Multi-armed Bandit

CAP 6629 Reinforcement Learning Project 1

Fanchen Bao

1 Introduction

Multi-armed bandit is one of the most fundamental problems in reinforcement learning [1]. In its simplest term, multi-armed bandit describes a problem where given multiple actions to choose from at any time step, which one should be taken to maximize rewards after a certain number of time steps. The simplest multi-armed bandit requires that the reward associated with each action is stationary throughout all time steps, i.e., the rule of reward for each action does not change as time step progresses. It is worth emphasizing that while the rule of reward remains stationary, the reward granted itself can be stochastic. For instance, the rule could be that an action's reward follows a stationary Gaussian Distribution. Although the Gaussian Distribution stays the same, the actual reward granted is a random variable from the distribution. Another restriction common in multi-armed bandit is that knowledge of the rule of reward is typically unavailable at the start of the problem.

To solve a multi-armed bandit problem, or in other words, to establish a strategy that maximizes total rewards after certain time steps, we must first obtain the true reward of each action. However, the true reward is unknown to us. Therefore, we need to estimate it. And a simple way to make the estimation is by sampling each action a lot of times. The expected reward (i.e. average of all the rewards sampled) thus acquired will converge to the true reward if the number of sampling is sufficiently large. Despite its simplicity, this idea is the backbone of all reinforcement learning methods.

Although it is theoretically possible to sample all actions sufficient number of times to obtain their converged expected rewards, in reality, this usually requires too much computation. Furthermore, we would also like to formulate the best strategy at the same time as we are sampling. Yet, the theoretical method does not support this; it always samples first and then constructs the best strategy based on the sampling outcome. This is not efficient, because sampling without a strategy, i.e. completely random, wastes time on actions that are known to be bad from prior experience. A better sampling process would favor good actions more often such that the rewards can be maximized faster.

Such better sampling process requires a balance between exploitation and exploration. Exploitation describes a rule that chooses an action greedily. To be specific, given expected rewards generated from the previous $t-1$ time steps of all actions, exploitation at the current time step t always chooses the action that has the highest expected reward. Exploration, on the other hand, chooses an action randomly, with no regard to the expected rewards. The theoretical method discussed above performs sampling completely based on exploration. It would only achieves sub-optimal total rewards because it disregards the information currently available, i.e. the expected rewards so far, and spends too much time on actions proven to be bad choices. On the other extreme, however, a sampling process based purely on exploitation also would yield sub-optimal rewards, because it can easily get stuck in a sub-optimal action. Therefore, a good sampling process must balance both exploitation and exploration: exploit most of the times to ensure that the expected rewards gleaned thus far are put into good use, but also explore from time to time to search the action space for any better options. This report aims to shed some light on how different exploitation-vs-exploration combinations affect the total and average rewards achieved in a multi-armed bandit problem.

This report is organized in the following manner. Section 2 introduces mathematical notations and formulas that model the multi-armed bandit. Pseudo-code is also presented to describe how the multi-armed bandit problem is approached algorithmically. Section 3 describes a generic test bed for a 10-armed bandit with different exploitation-vs-exploration combinations, and reveals the best combination based on average rewards. Section 4 presents another take on the multi-armed bandit using a dataset about click distribution over a variety of advertisements (ads), and shows how different exploitation-vs-exploration combinations affect the total rewards from ads. Finally, section 5 provides insight into the experimentation and concludes the report.

2 Formal Definition

Given a multi-armed bandit and let A be all possible actions, we can denote a as one of the actions, i.e. $a \in A$. We use $Q_t(a)$ to represent the estimated reward of a after t time steps. We use $R_1, R_2, ..., R_{N_t(a)}$ to represent the immediate reward granted by action a each time it is taken ($N_t(a)$ is the number of times action a is taken throughout the t time steps). Then $Q_t(a)$ can be expressed as Equation (1), where $k = N_t(a)$.

$$Q_t(a) = \frac{R_1 + R_2 + ... + R_k}{N_t(a)} \tag{1}$$

Two things are worth noticing in Equation (1). First, the rewards from action a are not necessarily the same each time the action is taken. This is because these rewards are variables extracted from a certain rule of reward, e.g. a statistical distribution, defined by action a. Making the reward non-stationary but conforming to a statistical distribution is a good approximation of what reward in real life looks like. That said, it is also perfectly fine setting all the rewards the same across all executions of action a. However, doing this is not beneficial, because when the reward distribution is deterministic, exploitation will always win over exploration. Hence, in order to examine the exploitation-vs-exploration combination, it is important to ensure that rewards from any action are stochastic within a defined distribution.

Second, computation of $Q_t(a)$ is simply the arithmetic mean of all the rewards granted by action a throughout t time steps. According to the law of large numbers, as the number of times action a is sampled increases, $Q_t(a)$ would eventually converge to the true reward, denoted as $q^*(a)$:

$$q^*(a) = \lim_{N_t(a) \to \infty} Q_t(a) \tag{2}$$

To efficiently compute $Q_t(a)$, we can use the incremental implementation as shown in Equation (3), where $k = N_t(a)$. It updates $Q_t(a)$ step by step, instead of waiting until all sampling is over to compute the average.

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N_t(a)}[R_k - Q_t(a)] \tag{3}$$

Note that Equation (3) is just an iterative version of Equation (1). The two expressions are essentially the same. Yet, it is much easier to construct an algorithm based on Equation (3) due to its iterative nature.

To examine the effect of different exploitation-vs-exploration combinations, we define exploitation (i.e. greedy) at time step t as choosing the action with the highest $Q_t(a)$ among all actions. This is expressed in Equation (4), where A_t is the exploited action that maximizes $Q_t(a)$.

$$A_t = \arg\max_a Q_t(a) \tag{4}$$

We define exploration at time step t as choosing a random action from all actions, regardless of the value of $Q_t(a)$.

Finally, the exploitation-vs-exploration combination is represented by ϵ, which denotes the probability that an action is chosen via exploration. Naturally, the probability of exploitation is denoted by $1-\epsilon$. If $\epsilon = 1$, action choice is completely random, whereas if $\epsilon = 0$, action choice is completely greedy. This method of combining exploitation and exploration is called $\epsilon - greedy$.

3 10-Armed Bandit Test Bed

We set up the test bed by creating 1,000 10-armed bandits. Each 10-armed bandit contains ten actions, with the true reward of each action randomly generated from a Gaussian Distribution $\mathcal{N}(0, 1)$. Each 10-armed bandit also contains storage capacity for the expected reward of each action and the number of times each action has been sampled.

At each time step, an action is selected in each of the bandits. To determine whether action selection is via exploitation or exploration, a random number is generated from Uniform Distribution $\mathcal{U}(0, 1)$ and compared to a predetermined value of ϵ. If the random number is smaller than ϵ, the action will be selected via exploration. Otherwise, the action will be chosen via exploitation.

After each time step, each bandit performs internal housekeeping, which includes incrementing the number of times the current action is sampled by one and updating its expected reward updated according to Equation (3). Externally, the average reward of the current time step is calculated as the mean of the current rewards of all bandits. In addition, the percentage of optimal action, i.e. the percentage of bandits that have taken the optimal action, at the current time step is also computed.

The test bed runs for 5,000 time steps. As described above, each time step receives two data point: average reward and percentage of optimal action. These data points are used to plot curves showing how average reward and percentage of optimal action change across all time steps.

In order to examine the effect of different exploitation-vs-exploration combinations, the test bed will be run under different ϵ values. A good ϵ value results in average reward and percentage of optimal action curves that have high plateau and plateau fast. In this report, we supply seven ϵ values $[0, 0.001, 0.01, 0.1, 0.3, 0.6, 1]$, one for each test bed. The rationale of using these values is to cover a wide range of exploitation-vs-exploration combination.

3.1 Pseudo-code

The pseudo-code for the multi-armed bandit algorithm is shown in Algorithm 1. It describes the procedure of the entire test bed, including how action and reward are acquired for each bandit and how the average reward and percentage of optimal action are computed for each time step.

3.2 Evaluation

Figure 1 shows the reward distribution of each action in a sample 10-armed bandit. Each reward distribution follows Gaussian Distribution centered at the true reward, which itself is randomly generated via $\mathcal{N}(0, 1)$, with standard deviation 1.

