WIDS Project - Monte Carlo Assignment
Week 4: Prediction & Control
Solutions

Monte Carlo Methods for Blackjack

Introduction

This document provides complete solutions to the Monte Carlo Prediction and Control assignment. We implement algorithms to learn optimal Blackjack strategies through experience, starting from zero knowledge.

1 Part I: Environment Setup

1.1 Using OpenAI Gymnasium

For this solution, we'll use the Gymnasium library as it provides a standardized Blackjack environment.

```python
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict
from tqdm import tqdm

# Create the Blackjack environment
env = gym.make('Blackjack-v1', sab=True)

# State format: (player_sum, dealer_card, usable_ace)
# Example: (18, 9, False) means player has 18, dealer shows 9, no usable
    ace
```

Listing 1: Environment Setup

2 Part II: Algorithm Implementation

2.1 Task 1: Monte Carlo Prediction

Goal: Estimate the value function $V(s)$ for a fixed policy using First-Visit Monte Carlo.

Policy: Stick if sum is 20 or 21, otherwise hit.

```python
def simple_policy(state):
    """
    Simple policy: Stick on 20 or 21, otherwise hit.

    Args:
        state: Tuple of (player_sum, dealer_card, usable_ace)

    Returns:
        action: 0 (stick) or 1 (hit)
    """
    player_sum = state[0]
    if player_sum >= 20:
        return 0  # Stick
    else:
        return 1  # Hit


def monte_carlo_prediction(env, policy, num_episodes=10000):
    """
    First-Visit Monte Carlo Prediction.

    Estimates V(s) by averaging returns from first visits to each state.

    Args:
        env: Gymnasium environment
        policy: Function that takes state and returns action
        num_episodes: Number of episodes to run

    Returns:
        V: Dictionary mapping states to their estimated values
    """
    # Store sum of returns and visit counts for each state
    returns_sum = defaultdict(float)
    returns_count = defaultdict(int)
    V = defaultdict(float)

    for episode in tqdm(range(num_episodes), desc="MC Prediction"):
        # Generate an episode
        episode_data = []
        state, info = env.reset()
        done = False

        while not done:
            action = policy(state)
            next_state, reward, terminated, truncated, info = env.step(
    action)
            episode_data.append((state, reward))
            state = next_state
            done = terminated or truncated

        # Track which states we've already seen in this episode
        visited_states = set()

        # Calculate returns for each state (working backwards)
```

```python
        G = 0  # Return
        for state, reward in reversed(episode_data):
            G = reward  # In Blackjack, only terminal reward matters

            # First-visit: only update if we haven't seen this state before
            if state not in visited_states:
                visited_states.add(state)
                returns_sum[state] += G
                returns_count[state] += 1
                V[state] = returns_sum[state] / returns_count[state]

    return V


# Run Monte Carlo Prediction
print("Running Monte Carlo Prediction...")
V = monte_carlo_prediction(env, simple_policy, num_episodes=10000)

# Print results for specific states
print("\n=== Value Function Results ===")
# Find states with player sum 21 and 5
states_21 = [s for s in V.keys() if s[0] == 21]
states_5 = [s for s in V.keys() if s[0] == 5]

if states_21:
    avg_value_21 = np.mean([V[s] for s in states_21])
    print(f"Average Value of Player Sum = 21: {avg_value_21:.4f}")
    print(f"  (This is high because 21 usually wins!)")
else:
    print("No states with player sum 21 visited")

if states_5:
    avg_value_5 = np.mean([V[s] for s in states_5])
    print(f"Average Value of Player Sum = 5: {avg_value_5:.4f}")
    print(f"  (This is low because 5 requires many hits, risking bust)")
else:
    print("No states with player sum 5 visited")
```

Listing 2: Monte Carlo Prediction Implementation

Explanation:

- We run 10,000 episodes following the simple policy
- For each state visited, we track the return (reward) received
- Using First-Visit MC, we only count the first time a state appears in an episode
- $V(s)$ = average of all returns from that state
- Player sum of 21 has high value (usually wins)
- Player sum of 5 has low value (far from 21, many hits needed)

2.2 Task 2: Monte Carlo Control

Goal: Find the optimal policy π^* by learning action-values $Q(s, a)$.

```python
def epsilon_greedy_policy(Q, state, epsilon, n_actions=2):
    """
    Epsilon-greedy policy: explore with probability epsilon,
    exploit (choose best action) with probability 1-epsilon.

    Args:
        Q: Action-value function (dictionary)
        state: Current state
        epsilon: Exploration probability
        n_actions: Number of possible actions

    Returns:
        action: Selected action (0 or 1)
    """
    if np.random.random() < epsilon:
        # Explore: choose random action
        return np.random.randint(n_actions)
    else:
        # Exploit: choose best action
        q_values = [Q[(state, a)] for a in range(n_actions)]
        return np.argmax(q_values)


def monte_carlo_control(env, num_episodes=500000, alpha=0.01,
                        epsilon_start=1.0, epsilon_end=0.1, epsilon_decay
    =0.99999):
    """
    Monte Carlo Control with epsilon-greedy exploration.

    Learns optimal policy by updating Q(s,a) values based on returns.

    Args:
        env: Gymnasium environment
        num_episodes: Number of training episodes
        alpha: Learning rate for Q-value updates
        epsilon_start: Initial exploration rate
        epsilon_end: Final exploration rate
        epsilon_decay: Rate of epsilon decay

    Returns:
        Q: Learned action-value function
        rewards_history: List of rewards per episode
        policy: Optimal policy extracted from Q
    """
    # Initialize Q(s,a) to zero for all state-action pairs
    Q = defaultdict(float)

    # Track rewards for plotting learning curve
    rewards_history = []
    epsilon = epsilon_start

```

```
    for episode in tqdm(range(num_episodes), desc="MC Control"):
        # Generate an episode using epsilon-greedy policy
        episode_data = []
        state, info = env.reset()
        done = False
        episode_reward = 0

        while not done:
            action = epsilon_greedy_policy(Q, state, epsilon)
            next_state, reward, terminated, truncated, info = env.step(action)
            episode_data.append((state, action, reward))
            episode_reward += reward
            state = next_state
            done = terminated or truncated

        rewards_history.append(episode_reward)

        # Update Q-values using returns from this episode
        G = 0  # Return
        visited_state_actions = set()

        for state, action, reward in reversed(episode_data):
            G = reward  # In Blackjack, only terminal reward

            # First-visit update
            if (state, action) not in visited_state_actions:
                visited_state_actions.add((state, action))

                # Incremental update formula
                Q[(state, action)] = Q[(state, action)] + alpha * (G - Q[(state, action)])

        # Decay epsilon (explore less over time)
        epsilon = max(epsilon_end, epsilon * epsilon_decay)

    # Extract optimal policy from Q-values
    policy = {}
    states_visited = set(s for (s, a) in Q.keys())
    for state in states_visited:
        q_values = [Q[(state, a)] for a in range(2)]
        policy[state] = np.argmax(q_values)

    return Q, rewards_history, policy


# Run Monte Carlo Control
print("\nRunning Monte Carlo Control...")
Q, rewards_history, optimal_policy = monte_carlo_control(
    env,
    num_episodes=500000,
    alpha=0.01,
    epsilon_start=1.0,
    epsilon_end=0.1,
```

```
    epsilon_decay=0.99999
)

print(f"\nLearned Q-values for {len(Q)} state-action pairs")
print(f"Derived optimal policy for {len(optimal_policy)} states")
```

Listing 3: Monte Carlo Control Implementation

Explanation:

- **Initialization:** Start with $Q(s, a) = 0$ for all states and actions
- **Epsilon-greedy:** Balance exploration (random actions) and exploitation (best known actions)
- **Episode generation:** Play through game using current policy
- **Q-value update:** $Q(s, a) \leftarrow Q(s, a) + \alpha[G_t - Q(s, a)]$
- **Epsilon decay:** Gradually reduce exploration as we learn
- Over 500,000 episodes, the agent learns which actions lead to wins

