assignment_3-2

May 5, 2024

1 Assignment 3: Reinforcement Learning

Goal: Get familiar with a reinforcement learning approach to solve multi-armed bandit problem.

We will implement a value-based reinforcement learning approach with two algorithm variations: epsilon-greedy (e-greedy) and upper confidence bound (UCB) algorithms and perform an analysis on their behavior.

Please answer the `Questions` and implement coding `Tasks` by filling **PLEASE FILL IN** sections. *Documentation* of your code is also important. You can find the grading scheme in implementation cells.

- Plagiarism is automatically checked and set to **0 points**
- It is allowed to learn from external resources but copying is not allowed. If you use any external resource, please cite them in the comments (e.g. `# source: https://...../` (see `fitness_function`))
- Use of generative AI to answer **ANY** part of the assignment is **strictly prohibited**, if any part of the assignment is found to be answered using generative AI, the question will be awarded **0 points**.

1.1 1. Introduction: Multi-Armed Bandit Problem

Imagine you are in a casino facing a row of slot machines, say there are 20 of them. Each slot machine is providing reward based on a certain probability distribution that is unknown to you.

This is your first time in this casino, thus you have no idea what to do next. You have just enough money to play for 100 times and each of these times, you can pick any machine you want and after "pulling"

What would you do?

The overall goal would of course be to find out the one that is providing the most reward, right? What should your algorithm be to get the most reward at the end?

1.2 2. Implementation

[57]:
```
%pip install "matplotlib>=3.7" "numpy>=1.25" "tqdm>=4.65" ipywidgets --user
```

```
Requirement already satisfied: matplotlib>=3.7 in
/opt/conda/lib/python3.11/site-packages (3.8.4)
```

```
Requirement already satisfied: numpy>=1.25 in /opt/conda/lib/python3.11/site-
packages (1.26.4)
Requirement already satisfied: tqdm>=4.65 in
/home/jovyan/.local/lib/python3.11/site-packages (4.66.4)
Requirement already satisfied: ipywidgets in /opt/conda/lib/python3.11/site-
packages (8.1.1)
Requirement already satisfied: contourpy>=1.0.1 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (1.1.1)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.11/site-
packages (from matplotlib>=3.7) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (4.43.1)
Requirement already satisfied: kiwisolver>=1.3.1 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (1.4.5)
Requirement already satisfied: packaging>=20.0 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (23.2)
Requirement already satisfied: pillow>=8 in /opt/conda/lib/python3.11/site-
packages (from matplotlib>=3.7) (10.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (2.8.2)
Requirement already satisfied: comm>=0.1.3 in /opt/conda/lib/python3.11/site-
packages (from ipywidgets) (0.1.4)
Requirement already satisfied: ipython>=6.1.0 in /opt/conda/lib/python3.11/site-
packages (from ipywidgets) (8.16.1)
Requirement already satisfied: traitlets>=4.3.1 in
/opt/conda/lib/python3.11/site-packages (from ipywidgets) (5.11.2)
Requirement already satisfied: widgetsnbextension~=4.0.9 in
/opt/conda/lib/python3.11/site-packages (from ipywidgets) (4.0.9)
Requirement already satisfied: jupyterlab-widgets~=3.0.9 in
/opt/conda/lib/python3.11/site-packages (from ipywidgets) (3.0.9)
Requirement already satisfied: backcall in /opt/conda/lib/python3.11/site-
packages (from ipython>=6.1.0->ipywidgets) (0.2.0)
Requirement already satisfied: decorator in /opt/conda/lib/python3.11/site-
packages (from ipython>=6.1.0->ipywidgets) (5.1.1)
Requirement already satisfied: jedi>=0.16 in /opt/conda/lib/python3.11/site-
packages (from ipython>=6.1.0->ipywidgets) (0.19.1)
Requirement already satisfied: matplotlib-inline in
/opt/conda/lib/python3.11/site-packages (from ipython>=6.1.0->ipywidgets)
(0.1.6)
Requirement already satisfied: pickleshare in /opt/conda/lib/python3.11/site-
packages (from ipython>=6.1.0->ipywidgets) (0.7.5)
Requirement already satisfied: prompt-toolkit!=3.0.37,<3.1.0,>=3.0.30 in
/opt/conda/lib/python3.11/site-packages (from ipython>=6.1.0->ipywidgets)
(3.0.39)
Requirement already satisfied: pygments>=2.4.0 in
/opt/conda/lib/python3.11/site-packages (from ipython>=6.1.0->ipywidgets)
```

```
(2.16.1)
Requirement already satisfied: stack-data in /opt/conda/lib/python3.11/site-
packages (from ipython>=6.1.0->ipywidgets) (0.6.2)
Requirement already satisfied: pexpect>4.3 in /opt/conda/lib/python3.11/site-
packages (from ipython>=6.1.0->ipywidgets) (4.8.0)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.11/site-
packages (from python-dateutil>=2.7->matplotlib>=3.7) (1.16.0)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in
/opt/conda/lib/python3.11/site-packages (from
jedi>=0.16->ipython>=6.1.0->ipywidgets) (0.8.3)
Requirement already satisfied: ptyprocess>=0.5 in
/opt/conda/lib/python3.11/site-packages (from
pexpect>4.3->ipython>=6.1.0->ipywidgets) (0.7.0)
Requirement already satisfied: wcwidth in /opt/conda/lib/python3.11/site-
packages (from prompt-
toolkit!=3.0.37,<3.1.0,>=3.0.30->ipython>=6.1.0->ipywidgets) (0.2.8)
Requirement already satisfied: executing>=1.2.0 in
/opt/conda/lib/python3.11/site-packages (from stack-
data->ipython>=6.1.0->ipywidgets) (1.2.0)
Requirement already satisfied: asttokens>=2.1.0 in
/opt/conda/lib/python3.11/site-packages (from stack-
data->ipython>=6.1.0->ipywidgets) (2.4.0)
Requirement already satisfied: pure-eval in /opt/conda/lib/python3.11/site-
packages (from stack-data->ipython>=6.1.0->ipywidgets) (0.2.2)
Note: you may need to restart the kernel to use updated packages.
```

