DRL HW03_MAB

7111056426_蘇亭云

💡 prompt：list 4 multi-armed bandit algorithms and their algorithms

Multi-armed bandit algorithms are used to solve the explore-exploit dilemma in reinforcement learning problems. Here are four popular algorithms with their corresponding algorithms:

1. Epsilon-greedy algorithm: This is a simple algorithm that randomly selects between exploration and exploitation based on a preset value of epsilon.

Algorithm:

- Initialize Q(a) and N(a) to zero for all actions a.
- Repeat for t=1, 2, ..., T:
    - With probability epsilon, select a random action (explore).
    - Otherwise, select the action with the highest Q-value (exploit).
    - Observe the reward R_t and update the Q-value and count for the selected action.
- Return the action with the highest Q-value.

1. Upper Confidence Bound (UCB) algorithm: This algorithm selects the action that maximizes the upper confidence bound of its estimated value.

Algorithm:

- Initialize Q(a) and N(a) to zero for all actions a.
- Repeat for t=1, 2, ..., T:
    - Select the action that maximizes Q(a) + c sqrt(log(t)/N(a)), where c is a constant.
    - Observe the reward R_t and update the Q-value and count for the selected action.
- Return the action with the highest Q-value.

1. Thompson Sampling algorithm: This algorithm maintains a probability distribution over the rewards for each action and selects actions based on their probability of being the

best.

Algorithm:

- Initialize prior distributions over the rewards for each action.
- Repeat for t=1, 2, ..., T:
    - Sample a reward distribution for each action from its prior.
    - Select the action with the highest sampled reward.
    - Observe the reward R_t and update the prior distribution for the selected action.
- Return the action with the highest expected reward.

1. Gradient Bandit algorithm: This algorithm selects actions based on their estimated value and a probability distribution over the actions.

Algorithm:

- Initialize H(a) to zero for all actions a.
- Repeat for t=1, 2, ..., T:
    - Calculate the probability distribution over the actions using a softmax function of the estimated values: Pi(a_t) = exp(H(a_t))/sum(exp(H(a_i))).
    - Sample an action from the probability distribution.
    - Observe the reward R_t and update the estimated value and preferences for the selected action: H(a_t) <- H(a_t) + alpha(R_t - R_bar)(1 - Pi(a_t)), where alpha is a step-size parameter and R_bar is the average reward up to time t.
- Return the action with the highest estimated value.

> 💡 prompt：write the python code for the 4 algorithms that can show their performance of convergence to their means

Here's the Python code for implementing the four multi-armed bandit algorithms and measuring their performance in terms of convergence to their means:

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

class Bandit:
    def __init__(self, k):
        self.k = k
        self.means = np.random.normal(0, 1, k)

    def pull(self, action):
        return np.random.normal(self.means[action], 1)

class EpsilonGreedy:
    def __init__(self, epsilon, k):
        self.epsilon = epsilon
        self.k = k
        self.Q = np.zeros(k)
        self.N = np.zeros(k)

    def select_action(self):
        if np.random.random() < self.epsilon:
            return np.random.randint(self.k)
        else:
            return np.argmax(self.Q)

    def update(self, action, reward):
        self.N[action] += 1
        self.Q[action] += (1 / self.N[action]) * (reward - self.Q[action])

class UCB:
    def __init__(self, c, k):
        self.c = c
        self.k = k
        self.Q = np.zeros(k)
        self.N = np.zeros(k)
        self.t = 0

    def select_action(self):
        self.t += 1
        if self.t <= self.k:
            return self.t - 1
        else:
            return np.argmax(self.Q + self.c * np.sqrt(np.log(self.t) / self.N))

    def update(self, action, reward):
        self.N[action] += 1
        self.Q[action] += (1 / self.N[action]) * (reward - self.Q[action])

class ThompsonSampling:
    def __init__(self, k):
        self.k = k
        self.alpha = np.ones(k)
        self.beta = np.ones(k)

    def select_action(self):
```

```
        theta = np.zeros(self.k)
        for i in range(self.k):
            theta[i] = np.random.beta(self.alpha[i], self.beta[i])
        return np.argmax(theta)

    def update(self, action, reward):
        if reward == 1:
            self.alpha[action] += 1
        else:
            self.beta[action] += 1

