{H_i}^i, r_{H_i}^i)$ is trajectory i and H_i is the length of trajectory i. For your implementation of the REINFORCE algorithm, please follow equation (2).

Instructions on the Implementation

The functions that you need to implement in `network_utils.py`, `policy.py`, `policy_gradient.py`, `baseline_network.py` are enumerated here. Detailed instructions for each function can be found in the comments in each of these files.
Note: The "batch size" for all the arguments is $\sum H_i$ since we already flattened out all the episode observations, actions and rewards for you.

In `network_utils.py`,

- `build_mlp`

In `policy.py`,

- `BaselinePolicy.act`
- `CategoricalPolicy.action_distribution`
- `GaussianPolicy.__init__`
- `GaussianPolicy.std`
- `GaussianPolicy.action_distribution`

In `policy_gradient.py`,

- `PolicyGradient.init_policy`
- `PolicyGradient.get_returns`
- `PolicyGradient.normalize_advantage`
- `PolicyGradient.update_policy`

In `baseline_network.py`,

- `BaselineNetwork.__init__`
- `BaselineNetwork.forward`
- `BaselineNetwork.calculate_advantage`
- `BaselineNetwork.update_baseline`

i. [20 points]: Compute Return To compute the REINFORCE estimator, you will need to calculate the values $\{G_t\}_{t=1}^{H}$ (we drop the trajectory index i for simplicity), where

$$G_t = \sum_{t'=t}^{H} \gamma^{t'-t} r_{t'}$$

Naively, computing all these values takes $O(H^2)$ time. Describe how to compute them in $O(H)$ time.

ii. [50 points]: REINFORCE Algorithm Implementation Implement the REINFORCE algorithm and test it on three OpenAI Gym environments: cartpole, pendulum and cheetah. We have provided some basic tests to sanity check your implementation.

Please note that the tests are not comprehensive, and passing them does not guarantee a correct implementation. Use the following command to run the tests:

```
python run_basic_tests.py
```

You can also add additional tests of your own design in `tests/test_basic.py`. We have the following expectation about performance to receive full credit:

- cartpole: Should reach the max reward of 200 (although it may not stay there)
- pendulum: Should reach the max reward of 1000 (although it may not stay there)
- cheetah: Should reach at least 200 (Could be as large as 950)

Note: In order to get good performance, you may need to adjust hyperparameters such as initialization of logstd, architecture configuration, learning rate, etc.

iii. [30 points]: Baseline Subtraction One difficulty of training with the REINFORCE algorithm is the sampled return(s) G_t^i 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:

$$MSE(\phi) = \frac{1}{|D|}\sum_{i=1}^{|D|}\sum_{t=1}^{H_i}(b_\phi(s_t^i) - G_t^i)^2$$

The general form for running your REINFORCE implementation is as follows:

```
python main.py --env-name ENV --seed SEES --no-baseline
```

if not using a baseline, or

```
python main.py --env-name ENV --seed SEES --baseline
```

if using a baseline. Here `ENV` should be `cartpole, pendulum` or, `cheetah`, and `SEED` should be a positive integer.

For each of the 3 environments, choose 3 random seeds and run the algorithm with both without baseline and with baseline. Then plot the result using

```
python plot.py --env-name ENV --seeds SEEDS
```

where `SEEDS` should be comma-separated list of seeds which you want to plot (e.g. `--seeds 1,2,3`).

Please include the plots (one for each environment) in your notebook, and comment on whether or not you observe improved performance using a baseline.

2 Practice Problem, You don't need to submit this: Value iteration and Policy Iteration.

In this problem you will program value iteration and policy iteration. You will use environments implementing the OpenAI Gym Environment API. For more information on Gym and the API, see https://gym.openai.com/. We will be working with different versions of Frozen Lake environments.

In this domain, the agent starts at a fixed starting position, marked with "S". The agent can move up, down, left and right. In deterministic versions, the up action action will always move the agent up, left will always move left, etc. In the stochastic versions, the up action will move up with a probability of 1/3, left with a probability of 1/3 and right with a probability of 1/3.

