Assignment 2

Tags	Assignment 2.pdf

question 1

Q1. (Programming Exercise - 10 points)

Consider a six-armed bandit problem, where each arm either gives a reward of 1$ or nothing. The probability of yielding a reward of 1$ for each arm is given by:

Arm 1	Arm 2	Arm 3	Arm 4	Arm 5	Arm 6
0.55	0.45	0.3	0.40	0.35	0.48

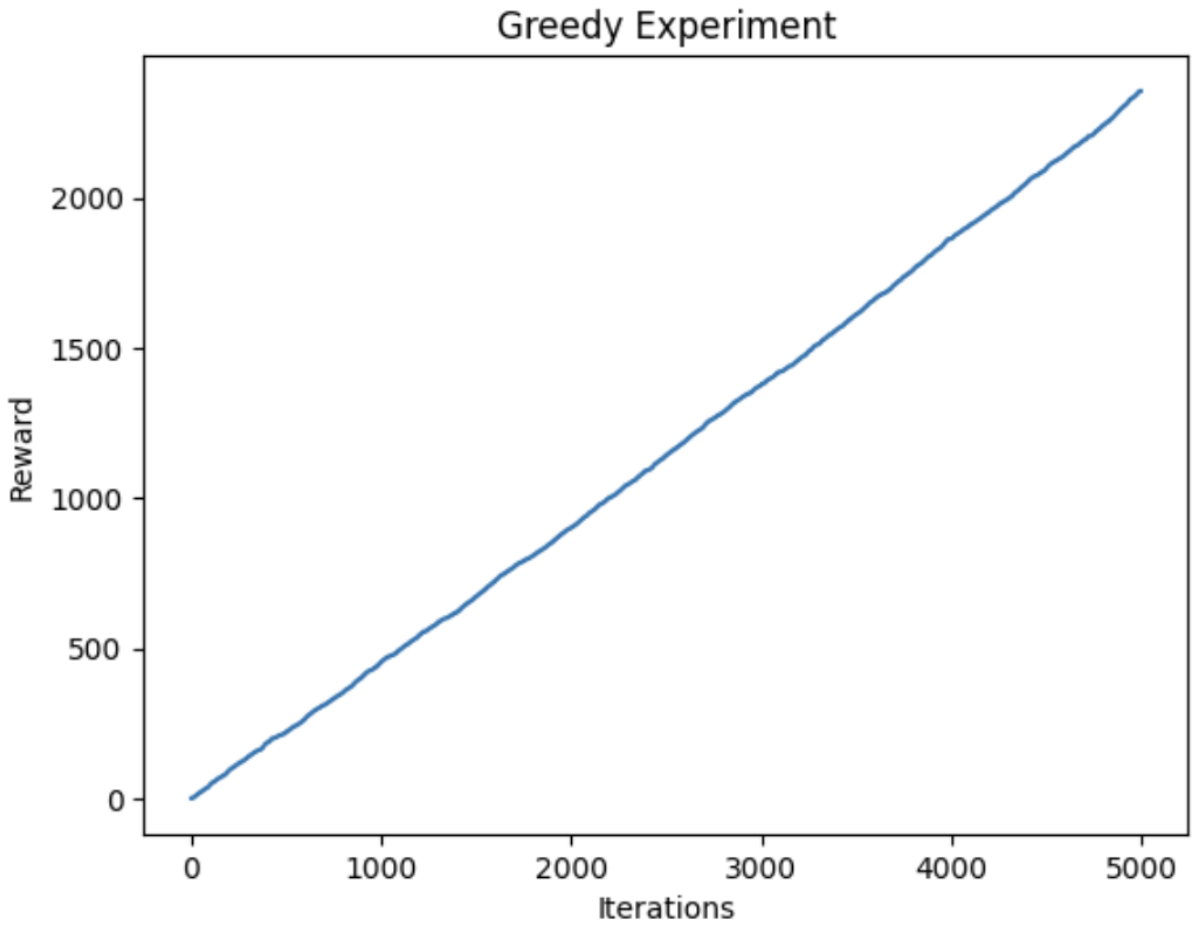
Simulate the different action selection algorithms, i.e.,

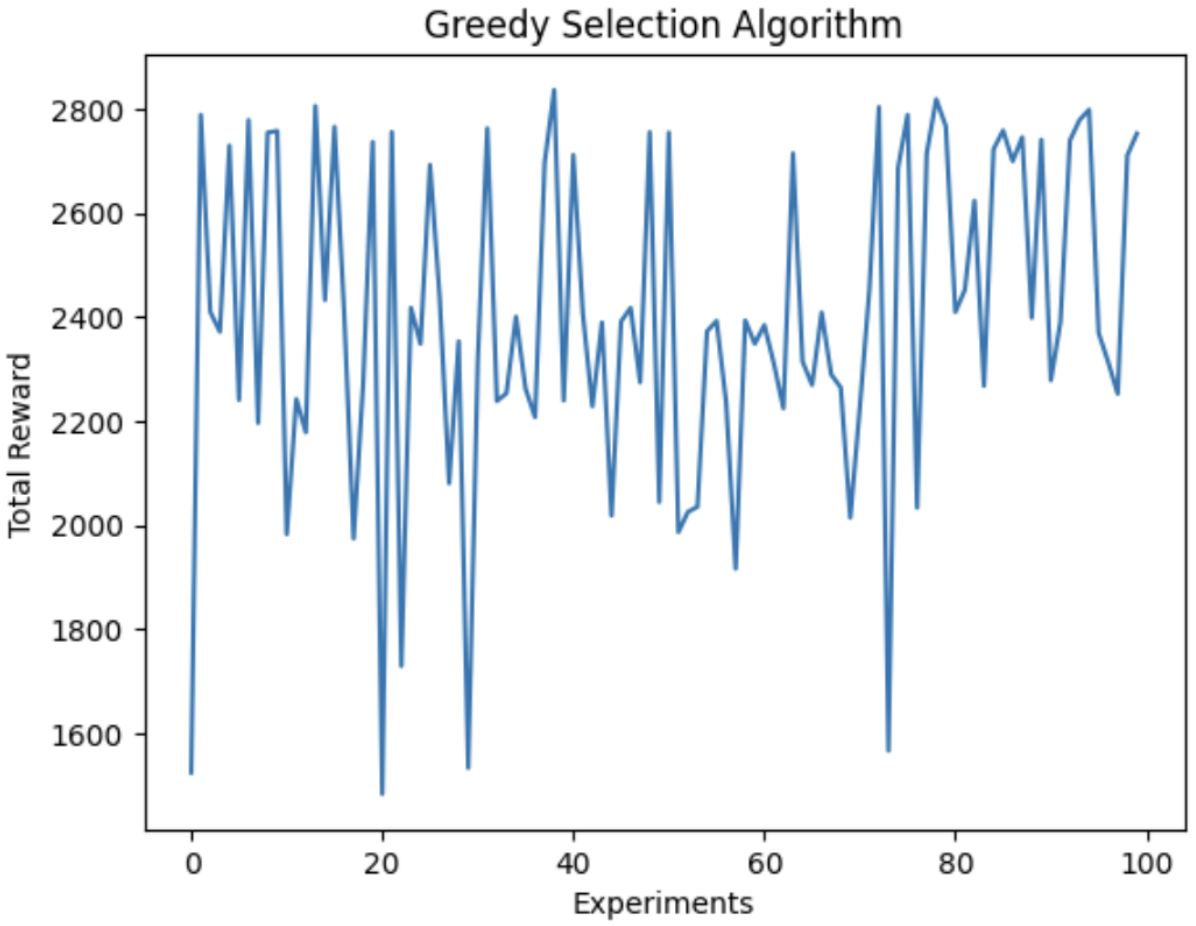
1. Greedy,
2. Epsilon-Greedy with epsilon = 0.1
3. Softmax
4. Upper confidence bound based action selection.
5. Thompson Sampling

perform over 5000 iterations. Please make assumptions on parameters as required.

Repeat the experiment 100 times for each algorithm. Plot the average reward earned with each of the algorithms over the 5000 iterations.

1. greedy


Greedy Experiment
Reward
2000
1500
1000
500
0
0
1000
2000
3000
4000
5000
Iterations




```
# Greedy Action Selection Algorithm

import matplotlib.pyplot as plt
import numpy as np

N_ARMS = 6
ARM_REWARD_PROBABILITY = np.array([0.55, 0.45, 0.3, 0.4, 0.35, 0.48])
REWARD = np.ones(N_ARMS)
EXPERIMENTS = 100
ITERATIONS = 5000

class Greedy:
  def __init__(self, N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS):
    self.N_ARMS = N_ARMS
    self.REWARD = REWARD
    self.ARM_REWARD_PROBABILITY = ARM_REWARD_PROBABILITY
    self.ITERATIONS = ITERATIONS
    self.avg_reward = np.zeros(N_ARMS)
    self.times_picked = np.zeros(N_ARMS)
    self.random_number_generator = np.random.default_rng()

  def pick_arm(self, arm_index):
    random = self.random_number_generator.random()
    reward_obtained = 0

    if random < self.ARM_REWARD_PROBABILITY[arm_index]:
      reward_obtained = self.REWARD[arm_index]

    numerator = self.avg_reward[arm_index] * self.times_picked[arm_index] + reward_obtained
    denominator = self.times_picked[arm_index] + 1

    self.avg_reward[arm_index] = numerator / denominator
    self.times_picked[arm_index] += 1

    return reward_obtained

  def start_experiment(self):
    experiment_reward = 0
```

```
    # initially we pick all the arms once
    for i in range(self.N_ARMS):
      experiment_reward += self.pick_arm(i)

    # compute best arm
    for i in range(self.ITERATIONS - self.N_ARMS):
      best_arm = np.argmax(self.avg_reward)
      experiment_reward += self.pick_arm(best_arm)

    return experiment_reward

  def view_sample_experiment(self):
    all_rewards = []
    experiment_reward = 0

     # initially we pick all the arms once
    for i in range(self.N_ARMS):
      experiment_reward += self.pick_arm(i)
      all_rewards.append(experiment_reward)

    # compute best arm
    for i in range(self.ITERATIONS - self.N_ARMS):
      best_arm = np.argmax(self.avg_reward)
      experiment_reward += self.pick_arm(best_arm)
      all_rewards.append(experiment_reward)

    print("experiment_reward: ", experiment_reward)
    fig, ax = plt.subplots()
    ax.plot(range(ITERATIONS), all_rewards)
    ax.set(xlabel="Iterations", ylabel="Reward", title="Greedy Experiment")
    plt.show()

greedy = Greedy(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS)
greedy.view_sample_experiment()

a1_rewards = []
for e in range(EXPERIMENTS):
  greedy = Greedy(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS)
  reward = greedy.start_experiment()
  a1_rewards.append(reward)

print("average_experiment_reward: ", np.mean(a1_rewards))
fig, ax = plt.subplots()
ax.plot(range(EXPERIMENTS), a1_rewards)
ax.set(xlabel="Experiments", ylabel="Total Reward", title="Greedy Selection Algorithm")
plt.show()
```

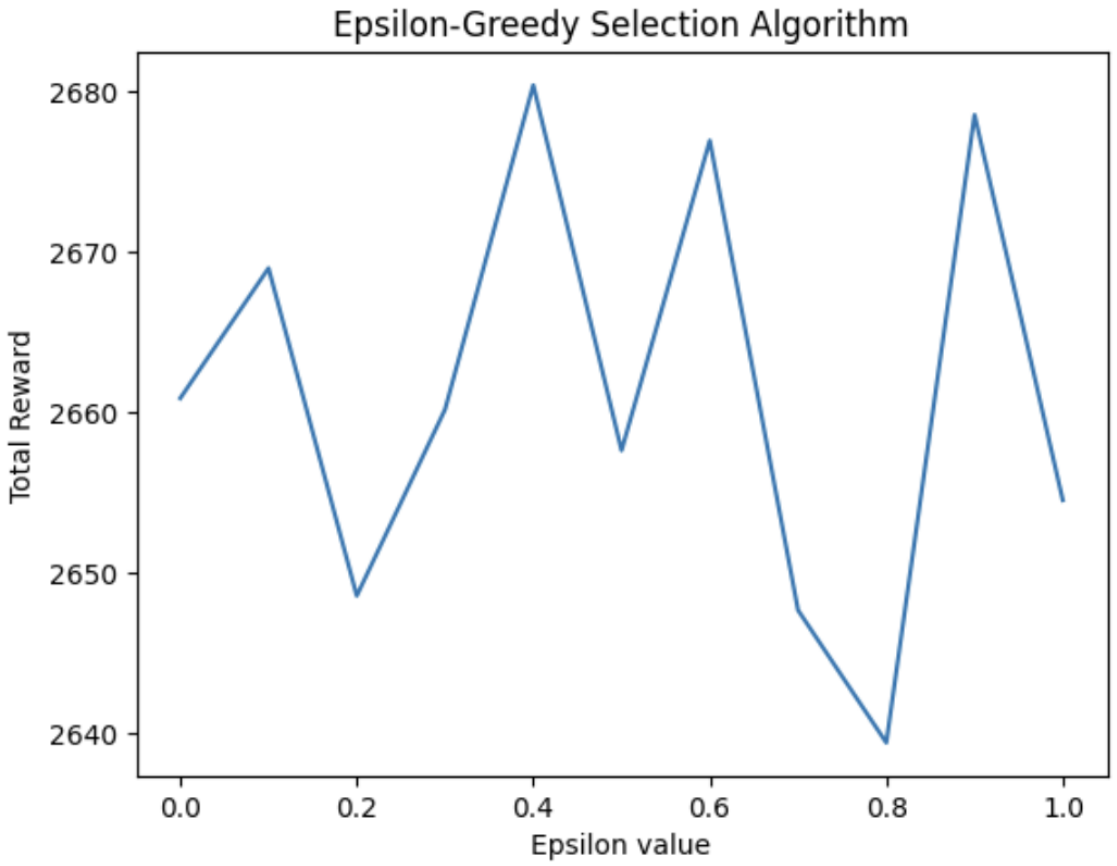
2. epsilon-greedy

2a. exploration of epsilon values

To investigate how experiment reward changes with the epsilon value in the Epsilon-Greedy Selection Algorithm, a simple test was conducted with epsilon values from 0 to 1 with a step size of 0.1, with the experiment reward calculated as the average of 20 experiments.

