AI Hw4 Report

112550097 楊弘奕

1. Implementation

a. Agent

(1)_init_()

```
def __init__(self, k, epsilon):
    self.k = k
    self.epsilon = epsilon
    self.reset()
```

(a) Pass parameters to create an instance of the agent
(b) Set attributes based on input parameters
(c) Call reset()

(2)reset()

```
def reset(self):
    # store estimated expected reward of each action
    self.q_values = np.zeros(self.k)
    # store number of times each action was selected
    self.action_cnt = np.zeros(self.k)
    # store total reward received for each action
    self.reward_sum = np.zeros(self.k)
```

(a) Initialize q_values, action counter, sum of rewards obtained by arm[i] as zeros

(3)select_action()

```
def select_action(self):
    if np.random.rand() < self.epsilon:
        # randomly select with p < epsilon
        return np.random.randint(0, self.k)
    else:
        # else, select action with max expected reward
        return np.argmax(self.q_values)
```

(a) implement epsilon-greedy
    (i) There is epsilon of probability for the agent to explore all arms
    (ii) Else, the agent would select the best q_values for now.

(4)update_q()

```python
def update_q(self, action, reward):
    # update the number of times action was selected
    self.action_cnt[action] += 1
    self.reward_sum[action] += reward
    self.q_values[action] = self.reward_sum[action] / self.action_cnt[action]
```

(a) Action count += 1
(b) Total reward obtained from arm[action] += reward in this step
(c) Use new empirical mean as the q value of arm[action]

b. Agent - Constant Step Size

(1)Update _init_()

```python
def __init__(self, k, epsilon, step_size = None):
    self.k = k
    self.epsilon = epsilon
    self.step_size = step_size
    self.reset()
```

(a) Add step_size parameter
(b) Set default as None, or set step_size to the parameter passed in

(2)Update update_q()

```python
def update_q(self, action, reward):
    # update the number of times action was selected
    self.action_cnt[action] += 1

    if self.step_size is None:
        self.reward_sum[action] += reward
        self.q_values[action] = self.reward_sum[action] / self.action_cnt[action]
    else:
        # update the estimated expected reward of the chosen action
        self.q_values[action] += self.step_size * (reward - self.q_values[action])
        # update the total reward received for the chosen action
        self.reward_sum[action] += reward
```

(a) Remain sample average method for step_size = None
(b) Else, update the q value with exponential recency-weighted average

$$Q(a) \leftarrow Q(a) + \alpha \cdot (R - Q(a)),\ \alpha = 0.1\ in\ this\ experiment$$

c. Environment

(1)_init_()

```
def __init__(self, k):
    self.k = k
    self.reset()
```

(a) Pass parameters to create an instance of the environment
(b) Set attributes based on input parameters
(c) Call reset()

(2)reset()

```
def reset(self):
    # true mean of the k arms, from N(0, 1)
    self.true_mean = np.random.normal(0, 1, self.k)
    self.action_history = []
    self.reward_history = []
    self.optimal_action = np.argmax(self.true_mean)
```

(a) Randomly set the initial true mean for k arms
(b) Initialize action_history, reward_history, and optimal action as the arm with the best true mean for now.

(3)step()

```
def step(self, action):
    # reward = N(true_mean[ith_ bandit], 1)
    reward = np.random.normal(self.true_mean[action], 1)
    self.action_history.append(action)
    self.reward_history.append(reward)
    return reward
```

(a) At each step, the reward for the chosen arm is drawn from a Normal distribution whose mean equals that arm's current true value.

(4)check_optimal_action()

```
def check_optimal_action(self, action):
    if self.true_mean[action] == self.true_mean[self.optimal_action]:
        return True
    return False
```

(a) Since there might be multiple arms that have max true mean, all of them must be counted as best action, so if the true mean of the action is same as the optimal_action now, it's consider as the best action.

d. Environment - Non-Stationary

a. Update _init_()

```
def __init__(self, k, stationary=True):
    self.k = k
    self.stationary = stationary
    self.reset()
```

i. Add stationary parameter
ii. Set default as True, or set stationary to the parameter passed in

b. Update step()

