Week-7: Matchbox Educable Noughts and Crosses Engine and Multi-Armed Bandits using ϵ-Greedy Reinforcement Learning

Chetankumar Kamani (20251603005), Devraj Sakariya (20251603006), Divyesh Dodiya (20251603007),
M.Tech Sem-1, IIIT Vadodara

I. Problem Statement

The learning objectives of this lab are:

- To understand the basic data structures needed for state-space search and reinforcement learning tasks.
- To study the use of random numbers in Markov Decision Processes (MDPs) and RL.
- To analyse the exploration–exploitation trade-off in simple n-armed bandit problems.
- To implement the ϵ-greedy algorithm for both stationary and non-stationary bandits.

The lab is based on the classical **Matchbox Educable Noughts and Crosses Engine (MENACE)** by Donald Michie [1], and on the introductory chapters of Sutton and Barto's *Reinforcement Learning: An Introduction* (2nd edition, Chapters 1–2) [2].

The problems assigned are:

1) Study the MENACE system, examine at least one implementation, and identify crucial data structures and update rules.
2) Implement an ϵ-greedy agent for a **binary bandit** with two Bernoulli arms.
3) Implement a **10-armed bandit** whose mean rewards perform independent random walks.
4) Modify the ϵ-greedy agent to track non-stationary rewards in the 10-armed bandit and evaluate its performance for at least 10,000 time steps.

In this report we summarize the MENACE study and present our Python implementations for Problems (2)–(4).

II. MENACE: Key Ideas (Problem 1)

MENACE is a physical reinforcement learning system to play Tic-Tac-Toe using matchboxes. Each matchbox corresponds to a distinct board state (up to symmetry) and contains coloured beads representing possible moves. The size of each colour pile encodes the preference for that move.

The crucial aspects observed from reference implementations are:

- **State representation:** Board positions are encoded as strings or tuples, often canonicalized by exploiting rotational and reflectional symmetries to reduce the state space.
- **Policy representation:** For each state, a multiset (or vector) of move counts represents the probability of choosing each legal action.
- **Learning rule:** After each game, bead counts are updated (rewarded or punished) depending on the game outcome (win, loss, draw), implementing a simple form of policy-gradient style learning.
- **Exploration:** The randomness of bead sampling naturally performs exploration, especially in the early stages when counts are small.

This idea directly motivates stochastic action selection and incremental learning rules in our bandit implementations.

III. Python Code

A. Binary Bandit with ϵ-Greedy (Problem 2)

We implement a **stationary** 2-armed bandit with Bernoulli rewards. Each arm $a \in \{0, 1\}$ returns reward $R_t \in \{0, 1\}$ with fixed probabilities p_a. An ϵ-greedy agent maintains value estimates $Q_t(a)$ and action counts $N_t(a)$ and updates them using the sample-average method.

```
import numpy as np

# Simulated binary bandit - stationary, 2 arms,
    each returns 0 or 1
class BinaryBandit:
    def __init__(self, prob_arm1=0.7, prob_arm2
        =0.5):
        self.probs = [prob_arm1, prob_arm2]

    def step(self, action):
        return 1 if np.random.rand() < self.probs[
            action] else 0

def epsilon_greedy_bandit(bandit, epsilon=0.1,
    steps=1000):
    Q = np.zeros(2)  # Action value estimates
    N = np.zeros(2)  # Action counts
    rewards = []

    for t in range(steps):
        # epsilon-greedy action selection
        if np.random.rand() < epsilon:
            a = np.random.choice([0, 1])
        else:
            a = np.argmax(Q)

        # interact with environment
        r = bandit.step(a)
```

```
        # incremental sample-average update
        N[a] += 1
        Q[a] += (r - Q[a]) / N[a]
        rewards.append(r)

    return Q, N, rewards

# Run example
if __name__ == '__main__':
    bandit = BinaryBandit(0.7, 0.5)  # Edit reward
         probabilities
    Q, N, rewards = epsilon_greedy_bandit(bandit,
        epsilon=0.1, steps=1000)
    print("Estimated Values (Q):", Q)
    print("Action Counts (N):", N)
    print("Total Reward:", sum(rewards))
```

B. 10-Armed Random-Walk Bandit with Sample-Average ϵ-Greedy (Problem 3)

For the 10-armed bandit, all mean rewards start equal and then follow independent random walks. On each step, the selected arm produces a Gaussian reward around its current mean, and *all* means are then perturbed by a small zero-mean Gaussian increment ($\sigma = 0.01$).