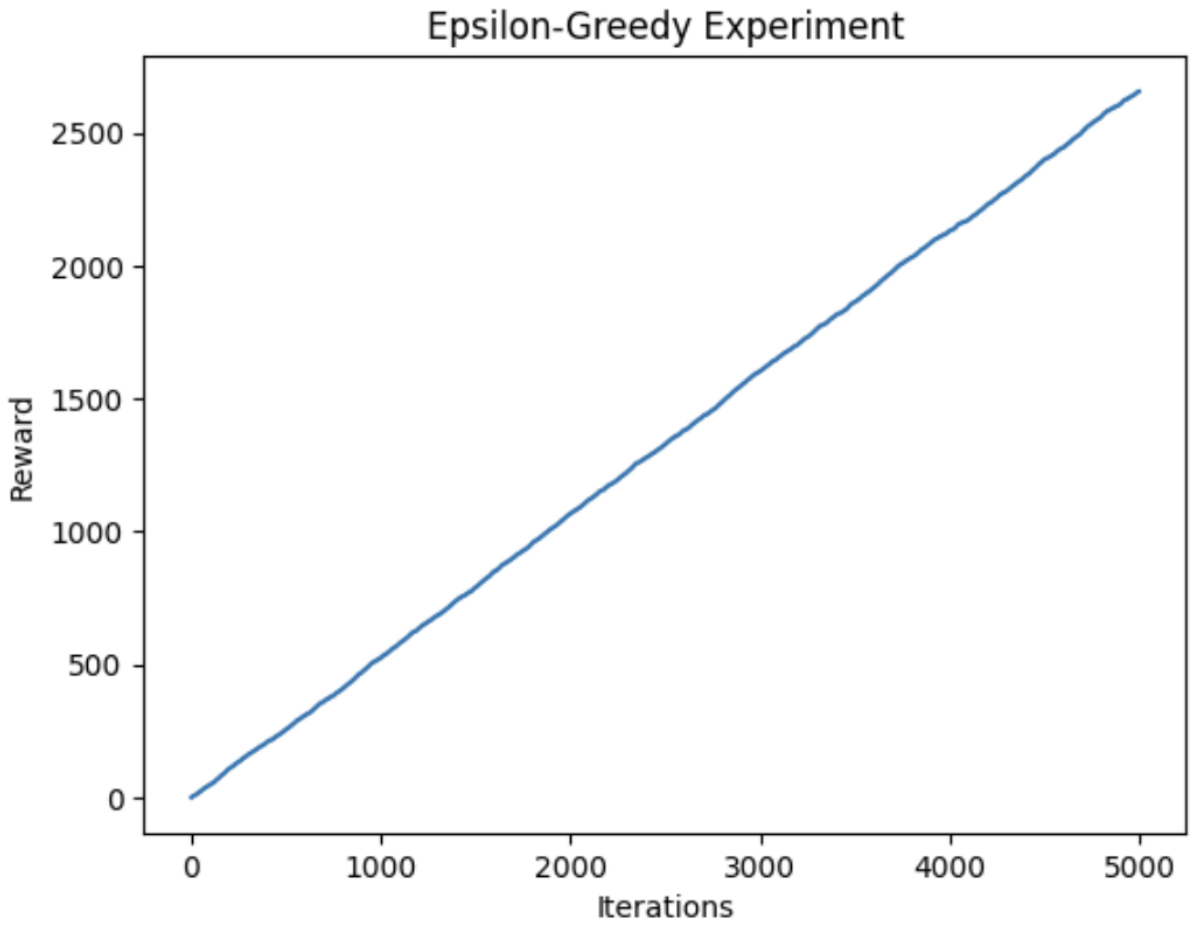
As can be seen from the corresponding graph, epsilon=0.4 gives a higher experiment reward as compared to other epsilon values.

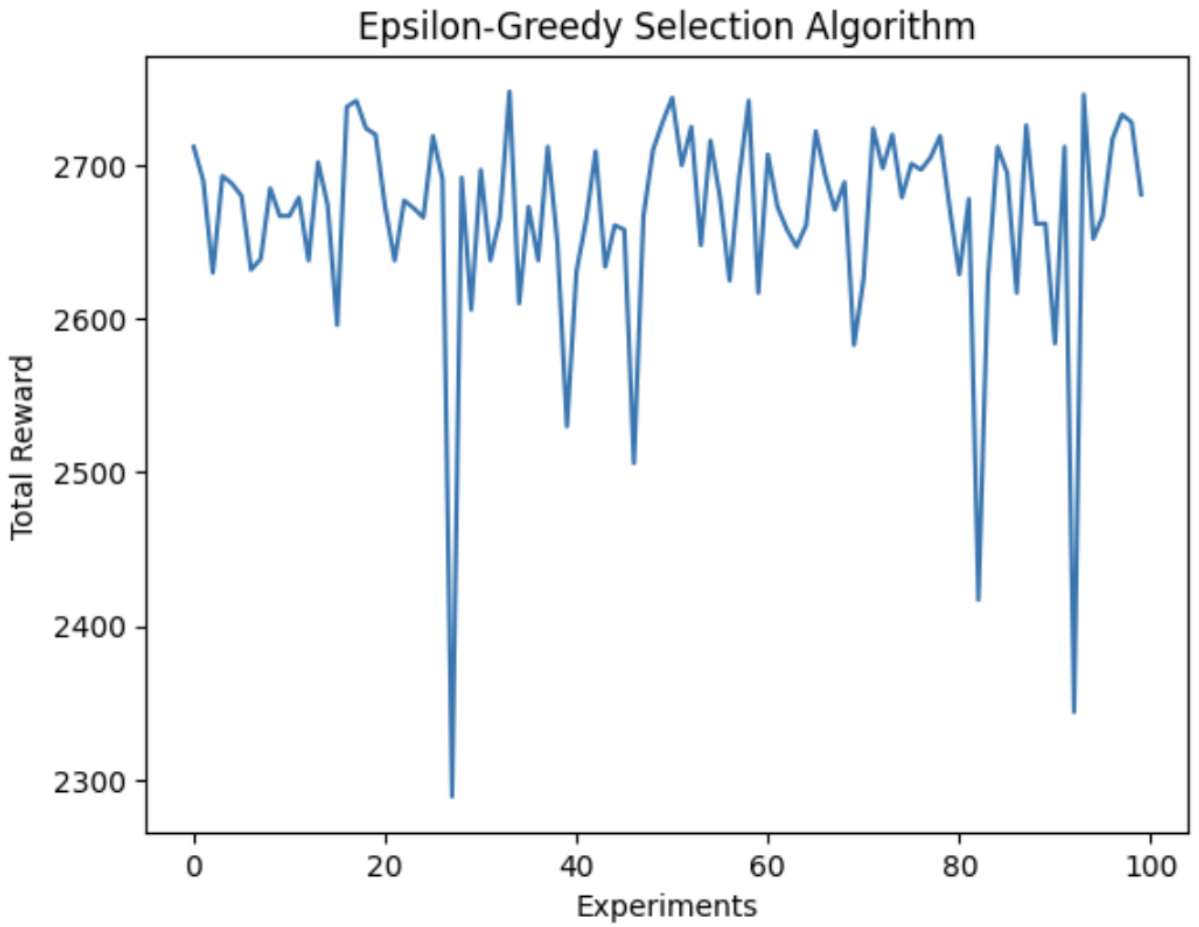
This may be due to the small differences in the probability of yielding a reward for each arm, which makes exploration more worthwhile to find the arm with the best reward function.




```
# Exploration for Epsilon values

epsilon_rewards = []
epsilon_values = []
MAX_EPSILON = 1
STEP = 0.1
EXPERIMENTS = 20

epsilon = 0
while epsilon <= MAX_EPSILON:
  e_rewards = []
  for e in range(EXPERIMENTS):
    epsilon_greedy = Epsilon_Greedy(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, EPSILON)
    e_rewards.append(epsilon_greedy.start_experiment())

  epsilon_rewards.append(np.mean(e_rewards))
  epsilon_values.append(epsilon)
  epsilon += STEP

print("max epsilon value: ", epsilon_values[np.argmax(epsilon_rewards)])
fig, ax = plt.subplots()
ax.plot(epsilon_values, epsilon_rewards)
ax.set(xlabel="Epsilon value", ylabel="Total Reward", title="Epsilon-Greedy Selection Algorithm")
plt.show()
```

2b. simulation of epsilon-greedy algorithm


Epsilon-Greedy Experiment
Reward
2500
2000
1500
1000
500
0
0
1000
2000
3000
4000
5000
Iterations

