# bandit_answers

February 13, 2026

# 1 Bandit, Exploration and Exploitation

The goal of this exercise is to implement a simple bandit algorithm and test it on a simple environment.

We will first start by understanding the problem.

## 1.1 1. Understanding the Bandit Problem

[57]:
```
import gymnasium as gym
import buffalo_gym

env = gym.make("Buffalo-v0", arms=3)
obs = env.reset()
count = 0
while count < 10:
    action = env.action_space.sample()
    obs, reward, terminated, truncated, info = env.step(action)
    print(f"Action: {action} - Reward: {reward}")
    count += 1
env.close()
```

```
Action: 0 - Reward: 0.198054005671376
Action: 1 - Reward: 4.602138457864179
Action: 2 - Reward: 11.17269235642712
Action: 0 - Reward: 1.8046242540316735
Action: 0 - Reward: 0.7713591800501506
Action: 2 - Reward: 10.589618755163713
Action: 2 - Reward: 10.54969895143081
Action: 1 - Reward: 3.401013191620456
Action: 0 - Reward: 2.9007831899537475
Action: 2 - Reward: 9.718329363803246
```

Answer the following questions: 1. What the code is doing? 2. What is the best option to take? 3. What is the expected reward of taking the best option?

## 1.2 2. Implementing an Incremental Update Rule

Complete the function `incremental_update` to implement the incremental update rule for the action-value estimates.

[75]:
```python
def incremental_update(Q, Times,action, reward):
    """
    Update the action-value estimate Q for the given action and reward using an␣
 ↪incremental update rule.

    Parameters:
    Q (list): A list of action-value estimates for each action.
    Times (list): A list of counts of how many times each action has been taken.
    action (int): The index of the action taken.
    reward (float): The reward received after taking the action.

    Returns:
    list: Updated list of action-value estimates.
    """

    Q[action] = Q[action] + (1.0 / Times[action]) * (reward - Q[action])

    return Q
```

And execute the following code to test your implementation:

[76]:
```python
import gymnasium as gym
import buffalo_gym

arms = 10
Q = [0.0 for _ in range(arms)]
Times = [0 for _ in range(arms)]
env = gym.make("Buffalo-v0", arms=arms)
obs = env.reset()
done = False
while not done:
    action = env.action_space.sample()
    obs, reward, terminated, truncated, info = env.step(action)
    #print(f"Action: {action} - Reward: {reward}")
    Times[action] += 1
    Q = incremental_update(Q, Times, action, reward)
    done = terminated or truncated
env.close()

print("Final action-value estimates:", Q)
```

```
Final action-value estimates: [2.829177745503516, 2.7730682439389334,
2.249581901883098, 3.4030128761428493, 4.157544156550374, 1.473789935499445,
1.7553003571190906, 1.994341176742099, 9.958102806132988, 4.90469549397031]
```

Questions: 1. Which is the best action? 2. What is the expected reward of taking the best action?

## 1.3 3. Greedy Action Selection

Now that we have implemented the incremental update rule, we can implement a greedy action selection strategy. Complete the function `greedy_action_selection` to implement a greedy action selection strategy. This function will be replace the instruction `action = env.action_space.sample()` in the code above.

[77]:
```
import numpy as np

# We don't want to use the argmax function from numpy because it doesn't break␣
↪ties randomly.
# We want to implement our own version of argmax that breaks ties randomly.

def argmax(q_values):
    """
    Takes in a list of q_values and returns the index of the item
    with the highest value. Breaks ties randomly.
    returns: int - the index of the highest value in q_values
    """
    top_value = float("-inf")
    ties = []

    for i in range(len(q_values)):
        # if a value in q_values is greater than the highest value update top␣
↪and reset ties to zero
        # if a value is equal to top value add the index to ties

        if q_values[i] > top_value:
            ties = []
            top_value = q_values[i]
            ties.append(i)
        elif q_values[i] == top_value:
            top_value = q_values[i]
            ties.append(i)

    # return a random selection from ties.
    return np.random.choice(ties)
```

