Department of Computer Science and Engineering (Data Science)

Subject: Reinforcement

Learning AY: 2023 - 24

Experiment 7

Aim: To solve the Blackjack Game using Monte Carlo methods.

Theory:

Monte Carlo method is used for estimating value functions and discovering optimal policies. Unlike the other methods, we do not assume complete knowledge of the environment. Monte Carlo methods require only experience—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment.

The term "Monte Carlo" is often used more broadly for any estimation method whose operation involves a significant random component. Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. To ensure that well-defined returns, Monte Carlo methods are defined only for episodic tasks. That is, we assume experience is divided into episodes, and that all episodes eventually terminate no matter what actions are selected. Only on the completion of an episode are value estimates and policies changed. Monte Carlo methods are thus incremental in an episode-by- episode sense.

Blackjack:

The object of the popular casino card game Blackjack is to obtain cards the sum of whose numerical values is as great as possible without exceeding 21. All face cards count as 10, and an ace can count either as 1 or as 11.

We consider the version in which each player competes independently against the dealer. The game begins with two cards dealt to both dealer and player. One of the dealer's cards is face up and the other is face down. If the player has 21 immediately (an ace and a 10-card), it is called a natural. He then wins unless the dealer also has a natural, in which case the game is a draw. If the player does not have a natural, then he can request additional cards, one by one (hits), until he either stops (sticks) or exceeds 21 (goes bust). If he goes bust, he loses; if he sticks, then it becomes the dealer's turn.

The dealer hits or sticks according to a fixed strategy without choice: he sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise, the outcome—win, lose, or draw—is determined by whose final sum is closer to 21.
Points to remember for solving the game using Monte Carlo:

- Each game of blackjack is an episode.
- Rewards of +1, −1, and 0 are given for winning, losing, and drawing, respectively.
- All rewards within a game are zero, and we do not discount ($\gamma = 1$).

- The player's actions are to hit or to stick.
- The states depend on the player's cards and the dealer's showing card.
- We assume that cards are dealt with from an infinite deck (i.e., with replacement).
- If the player holds an ace that he could count as 11 without going bust, then the ace is said to be usable. In this case it is always counted as 11 because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made because, obviously, the player should always hit.
- Thus, the player makes decisions based on three variables: his current sum (12–21), the dealer's one showing card (ace–10), and whether he holds a usable ace. This makes up 200 states.

Algorithm:

Initialize:
- $\pi \leftarrow$ policy to be evaluated
- $V \leftarrow$ an arbitrary state-value function
- $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
- Generate an episode using π
- For each state s appearing in the episode:
  - $G \leftarrow$ return following the first occurrence of s
  - Append G to $Returns(s)$
  - $V(s) \leftarrow \text{average}(Returns(s))$

Lab Assignment to do:

1. Develop a Blackjack Environment and its variables.
2. Implement the Monte Carlo methods for 10000 and 500000 episodes for usable and no usable ace cards.
3. Show the results graphically (3D graphs) for all four combinations. (10000 episodes with and without usable ace and 500000 episodes with and without usable ace)

```
import numpy as np

class BlackjackEnv:
    def __init__(self, num_decks=1):
```

```
        self.num_decks = num_decks
        self.reset()

    def reset(self):
        self.deck = self.num_decks * 4 * list(range(1, 14))
        np.random.shuffle(self.deck)
        self.player_hand = []
        self.dealer_hand = []
        self.player_sum = 0
        self.dealer_sum = 0
        self.usable_ace = False

        for _ in range(2):
            self.player_hand.append(self.draw_card())
            self.dealer_hand.append(self.draw_card())

        self.player_sum = self.calculate_hand_value(self.player_hand)
        self.dealer_sum = self.calculate_hand_value(self.dealer_hand)

        if 1 in self.player_hand and self.player_sum + 10 <= 21:
            self.player_sum += 10
            self.usable_ace = True
        return self.get_state()

    def draw_card(self):
        return self.deck.pop()

    def calculate_hand_value(self, hand):
        hand_value = sum(hand)
        num_aces = hand.count(1)
        while hand_value <= 11 and num_aces > 0:
            hand_value += 10
            num_aces -= 1
        return hand_value

    def step(self, action):
        if action == 'hit':
            self.player_hand.append(self.draw_card())
            self.player_sum = self.calculate_hand_value(self.player_hand)
            if 1 in self.player_hand and self.player_sum + 10 <= 21:
                self.player_sum += 10
```

```
                self.usable_ace = True
            if self.player_sum > 21:
                reward = -1
                done = True
            else:
                reward = 0
                done = False
        elif action == 'stick':
            done = True
            while self.dealer_sum < 17:
                self.dealer_hand.append(self.draw_card())
                self.dealer_sum =
self.calculate_hand_value(self.dealer_hand)
            if self.dealer_sum > 21 or self.dealer_sum < self.player_sum:
                reward = 1
            elif self.dealer_sum == self.player_sum:
                reward = 0
            else:
                reward = -1
        else:
            raise ValueError("Invalid action! Choose 'hit' or 'stick'.")

        return self.get_state(), reward, done

    def get_state(self):
        return self.player_sum, self.dealer_hand[0], self.usable_ace


def monte_carlo_evaluation(env, policy, num_episodes):
    state_action_values = {}
    state_action_returns = {}
    state_action_counts = {}

    for episode in range(num_episodes):
        episode_states = []
        episode_actions = []
        episode_rewards = []

        state = env.reset()
        done = False
```

```
        while not done:
            action = policy(state)
            next_state, reward, done = env.step(action)
            episode_states.append(state)
            episode_actions.append(action)
            episode_rewards.append(reward)
            state = next_state

        G = 0
        for t in reversed(range(len(episode_states))):
            state = episode_states[t]
            action = episode_actions[t]
            reward = episode_rewards[t]
            G += reward
            state_action_pair = (state, action)
            if state_action_pair not in episode_states[:t]:
                if state_action_pair in state_action_returns:
                    state_action_returns[state_action_pair] += G
                    state_action_counts[state_action_pair] += 1
                else:
                    state_action_returns[state_action_pair] = G
                    state_action_counts[state_action_pair] = 1
                state_action_values[state_action_pair] =
state_action_returns[state_action_pair] /
state_action_counts[state_action_pair]
    return state_action_values


def random_policy(state):
    return 'hit' if np.random.random() < 0.5 else 'stick'


env = BlackjackEnv()

num_episodes = 10000
state_action_values_10000 = monte_carlo_evaluation(env, random_policy,
num_episodes)

num_episodes = 500000
state_action_values_500000 = monte_carlo_evaluation(env, random_policy,
num_episodes)
```

Department of Computer Science and Engineering (Data Science)

```
def print_state_action_values(state_action_values):
    for state_action, value in state_action_values.items():
        state, action = state_action
        print(f"State: {state}, Action: {action}, Value: {value}")


print("State-action values for 10000 episodes with usable ace:")
print_state_action_values(state_action_values_10000)
```

```
State: (15, 6, False), Action: stick, Value: -0.44
State: (20, 13, False), Action: stick, Value: 0.87
State: (13, 7, False), Action: stick, Value: -0.5
State: (20, 10, False), Action: stick, Value: 0.525
State: (16, 4, False), Action: stick, Value: -0.15
State: (6, 11, False), Action: stick, Value: -0.07692307692307693
State: (13, 10, False), Action: stick, Value: -0.25
State: (18, 12, False), Action: hit, Value: -0.6444444444444445
State: (17, 12, False), Action: hit, Value: -0.75
State: (12, 13, False), Action: stick, Value: 0.1111111111111111
State: (18, 6, False), Action: hit, Value: -0.6578947368421053
State: (14, 6, False), Action: hit, Value: -0.48936170212765956
State: (20, 6, False), Action: hit, Value: -0.7560975609756098
State: (25, 2, False), Action: stick, Value: 1.0
State: (24, 13, False), Action: stick, Value: 1.0
State: (13, 12, False), Action: stick, Value: 0.15151515151515152
State: (17, 11, False), Action: stick, Value: -0.2708333333333333
State: (18, 9, False), Action: stick, Value: 0.16279069767441862
State: (21, 4, False), Action: stick, Value: 0.85
State: (18, 4, False), Action: hit, Value: -0.8444444444444444
State: (17, 7, False), Action: hit, Value: -0.7804878048780488
State: (20, 11, False), Action: hit, Value: -0.8333333333333334
State: (19, 11, False), Action: hit, Value: -0.75
State: (17, 11, False), Action: hit, Value: -0.8378378378378378
State: (21, 4, False), Action: hit, Value: -0.9555555555555556
State: (10, 4, False), Action: hit, Value: -0.4090909090909091
State: (14, 9, False), Action: hit, Value: -0.7547169811320755
State: (17, 9, False), Action: hit, Value: -0.6153846153846154
State: (13, 9, False), Action: hit, Value: -0.68
State: (14, 4, False), Action: hit, Value: -0.37209302325581395
State: (14, 11, False), Action: hit, Value: -0.5405405405405406
State: (11, 11, False), Action: hit, Value: -0.375
```