2.3 Task 3: Visualization

```
def plot_learning_curve(rewards_history, window=1000):
    """
    Plot the rolling average reward to show learning progress.

    Args:
        rewards_history: List of rewards from each episode
        window: Size of rolling average window
    """
    # Calculate rolling average
    rolling_avg = np.convolve(rewards_history,
                              np.ones(window)/window,
                              mode='valid')

    plt.figure(figsize=(12, 6))
    plt.plot(rolling_avg, linewidth=2)
    plt.xlabel('Episode', fontsize=12)
    plt.ylabel(f'Rolling Average Reward (window={window})', fontsize=12)
    plt.title('Monte Carlo Control: Learning Curve', fontsize=14, fontweight='bold')
    plt.grid(True, alpha=0.3)
    plt.axhline(y=0, color='r', linestyle='--', alpha=0.5, label='Break-even')
    plt.legend()
    plt.tight_layout()
    plt.savefig('learning_curve.png', dpi=300, bbox_inches='tight')
    plt.show()

```

```
    print(f"Final average reward: {np.mean(rewards_history[-10000:]):.4f}"
    )


def plot_strategy_heatmap(policy):
    """
    Create a heatmap showing the optimal action for each state.

    Args:
        policy: Dictionary mapping states to optimal actions
    """
    # Create separate heatmaps for with/without usable ace
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

    # Prepare data grids
    player_range = range(12, 22)  # Player sum from 12 to 21
    dealer_range = range(1, 11)   # Dealer card from 1 (Ace) to 10

    for usable_ace, ax, title in [(False, ax1, 'No Usable Ace'),
                                   (True, ax2, 'Usable Ace')]:
        strategy_grid = np.zeros((len(player_range), len(dealer_range)))

        for i, player_sum in enumerate(player_range):
            for j, dealer_card in enumerate(dealer_range):
                state = (player_sum, dealer_card, usable_ace)
                if state in policy:
                    strategy_grid[i, j] = policy[state]
                else:
                    strategy_grid[i, j] = np.nan

        # Create heatmap
        sns.heatmap(strategy_grid,
                   ax=ax,
                   cmap=['darkred', 'darkgreen'],
                   cbar_kws={'label': 'Action',
                            'ticks': [0.25, 0.75]},
                   xticklabels=['A'] + list(range(2, 11)),
                   yticklabels=player_range,
                   linewidths=0.5,
                   linecolor='gray',
                   vmin=0, vmax=1)

        ax.set_xlabel('Dealer Showing', fontsize=12, fontweight='bold')
        ax.set_ylabel('Player Sum', fontsize=12, fontweight='bold')
        ax.set_title(title, fontsize=14, fontweight='bold')

        # Fix colorbar labels
        cbar = ax.collections[0].colorbar
        cbar.set_ticklabels(['STICK (0)', 'HIT (1)'])

    plt.suptitle('Optimal Blackjack Strategy',
                fontsize=16, fontweight='bold', y=1.02)
    plt.tight_layout()
    plt.savefig('strategy_heatmap.png', dpi=300, bbox_inches='tight')
```

```
    plt.show()


# Generate visualizations
print("\nGenerating visualizations...")
plot_learning_curve(rewards_history, window=1000)
plot_strategy_heatmap(optimal_policy)
```

Listing 4: Visualization Code

Interpretation:

- **Learning Curve:** Shows improvement over time. Should trend upward and stabilize.
- **Strategy Heatmap:**
  - Green = HIT (action 1)
  - Red = STICK (action 0)
  - Generally: stick on high sums (17+), hit on low sums
  - Strategy differs with/without usable ace

3 Part III: Conceptual Questions

3.1 Question A: The Infinite Deck Assumption

3.1.1 1. Non-Stationarity and Markov Property

Answer:

In real casinos with finite decks (6-8 deck shoe), the environment becomes **non-Markovian** if we only use (PlayerSum, DealerCard) as the state.

Why?