Figure 2 demonstrates the performance of 10-armed bandit throughout 5,000 time steps and over different ϵ values. The top plot is the average reward, with x-axis being the time step and y-axis the average reward. The bottom plot is the percentage of optimal action, with x-axis being the time step and y-axis the percentage that the optimal action is taken across all the bandits.

It is clear from the top plot of Figure 2 that different ϵ values result in different performance. $\epsilon = 1$ performs the worst, followed by $\epsilon = 0.6$. This means relying too heavily on exploration yields very poor performance. As ϵ gets smaller, the performance peaks at $\epsilon = 0.1$ and $\epsilon = 0.01$,

Algorithm 1 10-armed Bandit

```
1: INITIALIZE
2: $A \leftarrow 10$    ▷ Number of arms in the bandit
3: $M \leftarrow 1000$    ▷ Number of multi-armed bandits
4: $T \leftarrow 5000$    ▷ Number of time steps
5: $\epsilon \leftarrow$ probability for exploration
6: $Q_a \leftarrow$ A matrix of size (M, A)    ▷ Expected reward of all actions over all bandits
7: $q_a^* \leftarrow$ A matrix of size (M, A)    ▷ True reward of all actions over all bandits
8: $N_a \leftarrow$ A matrix of size (M, A)    ▷ Number of times an action is picked for all bandits
9: $A^* \leftarrow$ An array of size M    ▷ Optimal action for all bandits
10: $R_{\text{ave}} \leftarrow$ An array of size T    ▷ Average rewards at each time step
11: $P_{\text{ave}} \leftarrow$ An array of size T    ▷ Percentage of optimal action at each time step
12: for $i \leftarrow 1...M$ do
13:     for $j \leftarrow 1...A$ do
14:         $Q_a[i][j] \leftarrow 0$    ▷ Initialize expected reward
15:         $q_a^*[i][j] \leftarrow \mathcal{N}(0, 1)$    ▷ Initialize true reward
16:         $N_a[i][j] \leftarrow 0$    ▷ Initialize number of picks
17:     end for
18: end for
19: for $i \leftarrow 1...M$ do
20:     $A^*[i] = \arg\max_j q_a^*[i][j]$    ▷ Best action per bandit
21: end for
22: for $t \leftarrow 1...T$ do
23:     $R_{\text{ave}}[t] \leftarrow 0$    ▷ Initialize average rewards
24:     $P_{\text{ave}}[t] \leftarrow 0$    ▷ Initialize percentage of optimal action
25: end for
26:
27: RUN TEST BED
28: for $t \leftarrow 1...T$ do    ▷ Do the following for each time step
29:     for $i \leftarrow 1...M$ do    ▷ Do the following for each bandit
30:         $j \leftarrow \begin{cases} \arg\max_i Q_a[i] & \text{probability } 1-\epsilon \\ \text{random value between 1 to A} & \text{probability } \epsilon \end{cases}$    ▷ $\epsilon - greedy$
31:         $N_a[i][j] \leftarrow N_a[i][j] + 1$    ▷ Increment counts for action $j$
32:         $R \leftarrow \mathcal{N}(q_a^*[i][j], 1)$    ▷ Get reward from action $j$
33:         $Q_a[i][j] \leftarrow Q_a[i][j] + \frac{1}{N_a[i][j]}[R - Q_a[i][j]]$    ▷ Update expected reward for action $j$
34:         $R_{\text{ave}}[t] \leftarrow R_{\text{ave}}[t] + R$    ▷ Accumulate rewards
35:         $P_{\text{ave}}[t] \leftarrow \begin{cases} P_{\text{ave}}[t] + 1 & j = A^*[i] \\ P_{\text{ave}}[t] & j \neq A^*[i] \end{cases}$    ▷ Accumulate picks of optimal action
36:     end for
37:     $R_{\text{ave}}[t] \leftarrow \text{mean}(R_{\text{ave}}[t])$    ▷ Average reward at time step $t$
38:     $P_{\text{ave}}[t] \leftarrow \text{mean}(P_{\text{ave}}[t])$    ▷ Percentage of optimal action at time step $t$
39: end for
```

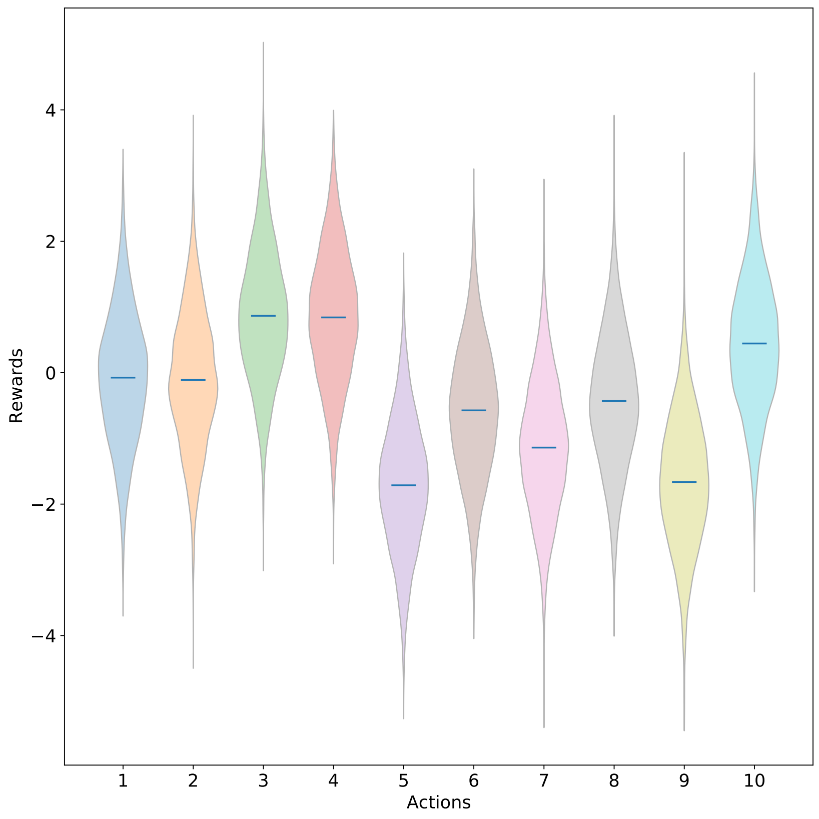


Figure 1: Reward Distribution from A Sample 10-Armed Bandit

and then drops again. This suggests that the best exploitation-vs-exploration combination is to exploit most of the time yet also explore from time to time. It is interesting to note that $\epsilon = 0.1$ has better performance than $\epsilon = 0.01$ during the first 1,000 time steps, yet the latter catches up and overtakes the former when the time step extends beyond 2,000. This implies that in real practice, the optimal ϵ value is dependent also on the total number of time steps to be taken.

The story of the bottom plot of Figure 2 is no less interesting. While the percentage of optimal action curves of $\epsilon = 1$, $\epsilon = 0.6$, $\epsilon = 0.3$ and $\epsilon = 0.1$ follow the same trend as depicted in the top plot, the other three curves demonstrate different behaviors. For $\epsilon = 0$ and $\epsilon = 0.001$, their percentage of optimal action curves are lower than that of $\epsilon = 0.3$, plateauing around 40%. However, the average reward of $\epsilon = 0$ and $\epsilon = 0.001$ are similar or higher than that of $\epsilon = 0.3$. This suggests that despite $\epsilon = 0$ and $\epsilon = 0.001$ not being able to branch out and pick the optimal action, their exploitation of sub-optimal actions compensates for the lack of optimal picks. On the contrary, although $\epsilon = 0.3$ chooses the optimal action more often, it also picks bad actions more often, which negates the benefit of optimal picks. In summary, high reliance on exploitation would pick many sub-optimal actions, whereas moderate reliance on exploration would pick many optimal as well as bad actions. The overall average rewards from these two strategies are similar.

4 The Ads Dataset

The ads dataset is a matrix of size $(10000, 10)$, where each row shows which ads are clicked, i.e., rewarded, at a specific time step and each column represents an ad. The structure of the ads dataset makes it a perfect fit for a 10-armed bandit, where each action corresponds to an ad and the reward of an action corresponds to whether the ad is clicked at a specific time step.

