Tutorial 9: DynaQ

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)

```
!pip install gymnasium
```

```
Requirement already satisfied: gymnasium in c:\python311\lib\site-packages (0.29.1)
Requirement already satisfied: numpy>=1.21.0 in c:\python311\lib\site-packages (from gymnasium) (1.23.5)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\python311\lib\site-packages (from gymnasium) (2.2.0)
Requirement already satisfied: typing-extensions>=4.3.0 in c:\python311\lib\site-packages (from gymnasium) (4.5.0)
Requirement already satisfied: farama-notifications>=0.0.1 in c:\python311\lib\site-packages (from gymnasium) (0.0.4)


[notice] A new release of pip is available: 23.2.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```python
import tqdm
import random
import numpy as np
import gymnasium as gym
from matplotlib import pyplot as plt
```

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

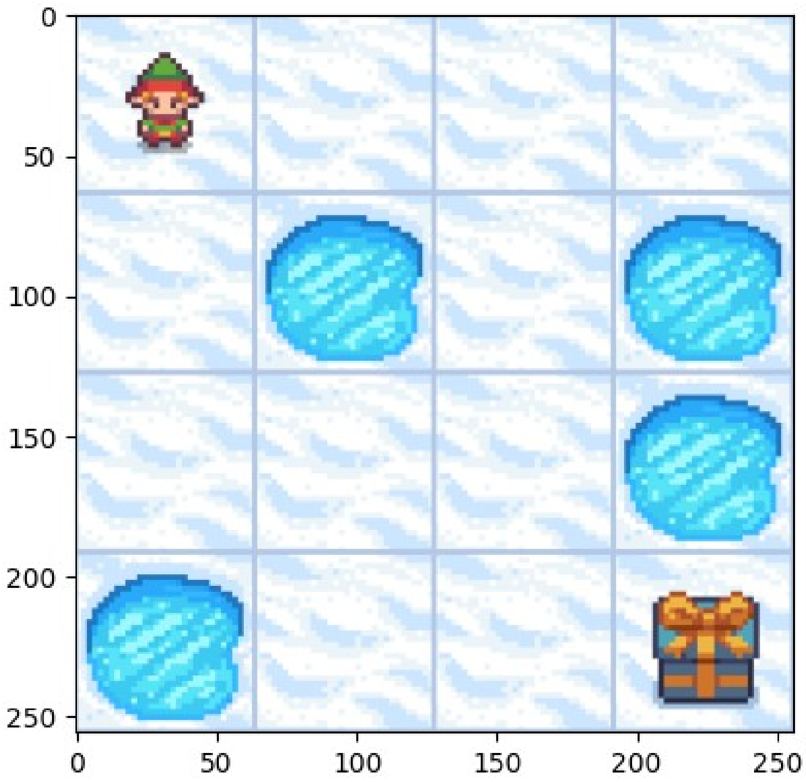
```
Requirement already satisfied: gymnasium[toy-text] in c:\python311\lib\site-packages (0.29.1)
Requirement already satisfied: numpy>=1.21.0 in c:\python311\lib\site-packages (from gymnasium[toy-text]) (1.23.5)
Requirement already satisfied: cloudpickle>=1.2.0 in c:\python311\lib\site-packages (from gymnasium[toy-text]) (2.2.0)
Requirement already satisfied: typing-extensions>=4.3.0 in c:\python311\lib\site-packages (from gymnasium[toy-text]) (4.5.0)
Requirement already satisfied: farama-notifications>=0.0.1 in c:\python311\lib\site-packages (from gymnasium[toy-text]) (0.0.4)
Collecting pygame>=2.1.3 (from gymnasium[toy-text])
  Obtaining dependency information for pygame>=2.1.3 from
https://files.pythonhosted.org/packages/82/61/93ae7afbd931a70510cfdf0a
```

```
7bb0007540020b8d80bc1d8762ebdc46479b/pygame-2.5.2-cp311-cp311-
win_amd64.whl.metadata
  Downloading pygame-2.5.2-cp311-cp311-win_amd64.whl.metadata (13 kB)
Downloading pygame-2.5.2-cp311-cp311-win_amd64.whl (10.8 MB)
   ---------------------------------------- 0.0/10.8 MB ? eta -:--:--
   ---------------------------------------- 0.0/10.8 MB 960.0 kB/s eta
0:00:12
    --------------------------------------- 0.2/10.8 MB 2.4 MB/s eta
0:00:05
   - -------------------------------------- 0.5/10.8 MB 3.1 MB/s eta
0:00:04
   --- ------------------------------------ 1.0/10.8 MB 5.5 MB/s eta
0:00:02
   ------- -------------------------------- 1.9/10.8 MB 8.2 MB/s eta
0:00:02
   -------- ------------------------------- 2.3/10.8 MB 8.5 MB/s eta
0:00:01
   -------- ------------------------------- 2.3/10.8 MB 7.9 MB/s eta
0:00:02
   -------- ------------------------------- 2.4/10.8 MB 6.7 MB/s eta
0:00:02
   --------- ------------------------------ 2.5/10.8 MB 6.2 MB/s eta
0:00:02
   --------- ------------------------------ 2.7/10.8 MB 5.8 MB/s eta
0:00:02
   ----------- ---------------------------- 3.0/10.8 MB 6.1 MB/s eta
0:00:02
   -------------- ------------------------- 3.9/10.8 MB 7.2 MB/s eta
0:00:01
   --------------------- ------------------ 5.7/10.8 MB 9.6 MB/s eta
0:00:01
   -------------------------- ------------- 7.0/10.8 MB 11.2 MB/s eta
0:00:01
   ------------------------------ --------- 8.2/10.8 MB 12.2 MB/s eta
0:00:01
   ------------------------------------ --- 9.7/10.8 MB 13.8 MB/s eta
0:00:01
   ---------------------------------------  10.8/10.8 MB 17.2 MB/s eta
0:00:01
   ---------------------------------------- 10.8/10.8 MB 16.8 MB/s eta
0:00:00
Installing collected packages: pygame
Successfully installed pygame-2.5.2


[notice] A new release of pip is available: 23.2.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
```

```
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())
```

```
<matplotlib.image.AxesImage at 0x15c9ce0d910>
```