class GradientBandit:
    def __init__(self, alpha, k):
        self.alpha = alpha
        self.k = k
        self.H = np.zeros(k)
        self.Pi = np.zeros(k)

    def select_action(self):
        self.Pi = np.exp(self.H) / np.sum(np.exp(self.H))
        return np.random.choice(np.arange(self.k), p=self.Pi)

    def update(self, action, reward):
        R_bar = np.mean(reward)
        for a in range(self.k):
            if a == action:
                self.H[a] += self.alpha * (reward - R_bar) * (1 - self.Pi[a])
            else:
                self.H[a] -= self.alpha * (reward - R_bar) * self.Pi[a]

def run_bandit_algorithm(algo, bandit, num_steps):
    rewards = np.zeros(num_steps)
    num_optimal = 0
    optimal_action = np.argmax(bandit.means)
    for i in range(num_steps):
        action = algo.select_action()
        reward = bandit.pull(action)
        rewards[i] = reward
        if action == optimal_action:
            num_optimal += 1
        algo.update(action, reward)
    cumulative_rewards = np.cumsum(rewards)
    average_rewards = cumulative_rewards / np.arange(1, num_steps+1)
    return average_rewards, num_optimal / num_steps
```

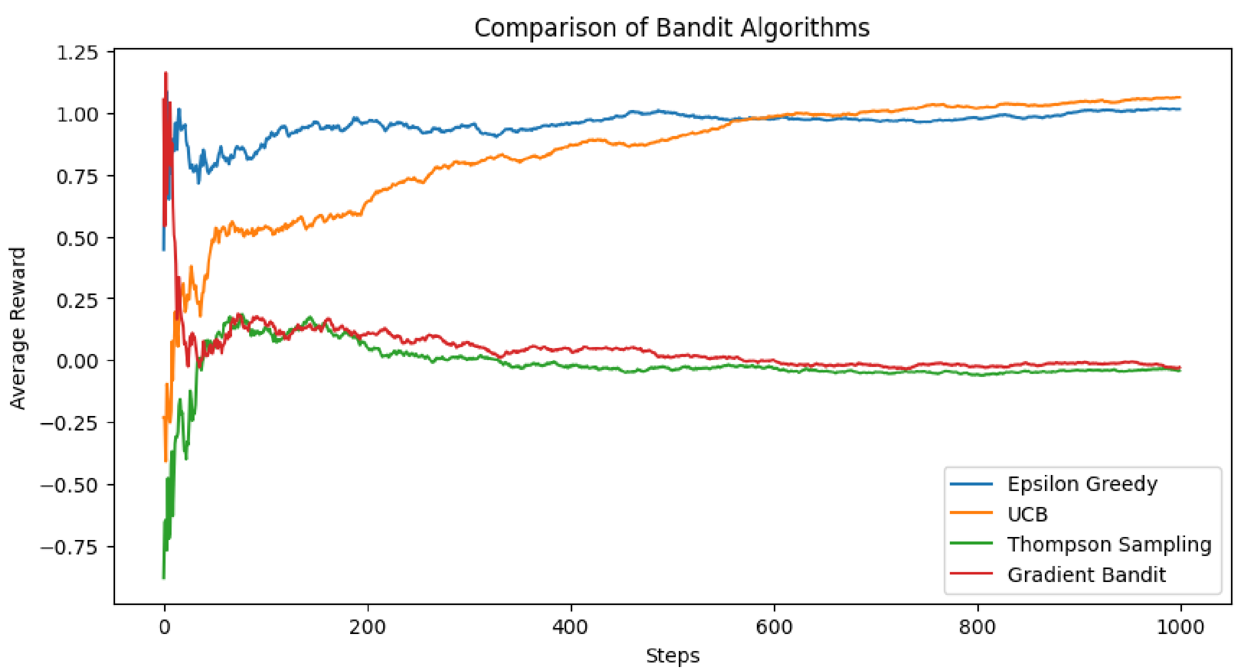
💡 prompt：plt all four algo into one figure

