[S&B Book] Chapter 2: Multi-armed Bandits

Tags

> The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that *evaluates* the action taken rather than *instructs* by giving correct actions

- **Evaluative feedback**

  Indicates how good the action taken was, but not whether it was the best or the worst action possible;

- **Instructive feedback**

  Indicates the correction to take, independently of the action actually taken.

In this section we study the evaluative aspect of reinforcement learning in a simplified setting, *k*-armed bandit problem.

▼ 2.1 A *k*-armed Bandit Problem

- *The k-armed bandit problem*

  > You are faced repeatedly with a choice among *k* different options, or actions. After each choice you receive a numerical reward chosen from a stationary

> probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period.

- The value of an arbitrary action: $q_*(a) \doteq \mathbb{E}[R_t|A_t = a]$
- **greedy actions**:

  When maintaining estimates of the action values, then at any time step there is at least one action whose estimated value is the greatest.
- *Exploration* and *exploitation:*
  - Exploitation - selecting one of the greedy actions; exploiting the current knowledge of the values of the actions;
  - Exploration - selecting one of the non-greedy actions; exploring enables you to improve your estimate of the non-greedy action's value;

  Reward is lower in the short run, during exploration, but higher in the long run because after you have discovered the better actions, you can exploit them many times.

▼ 2.2 Action-value Method

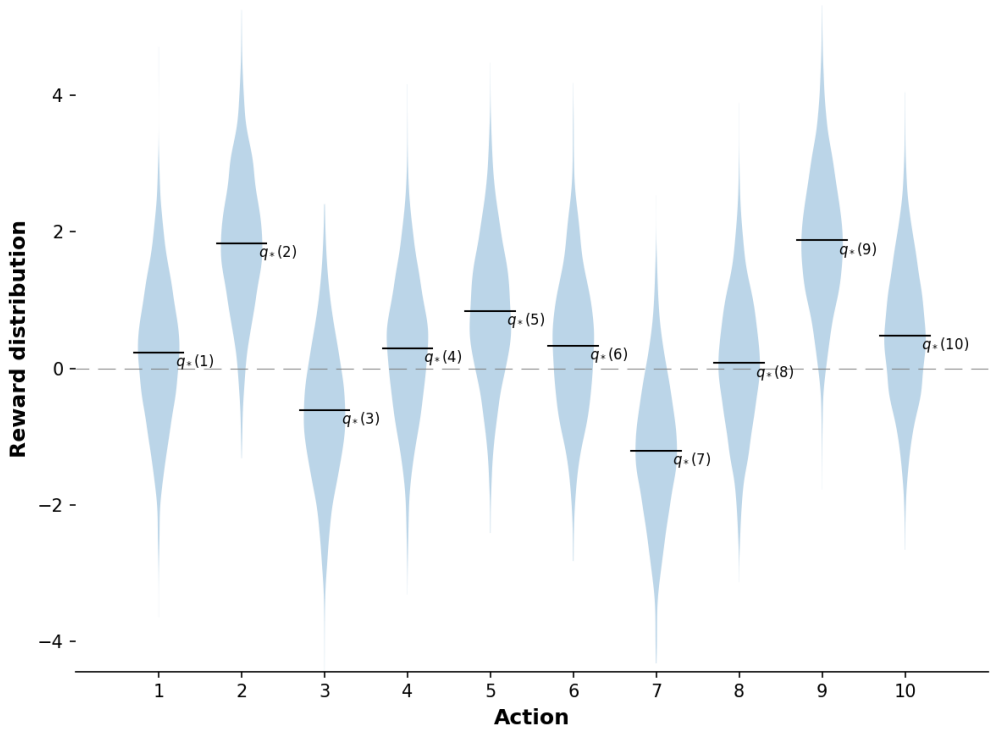
- The way of estimating action-value is by averaging the rewards actually received:

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_t=a}}$$

- *greedy* action selection method: $A_t \doteq \arg\max_a Q_t(a)$, with tie broken randomly; greedy action selection always exploits current knowledge to maximize immediate reward.
- $\varepsilon - greedy$ **method:** with a small probability ε, select randomly from among all the actions with equal probability; with probability $1 - \varepsilon$, select action greedily.

