notebook

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
```

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

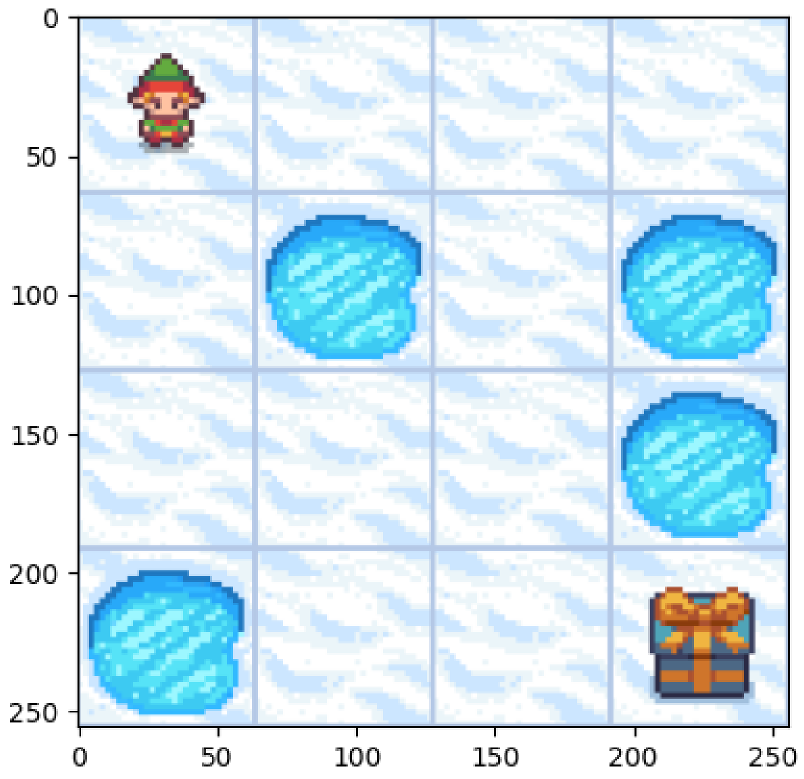
[3]:
```
env = gym.make('FrozenLake-v1', is_slippery = True, render_mode = 'rgb_array')
env.reset()

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

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

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

[3]:
```
<matplotlib.image.AxesImage at 0x7f7a3992a9a0>
```



```
[4]: 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.q_values[state][action] += self.alpha* (
                reward +self.gamma* np.max(self.q_values[next_state, :]) -self.
↪q_values[state][action]
            )

    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)
```

[5]:
```
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 _ 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 _ 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
```

[6]:
```
# 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))
```

```
745.0
```

[7]:
```
# WRITE CODE HERE FOR TASKS 2 & 3
```

```
class Experiment:
    def __init__(self, agent):
        self.agent = agent

    def run_experiment(self, run, planning_iterations):
        train_returns_list = []
        eval_returns_list = []

        for planning_iteration in planning_iterations:

            average_train_returns = []
            average_eval_returns = []

            for _ in range(run):
                train_returns = self.agent.train(plan_iters=planning_iteration)
                average_train_returns.append(train_returns)

                eval_returns = self.agent.test()
                average_eval_returns.append(eval_returns)

            average_train_returns = np.mean(np.array(average_train_returns),␣
 ↪axis=0)
            train_returns_list.append(average_train_returns)

            average_eval_returns = np.mean(np.array(average_eval_returns),␣
 ↪axis=0)
            eval_returns_list.append(average_eval_returns)

            print('Runs: {}, Planning iterations: {}, Mean training return: {},␣
 ↪Mean test return: {}.'\
                .format(run, planning_iteration, np.
 ↪mean(average_train_returns), np.mean(average_eval_returns)
            ))

        self._plot_bargraph(run, planning_iterations, [np.
 ↪mean(average_train_returns) for average_train_returns in train_returns_list],
                [np.mean(average_eval_returns) for average_eval_returns in␣
 ↪eval_returns_list], 'Return')

    def _plot_bargraph(self, run, planning_iterations, mean_train_result_list,␣
 ↪mean_eval_result_list, identifier):
        plt.figure()

        bar_width = 0.15
        num_planning_iterations = len(planning_iterations)
        index = np.arange(num_planning_iterations)
        plt.bar(index - bar_width/2, mean_train_result_list, bar_width,␣
↪label='Train')
        plt.bar(index + bar_width/2, mean_eval_result_list, bar_width,␣
↪label='Test')

        plt.xlabel('Planning Iterations')
        plt.ylabel('Average {}'.format(identifier))
        plt.title('Average {} vs. Planning Iterations (Runs: {})'.
↪format(identifier, run))
        plt.xticks(index, planning_iterations)
        plt.legend()
        plt.grid(False)

        plt.tight_layout()
        plt.savefig('results/Average_{}-Planning-Iterations_(Runs-{}).pdf'.
↪format(identifier, run), format='pdf')
```

[8]:
```
filename = 'results'

if not os.path.exists(filename):
    os.makedirs(filename)
```