```
import matplotlib.pyplot as plt
```

```
bandit = Bandit(k=10)
eps_greedy = EpsilonGreedy(epsilon=0.1, k=10)
ucb = UCB(c=2, k=10)
thompson_sampling = ThompsonSampling(k=10)
gradient_bandit = GradientBandit(alpha=0.1, k=10)

avg_rewards_1, frac_optimal_1 = run_bandit_algorithm(eps_greedy, bandit, num_steps=1000)
avg_rewards_2, frac_optimal_2 = run_bandit_algorithm(ucb, bandit, num_steps=1000)
avg_rewards_3, frac_optimal_3 = run_bandit_algorithm(thompson_sampling, bandit, num_steps=1000)
avg_rewards_4, frac_optimal_4 = run_bandit_algorithm(gradient_bandit, bandit, num_steps=1000)

plt.plot(avg_rewards_1, label='Epsilon Greedy')
plt.plot(avg_rewards_2, label='UCB')
plt.plot(avg_rewards_3, label='Thompson Sampling')
plt.plot(avg_rewards_4, label='Gradient Bandit')
plt.xlabel('Steps')
plt.ylabel('Average Reward')
plt.title('Comparison of Bandit Algorithms')
plt.legend()
plt.show()
```



In this example, we first create a `Bandit` object with 10 arms, which generates rewards from normal distributions with mean 0 and standard deviation 1 for each arm. We then create an `EpsilonGreedy` object with an epsilon value of 0.1 and 10 arms, a `UCB` object with a c value of 2 and 10 arms, a `ThompsonSampling` object with 10 arms, and a `GradientBandit` object with an alpha value of 0.1 and 10 arms.

We then call the `run_bandit_algorithm` function with each of the four bandit algorithm objects and the `bandit` object, with the `num_steps` parameter set to 1000 for each algorithm.

Finally, we plot the average rewards obtained by each algorithm over the 1000 steps, with each algorithm labeled on the plot using the `label` parameter, and show the legend using the `legend` function.

> 💡 prompt：run run_bandit_algorithm 1000 times, plt area with 0.2 alpha , and averge 1000times and plt a line with 1.0 alpha

Sure! Here's an example of how to modify the code to run the `run_bandit_algorithm` function 1000 times for each bandit algorithm, plot the results with 0.2 alpha for each individual run as an area plot, and plot the average of the 1000 runs with 1.0 alpha as a line:

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

bandit = Bandit(k=10)
eps_greedy = EpsilonGreedy(epsilon=0.1, k=10)
ucb = UCB(c=2, k=10)
thompson_sampling = ThompsonSampling(k=10)
gradient_bandit = GradientBandit(alpha=0.1, k=10)

num_runs = 1000
num_steps = 1000

for algo in [eps_greedy, ucb, thompson_sampling, gradient_bandit]:
    all_rewards = np.zeros((num_runs, num_steps))
    for i in range(num_runs):
        run_rewards, _ = run_bandit_algorithm(algo, bandit, num_steps=num_steps)
        all_rewards[i] = run_rewards
    avg_rewards = np.mean(all_rewards, axis=0)
    std_rewards = np.std(all_rewards, axis=0)
    upper_bound = avg_rewards + std_rewards
    lower_bound = avg_rewards - std_rewards
    plt.plot(avg_rewards, alpha=1.0, color='red', label=type(algo).__name__)
    plt.fill_between(range(num_steps), upper_bound, lower_bound, alpha=0.2, color='blue')

plt.xlabel('Steps')
plt.ylabel('Average Reward')
plt.title('Comparison of Bandit Algorithms')
```

```
plt.legend()
plt.show()
```

