Tutorial 9: DynaQ

Tasks to be done:

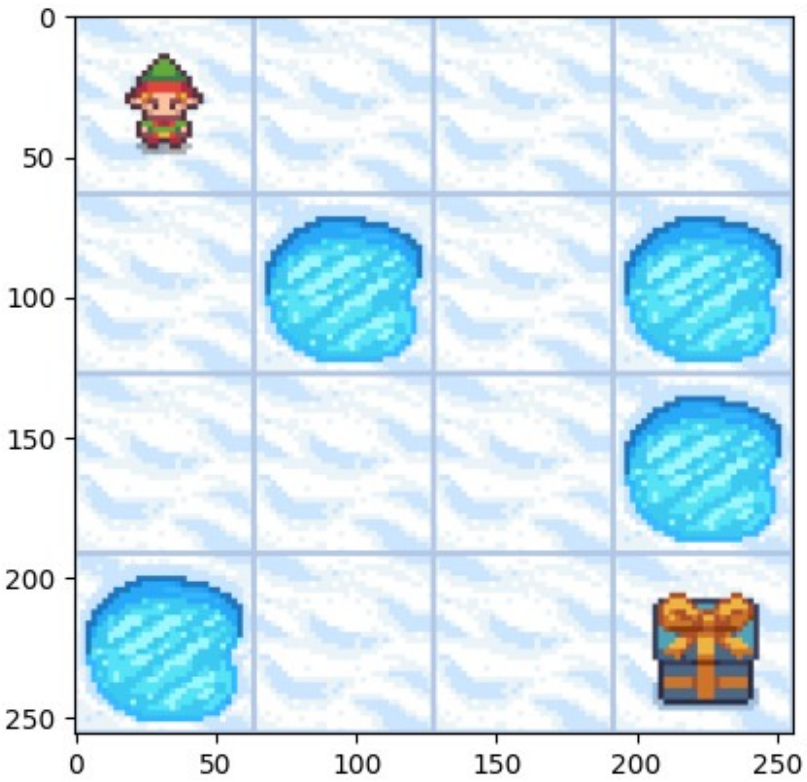
1. Complete code for Planning step update. (search for "TODO" marker)
2. Compare the performance (train and test returns) for the following values of planning iterations = **[0, 1, 2, 5, 10]**
3. For each value of planning iteration, average the results on **100 runs** (due to the combined stochasticity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

```
!pip install gymnasium
```

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

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())
```

```
/home/shuvrajeet/.local/lib/python3.11/site-packages/pygame/
pkgdata.py:25: DeprecationWarning: pkg_resources is deprecated as an
API. See https://setuptools.pypa.io/en/latest/pkg_resources.html
  from pkg_resources import resource_stream, resource_exists
```

```
<matplotlib.image.AxesImage at 0x7efd511baf90>
```




```python
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))
```

```
/home/shuvrajeet/.local/lib/python3.11/site-packages/gym/utils/
passive_env_checker.py:233: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
/home/shuvrajeet/.local/lib/python3.11/site-packages/gym/utils/passive
_env_checker.py:237: DeprecationWarning: `np.bool8` is a deprecated
alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(truncated, (bool, np.bool8)):

0.0
```

```
# WRITE CODE HERE FOR TASKS 2 & 3
env = gym.make('FrozenLake-v1', is_slippery = True)
agent = Trainer(env, alpha=0.01, epsilon=0.25)
```

```
for plan_steps in [0,1,2,5,10]:
    train_returns = np.zeros(100)
    eval_returns = np.zeros(100)
    for i in tqdm.tqdm(range(100)):
        train_returns[i] = np.sum(agent.train(num_episodes = 1000,
plan_iters = plan_steps))
        eval_returns[i] = np.sum(agent.test(num_episodes = 1000))
    print("Plan Steps: ", plan_steps, "Mean Train Return: ",
np.mean(train_returns), "Mean Eval Return: ", np.mean(eval_returns))
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
/home/shuvrajeet/.local/lib/python3.11/site-packages/gym/utils/
passive_env_checker.py:233: DeprecationWarning: `np.bool8` is a
deprecated alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(terminated, (bool, np.bool8)):
/home/shuvrajeet/.local/lib/python3.11/site-packages/gym/utils/passive
_env_checker.py:237: DeprecationWarning: `np.bool8` is a deprecated
alias for `np.bool_`.  (Deprecated NumPy 1.24)
  if not isinstance(truncated, (bool, np.bool8)):
100%|██████████| 100/100 [01:03<00:00,  1.58it/s]
Plan Steps:  0 Mean Train Return:  173.64 Mean Eval Return:  662.77
100%|██████████| 100/100 [01:26<00:00,  1.16it/s]
Plan Steps:  1 Mean Train Return:  185.63 Mean Eval Return:  715.25
100%|██████████| 100/100 [01:37<00:00,  1.02it/s]
Plan Steps:  2 Mean Train Return:  175.89 Mean Eval Return:  684.96
100%|██████████| 100/100 [02:09<00:00,  1.30s/it]
Plan Steps:  5 Mean Train Return:  155.01 Mean Eval Return:  581.22
100%|██████████| 100/100 [03:02<00:00,  1.82s/it]
Plan Steps:  10 Mean Train Return:  127.3 Mean Eval Return:  475.2
```

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)
```

```
# For 5 Planning step the result was best
```