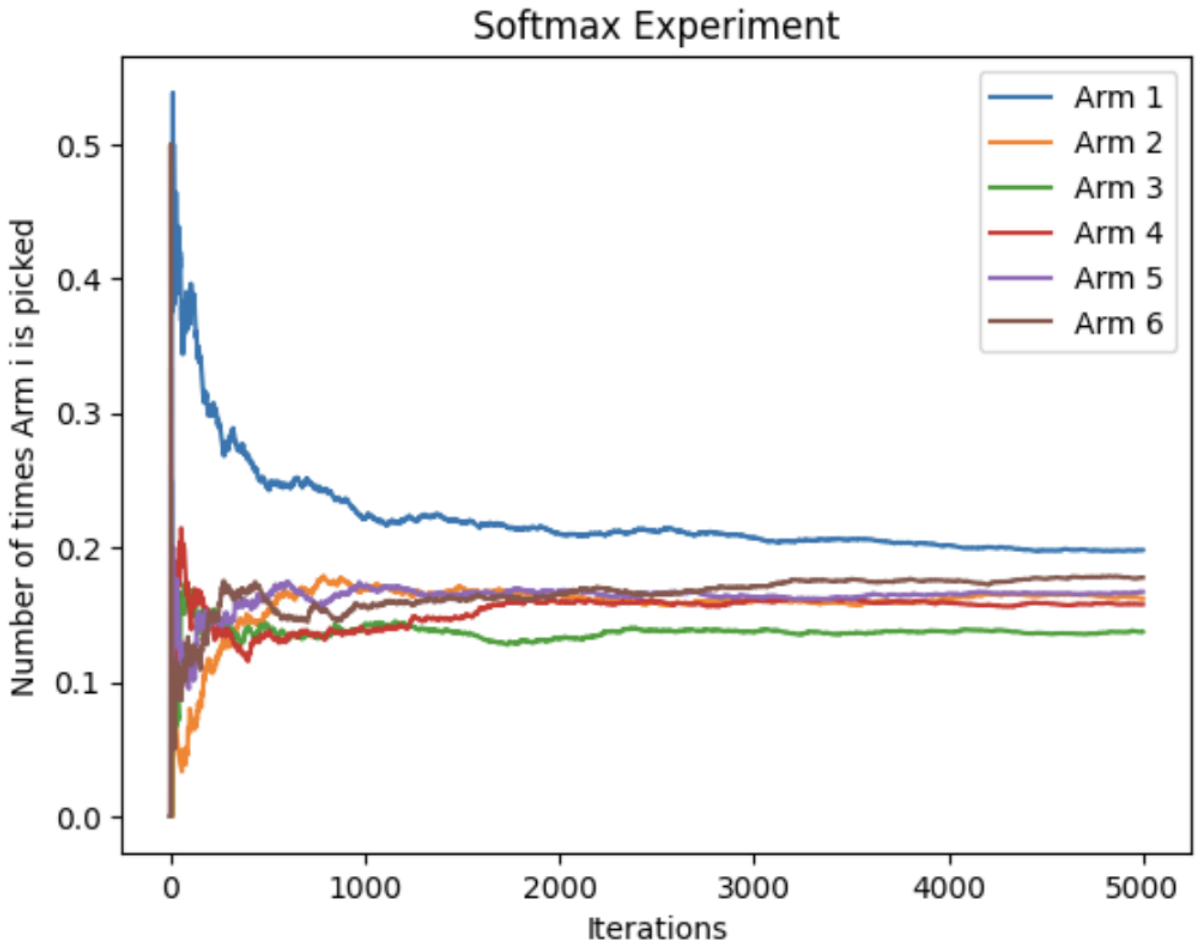


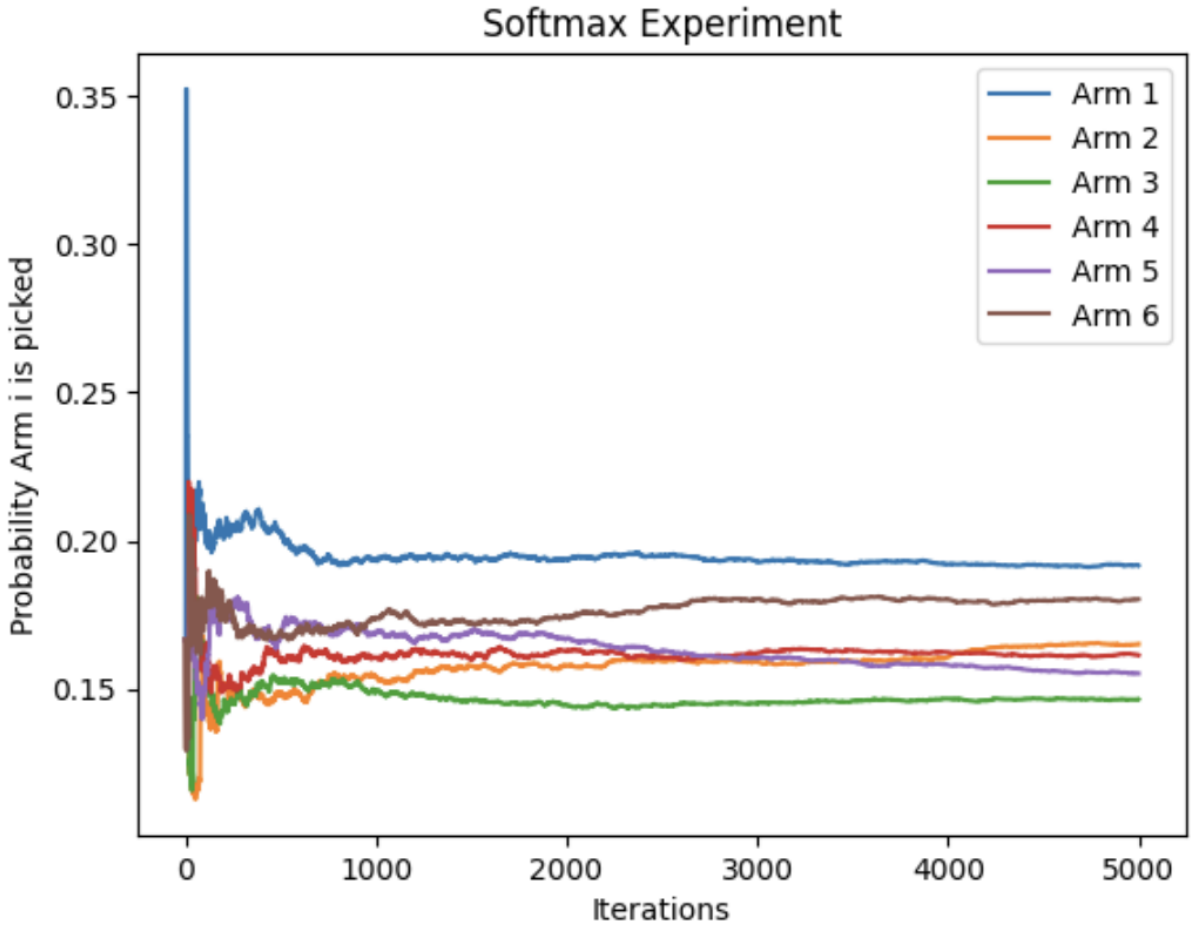

```
# Epsilon-Greedy Action Selection Algorithm with epsilon = 0.1

import matplotlib.pyplot as plt
import numpy as np

N_ARMS = 6
ARM_REWARD_PROBABILITY = np.array([0.55, 0.45, 0.3, 0.4, 0.35, 0.48])
REWARD = np.ones(N_ARMS)
EXPERIMENTS = 100
ITERATIONS = 5000
EPSILON = 0.1

class Epsilon_Greedy:
  def __init__(self, N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, EPSILON):
    self.N_ARMS = N_ARMS
    self.REWARD = REWARD
    self.ARM_REWARD_PROBABILITY = ARM_REWARD_PROBABILITY
    self.ITERATIONS = ITERATIONS
    self.EPSILON = EPSILON
    self.avg_reward = np.zeros(N_ARMS)
    self.times_picked = np.zeros(N_ARMS)
    self.random_number_generator = np.random.default_rng()

  def pick_arm(self, arm_index):
    random = self.random_number_generator.random()
    reward_obtained = 0

    if random < self.ARM_REWARD_PROBABILITY[arm_index]:
      reward_obtained = self.REWARD[arm_index]

    numerator = self.avg_reward[arm_index] * self.times_picked[arm_index] + reward_obtained
    denominator = self.times_picked[arm_index] + 1

    self.avg_reward[arm_index] = numerator / denominator
    self.times_picked[arm_index] += 1

    return reward_obtained

  def start_experiment(self):
```

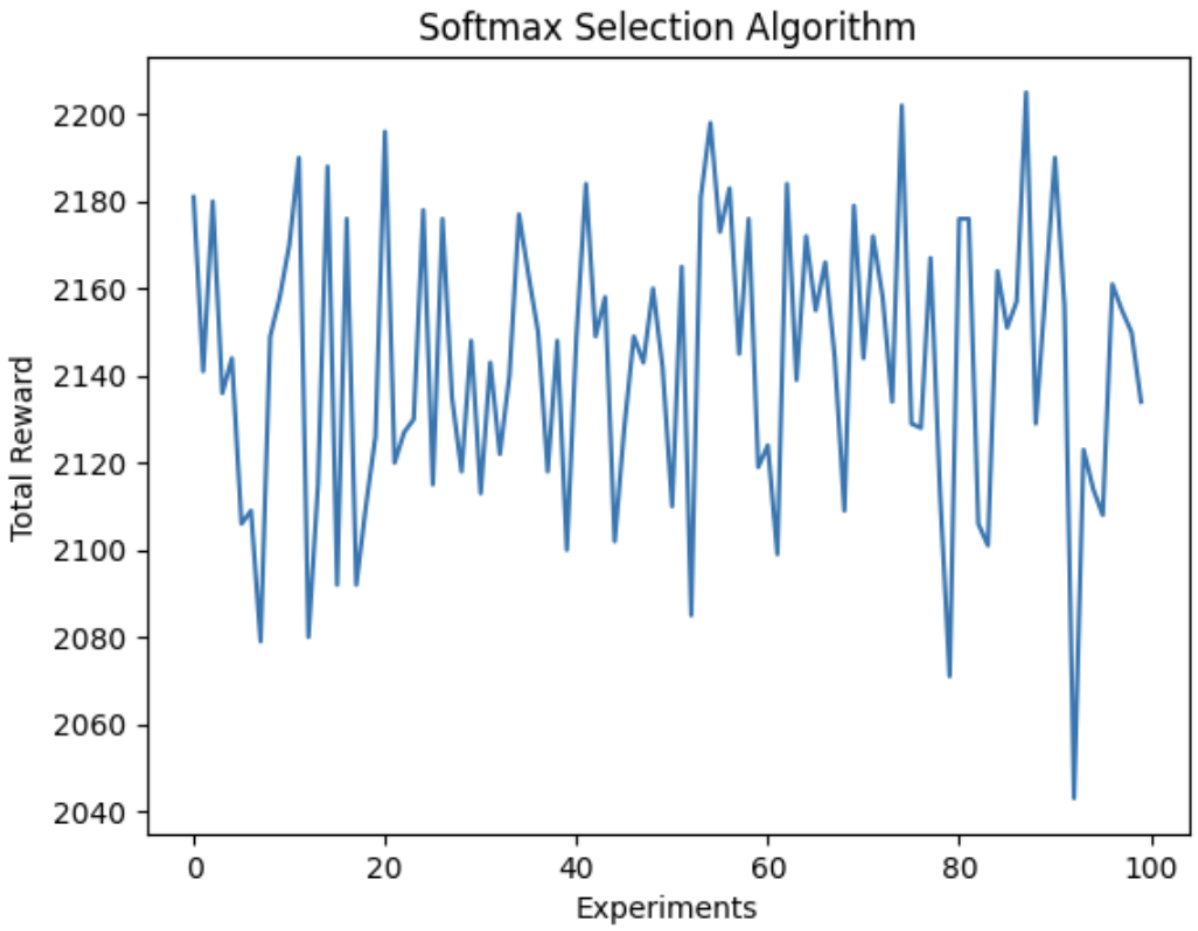
```
    experiment_reward = 0

    for i in range(self.ITERATIONS):
      random = self.random_number_generator.random()

      if random < self.EPSILON:
        # choose a random arm
        arm = np.random.choice(self.N_ARMS, 1)[0]
      else:
        # choose a greedy arm
        arm = np.argmax(self.avg_reward)

      experiment_reward += self.pick_arm(arm)

    return experiment_reward

  def view_sample_experiment(self):
    all_rewards = []
    experiment_reward = 0

    for i in range(self.ITERATIONS):
      random = self.random_number_generator.random()

      if random < self.EPSILON:
        # choose a random arm
        arm = np.random.choice(self.N_ARMS, 1)[0]
      else:
        # choose a greedy arm
        arm = np.argmax(self.avg_reward)

      experiment_reward += self.pick_arm(arm)
      all_rewards.append(experiment_reward)

    print("experiment_reward: ", experiment_reward)
    fig, ax = plt.subplots()
    ax.plot(range(ITERATIONS), all_rewards)
    ax.set(xlabel="Iterations", ylabel="Reward", title="Epsilon-Greedy Experiment")
    plt.show()

epsilon_greedy = Epsilon_Greedy(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, EPSILON)
epsilon_greedy.view_sample_experiment()

a2_rewards = []
for e in range(EXPERIMENTS):
  epsilon_greedy = Epsilon_Greedy(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, EPSILON)
  reward = epsilon_greedy.start_experiment()
  a2_rewards.append(reward)

print("average_experiment_reward: ", np.mean(a2_rewards))
fig, ax = plt.subplots()
ax.plot(range(EXPERIMENTS), a2_rewards)
ax.set(xlabel="Experiments", ylabel="Total Reward", title="Epsilon-Greedy Selection Algorithm")
plt.show()
```

3. softmax

Instead of uniformly exploring all arms like the epsilon-greedy algorithm, the softmax action selection algorithm provides different probabilities for arms being pulled based on their current reward averages. We can see that the probability that an arm is picked eventually converges to a similar order as the probability of that arm yielding a reward, as the number of iterations increase towards 5000.


Softmax Experiment
Arm 1
Arm 2
Arm 3
Arm 4
Arm 5
Arm 6
Number of times Arm i is picked
0.0
0.1
0.2
0.3
0.4
0.5
0
1000
2000
3000
4000
5000
Iterations


Softmax Experiment
Arm 1
Arm 2
Arm 3
Arm 4
Arm 5
Arm 6
Probability Arm i is picked
0.35
0.30
0.25
0.20
0.15
0
1000
2000
3000
4000
5000
Iterations




```
# Softmax Action Selection Algorithm

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

N_ARMS = 6
ARM_REWARD_PROBABILITY = np.array([0.55, 0.45, 0.3, 0.4, 0.35, 0.48])
REWARD = np.ones(N_ARMS)
EXPERIMENTS = 100
ITERATIONS = 5000

class Softmax():
  def __init__(self, N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS):
    self.N_ARMS = N_ARMS
    self.REWARD = REWARD
    self.ARM_REWARD_PROBABILITY = ARM_REWARD_PROBABILITY
    self.ITERATIONS = ITERATIONS
    self.arm_0_times_picked = []
    self.arm_1_times_picked = []
    self.arm_2_times_picked = []
    self.arm_3_times_picked = []
    self.arm_4_times_picked = []
    self.arm_5_times_picked = []
    self.p1 = []
    self.p2 = []
    self.p3 = []
    self.p4 = []
    self.p5 = []
    self.p6 = []
    self.avg_reward = np.zeros(N_ARMS)
    self.times_picked = np.zeros(N_ARMS)
    self.random_number_generator = np.random.default_rng()

  def pick_arm(self, arm_index):
    random = self.random_number_generator.random()
    reward_obtained = 0
```

```
    if random < self.ARM_REWARD_PROBABILITY[arm_index]:
      reward_obtained = self.REWARD[arm_index]

    numerator = self.avg_reward[arm_index] * self.times_picked[arm_index] + reward_obtained
    denominator = self.times_picked[arm_index] + 1

    self.avg_reward[arm_index] = numerator / denominator
    self.times_picked[arm_index] += 1

    return reward_obtained

  def select_arm(self):
    z = sum([math.exp(v) for v in self.avg_reward])
    probabilities = [math.exp(v) / z for v in self.avg_reward]
    self.p1.append(probabilities[0])
    self.p2.append(probabilities[1])
    self.p3.append(probabilities[2])
    self.p4.append(probabilities[3])
    self.p5.append(probabilities[4])
    self.p6.append(probabilities[5])

    random = self.random_number_generator.random()
    cumulative_probability = 0

    for i in range(len(probabilities)):
      cumulative_probability += probabilities[i]

      if cumulative_probability > random:
        return i

    return self.N_ARMS - 1

  def start_experiment(self):
    experiment_reward = 0

    for i in range(self.ITERATIONS):
      arm = self.select_arm()
      experiment_reward += self.pick_arm(arm)

    return experiment_reward

  def view_sample_experiment(self):
    all_rewards = []
    experiment_reward = 0

    for i in range(self.ITERATIONS):
      arm = self.select_arm()
      self.arm_0_times_picked.append(self.times_picked[0]/(i+1))
      self.arm_1_times_picked.append(self.times_picked[1]/(i+1))
      self.arm_2_times_picked.append(self.times_picked[2]/(i+1))
      self.arm_3_times_picked.append(self.times_picked[3]/(i+1))
      self.arm_4_times_picked.append(self.times_picked[4]/(i+1))
      self.arm_5_times_picked.append(self.times_picked[5]/(i+1))

      experiment_reward += self.pick_arm(arm)
      all_rewards.append(experiment_reward)

    fig, ax = plt.subplots()
    ax.plot(range(ITERATIONS), all_rewards)
    ax.set(xlabel="Iterations", ylabel="Reward", title="Softmax Experiment")
    plt.show()

a3_rewards = []
for e in range(EXPERIMENTS):
  softmax = Softmax(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS)
  reward = softmax.start_experiment()
  a3_rewards.append(reward)

print("average_experiment_reward: ", np.mean(a3_rewards))
fig, ax = plt.subplots()
ax.plot(range(EXPERIMENTS), a3_rewards)
ax.set(xlabel="Experiments", ylabel="Total Reward", title="Softmax Selection Algorithm")
plt.show()
```

4. upper confidence bound based

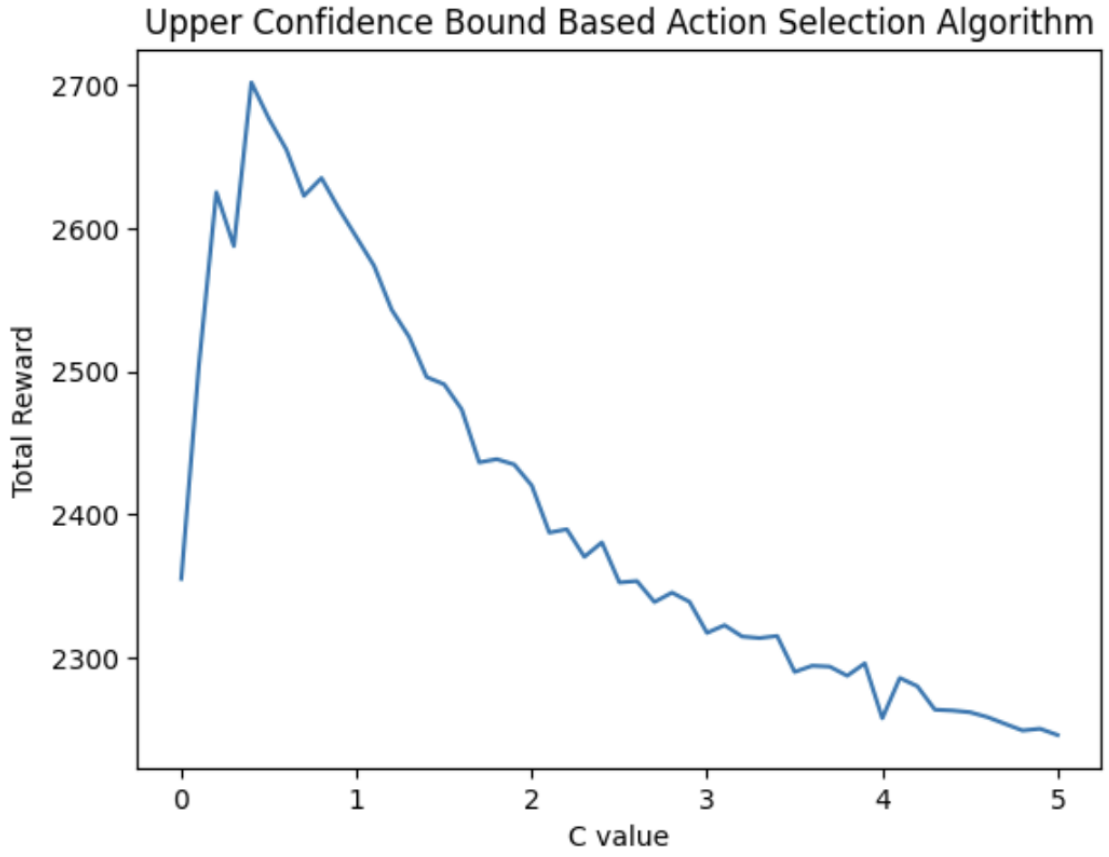
4a. exploration of C values

To investigate how experiment reward changes with the C value in the Upper Confidence Bound Based Action Selection Algorithm, a simple test was conducted with C values from 0 to 5 with a step size of 0.1, with the experiment reward calculated

as the average of 20 experiments.