We first apply the standard sample-average ϵ-greedy algorithm, which assumes stationarity and therefore struggles to track the moving optimal arm.

```
import numpy as np

class RandomWalkTenBandit:
    def __init__(self, n_arms=10, mu=0.0,
        sigma_walk=0.01, sigma_reward=1.0):
        self.n_arms = n_arms
        self.means = np.full(n_arms, mu)
        self.sigma_walk = sigma_walk
        self.sigma_reward = sigma_reward

    def step(self, action):
        # reward from chosen arm
        reward = np.random.normal(self.means[
            action], self.sigma_reward)
        # update all means via random walk
        self._random_walk()
        return reward

    def _random_walk(self):
        self.means += np.random.normal(0, self.
            sigma_walk, self.n_arms)

def run_stationary_bandit(bandit, epsilon=0.1,
    steps=1000):
    Q = np.zeros(bandit.n_arms)
    N = np.zeros(bandit.n_arms)
    rewards = []

    for t in range(steps):
        # epsilon-greedy
        if np.random.rand() < epsilon:
            a = np.random.randint(bandit.n_arms)
        else:
            a = np.argmax(Q)

        r = bandit.step(a)

        # sample-average update (bad for non-
            stationary problems)
        N[a] += 1
        Q[a] += (r - Q[a]) / N[a]
        rewards.append(r)

    return Q, N, rewards

if __name__ == "__main__":
    bandit = RandomWalkTenBandit()
    Q, N, rewards = run_stationary_bandit(bandit,
        epsilon=0.1, steps=10000)
    print("Final estimated Q's:", Q)
    print("Action counts:", N)
    print("Total reward:", sum(rewards))
```

C. Modified ϵ-Greedy for Non-Stationary Bandit (Problem 4)

To better handle non-stationary rewards, we replace the sample-average update with a **constant-step-size** update:

$$Q_{t+1}(A_t) \leftarrow Q_t(A_t) + \alpha\big(R_t - Q_t(A_t)\big),$$

where $\alpha \in (0, 1]$ controls how quickly the estimate adapts to recent rewards.

```
import numpy as np

class RandomWalkTenBandit:
    def __init__(self, n_arms=10, mu=0.0,
        sigma_walk=0.01, sigma_reward=1.0):
        self.n_arms = n_arms
        self.means = np.full(n_arms, mu)
        self.sigma_walk = sigma_walk
        self.sigma_reward = sigma_reward

    def step(self, action):
        reward = np.random.normal(self.means[
            action], self.sigma_reward)
        self._random_walk()
        return reward

    def _random_walk(self):
        self.means += np.random.normal(0, self.
            sigma_walk, self.n_arms)

def run_nonstationary_bandit(bandit, epsilon=0.1,
    alpha=0.1, steps=10000):
    Q = np.zeros(bandit.n_arms)
    rewards = []

    for t in range(steps):
        # epsilon-greedy
        if np.random.rand() < epsilon:
            a = np.random.randint(bandit.n_arms)
        else:
            a = np.argmax(Q)

        r = bandit.step(a)

        # constant-step-size update
        Q[a] += alpha * (r - Q[a])
        rewards.append(r)

    return Q, rewards

if __name__ == "__main__":
    bandit = RandomWalkTenBandit()
    Q, rewards = run_nonstationary_bandit(
        bandit, epsilon=0.1, alpha=0.1, steps
            =10000
    )
    print("Final estimated Q's:", Q)
    print("Total reward:", sum(rewards))
```

IV. Execution Instructions

```
pip install numpy

# Problem (2): Binary bandit with epsilon-greedy
python pr7-1.py

# Problem (3): 10-armed random-walk bandit with
    sample-average epsilon-greedy
python pr7-2.py

# Problem (4): 10-armed random-walk bandit with
    constant-step-size epsilon-greedy
python pr7-3.py
```

Each script prints the learned action-value estimates, action counts (where applicable), and the total accumulated reward.

V. Experimental Results

A. Binary Bandit (pr7-1.py)

For the binary bandit with true probabilities $(p_1, p_2) = (0.7, 0.5)$ and $\epsilon = 0.1$ over 1000 steps, a representative run produced:

- Estimated values $Q \approx [0.6880,\ 0.5417]$
- Action counts $N \approx [952,\ 48]$
- Total reward ≈ 681

The agent clearly favors arm 1 (the optimal arm), demonstrating successful exploitation while still occasionally exploring arm 2.

