CS 422 - Assignment 2

Bharat Gangwani

1 Question 1

As seen in Figure 1, Greedy outperforms all other strategies with comparable performance by UCB, Thompson Sampling and Epsilon Greedy. All these strategies perform the best because they converge on the best arm quickly (right after initialisation in the case of Greedy). Softmax performs the worst because it does not converge on the best arm quickly and hence does not exploit the best arm as much as the other strategies. Epsilon Greedy's long-term average reward would equal 0.9*0.55 = 0.495 ¡ 0.55 which is Greedy's long-term average reward.

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

mab = np.array([0.55, 0.45, 0.3, 0.4, 0.35, 0.48])

# Greedy
def greedy(mab, n, T):
    armHist = dict()
    rewardHist = []
    # Sample all arms T times to initate armHist
    for _ in range(T):
        for i, p in enumerate(mab):
            reward = int(np.random.random() < p)
            rewardHist.append(reward)
            if i not in armHist:
                armHist[i] = [reward]
            else:
                armHist[i].append(reward)

    for i in range(n-T*len(mab)):
        arm = np.argmax([np.mean(i) for i in armHist.values()])
        reward = int(np.random.random() < mab[arm])
        rewardHist.append(reward)
        armHist[arm].append(reward)

    return rewardHist

# Epsilon Greedy
def epsilonGreedy(mab, n, T, epsilon):
    armHist = dict()
    rewardHist = []
    # Sample all arms T times to initate armHist
    for _ in range(T):
        for i, p in enumerate(mab):
            reward = int(np.random.random() < p)
```

```
            rewardHist.append(reward)
            if i not in armHist:
                armHist[i] = [reward]
            else:
                armHist[i].append(reward)

    for i in range(n-T*len(mab)):
        if np.random.random() < epsilon:
            arm = np.random.randint(len(mab))
        else:
            arm = np.argmax([np.mean(i) for i in armHist.values()])
        reward = int(np.random.random() < mab[arm])
        rewardHist.append(reward)
        armHist[arm].append(reward)

    return rewardHist

# Softmax
def softmax(mab, n, T):
    armHist = dict()
    rewardHist = []

    # Sample all arms T times to initate armHist
    for _ in range(T):
        for i, p in enumerate(mab):
            reward = int(np.random.random() < p)
            rewardHist.append(reward)
            if i not in armHist:
                armHist[i] = [reward]
            else:
                armHist[i].append(reward)

    softMax = lambda x: np.exp(x) / np.sum(np.exp(x))
    for i in range(n-T*len(mab)):
        arm = np.random.choice(list(armHist.keys()), p = softMax([np.mean(i) for i in
             armHist.values()]))
        reward = int(np.random.random() < mab[arm])
        rewardHist.append(reward)
        armHist[arm].append(reward)

    return rewardHist

# UCB
def ucb(mab, n, T, c):
    armHist = dict()
    rewardHist = []

    # Sample all arms T times to initate armHist
    for _ in range(T):
        for i, p in enumerate(mab):
            reward = int(np.random.random() < p)
            rewardHist.append(reward)
            if i not in armHist:
                armHist[i] = [reward]
            else:
```

```
            armHist[i].append(reward)

    for i in range(n-T*len(mab)):
        arm = np.argmax([np.mean(i) + c*np.sqrt(np.log(len(rewardHist)) / len(i)) for
             i in armHist.values()])
        reward = int(np.random.random() < mab[arm])
        rewardHist.append(reward)
        armHist[arm].append(reward)

    return rewardHist

# Thompson Sampling
def thompson(mab, n, T):
    armHist = dict()
    rewardHist = []

    # Sample all arms T times to initate armHist
    for _ in range(T):
        for i, p in enumerate(mab):
            reward = int(np.random.random() < p)
            rewardHist.append(reward)
            if i not in armHist:
                armHist[i] = [reward]
            else:
                armHist[i].append(reward)

    for i in range(n-T*len(mab)):
        arm = np.argmax([np.random.beta(sum(i) + 1, len(i) - sum(i) + 1) for i in
             armHist.values()])
        reward = int(np.random.random() < mab[arm])
        rewardHist.append(reward)
        armHist[arm].append(reward)

    return rewardHist
```

```
n = 5000
T = 100
epsilon = 0.1
N = 100
c = np.sqrt(2)