[58]:
```
%pip install "matplotlib>=3.7" "numpy>=1.25" "tqdm>=4.65" --upgrade --user
```

```
Requirement already satisfied: matplotlib>=3.7 in
/opt/conda/lib/python3.11/site-packages (3.8.4)
Requirement already satisfied: numpy>=1.25 in /opt/conda/lib/python3.11/site-
packages (1.26.4)
Requirement already satisfied: tqdm>=4.65 in
/home/jovyan/.local/lib/python3.11/site-packages (4.66.4)
Requirement already satisfied: contourpy>=1.0.1 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (1.1.1)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.11/site-
packages (from matplotlib>=3.7) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (4.43.1)
Requirement already satisfied: kiwisolver>=1.3.1 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (1.4.5)
Requirement already satisfied: packaging>=20.0 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (23.2)
Requirement already satisfied: pillow>=8 in /opt/conda/lib/python3.11/site-
packages (from matplotlib>=3.7) (10.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (3.1.1)
```

```
Requirement already satisfied: python-dateutil>=2.7 in
/opt/conda/lib/python3.11/site-packages (from matplotlib>=3.7) (2.8.2)
Requirement already satisfied: six>=1.5 in /opt/conda/lib/python3.11/site-
packages (from python-dateutil>=2.7->matplotlib>=3.7) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
%matplotlib inline
```

[60]:
```
# First import the dependencies
import matplotlib.pyplot as plt
import numpy as np
from tqdm import trange
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import ranksums
from tqdm.notebook import trange
```

Question 1 (0-0.25-0.5pt): Please write down mathematical expressions arm selection for the e-greedy and UCB algorithms and discuss their parameters.

Answer:

Epsilon Greedy Algorithm

$$a_t = \begin{cases} \text{argmax}_a(Q_t(a)), & \text{if (rand} < 1-\epsilon), \\ \text{rand}(a), & \text{otherwise.} \end{cases}$$

- (Q_t(a)) represents the estimated value (mean reward) of taking action (a) as of the (t)-th time step. This value is typically updated based on the rewards received, reflecting the latest estimates of the action's value.

The parameter epsilon( ) determines the choice of the algorithm between exploiting and exploring. It is a probability value approximately between 0 and 1 by rand which provides a uniformly distributed random number between 0 and 1.

- With Probability 1− (Exploitation): The idea of exploitation is to maximise the reward based on past experiences with the estimated values that has been provided by actions. At this step algorithm selects the arm a that has the maximum estimated value Q_t(a) at the t-th time. The action that maximises the Q value is determined by argmax.
- With Probability (Exploration): By this the algorithm selects an arm at random, regardless of Q_t(a). The main purpose of the exploration is to prevent the algorithm from becoming too focused on the past maximisations, potentially missing out on better options that haven't been tried yet. Exploring allows algorithm to potentially discover other actions that could provide higher rewards than currently believed/estimated.
- Larger : increases exploration, potentially discovering better long-term strategies but at the cost of immediate performance. Could cause long-term consistency issues for algorithm's stability.
- Smaller : focuses more on exploiting the current best-known action, which might lead to suboptimal choices if not all actions have been sufficiently explored.

UCB Algorithm

The main idea of UCB algorithm is to keep the balance between exploration and exploitation to find the max reward through the trials.

$$A_t = \underset{a}{\operatorname{argmax}} \left(Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right)$$

- Estimated value Q_t(a) is the mean reward that received from action a thought the time t. It represents the algorithm's exploitation part, which is likely to select actions with a high historical performance.
- $$Exploration\ Term : c\sqrt{\frac{\ln t}{N_t(a)}}$$
- Numerator ln t: In each action the number for this part gets bigger but over time it becomes slower. It still explores, but with less urgency, because it gradually becomes more confident about which actions are best based on past experiences. This in a way ensures that exploration does not dominate the decision-making process constantly for the algorithm.
- Denominator N_t(a): This counts how many times a particular action a, has been selected up to the time t - 1. If an action hasn't been chosen many times, this part of the formula makes the algorithm more likely to try it again(increasing the exploration term). The idea here is to give less frequent actions more chances to be tested. Exploration term increases when the denominator is small and it decreases when the denaminator is growing.
- Parameter c: higher c provides more exploration, potentially discovering actions that has greater rewards than currently ones. A lower c focuses on exploiting known high-reward actions.

Task 1 (3 pt): Please implement the e-greedy and UCB algorithms in the code given below.

