HW4

112550109 楊榮竣

1 Implement

1.1 Agent

Class Variables:

1. **size : the number of arms.**
2. **epsilon : the value of epsilon.**
3. **q_values : a list of the estimated rewards of arms.**
4. **action_counts : a list of the count value of arms.**
5. **constant : (In part 6) step_size of agent.**

Class Method:

1.__init__()

Original(Pary 2): Firstly, this function update value of **self.size** and **self.epsilon** with input variable **k** and **epsilon**. Then, it use **reset()** function to initialize the **q_values** and **action_counts**.

Step-Size Update(Part 6): In this part, there is additional input **constant** which initial setting **None**. This function will store input value into **self.constant**. This variable will affect the motion of agent.

2.reset()

In this function, I use **np.zeros()** function to create the list which is fulled with zero. This function initialize two list **q_values** and **action_counts** to clear the history record.

3.select_action()

Firstly, this function use **random.uniform(0,1)** to generate a number between 0 and 1. Then, it will choose the movement by generated number **p**. If p is less than **epsilon**, it choose explore motion and randomly choose a arm by using **random.randint()** function. Otherwise, it will use **np.argmax()** to choose arm with maximal q_valies which be getten from history.

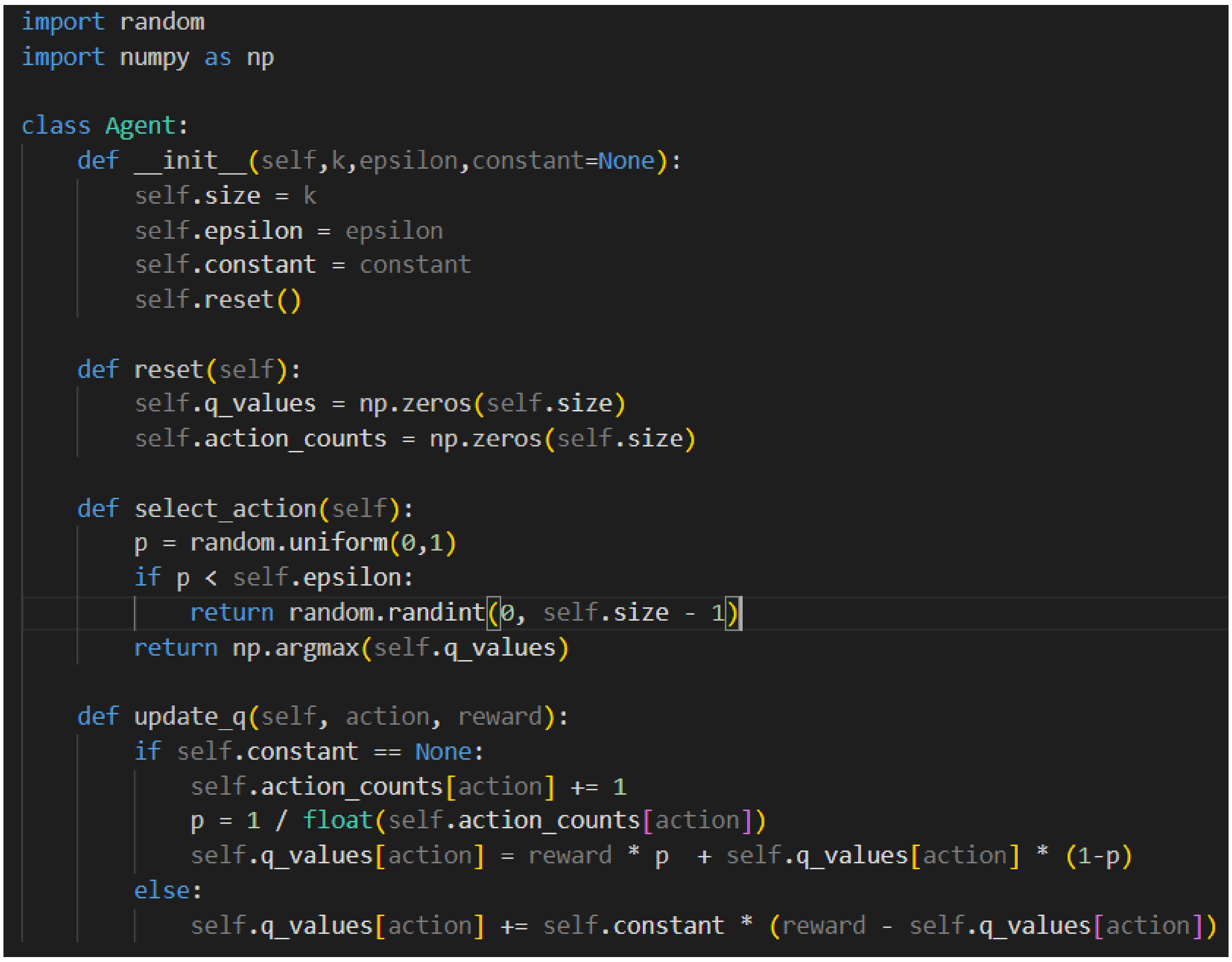
4.update_q()