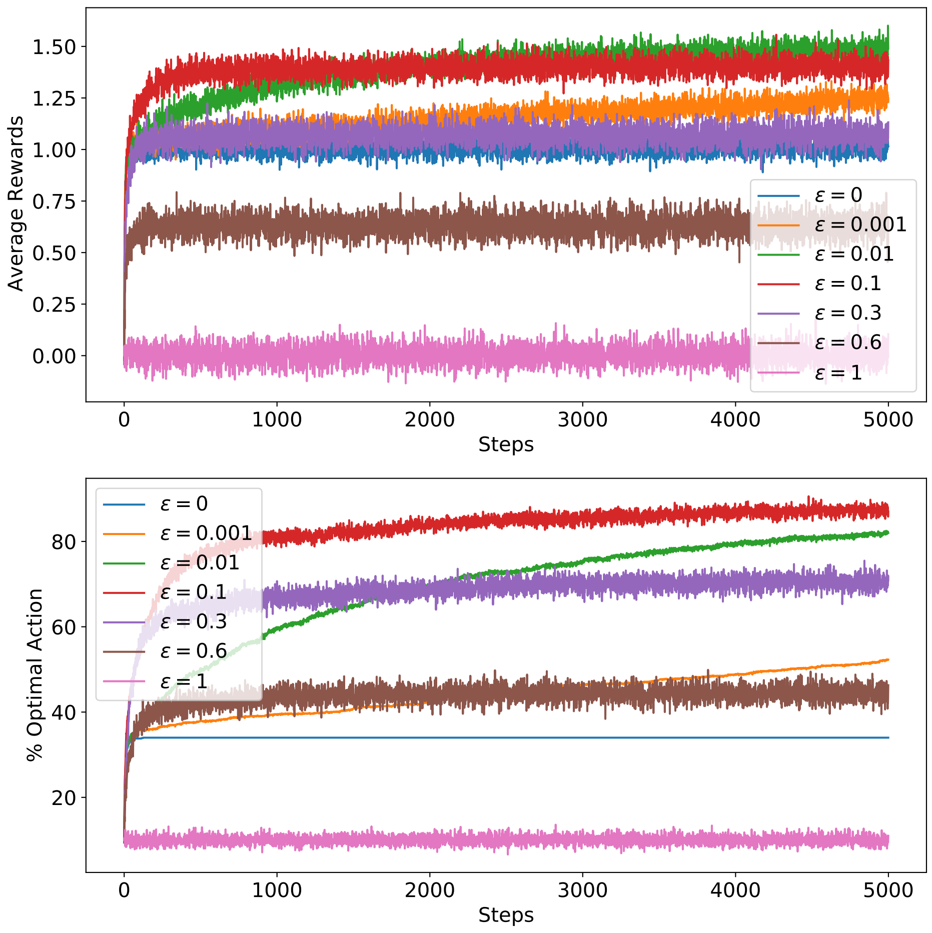


Figure 2: Average Reward and Percentage of Optimal Action Over Different ϵ

Our goal is to use the ads dataset to study two things:

1. What exploitation-vs-exploration combination yields the best total rewards for the entire ads dataset.
2. Which ad has the highest average reward.

Turning the ads dataset into a 10-armed bandit allows us to achieve these two goals comfortably.

The set up of the 10-armed bandit based on the ads dataset is almost the same as described in Section 3. In particular, 100 10-armed bandits are constructed, each based on the same ads dataset. At each time step, action is selected on each bandit according to the $\epsilon - greedy$ rule, and across the same seven ϵ values shown in Section 3. The immediate rewards from all bandits are averaged per time step and recorded.

One major difference between the 10-armed bandit based on the ads dataset and that from Section 3 is that the reward is not generated from Gaussian Distribution. Instead, the reward is extracted from each row of the ads dataset. For instance, suppose the first row of the ads data set is $[1, 0, 0, 0, 1, 0, 0, 0, 1, 0]$. Then if ad 1 is selected at time step 1, the reward will be 1; similarly, if ad 2 is selected, the reward will be 0. This rule of reward is less stochastic than that in Section 3, yet it is not completely deterministic, because the reward distribution varies across time steps.

Finally, additional data such as the average reward and average number of picks per ad are computed across all bandits after all time steps are completed. These data help us understand the performance of each ad and how different exploitation-vs-exploration combinations affect ad choices.

4.1 Evaluation

Figure 3 shows the total rewards of the ads dataset when executed as a 10-armed bandit over different ϵ values. The top plot shows the trend of total rewards across all 10,000 steps, with x-axis being the time steps and y-axis the total rewards. The bottom plot reveals the final rewards achieved under different ϵ values, with the x-axis being the ϵ values and y-axis the final rewards. From the top plot, it is apparent that all ϵ values lead to steady increase in total rewards across time steps. The ranking of performance in ϵ is stable from the beginning till the end, with $\epsilon = 0.1$ enjoying the best performance and $\epsilon = 1$ the worst. The same trend is also revealed in the bottom plot, where the final rewards peak at around 2460 when $\epsilon = 0.1$ and decreases on both higher or lower ϵ values.

Similar to the result shown in Figure 2, pure exploitation or exploration yields poor performance. High reward is always achieved with exploitation most of the time and exploration from time to time. Notice that in the ads dataset, peak performance occurs at a higher ϵ value than the 10-armed bandit test bed. This is likely due to the rewarding system in the ads dataset being binary. In the ads dataset, if an ad is to be rewarded, its reward is always 1, regardless whether the ad is optimal or not. This means it is easier for an exploitation strategy to get stuck with a sub-optimal ad. In the 10-armed bandit test bed, however, local max is harder to take hold because it is less likely for sub-optimal action to generate reward of the same magnitude as the optimal one.

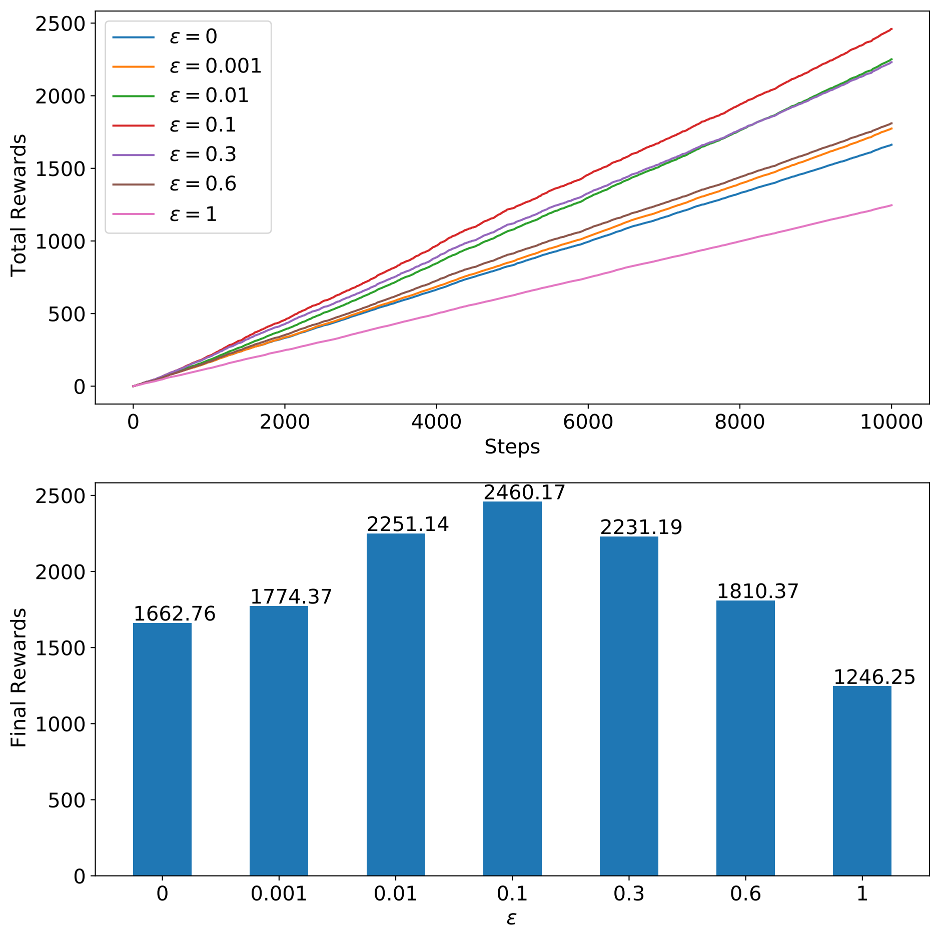


Figure 3: Total Rewards of The Ads Dataset Over Different ϵ

Figure 4 offers a detailed view into the performance of each ad and how different ϵ values affect ad choices. The top plot shows the average rewards of each ad, averaged over 100 bandits after 10,000 time steps, as a bar graph, with different color corresponding to a specific ϵ value. The bottom plot illustrates the average number of picks of each ad, also averaged over all bandits after all time steps, as a bar graph, colored according to the ϵ values.