Example:

A set of 2000 randomly generated k-armed bandits problem with k = 10.



Implementation:

```
import numpy as np
from matplotlib import pyplot as plt


if __name__ == "__main__":

    # Randomly sample mean reward for each action
    means = np.random.normal(size=(10, ))

    # Generate sample data based on normal distribution
    data = [np.random.normal(mean, 1, 2000) for mean in means]

    # Create violin plot
    plt.figure(figsize=(8, 6), dpi=150)
    plt.violinplot(dataset=data,
                   showextrema=False,
                   showmeans=False,
                   points=2000)

    # Draw mean marks
    for i, mean in enumerate(means):
        idx = i + 1
        plt.plot([idx - 0.3, idx + 0.3], [mean, mean],
                 c='black',
                 linewidth=1)
        plt.text(idx + 0.2, mean - 0.2,
                 s=f"$q_*({idx})$",
                 fontsize=8)

    # Draw 0 dashed line
    plt.plot(np.arange(0, 12), np.zeros(12),
                c='gray',
```

```
                linewidth=0.5,
                linestyle=(5, (20, 10)))

    plt.tick_params(axis='both', labelsize=10)
    plt.xticks(np.arange(1, 11))

    # get rid of the frame
    for i, spine in enumerate(plt.gca().spines.values()):
        if i == 2: continue
        spine.set_visible(False)


    # Draw labels
    label_font = {
        'fontsize': 12,
        'fontweight': 'bold'
    }

    plt.xlabel('Action', fontdict=label_font)
    plt.ylabel('Reward distribution', fontdict=label_font)
    plt.margins(0)

    plt.tight_layout()
    # plt.show()
    plt.savefig('./plots/example_2_1.png')
```

▼ 2.3 Incremental Implementation

Here I want to first take notes on the 2.3 then back to the e-greedy examples in 2.2 because 2.3 introduced the algorithms to implement the 10-armed bandit problem;

- **The action-value methods on a single action:**

  R_i denotes the reward received after the i-th selection of *this action*, and Q_n denotes the estimate of its action value after it has been selected $n-1$ times:

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}$$

> This method maintain a record of all the rewards and then perform this computation whenever the estimated value was needed. But its disadvantage is that, if it is done, the memory and computational requirements would grow over time as more reward are seen. Each additional reward would require additional memory to store it and additional computation to compute the sum in the numerator.

- Device incremental formulas for updating averages:

$$
\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
&= \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\
&= \frac{1}{n}(R_n + nQ_n - Q_n) \\
&= Q_n + \frac{1}{n}\left[R_n - Q_n\right]
\end{aligned}
$$

Thus, a general form of this method is:

$$NewEstimate \leftarrow OldEstimate + StepSize\left[Target - OldEstimate\right]$$

