- **Implementation Details**

Agent: self.k: 存總共有多少種 action(對應拉哪台 arm)。self.e: 出去探索未知選項的機率。self.alpha: 給定的學習率(默認為 None)。self.q_v: 學習到的 reward 報酬量。self.a_c: 紀錄同個 action 執行過多少次。reset:要清空學習內容，所以將 self.q_v 跟 self.a_c 清空，其他是學習規則不用清除。update_q: 有兩種狀況，self.alpha 有輸入值:a=輸入值。沒有輸入值: a=sample-average(使用次數分之 1)。然後用執行 action 後得到的 reward，減去目前該 action 的報酬量，再乘與 a(學習幅度)，更新 action 的報酬量。select_action: 隨機 0-1 的一個小數，如果該小數<=self.e，就會探索隨機 action。否則給出目前學習到 reward 最高的 action。例 self.e=0.1 時，有 10%會去隨機 action。

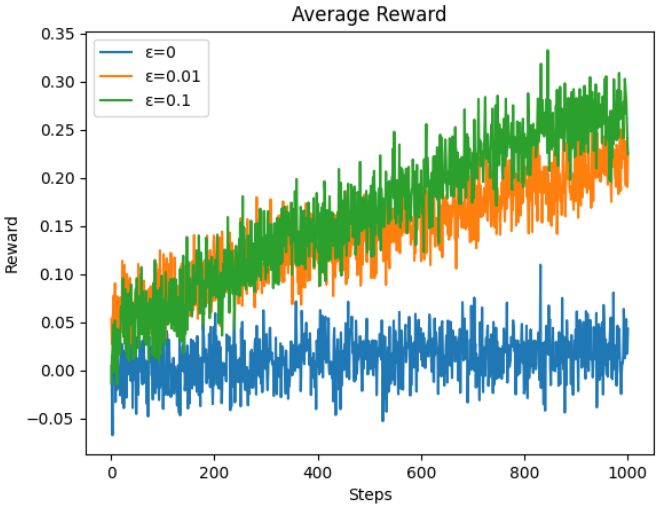
```python
import numpy as np

class Agent():
    代码解释 | 函数注释 | 调优建议 | 行间注释
    def __init__(self, k, epsilon, alpha = None):
        self.k = k
        self.e = epsilon
        self.alpha = alpha
        self.q_v = np.zeros(self.k)
        self.a_c = np.zeros(self.k)

    代码解释 | 函数注释 | 调优建议 | 行间注释 | 生成单测
    def reset(self):
        self.q_values = np.zeros(self.k)
        self.action_count = np.zeros(self.k)

    代码解释 | 函数注释 | 调优建议 | 行间注释 | 生成单测
    def  update_q(self, action, reward):
        self.a_c[action] += 1
        if self.alpha == None:
            a = 1/self.a_c[action]
        else:
            a = self.alpha

        self.q_v[action] += a* (reward - self.q_v[action])

    代码解释 | 函数注释 | 调优建议 | 行间注释 | 生成单测
    def select_action(self):
        if np.random.rand() <= self.e:
            return np.random.randint(self.k)
        else:
            return np.argmax(self.q_v)
```

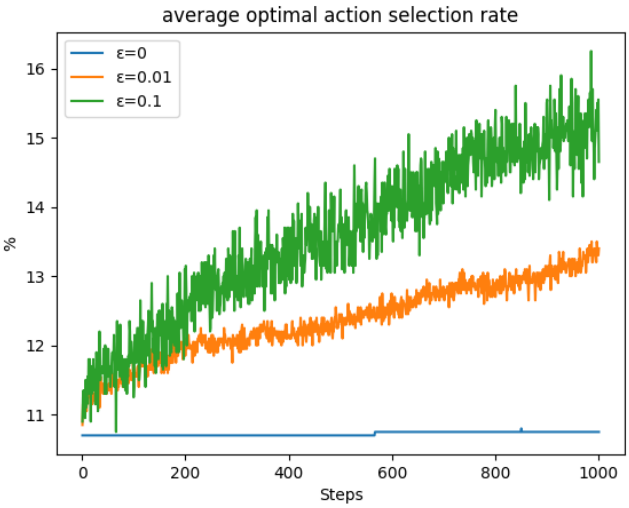
Environment: self.k: 存有幾台 arm。self.var: arm reward 的變異數(設定為 1)。self.s: 環境的穩定性，True 狀態 arm reward 的平均不會變，反之會變化(預設是 True)。self.means: arm reward 的平均，數值為題目要求 standard normal distribution。使用 numpy 庫的 random.standard_normal 取 k 個數值。self.action_history: 存取執行過的 action。self.reward_history: 存取對應 action 得到的 reward。reset: 重設 mean 的數值，清空 action、reward 兩個歷史紀錄。step: 根據 action 回傳 reward。當 self.s 是 False，也就是環境不穩定時，mean 會加上隨機值。依據題目要求，我使用 numpy 庫的 random.normal，mean=0、var=0.01。取得 action 的 reward，random.normal，mean=self.means 中對應 action 的 mean、var 是固定的 1。然後將 action 跟 reward 都塞進紀錄，再回傳 reward 值。export_history: 回傳 action、reward 歷史紀錄。

```python
import numpy as np

class BanditEnv():
    代码解释 | 函数注释 | 调优建议 | 行间注释
    def __init__(self, k, stationary = True):
        self.k = k
        self.var = 1
        self.s = stationary
        self.means = np.random.standard_normal(size=self.k)
        self.action_history = []
        self.reward_history = []

    代码解释 | 函数注释 | 调优建议 | 行间注释 | 生成单测
    def reset(self):
        self.means = np.random.standard_normal(size=self.k)
        self.action_history = []
        self.reward_history = []

    代码解释 | 函数注释 | 调优建议 | 行间注释 | 生成单测
    def step(self, action):
        if not self.s:
            walks = np.random.normal(0, 0.01, size=self.k)
            self.means += walks
        if 0 <= action < self.k:
            r = np.random.normal(self.means[action], self.var)
            self.action_history.append(action)
            self.reward_history.append(r)
            return r

    代码解释 | 函数注释 | 调优建议 | 行间注释 | 生成单测
    def export_history(self):
        return self.action_history, self.reward_history
```

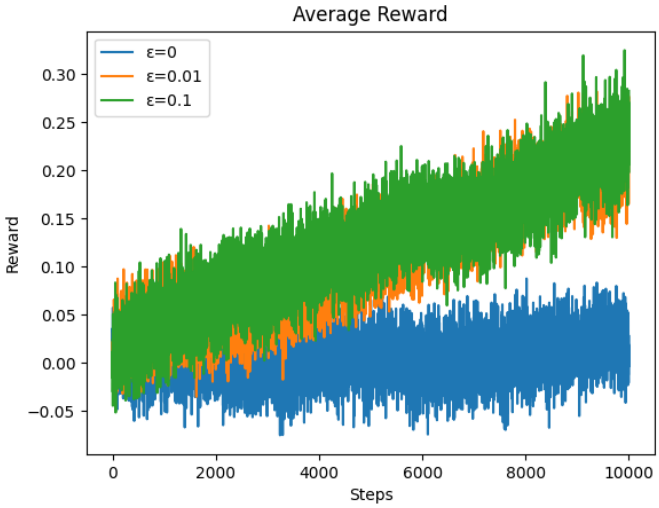
- **Experiment Results**

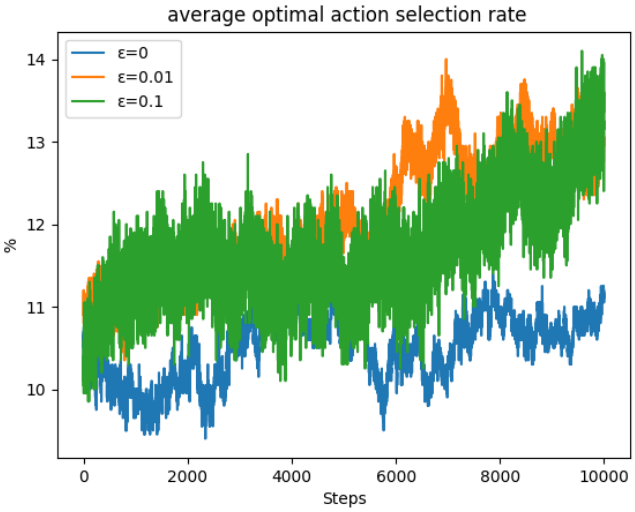
Part3: 因為有隨機性，所以會有局部起伏。但從整體的趨向可以看出不論是 reward 或 optimal action rate 都是 ε=0.1 學習效果最佳。ε=0 是沒有學習，因為它 optimal action rate 是幾乎沒有變化，所以它沒有跟隨 step 而進行學習行為。