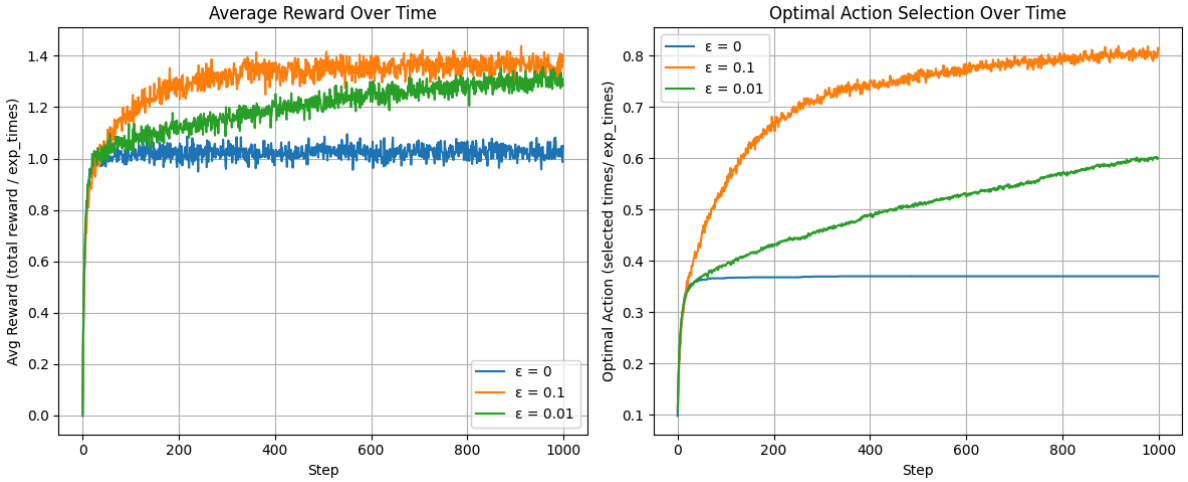
```
def step(self, action):
    # reward = N(true_mean[ith_ bandit], 1)
    reward = np.random.normal(self.true_mean[action], 1)
    self.action_history.append(action)
    self.reward_history.append(reward)

    # if not self.stationary:
    if not self.stationary:
        # random walk
        self.true_mean += np.random.normal(0, 0.01, self.k)
        # update optimal action
        self.optimal_action = np.argmax(self.true_mean)
    return reward
```

i. If the environment is not stationary, increment the true mean of each arm with random walk with N(0, 0.01)
ii. If the environment is non-stationary, the true mean reward of each action will take independent random walks by adding a normally distributed increment with mean 0 and standard deviation 0.01 to all the true mean rewards.
iii. Update the optimal action with new arm that has the max true mean

2. Experiment Results

a. Part 3 Stationary Environment



Under stationary environment, we observe how 3 diffrent value of ε impact performace of agent:

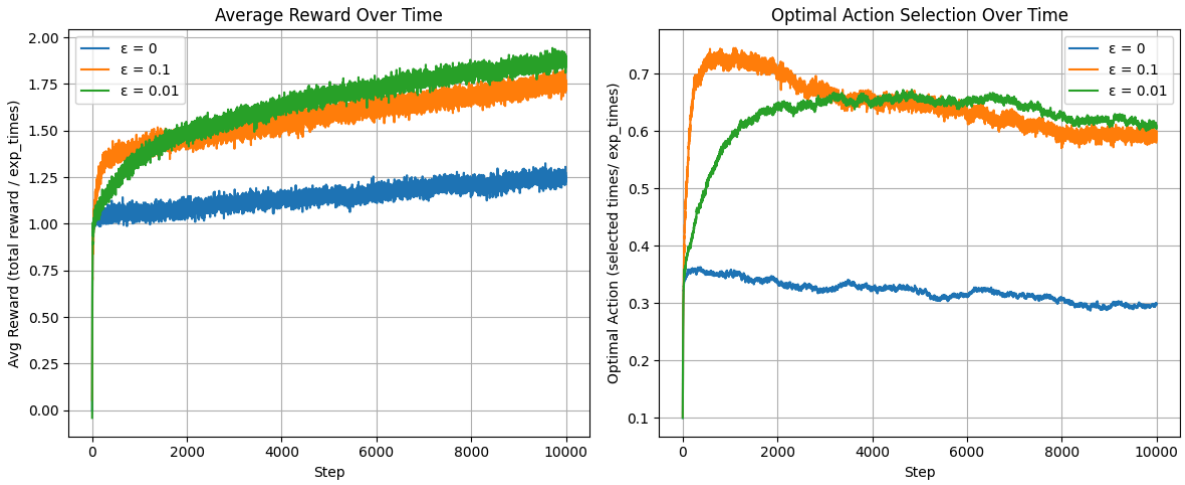
(1) Agent with ε = 0.1 get the higheset average reward because of its proper exploration. Enable the agent to find the optimal arm earlier and exploit it. This behavior is clearly reflected in the optimal action selection plot, where the agent selects the best arm about 80% of the time by the end of the step.

