0. Introduction

name: 蔡懷恩
ID: 112550020

1. Implementation Detail

Part1. environment

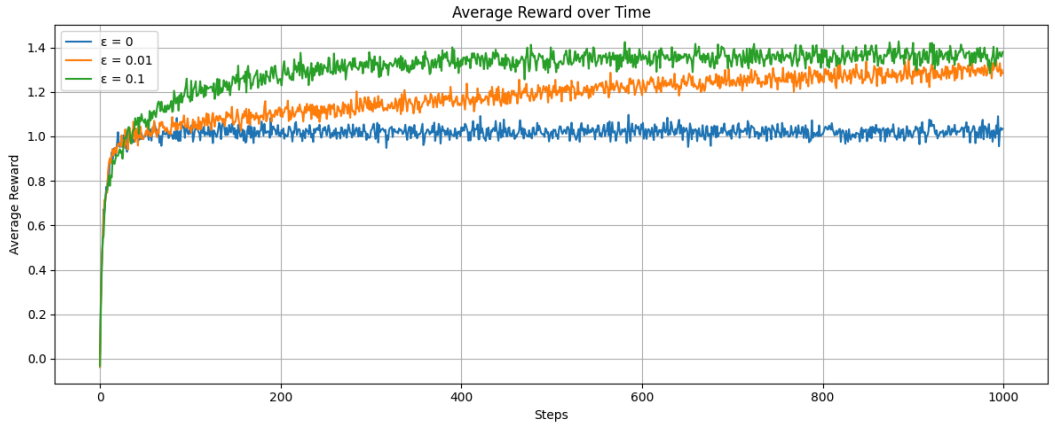
有把 part4 的部分一起加進去，part1 就把 stationary 設成 True

```python
class BanditEnv:
    def __init__(self, k: int, stationary=True): # (k臂拉霸機，是否為平穩環境)
        self.k = k
        self.stationary = stationary
        self.reset()

    def reset(self):
        # 重置每個 arm 的真實平均獎勵（mu），並清空歷史記錄
        self.q_true = np.random.normal(loc=0.0, scale=1.0, size=self.k) # q_true[a] 真實期望 reward，給定選擇動作 a
        self.action_history = []
        self.reward_history = []

    def step(self, action: int) -> float:
        # 防呆 避免選一個不存在的臂
        assert 0 <= action < self.k, f"Invalid action {action}, must be in [0, {self.k - 1}]"

        # 若為 non-stationary 環境，進行random walk
        if not self.stationary:
            self.q_true += np.random.normal(loc=0.0, scale=0.01, size=self.k)

        # 根據 q_true[action] + N(0, 1) 產生 reward
        reward = np.random.normal(loc=self.q_true[action], scale=1.0)

        # 記錄歷史
        self.action_history.append(action)
        self.reward_history.append(reward)

        return reward

    def export_history(self):
        return self.action_history, self.reward_history
```

Part2. agent

```python
class Agent:
    def __init__(self, k: int, epsilon: float):
        self.k = k
        self.epsilon = epsilon
        self.reset()

    def reset(self):
        # Q 值重置為0
        self.q_estimates = np.zeros(self.k)
        # 記錄每個動作被選過幾次
        self.action_counts = np.zeros(self.k, dtype=int)

    def select_action(self) -> int:
        if random.random() < self.epsilon: # 產生0~1的float
            # Exploration：隨機選擇一個動作
            return random.randint(0, self.k - 1)
        else:
            # Exploitation：選擇目前 Q 值最高的動作（若有多個並列會選最小index）
            return int(np.argmax(self.q_estimates))

    def update_q(self, action: int, reward: float):
        self.action_counts[action] += 1
        count = self.action_counts[action]

        # sample-average method: Qn = Qn-1 + (R-Qn-1)/n
        self.q_estimates[action] += (reward - self.q_estimates[action]) / count
```

Part3.