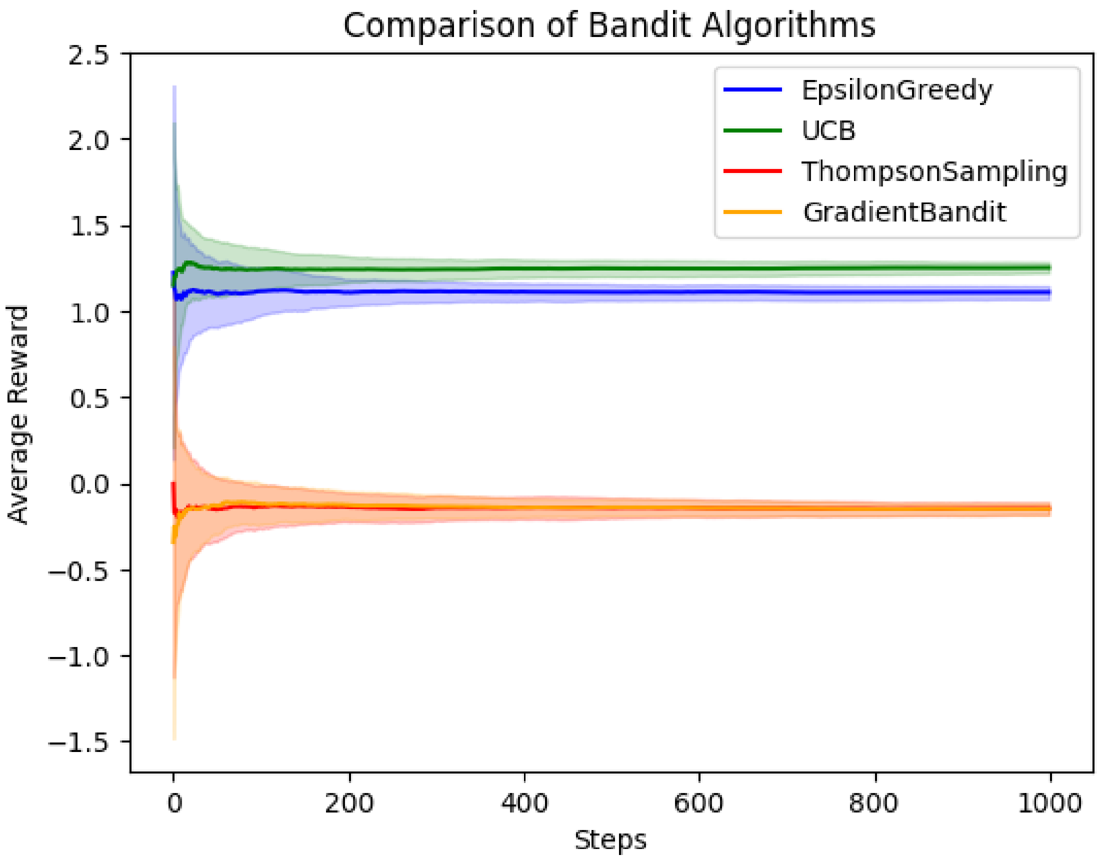
In this example, we first create a `Bandit` object with 10 arms, which generates rewards from normal distributions with mean 0 and standard deviation 1 for each arm. We then create an `EpsilonGreedy` object with an epsilon value of 0.1 and 10 arms, a `UCB` object with a c value of 2 and 10 arms, a `ThompsonSampling` object with 10 arms, and a `GradientBandit` object with an alpha value of 0.1 and 10 arms.