The top plot of Figure 4 shows that all ϵ values identify that ad 5 has the highest reward. However, as ϵ decreases, the absolute reward achievable in ad 5 decreases. For instance, when $\epsilon = 0$, the average reward from ad 5 is just above 0.05, whereas under $\epsilon = 1$, it is above 0.25. This is understandable, because high ϵ values (e.g. $\epsilon \geq 0.1$) suggest more random sampling, which converges expected reward to true reward better than low ϵ values (e.g. $\epsilon \leq 0.01$).

The bottom plot of Figure 4 demonstrates the average number of picks per ad. This plot can be viewed together with Figure 3 and help us understand how different ϵ values achieve their total rewards. For $\epsilon = 0.1$, it picks ad 5 the most number of times; hence it generates the best total rewards. $\epsilon = 0.3$ and $\epsilon = 0.01$ have similar total rewards, yet their strategies in ad choices are quite different. $\epsilon = 0.3$ picks a lot of ad 5, but at the same time, due to relatively high reliance on exploration, it also does not shy away from bad ads (e.g. ad 6), thus reducing the total rewards. On the other hand, $\epsilon = 0.01$ picks ad 5 less often than $\epsilon = 0.3$, yet it compensates for the missed opportunity by sticking with the next best ads such as ad 8, 9 and 1. Thus overall, the total rewards from $\epsilon = 0.01$ is comparable to $\epsilon = 0.3$.

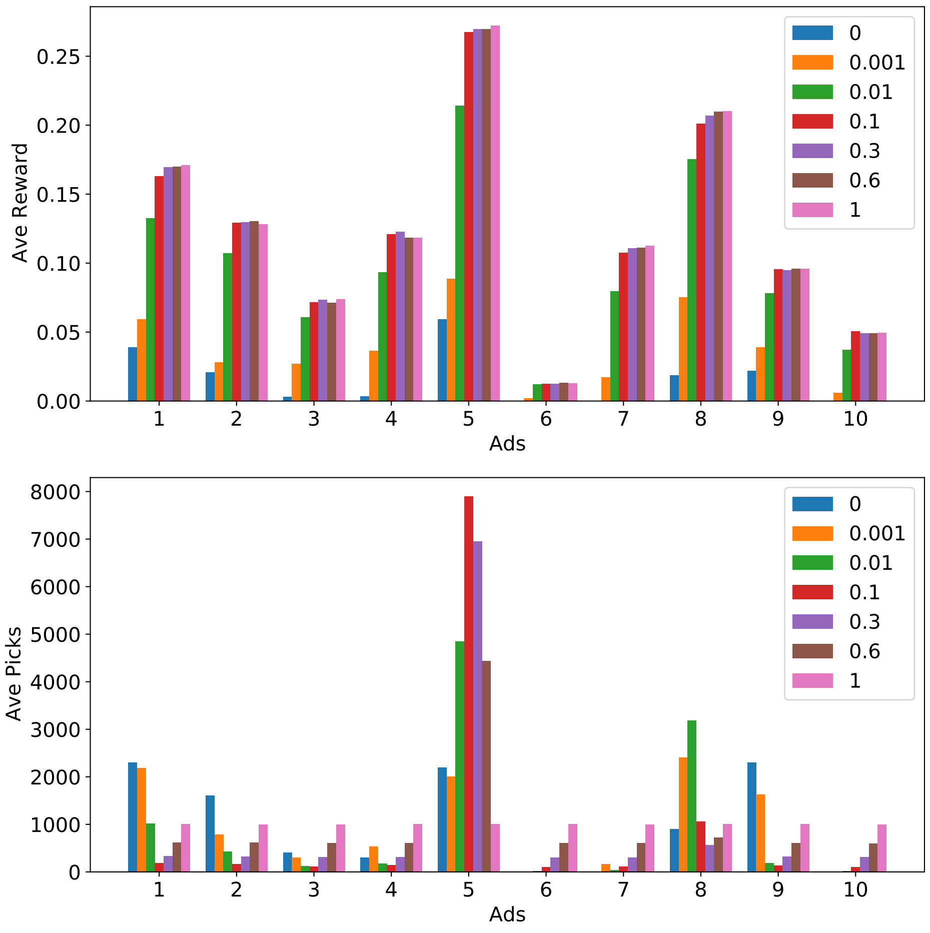


Figure 4: Average Reward and Number of Picks of Each Ad Over Different ϵ

5 Discussion And Conclusion

This report uses two experiment, 10-armed bandit test bed and the ads dataset, to examine the effect of different exploitation-vs-exploration combinations on rewards of multi-armed bandit. We have found that pure exploitation or exploration always leads to poor rewards, and the best performance always occurs with a high probability of exploitation and a low yet not negligible probability of exploration.

The exact ratio between exploitation and exploration seems dependent on how rewards are distributed among actions. If the rewards are granted in a more stochastic distribution model, such as the one used in the 10-armed bandit test bed, less exploration (0.99 exploitation and 0.01 exploration) is favored. Yet a more deterministic reward distribution model with no difference in the actual reward granted by an optimal and sub-optimal action, such as the one used in the ads dataset, sees better performance with slightly higher share of exploration (0.90 exploitation and 0.10 exploration). The dependency of exploitation-vs-exploration combination on rule of reward is understandable. The more stochastic the reward distribution, the more forgiving it is to rely on exploitation, as actions with lower true rewards are less likely to compete with the optimal action. However, if the rule of reward is more deterministic and the actual reward itself cannot differentiate optimal and sub-optimal actions, actions with lower true rewards are more likely to become the temporary best action and fool the exploitation strategy.

Finally, we observe an interesting compensation mechanism between moderately high and moderately low ϵ values. Both achieve fairly high total rewards, but moderately high ϵ value (e.g. $\epsilon = 0.3$ in both the 10-armed bandit test bed and the ads dataset) always chooses the optimal action more often than moderately low ϵ value (e.g. $\epsilon = 0.001$ in the 10-armed bandit test bed and $\epsilon = 0.01$ in the ads dataset). It is expected that moderately high ϵ value selects the optimal action more often, due to the increased reliance on exploration. However, more exploration also means that moderately high ϵ value also picks a lot more bad actions, which brings down the total rewards. In contrast, moderately low ϵ value does not pick the optimal action as often, yet the loss from not choosing the optimal action is compensated by it sticking with the next best sub-optimal action. At the end, the overall rewards from moderately high and low ϵ values tend to resemble each other.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning, An Introduction.* Cambridge, Massachusetts: The MIT Press, 2018.

p1_Bao

February 16, 2021

[1]:
```
# Ensure that the root directory is in Python's path. This is to make importing
# custom library easier.
from pathlib import Path
import sys
root = Path('.').absolute().parent.parent
if str(root) not in sys.path:
    sys.path.append(str(root))

# built-in packages
from collections import defaultdict
import json
from typing import List, Dict, Tuple, Callable
import dill

# external packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy

# matplotlib param
# https://stackoverflow.com/a/55188780/9723036
# SMALL_SIZE = 10
MEDIUM_SIZE = 15
BIGGER_SIZE = 17
TEXT_BOTTOM = 0.9

plt.rc('font', size=MEDIUM_SIZE)          # controls default text sizes
# plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=MEDIUM_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=MEDIUM_SIZE)    # legend fontsize
# plt.rc('figure', titlesize=BIGGER_SIZE)  # fontsize of the figure title
```