Step-Size Update(Part 2): Firstly, update **action_counts** and calculate percentage of new reward in total. Using following function to update **q_values**.

$$Q(a) \leftarrow R \cdot p + Q(a) \cdot (1 - p)$$

Step-Size Update(Part 6): Use this function to update **q_values**.

$$Q(a) \leftarrow Q(a) + \alpha \cdot (R - Q(a))$$



```
import random
import numpy as np

class Agent:
    def __init__(self,k,epsilon,constant=None):
        self.size = k
        self.epsilon = epsilon
        self.constant = constant
        self.reset()

    def reset(self):
        self.q_values = np.zeros(self.size)
        self.action_counts = np.zeros(self.size)

    def select_action(self):
        p = random.uniform(0,1)
        if p < self.epsilon:
            return random.randint(0, self.size - 1)
        return np.argmax(self.q_values)

    def update_q(self, action, reward):
        if self.constant == None:
            self.action_counts[action] += 1
            p = 1 / float(self.action_counts[action])
            self.q_values[action] = reward * p  + self.q_values[action] * (1-p)
        else:
            self.q_values[action] += self.constant * (reward - self.q_values[action])
```

Final Version of Agent

1.2 Game Environment(BanditEnv)

Class Variables:

1. **size : the number of arms.**
2. **actions : a list to store history of actions.**

3. **rewards : a list to store history of rewards.**
4. **means : a list of mean value of each arms.**
5. **opt_actions : the best choices in these arms each time, which means the arm with the largest mean.**
6. **stationary : (In part 4) To decide this environment wether is stationary.**

Class Method:

1. __init__()

Stationary(Part 1): Storing the value of **self.size** with input k and calling reset() function to reset environment.

Non-stationary(Part 5): New variable **stationary** to decide this environment wether is stationary.

2. reset()

The main function of this method is to reset this environment, so I set actions, rewards, and opt_actions as [] to clear the history and update means by **standard_normal()** function to get the random value from standard normal distribution.

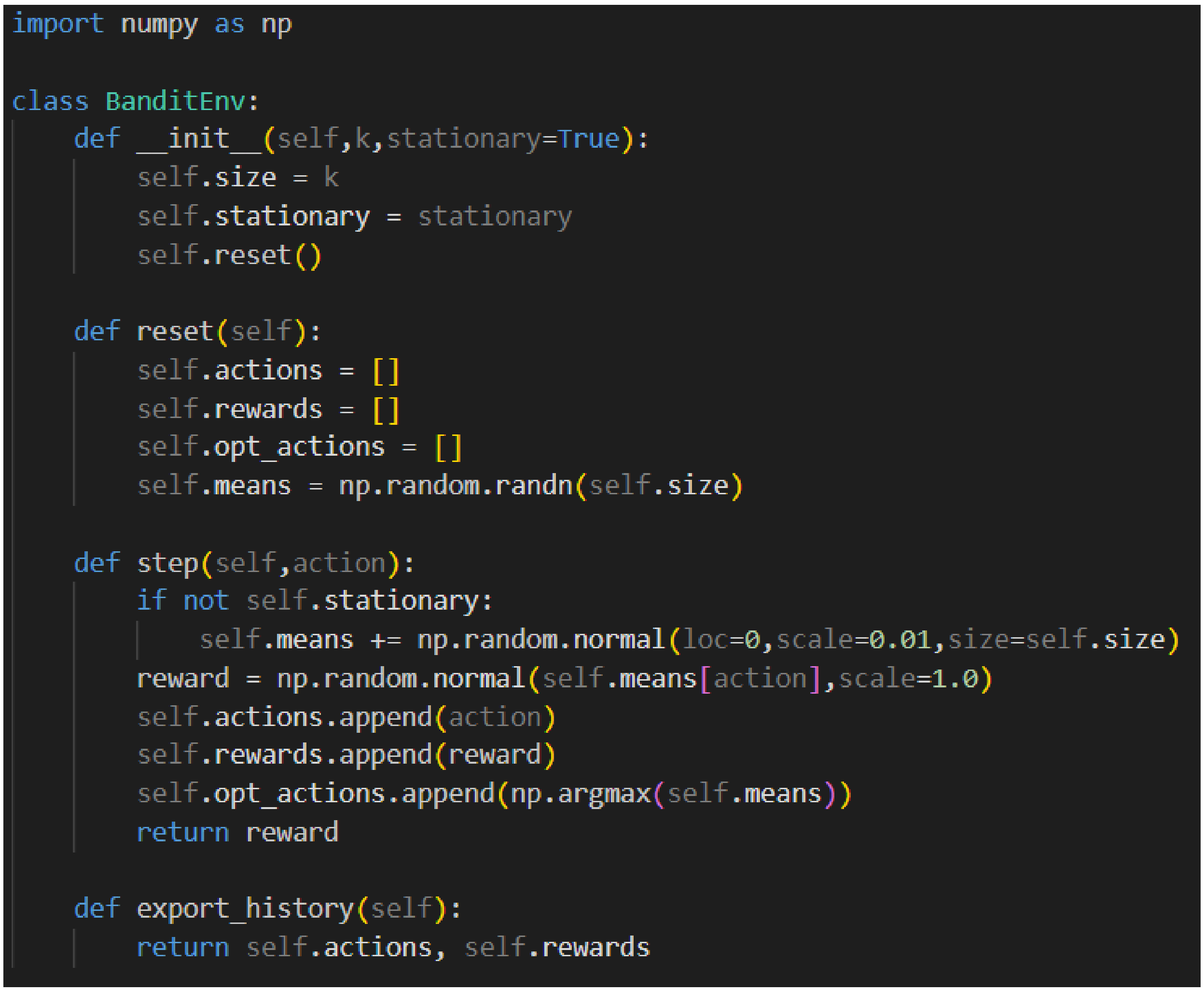
3. step()