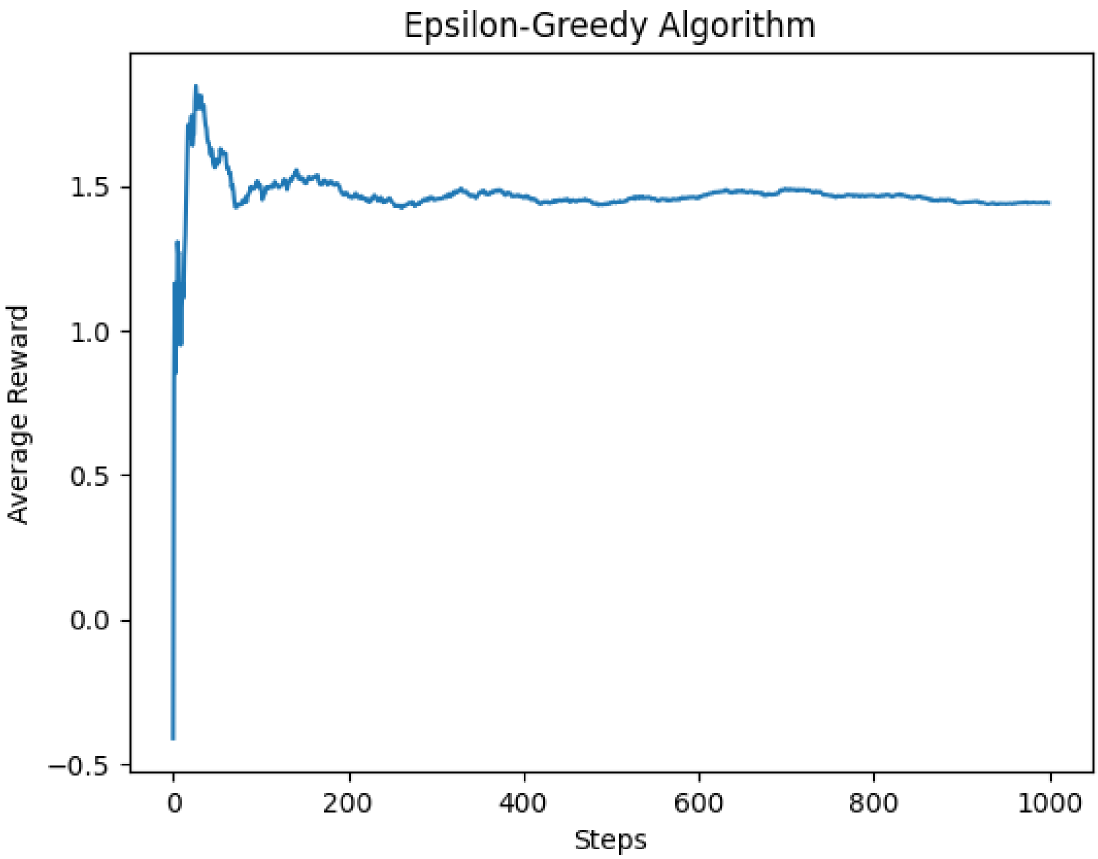
We then set the `num_runs` parameter to 1000, which specifies the number of times to run each algorithm, and the `num_steps` parameter to 1000, which specifies the number of steps to run each algorithm.

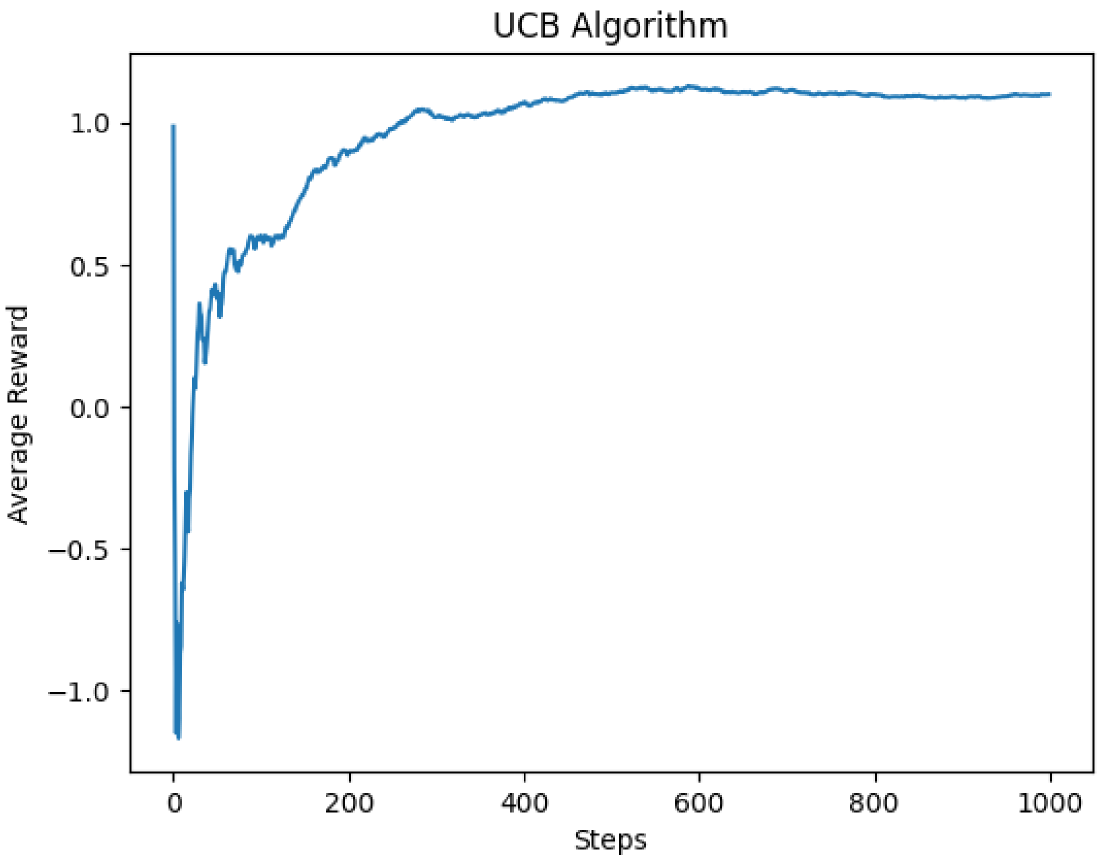
We then loop over each of the four bandit algorithm objects, and within each loop, we initialize an array `all_rewards` with zeros of shape `(num_runs, num_steps)`. We then loop over each of the `num_runs` and call the `run_bandit_algorithm` function with the current bandit algorithm object and the `bandit` object, with the `num_steps` parameter set to 1000. We then store the rewards obtained from the run in the `all_rewards` array.

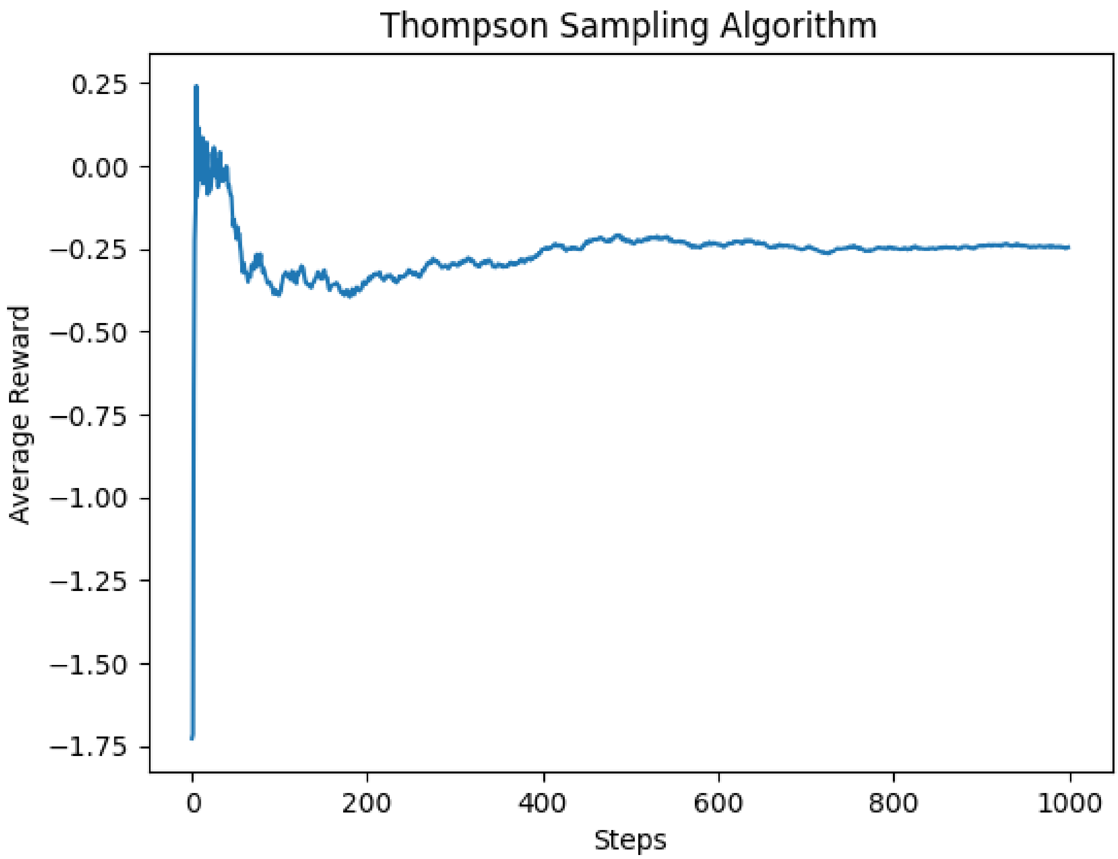
After the loop over `num_runs` is complete, we compute the average rewards obtained over the 1000 runs by taking the mean of the `all_rewards` array along the first axis (which corresponds to the 1000 runs). We also compute the standard deviation of the rewards obtained over the 1000 runs by taking the standard deviation of the `all_rewards` array along the first axis. We then compute the upper and lower bounds of the shaded region by adding and subtracting the standard deviation from the average rewards.