greedyRewards = np.zeros([n, 100])
epsilonGreedyRewards = np.zeros([n, 100])
softmaxRewards = np.zeros([n, 100])
ucbRewards = np.zeros([n, 100])
thompsonRewards = np.zeros([n, 100])

for i in tqdm(range(100)):
    greedyRewards[:, i] = greedy(mab, n, T)

for i in tqdm(range(100)):
    epsilonGreedyRewards[:, i] = epsilonGreedy(mab, n, T, epsilon)

for i in tqdm(range(100)):
    softmaxRewards[:, i] = softmax(mab, n, T)
```

```python
for i in tqdm(range(100)):
    ucbRewards[:, i] = ucb(mab, n, T, c)

for i in tqdm(range(100)):
    thompsonRewards[:, i] = thompson(mab, n, T)

def moving_average(a, n=3):
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n

fig, ax = plt.subplots(1, 2, figsize = (15, 5))
ax[0].plot(np.mean(greedyRewards, axis = 1))
ax[1].plot(moving_average(np.mean(greedyRewards, axis = 1), n = 100))
ax[0].set_title('Greedy Average Reward')
ax[1].set_title('Greedy Moving Average Reward, window = 100')

fig, ax = plt.subplots(1, 2, figsize = (15, 5))
ax[0].plot(np.mean(epsilonGreedyRewards, axis = 1))
ax[1].plot(moving_average(np.mean(epsilonGreedyRewards, axis = 1), n = 100))
ax[0].set_title('Epsilon Greedy Average Reward')
ax[1].set_title('Epsilon Greedy Moving Average Reward, window = 100')

fig, ax = plt.subplots(1, 2, figsize = (15, 5))
ax[0].plot(np.mean(softmaxRewards, axis = 1))
ax[1].plot(moving_average(np.mean(softmaxRewards, axis = 1), n = 100))
ax[0].set_title('Softmax Average Reward')
ax[1].set_title('Softmax Moving Average Reward, window = 100')

fig, ax = plt.subplots(1, 2, figsize = (15, 5))
ax[0].plot(np.mean(ucbRewards, axis = 1))
ax[1].plot(moving_average(np.mean(ucbRewards, axis = 1), n = 100))
ax[0].set_title('UCB Average Reward')
ax[1].set_title('UCB Moving Average Reward, window = 100')

fig, ax = plt.subplots(1, 2, figsize = (15, 5))
ax[0].plot(np.mean(thompsonRewards, axis = 1))
ax[1].plot(moving_average(np.mean(thompsonRewards, axis = 1), n = 100))
ax[0].set_title('Thompson Average Reward')
ax[1].set_title('Thompson Moving Average Reward, window = 100')
```

2 Question 2

Both DQN and REINFORCE don't perform well out of sample. Within training, DQN seems to slowly improve its reward compared to REINFORCE. Given the sparsity of rewards in Pong, I reward engineered the difference in y coordinates of the ball and the player paddle. However, the scale of the reward seems to either have been inappropriate. Given more time, more reward engineering could yield better results such as scaling wins or identifying ball hits (when the paddle touches the ball) and rewarding those. As it is, Pong has very sparse rewards and hence it is difficult to train a model to play it well.

2.1 DQN

```
import gymnasium as gym
import random
import matplotlib.pyplot as plt
from collections import deque, namedtuple
import tensorflow as tf
import numpy as np
from tqdm import tqdm

Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward',
    'final_state_bool'))
ramDict = dict(player_y=51, player_x=46, enemy_y=50, enemy_x=45, ball_x=49,
    ball_y=54) # Retrieved from github.com/mila-iqia/atari-representation-learning

class ReplayMemory(object):

    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        sample = random.sample(self.memory, batch_size)
        return sample

    def __len__(self):
        return len(self.memory)

def create_model(num_actions):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(300, activation='relu'),
        tf.keras.layers.Dense(num_actions)
    ])
    return model

def map_action(action):
    action_map = {0:0, 1:4, 2:5}
    return action_map[action]

def moving_average(a, n=3):
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n

def process_state(state):
    state = state.reshape(1, -1)
    state = state/255
    return state

env = gym.make("ALE/Pong-ram-v5") # Since we aren't using a convolution layer, we
    can use the ram version of the game
state, info = env.reset()
state = process_state(state)
```