Stationary(Part 1): Using **np.random.normal()** to get a random reward from Gaussian distribution with variance 1 and specific mean value. I use **np.argmax()** function to find the optimal selection in current time. Then, I use **.append()** to store history of this step into actions, rewards, and opt_actions. Finally, this function will return the reward.

Non-stationary(Part 5): In this part, this function will check the stationary wether is true in the beginning. **self.means()** will be increased values which are obtained from **np.random.normal()** function if stationary is true.

4. export_history()

return actions and rewards



```python
import numpy as np

class BanditEnv:
    def __init__(self,k,stationary=True):
        self.size = k
        self.stationary = stationary
        self.reset()

    def reset(self):
        self.actions = []
        self.rewards = []
        self.opt_actions = []
        self.means = np.random.randn(self.size)

    def step(self,action):
        if not self.stationary:
            self.means += np.random.normal(loc=0,scale=0.01,size=self.size)
        reward = np.random.normal(self.means[action],scale=1.0)
        self.actions.append(action)
        self.rewards.append(reward)
        self.opt_actions.append(np.argmax(self.means))
        return reward

    def export_history(self):
        return self.actions, self.rewards
```

Final Version of BanditEnv

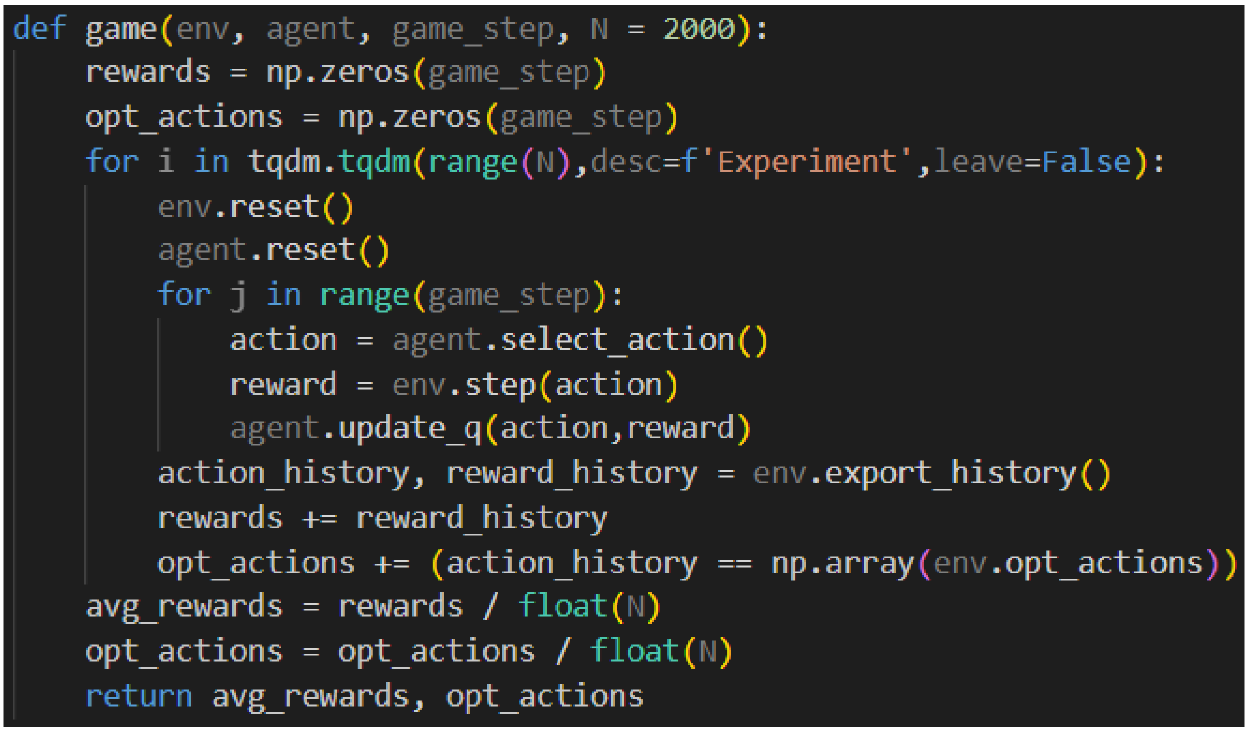
1.3 Main

game(env,agent,game_step,N=2000)

1. **env : the environment of game.**
2. **agent : the agent which need to use in the game.**
3. **game_step : the number of step in each round of game.**
4. **N : the number of experiment.(Initial setting 2000)**

This function is used to control the procession of game. In the beginning, this function will reset environment and the agent before each experiment start. Then, it uses agent to get the decision of each step in the game and put the decision into game environment. After game ending, it will obtain history of game procession by using **export_history()** function and add result on **rewards** and **opt_actions**. Finally, the results of list will divide the number of experiment. Then, I divide **N** on them to calculate the average reward. Furthermore, I use **tqdm** to keep track the process of experiment.