1 Part 1

1.1 Shared class and functions

[42]:
```python
# DEFAULTS
RANDOM_STATE = 42

class N_ARM_BANDIT(object):
    """A class to represent an n-armed bandit task"""

    def __init__(
        self,
        n: int,
        epsilon: float,
        random_state: int = RANDOM_STATE,
    ):
        """Initialize the n-armed bandit task.

        :param n: The number of arms in the bandit, i.e., the number of actions
            allowed in the task.
        :param epsilon: Probability that an action is not greedy.
        :param random_state: Seed for the random state. Default to the value
            stored in RANDOM_STATE.
        """
        # random number generator
        self.rng = np.random.default_rng(random_state)
        self.n = n
        self.epsilon = epsilon
        # estimated reward for each action
        self.Qas = np.array([0.0] * n)
        # number of times each action has been taken
        self.Nas = [0] * n
        # true reward for each action
        self.qas = [self.rng.normal() for _ in range(n)]
        # the optimal action
        self.optimal_act = np.argmax(self.qas)

    def reward(self, act: int) -> float:
        """Compute the current reward for a given action.

        :param act: The index of the action selected.
        :return: A reward value produced upon taking the action. Based on
            the rule of the n-armed bandit, the reward is a random variable from
            a Gaussian distribution defined by the true value of the action with
            standard deviation being 1.
        """
        return self.rng.normal(self.qas[act], 1)
    def pick_action(self) -> int:
        """Pick an action to execute.

        With epsilon probability, the picking is purely random (exploration).
        With the remaining probablity, the picking is always greedy
        (exploitation).

        If multiple actions have the same max reward estimate, the tie-breaker
        is by random selection.

        :return: The index of the action picked.
        """
        if self.rng.random() < self.epsilon:  # random exploration
            return self.rng.integers(0, self.n)
        # greedy with random tiebreaks
        pot_act = np.where(self.Qas == np.max(self.Qas))[0]
        return self.rng.choice(pot_act)

    def step(self):
        """Perform a step, i.e., take an action, in the n-armed bandit task.

        In a step, we first pick the action to take, find out the reward
        associated with the action, and update the estimated action value for
        the selected action.

        :return: A numpy array containing two elements: the current step's
            reward and whether the action picked is the optimal action.
        """
        act = self.pick_action()
        R = self.reward(act)
        self.Nas[act] += 1
        self.Qas[act] += 1 / self.Nas[act] * (R - self.Qas[act])
        return np.array([R, act == self.optimal_act])


def n_armed_bandit_test_bed(
    n: int,
    epsilons: List[float],
    num_tasks: int = 2000,
    num_steps: int = 1000,
    random_state: int = RANDOM_STATE,
) -> List[Dict]:
    """Run the test bed for the n-armed bandit experiment.

    The logic of the test bed is as follows: Run `num_steps` of steps. In each
    step, take an action in all `num_tasks` of tasks (bandits). Aggregate the
    mean rewards and percentage of chance that the optimal action has been
    selected in the current step across all tasks. Record the above-mentioned
    two aggregates.

    Once all steps have been taken, all bandits and the result (the aggregated
    mean rewards and optimal action percentage )are saved as pickle files. Also,
    the result is returned.

    :param n: The number of arms in the bandit task.
    :param epsilons: All the epsilon values to be examined.
    :param num_tasks: Total number of tasks (bandits).
    :param num_steps: Total number of steps.
    :param random_state: Seed for the random state. Default to the value stored
            in RANDOM_STATE.
    :reutrn: A list of dict. Each dict corresponds to an epsilon value. In each
        dict, there are three keys: "ave_reward" for a list of average rewards,
        one per step; "opt_act" for a list of percentage that optimal action has
        been selected for each step; and "epsilon" for the epsilon value.
    """
    rng = np.random.default_rng(random_state)
    # Create `num_tasks` random tasks for each epsilon
    epsilon_tasks = [
        [N_ARM_BANDIT(n, e, rng.integers(0, 1000000)) for _ in␣
↪range(num_tasks)] for e in epsilons
    ]
    res = [{'ave_reward': [], 'opt_act': [], 'epsilon': e} for e in epsilons]
    for _ in range(num_steps):
        # all the tasks under a specific epsilon
        for i, tasks in enumerate(epsilon_tasks):
            # perform action over all tasks
            R_optimal_act = np.array([task.step() for task in tasks])
            res[i]['ave_reward'].append(np.mean(R_optimal_act[:,0]))
            res[i]['opt_act'].append(np.mean(R_optimal_act[:,1]))

    with open('n_armed_bandit_test_bed_details.pickle', 'wb') as f_obj:
        dill.dump(epsilon_tasks, f_obj)
    with open('n_armed_bandit_test_bed_result.pickle', 'wb') as f_obj:
        dill.dump(res, f_obj)
    return res


def plot(
    test_bed_res: List[Dict],
    num_steps: int = 1000,
    filename: str = '',
) -> None:
    """Plot the step vs. average rewards, and step vs. optimal action %.
    :param test_bed_res: The result of the test bed. For details, see the
        return value explanation in the function `n_armed_bandit_test_bed`.
    :param num_steps: Total number of steps.
    :param filename: Name of the file where the plot is to be saved. Default
        to empty string, i.e., do not save the plot as a file.
    """
    steps = np.arange(num_steps)
    fig, axes = plt.subplots(2, 1, figsize=(10, 10))
    for res in test_bed_res:
        axes[0].plot(
            steps,
            res['ave_reward'],
            label=f'$\epsilon={{{res["epsilon"]}}}$',
        )
        axes[1].plot(
            steps,
            np.array(res['opt_act']) * 100,
            label=f'$\epsilon={{{res["epsilon"]}}}$',
        )
    axes[0].set_xlabel('Steps')
    axes[0].set_ylabel('Average Rewards')
    axes[1].set_xlabel('Steps')
    axes[1].set_ylabel('% Optimal Action')

    axes[0].legend()
    axes[1].legend()
    plt.tight_layout()
    if filename:
        plt.savefig(filename)
    else:
        plt.show()
```

1.2 Run The N-armed Bandit Test Bed

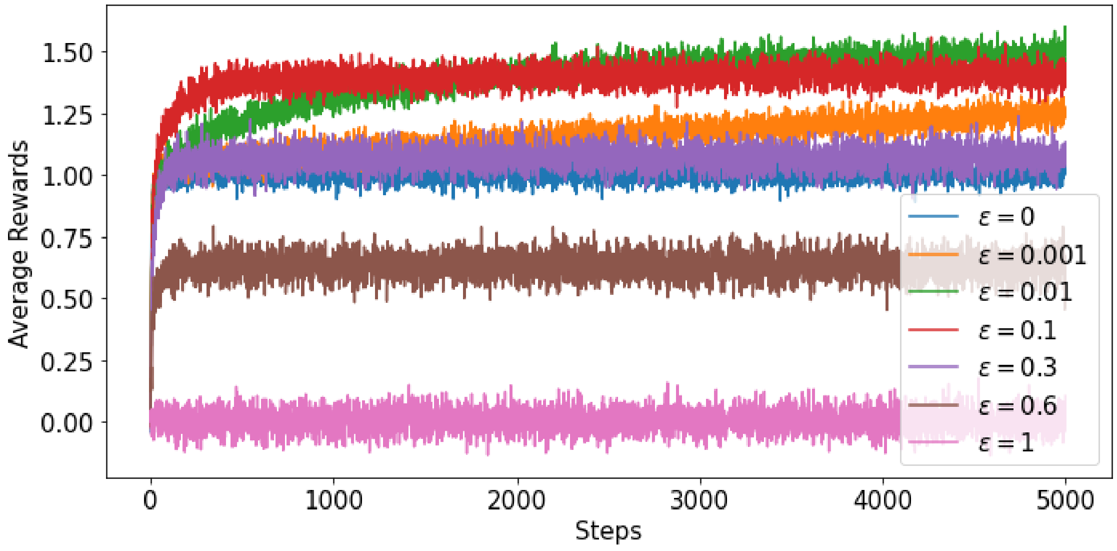
Note that in this test bed, the number of steps have been extended to 10,000. With such extension, it is clear that the $\epsilon = 0.01$ method catches up to $\epsilon = 0.1$ method in the end and becomes the best performing method.