(2) Agent with ε = 0.01 performs second best. Although it converges slowly, but still manages to reach about 1.4 by step 1000. And the percentage of optimal action selection is also steadily increasing along the step, showing that a small amount of exploration is still beneficial in the long run.

(3) Agent with ε = 0 performs the worst. Since it never explores, the best empirical mean of the agent would be the first arm it selected and has a reward > 0, once it discover this, it would stick with this arm for all steps and never discovers better ones. As a result, both its average reward remain significantly lower than the other 2 strategies and its optimal action selection percentage remain horizontal line after first arm with reward > 0 discovered.

(4) The agent with ε = 0 performs the worst. Since it never explores, it only selects the action that initially appears best. In a stationary environment, all q-values start at zero, so the first selected arm becomes dominant as soon as it receives a positive reward—its empirical mean then exceeds the others, which remain at zero. From that point on, the agent continues to exploit this suboptimal arm, even if better options exist, simply because it never gathers more information. As a result, its average reward plateaus early and remains significantly lower than the other two strategies. This behavior is also reflected in the optimal action selection plot, where the percentage quickly stabilizes into a flat line, determined solely by the proportion of agents who happened to select the optimal arm on their first try.

b. Part 5 Non-Stationary Environment



Compare to experiment in Part 3, I observe 3 phenomenon under non-stationary environment:

(1) Average reward increases

In the non-stationary environment, the true mean of each arm changes over time through a random walk. This introduces more opportunities for certain arms to become better than they originally were. If an agent continues to explore, it has a higher chance of discovering these newly optimal arms.

As a result, both agents with ε > 0 achieve higher average rewards than in the stationary setting.

As for the agent with ε = 0, since it never explores, its average reward is mostly influenced by the first arm it selects with a reward greater than 0. This makes the Q-value of that arm quickly surpass the others (which start at 0), causing the agent to keep choosing the same arm repeatedly. However, in a non-stationary environment where the true mean rewards change over time, the agent may start receiving lower rewards from that arm. Once the Q-value drops below other arms, the agent naturally shifts to a better option, allowing it to adapt over time despite having no explicit exploration.

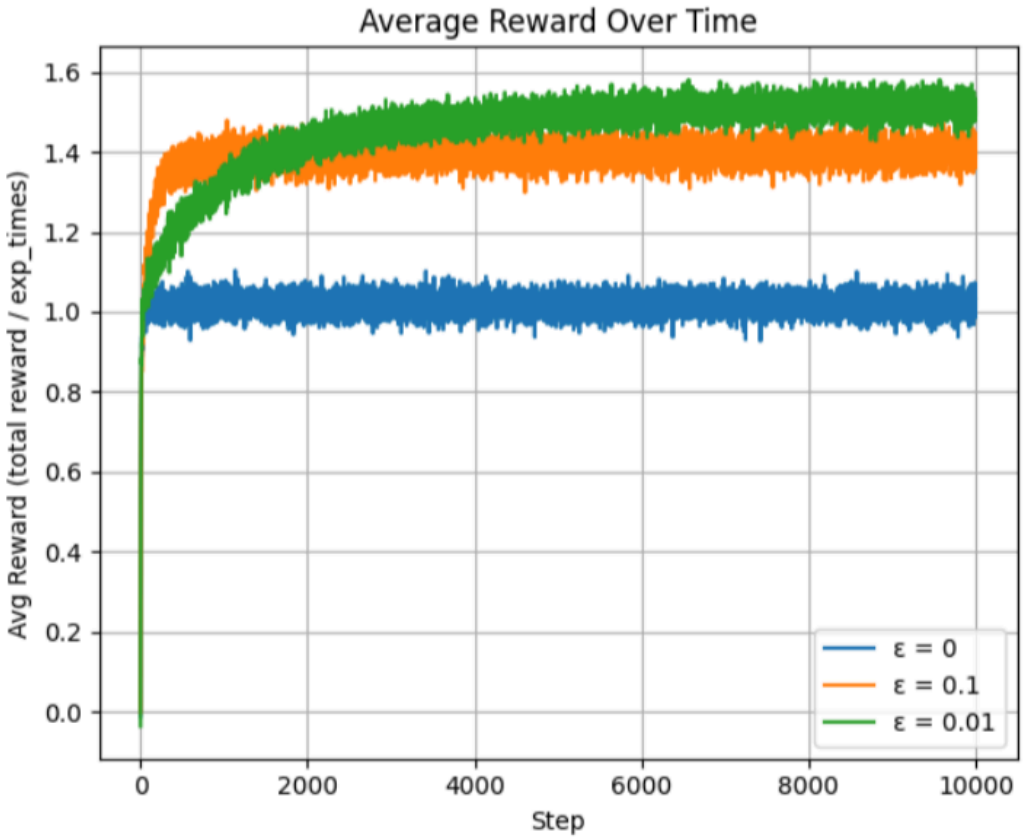
(2) Optimal action selection percentage decreases