```python
n_actions = 3 # Reduce the action space to only the relevant actions

device = tf.device("/GPU:0")
weightDir = "./q2a/policyWeights"

BATCH_SIZE = 128
GAMMA = 0.99
EPS_START = 1.0
EPS_END = 0.3
EPS_DECAY = 50000
TAU = 0.005
LR = 1e-4

loss_object = tf.keras.losses.Huber()
optimizer = tf.keras.optimizers.Adam(learning_rate=LR, clipvalue = 100)


policy_net = create_model(n_actions)
target_net = create_model(n_actions)

policy_net(state)
target_net(state)
target_net.set_weights(policy_net.get_weights())

memory = ReplayMemory(10000)

def select_action(state, steps_done):
    sample = random.random()
    eps_threshold = EPS_END + (EPS_START - EPS_END) * np.exp(-1.0 * steps_done /
        EPS_DECAY)

    if sample > eps_threshold:
        return policy_net(state).numpy().argmax()
    else:
        return random.randrange(0, n_actions)

@tf.function
def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
    transitions = memory.sample(BATCH_SIZE)
    # Transpose the batch (see https://stackoverflow.com/a/19343/3343043 for
        detailed explanation). This converts batch-array of Transitions to
        Transition of batch-arrays.
    batch = Transition(*zip(*transitions))

    # Compute a mask of non-final states and concatenate the batch elements (a
        final state would've been the one after which simulation ended)
    final_mask = tf.convert_to_tensor(batch.final_state_bool)
    non_final_next_states = tf.reshape(tf.convert_to_tensor([s.next_state for s in
        transitions if not s.final_state_bool]), [sum(~final_mask.numpy()), -1])

    state_batch = tf.reshape(tf.convert_to_tensor(batch.state), [BATCH_SIZE, -1])
    action_batch = tf.reshape(tf.convert_to_tensor(batch.action), [BATCH_SIZE, -1])
    reward_batch = tf.reshape(tf.convert_to_tensor(batch.reward), [BATCH_SIZE, -1])
```

```
    reward_batch = (reward_batch - tf.reduce_mean(reward_batch)) /
        tf.math.reduce_std(reward_batch)

    with tf.GradientTape(watch_accessed_variables=False) as tape:
        tape.watch(policy_net.trainable_variables)

        # Compute Q(s_t, a) - the model computes Q(s_t), then we select the columns
            of actions taken. These are the actions which would've been taken for
            each batch state according to policy_net
        state_action_values = tf.gather(policy_net(state_batch), action_batch, axis
            = 1, batch_dims = 1)

        # Compute V(s_{t+1}) for all next states. Expected values of actions for
            non_final_next_states are computed based on the "older" target_net;
            selecting their best reward with max(1)[0]. This is merged based on the
            mask, such that we'll have either the expected state value or 0 in case
            the state was final.
        next_state_values = np.zeros([BATCH_SIZE, 1])
        next_state_values[~final_mask] =
            tf.reshape(tf.reduce_max(target_net(non_final_next_states), axis = 1),
            [sum(~final_mask.numpy()), 1])
        # Compute the expected Q values
        expected_state_action_values = (next_state_values * GAMMA) + reward_batch

        # Compute Huber loss
        loss = loss_object(state_action_values, expected_state_action_values)


    # Optimize the model
    gradients = tape.gradient(loss, policy_net.trainable_variables)
    optimizer.apply_gradients(zip(gradients, policy_net.trainable_variables))

num_episodes = 500
sumr = 0
steps_done = 0
rewards = []

for i_episode in tqdm(range(0, num_episodes)):
    # Initialize the environment and get its state
    state, info = env.reset(seed = i_episode)
    state = process_state(state)
    done = False
    while not done:
        action = select_action(state, steps_done)
        steps_done += 1
        exec_action = map_action(action)
        next_state, reward, terminated, truncated, info = env.step(exec_action)
        next_state = process_state(next_state)
        done = terminated or truncated

        reward -= abs(next_state[0][ramDict["ball_y"]] -
            next_state[0][ramDict["player_y"]]) # Punishment for not moving towards
            the ball

        sumr += reward
```

```
        # Store the transition in memory
        memory.push(state, action, next_state, reward, done)