In each experiment, this function will update **opt_actions**. Firstly, I retrieve the history of opt_action from **env.opt_actions**. Then, I create a list with zero or one by equaling **action_history** and **env.opt_action**. The element of this new list represent that the element with responding position in **action_history** wether its action is optimal action. Finally, this function will calculate the sum of **opt_action** and new list to update the value of **opt_action**.

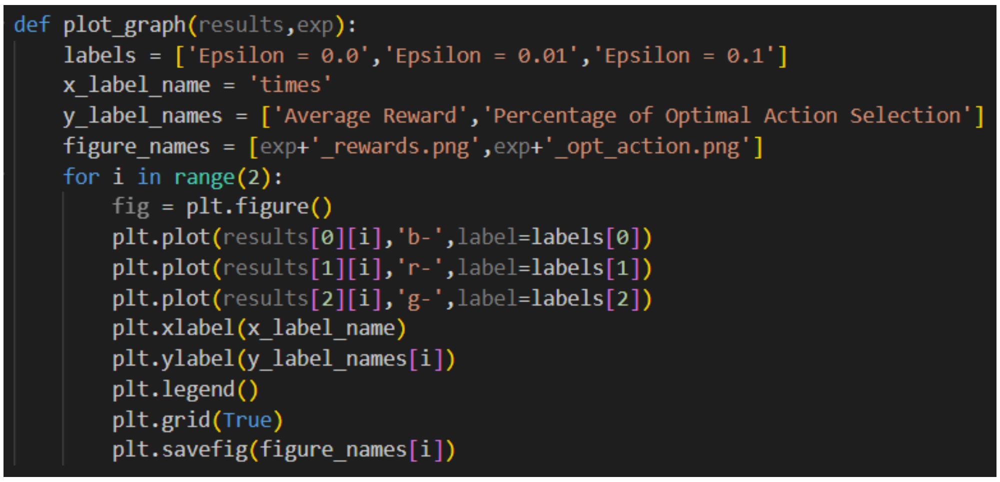


```
def game(env, agent, game_step, N = 2000):
    rewards = np.zeros(game_step)
    opt_actions = np.zeros(game_step)
    for i in tqdm.tqdm(range(N),desc=f'Experiment',leave=False):
        env.reset()
        agent.reset()
        for j in range(game_step):
            action = agent.select_action()
            reward = env.step(action)
            agent.update_q(action,reward)
        action_history, reward_history = env.export_history()
        rewards += reward_history
        opt_actions += (action_history == np.array(env.opt_actions))
    avg_rewards = rewards / float(N)
    opt_actions = opt_actions / float(N)
    return avg_rewards, opt_actions
```

game()

plot_graph(results,exp)

This function is able to plot results of experiment on the graph. I use some lists to store the name or information of graph to decrease the number of lines pf code. This function will save two graph (exp_rewards.png and exp_opt_action.png) after running. Firstly, it will draw the graph of average rewards, using **plt.plot()** function to draw three lists of data(epsilon=0.0, 0.01, 0.1) on the graph with label. Then, it use **plt.xlabel()** and **plt.ylabel()** to show the x-axis and y-axis name , show grid of graph, and save graph with specific name. Secondly, it will draw the graph of average optimal selection by same progress.

