Chapter 2: Multi-Armed Bandit

In this Chapter

In this chapter, we will cover the following key topics:

- Fundamentals of the Markov Property
- Understanding the Multi-Armed Bandit (MAB) Problem
- The Exploration-Exploitation Dilemma
- The ϵ-Greedy Algorithm
- Upper Confidence Bounds (UCB) Algorithm
- Thompson Sampling Algorithm
- Applications and Real-World Examples
- Comparison of Algorithms

Aim of this Chapter

The primary aim of this chapter is to provide a comprehensive understanding of the Multi-Armed Bandit (MAB) problem, a foundational concept in Reinforcement Learning (RL). We will explore various algorithms developed to tackle MAB challenges, with a focus on the crucial exploration-exploitation trade-off.

Fundamentals of the Markov Property

The Markov property posits that the future state of a system is dependent solely on its present state and not on past states. Formally, it is expressed as:

$$P(S_{t+1}|S_t, S_{t-1}, \ldots, S_0) = P(S_{t+1}|S_t)$$

In the context of MAB, this property streamlines decision-making, as the choice of arm at time t is based solely on current reward estimates rather than historical choices.

Understanding the Multi-Armed Bandit (MAB) Problem

The MAB problem revolves around the selection among K different actions (arms), each with an unknown reward distribution, with the objective of maximizing cumulative rewards over T trials.

Mathematical Formulation

Define $X_{t,k}$ as the reward from arm k at time t. The expected reward for arm k is given by:

$$\mu_k = \mathbb{E}[X_{t,k}]$$

Our goal is to maximize the cumulative reward:

$$R_T = \sum_{t=1}^{T} X_{t,A_t}$$

where A_t represents the selected arm at time t.

Types of Multi-Armed Bandit Problems

Variations of the MAB problem include:

- **Stochastic Bandits**: Rewards are drawn from a fixed probability distribution.
- **Adversarial Bandits**: Rewards are determined by an adversary, introducing unpredictability.
- **Contextual Bandits**: Arm selection is influenced by context or side information.

The Exploration-Exploitation Dilemma

The MAB problem centers around achieving a balance between:

- **Exploitation**: Selecting the arm with the highest estimated reward.

$$A_t = \arg\max_k \hat{\mu}_k$$

- **Exploration**: Trying less-tested arms to obtain more accurate reward estimates.

Effectively managing this balance is essential for maximizing long-term rewards.

The ϵ-Greedy Algorithm

The ϵ-Greedy algorithm strikes a balance between exploration and exploitation by selecting a random arm with probability ϵ and the arm with the highest estimated reward with probability $1 - \epsilon$.

Algorithm Description

The algorithm is summarized as follows:

Pseudocode for ϵ-Greedy Algorithm

```
Initialize:
  For each arm k:
    count[k] = 0        // Number of times arm k has been selected
    reward[k] = 0       // Total reward received from arm k

Set:
  epsilon = 0.1        // Exploration rate

For each time step t = 1 to T:
  Generate a random number r from Uniform(0, 1)

  If r < epsilon:
    // Exploration
    A_t = random arm selected uniformly from available arms
  Else:
    // Exploitation
    A_t = argmax_k (reward[k] / count[k]) if count[k] > 0
              else select randomly from all arms // Handle untried
                 arms

  // Observe reward from selected arm
  reward_received = X_{t, A_t}  // Reward from the chosen arm A_t

  // Update counts and total rewards
  count[A_t] = count[A_t] + 1
  reward[A_t] = reward[A_t] + reward_received
```

Python Implementation

Python Code for ϵ-Greedy Algorithm

```
import numpy as np

def epsilon_greedy(epsilon, num_arms, num_steps):
    counts = np.zeros(num_arms)  # Number of times each arm is
       selected
    rewards = np.zeros(num_arms)  # Total rewards for each arm
    total_reward = 0

    for step in range(num_steps):
        if np.random.random() < epsilon:
            # Exploration
            arm = np.random.randint(num_arms)  # Randomly select an
               arm
        else:
            # Exploitation
            arm = np.argmax(rewards / (counts + 1e-5))
        # Simulate reward from the chosen arm
        reward_received = np.random.normal(loc=arm + 1, scale=1)
        counts[arm] += 1
        rewards[arm] += reward_received
        total_reward += reward_received

    return total_reward, counts, rewards
```

Advantages	**Disadvantages**
Simple and easy to implement	Fixed exploration rate may not adapt well
Suitable for various applications	Can lead to suboptimal performance
Flexible with adjustable ϵ	Requires tuning for different environments

Table 1: Advantages and Disadvantages of the ϵ-Greedy Algorithm

Upper Confidence Bounds (UCB) Algorithm

The UCB algorithm selects arms based on a confidence interval that adjusts for each arm's uncertainty, effectively balancing exploration and exploitation.