        # Move to the next state
        state = next_state

        # Perform one step of the optimization (on the policy network)
        optimize_model()

        new_weights = [TAU*x + (1-TAU)*y for x,y in zip(policy_net.get_weights(),
             target_net.get_weights())]
        target_net.set_weights(new_weights)

        if done:
            if (i_episode + 1) % 50 == 0:
                policy_net.save_weights(weightDir + "_" + str(i_episode+1))
            rewards.append(sumr)
            if (i_episode % 100) == 0:
                print('episode: %3d \t return: %.3f' % (i_episode, np.mean(rewards)))
            sumr = 0
            break
```

```
# Test
from time import sleep
rewards = []
num_episodes = 10
sumr = 0
env = gym.make("ALE/Pong-ram-v5") # Since we aren't using a convolution layer, we
     can use the ram version of the game
state, info = env.reset()

for i_episode in tqdm(range(num_episodes)):
    # Initialize the environment and get its state
    state, info = env.reset(seed=i_episode)
    state = process_state(state)
    done = False
    while not done:
        action = policy_net(state).numpy().argmax()
        next_state, reward, terminated, truncated, info = env.step(action)
        next_state = process_state(next_state)
        done = terminated or truncated

        sumr += reward

        if done:
            rewards.append(sumr)
            sumr = 0
            break

plt.plot(rewards)
```

2.2 REINFORCE

```
import gymnasium as gym
import tensorflow as tf
import tensorflow_probability as tfp
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

def create_model(number_observation_features: int, number_actions: int) ->
    tf.keras.Sequential:

    hidden_layer_features = 128

    return tf.keras.Sequential([
        tf.keras.layers.Dense(hidden_layer_features, activation='relu',
            input_shape=(number_observation_features,)),
        tf.keras.layers.Dense(number_actions)]
    )


def get_policy(model: tf.keras.Sequential, observation: np.ndarray) ->
    tfp.distributions.Categorical:

    logits = model(observation)

    # Categorical will also normalize the logits for us
    return tfp.distributions.Categorical(logits=logits)


def get_action(policy: tfp.distributions.Categorical) -> tuple[int, tf.Tensor]:

    action = policy.sample() # Unit tensor

    # Converts to an int, as this is what Gym environments require
    action_int = action.numpy()[0]

    # Calculate the log probability of the action, which is required for
    # calculating the loss later
    log_probability_action = policy.log_prob(action)

    return action_int, log_probability_action


def calculate_loss(epoch_log_probability_actions: tf.Tensor, epoch_action_rewards:
    tf.Tensor) -> tf.Tensor:
    epoch_action_rewards = (epoch_action_rewards -
        tf.reduce_mean(epoch_action_rewards))/tf.math.reduce_std(epoch_action_rewards)
        # Normalize rewards
    return -tf.reduce_mean(epoch_log_probability_actions * epoch_action_rewards)


def train_one_epoch(
        env: gym.Env,
        model: tf.keras.Sequential,
```

```
    optimizer: tf.keras.optimizers.Adam,
    max_timesteps=5000,
    episode_timesteps=1500,
    rewardType = "act") -> float:

epoch_total_timesteps = 0

# Returns from each episode (to keep track of progress)
epoch_returns: list[float] = []

# Action log probabilities and rewards per step (for calculating loss)
epoch_log_probability_actions = []
epoch_action_rewards = []

with tf.GradientTape(watch_accessed_variables=False) as tape:
    tape.watch(model.trainable_variables)
    # Loop through episodes
    while True:

        # Stop if we've done over the total number of timesteps
        if epoch_total_timesteps > max_timesteps:
            break

        # Running total of this episode's rewards
        episode_reward: float = 0
        episode_rewards: list[float] = []

        # Reset the environment and get a fresh observation
        observation, info = env.reset()
        observation = process_state(observation)

        # Loop through timesteps until the episode is done (or the max is hit)
        for timestep in range(episode_timesteps):
            epoch_total_timesteps += 1

            # Get the policy and act
            policy = get_policy(model, observation)
            action, log_probability_action = get_action(policy)
            action = map_action(action)
            observation, reward, terminated, truncated, info = env.step(action)
            observation = process_state(observation)
            done = terminated or truncated

