COMP 138: Homework 01

Shivprasad Shivratri

February 10, 2025

1 The Multi-Armed Bandit Problem

The multi-armed bandit (MAB) problem is a fundamental concept in reinforcement learning where an agent repeatedly chooses between different actions (arms) to maximize cumulative reward. The challenge lies in balancing exploration (trying different arms to learn their values) and exploitation (choosing the arm believed to be the best). The reward received for each arm is based on an unknown probability distribution.

1.1 Action-Value Function

The true value of an action a is the expected reward:

$$q_*(a) = E[R_t|A_t = a]$$

The goal is to estimate these values to make optimal decisions.

1.2 Estimation Methods

Two common methods for estimating action values are:

- **Sample-Average Method:** This method calculates the average reward received for each action. Incrementally, this is updated as:

$$Q_{n+1}(a) = Q_n(a) + \frac{1}{n}[R_n - Q_n(a)]$$

- **Constant Step-Size Method:** This method uses a fixed learning rate (α) to update estimates. It gives more weight to recent rewards.

$$Q_{n+1}(a) = Q_n(a) + \alpha[R_n - Q_n(a)], \quad \text{where } 0 < \alpha \leq 1$$

2 Theoretical Analysis

The Sample-Average method converges to the true action values in *stationary* environments (where the reward distributions don't change). However, in *nonstationary* environments, the Constant Step-Size method is generally preferred. This is because the sample-average method averages over the entire history, which may not accurately reflect the current value of an action if the environment changes. The Constant Step-Size method adapts to changes because it weights recent rewards more heavily.

2.1 Behavior in Non-Stationary Environments

- **Sample-Average Method:** It becomes slow to adapt to the changes, and the estimate represents an average over the entire history of rewards, which may not reflect the current true value.
- **Constant Step-Size Method:** It gives more weight to recent rewards allowing it to track changes more effectively. The estimate represents a weighted average of recent rewards, which is more likely to reflect the current true value. Alpha allows for control over the trade-off between adaptation speed and estimate stability.

3 Experiments

3.1 Experimental Setup

To demonstrate these concepts, a nonstationary MAB environment can be simulated. The environment has multiple arms, each with a reward distribution that changes over time (random walk). Agents using both Sample-Average and Constant Step-Size methods can be tested. The agent uses an epsilon-greedy strategy to select actions, balancing exploration/exploitation.

3.2 NonstationaryBandit Class

This class represents the non-stationary multi-armed bandit environment.

```
class NonstationaryBandit:
    def __init__(self, arms=10, std_dev_q=0.01,
       std_dev_reward=1):
        self.arms = arms
        self.std_dev_q = std_dev_q
        self.std_dev_reward = std_dev_reward
        self.q_true = np.zeros(arms)

    def step(self):
        self.q_true += np.random.normal(0, self.
           std_dev_q, self.arms)

    def reward(self, action):
        return np.random.normal(self.q_true[action],
           self.std_dev_reward)
```

Explanation:

- The `__init__` method initializes the bandit with a specified number of arms and standard deviations for the true values (`std_dev_q`) and rewards (`std_dev_reward`).
- The `step` method simulates the non-stationarity by adding Gaussian noise to the true values of each arm.
- The `reward` method returns a reward for a given action, drawn from a Gaussian distribution centered at the true value of the chosen arm.

3.3 Agent Class

This class represents the learning agent that interacts with the bandit environment.

```
class Agent:
    def __init__(self, arms=10, epsilon=0.1, alpha=None)
        :
        self.arms = arms
        self.epsilon = epsilon
        self.alpha = alpha
        self.q_est = np.zeros(arms)
        self.action_count = np.zeros(arms)

    def select_action(self):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.arms)
        else:
            return np.argmax(self.q_est)

    def update_q_est(self, action, reward):
        self.action_count[action] += 1
        if self.alpha:
            self.q_est[action] += self.alpha * (reward -
                self.q_est[action])
        else:
            self.q_est[action] += (reward - self.q_est[
               action]) / self.action_count[action]
```

Explanation:

- The `__init__` method sets up the agent with a specified number of arms, exploration rate (`epsilon`), and learning rate (`alpha`).
- The `select_action` method implements the epsilon-greedy strategy for action selection.
- The `update_q_est` method updates the estimated value of an action after receiving a reward, using either the constant step-size method (if `alpha` is specified) or the sample-average method.

