Tutorial 9

● Graded

Student

Anirud N

Total Points

7.5 / 10 pts

Question 1

1 **5** / 5 pts

✔ **– 0 pts** Correct

– 5 pts Not done

Question 2

2 & 3 **2.5** / 5 pts

✔ **– 0 pts** Correct

✔ **– 2.5 pts** Incorrect

– 5 pts Fully Wrong

No questions assigned to the following page.

cs6700-tutorial-9-dynaq

April 20, 2024

1 Tutorial 9: DynaQ

1.0.1 Tasks to be done:

1. Complete code for Planning step update. (search for "TODO" marker)
2. Compare the performance (train and test returns) for the following values of planning iterations = **[0, 1, 2, 5, 10]**
3. For each value of planning iteration, average the results on **100 runs** (due to the combined stochasticity in the env, epsilon-greedy and planning steps, we need you to average the results over a larger set of runs)

[1]:
```
!pip install gymnasium
```

```
Requirement already satisfied: gymnasium in e:\python\lib\site-packages (0.29.1)
Requirement already satisfied: numpy>=1.21.0 in e:\python\lib\site-packages
(from gymnasium) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in e:\python\lib\site-packages
(from gymnasium) (3.0.0)
Requirement already satisfied: typing-extensions>=4.3.0 in e:\python\lib\site-
packages (from gymnasium) (4.9.0)
Requirement already satisfied: farama-notifications>=0.0.1 in
e:\python\lib\site-packages (from gymnasium) (0.0.4)
```

[2]:
```
import tqdm
import random
import numpy as np
import gymnasium as gym
from matplotlib import pyplot as plt
```

[3]:
```
!pip install gymnasium[toy-text]
```

```
Requirement already satisfied: gymnasium[toy-text] in e:\python\lib\site-
packages (0.29.1)
Requirement already satisfied: numpy>=1.21.0 in e:\python\lib\site-packages
(from gymnasium[toy-text]) (1.26.4)
Requirement already satisfied: cloudpickle>=1.2.0 in e:\python\lib\site-packages
(from gymnasium[toy-text]) (3.0.0)
Requirement already satisfied: typing-extensions>=4.3.0 in e:\python\lib\site-
packages (from gymnasium[toy-text]) (4.9.0)
```

No questions assigned to the following page.

```
Requirement already satisfied: farama-notifications>=0.0.1 in
e:\python\lib\site-packages (from gymnasium[toy-text]) (0.0.4)
Requirement already satisfied: pygame>=2.1.3 in e:\python\lib\site-packages
(from gymnasium[toy-text]) (2.5.2)
```

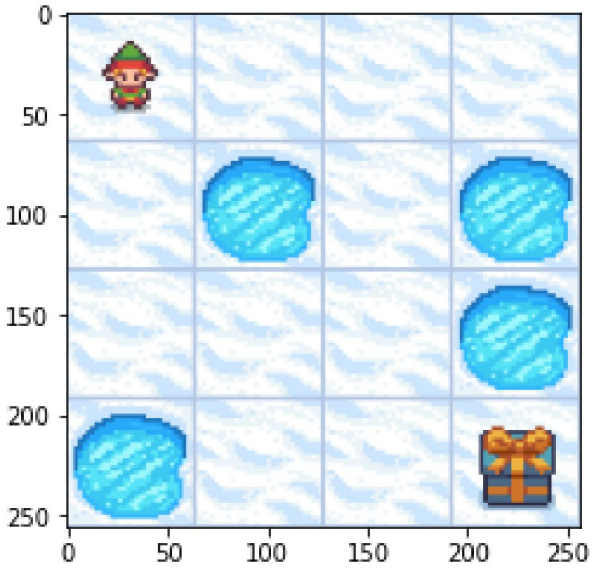
[4]:
```
env = gym.make('FrozenLake-v1', is_slippery = True, render_mode = 'rgb_array')
env.reset()

# https://gymnasium.farama.org/environments/toy_text/frozen_lake

# if pygame is not installed run: "!pip install gymnasium[toy-text]"

plt.imshow(env.render())
```

[4]:
```
<matplotlib.image.AxesImage at 0x230d7e5d8a0>
```



[5]:
```
class DynaQ:
    def __init__(self, num_states, num_actions, gamma=0.99, alpha=0.01,␣
 ↪epsilon=0.25):
        self.num_states = num_states
        self.num_actions = num_actions
        self.gamma = gamma  # discount factor
        self.alpha = alpha  # learning rate
        self.epsilon = epsilon  # exploration rate
        self.q_values = np.zeros((num_states, num_actions))  # Q-values
```