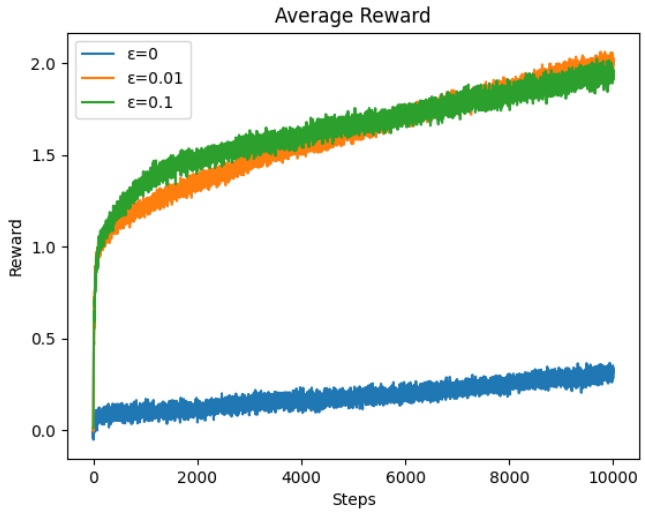


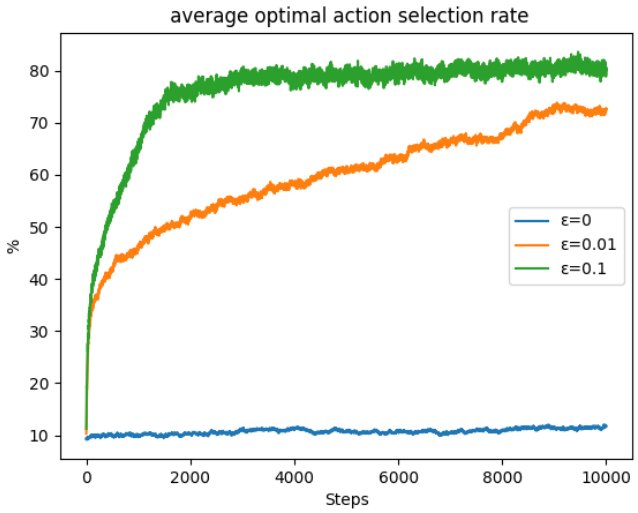
Part5: 因為具有跟隨時間而變化的環境，隨機性更強了，所以局部起伏比起 part3 更劇烈。ε=0 依舊是學習狀態最差的，但是因為不同 arm reward means 跟隨時間變化，所以 optimal action rate 相較 part3 更有起伏變化。而 ε=0.1 及 0.01 看起來差距不大，有可能是局部浮動導致難以判斷。但是從 optimal action rate 可以看出 ε=0.1 因為探索性更大，所以起伏劇烈程度也更大。看起來 ε=0.01 有一部分選擇 ε=0.1 要好一些，但是因為初始選擇不好，所以 reward 部分差異不大。





Part7: ε=0 依舊是學習狀態最差的，另外兩者學習效果得到大幅增益。Reward 能看到 ε=0.1 在初期一直是最佳，但是在大約 7000 step 被 ε=0.01 反超。Optimal action rate 能看到 ε=0.1 在大約 2000 step 已經趨於平衡，在 80%上下起伏。而 ε=0.01 還繼續在慢慢提升。前面 reward 被反超，可能是因為 ε=0.1 已經飽和。在擁有最佳選擇的狀態，還是有 10%機率去探索到非最佳選擇。





- **Discussion**

在 part7 能看到 ε=0.1 在後期會因為在探索到最佳選擇時，因為較大探索率而去選到非最佳選擇。epsilon 遞減應該是可以有效減少該情況發生，在 Select_action 的 if 成立時，讓 self.e 能夠緩減，且不要讓它小於 0。也許能達成前期多探索，後期多利用的狀態。