[61]:
```
##################################
# Grading
# 0 pts if the code does not work, code works but it is fundamentally incorrect
# 0.75 pts if the code works but some functions are incorrect and it is badly␣
 ↪explained
# 1.5 pts if the code works but some functions are incorrect but it is␣
 ↪explained well
# 2.25 pts if the code works very well aligned with the task without any␣
 ↪mistakes, but it is badly explained
# 3 pts if the code works very well aligned with the task without any mistakes,␣
 ↪and it is well explained
#####################################################################


# ============ PLEASE DO NOT CHANGE ============ #
```

```python
def initialize(n_arms):
    rng = np.random.default_rng()
    R = rng.uniform(low=0.45, high=0.55, size=n_arms)
    R[rng.integers(n_arms)] = 0.9
    # return actual mean of the reward probabilities
    return R


# ================================================ #


# the epsilon-greedy algorithm (ignore kwargs)
def e_greedy(Q, epsilon, **kwargs):
    ############ PLEASE FILL IN ###############

    # if generated number 0 to 1 is less than probability epsilon, exploration
    if np.random.rand() < epsilon: #10% or 20% chance to explore.
        # Exploration: choose a random arm
        selection = np.random.randint(len(Q))
    else:
        # Exploitation: choose the best known arm
        selection = np.argmax(Q)

    ##########################################
    return selection


# The upper confidence bound algorithm (ignore kwargs)
def UCB(Q, selection_counter, t, **kwargs):
    C = 0.5  # Parameter (keep it constant)
    ############ PLEASE FILL IN ###############

    # Avoid division by zero error by adding 1 to all counts
    adjusted_counts = selection_counter + 1

    # Calculate the UCB for each arm
    upper_confidence_bounds = Q + C * np.sqrt(np.log(t + 1) / adjusted_counts)

    # Select the arm with the highest UCB
    selection = np.argmax(upper_confidence_bounds)

    ##########################################
    return selection


def MAB(
```

```
    trials,  # total number of arm pulls
    n_arms,  # number of arms to pull
    epsilon,  # exploration parameter for the epsilon-greedy algorithm
    alpha,  # learning rate for updating Q-values
    init,  # initial starting value of the Q-values
    algorithm,  # the type of update: e_greedy or UCB
):
    # ============ PLEASE DO NOT CHANGE ============ #
    # initialization of the reward distributions unknown to the player
    R = initialize(n_arms)
    cumulative_reward_trend = np.zeros(trials)
    selection_trend = np.zeros(trials)
    reward_trend = np.zeros(trials)
    cumulative_reward = 0

    # initialize counter of selection for each arms
    selection_counter = np.zeros(n_arms)

    # initialize initial estimates of rewards
    Q = np.ones(n_arms) * init
    # ============================================== #

    for i in trange(trials, leave=False):
        # ============ PLEASE DO NOT CHANGE ============ #
        # select an arm to pull based on reward estimates and other
        kwargs = {
            "Q": Q,
            "epsilon": epsilon,
            "selection_counter": selection_counter,
            "t": i,
        }
        selection = algorithm(**kwargs)
        reward = np.random.normal(R[selection], 0.01)
        # ============================================== #

        ############ PLEASE FILL IN ##############
        # Update Q values
        # Q ?
        # Update Q values after getting the reward
        #T (reward - Q[selection]): the error or difference between the␣
↪received reward and the current estimate.
        #allows algorithm to learn
        Q[selection] = Q[selection] + alpha * (reward - Q[selection])

        ##########################################

        # ============ PLEASE DO NOT CHANGE ============ #
```

```
        reward_trend[i] = reward
        selection_trend[i] = selection
        selection_counter[selection] += 1
        cumulative_reward += reward
        cumulative_reward_trend[i] = cumulative_reward
        # ================================================ #

    return reward_trend
```

Question 2 (0-0.25-0.5pt): Please explain the concept of exploration and exploitation in the context epsilon-greedy and UCB algorithm contexts. How does the epsilon-greedy algorithm balance exploration and exploitation?

Answer:

Q values are the estimated values for the expected reward in each trial to be received based on the chosen actions.

Epsilon-Greedy Algorithm: - The balance between exploration and exploitation kept using parameter .

Exploration (Probability): (epsilon) is a predefined probability that dictates how often the algorithm will explore rather than exploit. Thats why, if we look at the code, the function handles less frequent case (exploration) first based on a conditional probability check.

```
if np.random.rand() < epsilon:
        # Exploration: choose a random arm
        selection = np.random.randint(len(Q))
```

In this part the algorithm explores other options by choosing random arm. This potantially could lead better options. Exploring allows algorithm to potentially discover other actions which might provide higher rewards than currently estimated (by Q_t(a)).

