RL Lab 2

Harsh Raj (180010017)
Akhilesh Bharadwaj (180010009)

Note: Code for all questions (including simulations etc) is attached in the submitted zip. Artifacts including the images generated are also attached in the zip. Please make sure to install dependencies mentioned in the requirements.txt before running any submitted code.

```
# Install dependencies
pip3 install -r requirements.txt

# run codes corresponding to each question
python3 runner.py

# change hyperparameters:
# either change manually from conf/configs.yaml for each ques
# or change dynamically from command line
python3 runner.py seed=5
# config management is done via hydra. Read more
https://github.com/facebookresearch/hydra
```

1. Bernoulli Distribution of underlying rewards for each arm:

Each Algorithm is given 5_000 chances per episode. In the case of Bernoulli distribution, each arm has an underlying distribution defined by p=1/(arm_index+1), index starts from 1.

- Epsilon Greedy:
    - Probability of randomly selecting an action: 0.3

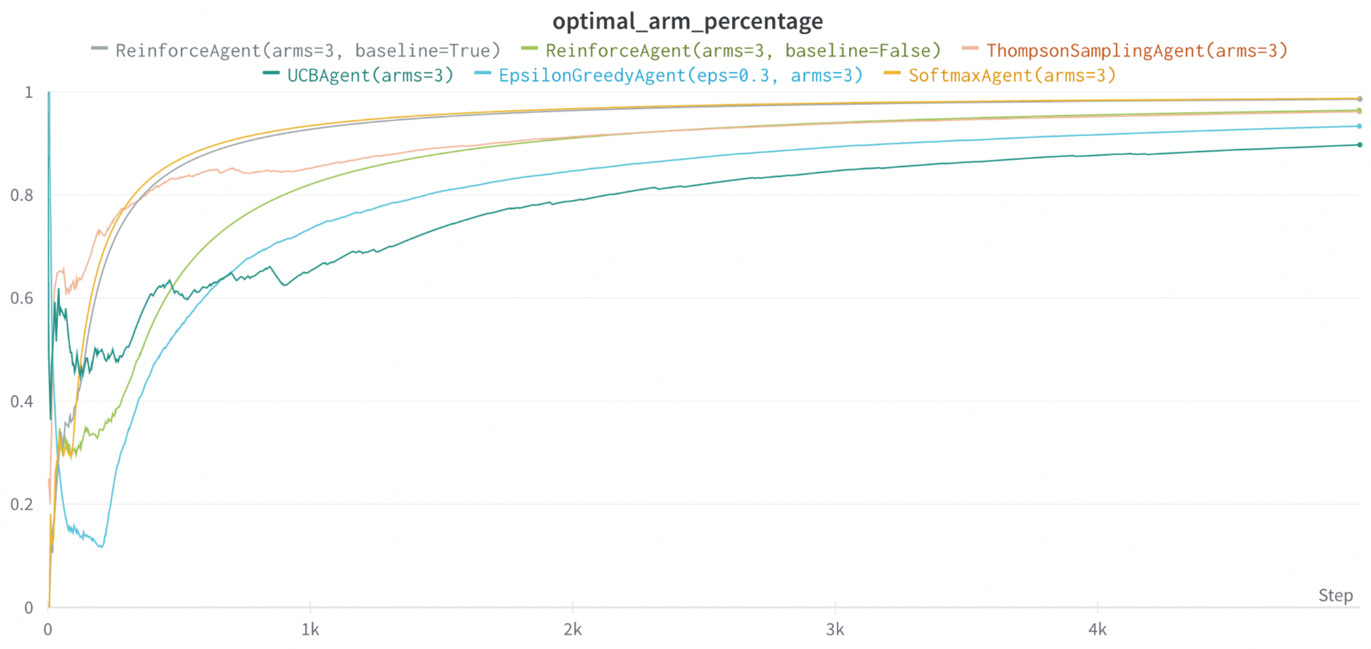
- Softmax:
    - Initial Temperature: 1_000
    - Decay factor: 0.9, ie. after each time step, `temp = temp * decay_factor`
    - `mean / temp` ratio is clipped to be in range: 0.001 - 700. This is to ensure that the softmax does not shoot up to infinity

- UCB:
    - All default arguments.