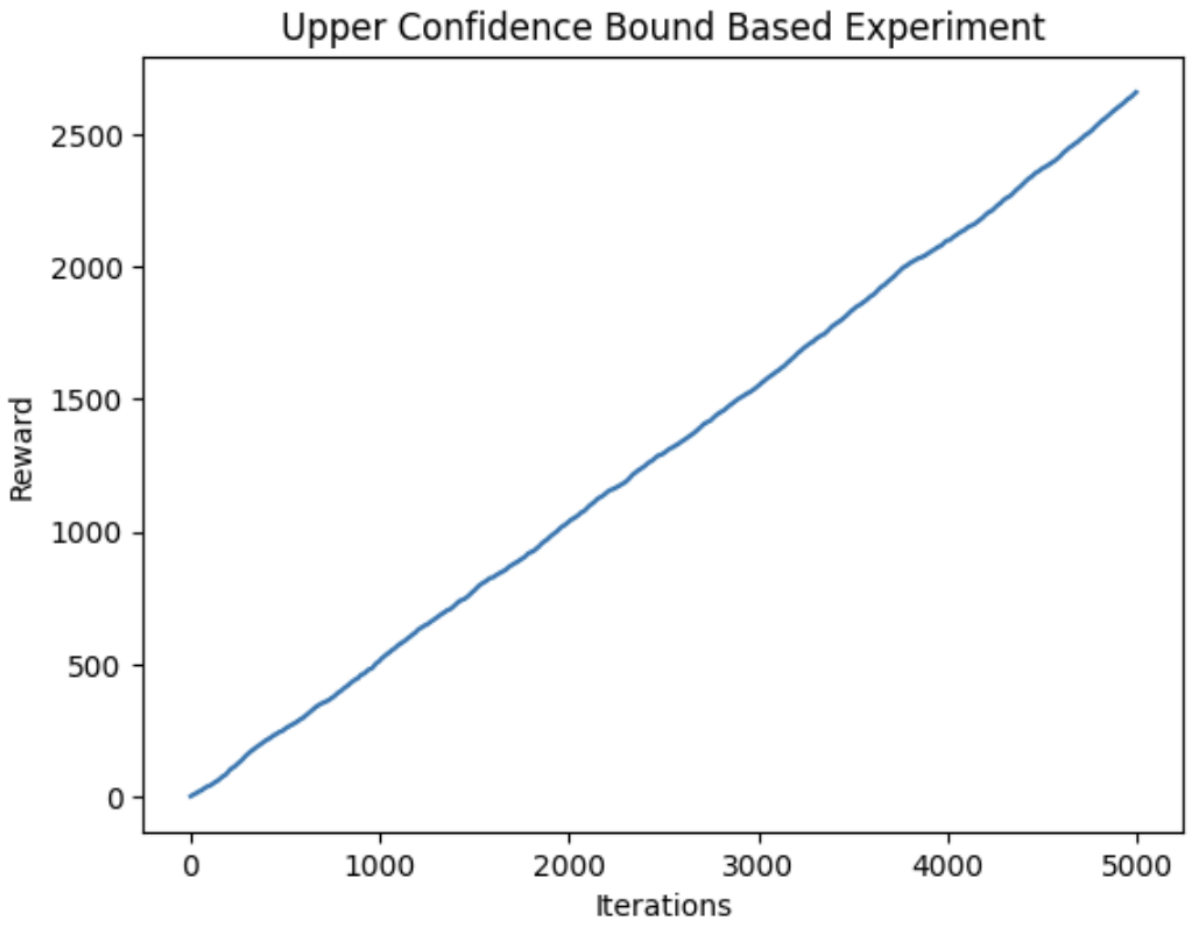
As can be seen from the corresponding graph, C=0.4 gives a higher experiment reward as compared to other C values.

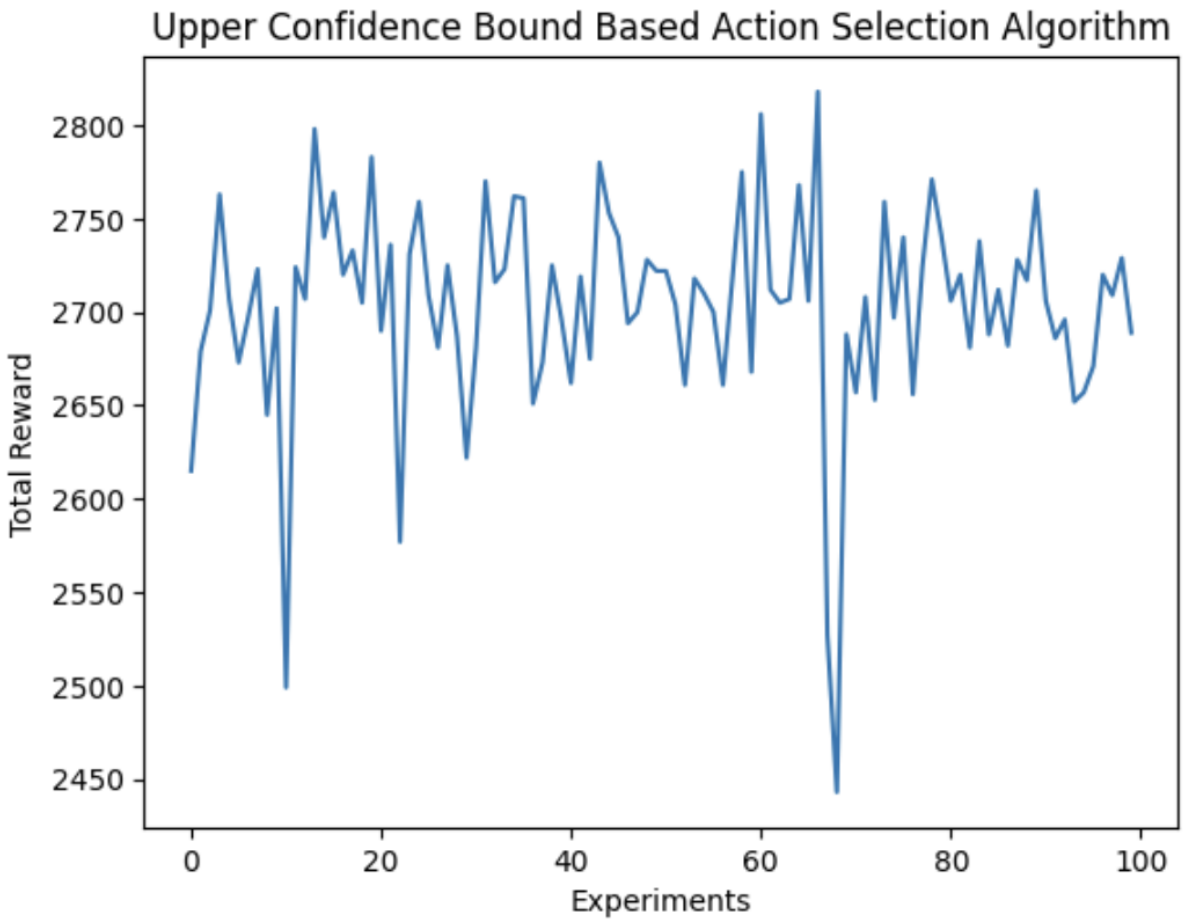
Hence, for the simulation of UCB, a value of C=0.4 will be used to achieve higher experiment rewards.




```
# Exploration for C values

c_rewards = []
c_values = []
MAX_C = 5
STEP = 0.1
EXPERIMENTS = 20

c = 0
while c <= MAX_C:
  e_rewards = []
  for e in range(EXPERIMENTS):
    ucb = UCB(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, c)
    e_rewards.append(ucb.start_experiment())

  c_rewards.append(np.mean(e_rewards))
  c_values.append(c)
  c += STEP

print("max c value: ", c_values[np.argmax(c_rewards)])
fig, ax = plt.subplots()
ax.plot(c_values, c_rewards)
ax.set(xlabel="C value", ylabel="Total Reward", title="Upper Confidence Bound Based Action Selection Algorithm")
plt.show()
```

4b. simulation of UCB algorithm

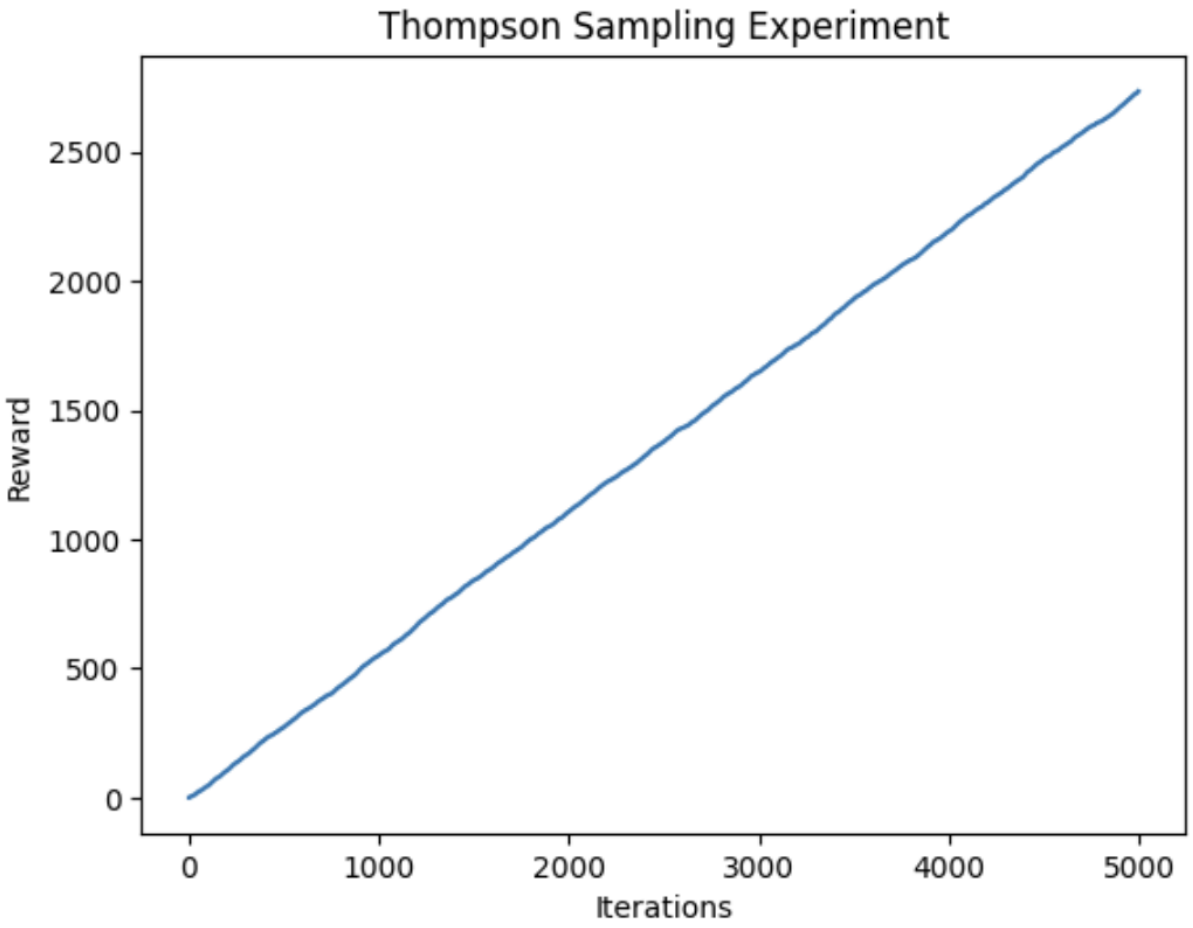

Upper Confidence Bound Based Experiment
Reward
Iterations
0
500
1000
1500
2000
2500
0
1000
2000
3000
4000
5000