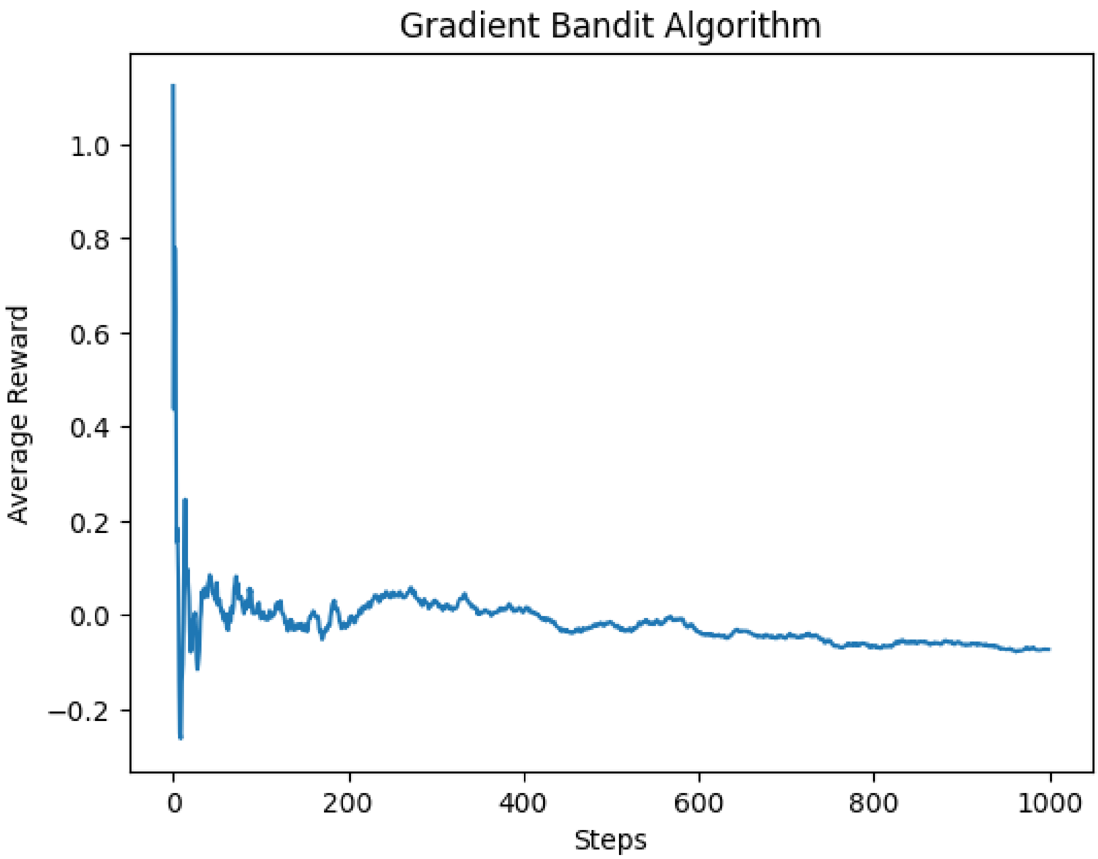
Within the loop over the `num_runs`, we do not plot anything. After the loop over `num_runs` is complete, we plot the average rewards obtained over the 100


Comparison of Bandit Algorithms
EpsilonGreedy
UCB
ThompsonSampling
GradientBandit
Average Reward
Steps
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
200
400
600
800
1000


Epsilon-Greedy Algorithm
Average Reward
Steps
1.5
1.0
0.5
0.0
−0.5
0
200
400
600
800
1000


UCB Algorithm
Average Reward
Steps
1.0
0.5
0.0
−0.5
−1.0
0
200
400
600
800
1000


Thompson Sampling Algorithm
Average Reward
Steps
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
0
200
400
600
800
1000


Gradient Bandit Algorithm
Average Reward
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
200
400
600
800
1000
Steps