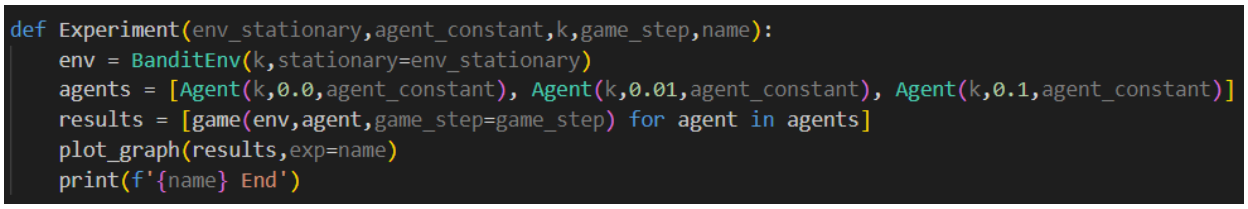


```python
def plot_graph(results,exp):
    labels = ['Epsilon = 0.0','Epsilon = 0.01','Epsilon = 0.1']
    x_label_name = 'times'
    y_label_names = ['Average Reward','Percentage of Optimal Action Selection']
    figure_names = [exp+'_rewards.png',exp+'_opt_action.png']
    for i in range(2):
        fig = plt.figure()
        plt.plot(results[0][i],'b-',label=labels[0])
        plt.plot(results[1][i],'r-',label=labels[1])
        plt.plot(results[2][i],'g-',label=labels[2])
        plt.xlabel(x_label_name)
        plt.ylabel(y_label_names[i])
        plt.legend()
        plt.grid(True)
        plt.savefig(figure_names[i])
```

plot_graph()

Experiment(env_stationary,agent_constant,k,game_step,name)

This function creates a environment according to input **k** and **env_stationary**. Then, it creates three agents with different epsilon(0.0, 0.01, 0.1) and decide the game step by input **agent_constant**. After creating environment and list of agents, this function will run **game()** function for every agent and environment and put the result into a list **results**. Finally, using **plot_graph()** to plot the **results** on the graph.

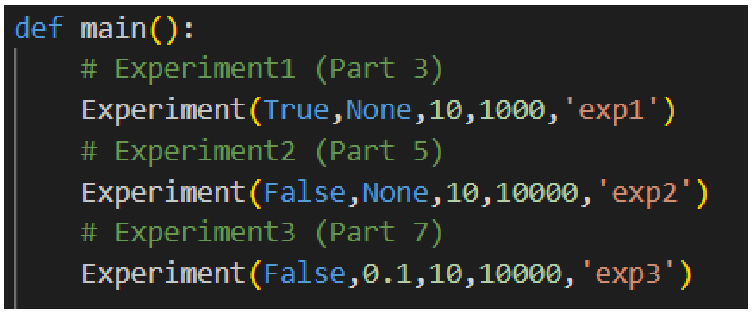


```python
def Experiment(env_stationary,agent_constant,k,game_step,name):
    env = BanditEnv(k,stationary=env_stationary)
    agents = [Agent(k,0.0,agent_constant), Agent(k,0.01,agent_constant), Agent(k,0.1,agent_constant)]
    results = [game(env,agent,game_step=game_step) for agent in agents]
    plot_graph(results,exp=name)
    print(f'{name} End')
```

Experiment()

main()

This function is the main function to run experiments.

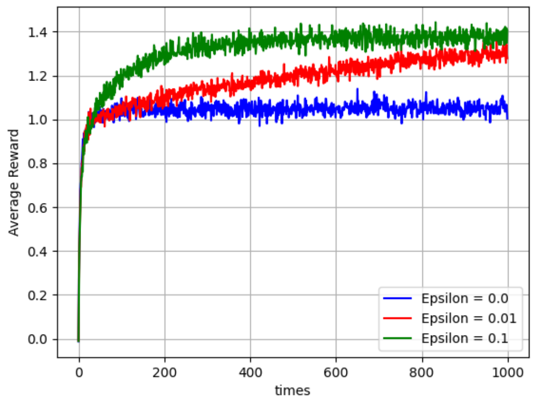


```python
def main():
    # Experiment1 (Part 3)
    Experiment(True,None,10,1000,'exp1')
    # Experiment2 (Part 5)
    Experiment(False,None,10,10000,'exp2')
    # Experiment3 (Part 7)
    Experiment(False,0.1,10,10000,'exp3')
```

