HW4 Report

Jun-Kai Chen, 111550001

May 7, 2025

Implementation Details

Agent

Part 2

The constructor of the `Agent` class takes several parameters: the number of bandits `k`, and the probability of choosing to explore during action selection `epsilon`. It stores these values and initializes the number of times each bandit has been selected, as well as the estimated expected values. Since the agent selects the optimal action based on the current expected values, we must consider how the initialization of those values may impact learning. If a bandit's true expected value is higher than the initial estimate, the agent may mistakenly believe it has already found the optimal action from the first pull. This issue is generally mitigated through exploration, but since the assignment requires testing with ϵ set to 0, such edge cases may lead to suboptimal outcomes. Therefore, the expected values are initialized to 0 to match the assumption that the true rewards are also centered around 0.

In the `select_action()` function, the program uses the random module to generate a value between 0 and 1. If this value is smaller than the given ϵ, the agent explores by randomly choosing an index between 0 and k-1. Otherwise, it exploits by selecting the index with the highest current expected value.

The `update_q()` function takes two arguments: the chosen bandit a and the received reward R (corresponding to the `action` and `reward` parameters in the code). The agent increments the count for bandit a and updates its estimated value based on the following formula:

$$Q(a) \leftarrow Q(a) + (R - Q(a))/N(a)$$

The last function is `reset()`, which simply resets all estimated values and selection counts for each bandit.

Part 5

In Part 5, the agent is updated to accept an optional parameter `alpha`, which represents the learning rate. If the Agent instance has an `alpha` field, the update to expected values follows this formula instead:

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

Generally, after multiple iterations, the *alpha* value is usually larger than $\frac{1}{N(a)}$, which means recent learning experiences will have a greater influence on the current estimate.

Environment

Part 1

The constructor of `BanditEnv` accepts a parameter k, representing the number of bandits to create. It initializes the true expected values for each bandit using a normal distribution with mean 0 and standard deviation 1, as described in the assignment. It also initializes two lists to record the selected actions (`steps`) and received rewards (`rewards`).

The `reset()` function re-generates the true expected values for all bandits and clears both the `steps` and `rewards` lists.

The `step()` function takes an argument a as the selected action. It first checks whether a is valid (i.e. a valid bandit index). Then it samples a reward based on a Gaussian distribution with the bandit's mean and standard deviation of 1, and returns the result.

The `export_history()` function returns the steps and rewards lists. A strange thing is that it seems this function won't be used in general situations. Previous functions could exactly handle the works.

In Part 3, one experimental requirement was to track whether the agent obtained the optimal action. However, neither the agent nor the main function can directly know the actual best action of the `BanditEnv` object. Therefore, a new function `get_optimal_action`() was added, which scans the current expected values and returns the index of the highest one.

Part 4

In Part 4, `BanditEnv` is modified to accept an optional parameter `stationary`, which indicates whether the environment is stationary. This affects two parts of the implementation.

First is the `step()` function. According to the assignment, when the environment is non-stationary, after each call to `step()`, the true expected values of all bandits should be adjusted by adding a small noise sampled from a normal distribution with mean 0 and standard deviation 0.01.

The second part is the `get_optimal_action()` function. In a stationary setting, the optimal action is fixed after either invoking constructor or `reset()`. However, in the non-stationary case, this function must be called before each step to determine the current best action.

Experiment Results

Part 3

The average reward per step and the frequency of choosing the optimal action are shown in Figures 1 and 2, respectively.

From Figure 1, we can see that with $\epsilon \neq 0$, the average reward approaches the theoretical maximum (approximately 1.4 in this case). Moreover, higher ϵ values lead to faster convergence. In contrast, when ϵ is 0 (i.e. no exploration), the performance is notably worse. This is likely because the agent prematurely settles on a suboptimal action that happens to yield a decent reward early on, while never exploring potentially better options.

Figure 2 shows that with exploration ($\epsilon > 0$), the agent eventually discovers the true optimal action. $\epsilon = 0.1$ performs the best in this experiment. However, based on the trends, both $\epsilon = 0.1$ and 0.01 appear to converge toward around 80% optimal action selection if more steps were allowed.

Figure 1: The average reward of time.