Question assigned to the following page: 1

```
        self.model = {}  # environment model, mapping state-action pairs to␣
↪next state and reward
        self.visited_states = []  # dictionary to track visited state-action␣
↪pairs

    def choose_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.num_actions)
        else:
            return np.argmax(self.q_values[state])

    def update_q_values(self, state, action, reward, next_state):
        # Update Q-value using Q-learning
        best_next_action = np.argmax(self.q_values[next_state])
        td_target = reward + self.gamma * self.
↪q_values[next_state][best_next_action]
        td_error = td_target - self.q_values[state][action]
        self.q_values[state][action] += self.alpha * td_error

    def update_model(self, state, action, reward, next_state):
        # Update model with observed transition
        self.model[(state, action)] = (reward, next_state)

    def planning(self, plan_iters):
        # Perform planning using the learned model
        for _ in range(plan_iters):
            # TODO
            # WRITE CODE HERE FOR TASK 1
            # Update q-value by sampling state-action pairs
            state, action = self.sample_state_action()
            reward, next_state = self.model[(state, action)]
            self.update_q_values(state, action, reward, next_state)

    def sample_state_action(self):
        # Sample a state-action pair from the dictionary of visited␣
↪state-action pairs
        state_action = random.sample(self.visited_states, 1)
        state, action = state_action[0]
        return state, action

    def learn(self, state, action, reward, next_state, plan_iters):
        # Update Q-values, model, and perform planning
        self.update_q_values(state, action, reward, next_state)
        self.update_model(state, action, reward, next_state)

        # Update the visited state-action value
        self.visited_states.append((state, action))
```

No questions assigned to the following page.

```
        self.planning(plan_iters)
```

[6]:
```
class Trainer:
    def __init__(self, env, gamma = 0.99, alpha = 0.01, epsilon = 0.25):
        self.env = env
        self.agent = DynaQ(env.observation_space.n, env.action_space.n, gamma,␣
↪alpha, epsilon)

    def train(self, num_episodes = 1000, plan_iters = 10):
        # training the agent
        all_returns = []
        for episode in range(num_episodes):
            state, _ = self.env.reset()
            done = False
            episodic_return = 0
            while not done:
                action = self.agent.choose_action(state)
                next_state, reward, terminated, truncated, _ = self.env.
↪step(action)
                episodic_return += reward
                self.agent.learn(state, action, reward, next_state, plan_iters)
                state = next_state
                done = terminated or truncated
            all_returns.append(episodic_return)

        return all_returns

    def test(self, num_episodes=500):
        # testing the agent
        all_returns = []
        for episode in range(num_episodes):
            episodic_return = 0
            state, _ = self.env.reset()
            done = False
            while not done:
                action = np.argmax(self.agent.q_values[state]) # Act greedy wrt␣
↪the q-values
                next_state, reward, terminated, truncated, _ = self.env.
↪step(action)
                episodic_return += reward
                state = next_state
                done = terminated or truncated
            all_returns.append(episodic_return)
        return all_returns
```

Question assigned to the following page: 2

[7]:
```
# Example usage:
env = gym.make('FrozenLake-v1', is_slippery = True)
agent = Trainer(env, alpha=0.01, epsilon=0.25)
train_returns = agent.train(num_episodes = 1000, plan_iters = 10)
eval_returns = agent.test(num_episodes = 1000)
print(sum(eval_returns))
```

```
714.0
```

[8]:
```
import time
```

[9]:
```
# WRITE CODE HERE FOR TASKS 2 & 3
plan_iter = [0,1,2,5,10]
list_plan_iter = [[] for i in range(len(plan_iter))]
list_avg_plan_iter = [0 for i in range(len(plan_iter))]
h=0
avg_time = []
dur = []
for pi in plan_iter:
    start = time.time()
    av = 0
    av_l = [0 for i in range(1000)]
    for i in range(100):
        env = gym.make('FrozenLake-v1', is_slippery = True)
        agent = Trainer(env, alpha=0.01, epsilon=0.25)
        train_returns = agent.train(num_episodes = 1000, plan_iters = pi)
        eval_returns = agent.test(num_episodes = 1000)
        end = time.time()
        dur.append(end-start)
        for j in range(len(av_l)):
            av_l[j] += eval_returns[j]
    avg_time.append(np.mean(dur))
    for j in range(len(av_l)):
        av_l[j] = av_l[j] / 100
    av = sum(av_l)
    list_avg_plan_iter[h] = av
    list_plan_iter[h] = av_l
    h+=1
```