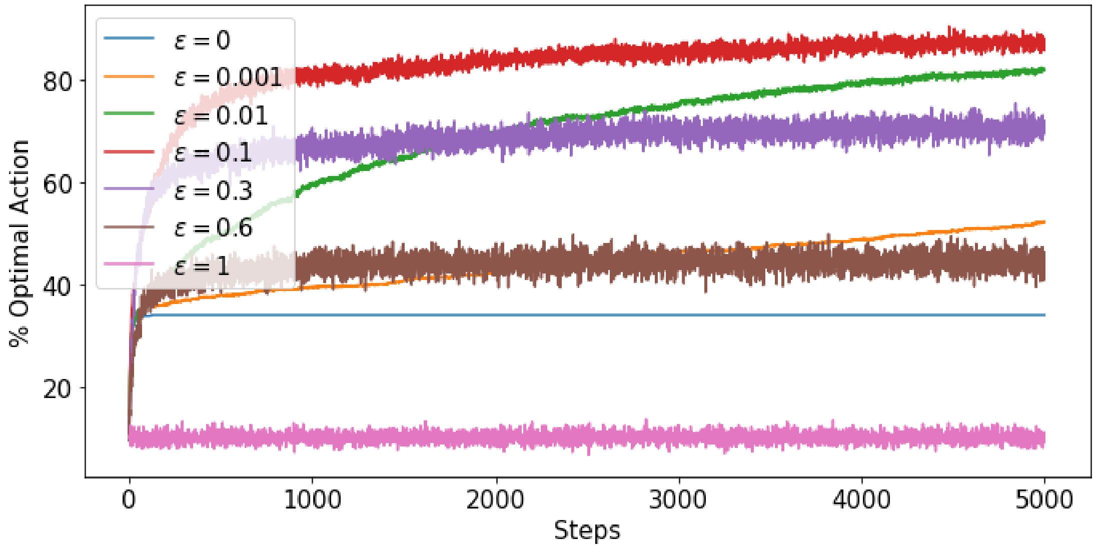
This confirms the statement on p30 of the textbook, which states that the $\epsilon = 0.01$ method outperforms the $\epsilon = 0.1$ method eventually.

[12]:
```
# NOTE this cell takes A LONG TIME to run. Do not run often
# Param configuration
num_arms = 10
num_tasks = 1000
num_steps = 5000
epsilons = [0, 0.001, 0.01, 0.1, 0.3, 0.6, 1]
```

```
res = n_armed_bandit_test_bed(
    num_arms,
    epsilons,
    num_tasks=num_tasks,
    num_steps=num_steps,
)
```

[43]:
```
plot(
    res,
    num_steps=num_steps,
    filename='performance_comparison_epsilon.pdf'
)
```

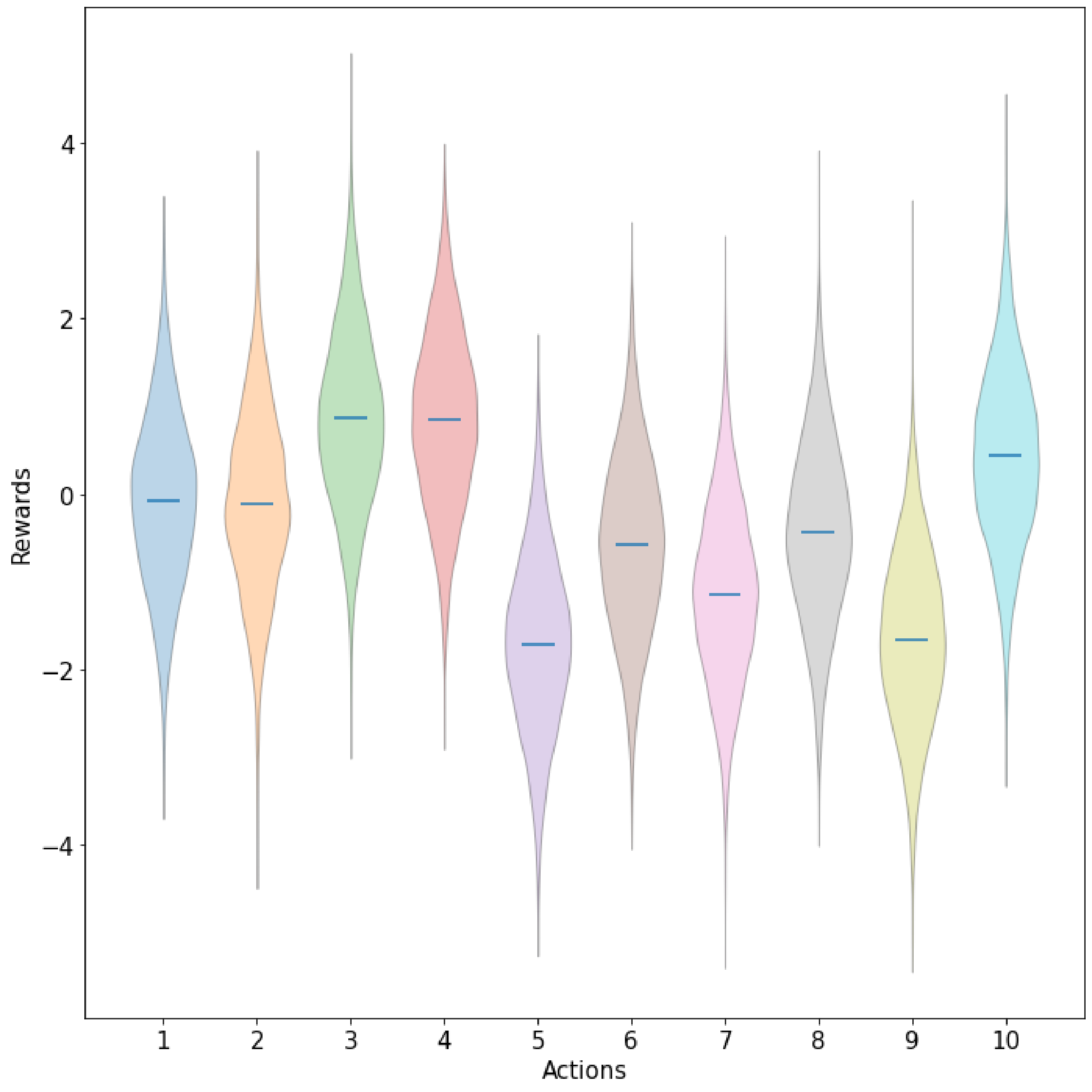




[41]:
```
# Reward distribution of a sample 10-armed bandit
with open('n_armed_bandit_test_bed_details.pickle', 'rb') as f_obj:
    bandits = dill.load(f_obj)

rng = np.random.default_rng(RANDOM_STATE)
rewards_dist = np.array([rng.normal(u, 1, num_steps) for u in bandits[0][0].
 ↪qas]).T

fig, ax = plt.subplots(1, 1, figsize=(10, 10))
vp = ax.violinplot(rewards_dist, showmeans=True, showextrema=False, widths=0.7)
for i, v in enumerate(vp['bodies']):
    v.set_facecolor(f'C{i}')
    v.set_edgecolor('black')
ax.set_xticks(np.arange(1, num_arms + 1))
ax.set_xlabel('Actions')
ax.set_ylabel('Rewards')
plt.tight_layout()
plt.savefig('reward_distribution.pdf')
```



2 Part 2

2.1 Shared class and functions

[59]:
```
# CONSTANTS
RANDOM_STATE = 42

class AD_BANDIT(object):
    """A class to represent an advertisement bandit"""

    def __init__(
        self,
```

```python
        r_dist,
        epsilon: float,
        random_state: int = RANDOM_STATE,
    ):
        """Initialize the ad bandit.

        :param r_dist: A numpy array of array recording the reward
            distribution for all ads and over all steps.
        :param epsilon: Probability that an action is not greedy.
        :param random_state: Seed for the random state. Default to the value
            stored in RANDOM_STATE.
        """
        # random number generator
        self.rng = np.random.default_rng(random_state)
        self.r_dist = r_dist
        self.epsilon = epsilon
        # estimated reward for each action
        self.Qas = np.array([0.0] * self.r_dist.shape[1])
        # number of times each action has been taken
        self.Nas = [0] * self.r_dist.shape[1]

    def reward(self, act: int, step: int) -> float:
        """Find the reward for a given action at the given step.

        :param act: The index of the action selected.
        :param step: The current step.
        :return: A reward obtained from the reward distribution
            corresponding to `step` row and `act` column.
        """
        return self.r_dist[step, act]

    def pick_action(self) -> int:
        """Pick an action to execute.

        With epsilon probability, the picking is purely random (exploration).
        With the remaining probablity, the picking is always greedy
        (exploitation). If multiple actions have the same max reward estimate,
        the tie-breaker is by random selection.

        :return: The index of the action picked.
        """
        if self.rng.random() < self.epsilon:  # random exploration
            return self.rng.integers(0, self.r_dist.shape[1])
        # greedy with random tiebreaks
        pot_act = np.where(self.Qas == np.max(self.Qas))[0]
        return self.rng.choice(pot_act)
    def step(self, step) -> float:
        """Perform a step, i.e., take an action, in an ad bandit.

        In a step, we first pick the action to take, then find out the reward
        associated with the action, and update the estimated action value for
        the selected action.

        :param step: The current step.
        :return: The reward of the current step.
        """
        act = self.pick_action()
        R = self.reward(act, step)
        self.Nas[act] += 1
        self.Qas[act] += 1 / self.Nas[act] * (R - self.Qas[act])
        return R


def compute_ad_reward(
    r_dist,
    epsilons: List[float],
    num_repeats: int = 2000,
    random_state: int = RANDOM_STATE,
) -> List[Dict]:
    """Compute the average reward and acumulative it after each step.

    We run `num_repeats` number of bandits (i.e. the ad bandit) per step
    per epsilon to obtain the average reward attainable under different
    method. We also accumulate the rewards after each step to obtain the
    max possible rewards achieveable under different method.

    :param r_dist: A numpy array of array representing the reward
        distribution.
    :param epsilons: A list of probability that an action is not greedy.
        Each epsilon value represent one epsilon-greedy method.
    :param num_repeats: The number of repetitions the ad bandit is run in
        each step.
    :param random_state: Seed for the random state. Default to the value
            stored in RANDOM_STATE.
    """
    rng = np.random.default_rng(random_state)
    # Create `num_repeats` random tasks for each epsilon
    epsilon_bandits = [
        [AD_BANDIT(r_dist, e, rng.integers(0, 1000000)) for _ in␣
 ↪range(num_repeats)] for e in epsilons
    ]
    res = [
        {
            'total_reward': [0.0],
            'epsilon': e,
            'reward_per_ad': None,
            'picks_per_ad': None,
        } for e in epsilons
    ]
    for step in range(r_dist.shape[0]):
        # all the bandits under a specific epsilon
        for i, bandits in enumerate(epsilon_bandits):
            # perform action over all bandits
            Rs = np.array([bandit.step(step) for bandit in bandits])
            # Accumulate the reward (averaged over `num_repeats` number of
            # bandits) of each step,
            res[i]['total_reward'].append(
                np.mean(Rs) + res[i]['total_reward'][-1],
            )
    # Compute average reward and picks per add
    for i, bandits in enumerate(epsilon_bandits):
        all_ad_rewards = np.array([bandit.Qas for bandit in bandits])
        res[i]['reward_per_ad'] = np.mean(all_ad_rewards, axis=0)
        all_ad_picks = np.array([bandit.Nas for bandit in bandits])
        res[i]['picks_per_ad'] = np.mean(all_ad_picks, axis=0)
    return res


def plot_total_rewards(
    ad_reward_res: List[Dict],
    r_dist,
    epsilons: List[float],
    filename: str = '',
) -> None:
    """Plot the step vs. total rewards curve and final reward bar plot.

    :param ad_reward_res: The average reward of running the ads under different
        epsilon values.
    :param r_dist: A numpy array of array representing the reward distribution.
    :param filename: Name of the file where the plot is to be saved. Default
        to empty string, i.e., do not save the plot as a file.
    """
    steps = np.arange(r_dist.shape[0] + 1)
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 10))
    # Reward trend plot
    for res in ad_reward_res:
        ax1.plot(
            steps,
            res['total_reward'],
            label=f'$\epsilon={{{res["epsilon"]}}}$',
        )
    ax1.set_xlabel('Steps')
    ax1.set_ylabel('Total Rewards')

    # Final reward per epsilon plot
    bar_x = np.arange(1, len(epsilons) + 1)
    bar_y = np.array([res['total_reward'][-1] for res in ad_reward_res])
    rects = ax2.bar(bar_x, bar_y, 0.5)
    for rect in rects:
        height = rect.get_height()
        ax2.annotate(  # add the value of each bar on top of the bar
            '{0:.2f}'.format(height),
            xy=(rect.get_x(), height + 15),
        )
    ax2.set_xticks(bar_x)
    ax2.set_xticklabels([str(e) for e in epsilons])
    ax2.set_xlabel('$\epsilon$')
    ax2.set_ylabel('Final Rewards')

    ax1.legend()
    plt.tight_layout()
    if filename:
        plt.savefig(filename)
    else:
        plt.show()


def plot_ad_data(
    ad_reward_res: List[Dict],
    num_ads: int,
    epsilons: List[float],
    filename: str = '',
) -> None:
    """Plot the average reward and picks per ad over different epsilons.

    The plot is a bar-plot, with x-axis being the label for each ad and
    y-axis the average reward or average number of picks. Different epsilon
    value is depicted with a unique color.

    :param ad_reward_res: The average reward of running the ads under different
        epsilon values. It contains the average reward per ad.
    :param num_ads: Number of ads.
    :param filename: Name of the file where the plot is to be saved. Default
        to empty string, i.e., do not save the plot as a file.
    """
    num_bars = len(epsilons)
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 10))
    # coordinates where the ad label is shown
    x = np.arange(0, num_bars * num_ads, num_bars)
    width = 0.8  # width of each bar
    # The amount of shifts each bar requires, one per epsilon valu
    deltas = np.arange(-(num_bars - 1) / 2, (num_bars - 1) / 2 + 1, 1)
    # Plot one bar per ad, whose coordinate is computed dynamically
    for res, delta in zip(ad_reward_res, deltas):
        ax1.bar(
            x + delta * width,
            res['reward_per_ad'],
            width,
            label=res['epsilon'],
        )
        ax2.bar(
            x + delta * width,
            res['picks_per_ad'],
            width,
            label=res['epsilon'],
        )

    x_tick_label = [str(n) for n in range(1, num_ads + 1)]
    ax1.set_xlabel('Ads')
    ax1.set_ylabel('Ave Reward')
    ax1.set_xticks(x)
    ax1.set_xticklabels(x_tick_label)
    ax1.legend()

    ax2.set_xlabel('Ads')
    ax2.set_ylabel('Ave Picks')
    ax2.set_xticks(x)
    ax2.set_xticklabels(x_tick_label)
    ax2.legend()

    plt.tight_layout()
    if filename:
        plt.savefig(filename)
    else:
        plt.show()
```

2.2 Compute the rewards

Parameters used:

- ϵ: Seven ϵ values are used $[0, 0.001, 0.01, 0.1, 0.3, 0.6, 1]$
- Number of ad bandits per step: 100

[46]:
```
r_dist = pd.read_csv('Ads_Optimisation.csv').to_numpy()
epsilons = [0, 0.001, 0.01, 0.1, 0.3, 0.6, 1]
rewards = compute_ad_reward(r_dist, epsilons, num_repeats=100)
```

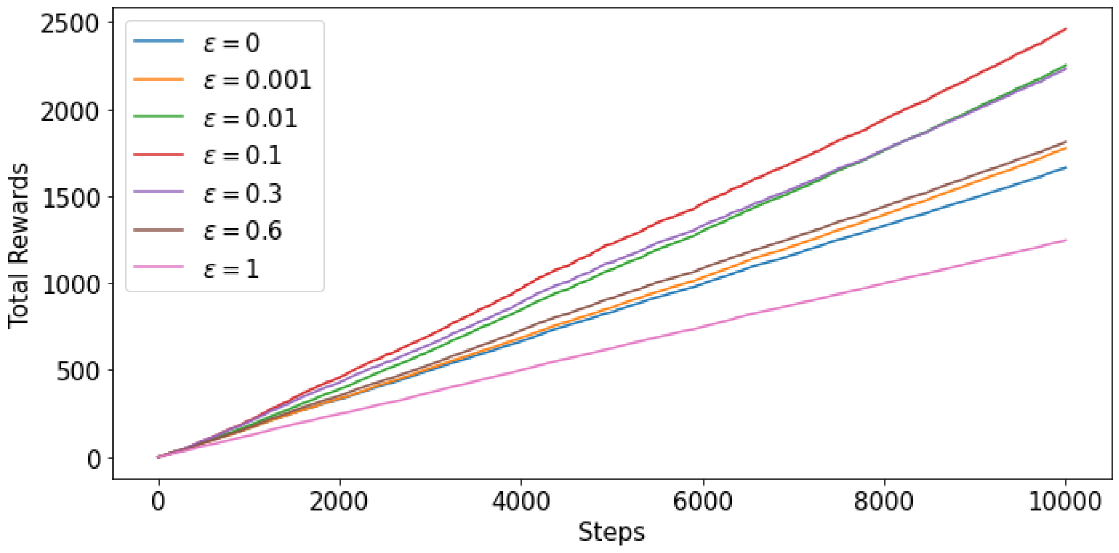
2.3 Reward trend and final reward per ϵ

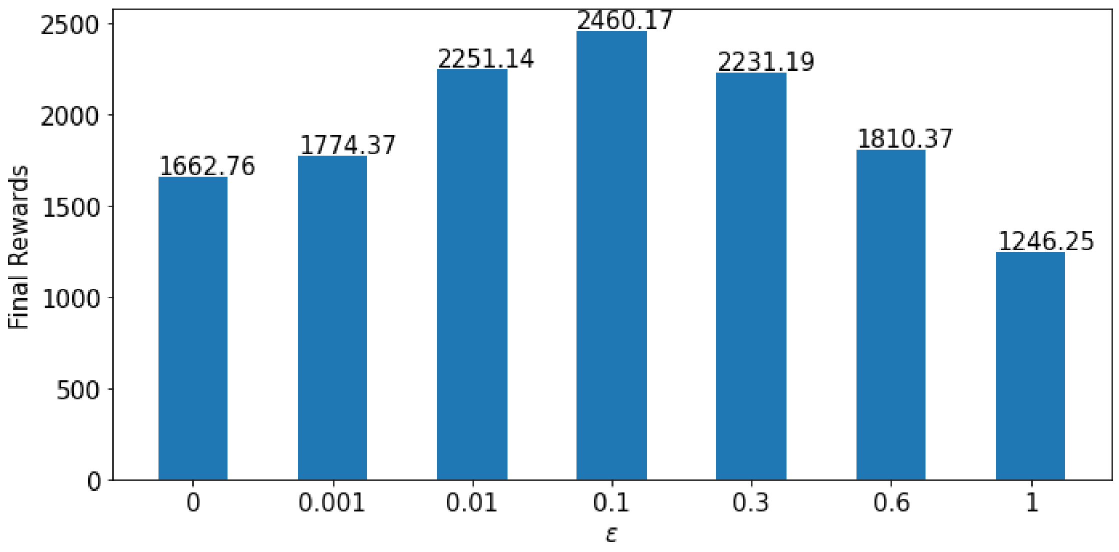
The plots below show the reward trend and the final reward per ϵ value.

The top plot shows the trend of the accumulated rewards over steps for each ϵ. It is obvious that the ranking of the ϵ values is determined pretty early in the steps, and mostly does not change over the entire 10,000 steps. $\epsilon = 0.1$ is the clear winner, while $\epsilon = 1$ performs the worst.

The bottom plot provides a detailed view of the final rewards after 10,000 steps for each ϵ value, averaged over the 100 ad bandits. We observe that as the ϵ value increases, the reward first increases until a peak reached when $\epsilon = 0.1$. Afterwards, the reward decreases. This plot clearly shows that neither pure greedy ($\epsilon = 0$) nor pure random ($\epsilon = 1$) is the best strategy for running the ads. The best performing method is when exploration happens around 10% of the time ($\epsilon = 0.1$).

[60]:
```
plot_total_rewards(
    rewards,
    r_dist,
    epsilons,
    filename='ads_data_total_rewards.pdf'
)
```





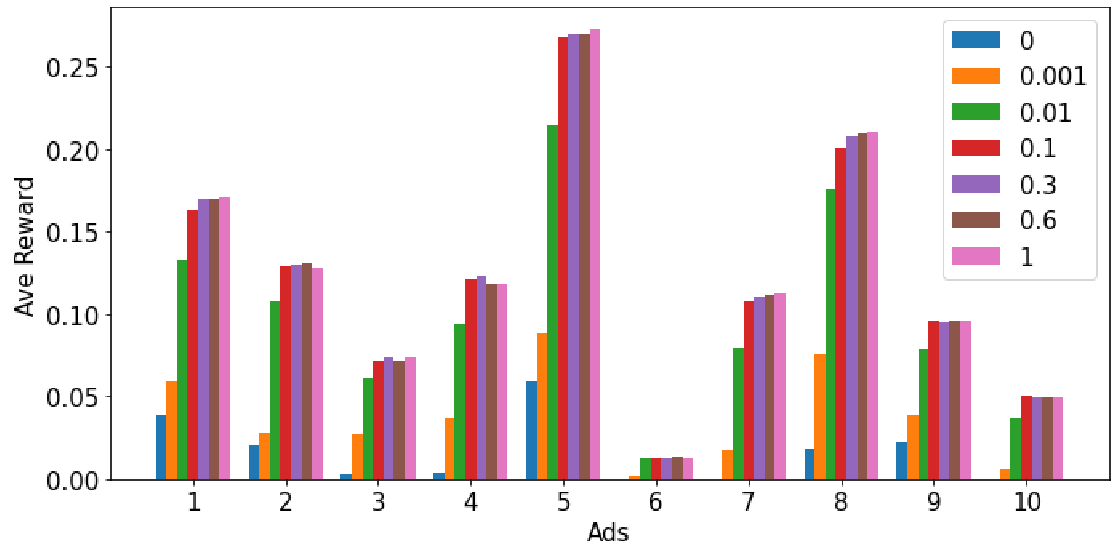
2.4 Average return and number of picks per ad

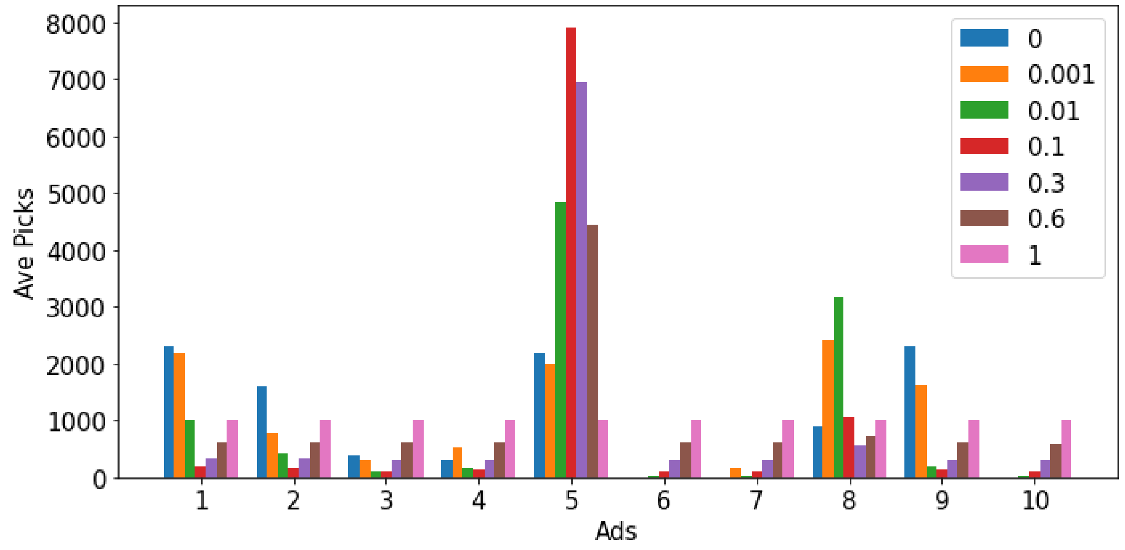
The plots below show the average return and number of picks for each ad under different ϵ values.

The top plot demonstrate the average return of each ad. It is clear that as the probability of exploration increases, the average return computed converges more towards the true reward of each ad. All the methods with $\epsilon \geq 0.1$ have good convergence of each ad's true reward, yet the rest methods with smaller ϵ values have poor convergence. Among all the ads, Ad 5 clearly has the best return.

The bottom plot illustrates the average number of picks through out the 10,000 steps for each ad. The best method is clearly when $\epsilon = 0.1$, as it almost exclusively picks Ad 5. The other methods don't pick Ad 5 as often, hence their inferior performance compared to $\epsilon = 0.1$. When $\epsilon = 1$, i.e. pure exloration, we can see that each ad gets picked about the same number of times.

[54]:
```
plot_ad_data(
    rewards,
    r_dist.shape[1],
    epsilons,
    filename='ads_data_ave_reward_picks.pdf'
)
```





[ ]:
```
```