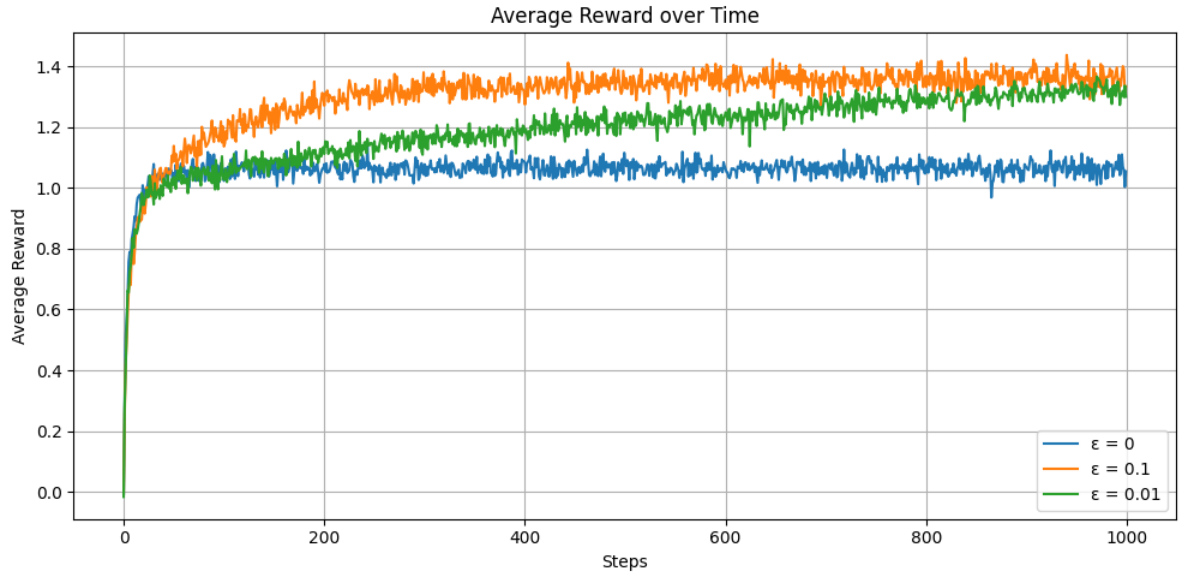


Figure 2: The percentage of optimal action of time.

Part 5

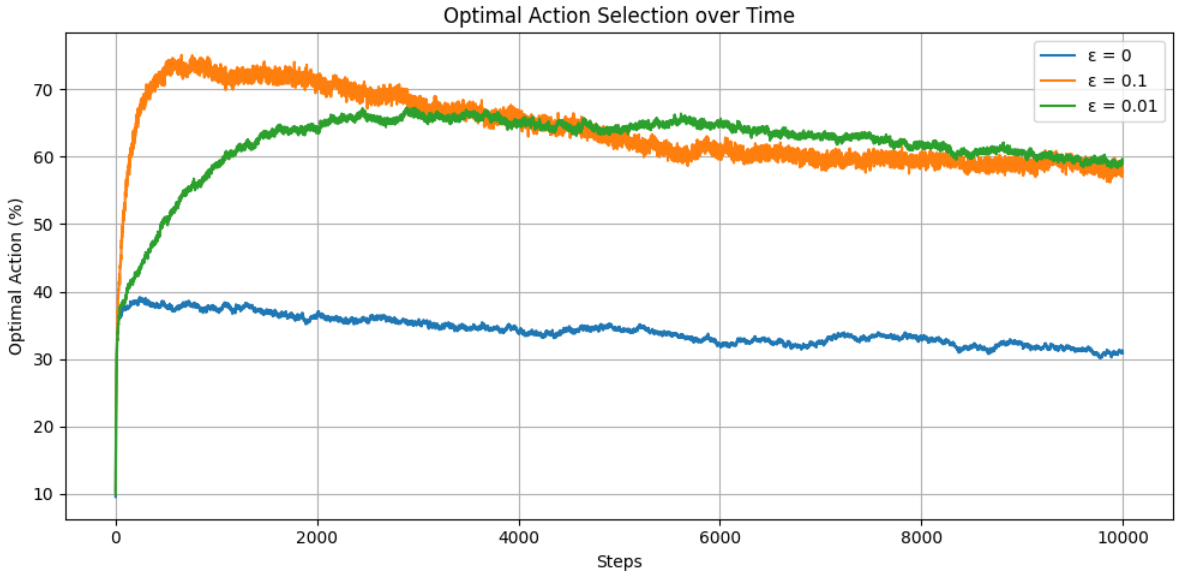
The average reward per step and the probability of choosing the optimal action are shown in Figures 3 and 4, respectively.

From Figure 3, we can see that in a stationary environment, the reward increases over time regardless of the ϵ value, and convergence has not yet been observed. Additionally, after step 2000, the agent with $\epsilon = 0.01$ receives better rewards. This may indicate that a more conservative agent performs better in a stationary environment.

From Figure 4, we see that regardless of the ϵ value, the probability of the agent selecting optimal actions decreases as the number of steps increases. Together with the information in Figure 3, we can infer that even though the reward increases over time, his agent does not adapt well to a stationary environment.

Figure 3: The average reward of time in stationary environment.

Figure 4: The percentage of optimal action of time in stationary environment.



Part 7

The average reward per step and the probability of selecting the optimal action are shown in Figure 5 and 6, respectively.

Starting with Figure 6: unlike what we observed in Figure 4, the proportion of optimal action selection does not decrease. This shows that an agent with a constant step-size can adapt to a non-stationary environment. Moreover, a larger ϵ leads to faster convergence and achieves a higher converged value than the maximum value seen in Figure 4. I think with more training steps, the *epsilon* $= 0.01$ agent should also converge to a similar value. On the other hand, a greedy agent performs significantly worse than agents with exploration capability.

Comparing Figure 3 and Figure 5, there is a noticeable improvement in the agent's reward. This implies that as the optimal action rate increases, the rewards also improve accordingly. Another noteworthy point is that, unlike the sample average method, agents using a constant step-size can perform well even when $\epsilon = 0$.

Figure 5: The average reward of time in stationary environment with constatn step-size.

Figure 6: The percentage of optimal action of time in stationary environment with constatn step-size.