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

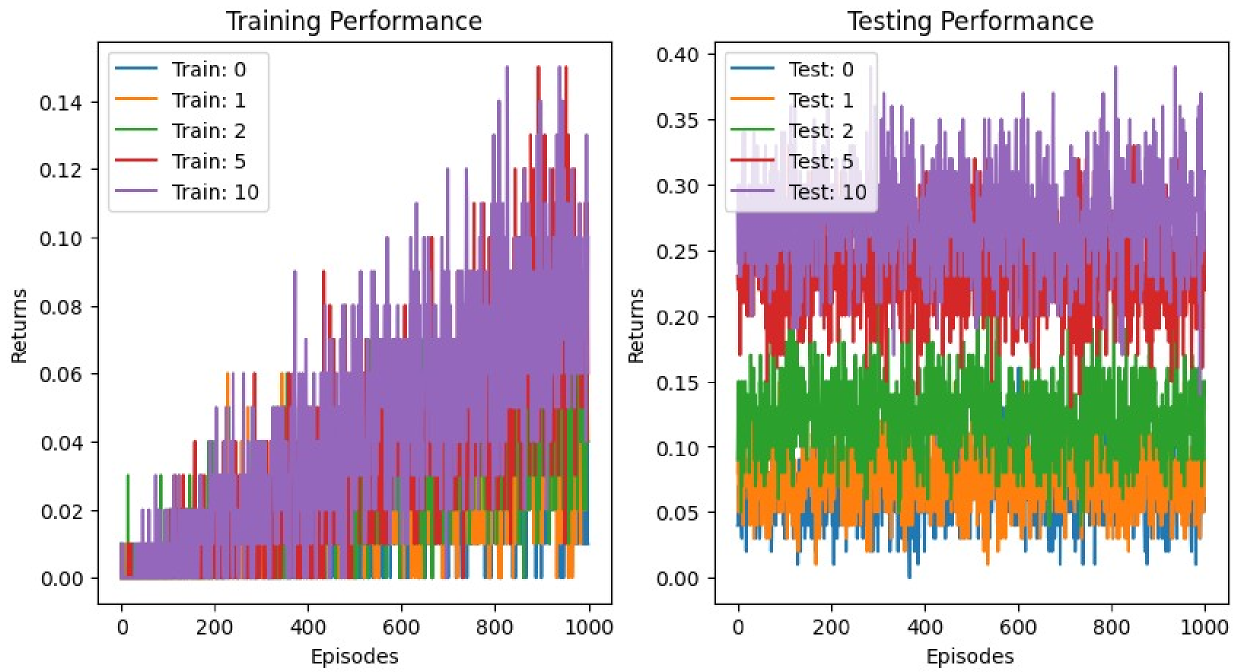
```
    def choose_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.num_actions)
        else:
            return np.argmax(self.q_values[state])

    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))
        self.planning(plan_iters)

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

# 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))
```

```
388.0
```

```python
# WRITE CODE HERE FOR TASKS 2 & 3

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))

for pi in [0, 1, 2, 5, 10]:
    train = []
    test = []
    for _ 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)
        train.append(train_returns)
        eval_returns = agent.test(num_episodes = 1000)
        test.append(eval_returns)
    train_avg = np.mean(np.array(train), axis = 0)
    test_avg = np.mean(np.array(test), axis = 0)
    print(f'Planning Iterations: {pi}, Train: {sum(train_avg)}, Test:
{sum(test_avg)}')
    axes[0].plot(train_avg, label = f'Train: {pi}')
    axes[1].plot(test_avg, label = f'Test: {pi}')

axes[0].set_title('Training Performance')
axes[0].set_xlabel('Episodes')
axes[0].set_ylabel('Returns')

axes[1].set_title('Testing Performance')
axes[1].set_xlabel('Episodes')
axes[1].set_ylabel('Returns')

axes[0].legend()
axes[1].legend()

plt.show()
```