Although the average reward is increasing, the optimal selection percentage is generally decreasing. As mentioned above, the true mean of each arm differs along the time, therefore the optimal arm is no longer fixed, it keeps changing as the true mean drifts. The agent uses a sample average to estimate the reward would obtain from arm[action], this makes it too slow to react to current changes. Since its Q-values are based on cumulative past data, they tend to reflect historical performance rather than current conditions.

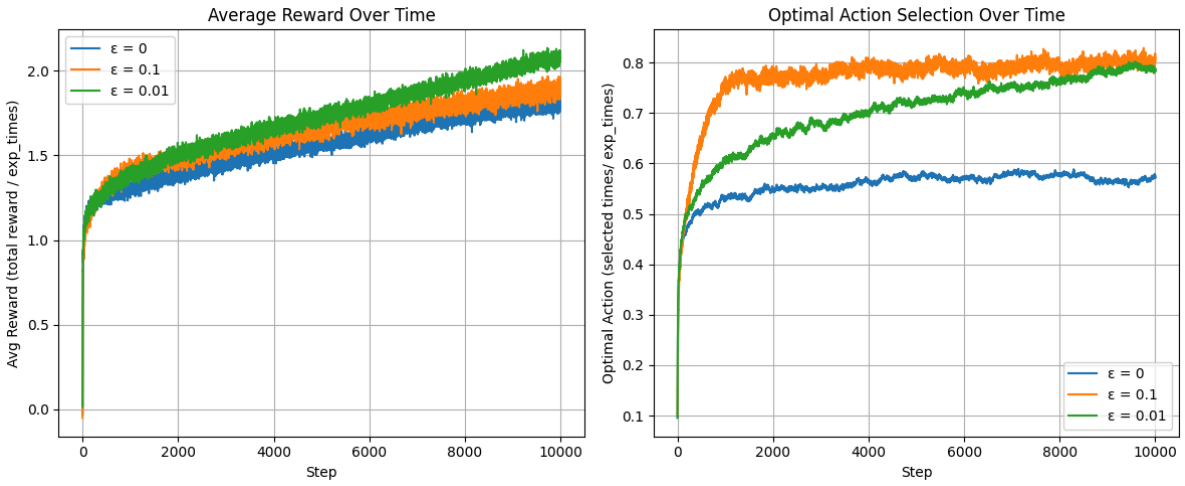
(3) Agent with ε = 0.01 outperform agent with ε = 0.1 after 2000 steps

In the long run, agents might have already adapted to the environment and identified good actions. At this stage, it becomes more beneficial to exploit the known best actions rather than continue exploring, which carries unnecessary risk. Since the agent with ε = 0.01 explores less than the one with ε = 0.1, it focuses more on exploiting the known high-reward actions, leading to better overall performance in the long run.

This phenomenon can also be seen in a stationary environment, in this experiment below, we can see that the agent with ε = 0.01 also surpasses the agent with ε = 0.1 after around step 2000.



a. Part 7 Constant Step-Size



Compare to experiment in Part 5, I observe 2 phenomenon under non-stationary environment with constant step size agent:

(1) Average reward increases

The average reward converges to a higher value because the constant step-size update makes the agent focus more on recent rewards rather than evaluating an arm based on historical averages as the sample average method did. This allows the agent to adapt more quickly to the changing reward distribution, especially benefiting the ε = 0 agent, since it sticks to the known optimal arm, focusing on recent rewards makes it able to find out whether the performance is still optimal at that stage quicker.

Similar to the result in Part 5, agent with ε = 0.01 still outperforms agent with ε = 0.1 after around step 2000. The reason is also the same as mentioned above, agent with with ε = 0.01 reducing unnecessary exploration when the agent has already gathered enough knowledge about the environment.