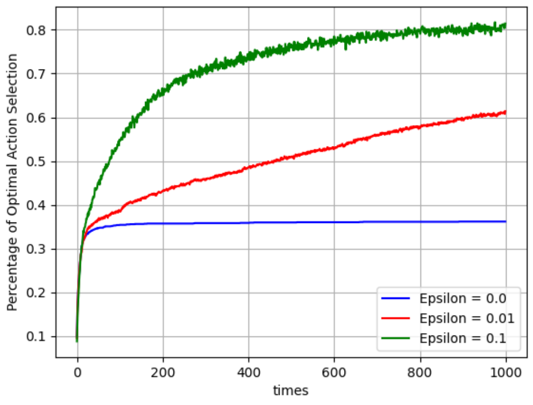
main()

2 Experiment

2.1 Experiment 1 (Part 3)



Average Reward over Time



Percentage of Optimal Action Selection

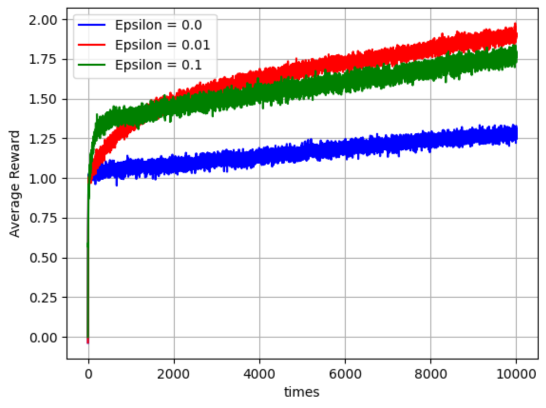
Result of Experiment 1 (Part 3)

These two picture are the result of experiment in part 3. Green line is represent the result of agent with epsilon 0.1, and red line is with epsilon 0.01, and blue line is with epsilon 0. The x-label is time in every single game, y-label is the average of reward and percentage of optimal action selection.

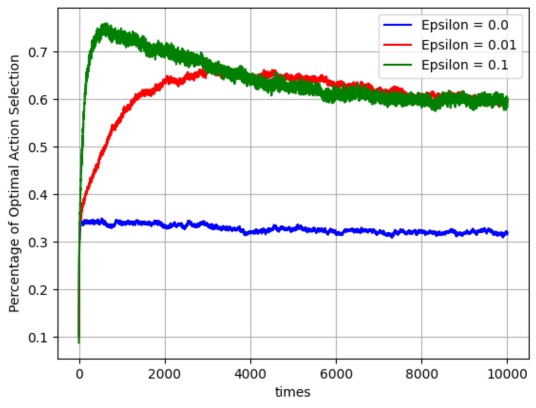
According these two graph, we can see that the percentage of optimal action selection of agent with epsilon 0.01 and 0.1 is higher over time, and the percentage of agent with epsilon 0.1 is significantly higher than agent with epsilon 0.01. The average reward of agent with epsilon 0.1 have significant increasing rate in early time, however it will stay at a moment, and average reward of agent with epsilon 0.01 increase slowly.

I think the reason to cause this result is that the exploration motion is important for this game, so the agent with higher probability of exploration will have great performance in early game.

2.2 Experiment 2 (Part 5)



Average Reward over Time



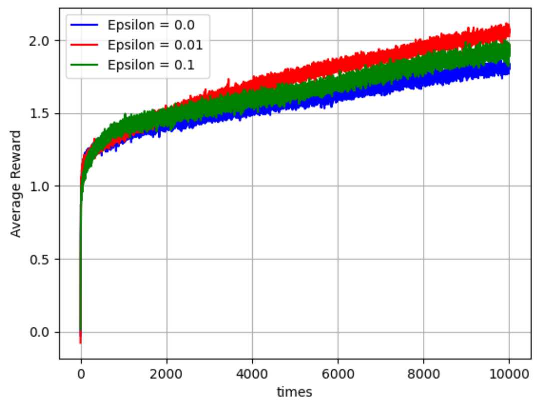
Percentage of Optimal Action Selection

Result of Experiment 2 (Part 5)

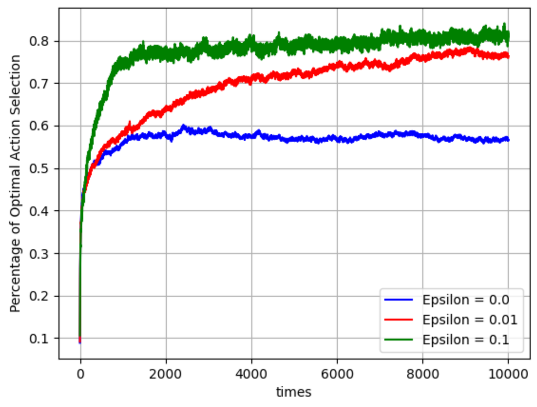
In this experiment, I observe that the agent with epsilon 0.1 will have bad performance in this experiment. According to right graph, we can see that the percentage of optimal selection is decrease before pulling arms 2000 times, and the agent with epsilon 0.01 is also decrease with smaller rate. By observing right graph, I know that the average reward of all agent will increase over time, but the agent with epsilon 0.1 has smaller reward than with epsilon 0.01.