The **Markov Property** states that the future depends only on the current state, not on the history. Mathematically:

$$P(S_{t+1}|S_t, S_{t-1}, \ldots, S_0) = P(S_{t+1}|S_t) \quad (1)$$

With a finite deck:

- Cards already dealt affect future probabilities
- If many 10-value cards have been dealt, the probability of drawing another 10 decreases
- The state (18, 10) has different implications early vs. late in the shoe
- **Example:** If 30 face cards have been dealt, hitting on 12 is safer than if only 5 face cards have been dealt

Therefore, (PlayerSum, DealerCard) alone does **not** contain all information needed to predict the future, violating the Markov property.

3.1.2 2. State Space Design for Card Counting

Answer:

To enable card counting, we need to augment the state to capture information about remaining cards:

Option 1: Running Count

$$S = (\text{PlayerSum}, \text{DealerCard}, \text{UsableAce}, \text{RunningCount}) \tag{2}$$

Where RunningCount is:

- +1 for each low card dealt (2-6)
- 0 for each neutral card (7-9)
- -1 for each high card dealt (10, J, Q, K, A)

Option 2: True Count

$$S = (\text{PlayerSum}, \text{DealerCard}, \text{UsableAce}, \text{TrueCount}, \text{DecksRemaining}) \tag{3}$$

Where: $\text{TrueCount} = \frac{\text{RunningCount}}{\text{DecksRemaining}}$

Option 3: Full Deck Composition (Most Markovian)

$$S = (\text{PlayerSum}, \text{DealerCard}, \text{UsableAce}, \mathbf{c}) \tag{4}$$

Where $\mathbf{c} = (c_1, c_2, \ldots, c_{10})$ is a vector of remaining card counts.

3.1.3 3. Trade-off: Convergence Speed

Answer:

Expanding the state space creates a fundamental trade-off:

Pros	**Cons**
More Markovian (better)	Larger state space
Better optimal policy	Slower convergence
Can exploit card counting	Need more episodes
Higher theoretical reward	More memory required

Mathematical Explanation:

With the original state space of size $|S_1| \approx 200$ (combinations of player sum, dealer card, usable ace), adding a running count from -50 to +50 increases the state space to:

$$|S_2| = |S_1| \times 100 \approx 20{,}000 \text{ states} \tag{5}$$

Since Monte Carlo methods require visiting each state-action pair multiple times:

- Episodes needed grows roughly with $|S|$
- $100\times$ more states $\Rightarrow$ $100\times$ more episodes needed
- Each state visited less frequently
- Slower learning, longer training time

Practical Solution: Use function approximation (neural networks) instead of tabular methods for large state spaces.

3.2 Question B: First-Visit vs. Every-Visit

Scenario: $A \to B \to A \to \text{Terminate}$

Rewards:

- $R(A \to B) = +1$
- $R(B \to A) = -1$
- $R(A \to \text{Term}) = +10$

3.2.1 1. First-Visit Monte Carlo

Solution:

In First-Visit MC, we only count the **first time** state A appears.

Calculation:

$$G_{\text{first}} = R_1 + R_2 + R_3 \tag{6}$$
$$= (+1) + (-1) + (+10) \tag{7}$$
$$= +10 \tag{8}$$

The return from the first visit to A is **+10**.

After many episodes, $V(A) \approx$ average of all such first-visit returns.

3.2.2 2. Every-Visit Monte Carlo

Solution:

In Every-Visit MC, we count **both** times state A appears.

Visit 1 (first occurrence):

$$G_1 = (+1) + (-1) + (+10) = +10 \tag{9}$$

Visit 2 (second occurrence):

$$G_2 = (+10) = +10 \tag{10}$$

For this episode, state A contributes two returns: $+10$ and $+10$.

After many episodes, $V(A) \approx$ average of **all** returns from visiting A.

