PROBLEM 1

Implement a multi-armed bandit algorithm

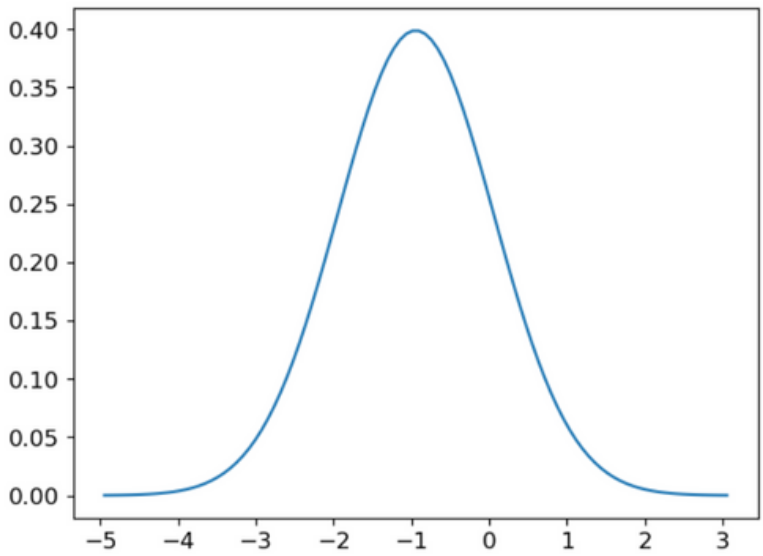
The multi-armed bandit problem is a challenge in reinforcement learning where an agent has to select from multiple options with unknown reward distributions to maximize its total reward over time by balancing exploration and exploitation. This report provides a Python implementation of the problem using the epsilon-greedy algorithm, consisting of three functions: init_bandit(), select_action(), and run_bandit().

The init_bandit() function

The init_bandit() function initializes a multi-armed bandit problem with n arms. It returns the true mean reward for each arm (q_star), the agent's estimated mean reward for each arm (Q), and the number of times each arm has been chosen (N). The initial values of Q and N are set to zero, while the true mean reward for each arm is sampled from a normal distribution.

```
In [14]: def init_bandit(n):
             """
             Initializes a multi-armed bandit problem with `n` arms.

             Args:
                 n (int): The number of arms in the bandit problem.

             Returns:
                 q_star (ndarray): The true mean reward for each arm.
                 Q (ndarray): The agent's estimated mean reward for each arm.
                 N (ndarray): The number of times each arm has been chosen.
             """
             # Initialising true mean reward for each arm
             q_star = np.random.normal(size=n)

             # Initialising agent's estimated mean reward for each arm
             Q = np.zeros(n)

             # Initialising number of times each arm has been chosen
             N = np.zeros(n)

             # returning the initial values of the true mean reward, the estimated reward and the number of times each arm is selected
             return q_star, Q, N
```

The normal distribution of true mean reward for each arm



The select_action() function

The select_action() function selects an action for the agent to take based on its current estimates of the mean rewards for each arm (Q). The function takes two arguments: Q, which is the agent's estimated mean reward for each arm, and epsilon, which is the degree of exploration vs. exploitation. If a random number between 0 and 1 is less than epsilon, the function chooses a random arm to explore. Otherwise, it selects the arm with the highest estimated reward to exploit. If there is a tie for the highest estimate, the function chooses randomly among the tied arms.

In [16]:
```
def select_action(Q, epsilon):
    """
    Selects an action for the agent to take based on its current estimates of the mean rewards for each

    Args:
        Q (ndarray): The agent's estimated mean reward for each arm.
        epsilon (float): The degree of exploration vs. exploitation.

    Returns:
        action (int): The index of the arm to choose.
    """


    if np.random.rand() < epsilon:

        # Explore: choose a random action
        action = np.random.randint(len(Q))

    else:
        # Exploit: choose the action with highest estimate
        maxQ = np.max(Q)

        # If there is a tie for the highest estimate, chooses randomly among the tied arms.
        best = np.where(Q == maxQ)[0]
        if len(best) > 1:
            action = np.random.choice(best)
        else:
            action = best[0]
    return action
```

The run_bandit() function

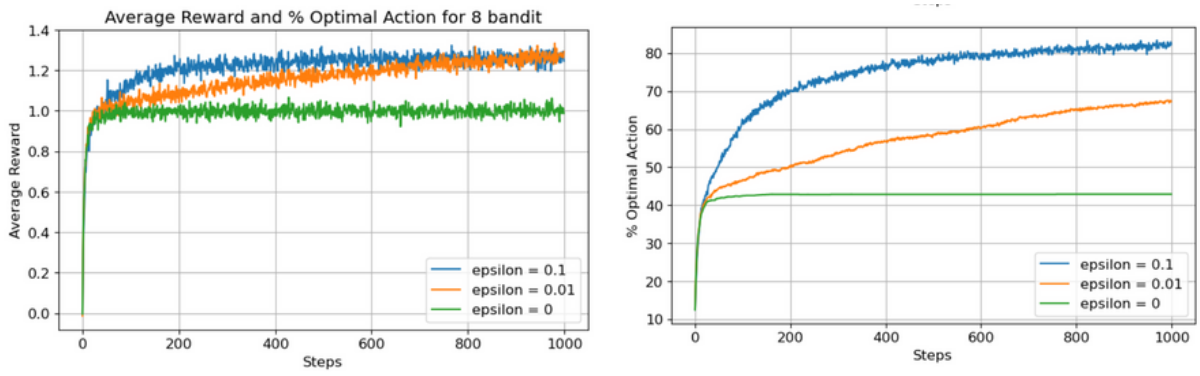
The run_bandit() function runs a multi-armed bandit problem for num_steps time steps with n arms and a given value of epsilon. It returns the rewards received at each time step (rewards) and whether each chosen arm was optimal (i.e., had the highest true mean reward) at each time step (optimal). The function first calls the init_bandit() function to initialize the problem. At each time step, the function calls the select_action() function to select an arm, and then receives a reward by sampling from the true reward distribution of the chosen arm. The function updates the estimated mean reward and the number of times each arm has been chosen based on the reward received.