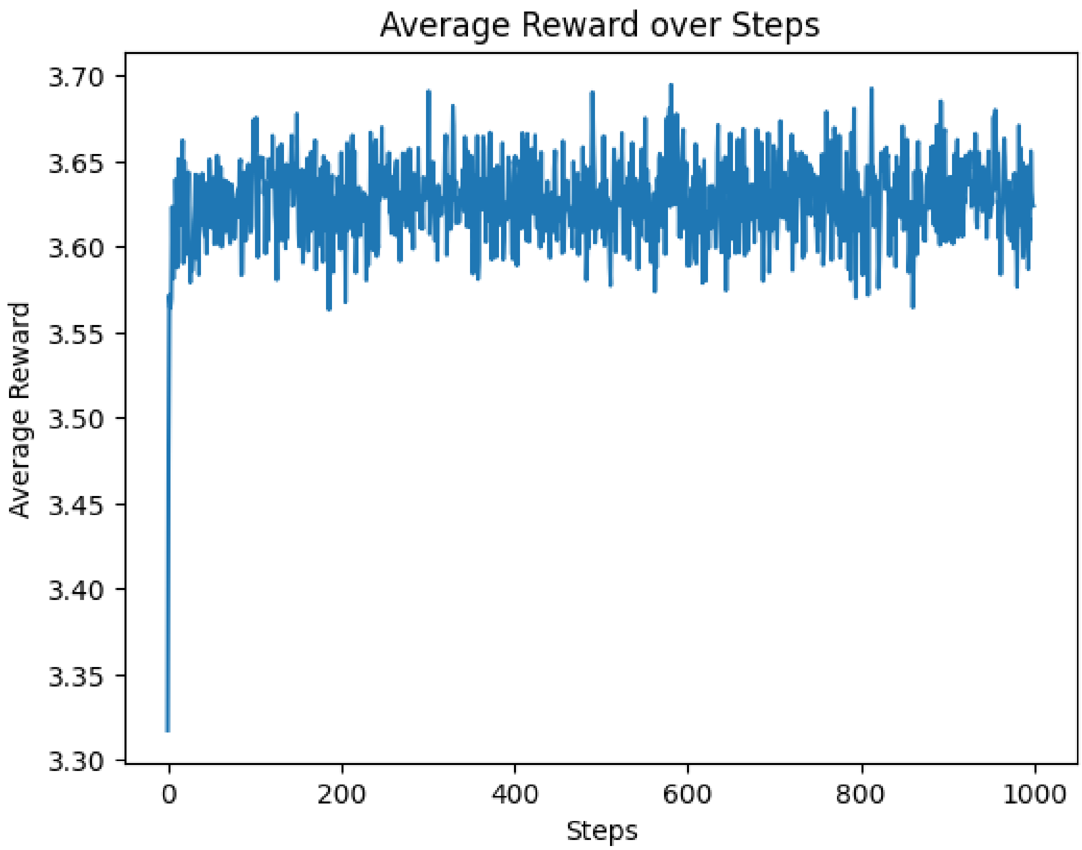
[78]:
```
# --------------
# Debugging Cell
# --------------
# Feel free to make any changes to this cell to debug your code

test_array = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
assert argmax(test_array) == 8, "Check your argmax implementation returns the␣
↪index of the largest value"

# make sure np.random.choice is called correctly
```

```
np.random.seed(0)
test_array = [1, 0, 0, 1]

assert argmax(test_array) == 0
```

[79]:
```
# More testing to make sure argmax does not always choose first entry

test_array = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
assert argmax(test_array) == 8, "Check your argmax implementation returns the␣
↪index of the largest value"

# set random seed so results are deterministic
np.random.seed(0)
test_array = [1, 0, 0, 1]

counts = [0, 0, 0, 0]
for _ in range(100):
    a = argmax(test_array)
    counts[a] += 1

# make sure argmax does not always choose first entry
assert counts[0] != 100, "Make sure your argmax implementation randomly␣
↪choooses among the largest values."

# make sure argmax does not always choose last entry
assert counts[3] != 100, "Make sure your argmax implementation randomly␣
↪choooses among the largest values."

# make sure the random number generator is called exactly once whenver `argmax`␣
↪is called
expected = [44, 0, 0, 56] # <-- notice not perfectly uniform due to randomness
assert counts == expected
```

[80]:
```
def greedy_action_selection(Q):
    """
    Select an action using a greedy action selection strategy based on the␣
↪action-value estimates Q.

    Parameters:
    Q (list): A list of action-value estimates for each action.

    Returns:
    int: The index of the selected action.
    """
    return argmax(Q)
```

[ ]:
```python
# In this version, we will run 1000 steps of the
# bandit problem and calculate the accumulated reward at each step.
# We will run 100 times and average the rewards over time to see if
# the agent is learning to select the best action.

import gymnasium as gym
import buffalo_gym

arms = 10
steps = 1000
runs = 2000

average_rewards = [0.0 for _ in range(steps)]

for run in range(runs):

    Q = [0.0 for _ in range(arms)]
    Times = [0 for _ in range(arms)]

    Rewards = [0.0 for _ in range(steps)]
    env = gym.make("Buffalo-v0", arms=arms)

    obs = env.reset()
    step = 0
    while step < steps:
        action = greedy_action_selection(Q)
        obs, reward, terminated, truncated, info = env.step(action)
        #print(f"Action: {action} - Reward: {reward}")
        Times[action] += 1
        Q = incremental_update(Q, Times, action, reward)
        Rewards[step] = reward
        step += 1

    env.close()
    #print("Final action-value estimates:", Q)
    average_rewards += np.array(Rewards)

average_rewards /= runs

# plot the average rewards over steps
import matplotlib.pyplot as plt
plt.plot(average_rewards)
plt.xlabel("Steps")
plt.ylabel("Average Reward")
plt.title("Average Reward over Steps")
plt.show()
```




Questions: 1. Is the agent able to find the best action? 2. Are the rewards improving over time?

### 1.4 4. Epsilon-Greedy Action Selection

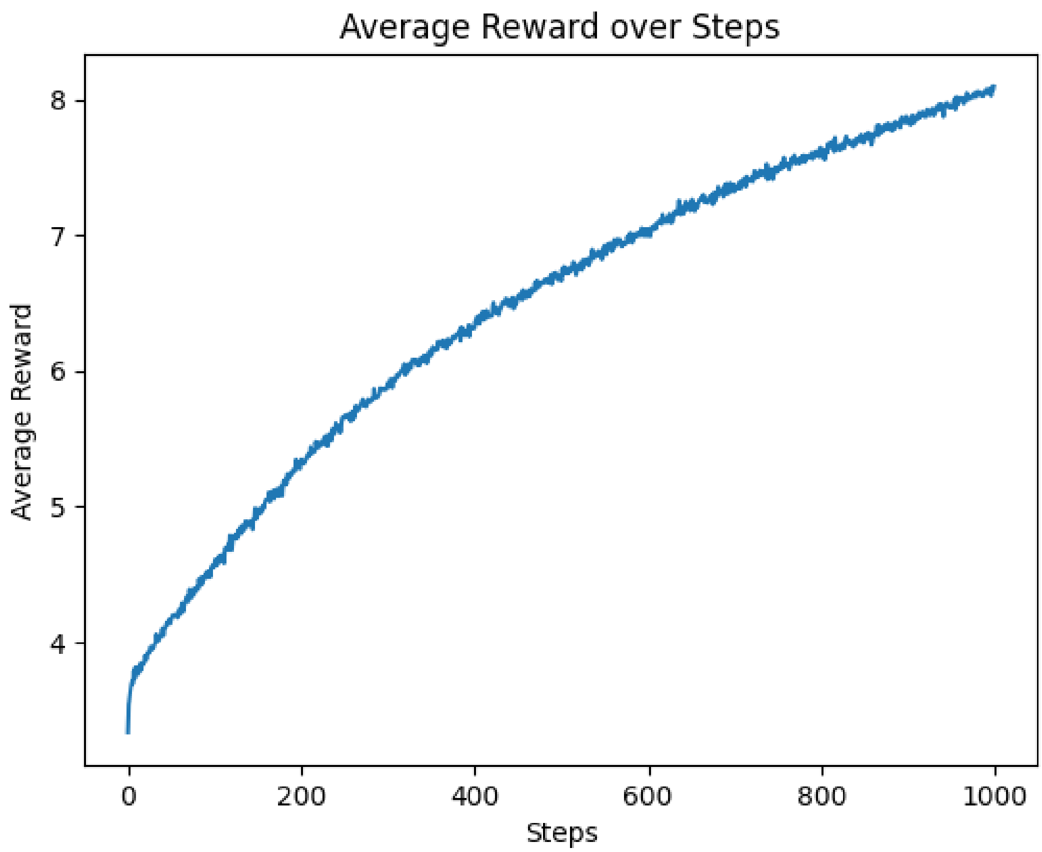
Now that we have implemented a greedy action selection strategy, we can implement an epsilon-greedy action selection strategy. Complete the function `epsilon_greedy_action_selection` to implement an epsilon-greedy action selection strategy. This function will be replace the instruction `action = greedy_action_selection(Q)` in the code above.

[82]:
```
def epsilon_greedy_action_selection(Q, epsilon):
    """
    Select an action using an epsilon-greedy action selection strategy based on␣
 ↪the action-value estimates Q.

    Parameters:
    Q (list): A list of action-value estimates for each action.
    epsilon (float): The probability of selecting a random action (exploration␣
 ↪rate).

    Returns:
```

```
    int: The index of the selected action.
    """
    if np.random.rand() < epsilon:
        return np.random.randint(len(Q))  # Explore: select a random action
    else:
        return argmax(Q)  # Exploit: select the action with the highest value
```

[ ]:
```
# In this version, we will run 1000 steps of the
# bandit problem and calculate the accumulated reward at each step.
# We will run 100 times and average the rewards over time to see if
# the agent is learning to select the best action.

import gymnasium as gym
import buffalo_gym

arms = 10
steps = 1000
runs = 2000

average_rewards_ep = [0.0 for _ in range(steps)]

for run in range(runs):

    Q = [0.0 for _ in range(arms)]
    Times = [0 for _ in range(arms)]

    Rewards = [0.0 for _ in range(steps)]
    env = gym.make("Buffalo-v0", arms=arms)

    obs = env.reset()
    step = 0
    while step < steps:
        action = epsilon_greedy_action_selection(Q, epsilon=0.01)
        obs, reward, terminated, truncated, info = env.step(action)
        #print(f"Action: {action} - Reward: {reward}")
        Times[action] += 1
        Q = incremental_update(Q, Times, action, reward)
        Rewards[step] = reward
        step += 1

    env.close()
    #print("Final action-value estimates:", Q)
    average_rewards_ep += np.array(Rewards)

average_rewards_ep /= runs

# plot the average rewards over steps
```

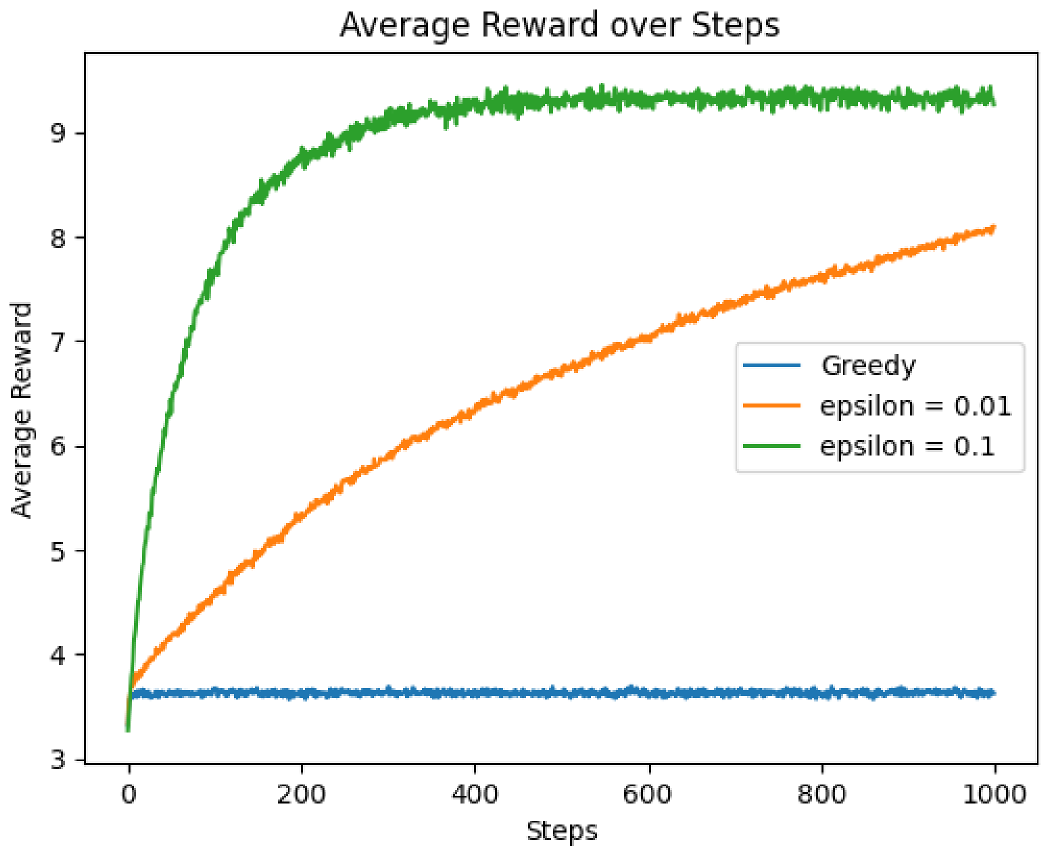
```python
import matplotlib.pyplot as plt
plt.plot(average_rewards_ep)
plt.xlabel("Steps")
plt.ylabel("Average Reward")
plt.title("Average Reward over Steps")
plt.show()
```



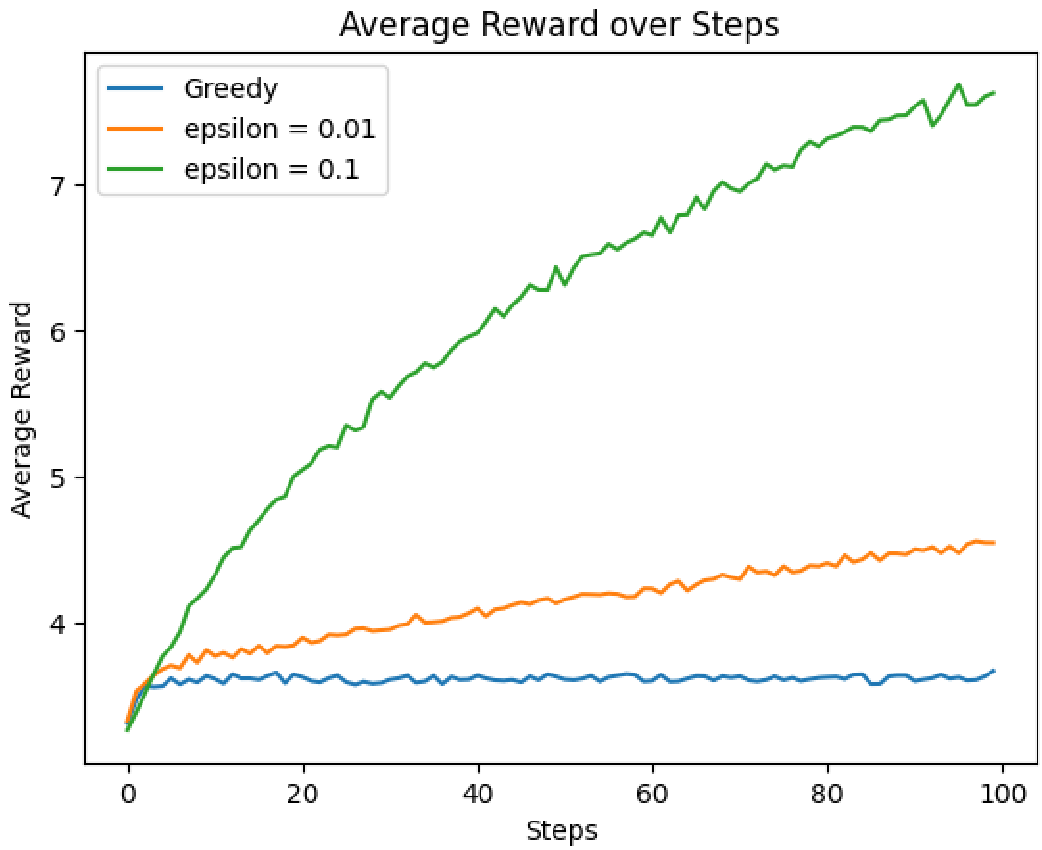

[ ]:
```python
# plot both greedy and epsilon-greedy rewards over steps
import matplotlib.pyplot as plt
plt.plot(average_rewards, label="Greedy")
plt.plot(average_rewards_ep, label="Epsilon-Greedy e=0.01")
plt.xlabel("Steps")
plt.ylabel("Average Reward")
plt.title("Average Reward over Steps")
plt.legend()
plt.show()
```


[ ]:
```python
import gymnasium as gym
import buffalo_gym

# Now we will run the same experiment with epsilon = 0.1

epsilon = 0.1
arms = 10
steps = 1000
runs = 2000

average_rewards_ep_2 = [0.0 for _ in range(steps)]

for run in range(runs):

    Q = [0.0 for _ in range(arms)]
    Times = [0 for _ in range(arms)]

    Rewards = [0.0 for _ in range(steps)]
    env = gym.make("Buffalo-v0", arms=arms)
```

```
    obs = env.reset()
    step = 0
    while step < steps:
        action = epsilon_greedy_action_selection(Q, epsilon=epsilon)
        obs, reward, terminated, truncated, info = env.step(action)
        #print(f"Action: {action} - Reward: {reward}")
        Times[action] += 1
        Q = incremental_update(Q, Times, action, reward)
        Rewards[step] = reward
        step += 1

    env.close()
    #print("Final action-value estimates:", Q)
    average_rewards_ep_2 += np.array(Rewards)

average_rewards_ep_2 /= runs

# plot the average rewards over steps
import matplotlib.pyplot as plt
plt.plot(average_rewards, label="Greedy")
plt.plot(average_rewards_ep, label="epsilon = 0.01")
plt.plot(average_rewards_ep_2, label="epsilon = 0.1")
plt.xlabel("Steps")
plt.ylabel("Average Reward")
plt.title("Average Reward over Steps")
plt.legend()
plt.show()
```




[86]:
```
# plot the average rewards over steps (first 100 steps)
import matplotlib.pyplot as plt
max_steps = min(100, len(average_rewards), len(average_rewards_ep),␣
 ↪len(average_rewards_ep_2))
plt.plot(average_rewards[:max_steps], label="Greedy")
plt.plot(average_rewards_ep[:max_steps], label="epsilon = 0.01")
plt.plot(average_rewards_ep_2[:max_steps], label="epsilon = 0.1")
plt.xlabel("Steps")
plt.ylabel("Average Reward")
plt.title("Average Reward over Steps")
plt.legend()
plt.show()
```



Average Reward over Steps
Greedy
epsilon = 0.01
epsilon = 0.1
Average Reward
Steps
0
20
40
60
80
100
4
5
6
7