3.2.3 3. Comparison

Method	Returns from A	Average
First-Visit MC	$\{+10\}$	$+10$
Every-Visit MC	$\{+10, +10\}$	$+10$

In this particular example, both methods give the same result. However, in general:

- Every-Visit uses more data (all visits)
- First-Visit has theoretical guarantees under weaker conditions
- Both converge to the true value function $V(s)$ as episodes $\to \infty$

4 Complete Code Solution

Here is the complete, runnable code combining all components:

```python
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict
from tqdm import tqdm

# ============================================
# PART 1: MONTE CARLO PREDICTION
# ============================================

def simple_policy(state):
    """Stick on 20-21, otherwise hit."""
    return 0 if state[0] >= 20 else 1

def monte_carlo_prediction(env, policy, num_episodes=10000):
    """First-Visit MC Prediction."""
    returns_sum = defaultdict(float)
    returns_count = defaultdict(int)
    V = defaultdict(float)

    for _ in tqdm(range(num_episodes), desc="MC Prediction"):
        episode_data = []
        state, _ = env.reset()
        done = False

        while not done:
            action = policy(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            episode_data.append((state, reward))
            state = next_state
            done = terminated or truncated

        visited_states = set()
        G = 0

        for state, reward in reversed(episode_data):
            G = reward
            if state not in visited_states:
                visited_states.add(state)
                returns_sum[state] += G
                returns_count[state] += 1
                V[state] = returns_sum[state] / returns_count[state]

    return V

# ============================================
# PART 2: MONTE CARLO CONTROL
# ============================================
```

```python

def epsilon_greedy_policy(Q, state, epsilon, n_actions=2):
    """Epsilon-greedy action selection."""
    if np.random.random() < epsilon:
        return np.random.randint(n_actions)
    else:
        q_values = [Q[(state, a)] for a in range(n_actions)]
        return np.argmax(q_values)

def monte_carlo_control(env, num_episodes=500000, alpha=0.01):
    """MC Control with epsilon-greedy."""
    Q = defaultdict(float)
    rewards_history = []
    epsilon = 1.0
    epsilon_min = 0.1
    epsilon_decay = 0.99999

    for _ in tqdm(range(num_episodes), desc="MC Control"):
        episode_data = []
        state, _ = env.reset()
        done = False
        episode_reward = 0

        while not done:
            action = epsilon_greedy_policy(Q, state, epsilon)
            next_state, reward, terminated, truncated, _ = env.step(action)
            episode_data.append((state, action, reward))
            episode_reward += reward
            state = next_state
            done = terminated or truncated

        rewards_history.append(episode_reward)

        visited = set()
        G = 0

        for state, action, reward in reversed(episode_data):
            G = reward
            if (state, action) not in visited:
                visited.add((state, action))
                Q[(state, action)] += alpha * (G - Q[(state, action)])

        epsilon = max(epsilon_min, epsilon * epsilon_decay)

    # Extract policy
    policy = {}
    for (state, action) in Q.keys():
        if state not in policy:
            q_vals = [Q[(state, a)] for a in range(2)]
            policy[state] = np.argmax(q_vals)

    return Q, rewards_history, policy

```

```python
# ============================================
# MAIN EXECUTION
# ============================================

if __name__ == "__main__":
    # Setup
    env = gym.make('Blackjack-v1', sab=True)

    # Task 1: Prediction
    print("Task 1: Monte Carlo Prediction")
    V = monte_carlo_prediction(env, simple_policy, 10000)

    states_21 = [s for s in V.keys() if s[0] == 21]
    states_5 = [s for s in V.keys() if s[0] == 5]

    if states_21:
        print(f"V(player_sum=21): {np.mean([V[s] for s in states_21]):.4f}
    ")
    if states_5:
        print(f"V(player_sum=5): {np.mean([V[s] for s in states_5]):.4f}")

    # Task 2: Control
    print("\nTask 2: Monte Carlo Control")
    Q, rewards, policy = monte_carlo_control(env, 500000)
    print(f"Learned policy for {len(policy)} states")
    print(f"Final avg reward: {np.mean(rewards[-10000:]):.4f}")

    env.close()
```

Listing 5: Complete Solution Script

5 Conclusion

This assignment demonstrates the power of Monte Carlo methods:

- **Prediction:** Evaluates a fixed policy by averaging returns
- **Control:** Learns optimal policy through exploration and exploitation
- **Model-free:** No knowledge of transition probabilities needed
- **Experience-based:** Learns purely from playing the game

The agent starts with zero knowledge and discovers the optimal Blackjack strategy through trial and error!