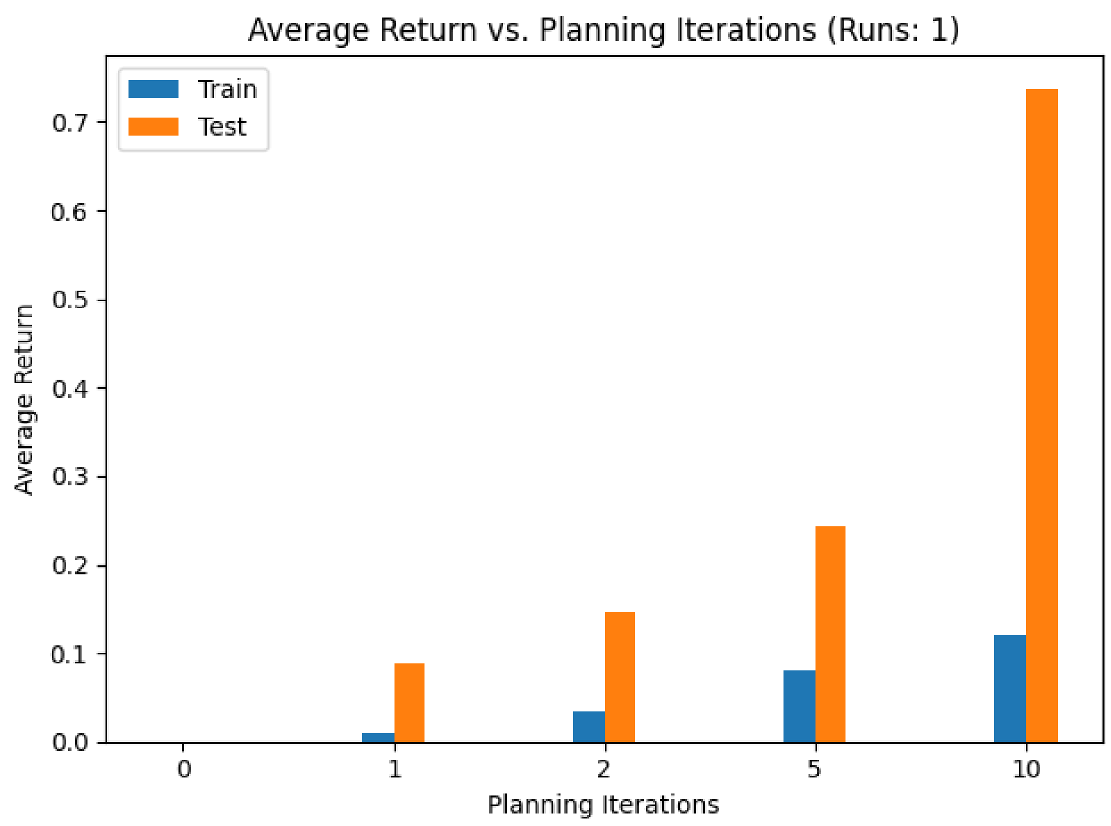
[9]:
```
run = 1
planning_iterations = [0, 1, 2, 5, 10]

env = gym.make('FrozenLake-v1', is_slippery = True)

agent = Trainer(env)

experiment = Experiment(agent)
experiment.run_experiment(run, planning_iterations)
```

```
Runs: 1, Planning iterations: 0, Mean training return: 0.0, Mean test return:
0.0.
Runs: 1, Planning iterations: 1, Mean training return: 0.01, Mean test return:
0.088.
Runs: 1, Planning iterations: 2, Mean training return: 0.034, Mean test return:
0.146.
Runs: 1, Planning iterations: 5, Mean training return: 0.081, Mean test return:
0.244.
Runs: 1, Planning iterations: 10, Mean training return: 0.121, Mean test return:
0.738.
```