- Put all together: Pseudocode for **A Simple Bandit Algorithm**

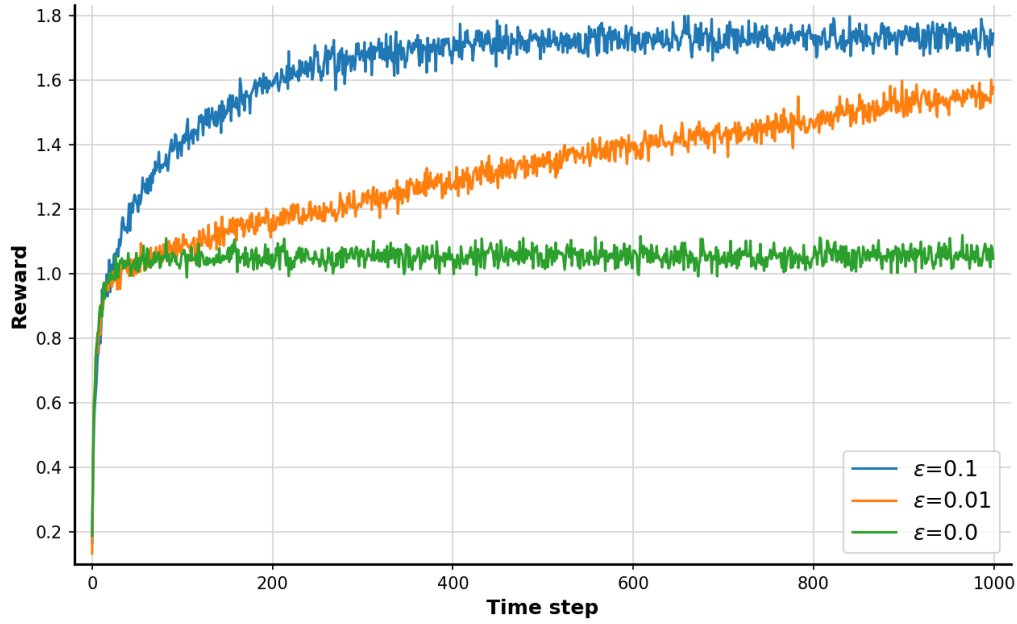
A simple bandit algorithm

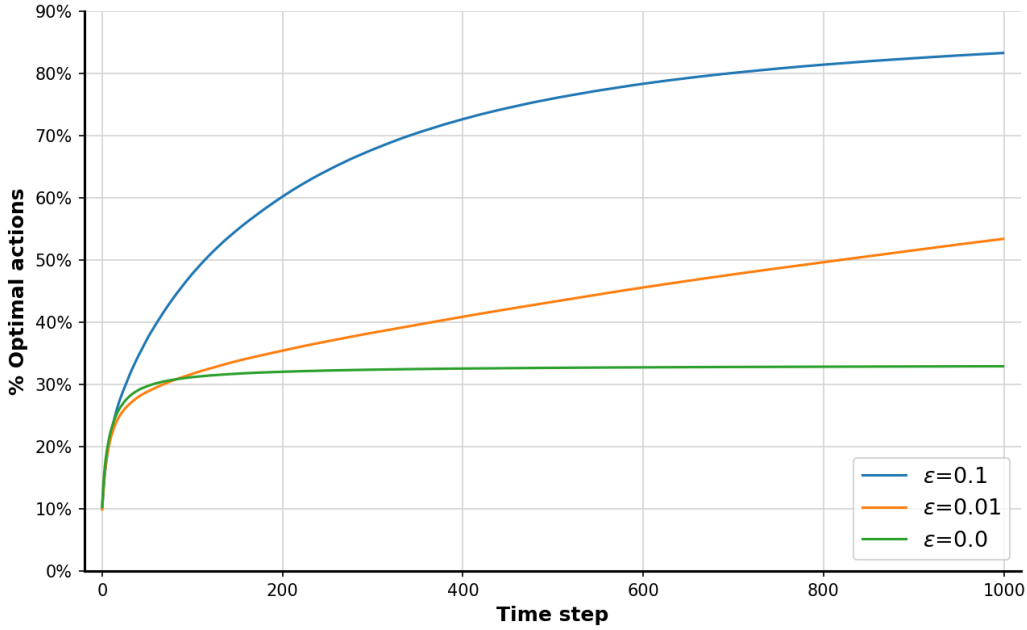
Initialize, for $a = 1$ to k:
  $Q(a) \leftarrow 0$
  $N(a) \leftarrow 0$

Loop forever:
  $A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
  $R \leftarrow bandit(A)$
  $N(A) \leftarrow N(A) + 1$
  $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Example:

Run the 10-armed bandit algorithm on the testbed and compare the effect of using different ε value.





Implementation:

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
```

```
SEED = 200
np.random.seed(SEED)

# Get the action with the max Q value
def get_argmax(G:np.array) -> int:
    candidates = np.argwhere(G == G.max()).flatten()
    # return the only index if there's only one max
    if len(candidates) == 1:
        return candidates[0]
    else:
        # instead break the tie randomly
        return np.random.choice(candidates)


# select arm and get the reward
def bandit(q_star:np.array,
           act:int) -> tuple:
    real_rewards = np.random.normal(q_star, 1.0)
    optim_choice = int(q_star[act] == q_star.max())
    return real_rewards[act], optim_choice