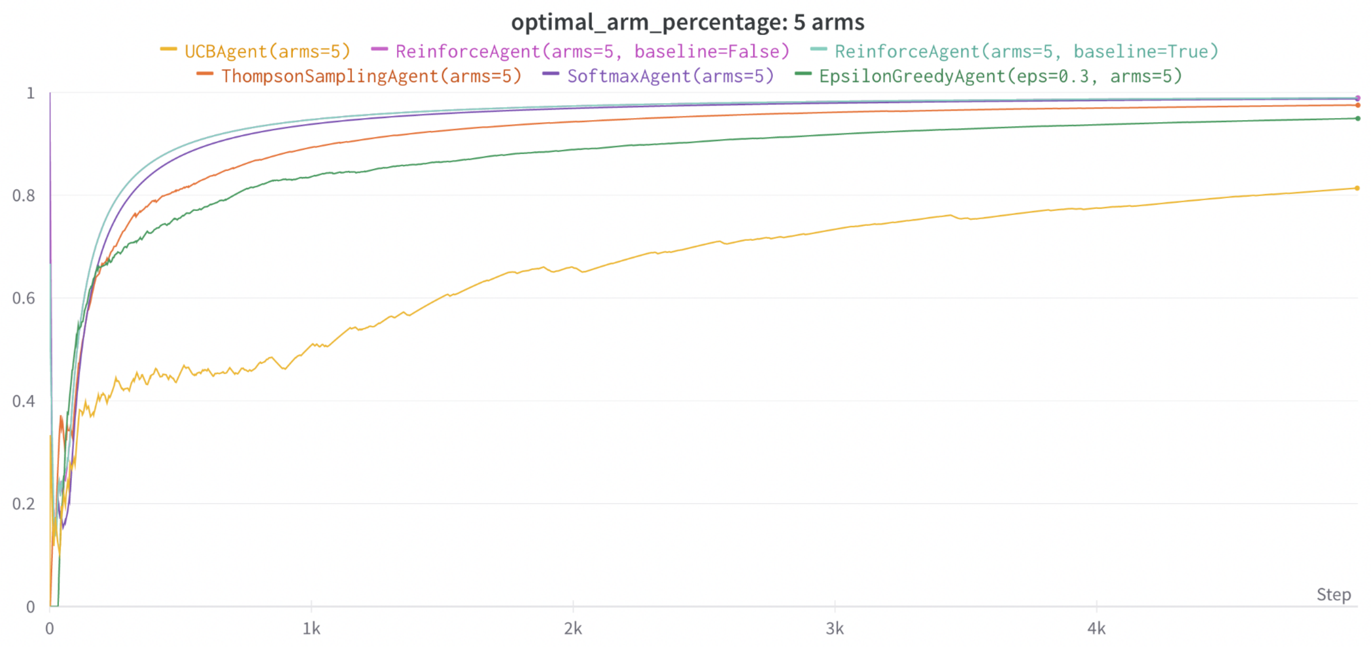
- Thompson Sampling:
    - All default arguments.