3.4 run_experiment Function

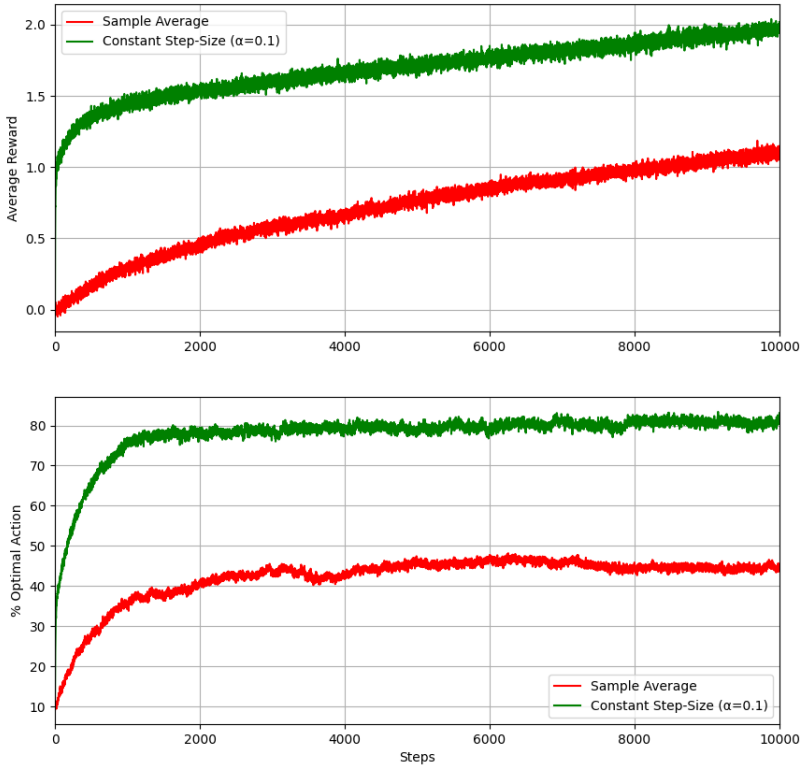
This function simulates the interaction between the agent and the bandit environment over a specified number of steps.

```
def run_experiment(bandit, agent, steps=10000):
    rewards = np.zeros(steps)
    optimal_action_count = np.zeros(steps)

    for t in range(steps):
        action = agent.select_action()
        reward = bandit.reward(action)
        agent.update_q_est(action, reward)
        bandit.step()

        rewards[t] = reward
        optimal_action_count[t] = (action == np.argmax(
            bandit.q_true))

    return rewards, optimal_action_count
```

Explanation:

- The function runs for a specified number of steps, in each step:
  - The agent selects an action
  - The bandit provides a reward for that action
  - The agent updates its estimate of the action's value
  - The bandit's true values are updated (non-stationarity)
- The function keeps track of the rewards received and whether the optimal action was chosen at each step.
- It returns the history of rewards and optimal action selections.

3.5 Experimental results



We can conclude from the results that the constant step-size methods are more effective than sample-average methods in the Non-stationary environments. This is because of their ability to adapt to changing reward distributions as explained earlier. We could also mention that the choice of method depends on the nature of the environment.

3.6 Archive of the code

Clicking here shall redirect you to the Github Repository for Programming Assignment #1

4 EXTRA CREDIT

Exercise 2.6: Mysterious Spikes

The results shown in Figure 2.3 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? In other words, what might make this method perform particularly better or worse, on average, on particular early steps?

Solution:

The gap between different rewards is much larger in the earlier steps compared to the later steps. This large gap between the rewards creates the oscillations and as a result we see the performance as better or worse on early steps.

Exercise 2.8: UCB Spikes

In Figure 2.4 the UCB algorithm shows a distinct spike in performance on the 11th step. Why is this? Note that for your answer to be fully satisfactory it must explain both why the reward increases on the 11th step and why it decreases on the subsequent steps. Hint: If c = 1, then the spike is less prominent.

Solution:

- *Initial exploration*: In the first 10 steps, the UCB algorithm explores all 10 arms once, as the UCB term is infinite for untested arms.
- *Spike on the 11th step*: On the 11th step, all arms have been tested once, and their UCB terms are equal. The algorithm is likely to choose the arm that yielded the highest reward in the first round, which is often the optimal arm.
- *Decrease in subsequent steps*: After the 11th step, the UCB term for the selected arm decreases, prompting the algorithm to explore other arms again. This causes a decrease in performance as suboptimal arms are tested.