Mathematical Formulation

The UCB for arm k at time t is defined as:

$$UCB_k(t) = \hat{\mu}_k + \sqrt{\frac{2 \ln t}{n_k}}$$

where $\hat{\mu}_k$ is the empirical mean of rewards for arm k, and n_k is the count of selections for arm k.

Algorithm Description

The UCB algorithm can be described as follows:

Pseudocode for UCB Algorithm

```
Initialize:
  For each arm k:
    count[k] = 0        // Number of times arm k has been selected
    reward[k] = 0       // Total reward received from arm k

For each time step t = 1 to T:
  For each arm k:
    If count[k] > 0:
      UCB[k] = reward[k] / count[k] + sqrt(2 * log(t) / count[k])
    Else:
      UCB[k] = infinity   // Ensure untried arms are selected

  A_t = argmax_k (UCB[k]) // Select arm with highest UCB

  // Observe reward from selected arm
  reward_received = X_{t, A_t}

  // Update counts and total rewards
  count[A_t] += 1
  reward[A_t] += reward_received
```

Python Implementation

Python Code for UCB Algorithm

```
import numpy as np

def ucb(num_arms, num_steps):
    counts = np.zeros(num_arms)  # Number of times each arm is
       selected
    rewards = np.zeros(num_arms)  # Total rewards for each arm
    total_reward = 0

    for step in range(num_steps):
        ucb_values = np.zeros(num_arms)

        for arm in range(num_arms):
            if counts[arm] > 0:
                ucb_values[arm] = rewards[arm] / counts[arm] + np.
                   sqrt(2 * np.log(step + 1) / counts[arm])
            else:
                ucb_values[arm] = float('inf')  # Ensure untried arms
                    are selected

        arm = np.argmax(ucb_values)  # Select arm with highest UCB
        reward_received = np.random.normal(loc=arm + 1, scale=1)  #
           Simulate reward
        counts[arm] += 1
        rewards[arm] += reward_received
        total_reward += reward_received

    return total_reward, counts, rewards
```

Advantages	**Disadvantages**
Theoretical guarantees on performance	More complex than ϵ-greedy
Automatically balances exploration and exploitation	Sensitive to parameter choices
Efficient in high-dimensional spaces	May underperform in non-stationary environments

Table 2: Advantages and Disadvantages of the UCB Algorithm

Thompson Sampling Algorithm

Thompson Sampling is a Bayesian approach that selects arms based on probability distributions for rewards, allowing for effective exploration.

Mathematical Formulation

Thompson Sampling utilizes a beta distribution for reward probabilities. Given arm k has been selected n_k times with r_k successes:

$$\theta_k \sim \text{Beta}(r_k + 1, n_k - r_k + 1)$$

The arm is chosen by sampling from these distributions.

Algorithm Description

The Thompson Sampling algorithm is summarized as follows:

Pseudocode for Thompson Sampling Algorithm

```
Initialize:
  For each arm k:
    success[k] = 0       // Number of successful rewards for arm k
    trials[k] = 0        // Total trials for arm k

For each time step t = 1 to T:
  For each arm k:
    sample_theta_k = sample from Beta(success[k]+1, trials[k]-success
       [k]+1)

  A_t = argmax_k (sample_theta_k) // Select arm with highest sampled
     theta

  // Observe reward from selected arm
  reward_received = X_{t, A_t}

  // Update counts and total rewards
  if reward_received > 0:
    success[A_t] += 1
  trials[A_t] += 1
```

Python Implementation

Python Code for Thompson Sampling Algorithm

```
import numpy as np

def thompson_sampling(num_arms, num_steps):
    successes = np.zeros(num_arms)  # Number of successful rewards
        for each arm
    trials = np.zeros(num_arms)      # Total trials for each arm
    total_reward = 0

    for step in range(num_steps):
        theta_samples = np.random.beta(successes + 1, trials -
            successes + 1)
        arm = np.argmax(theta_samples)  # Select arm with highest
            sampled theta
        reward_received = np.random.normal(loc=arm + 1, scale=1)  #
            Simulate reward

        if reward_received > 0:
             successes[arm] += 1
        trials[arm] += 1
        total_reward += reward_received

    return total_reward, successes, trials
```

Advantages	**Disadvantages**
Effective for non-stationary environments	Requires a probabilistic model of rewards
Naturally balances exploration and exploitation	Can be computationally intensive
Theoretically grounded in Bayesian inference	May perform poorly with limited data

Table 3: Advantages and Disadvantages of the Thompson Sampling Algorithm

Applications and Real-World Examples

The MAB framework finds applications across various domains, including:

- Online Advertising: Optimizing ad placements to maximize click-through rates.
- Clinical Trials: Efficiently selecting treatment options for patients.
- A/B Testing: Evaluating multiple variants of a webpage or product.