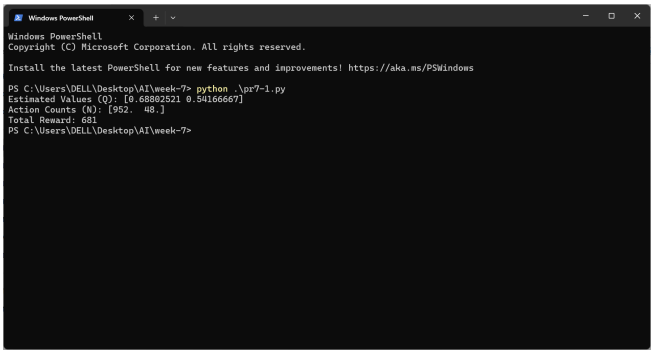

Fig. 1. Console output for the binary bandit (ϵ-greedy, 1000 steps).

B. 10-Armed Random-Walk Bandit with Sample-Average Update (pr7-2.py)

In the non-stationary 10-armed bandit with sample-average updates (10,000 steps, $\epsilon = 0.1$), we observed:

- Highly variable final Q-values, some significantly negative.
- Very uneven action counts, with some arms explored heavily and others rarely.
- Moderate total reward (≈ 5304 in one run).

Because old data are weighted equally with recent data, the agent adapts slowly to changes in the underlying means.

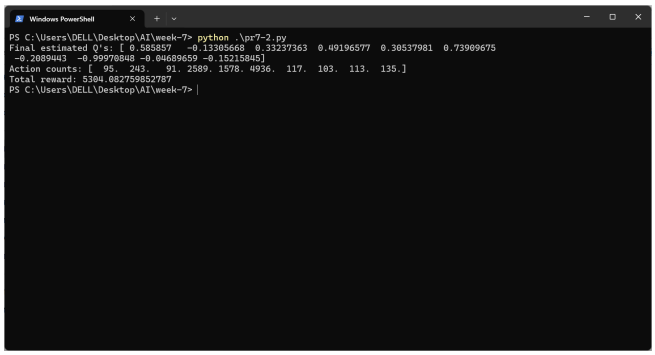

Fig. 2. Output for 10-armed random-walk bandit with sample-average ϵ-greedy (10,000 steps).

C. 10-Armed Random-Walk Bandit with Constant Step Size (pr7-3.py)

Using the constant-step-size update with $\alpha = 0.1$ and $\epsilon = 0.1$ for 10,000 steps, the same environment yields:

- More focused Q-values around the current good arms.
- Improved total reward (e.g., $\approx$ 11,352 in one run), roughly double the previous case.

This shows that the modified ϵ-greedy agent is able to *track* the drifting optimal action more effectively.

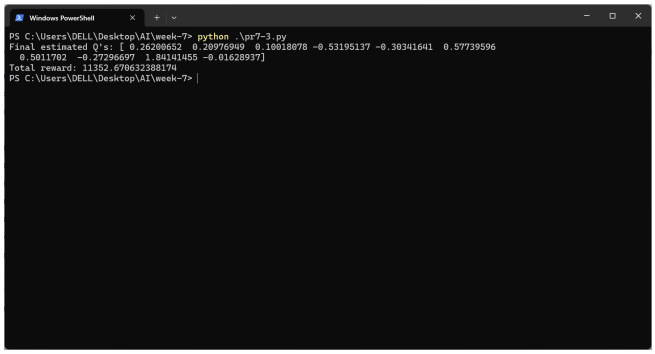

Fig. 3. Output for 10-armed random-walk bandit with constant-step-size ϵ-greedy (10,000 steps).

VI. Discussion

The experiments clearly illustrate the exploration–exploitation trade-off and the importance of using appropriate value update rules:

- In the *stationary* binary bandit, sample-average ϵ-greedy converges to the optimal arm while still exploring.
- In the *non-stationary* 10-armed bandit, sample-average updates are too conservative, failing to quickly adapt to changing means.
- Replacing the sample-average with a constant-step-size update significantly improves performance, as recommended in Sutton and Barto for non-stationary problems.

These results conceptually connect back to MENACE, where the bead counts for moves are updated more strongly based on recent games, making the physical system inherently more responsive to recent experience.

VII. Code Availability

- The complete source code for this lab is available at: GitHub Repository (CS659 – AI Laboratory, Week-7).
- Main repository: https://github.com/ChetanKamani/CS659-LAB-TASK

References

[1] D. Michie, "Experiments on the mechanization of game-learning. Part I. Characterization of the model and its parameters," *Computer Journal*, vol. 6, no. 3, pp. 232–236, 1963. (MENACE: Matchbox Educable Noughts and Crosses Engine).

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.