EXPERIMENT ASSESSMENT

ACADEMIC YEAR 2023-24

Course: Reinforcement Learning Lab

Course code: CSDOL8013

Year: BE **Sem:** VIII

Experiment No.: 5
Aim: - Implementing Monte Carlo control and Temporal Difference (TD) learning algorithms to train an agent in a grid-world environment
Name:
Roll Number:
Date of Performance:
Date of Submission:

Evaluation

Performance Indicator	Max. Marks	Marks Obtained
Performance	5	
Understanding	5	
Journal work and timely submission.	10	
Total	**20**	

Performance Indicator	Exceed Expectations (EE)	Meet Expectations (ME)	Below Expectations (BE)
Performance	5	3	2
Understanding	5	3	2
Journal work and timely submission.	10	8	4

Checked by

Name of Faculty : Rujuta Vartak

Signature :

Date :

Experiment - 05

Aim: Implementing Monte Carlo control and Temporal Difference (TD) learning algorithms to train an agent in a grid-world environment.

Objective: The objective is to implement Monte Carlo control and Temporal Difference (TD) learning algorithms to train an agent within a grid-world environment, aiming to understand and compare the learning dynamics and performance of these two reinforcement learning methods while navigating a simplified spatial domain.

Theory:
Monte Carlo Control
Monte Carlo methods are a class of algorithms used in reinforcement learning for estimating value functions and finding optimal policies. Unlike dynamic programming methods, Monte Carlo methods do not require a model of the environment and instead learn directly from experience by interacting with the environment.
Monte Carlo control combines Monte Carlo prediction with policy improvement to find the optimal policy directly from experience. After estimating the value function, the policy is improved by selecting actions that maximize the value function estimates.
Monte Carlo methods learn from complete episodes of experience. Here's the algorithm for Monte Carlo Control:
Input:

- MDP: (S, A, P, R, γ), where
    - S is the set of states.
    - A is the set of actions.
    - P is the transition probability matrix.
    - R is the reward function.
    - γ is the discount factor.
- Number of episodes **num_episodes**.
- Exploration rate **epsilon**.

Algorithm:

- Optimal policy π.

1. **Initialize**:
    - Q-function: Q(s, a) for all s in S and a in A.
    - State-action visit count: N(s, a) for all s in S and a in A.

```
        ● Policy π arbitrarily.

2. For each episode from 1 to num_episodes:
        ● Generate an episode following π: (s1, a1, r1), (s2, a2, r2), ..., (sT, aT, rT).
        ● Set G = 0.
        ● For each time step t from T down to 1**:
                ● Update return G: G = γ * G + rt+1.
                ● If (st, at) has not been visited in this episode before:
                        ● Increment visit count: N(st, at) += 1.
                        ● Update action-value function: Q(st, at) = Q(st, at) + (1/N(st, at)) * (G - Q(st, at)).
                        ● Update policy: π(st) = argmax(Q(st, a)) for all a in A.
3. Return the optimal policy π.
```

Temporal Difference (TD) Learning

Temporal Difference learning methods update estimates based on the difference between current and future estimates. Here's the algorithm for TD Learning:

Input:

- MDP: (S, A, P, R, γ), where
  - S is the set of states.
  - A is the set of actions.
  - P is the transition probability matrix.
  - R is the reward function.
  - γ is the discount factor.
- Number of episodes **num_episodes**.
- Step size **alpha**.
- Exploration rate **epsilon**.

Algorithm:

```
● Optimal policy π.
1. Initialize:
        ● Q-function: Q(s, a) for all s in S and a in A.
        ● Policy π arbitrarily.
2. For each episode from 1 to num_episodes:
        ● Initialize state s as the starting state.
        ● While s is not a terminal state:
                ● Choose action a using policy derived from Q (e.g., ε-greedy).
                ● Take action a, observe reward r and next state s'.
```

- Update action-value function: Q(s, a) = Q(s, a) + α * (r + γ * max(Q(s', a')) - Q(s, a)).
- Update policy: π(s) = argmax(Q(s, a)) for all a in A.
- Set s = s'.

3. **Return** the optimal policy π.

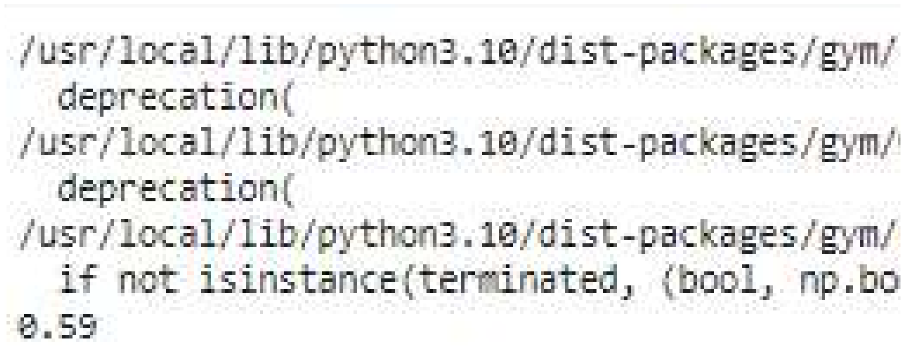
These are high-level algorithms. The specific implementation details and additional considerations (like exploration strategy, eligibility traces, etc.) might vary based on the problem and requirements.