The environment in this scenario is non-stationary, meaning the optimal action may change over time. The agent, however, makes exploitation decisions based on the estimated expected rewards, which are calculated from historical data. This motion will cause that the historical data in agent is not match to current environment, causing some bad affect on agent choice. Additionally, the agent with epsilon 0.1 will tend to explore environment more frequently than agent with epsilon 0.01, so agent with epsilon 0.1 has more data in history. This agent is more heavily influenced by outdated or irrelevant data. Therefore, the agent with epsilon 0.01 has higher performance in this part, and its percentage of optimal selection will start to decreasing later than agent with epsilon 0.1 .

2.3 Experiment 3 (Part 7)



Average Reward over Time



Percentage of Optimal Action Selection

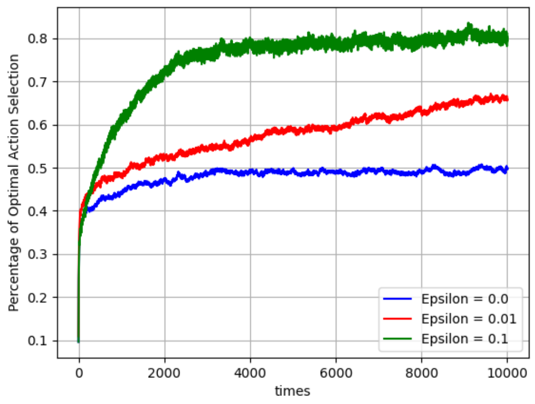
Result of Experiment 3 (Part 7)

In this part, the percentage of optimal selection of all agents will not strongly decreasing. The agent in this part will use step size to calculate q values instead of original way. This way may avoid the affect of odd data such that agent will response correctly. Therefore, the curve in the right graph is not decreasing in experiment, and keep staying after a specific time.

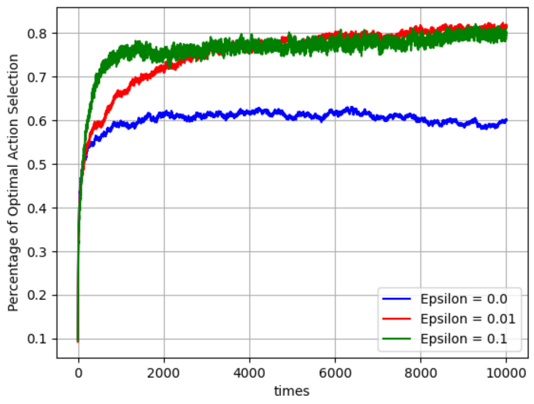
Comparing agent with epsilon 0.1 and 0.01, another observation of the result is that the average reward of agent with epsilon 0.01 is higher than agent with epsilon 0.1. However, the percentage of optimal selection of agent with epsilon 0.1 is higher than the other. I think this result is represent that the agent with higher epsilon will tend to do exploration action in experiment, so they will choose more optimal selection. However, the agent with epsilon 0.01 will spend less time in exploration, so it can choose more best action which it expected. Therefore, the agent with epsilon 0.01 have higher average reward than agent with epsilon 0.1.

3 Discussion

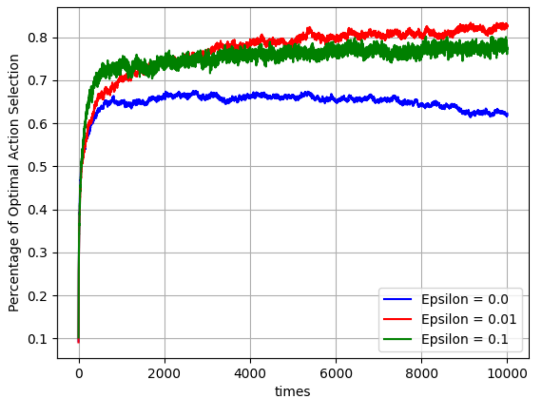
In previous part, I know that use step-size updating can solve the non-stationary environment. Now, in this part, I want to discuss how step-size affect agent select. Therefore, I run more experiments with changing step-size.