- Reinforce:
    - We experiment with alpha=0.8 and beta=0.3,

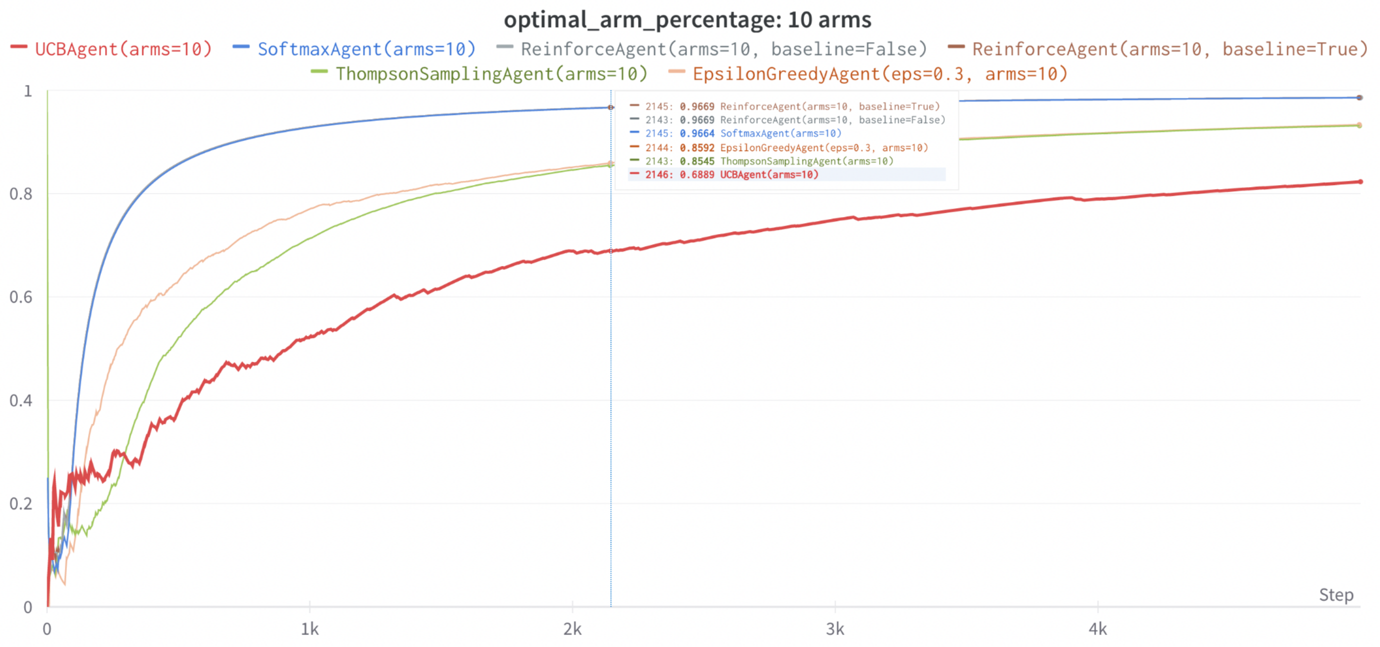
      ```
      running_mean_reward = (1-alpha) * running_mean_reward + alpha * reward
      ```
    - The preference of each arm is clipped in range: 0 - 200. This is to ensure that the softmax does not shoot up to infinity.
    - For baseline, we use the running mean reward.



Ratio of optimal arm hits for different algorithms, for 3-arm Bernoulli Bandit



Ratio of optimal arm hits for different algorithms, for 5-arm Bernoulli Bandit



Ratio of optimal arm hits for different algorithms, for 10-arm Bernoulli Bandit
Note that Reinforce and Softmax perform equally well in this case

All the charts can be accessed online on this wandb repo.

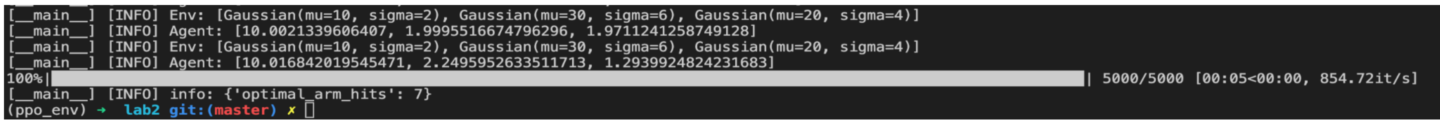
2. Gaussian Distribution of underlying rewards for each arm:

Each Algorithm is given 5_000 chances per episode. For debugging purposes, we tried the algorithms with fixed underlying reward distribution as shown in the screenshot below. We experimented with various other initializations, but results were quite similar

To do:

- Fix Reinforce for Gaussian
- Get thompson working for Gaussian Distribution

Thompson Sampling:

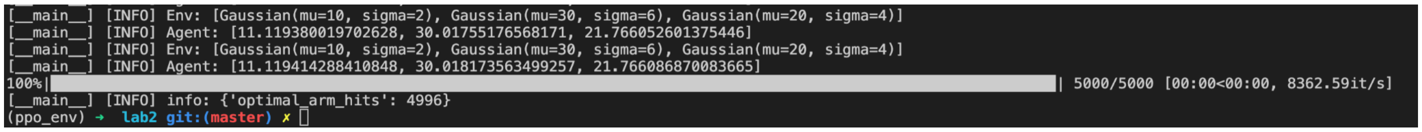


```
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [10.0021339606407, 1.9995516674796296, 1.9711241258749128]
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [10.016842019545471, 2.2495952633511713, 1.2939924824231683]
100%|████████████████████████████████████████████████████████████| 5000/5000 [00:05<00:00, 854.72it/s]
[__main__] [INFO] info: {'optimal_arm_hits': 7}
(ppo_env) ➜  lab2 git:(master) ✗ 
```

Thompson sampling is not able to accurately identify the underlying mean of each arm even after 5_000 chances. Most of the time, the arm results in selecting the second arm.

Remarks: From the paper, it is not very clear, what should be the initial assumption of mean and std of the underlying distribution. It is observed that the choice of std affects the number of times the optimal arm is selected, as with different rewards, the estimated mean changes quite a lot at each timestep.

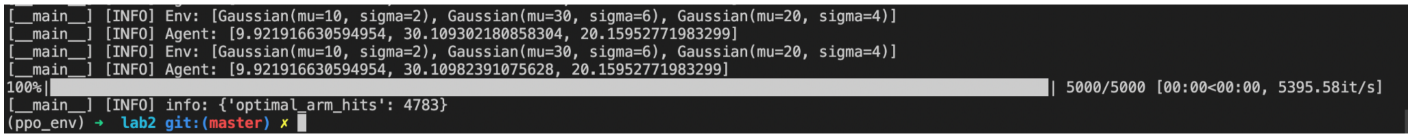
UCB:



```
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [11.119380019702628, 30.01755176568171, 21.766052601375446]
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [11.119414288410848, 30.018173563499257, 21.766086870083665]
100%|████████████████████████████████████████████████████████████| 5000/5000 [00:00<00:00, 8362.59it/s]
[__main__] [INFO] info: {'optimal_arm_hits': 4996}
(ppo_env) ➜  lab2 git:(master) ✗ 
```

UCB has been performing quite well. Identifying the underlying mean pretty well, and hitting optimal arms quite high.

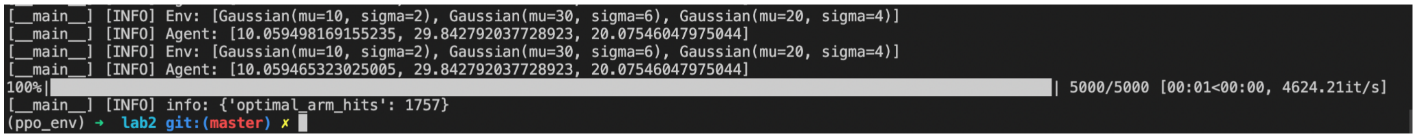
Epsilon Greedy



```
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [9.921916630594954, 30.109302180858304, 20.15952771983299]
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [9.921916630594954, 30.10982391075628, 20.15952771983299]
100%|████████████████████████████████████████████████████████████| 5000/5000 [00:00<00:00, 5395.58it/s]
[__main__] [INFO] info: {'optimal_arm_hits': 4783}
(ppo_env) ➜  lab2 git:(master) ✗ 
```

Epsilon Greedy Agent also performs reasonably well over here.

Softmax



```
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [10.059498169155235, 29.842792037728923, 20.07546047975044]
[__main__] [INFO] Env: [Gaussian(mu=10, sigma=2), Gaussian(mu=30, sigma=6), Gaussian(mu=20, sigma=4)]
[__main__] [INFO] Agent: [10.059465323025005, 29.842792037728923, 20.07546047975044]
100%|████████████████████████████████████████████████████████████████| 5000/5000 [00:01<00:00, 4624.21it/s]
[__main__] [INFO] info: {'optimal_arm_hits': 1757}
(ppo_env) ➜  lab2 git:(master) ✗ 
```

Though softmax is able to capture the mean of underlying reward distribution, it has quite high regret in this case.