Exploitation (Probability 1−): If the random number is not less than , the algorithm defaults to exploitation. In this part of the code, algorithm chooses the arm with the highest estimated reward.

```
else:
        # Exploitation: choose the best known arm
        selection = np.argmax(Q)
```

UCB algorithm:

- Q is for exploitation
- C * np.sqrt(np.log(t + 1) / adjusted_counts) of the code is for exploration The algorithm synthesizes both into one decision metric, which is returned as selection in this case.

```
# Calculate the UCB for each arm
upper_confidence_bounds = Q + C * np.sqrt(np.log(t + 1) / adjusted_counts)

# Select the arm with the highest UCB
selection = np.argmax(upper_confidence_bounds)
```

- Exploitation: If an arm has been selected multiple times and provides high rewards consistently, then Q_t(a) will be quite high. And if N_t(a), the number of times arm a has been selected, is also high, the exploration term **C * np.sqrt(np.log(t + 1) / adjusted_counts)** becomes smaller, because the denominator is large(considering the nature of the root fucntion). Thus, for well-performing, frequently chosen arms, the exploitation component Q_t(a) will likely dominate the UCB value.
- Exploration: Conversely, for an arm that hasn't been chosen often, N_t(a) is low, which makes the exploration term larger. Even if Q_t(a) is not very high (potentially due to less data from fewer selections), the large exploration term can make this arm's overall UCB value high. This leads the algorithm to choose lesser-known options to gather more information, so exploration.

1.3 3. Algorithm Analysis

[62]:
```
# ============ PLEASE DO NOT REMOVE ============ #
def plot_experiments(experiment1, experiment2, labels):
    experiment1 = np.array(experiment1)
    experiment1_std = np.std(experiment1, axis=0)
    experiment1_mean = np.mean(experiment1, axis=0)

    experiment2 = np.array(experiment2)
    experiment2_std = np.std(experiment2, axis=0)
    experiment2_mean = np.mean(experiment2, axis=0)

    mean = [experiment1_mean, experiment2_mean]
    std = [experiment1_std, experiment2_std]

    plt.figure(figsize=(12, 6))
    y_values = np.arange(0, len(mean[0]))
    for i in range(len(mean)):
        plt.plot(y_values, mean[i], label=labels[i])
        plt.fill_between(y_values, mean[i] + std[i], mean[i] - std[i], alpha=0.
 ↪2)

    plt.xlabel("Arm pulls (trials)")
    plt.ylabel("Average reward of 20 runs")

    plt.legend()
    plt.show()


# ============================================== #
```

1.4 2. Comparison of e-greedy and UCB algorithms

Running the code below will launch all the experiments that we would like to plot and perform analysis on.

[70]:
```
experiment1 = []  # epsilon greedy epsilon: 0.1 initial Q: 0
experiment2 = []  # epsilon greedy epsilon: 0.1 initial Q: 1
experiment3 = []  # epsilon greedy epsilon: 0.2 initial Q: 0

experiment4 = []  # UCB initial Q: 0
experiment5 = []  # UCB initial Q: 1

for _ in trange(20):
    # MAB(trials, n_arms, epsilon, alpha, init, algorithm)
    experiment1.append(MAB(5_000, 20, 0.1, 0.1, 0, e_greedy))
    experiment2.append(MAB(5_000, 20, 0.1, 0.1, 1, e_greedy))
    experiment3.append(MAB(5_000, 20, 0.2, 0.1, 0, e_greedy))

    experiment4.append(MAB(5_000, 20, 0.1, 0.1, 0, UCB))
    experiment5.append(MAB(5_000, 20, 0.1, 0.1, 1, UCB))
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
  0%|          | 0/5000 [00:00<?, ?it/s]
```

```
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
```

```
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
```

```
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
0%|          | 0/5000 [00:00<?, ?it/s]
```

Question 3 (0-0.5-1pt): Plot and compare the average rewards for e-greedy and UCB algorithms for initial Q: 0 in different period of the process.

Please discuss in each phase of the process, which algorithm worked better and why?

Answer: As can be seen from the graph, the Epsilon-Greedy algorithm grows slower than the UCB algorithm at the beginning. This is because, in the first trials, not all the Q values are explored, which means that over time, and with more exploration and new estimations for Q, the Epsilon-Greedy algorithm gradually finds the maximum rewards. This is due to the nature of the Epsilon-Greedy algorithm. If we look at the code, we see that the exploration probability is less than the exploitation.

```
# if generated number 0 to 1 is less than probability epsilon, exploration
    if np.random.rand() < epsilon: #10% chance to explore.
        # Exploration: choose a random arm
        selection = np.random.randint(len(Q))
    else:
        # Exploitation: choose the best known arm
        selection = np.argmax(Q)
```

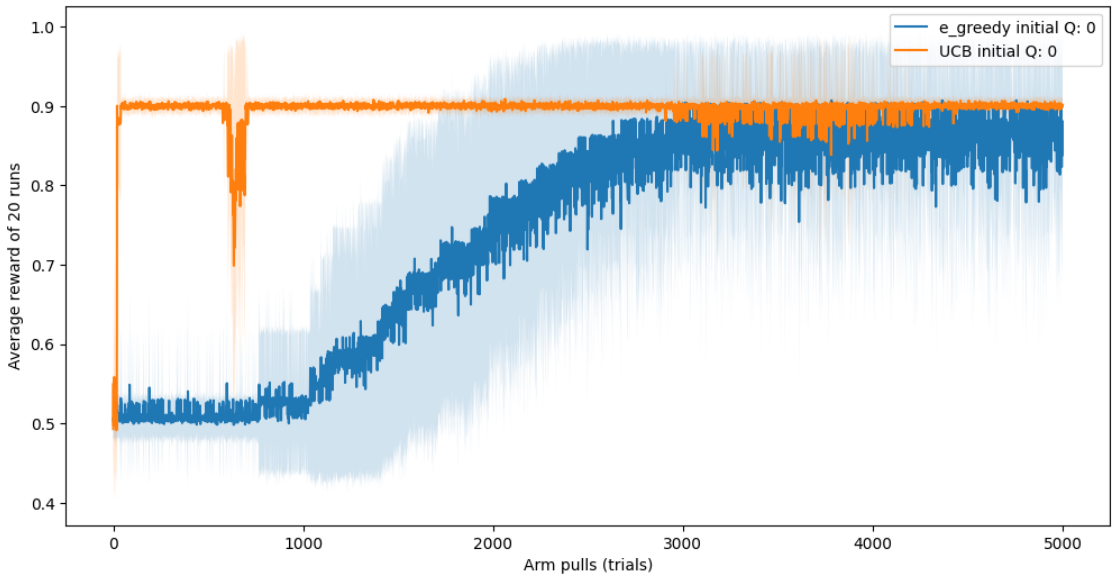
Considering the given case; epsilon is 0.1, which means there is only a 10% chance for exploration. Since 90% is for exploitation, this means 90% of the time, the algorithm is choosing the maximum Q value as an estimation for the reward that has already been explored. Therefore, in the first trials, the algorithm tries to explore other Q values with the recieved rewards in each trial and updates the Q values according to that. The chance of having exploration in each trial is low; and it should explore more to find the max and that's why it takes a lot of trials to find the max reward for e-greed algorithm.

The UCB algorithm behaves faster and reaches the maximum average reward in the initial trials compared to the Epsilon-Greedy algorithm. Due to the nature of the UCB formula, when the N_t(a) value is low, the denominator is also low, which indicates that the algorithm will be more prone to exploration. In the Python code, the selection_counter starts at 0 so N_t(a) value is also 0 (I added 1 to prevent division by 0 error) which increases throughout the 20 runs. For this reason, when the initial Q values are set to 0 (so no prone for algorithm to exploitation), the algorithm is more inclined to explore. This situation is reflected in the graph, showing that the algorithm quickly reaches the maximum reward in the first few trials.

Thus UCB algorithm can be considered as better since it has a faster and consistent performance.

[64]:
```
label = ["e_greedy initial Q: 0", "UCB initial Q: 0"]

plot_experiments(
    ############ PLEASE FILL IN ##############
    experiment1,
    experiment4,
    ##########################################
    label,
)
```



Question 4 (0-0.5-1pt): Plot and compare the average rewards for e-greedy and UCB algorithms for initial Q: 1 in different period of the process.

Please discuss in each phase of the process, which algorithm worked better and why?

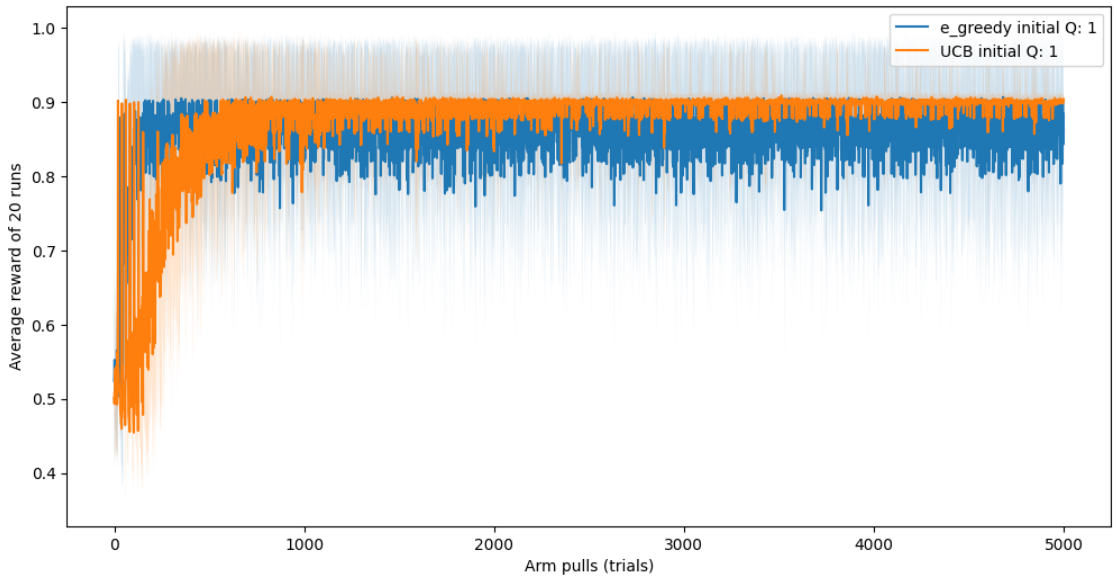
Answer:

For e_greedy initial Q1, algorithm assumes that every action have reward of 1. Because of the initial optimistic estimation of the Q values, the algorithm is encouraged to exploitation, since it already has a super higher estimaiton for reward. During exploitation phases, it will tend to select actions based on these high initial Q-values. During the learning process and the update of Q; the formula Q = Q + alpha * (reward - Q) used. Due to the nature of this formula, for instance even if the received reward is 0.2, the Q-value is updated to 0.98 after one trial, assuming an initial Q of 1 and of 0.1. This adjustment results in the Q-value quickly reflecting more accurate estimates of the actual rewards, which helps the algorithm find the optimal or better-performing actions more rapidly.

Due to the nature of the UCB formula, when the N_t(a) value is low, the denominator is also low, which indicates that the algorithm will be more prone to exploration. For this reason, in the first hundreds of trials the algorithm will explore to find the max reward within each trial. Moreover, because of the optimistic initial values, the algorithm temporary decreased at first. The algorithm might initially select arms based on the high Q-values, assuming they will provide large rewards. If these arms encounters lower-than-expected rewards, the Q-values will be adjusted downward, which might lead to a temporary decrease(as can be seen from the figure) in the observed average reward before the stabilisation of the algorithm for balance between exploration and exploitation.

Although e_greedy seems faster to find the maximum reward in the beginning, it could have potatial stability problem. Q=1 for e_greedy can lead overestimations of action values later on. This can be seen from the graph too that blue line goes constanty between aproximately 0.8 to 0.9, which seems less stable. If the actual rewards are significantly lower, the algorithm may spend initial trials "correcting" these overestimations, which can introduce volatility in performance as Q-values may oscillate before stabilising. On the other hand, as can be seen from the graph, UCB is more consistent in the later trials. It is more balanced between exploration and exploitation and it has potantial to avoid overestimations, which also is an efficient learner. This is because after 1000(aproximitely) and gathered informations from the trials algorithm focuses on explitation in place of exploration. And this provides a stability for max reward. Thus I believe in this case UCB is better.

[65]:
```
label = ["e_greedy initial Q: 1", "UCB initial Q: 1"]
plot_experiments(
    ############ PLEASE FILL IN ##############
    experiment2,
    experiment5,
    ##########################################
    label,
)
```



Question 5 (0-0.5-1pt): Plot and compare the average rewards for e-greedy for initial Q: 0 for epsilon values 0.1 and 0.2 in different period of the process.

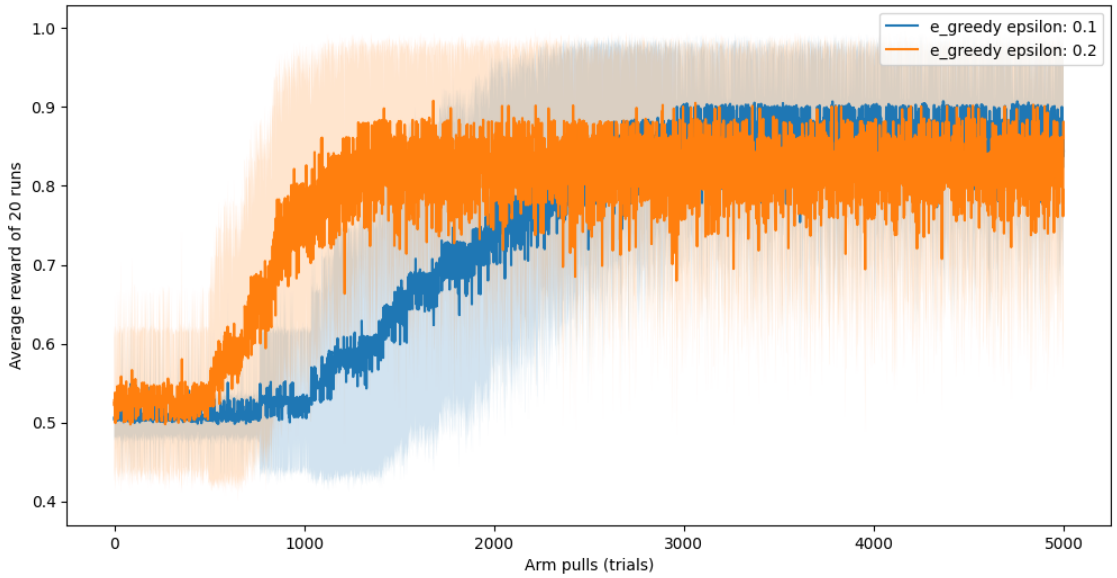
Please discuss in each phase of the process, which algorithm worked better and why?

Answer: The main difference between these two experiments is the difference in exploration chances; 10% and 20% . Given that a higher chance for exploration can lead to quicker discovery of the optimal actions, which we can see with the orange line, Epsilon-0.2, that shows better performance in the initial phase than the 0.1 version. In contrast, the algorithm with a 10% chance for exploration requires more trials to achieve the maximum reward. The less frequent exploration results in a slower ascent to the highest average reward. This is due to the inherent nature of the e-greedy algorithm; the 20% exploration rate inherently provides more opportunities to eexplore the optimal action earlier in the process.

However, higher exploration rates has potential to lead less consistent exploitation of the best-known action over time. As the algorithm continues(lets say there is more than 5000 trials) and gathers more informations, the possibility of algorithm to make suboptimal choices higher. This might lead to a lower average reward in the long run compared to an algorithm with a more balanced exploration-exploitation ratio. Additionaly, as can be seen from the orange line that compared to 10%, it is less stable and 10% reaches the maximum reward fist even tho it started slower.

Thus even tho Epsilon-0.2 algorithm performs better than the Epsilon-0.1 in the early trials, it is still important to consider the possible trade-offs in long-term performance driven on by continuous exploration due to the high epsilon probability rate (20%). So as can be seen from the figure, around 3700th trial the 0.1 received the maximum reward even tho its slower progress. Therefore, this may show us that the long term consistency of 0.1 is better than 0.2; so 0.1 is much more stable and consistent. Thus I believe 0.1 one can be considered better.