# running the k-armed bandit algorithm
def run_bandit(K:int,
            q_star:np.array,
            rewards:np.array,
            optim_acts_ratio: np.array,
            epsilon: float,
            num_steps:int=1000) -> None:

    Q = np.zeros(K) # The average action-value for each actions
    N = np.zeros(K) # The number of times each action been selected
    ttl_optim_acts = 0

    for i in range(num_steps):
        # print(q_star)
        # get action
        A = None
        if np.random.random() > epsilon:
            A = get_argmax(Q)
        else:
            A = np.random.randint(0, K)

        R, is_optim = bandit(q_star, A)
        N[A] += 1
        Q[A] += (R - Q[A]) / N[A]

        ttl_optim_acts += is_optim
        rewards[i] = R
        optim_acts_ratio[i] = ttl_optim_acts / (i + 1)


if __name__ == "__main__":

    # Initializing the hyper-parameters
    K = 10 # Number of arms
    epsilons = [0.0, 0.01, 0.1]
    num_steps = 1000
    total_rounds = 2000

    # Initialize the environment
    q_star = np.random.normal(loc=0, scale=1.0, size=K)
    rewards = np.zeros(shape=(len(epsilons), total_rounds, num_steps))
    optim_acts_ratio = np.zeros(shape=(len(epsilons), total_rounds, num_steps))
```

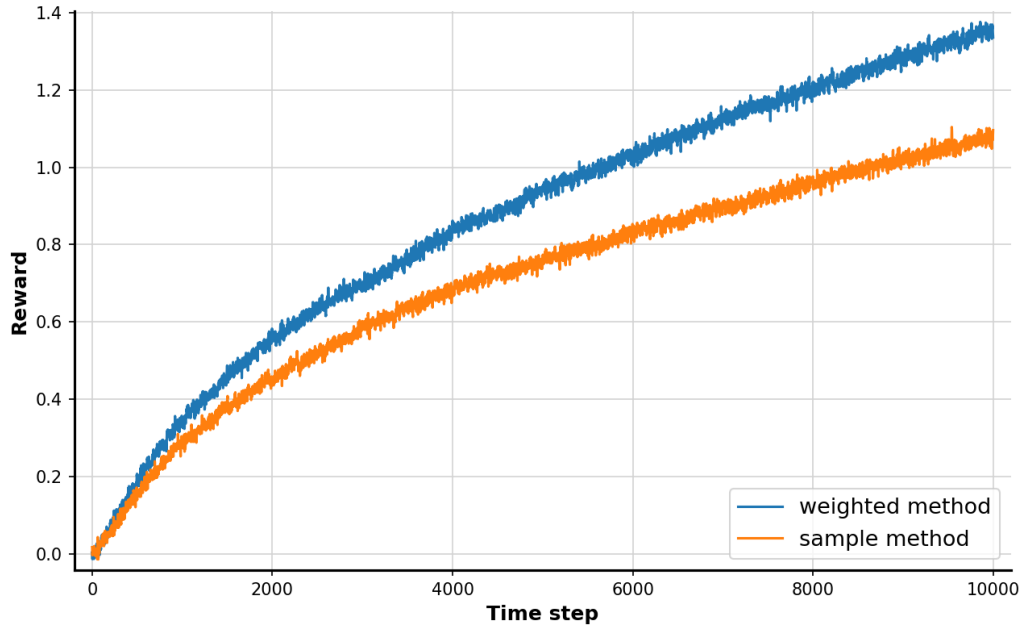
```
    # Run the k-armed bandits alg.
    for i, epsilon in enumerate(epsilons):
        for curr_round in range(total_rounds):
            run_bandit(K, q_star, rewards[i, curr_round], optim_acts_ratio[i, curr_round], epsilon, num_steps)

    rewards = rewards.mean(axis=1)
    optim_acts_ratio = optim_acts_ratio.mean(axis=1)

    record = {
        'epsilons': epsilons,
        'rewards': rewards,
        'optim_ratio': optim_acts_ratio
    }