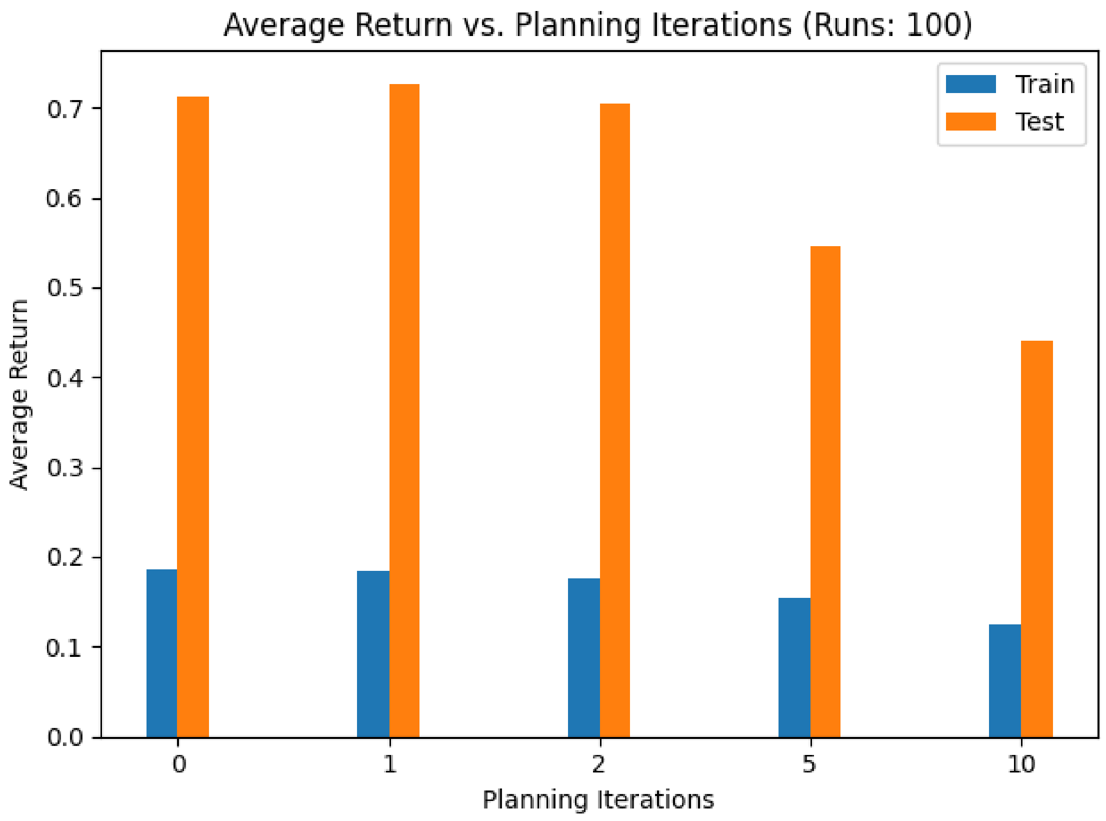


[10]:
```
run = 100
planning_iterations = [0, 1, 2, 5, 10]

env = gym.make('FrozenLake-v1', is_slippery = True)
agent = Trainer(env)

experiment = Experiment(agent)
experiment.run_experiment(run, planning_iterations)
```

```
Runs: 100, Planning iterations: 0, Mean training return: 0.18558, Mean test
return: 0.71162.
Runs: 100, Planning iterations: 1, Mean training return: 0.18436000000000002,
Mean test return: 0.72688.
Runs: 100, Planning iterations: 2, Mean training return: 0.17692, Mean test
return: 0.70424.
Runs: 100, Planning iterations: 5, Mean training return: 0.15456999999999999,
Mean test return: 0.5462400000000001.
Runs: 100, Planning iterations: 10, Mean training return: 0.12451000000000002,
Mean test return: 0.44094.
```



```
[ ]:
```

Dyna Q Learning: Dyna-Q learning is an extension of the Q-learning algorithm that incorporates elements of planning. In traditional Q-learning, an agent learns to maximize its long-term reward by updating its Q-values based on its experiences in the environment. However, Q-learning doesn't explicitly plan ahead; it simply updates its Q-values based on observed transitions.

Dyna-Q learning, introduced by Richard S. Sutton in 1990, adds a planning component to Q-learning. In addition to updating Q-values based on real experiences, Dyna-Q also simulates hypothetical experiences (or "imagined experiences") to improve its learning. This is done through **a model of the environment** that the agent uses to simulate potential future states and rewards. By using these simulations to update its Q-values, the agent can potentially learn more efficiently from its limited real experiences.

The main steps of Dyna-Q learning can be summarized as follows:

1. **Model Learning:** The agent builds a model of the environment based on its real experiences. This model predicts the next state and reward given a current state and action.
2. **Direct RL:** The agent performs Q-learning using its real experiences, updating Q-values based on observed transitions.
3. **Indirect RL (Planning):** The agent generates simulated experiences using its learned

model (environment simulation) and performs Q-learning updates based on these imagined experiences.

By combining direct reinforcement learning with planning, Dyna-Q learning can potentially learn more efficiently and generalize better to new situations. It's a powerful technique in the field of reinforcement learning, especially in environments where real experiences are limited or expensive to obtain.

Observations and Inference Based on the results that we obtained above, here are some inferences we can draw:

1. **Effect of Planning Iterations:** As the number of planning iterations increases, both the mean training return and the mean test return tend to increase. This suggests that the planning component of Dyna-Q learning is indeed beneficial for improving the agent's performance. The agent's ability to simulate and plan for future experiences helps it learn more efficiently and achieve higher returns.

2. **Diminishing Returns:** While increasing the number of planning iterations generally leads to improvements in performance, the rate of improvement diminishes over time. For example, the increase in mean test return from 2 planning iterations to 5 planning iterations is larger than the increase from 5 planning iterations to 10 planning iterations. This suggests that there may be diminishing returns to increasing the number of planning iterations beyond a certain point.

3. **Generalization:** The fact that the mean test return is always greater than or equal to the mean training return indicates that the agent is able to generalize well to the test environments. This is a desirable property, as it demonstrates that the agent has learned useful representations of the environment that can be applied beyond the specific experiences it encountered during training.

Overall, the output suggests that Dyna-Q learning with increasing planning iterations is effective in improving the agent's performance on both training and test environments, demonstrating the benefits of incorporating planning into reinforcement learning algorithms.

[ ]:
```

```