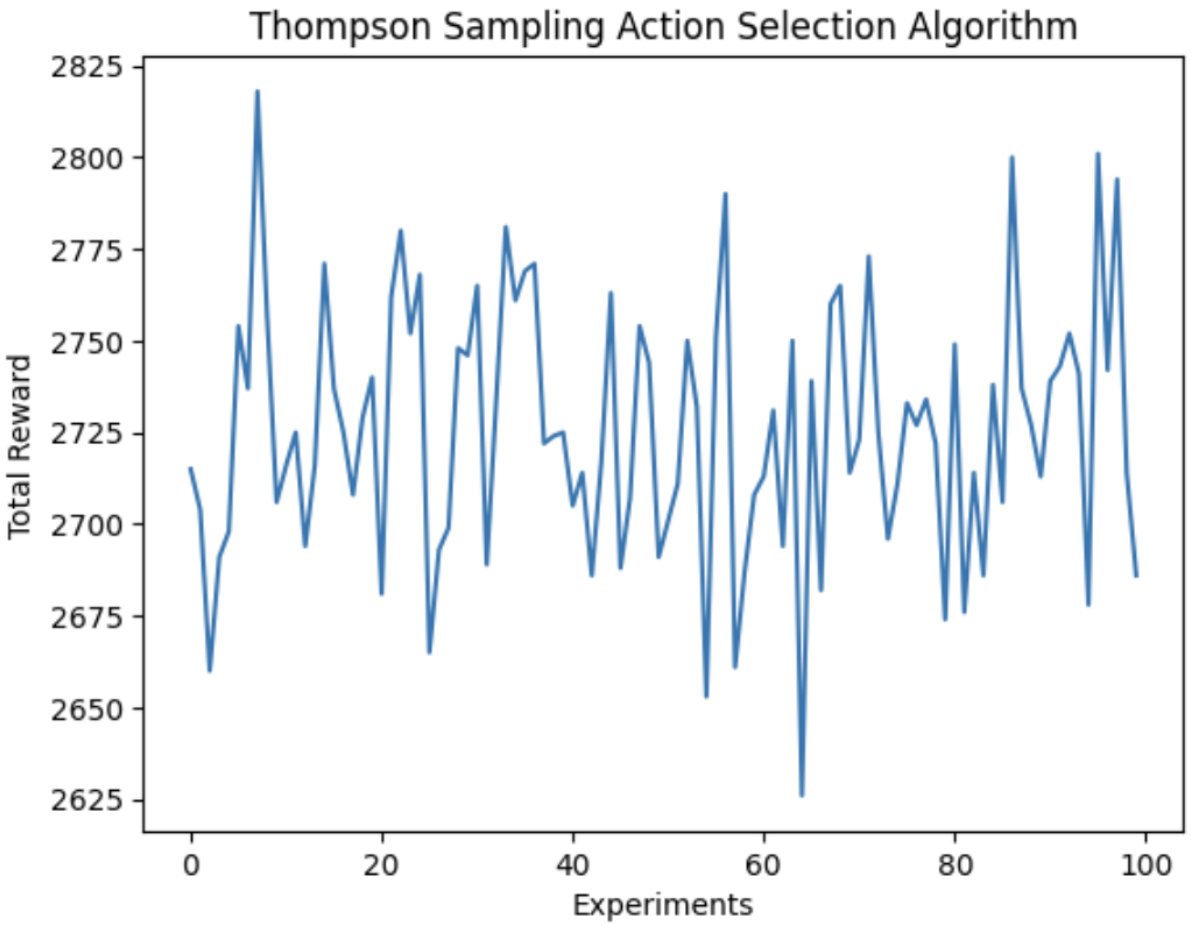



```
# Upper Confidence Bound Based Action Selection

import matplotlib.pyplot as plt
import numpy as np

N_ARMS = 6
ARM_REWARD_PROBABILITY = np.array([0.55, 0.45, 0.3, 0.4, 0.35, 0.48])
REWARD = np.ones(N_ARMS)
EXPERIMENTS = 100
ITERATIONS = 5000
C = 0.4

class UCB():
  def __init__(self, N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, C):
    self.N_ARMS = N_ARMS
    self.REWARD = REWARD
    self.ARM_REWARD_PROBABILITY = ARM_REWARD_PROBABILITY
    self.ITERATIONS = ITERATIONS
    self.C = C
    self.avg_reward = np.zeros(N_ARMS)
    self.times_picked = np.zeros(N_ARMS)
    self.random_number_generator = np.random.default_rng()

  def pick_arm(self, arm_index):
    random = self.random_number_generator.random()
    reward_obtained = 0

    if random < self.ARM_REWARD_PROBABILITY[arm_index]:
      reward_obtained = self.REWARD[arm_index]

    numerator = self.avg_reward[arm_index] * self.times_picked[arm_index] + reward_obtained
    denominator = self.times_picked[arm_index] + 1

    self.avg_reward[arm_index] = numerator / denominator
    self.times_picked[arm_index] += 1

    return reward_obtained
```

```
  def start_experiment(self):
    experiment_reward = 0

    # initially we pick all the arms once to avoid zero division error
    for i in range(self.N_ARMS):
      experiment_reward += self.pick_arm(i)

    for i in range(ITERATIONS - N_ARMS):
      best_arm = np.argmax(self.avg_reward + self.C * (np.array([np.log(i + N_ARMS)])/self.times_picked)**0.5)
      experiment_reward += self.pick_arm(best_arm)

    return experiment_reward

  def view_sample_experiment(self):
    all_rewards = []
    experiment_reward = 0

    # initially we pick all the arms once to avoid zero division error
    for i in range(self.N_ARMS):
      experiment_reward += self.pick_arm(i)
      all_rewards.append(experiment_reward)

    for i in range(ITERATIONS - N_ARMS):
      best_arm = np.argmax(self.avg_reward + self.C * (np.array([np.log(i + N_ARMS)])/self.times_picked)**0.5)
      experiment_reward += self.pick_arm(best_arm)
      all_rewards.append(experiment_reward)

    print("experiment_reward: ", experiment_reward)
    fig, ax = plt.subplots()
    ax.plot(range(ITERATIONS), all_rewards)
    ax.set(xlabel="Iterations", ylabel="Reward", title="Upper Confidence Bound Based Experiment")
    plt.show()

ucb = UCB(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, C)
ucb.view_sample_experiment()

a4_rewards = []
for e in range(EXPERIMENTS):
  ucb = UCB(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS, C)
  reward = ucb.start_experiment()
  a4_rewards.append(reward)

print("average_experiment_reward: ", np.mean(a4_rewards))
fig, ax = plt.subplots()
ax.plot(range(EXPERIMENTS), a4_rewards)
ax.set(xlabel="Experiments", ylabel="Total Reward", title="Upper Confidence Bound Based Action Selection Algorithm")
plt.show()
```

5. thompson sampling


Thompson Sampling Experiment
Reward
Iterations
0
500
1000
1500
2000
2500
0
1000
2000
3000
4000
5000




```
# Thompson Sampling

import matplotlib.pyplot as plt
import numpy as np

N_ARMS = 6
ARM_REWARD_PROBABILITY = np.array([0.55, 0.45, 0.3, 0.4, 0.35, 0.48])
REWARD = np.ones(N_ARMS)
EXPERIMENTS = 100
ITERATIONS = 5000

class Thompson_Sampling():
  def __init__(self, N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS):
    self.N_ARMS = N_ARMS
    self.REWARD = REWARD
    self.ARM_REWARD_PROBABILITY = ARM_REWARD_PROBABILITY
    self.ITERATIONS = ITERATIONS
    self.succ_fail = [[0,0] for i in range(N_ARMS)]
    self.random_number_generator = np.random.default_rng()

  def pick_arm(self, arm_index):
    random = self.random_number_generator.random()

    if random < self.ARM_REWARD_PROBABILITY[arm_index]:
      self.succ_fail[0] += REWARD[arm_index]
      self.succ_fail[1] += (1 - REWARD[arm_index])
      reward_obtained = REWARD[arm_index]
    else:
      self.succ_fail[arm_index][0] += 0
      self.succ_fail[arm_index][1] += (1 - 0)
      reward_obtained = 0

    return reward_obtained

  def start_experiment(self):
    experiment_reward = 0

    for i in range(self.ITERATIONS):
```

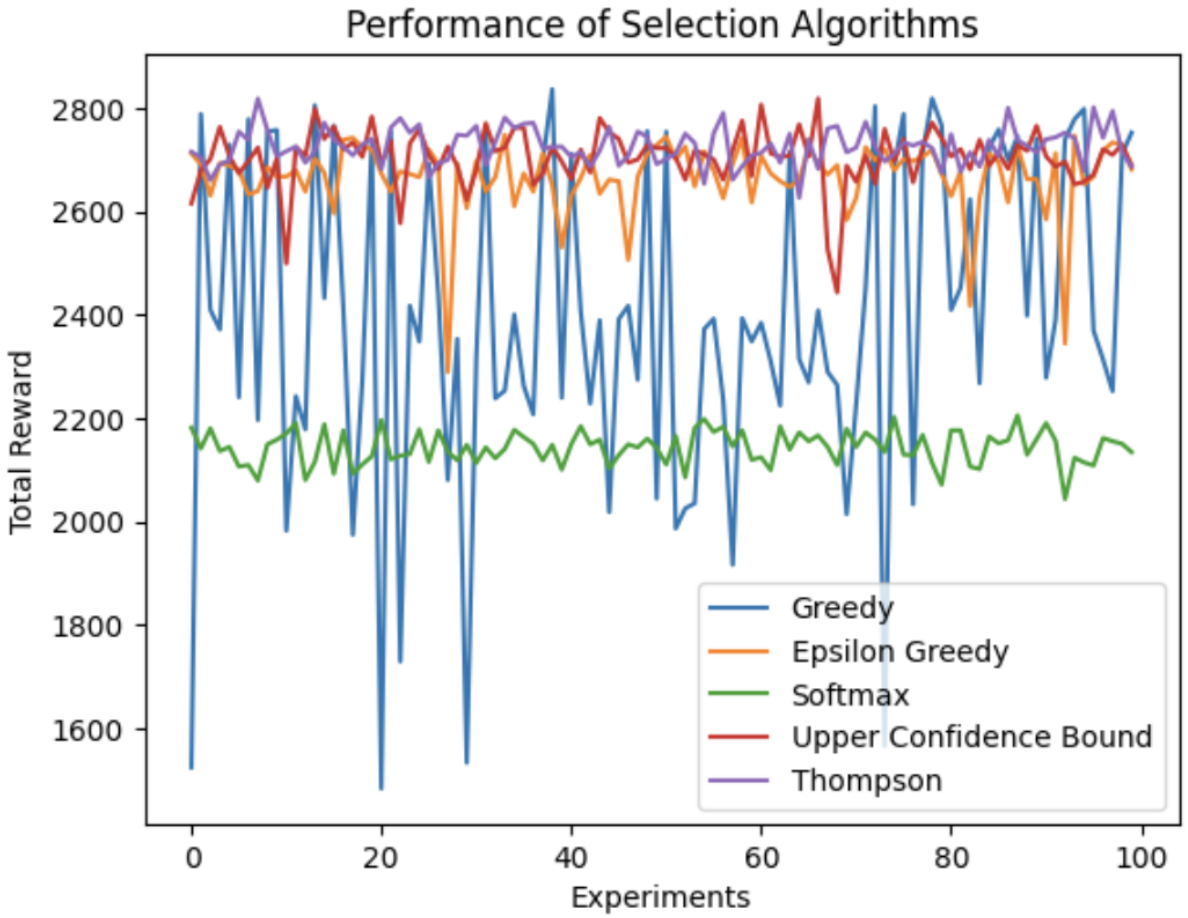
```
      # Sample a data point (thompson sampling) from all arms' Beta distrib
      samples = [np.random.beta(s+1, f+1) for s, f in self.succ_fail]  # add 1 because can't pass 0

      # Pick the arm with highest sampled estimate
      best_arm = np.argmax(samples)

      experiment_reward += self.pick_arm(best_arm)

    return experiment_reward

  def view_sample_experiment(self):
    all_rewards = []
    experiment_reward = 0

    for i in range(self.ITERATIONS):
      # Sample a data point (thompson sampling) from all arms' Beta distrib
      samples = [np.random.beta(s+1, f+1) for s, f in self.succ_fail]  # add 1 because can't pass 0

      # Pick the arm with highest sampled estimate
      best_arm = np.argmax(samples)

      experiment_reward += self.pick_arm(best_arm)
      all_rewards.append(experiment_reward)

    print("experiment_reward: ", experiment_reward)
    fig, ax = plt.subplots()
    ax.plot(range(ITERATIONS), all_rewards)
    ax.set(xlabel="Iterations", ylabel="Reward", title="Thompson Sampling Experiment")
    plt.show()

thompson = Thompson_Sampling(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS)
thompson.view_sample_experiment()

a5_rewards = []
for e in range(EXPERIMENTS):
  thompson = Thompson_Sampling(N_ARMS, REWARD, ARM_REWARD_PROBABILITY, ITERATIONS)
  reward = thompson.start_experiment()
  a5_rewards.append(reward)

print("average_experiment_reward: ", np.mean(a5_rewards))
fig, ax = plt.subplots()
ax.plot(range(EXPERIMENTS), a5_rewards)
ax.set(xlabel="Experiments", ylabel="Total Reward", title="Thompson Sampling Action Selection Algorithm")
plt.show()
```

overall comparison

The greedy algorithm has the largest variance for experiment reward. This may be due to how the greedy algorithm may focus on arms that have given them rewards at the start of the experiment despite them having a poorer reward function.

The softmax algorithm consistently gives a lower total reward compared to the other algorithms, which may be tuned using a hyperparameter such as temperature.

algorithm	average experiment reward
greedy	2397.08
epsilon greedy	2665.71
softmax	2142.85
UCB	2704.01
thompson sampling	2725.77




```
print("average reward for Greedy: ", np.mean(a1_rewards))
print("average reward for Epsilon Greedy: ", np.mean(a2_rewards))
print("average reward for Softmax: ", np.mean(a3_rewards))
print("average reward for Upper Confidence Bound: ", np.mean(a4_rewards))
print("average reward for Thompson: ", np.mean(a5_rewards))

fig, ax = plt.subplots()
ax.plot(range(EXPERIMENTS), a1_rewards, label="Greedy")
ax.plot(range(EXPERIMENTS), a2_rewards, label="Epsilon Greedy")
ax.plot(range(EXPERIMENTS), a3_rewards, label="Softmax")
ax.plot(range(EXPERIMENTS), a4_rewards, label="Upper Confidence Bound")
ax.plot(range(EXPERIMENTS), a5_rewards, label="Thompson")
ax.set(xlabel="Experiments", ylabel="Total Reward", title="Performance of Selection Algorithms")
plt.legend()
plt.show()
```

question 2

Q2. (Programming Exercise - 15 points)

Implement the Pong example described in class (Lecture 10) using the Gym or Gymnasium environment using both DQN and Reinforce. In the class, I explained how Reinforce is implemented on Pong. Here the target is for you to implement both DQN and Reinforce using a single hidden layer neural network (for either policy or value). Compare and contrast the training and testing performance of DQN and Reinforce.

Please provide the code and the results.

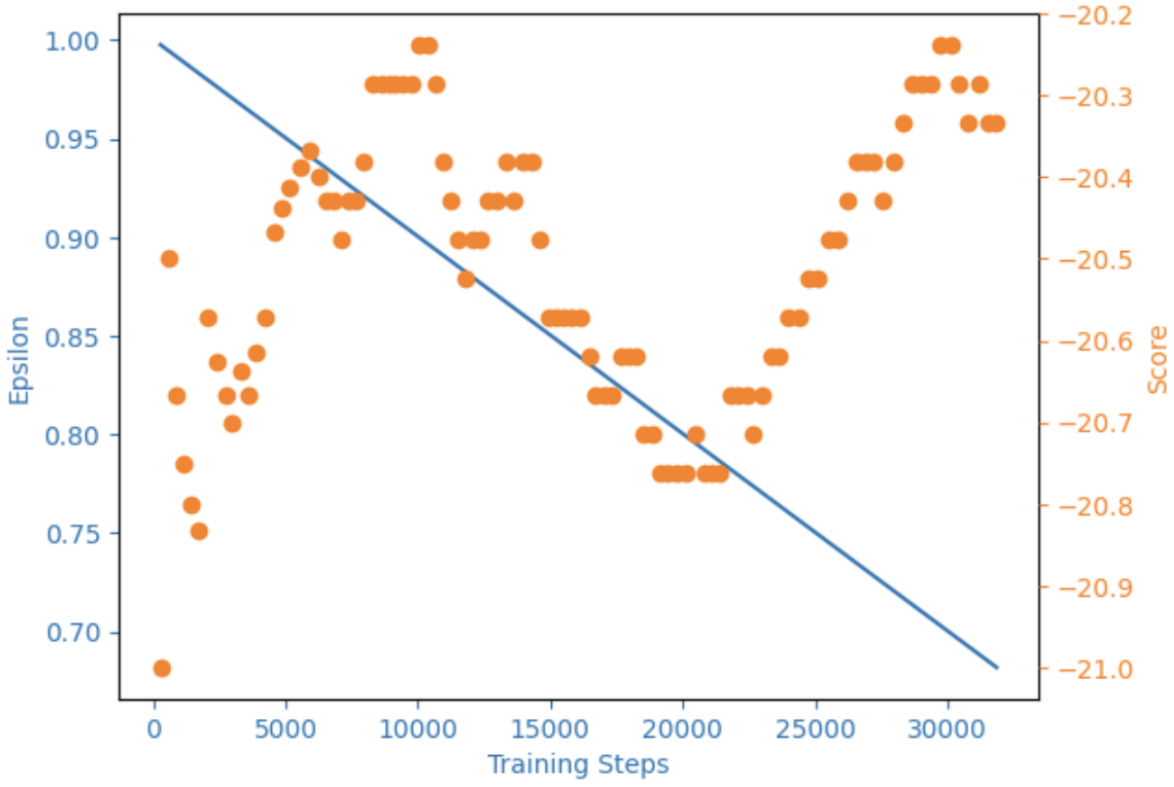
Note: You are free to study and explore other implementations, but you are expected to write your own code similar to what was described in class.

As Pong is a complex environment with a large state and action space, DQN may struggle to handle this complexity and may require a large number of training steps to learn an optimal policy. Hence, we can see larger fluctuations in the score obtained for the DQN model in comparison to a steadier upward slope for REINFORCE. This may be because REINFORCE updates parameters of the policy network to increase Q value over time, which forces the model to improve.

Deep Q Network

Reference: https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code/blob/master/DQN/deep_q_network.py

Training the model with 100 episodes, we can see that there is a general increase in the reward obtained but this is unstable, resulting in a wave shape.




```
import os
import torch as T
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np

class DeepQNetwork(nn.Module):
```

```
    def __init__(self, lr, n_actions, name, input_dims, chkpt_dir):
        super(DeepQNetwork, self).__init__()
        self.checkpoint_dir = chkpt_dir
        self.checkpoint_file = os.path.join(self.checkpoint_dir, name)

        self.conv1 = nn.Conv2d(input_dims[0], 32, 8, stride=4)
        self.conv2 = nn.Conv2d(32, 64, 4, stride=2)
        self.conv3 = nn.Conv2d(64, 64, 3, stride=1)

        fc_input_dims = self.calculate_conv_output_dims(input_dims)

        self.fc1 = nn.Linear(fc_input_dims, 512)
        self.fc2 = nn.Linear(512, n_actions)

        self.optimizer = optim.RMSprop(self.parameters(), lr=lr)

        self.loss = nn.MSELoss()
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def calculate_conv_output_dims(self, input_dims):
        state = T.zeros(1, *input_dims)
        dims = self.conv1(state)
        dims = self.conv2(dims)
        dims = self.conv3(dims)
        return int(np.prod(dims.size()))

    def forward(self, state):
        conv1 = F.relu(self.conv1(state))
        conv2 = F.relu(self.conv2(conv1))
        conv3 = F.relu(self.conv3(conv2))
        # conv3 shape is BS x n_filters x H x W
        conv_state = conv3.view(conv3.size()[0], -1)
        # conv_state shape is BS x (n_filters * H * W)
        flat1 = F.relu(self.fc1(conv_state))
        actions = self.fc2(flat1)

        return actions

    def save_checkpoint(self):
        print('... saving checkpoint ...')
        T.save(self.state_dict(), self.checkpoint_file)

    def load_checkpoint(self):
        print('... loading checkpoint ...')
        self.load_state_dict(T.load(self.checkpoint_file))
```

```
import numpy as np

class ReplayBuffer(object):
    def __init__(self, max_size, input_shape, n_actions):
        self.mem_size = max_size
        self.mem_cntr = 0
        self.state_memory = np.zeros((self.mem_size, *input_shape),
                                     dtype=np.float32)
        self.new_state_memory = np.zeros((self.mem_size, *input_shape),
                                         dtype=np.float32)

        self.action_memory = np.zeros(self.mem_size, dtype=np.int64)
        self.reward_memory = np.zeros(self.mem_size, dtype=np.float32)
        self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool)

    def store_transition(self, state, action, reward, state_, done):
        index = self.mem_cntr % self.mem_size
        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.action_memory[index] = action
        self.reward_memory[index] = reward
        self.terminal_memory[index] = done
        self.mem_cntr += 1

    def sample_buffer(self, batch_size):
        max_mem = min(self.mem_cntr, self.mem_size)
        batch = np.random.choice(max_mem, batch_size, replace=False)

        states = self.state_memory[batch]
        actions = self.action_memory[batch]
        rewards = self.reward_memory[batch]
        states_ = self.new_state_memory[batch]
        terminal = self.terminal_memory[batch]
```

```
        return states, actions, rewards, states_, terminal
```

```
import numpy as np
import torch as T

class DQNAgent(object):
    def __init__(self, gamma, epsilon, lr, n_actions, input_dims,
                 mem_size, batch_size, eps_min=0.01, eps_dec=5e-7,
                 replace=1000, algo=None, env_name=None, chkpt_dir='tmp/dqn'):
        self.gamma = gamma
        self.epsilon = epsilon
        self.lr = lr
        self.n_actions = n_actions
        self.input_dims = input_dims
        self.batch_size = batch_size
        self.eps_min = eps_min
        self.eps_dec = eps_dec
        self.replace_target_cnt = replace
        self.algo = algo
        self.env_name = env_name
        self.chkpt_dir = chkpt_dir
        self.action_space = [i for i in range(n_actions)]
        self.learn_step_counter = 0

        self.memory = ReplayBuffer(mem_size, input_dims, n_actions)

        self.q_eval = DeepQNetwork(self.lr, self.n_actions,
                                   input_dims=self.input_dims,
                                   name=self.env_name+'_'+self.algo+'_q_eval',
                                   chkpt_dir=self.chkpt_dir)

        self.q_next = DeepQNetwork(self.lr, self.n_actions,
                                   input_dims=self.input_dims,
                                   name=self.env_name+'_'+self.algo+'_q_next',
                                   chkpt_dir=self.chkpt_dir)

    def choose_action(self, observation):
        if np.random.random() > self.epsilon:
            state = T.tensor([observation],dtype=T.float).to(self.q_eval.device)
            actions = self.q_eval.forward(state)
            action = T.argmax(actions).item()
        else:
            action = np.random.choice(self.action_space)

        return action

    def store_transition(self, state, action, reward, state_, done):
        self.memory.store_transition(state, action, reward, state_, done)

    def sample_memory(self):
        state, action, reward, new_state, done = \
                                self.memory.sample_buffer(self.batch_size)

        states = T.tensor(state).to(self.q_eval.device)
        rewards = T.tensor(reward).to(self.q_eval.device)
        dones = T.tensor(done).to(self.q_eval.device)
        actions = T.tensor(action).to(self.q_eval.device)
        states_ = T.tensor(new_state).to(self.q_eval.device)

        return states, actions, rewards, states_, dones

    def replace_target_network(self):
        if self.learn_step_counter % self.replace_target_cnt == 0:
            self.q_next.load_state_dict(self.q_eval.state_dict())

    def decrement_epsilon(self):
        self.epsilon = self.epsilon - self.eps_dec \
                           if self.epsilon > self.eps_min else self.eps_min

    def save_models(self):
        self.q_eval.save_checkpoint()
        self.q_next.save_checkpoint()

    def load_models(self):
        self.q_eval.load_checkpoint()
        self.q_next.load_checkpoint()

    def learn(self):
        if self.memory.mem_cntr < self.batch_size:
            return
```

```
        self.q_eval.optimizer.zero_grad()

        self.replace_target_network()

        states, actions, rewards, states_, dones = self.sample_memory()
        indices = np.arange(self.batch_size)

        q_pred = self.q_eval.forward(states)[indices, actions]
        q_next = self.q_next.forward(states_).max(dim=1)[0]

        q_next[dones] = 0.0
        q_target = rewards + self.gamma*q_next

        loss = self.q_eval.loss(q_target, q_pred).to(self.q_eval.device)
        loss.backward()
        self.q_eval.optimizer.step()
        self.learn_step_counter += 1

        self.decrement_epsilon()
```

```
import collections
import cv2
import numpy as np
import matplotlib.pyplot as plt
import gym

def plot_learning_curve(x, scores, epsilons, lines=None):
    fig=plt.figure()
    ax=fig.add_subplot(111, label="1")
    ax2=fig.add_subplot(111, label="2", frame_on=False)

    ax.plot(x, epsilons, color="C0")
    ax.set_xlabel("Training Steps", color="C0")
    ax.set_ylabel("Epsilon", color="C0")
    ax.tick_params(axis='x', colors="C0")
    ax.tick_params(axis='y', colors="C0")

    N = len(scores)
    running_avg = np.empty(N)
    for t in range(N):
      running_avg[t] = np.mean(scores[max(0, t-20):(t+1)])

    ax2.scatter(x, running_avg, color="C1")
    ax2.axes.get_xaxis().set_visible(False)
    ax2.yaxis.tick_right()
    ax2.set_ylabel('Score', color="C1")
    ax2.yaxis.set_label_position('right')
    ax2.tick_params(axis='y', colors="C1")

    if lines is not None:
        for line in lines:
            plt.axvline(x=line)

    # plt.savefig(filename)

class RepeatActionAndMaxFrame(gym.Wrapper):
    def __init__(self, env=None, repeat=4, clip_reward=False, no_ops=0,
                 fire_first=False):
        super(RepeatActionAndMaxFrame, self).__init__(env)
        self.repeat = repeat
        self.shape = env.observation_space.low.shape
        self.frame_buffer = np.zeros_like((2, self.shape))
        self.clip_reward = clip_reward
        self.no_ops = no_ops
        self.fire_first = fire_first

    def step(self, action):
        t_reward = 0.0
        done = False
        for i in range(self.repeat):
            obs, reward, done, info = self.env.step(action)
            if self.clip_reward:
                reward = np.clip(np.array([reward]), -1, 1)[0]
            t_reward += reward
            idx = i % 2
            self.frame_buffer[idx] = obs
            if done:
                break

        max_frame = np.maximum(self.frame_buffer[0], self.frame_buffer[1])
        return max_frame, t_reward, done, info
```

```
    def reset(self):
        obs = self.env.reset()
        no_ops = np.random.randint(self.no_ops)+1 if self.no_ops > 0 else 0
        for _ in range(no_ops):
            _, _, done, _ = self.env.step(0)
            if done:
                self.env.reset()
        if self.fire_first:
            assert self.env.unwrapped.get_action_meanings()[1] == 'FIRE'
            obs, _, _, _ = self.env.step(1)

        self.frame_buffer = np.zeros_like((2,self.shape))
        self.frame_buffer[0] = obs

        return obs

class PreprocessFrame(gym.ObservationWrapper):
    def __init__(self, shape, env=None):
        super(PreprocessFrame, self).__init__(env)
        self.shape = (shape[2], shape[0], shape[1])
        self.observation_space = gym.spaces.Box(low=0.0, high=1.0,
                                    shape=self.shape, dtype=np.float32)

    def observation(self, obs):
        new_frame = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY)
        resized_screen = cv2.resize(new_frame, self.shape[1:],
                                    interpolation=cv2.INTER_AREA)
        new_obs = np.array(resized_screen, dtype=np.uint8).reshape(self.shape)
        new_obs = new_obs / 255.0

        return new_obs

class StackFrames(gym.ObservationWrapper):
    def __init__(self, env, repeat):
        super(StackFrames, self).__init__(env)
        self.observation_space = gym.spaces.Box(
                             env.observation_space.low.repeat(repeat, axis=0),
                             env.observation_space.high.repeat(repeat, axis=0),
                             dtype=np.float32)
        self.stack = collections.deque(maxlen=repeat)

    def reset(self):
        self.stack.clear()
        observation = self.env.reset()
        for _ in range(self.stack.maxlen):
            self.stack.append(observation)

        return np.array(self.stack).reshape(self.observation_space.low.shape)

    def observation(self, observation):
        self.stack.append(observation)

        return np.array(self.stack).reshape(self.observation_space.low.shape)

def make_env(env_name, shape=(84,84,1), repeat=4, clip_rewards=False,
             no_ops=0, fire_first=False):
    env = gym.make(env_name)
    env = RepeatActionAndMaxFrame(env, repeat, clip_rewards, no_ops, fire_first)
    env = PreprocessFrame(shape, env)
    env = StackFrames(env, repeat)

    return env
```

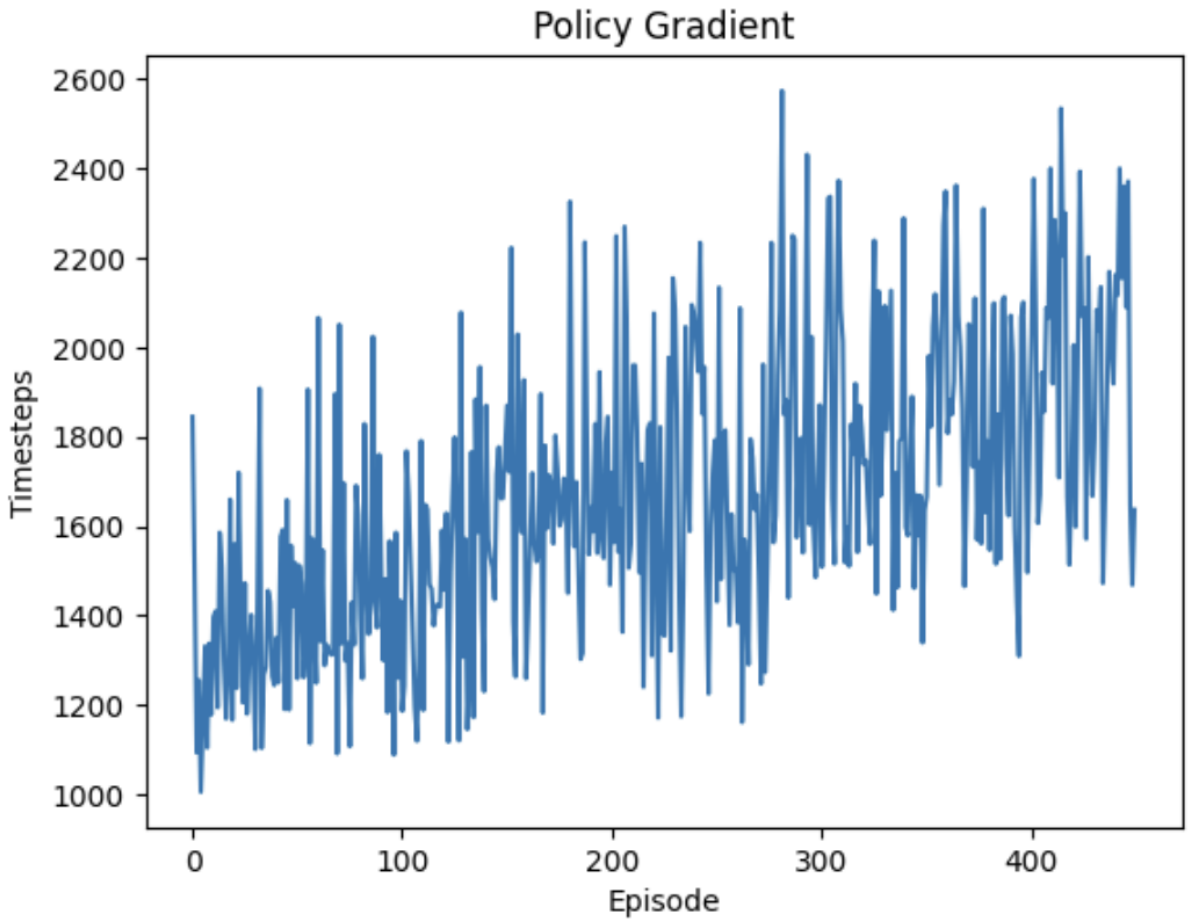
```
import gym
import numpy as np
from gym import wrappers

if __name__ == '__main__':
    env = make_env('Pong-v0')
    best_score = -np.inf
    load_checkpoint = False
    n_games = 200

    agent = DQNAgent(gamma=0.99, epsilon=1, lr=0.0001,
                     input_dims=(env.observation_space.shape),
                     n_actions=env.action_space.n, mem_size=50000, eps_min=0.1,
                     batch_size=32, replace=1000, eps_dec=1e-5,
                     chkpt_dir='models/', algo='DQNAgent',
                     env_name='Pong-v0')

    if load_checkpoint:
        agent.load_models()
```

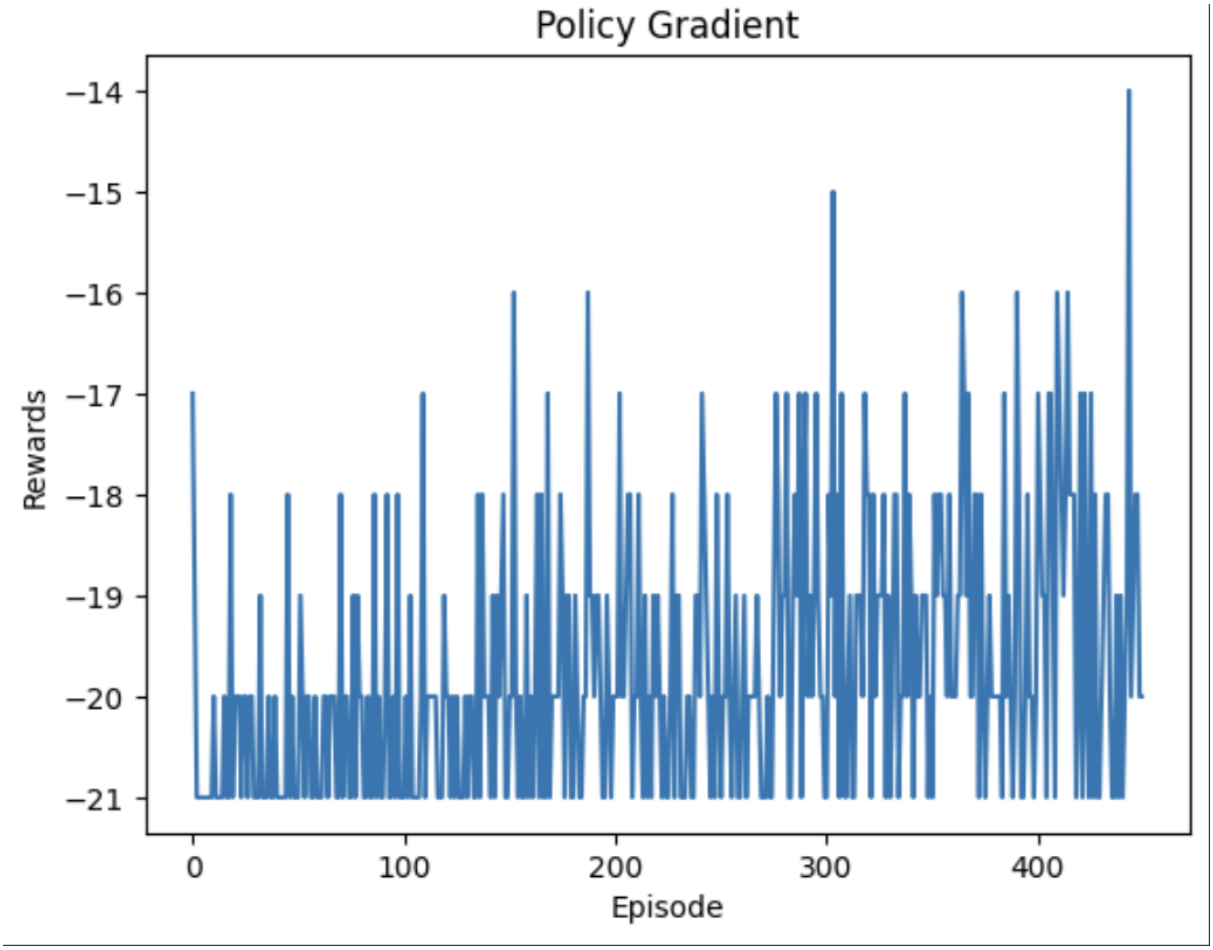
```
fname = agent.algo + '_' + agent.env_name + '_lr' + str(agent.lr) +'_' \
        + str(n_games) + 'games'
figure_file = 'plots/' + fname + '.png'
# if you want to record video of your agent playing, do a mkdir tmp && mkdir tmp/dqn-video
# and uncomment the following 2 lines.
#env = wrappers.Monitor(env, "tmp/dqn-video",
#                    video_callable=lambda episode_id: True, force=True)
n_steps = 0
scores, eps_history, steps_array = [], [], []

for i in range(n_games):
    done = False
    observation = env.reset()

    score = 0
    while not done:
        action = agent.choose_action(observation)
        observation_, reward, done, info = env.step(action)
        score += reward

        if not load_checkpoint:
            agent.store_transition(observation, action,
                                   reward, observation_, done)
            agent.learn()
        observation = observation_
        n_steps += 1
    scores.append(score)
    steps_array.append(n_steps)

    avg_score = np.mean(scores[-100:])
    print('episode: ', i,'score: ', score,
         ' average score %.1f' % avg_score, 'best score %.2f' % best_score,
        'epsilon %.2f' % agent.epsilon, 'steps', n_steps)

    if avg_score > best_score:
        #if not load_checkpoint:
            #agent.save_models()
        best_score = avg_score

    eps_history.append(agent.epsilon)

x = [i+1 for i in range(len(scores))]
plot_learning_curve(steps_array, scores, eps_history)
```

REINFORCE

Reference: https://towardsdatascience.com/intro-to-reinforcement-learning-pong-92a94aa0f84d

Training the model with 450 episodes, we can see that the model gradually improves in terms of the number of time steps and reward obtained.


Policy Gradient
Timesteps
Episode
2600
2400
2200
2000
1800
1600
1400
1200
1000
0
100
200
300
400




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

from gym import wrappers

# hyperparameters to tune
H = 200 # number of hidden layer neurons
batch_size = 10 # used to perform a RMS prop param update every batch_size steps
learning_rate = 1e-3 # learning rate used in RMS prop
gamma = 0.99 # discount factor for reward
decay_rate = 0.99 # decay factor for RMSProp leaky sum of grad^2

# Config flags - video output and res
resume = False # resume training from previous checkpoint (from save.p  file)?
render = False # render video output?

# model initialization
D = 75 * 80 # input dimensionality: 75x80 grid
if resume:
  model = pickle.load(open('save.p', 'rb'))
else:
  model = {}
  model['W1'] = np.random.randn(H,D) / np.sqrt(D) # "Xavier" initialization - Shape will be H x D
  model['W2'] = np.random.randn(H) / np.sqrt(H) # Shape will be H

grad_buffer = { k : np.zeros_like(v) for k,v in model.items() } # update buffers that add up gradients over a batch
rmsprop_cache = { k : np.zeros_like(v) for k,v in model.items() } # rmsprop memory

def sigmoid(x):
  return 1.0 / (1.0 + np.exp(-x)) # sigmoid "squashing" function to interval [0,1]

def prepro(I):
  """ prepro 210x160x3 uint8 frame into 6000 (75x80) 1D float vector """
  I = I[35:185] # crop - remove 35px from start & 25px from end of image in x, to reduce redundant parts of image (i.e. after ball pass
  I = I[::2,::2,0] # downsample by factor of 2.
  I[I == 144] = 0 # erase background (background type 1)
  I[I == 109] = 0 # erase background (background type 2)
```

```
  I[I != 0] = 1 # everything else (paddles, ball) just set to 1. this makes the image grayscale effectively
  return I.astype(np.float).ravel() # ravel flattens an array and collapses it into a column vector

def discount_rewards(r):
  """ take 1D float array of rewards and compute discounted reward """
  """ this function discounts from the action closest to the end of the completed game backwards
  so that the most recent action has a greater weight """
  discounted_r = np.zeros_like(r)
  running_add = 0
  for t in reversed(range(0, r.size)): # xrange is no longer supported in Python 3
    if r[t] != 0: running_add = 0 # reset the sum, since this was a game boundary (pong specific!)
    running_add = running_add * gamma + r[t]
    discounted_r[t] = running_add
  return discounted_r

def policy_forward(x):
  """This is a manual implementation of a forward prop"""
  h = np.dot(model['W1'], x) # (H x D) . (D x 1) = (H x 1) (200 x 1)
  h[h<0] = 0 # ReLU introduces non-linearity
  logp = np.dot(model['W2'], h) # This is a logits function and outputs a decimal.   (1 x H) . (H x 1) = 1 (scalar)
  p = sigmoid(logp)  # squashes output to  between 0 & 1 range
  return p, h # return probability of taking action 2 (UP), and hidden state

def policy_backward(eph, epx, epdlogp):
  """ backward pass. (eph is array of intermediate hidden states) """
  """ Manual implementation of a backward prop"""
  """ It takes an array of the hidden states that corresponds to all the images that were
  fed to the NN (for the entire episode, so a bunch of games) and their corresponding logp"""
  dW2 = np.dot(eph.T, epdlogp).ravel()
  dh = np.outer(epdlogp, model['W2'])
  dh[eph <= 0] = 0 # backpro prelu
  dW1 = np.dot(dh.T, epx)
  return {'W1':dW1, 'W2':dW2}

env = gym.make("Pong-v0")
# env = wrappers.Monitor(env, 'tmp/pong-base', force=True) # record the game as as an mp4 file
observation = env.reset()
prev_x = None # used in computing the difference frame
xs,hs,dlogps,drs = [],[],[],[]
running_reward = None
rewards = []
reward_sum = 0
episode_number = 0
MAX_EPISODES = 450

timesteps = []
timestep = 0

while True:
  if render: env.render()

  # preprocess the observation, set input to network to be difference image
  cur_x = prepro(observation)
  # we take the difference in the pixel input, since this is more likely to account for interesting information
  # e.g. motion
  x = cur_x - prev_x if prev_x is not None else np.zeros(D)
  prev_x = cur_x