    # for ratio in optim_acts_ratio:
    #     plt.plot(ratio)
    # plt.show()
    with open('./history/record.pkl', 'wb') as f:
        pickle.dump(record, f)
```

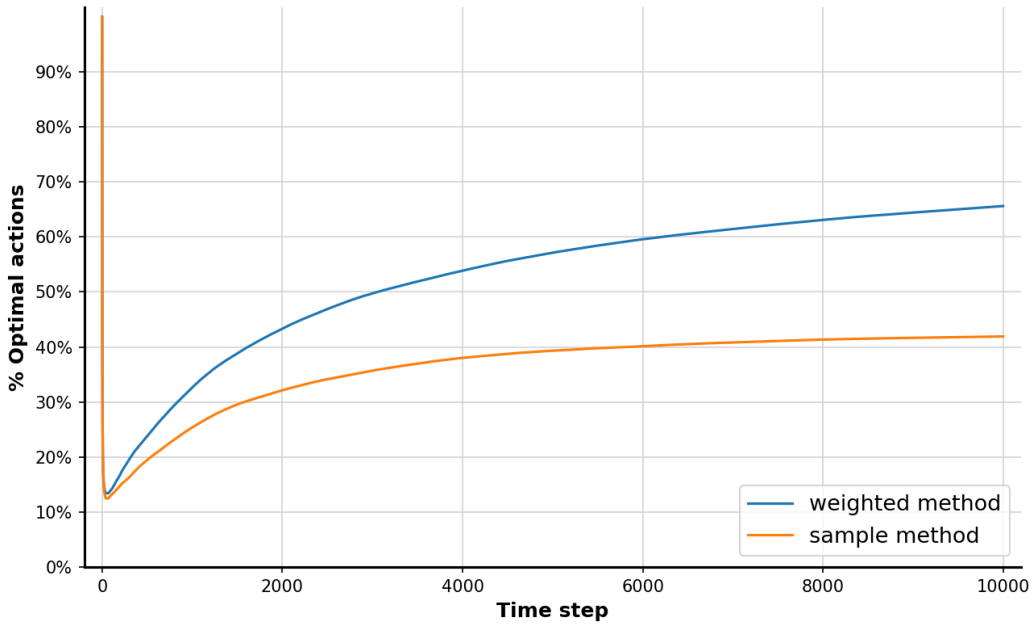
▼ 2.4 Tracking a Nonstationary Problem

Exercise:

Design and conduct an experiment to demonstrate the difficulties that sample-average methods have for non-stationary problems. Use a modified version of the 10-armed testbed in which all the $q_*(a)$ start out equal and then take independent random walks (say by adding a normally distributed increment with mean zero and standard deviation 0.01 to all the $q_*(a)$ on each step). Prepare plots like Figure 2.2 for an action-value method using sample averages, incrementally computed, and another action-value method using a constant step-size parameter, $\alpha = 0.1$ Use $\varepsilon = 0.1$ and longer runs, say of 10,000 steps.

Implementation:






```
# running the k-armed bandit algorithm on non-stationary environment
def run_bandit_non_stationary(K:int,
            q_star:np.array,
            rewards:np.array,
            optim_acts_ratio: np.array,
            epsilon: float,
```

```
        method: str, # method should be in ["sample", "weighted"]
        alpha: float=None,
        num_steps: int=1000) -> None:

Q = np.zeros(K) # The action-value for each actions
N = np.zeros(K) # The number of times each action been selected
ttl_optim_acts = 0

q_star_temp = np.copy(q_star)

assert method in ['sample', 'weighted'], "The method should be 'sample' or 'weighted'"

for i in range(num_steps):
    # print(q_star)
    # get action
    A = None
    if np.random.random() > epsilon:
        A = get_argmax(Q)
    else:
        A = np.random.randint(0, K)

    R, is_optim = bandit(q_star_temp, A)

    if method == 'sample':
        N[A] += 1
        Q[A] += (R - Q[A]) / N[A]
    else:
        Q[A] += alpha * (R - Q[A])

    ttl_optim_acts += is_optim
    rewards[i] = R
    optim_acts_ratio[i] = ttl_optim_acts / (i + 1)

    # Updating q_star values
    q_step_scale = 0.01
    q_steps = np.random.normal(loc=0, scale=q_step_scale, size=q_star_temp.shape)
    q_star_temp += q_steps
```