[66]:
```
label = ["e_greedy epsilon: 0.1", "e_greedy epsilon: 0.2"]
plot_experiments(
    ########### PLEASE FILL IN ##############
    experiment1,
    experiment3,
    ########################################
    label,
)
```



Question 6 (0-0.5-1pt): Plot and compare the average rewards for e-greedy for epsilon = 0.1 initial Q: 0 and 1 in different period of the process.

Please discuss in each phase of the process, which algorithm worked better and why?

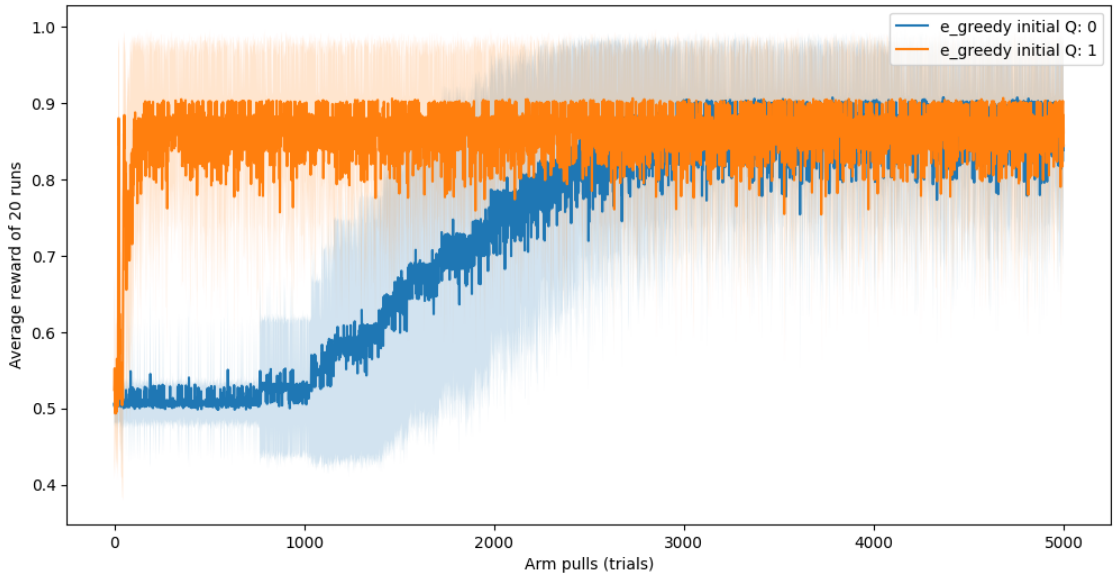
Answer:

Q values are the estimated values for the expected reward in each trial to be received based on the chosen actions. For Q=1, algorithm assumes that every action have reward of 1. Because of the initial optimistic estimation of the Q values, the algorithm is encouraged to exploitation, since it already has a super higher estimaiton for reward. During exploitation phases, it will tend to select actions based on these high initial Q-values. For the learning and the update of Q; the formula Q = Q + alpha * (reward - Q) used. Due to the nature of this formula, even if the received reward is 0.2, the Q-value is updated to 0.98 after one trial, assuming an initial Q of 1 and of 0.1. This adjustment results in the Q-value quickly reflecting more accurate estimation of the actual rewards, which helps the algorithm find the optimal or better-performing actions more rapidly.

On the other hand, for Q = 0, algorithm starts the estimation of reward at 0 and explore for exploitation to update Q and find the optimum. However, in this case algorithm learns slower than the other algorithm where Q=1.

Although e_greedy initial Q1 seems to be faster than the e_greedy initial Q0 for finding the max rewards within the given trials, the consistency of the algorithm might not be stable comparing to the slower one in the next thousands of trials. Since Q1 starts with an optimistic initial value, this can lead overestimations of action values later on. If the actual rewards are significantly lower, the algorithm may spend initial trials "correcting" these overestimations, which can introduce volatility in performance as Q-values may oscillate before stabilising. This also means that it could be more sensitive to the actual reward distributions. So we can say e_greedy initial Q1 is better than e_greedy initial Q0 to find the optimum reward in terms of speed. However, for the long term consistency e_greedy initial Q0 may be better.

[67]:
```
label = ["e_greedy initial Q: 0", "e_greedy initial Q: 1"]
plot_experiments(
    ########### PLEASE FILL IN ##############
    experiment1,
    experiment2,
    ########################################
    label,
)
```



Question 7 (0-0.5-1pt): Plot and compare the average rewards for UCB initial Q: 0 and 1 in different period of the process.

Please discuss in each phase of the process, which algorithm worked better and why?