(2) Optimal selection of all agents converge to a higher value

The plot shows a trend similar to the one in the stationary environment. Agent with ε = 0.1 converges the fastest, while agent with ε = 0.01 converges more slowly but eventually reaches a similar optimal action selection rate as agent with ε = 0.1. Although agent with ε = 0 still performs the worst, its performance slightly improves. This is because the constant step-size update allows the agent to detect when the performance of its initially chosen arm starts to decline, enabling it to switch to a better arm earlier than it could with the sample-average method.

3. Discussion

a. Problem encountered

(1) Plotting over step instead of time

(a) Initially, I misunderstood the requirement and plotted the results over independent experiments instead of averaging results over time across multiple experiments. As a result, the early plots for average reward showed high variance and no convergence. Each point represented a different experiment's final reward, rather than a time-evolving learning process. After correcting this and averaging over time (for each step t, compute average over 2000 runs), the curves began to exhibit the expected convergence behaviors.

(2) Many optimal action

(a) Initially, I didn't realize that there might be multiple optimal arms, so I count selected arm as the optimal selection only if the selected arm equals to the index of argmax(true_mean). This lead to the problem of low optimal selection percentage. Then I changed the condition of cheking optimal selection to true_mean[action] == ture_mean[optimal_action], that is if the true mean of action if same as the true mean of current optimal action for now, it's counted as the optimal selection.

b. Others

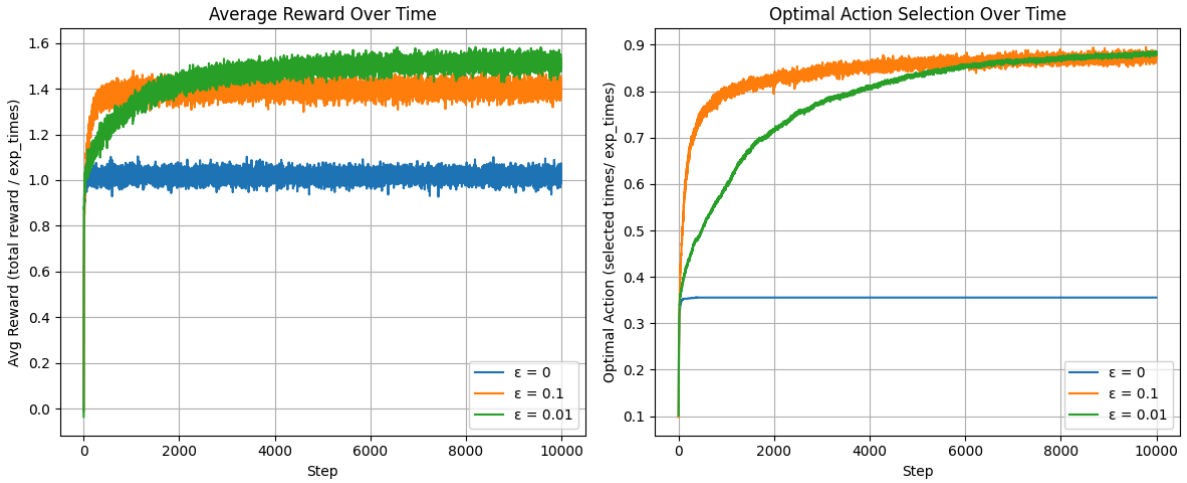
(1) Close to true mean?

(a) To further examine whether the agent's estimated Q-values (empirical mean) can converge to the true mean of the optimal arm, I modified the experiment in Part 3 under a stationary environment, and extended the number of steps per run to 10,000. And print out:

(i) The average true mean of the optimal arm over each experiment

(ii) The maximum average reward achieved by each agent

(b) Result




```
PS C:\Users\harry\OneDrive\Desktop\Courses\AI概\hw4> python -u "c:\Users\harry\OneDrive\Desktop\Courses\AI概\hw4\cal.py"
max_avg_reward:  [1.1036019560199999, 1.50603275428, 1.581878007255]
Average True Mean of Optimal Action:  [1.53955789, 1.55619762, 1.54717906]
```

(c) Conclusion

This experiment confirms that, in a stationary setting and with sufficient steps, the empirical Q-values of agents with exploration do converge to the true mean of the optimal arm. Meanwhile, purely greedy agents (ε = 0) remain trapped with suboptimal estimates due to their lack of exploration.