            # Add the reward to the episode total
            episode_rewards.append(-abs(observation[0][ramDict["ball_y"]] -
                observation[0][ramDict["player_y"]])) # Punishment for not
                moving towards the ball
            episode_reward += reward

            # Add epoch action log probabilities
            epoch_log_probability_actions.append(log_probability_action)

            # Finish the action loop if this episode is done
            if done:
                # Add one reward per timestep
```

```
                if rewardType == "const":
                    epoch_action_rewards.extend([x + episode_reward for x in
                        episode_rewards])
                else:
                    epoch_action_rewards.extend([episode_reward for _ in
                        range(timestep + 1)])

                break

            # Increment the epoch returns
            epoch_returns.append(episode_reward)

        epoch_loss =
            calculate_loss(tf.convert_to_tensor(epoch_log_probability_actions,
            tf.float64), tf.convert_to_tensor(epoch_action_rewards, tf.float64))

    # Calculate the policy gradient, and use it to step the weights & biases
    gradients = tape.gradient(epoch_loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    return float(np.mean(epoch_returns))

ramDict = dict(player_y=51, player_x=46, enemy_y=50, enemy_x=45, ball_x=49,
    ball_y=54) # Retrieved from github.com/mila-iqia/atari-representation-learning
def process_state(state):
    state = state.reshape(1, -1)
    state = state/255
    return state

def map_action(action):
    action_map = {0:0, 1:4, 2:5}
    return action_map[action]

def train(epochs=50, rewardType = "act") -> None:

    # Create the Gym Environment
    env = gym.make('ALE/Pong-ram-v5')

    # Use random seeds (to make experiments deterministic)
    tf.random.set_seed(0)

    # Create the MLP model
    number_observation_features = env.observation_space.shape[0]
    number_actions = 3
    model = create_model(number_observation_features, number_actions)

    # Create the optimizer
    optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-2)

    returns_over_epochs = []
    # Loop for each epoch
    for epoch in tqdm(range(epochs)):
        average_return = train_one_epoch(env, model, optimizer, rewardType =
            rewardType)
        returns_over_epochs.append(average_return)
```

```python
        print('epoch: %3d \t return: %.3f' % (epoch, average_return))
        if (epoch+1) % 10 == 0:
            model.save_weights(f"./q2b/{rewardType}RewardModel{epoch}")

    plt.plot(returns_over_epochs)
    return model, returns_over_epochs

policyNet, train_returnsAct = train(30)
```

```python
testEnv = gym.make('ALE/Pong-ram-v5')
policyNet = create_model(testEnv.observation_space.shape[0], 3)
policyNet.load_weights("./q2b/constRewardModel49")
epsRewards = []
for _ in tqdm(range(10)):
    observation, info = testEnv.reset()
    observation = process_state(observation)
    episode_reward = 0
    while True:
        policy = get_policy(policyNet, observation)
        action, _ = get_action(policy)
        action = map_action(action)
        observation, reward, terminated, truncated, info = testEnv.step(action)
        observation = process_state(observation)
        episode_reward += reward
        if terminated or truncated:
            epsRewards.append(episode_reward)
            break

plt.plot(epsRewards)
```

3 Question 3

$$R(0) = 0$$
$$R(1) = 0$$
$$V^0(0, W) = V^0(1, W) = 0$$
$$V^1(0, W) = W$$
$$V^1(1, W) = 1 + W$$
$$Q^2(0, 0, W) = 0.3 + 2W$$
$$Q^2(0, 1, W) = 0.8 + W$$
$$Q^2(1, 0, W) = 1.75 + 2W$$
$$Q^2(1, 1, W) = 1.95 + 2$$
$$\text{For arms 1 and 4 :}$$
$$0.3 + 2W = 0.8 + W$$
$$\implies W = 0.5$$
$$\text{For arms 2 and 3 :}$$
$$1.75 + 2W = 1.95 + 2$$
$$\implies W = 0.2$$

4 Question 4

The Behavioural Cloning Algorithm was trained using 50 trajectories. As seen in the average similarity action reward plot (Figure 6), the similarity between the expert and the agent is quite variable and hasn't converged. A higher number of expert trajectories and a deeper neural network could help reach convergence and improve the similarity score. The similarity is the fraction of actions that are the same between the agent and the expert. The similarity is averaged over 10 trajectories.

```
from stable_baselines3 import DQN
import matplotlib.pyplot as plt
expert = DQN("MlpPolicy", "CartPole-v1", device = "cuda").learn(10000)

import numpy as np
import gymnasium as gym
from stable_baselines3.common.evaluation import evaluate_policy

from imitation.algorithms import bc
from imitation.data import rollout
from imitation.data.wrappers import RolloutInfoWrapper
from imitation.policies.serialize import load_policy
from imitation.util.util import make_vec_env

rng = np.random.default_rng(0)
env = make_vec_env("CartPole-v1", rng=rng, n_envs=1, post_wrappers=[lambda env, _:
    RolloutInfoWrapper(env)]) # for computing rollouts

rollouts = rollout.rollout(
```

```
    expert,
    env,
    rollout.make_sample_until(min_timesteps=None, min_episodes=50),
    rng=rng,
)
transitions = rollout.flatten_trajectories(rollouts)

bc_trainer = bc.BC(
    observation_space=env.observation_space,
    action_space=env.action_space,
    demonstrations=transitions,
    rng=rng,
)
bc_trainer.train(n_epochs=1)

import random

rewards = []
for __ in range(5):
    trajectory = random.sample(rollouts, 1)[0]
    simRew = []
    for _ in range(10):
        action, _ = bc_trainer.policy.predict(trajectory.obs[:-1])
        simRew.append((action == trajectory.acts).astype(int))
    rewards.append(np.stack(simRew).mean(0))

for x in rewards:
    plt.plot(x)
```

Figure 1: Q1 Training Rewards

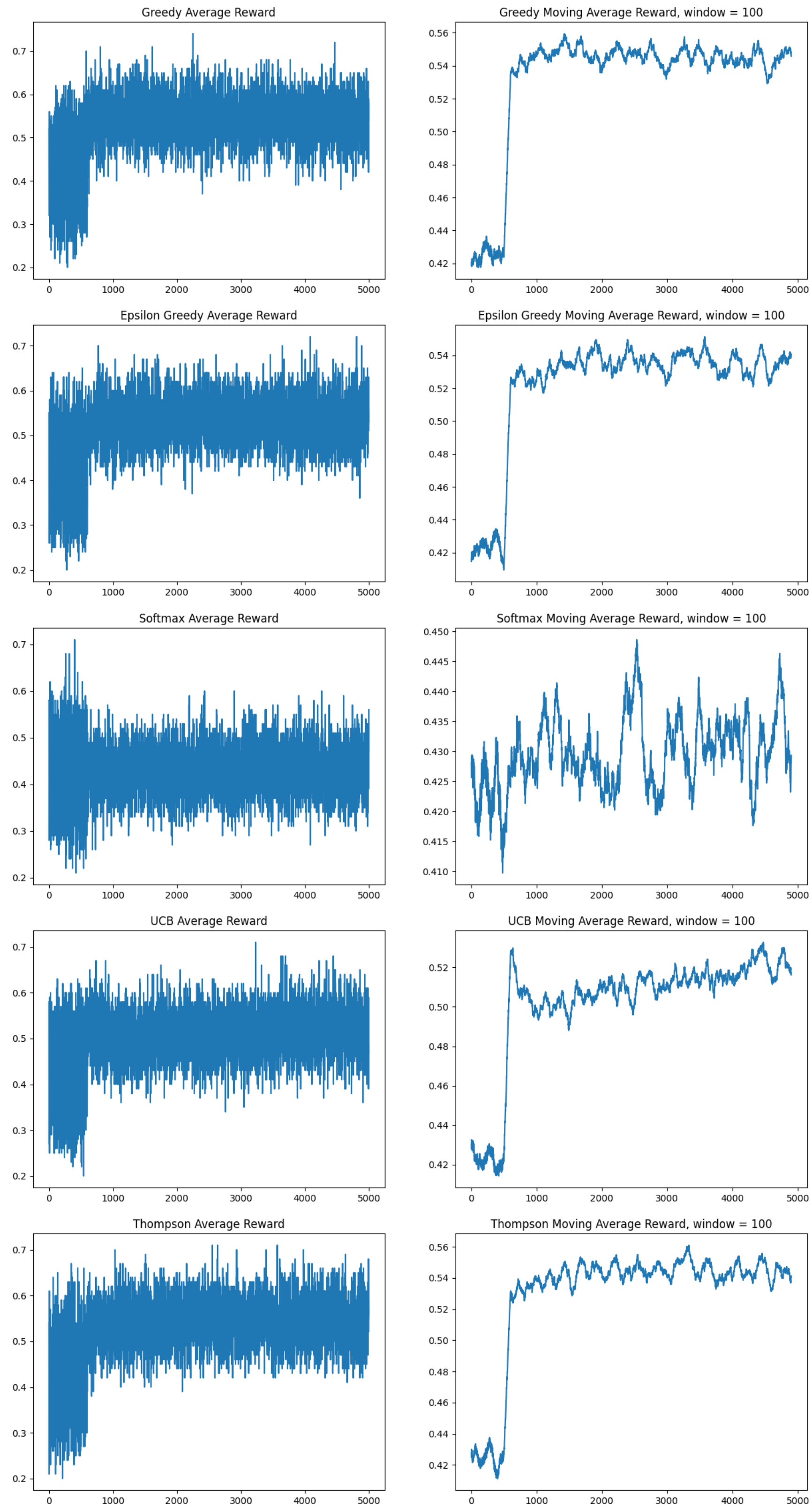

Figure 2: Q2 DQN Training Rewards

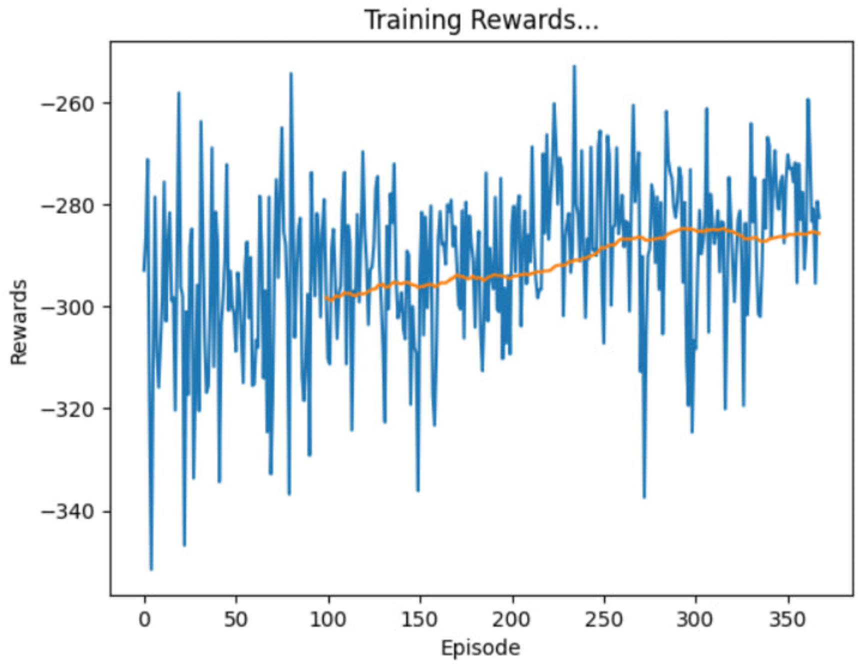

Figure 3: Q2 DQN Test Rewards

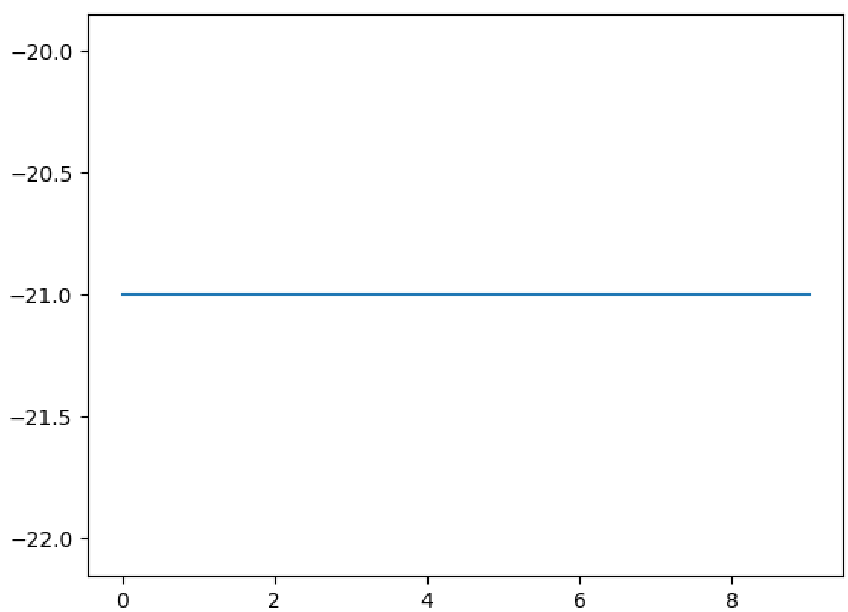

Figure 4: Q2 REINFORCE Train Rewards

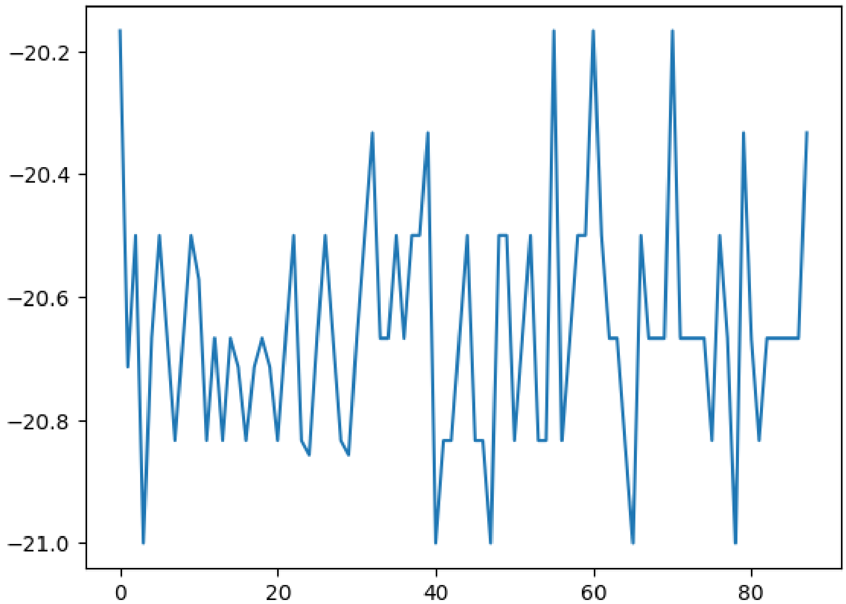

Figure 5: Q2 REINFORCE Train Rewards

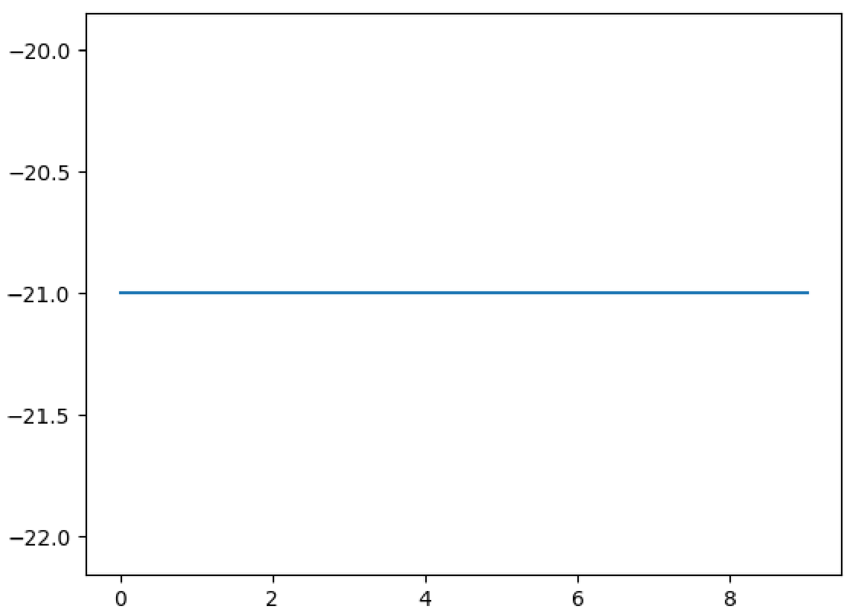

Figure 6: Q4 BC Average Test Rewards (across 5 trajectories)