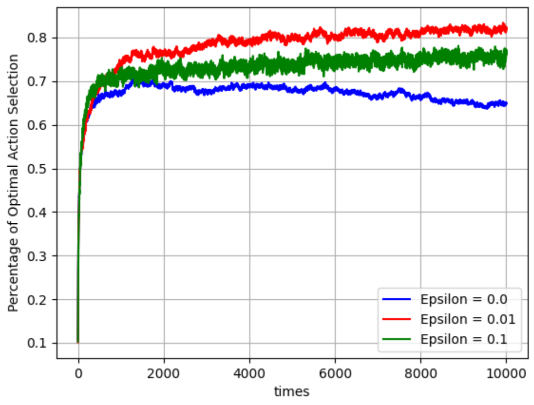
step size = 0.05



step size = 0.15



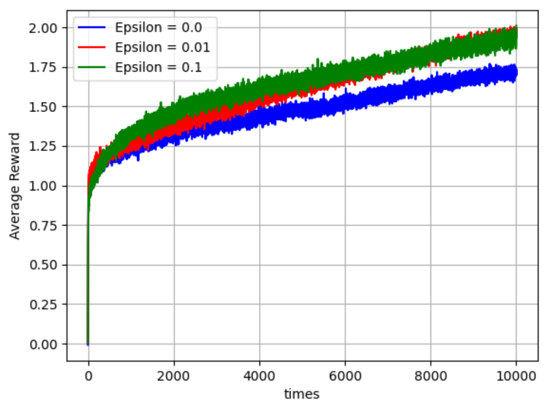
step size = 0.2



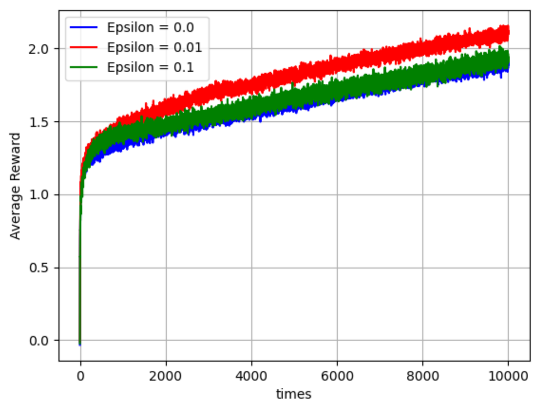
step size = 0.25

Percentage of optimal selection

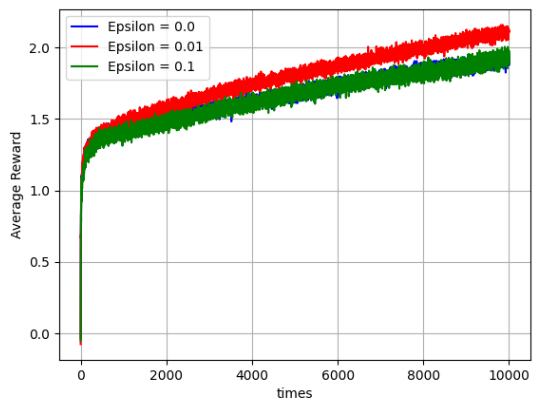
An agent with a larger step size may quickly reach a steady state, where its behavior no longer changes significantly over time. Additionally, the percentage of optimal action selection for the agent with epsilon 0.1 tends to decrease as the step size increases. When the step size is 0.15 or 0.2, the agent with epsilon 0.1 selects the optimal action at approximately the same rate as the agent with epsilon 0.01. However, when the step size increases to 0.25, the agent with epsilon 0.1 exhibits a lower percentage of optimal action selection compared to the agent with epsilon 0.01.



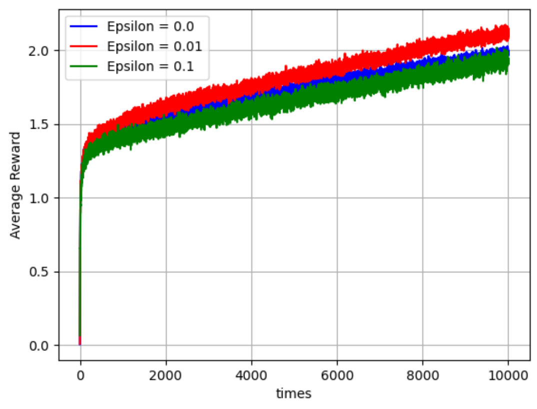
step size = 0.05



step size = 0.15



step size = 0.2



step size = 0.25

Average Rewards

In this part, average rewards of agent with epsilon 0.1 is higher than agent with epsilon 0.01 in step size 0.05 only. In other experiments, the average rewards of agent with epsilon 0.1 is lower than the other.

Based on the experimental results, the agent with a larger epsilon performs better when the step size is small. However, as the step size increases, the agent with a smaller epsilon shows better performance. I think this phenomenon occurs because the step size directly affects the learning speed of the agent. A larger step size allows the agent to learn more quickly, while a smaller step size leads to slower, more gradual learning. In the case of a small step size, the agent requires more exploration to effectively learn about the environment, making a higher epsilon (i.e., more exploration) beneficial. Conversely, with a large step size, excessive exploration may introduce instability, and a smaller epsilon (i.e., more exploitation) becomes more advantageous.