[10]:
```
print(av/100)
```

```
2.5218000000000016
```

[11]:
```
from matplotlib import pyplot as plt
```

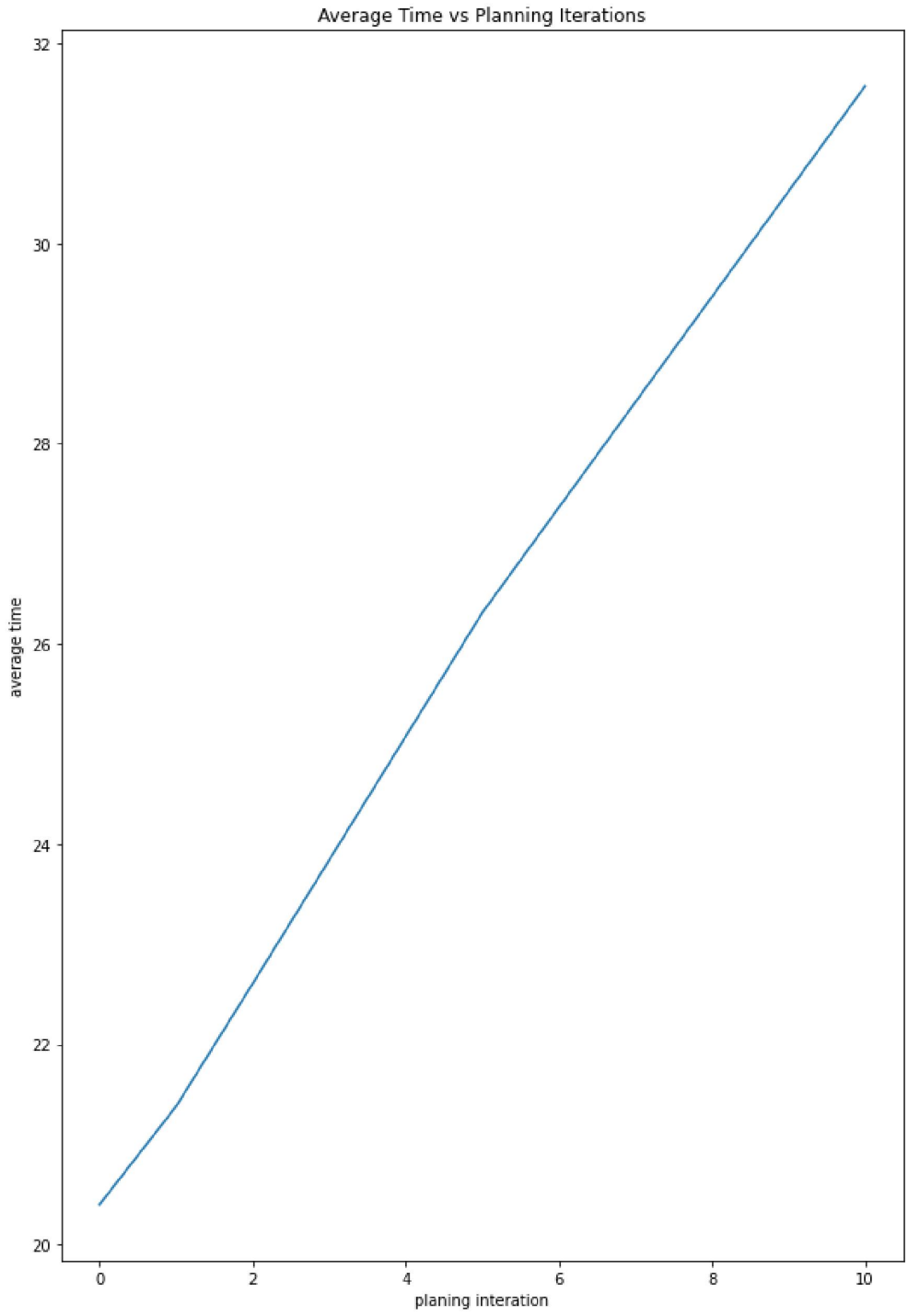
[17]:
```
fig, ax1 = plt.subplots(1, 1, figsize=(10, 15))
ax1.plot(plan_iter, avg_time)
```

Question assigned to the following page: 2

```
ax1.set_xlabel('planing interation')
ax1.set_ylabel('average time')
ax1.set_title('Average Time vs Planning Iterations')
```

[17]: Text(0.5, 1.0, 'Average Time vs Planning Iterations')