In [18]:
```
def run_bandit(n, num_steps, epsilon):
    """
    Runs a multi-armed bandit problem for `num_steps` time steps with `n` arms and a given value of `ep

    Args:
        n (int): The number of arms in the bandit problem.
        num_steps (int): The number of time steps to run the bandit problem.
        epsilon (float): The degree of exploration vs. exploitation.

    Returns:
        rewards (ndarray): The rewards received at each time step.
        optimal (ndarray): Whether each chosen arm was optimal (i.e., had the highest true mean reward)
    """
    q_star, Q, N = init_bandit(n)
    rewards = np.zeros(num_steps)
    optimal = np.zeros(num_steps)

    for i in range(num_steps):
        action = select_action(Q, epsilon)
        reward = np.random.normal(q_star[action], 1)
        rewards[i] = reward
        N[action] += 1
        Q[action] += (reward - Q[action]) / N[action]

        if action == np.argmax(q_star):
            optimal[i] = 1

    return rewards, optimal
```

Experimental Results

The implementation of the multi-armed bandit problem is evaluated using the epsilon-greedy algorithm with three different values of epsilon: 0.1, 0.01, and 0 (i.e., always exploit). The number of arms n is randomly chosen to be either 8 or 10. The number of time steps is set to 1000, and the number of runs is set to 2000.

The results of the experiment are shown in a plot that displays the average reward and percentage of optimal action for each value of epsilon over the 2000 runs.



As can be seen from the plot, the performance of the algorithm is affected by the value of epsilon. When epsilon is high (i.e., 0.1), the algorithm explores more and tends to achieve a higher percentage of optimal action, but with a lower epsilon, the algorithm does not explore as much. *(in this case of this run, algorithm selected to run for 8 arm bandit)*

Discussion of the Results.

As we have seen, ur experiment compares a greedy method (epsilon = 0) with two other epsilon-greedy methods (epsilon = 0.01 and epsilon = 0.1), as described above, The number of arms n is randomly chosen to be either 8 or 10. Both methods formed their action-value estimates using the sample-average technique.

The first graph shows the increase in expected reward over time for each method. The greedy method performs slightly better initially, but it levels off quickly and achieves a reward per step of only about 1, compared to the best possible of 1.33. This is because the greedy method often gets stuck performing suboptimal actions. The epsilon-greedy methods continue to explore, leading to better recognition of the optimal action and better performance in the long run.

The second graph shows that the greedy method only found the optimal action in about 40% of the tasks, whereas the epsilon-greedy methods perform better, with the epsilon = 0.1 method finding the optimal action earlier but not always selecting it. The epsilon = 0.01 method improves more slowly but performs better on both performance measures. We can then conclude that, when epsilon is greater, the multi-armbandit needs to balance exploitation and exploraration more and in the long run acquires more rewards.

PROBLEM 2 & 3

Exploration and Exploitation for Multi-armed Bandits

One of the challenges that arise in reinforcement learning is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before.

For the Multi-Arm Bandit problems, the explore-exploit tradeoff arises because the agent must balance between exploiting the arms with the highest known reward ***(exploitation)*** and trying out other arms in the hopes of finding one with a higher reward ***(exploration)***

Note: If the agent only exploits, it will quickly converge on a suboptimal arm and miss out on potentially higher rewards from other arms. Conversely, if the agent only explores, it will never take advantage of high-reward arms and will achieve poor overall performance.

Thus, the agent must continually balance its exploration and exploitation strategies to maximize its total reward over time.

Action-value methods

In reinforcement learning, an agent interacts with an environment by selecting actions and observing rewards. The goal of the agent is to maximize its cumulative reward over time. To achieve this, the agent must learn a policy that maps states to actions that lead to high cumulative reward. The action value represents the expected return of a particular action when taken in a given state. The true value of an action is unknown and needs to be estimated. The sample-average method is one way to estimate the value of an action, which involves averaging the rewards received when the action was selected up to the current time step. The estimated action values can be used to make action selection decisions. The greedy method selects the action with the highest estimated action value, while the ε-greedy method selects the greedy action with probability 1-ε and a random action with probability ε. The ε-greedy method ensures that all actions are sampled and the estimated action values converge to their true values.

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{N_t(a)}}{N_t(a)}.$$

where;

- Qt(a) is the estimated value of action a at time step t,
- Nt(a) is the number of times action a has been selected up to time step t,
- R1, R2, ..., RNt(a) are the rewards received for each of the Nt(a) selections of action a up to time step t.

Reference

1. Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto
2. Algorithms for Reinforcement Learning by Csaba Szepesvári
3. Bandit Algorithms for Website Optimization by John Myles White
4. Deep Reinforcement Learning by Sergey Levine
5. "Playing Atari with Deep Reinforcement Learning" by Volodymyr Mnih et al.
6. "Human-level control through deep reinforcement learning" by Volodymyr Mnih et al.
7. "Multi-armed Bandit Algorithms and Empirical Evaluation" by Tor Lattimore and Csaba Szepesvári (https://arxiv.org/abs/1910.05222)
8. "A Tutorial on Thompson Sampling" by Olivier Chapelle and Lihong Li (https://arxiv.org/abs/1707.02038)
9. "Asynchronous Methods for Deep Reinforcement Learning" by Volodymyr Mnih et al.