Answer:

Description from previous questions:

- Due to the nature of the UCB formula, when the N_t(a) value is low, the denominator is also

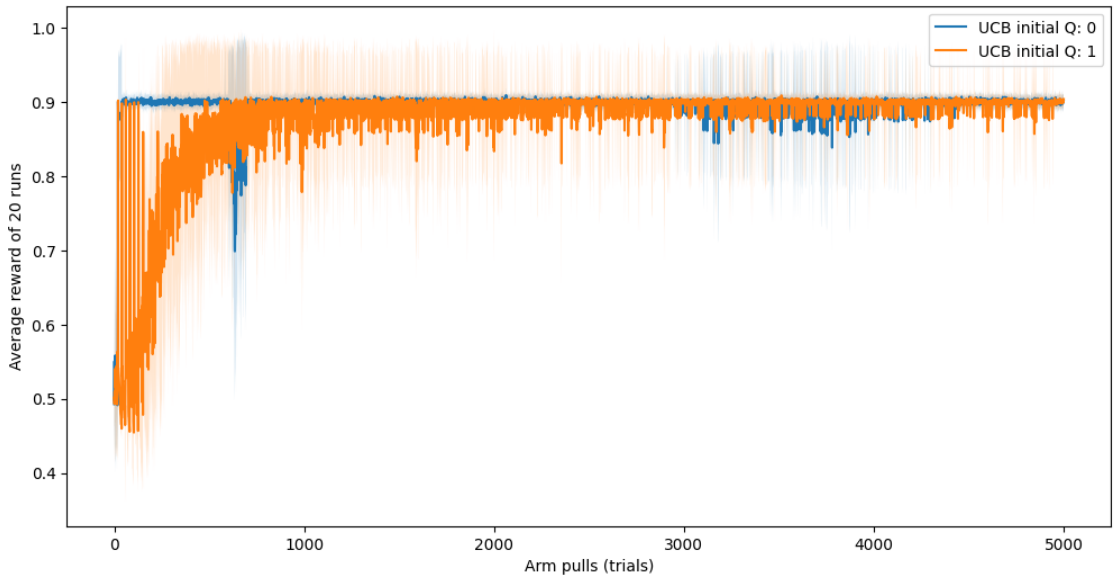
low, which indicates that the algorithm will be more prone to exploration. For this reason, in the first hundreds of trials the algorithm will explore even though Q=1. Moreover, because of the optimistic initial values, the algorithm temporary decreased at first. The algorithm might initially select arms based on the high Q-values, assuming they will provide large rewards. If these arms encounters lower-than-expected rewards, the Q-values will be adjusted downward, which might lead to a temporary decrease(as can be seen from the figure, between 90-200th trials) in the observed average reward before the stabilisation of the algorithm for balance between exploration and exploitation.

- The UCB Q0 behaves faster and reaches the maximum average reward within the initial trials compared to the Q1 algorithm. Due to the nature of the UCB formula, when the N_t(a) value is low, the denominator is also low, which indicates that the algorithm will be more prone to exploration. In the Python code, the selection_counter starts at 0 so N_t(a) value is also 0 (I added 1 to prevent division by 0 error) which increases throughout the 20 runs. For this reason, when the initial Q values are set to 0 (so no prone for algorithm to exploitation), the algorithm is more inclined to explore. This situation is reflected in the graph, showing that the algorithm quickly reaches the maximum reward in the first few trials.

Conclusion

UCB Q0 demonstrates a faster approach to reinforcement learning with the rapid exploration for finding max reward, which leads to a more informed exploitation strategy within the trials. For UCB Q1; it initially starts the process with an optimistic estimate, which potentially lead algorithm to make early exploitation of suboptimal choices (as can be seen from the figure too that orange line initially is not stable). So the algorithm for Q1 first has to "correct" the optimum rewards within the initial trials and this could potentially delay the convergence to optimal actions. Moreover, stability of Q0 for the long-term is more consistent compared to Q1. I believe Q0 better in finding the optimum reward faster and could potentially perform better for long-term.

[68]:
```
label = ["UCB initial Q: 0", "UCB initial Q: 1"]
plot_experiments(
    ############ PLEASE FILL IN ##############
    experiment4,
    experiment5,
    ##########################################
    label,
)
```



1.5 3. Final remarks

Question 8 (0-0.5-1pt): Based on the all plots and analysis, please plot the best and worst performing algorithms and discuss the comparison? Discuss why that may be the case.

Answer: I believe the best-performing algorithm is UCB with Q=0. The UCB algorithm effectively balances exploration and exploitation, which contributes to its consistent performance around the maximum average reward, as evidenced by the graph. Furthermore, it reaches the maximum reward more rapidly compared to other experiments. Although a minor overestimation occurs approximately between the 700-900 trials, likely due to a transient imbalance in the exploration-exploitation trade-off, it quickly stabilizes thereafter.

Generally, UCB is considered more efficient than -greedy. The -greedy algorithm with =0.2 allocates a higher probability to exploration at the expense of exploitation. This approach challenges the algorithm's ability to reach and maintain maximum reward, resulting in a less stable reward trajectory. As shown in the graph, it takes more trials for this algorithm to reach maximum reward, and its performance fluctuates significantly, often varying between 0.9 and 0.7 rewards. This results in a less efficient learning curve and overall lower rewards.

In reinforcement learning tasks, the balance between exploration and exploitation can significantly influence the performance of the algorithm. Which in this case their way of handling the balance influences the efficiency. Therefore, UCB with Q=0 could be considered as best performing algorithm in this experiment, while -greedy with =0.2 is may considered the worst.

[69]:
```
# PLOT THE BEST AND WORST PERFORMING ALGORITHMS AND COMPARE
label = [
    ############ PLEASE FILL IN ##############
    "Best performing: UCB initial Q: 0",
```

```
    "Worst performing: e_greedy epsilon: 0.2 initial Q: 0"
    ########################################
]

plot_experiments(

    ########### PLEASE FILL IN ##############
    experiment4,
    experiment3,
    ########################################
    label,
)
```