Code :

```
import gym
import numpy as np
import operator
from IPython.display import clear_output
from time import sleep
import random
import itertools
import tqdm


tqdm.monitor_interval = 0


def create_random_policy(env):
    policy = {}
    for key in range(0, env.observation_space.n):
        current_end = 0
        p = {}
        for action in range(0, env.action_space.n):
            p[action] = 1 / env.action_space.n
        policy[key] = p
    return policy
def create_state_action_dictionary(env, policy):
   Q = {}
   for key in policy.keys():
       Q[key] = {a: 0.0 for a in range(0, env.action_space.n)}
   return Q
import random
from time import sleep
from IPython.display import clear_output


def run_game(env, policy, display=True):
    env.reset()
    episode = []
    finished = False


    while not finished:
        s = env.env.s
        if display:
            clear_output(True)
            env.render()
            sleep(1)


        timestep = []
        timestep.append(s)
        n = random.uniform(0, sum(policy[s].values()))
        top_range = 0
        for prob in policy[s].items():
            top_range += prob[1]
            if n < top_range:
                action = prob[0]
                break
        state, reward, finished, info = env.step(action)
        timestep.append(action)
        timestep.append(reward)


        episode.append(timestep)


    if display:
        clear_output(True)
        env.render()
        sleep(1)
    return episode


def test_policy(policy, env):
      wins = 0
      r = 100
      for i in range(r):
            w = run_game(env, policy, display=False)[-1][-1]
            if w == 1:
                  wins += 1
      return wins / r
def monte_carlo_e_soft(env, episodes=100, policy=None, epsilon=0.01):
    if not policy:
        policy = create_random_policy(env)  # Create an empty dictionary to store state action values
       Q = create_state_action_dictionary(env, policy) # Empty dictionary for storing rewards for each
state-action pair
    returns = {} # 3.


    for _ in range(episodes): # Looping through episodes
        G = 0 # Store cumulative reward in G (initialized at 0)
               episode = run_game(env=env, policy=policy, display=False) # Store state, action and value
respectively


        # for loop through reversed indices of episode array.
        # The logic behind it being reversed is that the eventual reward would be at the end.
        # So we have to go back from the last timestep to the first one propagating result from the future.


        for i in reversed(range(0, len(episode))):
            s_t, a_t, r_t = episode[i]
            state_action = (s_t, a_t)
            G += r_t # Increment total reward by reward on current timestep


            if not state_action in [(x[0], x[1]) for x in episode[0:i]]: #
                if returns.get(state_action):
```

```
                returns[state_action].append(G)
            else:
                returns[state_action] = [G]


                Q[s_t][a_t] = sum(returns[state_action]) / len(returns[state_action]) # Average reward across
episodes


            Q_list = list(map(lambda x: x[1], Q[s_t].items())) # Finding the action with maximum value
            indices = [i for i, x in enumerate(Q_list) if x == max(Q_list)]
            max_Q = random.choice(indices)


            A_star = max_Q # 14.


            for a in policy[s_t].items(): # Update action probability for s_t in policy
                if a[0] == A_star:
                    policy[s_t][a[0]] = 1 - epsilon + (epsilon / abs(sum(policy[s_t].values())))
                else:
                    policy[s_t][a[0]] = (epsilon / abs(sum(policy[s_t].values())))

    return policy
env = gym.make('FrozenLake8x8-v1')
policy = monte_carlo_e_soft(env, episodes= 50000)
test_policy(policy, env)
```

Output :



```
/usr/local/lib/python3.10/dist-packages/gym/
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/
  if not isinstance(terminated, (bool, np.bo
0.59
```

Conclusion:

1. **How do Monte Carlo control and Temporal Difference (TD) learning algorithms differ in training an agent within a grid-world environment?**

In training an agent within a grid-world environment, Monte Carlo (MC) control and Temporal Difference (TD) learning algorithms differ fundamentally in their approach. MC control learns by experiencing complete episodes, updating its value function based on the total return observed after each episode concludes. In contrast, TD learning updates the value function incrementally after each time step, utilizing a bootstrapping approach with observed rewards and estimated future values.

2. **Compare and contrast the key characteristics, advantages, and limitations of each approach**.

Comparing the two approaches, MC control has the advantage of converging to the optimal policy without requiring knowledge of the environment dynamics. However, it suffers from high variance, making it inefficient for non-episodic tasks and computationally expensive, particularly in large state spaces or with lengthy episodes. On the other hand, TD learning offers lower variance and is suitable for both episodic and non-episodic tasks. It is also more computationally efficient and can update its estimates online as new data becomes available. However, TD learning is sensitive to the choice of hyperparameters, may suffer from bias, and could converge to suboptimal solutions depending on the exploration-exploitation strategy employed. Thus, the choice between MC control and TD learning depends on factors such as the task characteristics, available computational resources, and desired trade-offs between convergence speed, computational efficiency, and robustness.