  # forward the policy network and sample an action from the returned probability
  aprob, h = policy_forward(x)
  # The following step is randomly choosing a number which is the basis of making an action decision
  # If the random number is less than the probability of UP output from our neural network given the image
  # then go down.  The randomness introduces 'exploration' of the Agent
  action = 2 if np.random.uniform() < aprob else 3 # roll the dice! 2 is UP, 3 is DOWN, 0 is stay the same

  # record various intermediates (needed later for backprop).
  # This code would have otherwise been handled by a NN library
  xs.append(x) # observation
  hs.append(h) # hidden state
  y = 1 if action == 2 else 0 # a "fake label" - this is the label that we're passing to the neural network
  # to fake labels for supervised learning. It's fake because it is generated algorithmically, and not based
  # on a ground truth, as is typically the case for Supervised learning

  dlogps.append(y - aprob) # grad that encourages the action that was taken to be taken (see http://cs231n.github.io/neural-networks-2/

  # step the environment and get new measurements
  observation, reward, done, info = env.step(action)
  timestep += 1
  reward_sum += reward
  drs.append(reward) # record reward (has to be done after we call step() to get reward for previous action)

  if done: # an episode finished
    timesteps.append(timestep+1)
    timestep = 0
```

```
    episode_number += 1

    # stack together all inputs, hidden states, action gradients, and rewards for this episode
    epx = np.vstack(xs)
    eph = np.vstack(hs)
    epdlogp = np.vstack(dlogps)
    epr = np.vstack(drs)
    xs,hs,dlogps,drs = [],[],[],[] # reset array memory

    # compute the discounted reward backwards through time
    discounted_epr = discount_rewards(epr)
    # standardize the rewards to be unit normal (helps control the gradient estimator variance)
    discounted_epr -= np.mean(discounted_epr)
    discounted_epr /= np.std(discounted_epr)

    epdlogp *= discounted_epr # modulate the gradient with advantage (Policy Grad magic happens right here.)
    grad = policy_backward(eph, epx, epdlogp)
    for k in model: grad_buffer[k] += grad[k] # accumulate grad over batch

    # perform rmsprop parameter update every batch_size episodes
    if episode_number % batch_size == 0:
      for k,v in model.items():
        g = grad_buffer[k] # gradient
        rmsprop_cache[k] = decay_rate * rmsprop_cache[k] + (1 - decay_rate) * g**2
        model[k] += learning_rate * g / (np.sqrt(rmsprop_cache[k]) + 1e-5)
        grad_buffer[k] = np.zeros_like(v) # reset batch gradient buffer

    rewards.append(reward_sum)
    # boring book-keeping
    running_reward = reward_sum if running_reward is None else running_reward * 0.99 + reward_sum * 0.01
    print('resetting env. episode reward total was %f. running mean: %f' % (reward_sum, running_reward))

    if episode_number == MAX_EPISODES: break
    # if episode_number % 100 == 0: pickle.dump(model, open('save.p', 'wb'))
    reward_sum = 0
    observation = env.reset() # reset env
    prev_x = None

  # if reward != 0: # Pong has either +1 or -1 reward exactly when game ends.
    # print ('ep %d: game finished, reward: %f' % (episode_number, reward)) + ('' if reward == -1 else ' !!!!!!!!')

fig, ax = plt.subplots()
ax.plot(range(episode_number), timesteps)
ax.set(xlabel="Episode", ylabel="Timesteps", title="Policy Gradient")
plt.show()

fig, ax = plt.subplots()
ax.plot(range(episode_number), rewards)
ax.set(xlabel="Episode", ylabel="Rewards", title="Policy Gradient")
plt.show()
```

question 3

Q3. (By Hand or Programming – 10 points)

Consider a Restless Multi-Arm Bandit Problem, where there are four arms, all having the same MDP model.

States: 0, 1

Actions: 0 (passive), 1 (active)

Transition Probabilities:

Action = 0	0	1
0	0.7	0.3
1	0.25	0.75

Action = 1	0	1
0	0.2	0.8
1	0.05	0.95

Rewards, R(s): R(0) = 0; R(1) = 1

Horizon is 2 and Discount factor is 1. **Compute Whittle Index for each of the arms at the first-time step** (i.e., one more decision to go after the current decision), if they are in the following states:

Arm 1, State 0

Arm 2, State 1

Arm 3, State 1

Arm 4, State 0

Let us assume an initial value for the subsidy W = 0.6

$V^0(s', W) = 0$

$V^1(0, 0.6) = \max(R(0) + \sum_{\{s'\}} P^1_{\{s,s'\}} V^0(s', W), R(1) + W + \sum_{\{s'\}} P^0_{\{s,s'\}} V^0(s', W))$

$V^1(0, 0.6) = \max(0 + 0, 0 + 0.6 + 0) = 0.6$

$V^1(1, 0.6) = \max(R(1) + \sum_{\{s'\}} P^1_{\{s,s'\}} V^0(s', W), R(1) + W + \sum_{\{s'\}} P^0_{\{s,s'\}} V^0(s', W))$

$V^1(1, 0.6) = \max(1 + 0, 1 + 0.6 + 0) = 1.6$

$Q^2(0, 1, 0.6) = R(0) + \sum_{\{s'\}} P^1_{\{s,s'\}} V^1(s', W)$

$Q^2(0, 1, 0.6) = 0 + (0.2) * V^1(0, 0.6) + (0.8) * V^1(1, 0.6)$

$Q^2(0, 1, 0.6) = 0 + (0.2) * (0.6) + (0.8) * (1.6) = 1.4$

$Q^2(1,1,0.6) = R(1) + \sum_{\{s'\}} P^1_{\{s,s'\}} V^1(s', W)$
$Q^2(1,1,0.6) = 1 + (0.05) * V^1(0,0.6) + (0.95) * V^1(1,0.6)$
$Q^2(1,1,0.6) = 1 + (0.05) * (0.6) + (0.95) * (1.6) = 1.55$

$Q^2(0,0,0.6) = R(0) + 0.6 + \sum_{\{s'\}} P^0_{\{s,s'\}} V^1(s', W)$
$Q^2(0,0,0.6) = 0 + 0.6 + (0.7) * V^1(0,0.6) + (0.3) * V^1(1,0.6)$
$Q^2(0,0,0.6) = 0 + 0.6 + (0.7) * (0.6) + (0.3) * (1.6) = 1.5$

$Q^2(1,0,0.6) = R(1) + 0.6 + \sum_{\{s'\}} P^0_{\{s,s'\}} V^1(s', W)$
$Q^2(1,0,0.6) = 1 + 0.6 + (0.25) * V^1(0,0.6) + (0.75) * V^1(1,0.6)$
$Q^2(1,0,0.6) = 1 + 0.6 + (0.25) * (0.6) + (0.75) * (1.6) = 2.95$

$V^2(0,0.6) = \max(Q^2(0,1,0.6), Q^2(0,0,0.6))$
$V^2(0,0.6) = \max(1.4, 1.5) = 1.5$

$V^2(1,0.6) = \max(Q^2(1,1,0.6), Q^2(1,0,0.6))$
$V^2(1,0.6) = \max(1.55, 2.95) = 2.95$

arm 1, state 0

$\triangle_1 = Q^2(0,0,0.6) - Q^2(0,1,0.6) = 1.5 - 1.4 = 0.1$

Since delta > 0, we reduce W by 0.1 to 0.6 - 0.1 = 0.5

Hence, whittle index for arm 1 is 0.5

arm 2, state 1

$\triangle_1 = Q^2(1,0,0.6) - Q^2(1,1,0.6) = 2.95 - 1.55 = 1.4$

Since delta > 0, we reduce W by 1.4 to 0.6 - 1.4 = -0.8

Hence, whittle index for arm 2 is -0.8

arm 3, state 1

Whittle index for arm 3 is the same as for arm 2 because they have the same state

arm 4, state 0

Whittle index for arm 4 is the same as for arm 1 because they have the same state

question 4

Q4. (Programming – 10 points)

In this question, you are expected to code up the behavior cloning algorithm discussed in class.

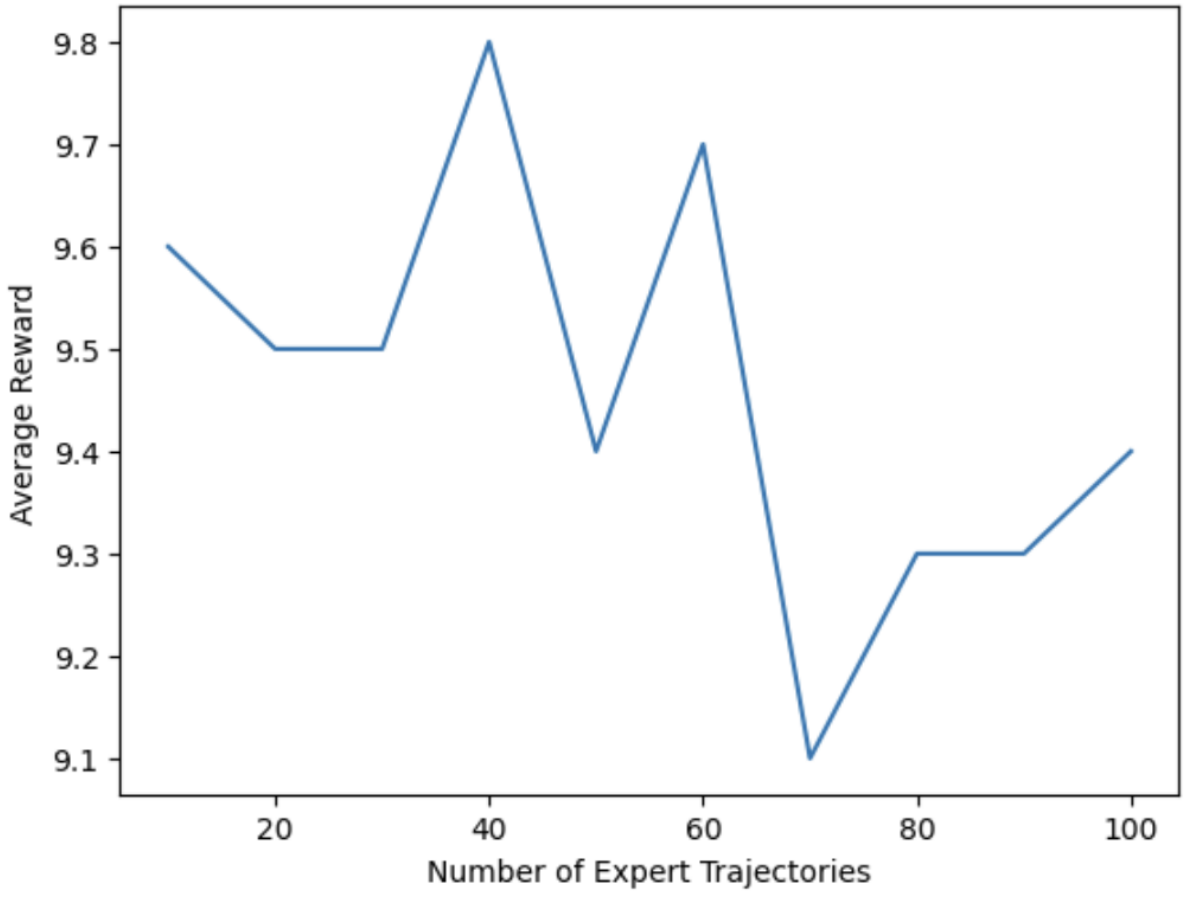
Please take the cartpole-v0 example from Gym and find expert trajectories. Please use the DQN algorithm from the StableBaselines library.

For the obtained set of expert trajectories, please train the policy neural network (single hidden layer) as per the Behavioral Cloning algorithm. The obtained policy network is the imitation learning solution.

Plot how well the policy learnt by Behavior Cloning performs across multiple simulations (10) on average for 5 different numbers of expert trajectories.

The quality of the learned policy is directly related to the quality and quantity of the expert's demonstrations. If there are too few expert trajectories, the behaviour cloning algorithm may not perform well on states that are not covered by the expert trajectories. On the other hand, if there are too many expert trajectories, the behaviour cloning algorithm might learn a policy that is too rigid and does not generalise well to new states or actions.

Based on the average reward obtained for a range of expert trajectories from 10 to 100 with a step size of 10, the optimal number of expert trajectories for the cartpole-v0 problem seems to be 40.



Reference: https://imitation.readthedocs.io/en/latest/tutorials/1_train_bc.html

```
import gymnasium as gym
from stable_baselines3 import DQN
from imitation.data import rollout
from imitation.algorithms import bc
import matplotlib.pyplot as plt

rng = np.random.default_rng()
vec_env = make_vec_env(
    "seals:seals/CartPole-v0",
    rng=np.random.default_rng(),
    post_wrappers=[
        lambda env, _: RolloutInfoWrapper(env)
    ],  # needed for computing rollouts later
)

expert = DQN("MlpPolicy", vec_env, verbose=1).learn(total_timesteps=10000, log_interval=4)

bc_rewards = []
NUM_TRAJECTORIES = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

for num_trajectories in NUM_TRAJECTORIES:

  rollouts = rollout.rollout(
      expert,
      vec_env,
      rollout.make_sample_until(min_timesteps=None, min_episodes=num_trajectories),
      rng=rng,
  )
  transitions = rollout.flatten_trajectories(rollouts)

  print(
      f"""The `rollout` function generated a list of {len(rollouts)} {type(rollouts[0])}.
  After flattening, this list is turned into a {type(transitions)} object containing {len(transitions)} transitions.
  The transitions object contains arrays for: {', '.join(transitions.__dict__.keys())}."
  """
  )

  bc_trainer = bc.BC(
      observation_space=env.observation_space,
      action_space=env.action_space,
      demonstrations=transitions,
      rng=rng,
  )

  reward_before_training, _ = evaluate_policy(bc_trainer.policy, env, 10)
  print(f"Reward before training: {reward_before_training}")

  bc_trainer.train(n_epochs=1)

  reward_after_training, _ = evaluate_policy(bc_trainer.policy, env, 10)
  print(f"Reward after training: {reward_after_training}")

  bc_rewards.append(reward_after_training)

plt.plot(NUM_TRAJECTORIES, bc_rewards)
plt.xlabel('Number of Expert Trajectories')
plt.ylabel('Average Reward')
plt.show()
```