ε = 0.1：
最快達到高 reward，因為會找出最好的 arm 並持續 exploit

ε = 0.01：
慢慢追上，但因為探索少、收斂慢

ε = 0：
雖然初期 reward 跟其他差不多，但沒進步，最終 reward 停在相對低水準

ε = 0.1：
探索多，成長最快，最後穩定在 75%~80% 左右，早期就能發現哪個 arm 最好

ε = 0.01：
探索太少，學得慢，但最後還是持續上升，相對保守但可收斂型

ε = 0：完全不探索（純貪婪）
前幾步選到哪個 arm 就永遠相信它，不會修正
所以大概卡在 36% 左右（運氣決定對錯）

part4. environment update

如果是 non-stationary 就在初始化拉霸機時 將stationary設為False
並在每一步前面都把true mean reward加上隨機值

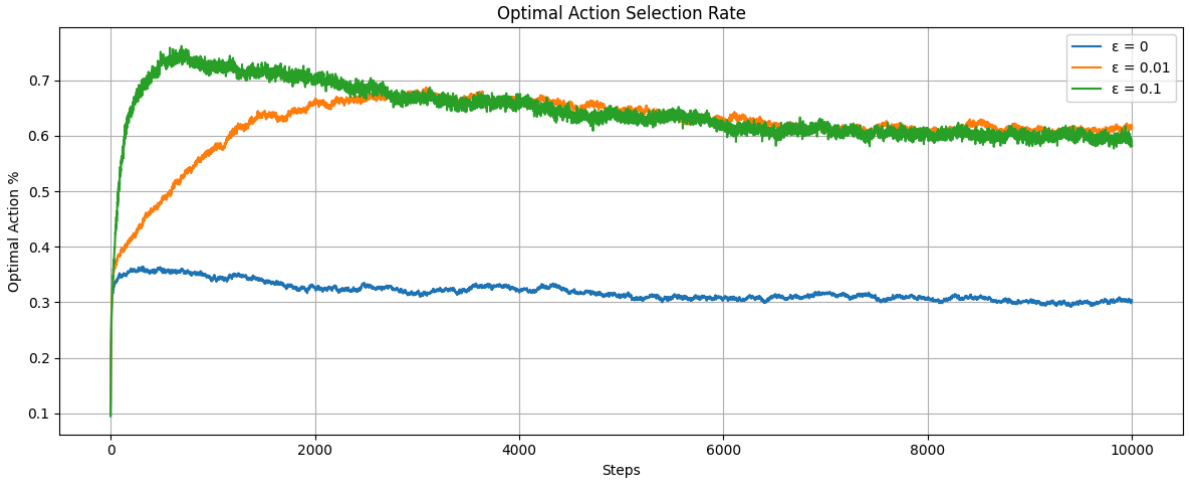
```
# 若為 non-stationary 環境，進行random walk
if not self.stationary:
    self.q_true += np.random.normal(loc=0.0, scale=0.01, size=self.k)
```

Part5.

ε = 0.1：
早期 reward 高，之後成長趨緩
能幫助early learning，但會因為常常探索所以反而得不到最佳reward

ε = 0.01：
慢慢追上，甚至超過 ε = 0.1

ε = 0：
持續卡在差的reward，純運氣



ε = 0.1：
一開始能快速選到最好，但中後期表現下降
因為sample average Q 收斂太慢，環境變了但 Q 沒跟上

ε = 0.01：
上升慢，但下降也慢，後面反而能維持住

ε = 0：
完全不探索，學不到環境變化，卡在 30幾%

Part6. Agent update

agent 的呼叫新增了 step_size (預設 None)
如果 step_size ≠ None，使用 constant step-size update method
如果 step_size = None，使用原本的 sample-average method

```
if self.step_size is not None: # constant step-size update method: Q_n = Q_n-1 + alpha * (R-Q_n-1)
    alpha = self.step_size
else: # sample-average method: Q_n = Q_n-1 + (R-Q_n-1)/n
    alpha = 1 / count

self.q_estimates[action] += alpha * (reward - self.q_estimates[action])
```

Part7.

ε = 0.1
起步最快
reward 隨著時間穩定上升，代表有在有效追蹤環境的變化

ε = 0.01
起步較 0.1 慢，但後期 reward 上升甚至超過 ε = 0.1
表示探索夠少，所以不太浪費已經找到的最佳 arm；
但 α = 0.1 又能跟上 μ 的變化 → 長期 reward 穩穩上升

ε = 0
起步不差，但也不至於和前面一樣完全不升高，我猜是會因為Q值變負的而換選擇，導致reward緩慢變高，所以相對於前面part5，會因為Q值難以變化而reward持續在差的地方

ε = 0.1
快速收斂到 80% 左右，
因為經常探索，更有機會發現reward高的 arm

ε = 0.01
上升速度較慢，但後期追上來，
因為探索少，但有固定 α，所以逐步找到對的 arm
但還是會比 0.1 的少一點

ε = 0
卡在 55%，相較於前面的幾乎不換arm，改用 constant step-size update method 之後多少會換一點arm，所以比前面的 35% 左右高，但 optimal action rate 不會升高，因為即使 reward 上升也不是選到最佳的

main function

```python
def run_experiment(epsilon, runs=2000, steps=10000, Stationary=False, Step_size=0.1): # 變數: epsilon, runs 執行次數, steps 幾步
    k = 10                                                                             #      Stationary 是否為平穩環境, Step_size = alpha
    rewards = np.zeros((runs, steps))
    optimal_actions = np.zeros((runs, steps))

    for run in range(runs):
        env = BanditEnv(k, stationary=Stationary)
        agent = Agent(k, epsilon, step_size=Step_size)

        for step in range(steps):
            action = agent.select_action()
            reward = env.step(action)
            agent.update_q(action, reward)

            rewards[run, step] = reward
            if action == np.argmax(env.q_true):
                optimal_actions[run, step] = 1

    avg_rewards = rewards.mean(axis=0) # avg_rewards[t]：第 t 步的 reward 平均值（所有 run）
    optimal_action_rate = optimal_actions.mean(axis=0)

    return avg_rewards, optimal_action_rate
```

利用一開始的變數去滿足不同 part 的要求
part3:
(epsilon, runs=2000, steps=1000, Stationary=True, Step_size=None)
part5:
(epsilon, runs=2000, steps=10000, Stationary=False, Step_size=None)

part7:
(epsilon, runs=2000, steps=10000, Stationary=False, Step_size=0.1)

```
epsilons = [0, 0.01, 0.1]
results = {}

for eps in epsilons:
    rewards, optimal_rate = run_experiment(eps)
    results[eps] = (rewards, optimal_rate)

# Plot 平均獎勵
plt.figure(figsize=(12, 5))
for eps in epsilons:
    plt.plot(results[eps][0], label=f"ε = {eps}")
plt.title("Average Reward over Time")
plt.xlabel("Steps")
plt.ylabel("Average Reward")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("reward_plot.png")

# Plot 最佳動作選擇率
plt.figure(figsize=(12, 5))
for eps in epsilons:
    plt.plot(results[eps][1], label=f"ε = {eps}")
plt.title("Optimal Action Selection Rate")
plt.xlabel("Steps")
plt.ylabel("Optimal Action %")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("optimal_action_plot.png")
```

接著跑不同 epsilons 紀錄結果，最後畫圖