There are three different tile types: frozen, hole and ground. When the agent lands on a frozen tile, it receives 0 reward. When the agent lands on a hole tile it receives 0 reward and the episode ends. Then the agent lands on the goal tile, it receives +1 reward and the episode ends. We have provided you 2 different maps. An invalid action (e.g. towards the boundary) will keep the agent where it is.

States are represented as integers numbered from left to right, top to bottom starting from 0. So the upper left corner of the 4x4 map is state 0, the bottom right corner is state 15.

You will implement value iteration and policy iteration using the provided environments. Some function templates are provided for you to fill in. Specific coding instructions are provided in the source code files.

Be careful implementing value iteration and policy evaluation. Keep in mind that in this environment the reward function depends on the current state, the current action and the next state. Also terminal states are slightly different.

i. [0 points] Deterministic Frozen Lake Answer the following questions for the maps `Deterministic-4x4-FrozenLake-v0` and `Deterministic-8x8-FrozenLake-v0`.

1. Using the environment, find the optimal policy iteration. Record the time taken for execution, then number of policy improvement steps and the total number of policy evaluation steps. Use $\gamma = 0.9$. Use a stopping tolerance of 10^{-3} for the policy evaluations.
2. What is the optimal policy for this map? Show as a grid of letters with "U", "D", "L", "R" representing the actions up, down, left, right respectively. An example output is shown below.

3. Find the value function of this policy. Plot it as a color image as shown, where each square has its value as a color.

4. Find the optimal value function directly using the value iteration. Record the time taken for execution, and the number of iterations required. Use $\gamma = 0.9$, a stopping tolerance of 10^{-3}.

5. Plot this value function as a color image, where each square shows its value as a color.

6. Which algorithm was faster? Which took less iterations?

7. Are there any differences in the value function?

8. Convert the optimal value function to the optimal policy. Show as a grid of letters with "U", "D", "L", "R" representing the actions up, down, left, right respectively.

9. Write an agent that executes the optimal policy. Record the total cumulative discounted reward. Does the value match the value computed for the starting state? If not, explain why.

```
LLLL
DDDD
UUUU
RRRR
```

Figure 1: Example policy for `FrozenLake-v0`

ii. [0 points]: Stochastic Frozen Lake Answer the following questions for the map `Stochastic-8x8-FrozenLake-v0`.

1. Using value iteration, find the optimal value function. Record the time taken for execution, then number of policy improvement steps and the total number of policy evaluation steps. Use $\gamma = 0.9$. Use a stopping tolerance of 10^{-3}.

2. Plot the value function as a color map. Is the value function different compared to deterministic versions of the maps?

3. Convert this value function to the optimal policy and include it in the notebook.

4. Does the optimal policy differ from the optimal policy in the deterministic maps? If so, pick a state where the policy differs and explain why the action is different.

5. Write an agent that executes the optimal policy. Run this agent 100 times on the map and record the total cumulative discounted reward. Average the results. Does this value match the value computed for the starting state? If not, explain why.

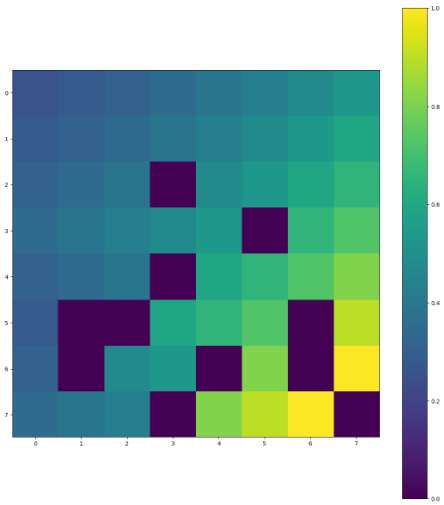

Figure 2: Example of value function color plot. Make sure you include the color bar or some kind of key.