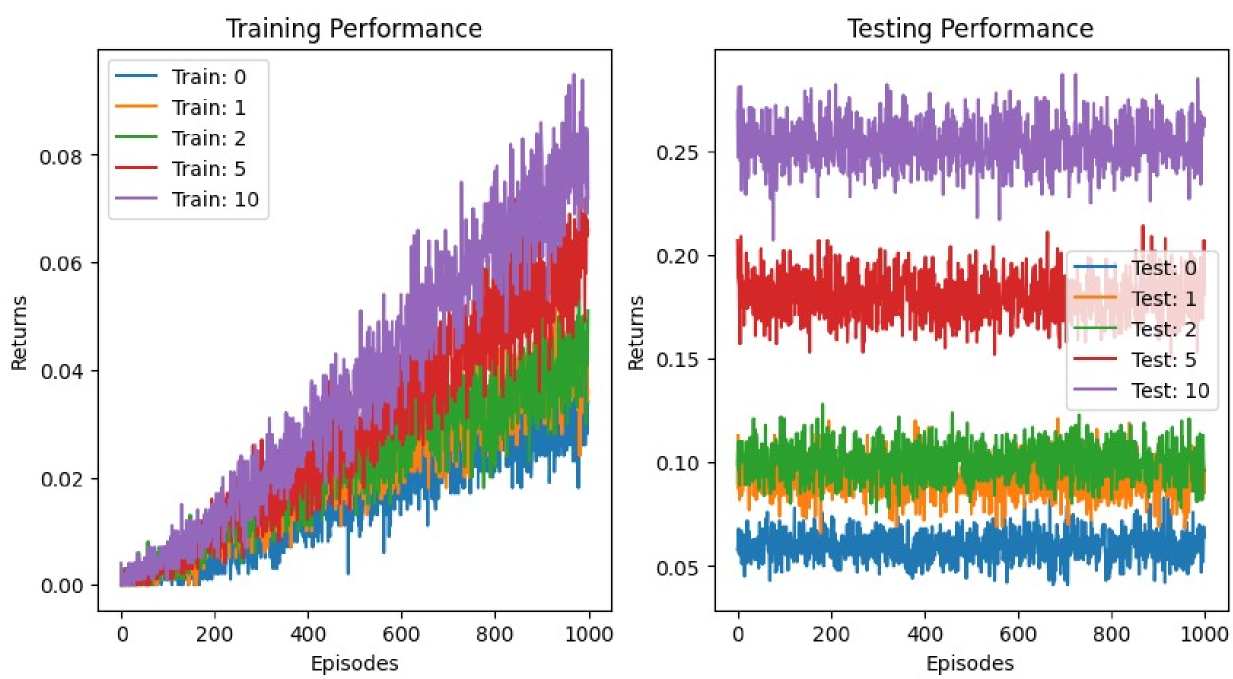
```
Planning Iterations: 0, Train: 16.949999999999893, Test:
66.77000000000002
Planning Iterations: 1, Train: 20.689999999999902, Test:
77.48000000000009
Planning Iterations: 2, Train: 28.219999999999867, Test:
123.34999999999995
Planning Iterations: 5, Train: 35.92999999999994, Test:
238.5599999999997
Planning Iterations: 10, Train: 42.49999999999998, Test:
276.09000000000003
```




```
# Averaging over 1000 runs instead of 100, to get better results

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))

for pi in [0, 1, 2, 5, 10]:
    train = []
    test = []
    for _ in range(1000):
        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)
        train.append(train_returns)
        eval_returns = agent.test(num_episodes = 1000)
        test.append(eval_returns)
    train_avg = np.mean(np.array(train), axis = 0)
    test_avg = np.mean(np.array(test), axis = 0)
    print(f'Planning Iterations: {pi}, Train: {sum(train_avg)}, Test:
{sum(test_avg)}')
    axes[0].plot(train_avg, label = f'Train: {pi}')
    axes[1].plot(test_avg, label = f'Test: {pi}')

axes[0].set_title('Training Performance')
axes[0].set_xlabel('Episodes')
axes[0].set_ylabel('Returns')

axes[1].set_title('Testing Performance')
axes[1].set_xlabel('Episodes')
axes[1].set_ylabel('Returns')
```

```
axes[0].legend()
axes[1].legend()

plt.show()
```

```
Planning Iterations: 0, Train: 15.701999999999993, Test:
59.352999999999895
Planning Iterations: 1, Train: 21.349999999999994, Test:
92.36099999999976
Planning Iterations: 2, Train: 22.471000000000007, Test:
100.5059999999996
Planning Iterations: 5, Train: 29.079999999999963, Test:
179.74800000000042
Planning Iterations: 10, Train: 37.428000000000054, Test:
253.52500000000018
```



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)
```

Inferences

- From averaging over 100 runs and 1000 runs we observe that the performance (episodic return) increases faster while training when planning is employed.
- Further, we also observe that the increasing the number of planning steps improves the agents performance. While testing, we observe a similar nature, as the number of planning steps increases the agent performs better.
- The printed lines of the corresponding cells are the sum of returns while training and testing. The values consistently increase with the increase in the number of planning steps. This is shows the improvement in performance.
- There is a lot of stochasticity in the environment, therefore averaging over 1000 runs made the graphs clearer.