Question assigned to the following page: 2

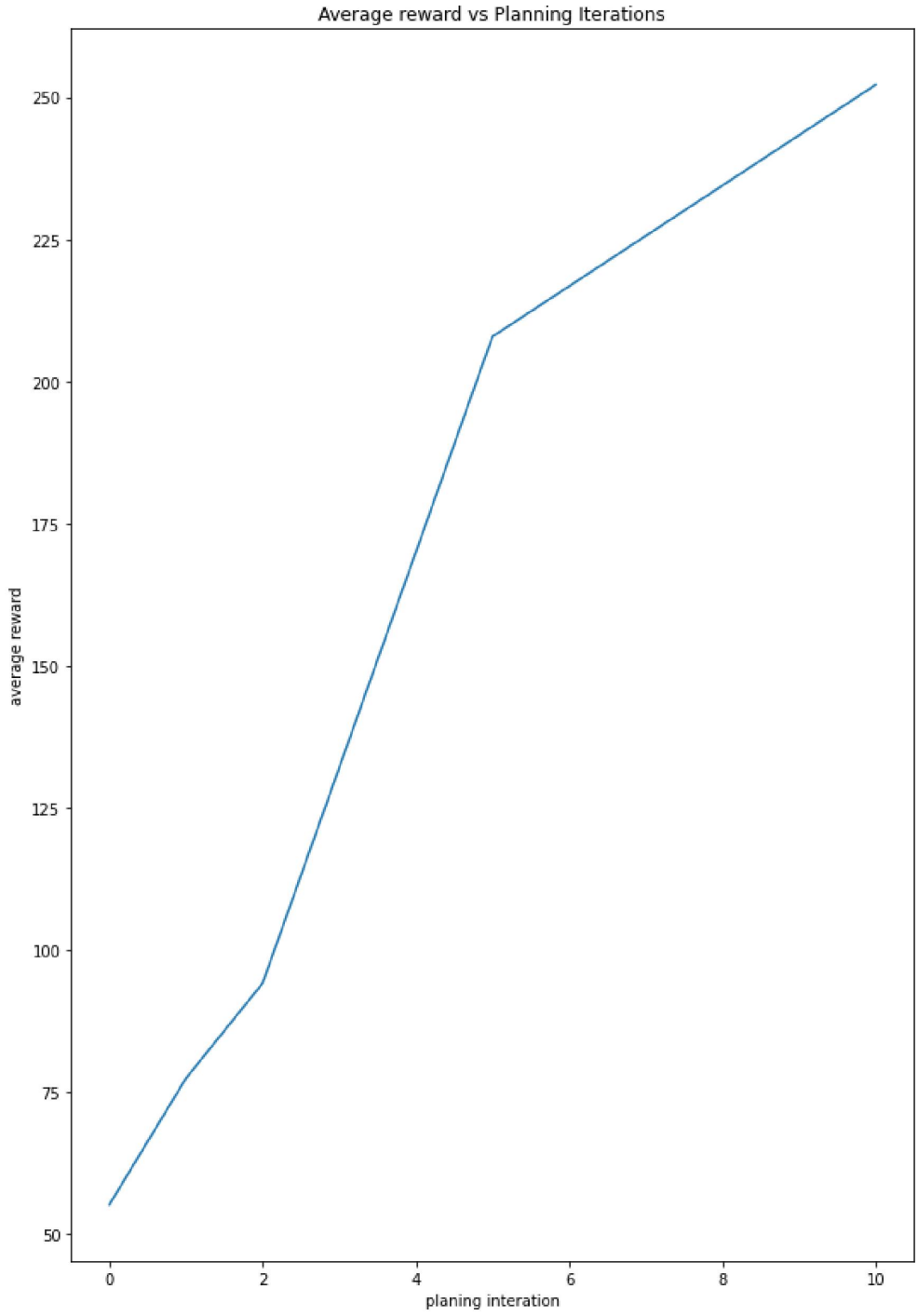

Average Time vs Planning Iterations
average time
planing interation
32
30
28
26
24
22
20
0
2
4
6
8
10

Question assigned to the following page: 2

[16]:
```
fig, ax1 = plt.subplots(1, 1, figsize=(10, 15))
ax1.plot(plan_iter, list_avg_plan_iter)
ax1.set_xlabel('planing interation')
ax1.set_ylabel('average reward')
ax1.set_title('Average reward vs Planning Iterations')
```

[16]:
```
Text(0.5, 1.0, 'Average reward vs Planning Iterations')
```

Question assigned to the following page: 2


Average reward vs Planning Iterations
average reward
planing interation
250
225
200
175
150
125
100
75
50
0
2
4
6
8
10

Question assigned to the following page: 2

Inference: More is the planning iteration, ie more is the planning, more is the time taken. But more is the planning also means, that we are expected to get more average rewards. Also higher the planning, we are expected to get faster convergence than lesser planning

TODO: - Compare the performance (train and test returns) for the following values of planning iterations = [0, 1, 2, 5, 10] - For each value of planning iteration, average the results on 100 runs (due to the combined stochasticity in the env, epsilon-greedy and planning steps, we need you to average the results over a larger set of runs)

Sample Skeleton Code:

```
for pi in plan_iter:
    for 100 times:
        train(pi)
        test()
    print(avg_performance)
```

[ ]: