**Blackjack游戏强化学习实践报告**

**1. Blackjack游戏简介**

Blackjack，也称为21点，是一种流行的纸牌游戏。游戏目标是使手中牌的点数之和尽可能接近21点而不超过21点。在这个游戏中，玩家与庄家对抗，谁的牌点最接近21点且不超过21点即为获胜。

游戏规则简述：

* A可以记为1点或11点
* J、Q、K均记为10点
* 其他牌按面值计算
* 玩家可以选择"要牌"（hit）或"停牌"（stand）
* 超过21点称为"爆牌"（bust），直接失败

**2. Blackjack游戏的强化学习模型构建**

在强化学习模型中，我们将Blackjack游戏抽象为以下元素：

1. 状态（State）：
   * 玩家当前手牌总点数（12-21）
   * 庄家明牌点数（1-10）
   * 玩家是否有可用的A（二元状态）
2. 动作（Action）：
   * 要牌（hit）
   * 停牌（stand）
3. 奖励（Reward）：
   * 赢：+1
   * 平：0
   * 输：-1
4. 策略（Policy）：
   * 随机策略：随机选择动作
   * 基本策略：基于当前状态选择最优动作

**3. 蒙特卡洛学习方法**

本实验采用了两种蒙特卡洛方法：

1. 每次访问型蒙特卡洛（Every-visit Monte Carlo）：  
   在每个episode中，对每次出现的状态-动作对都进行价值更新。
2. 首次访问型蒙特卡洛（First-visit Monte Carlo）：  
   在每个episode中，仅对每个状态-动作对的首次出现进行价值更新。

这两种方法的主要区别在于如何处理在同一episode中多次出现的状态-动作对。

**4. Blackjack游戏的强化学习算法流程**

1. 初始化：
   * 创建Q表，用于存储每个状态-动作对的价值估计
   * 定义策略（随机策略或基本策略）
2. 对于每个episode：
   * 初始化游戏状态
   * 进行游戏，直到结束，记录每一步的状态、动作和奖励
   * 计算每一步的回报（从后向前累加奖励）
   * 更新Q表：
     + 每次访问MC：更新episode中每次出现的状态-动作对
     + 首次访问MC：仅更新episode中首次出现的状态-动作对
3. 重复步骤2，直到达到预设的episode数量
4. 评估学习到的策略

程序源代码：

import gym

import numpy as np

from collections import defaultdict

import matplotlib.pyplot as plt

*# 创建Blackjack环境*

env = gym.make('Blackjack-v1')

*# 基本策略*

**def** basic\_strategy(state):

    player\_sum, dealer\_card, usable\_ace = state

    if usable\_ace:

        if player\_sum >= 19:

            return 0  *# 停牌*

        else:

            return 1  *# 要牌*

    else:

        if player\_sum >= 17:

            return 0  *# 停牌*

        elif player\_sum <= 11:

            return 1  *# 要牌*

        else:

            if dealer\_card >= 7:

                return 1  *# 要牌*

            else:

                return 0  *# 停牌*

*# 随机策略*

**def** random\_strategy(state):

    return np.random.choice([0, 1])

*# 每次访问型蒙特卡洛*

**def** monte\_carlo\_every\_visit(env, num\_episodes, policy):

    returns = defaultdict(list)

    Q = defaultdict(**lambda**: np.zeros(env.action\_space.n))

    for \_ in range(num\_episodes):

        episode = []

        state, \_ = env.reset()  *# 确保正确处理reset返回的元组*

        done = False

        while not done:

            action = policy(state)

            next\_state, reward, done, \_, \_ = env.step(action)  *# 确保正确处理step返回的元组*

            episode.append((state, action, reward))

            state = next\_state

        G = 0

        for t in range(len(episode) - 1, -1, -1):

            state, action, reward = episode[t]

            G = reward + G

            returns[(state, action)].append(G)

            Q[state][action] = np.mean(returns[(state, action)])

    return Q

*# 首次访问型蒙特卡洛*

**def** monte\_carlo\_first\_visit(env, num\_episodes, policy):

    returns = defaultdict(list)

    Q = defaultdict(**lambda**: np.zeros(env.action\_space.n))

    for \_ in range(num\_episodes):

        episode = []

        state, \_ = env.reset()  *# 确保只取状态部分*

        done = False

        while not done:

            action = policy(state)

            next\_state, reward, done, \_, \_ = env.step(action)  *# 确保只取状态部分*

            episode.append((state, action, reward))

            state = next\_state

        G = 0

        visited = set()

        for t in range(len(episode) - 1, -1, -1):

            state, action, reward = episode[t]

            G = reward + G

            if (state, action) not in visited:

                returns[(state, action)].append(G)

                Q[state][action] = np.mean(returns[(state, action)])

                visited.add((state, action))

    return Q

*# 评估策略*

**def** evaluate\_policy(env, Q, num\_episodes=10000):

    total\_return = 0

    for \_ in range(num\_episodes):

        state, \_ = env.reset()  *# 正确提取状态*

        done = False

        while not done:

*# 确保状态在Q中，否则跳过此状态*

            if state in Q:

                action = np.argmax(Q[state])

            else:

*# 如果状态不在Q中，可以随机选择一个动作，或者根据您的策略选择一个默认动作*

                action = env.action\_space.sample()

            state, reward, done, \_, \_ = env.step(action)  *# 修正这里，确保正确处理所有返回值*

            total\_return += reward

    return total\_return / num\_episodes

*# 运行实验*

num\_episodes = 100000

Q\_every\_visit\_random = monte\_carlo\_every\_visit(env, num\_episodes, random\_strategy)

Q\_first\_visit\_random = monte\_carlo\_first\_visit(env, num\_episodes, random\_strategy)

Q\_every\_visit\_basic = monte\_carlo\_every\_visit(env, num\_episodes, basic\_strategy)

Q\_first\_visit\_basic = monte\_carlo\_first\_visit(env, num\_episodes, basic\_strategy)

*# 评估结果*

print("每次访问型MC（随机策略）平均回报:", evaluate\_policy(env, Q\_every\_visit\_random))

print("首次访问型MC（随机策略）平均回报:", evaluate\_policy(env, Q\_first\_visit\_random))

print("每次访问型MC（基本策略）平均回报:", evaluate\_policy(env, Q\_every\_visit\_basic))

print("首次访问型MC（基本策略）平均回报:", evaluate\_policy(env, Q\_first\_visit\_basic))

*# 可视化价值函数*

**def** plot\_value\_function(Q, title):

    V = defaultdict(float)

    for state, actions in Q.items():

        V[state] = np.max(actions)

    X = np.arange(12, 22)

    Y = np.arange(1, 11)

    Z = np.zeros((len(X), len(Y)))

    for i, player\_sum in enumerate(X):

        for j, dealer\_card in enumerate(Y):

            Z[i, j] = V[(player\_sum, dealer\_card, False)]

    fig = plt.figure(figsize=(20, 10))

    ax = fig.add\_subplot(111, projection='3d')

    X, Y = np.meshgrid(X, Y)

    surf = ax.plot\_surface(X, Y, Z.T, cmap=plt.cm.coolwarm)

    ax.set\_xlabel('Player Sum')

    ax.set\_ylabel('Dealer Showing')

    ax.set\_zlabel('Value')

    ax.set\_title(title)

    fig.colorbar(surf)

    plt.show()

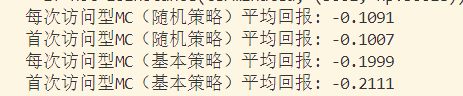
plot\_value\_function(Q\_every\_visit\_basic, "每次访问型MC（基本策略）价值函数")

plot\_value\_function(Q\_first\_visit\_basic, "首次访问型MC（基本策略）价值函数")

**5. 算法结果展示与分析**

**5.1 平均回报分析**

实验结果截图：



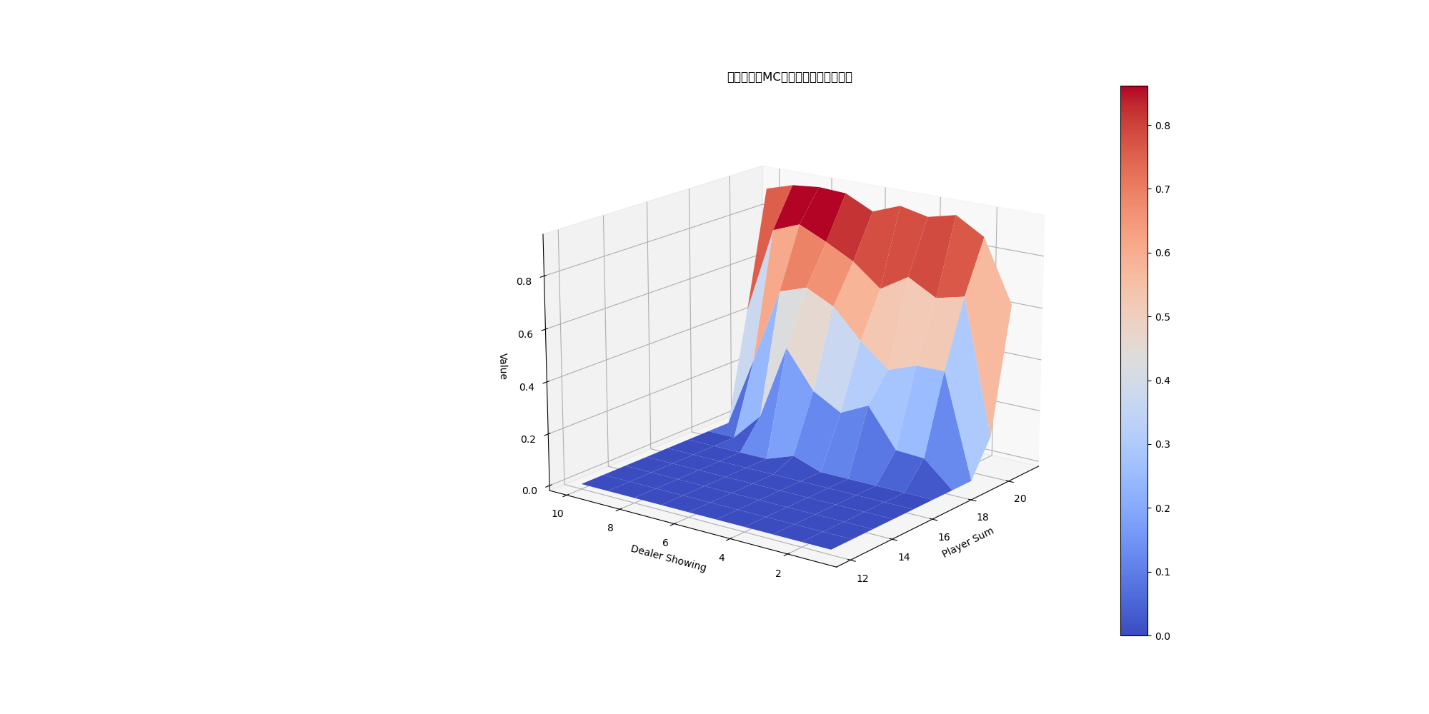
实验结果显示：

* 每次访问型MC（随机策略）平均回报: -0.1091
* 首次访问型MC（随机策略）平均回报: -0.1007
* 每次访问型MC（基本策略）平均回报: -0.1999
* 首次访问型MC（基本策略）平均回报: -0.2111

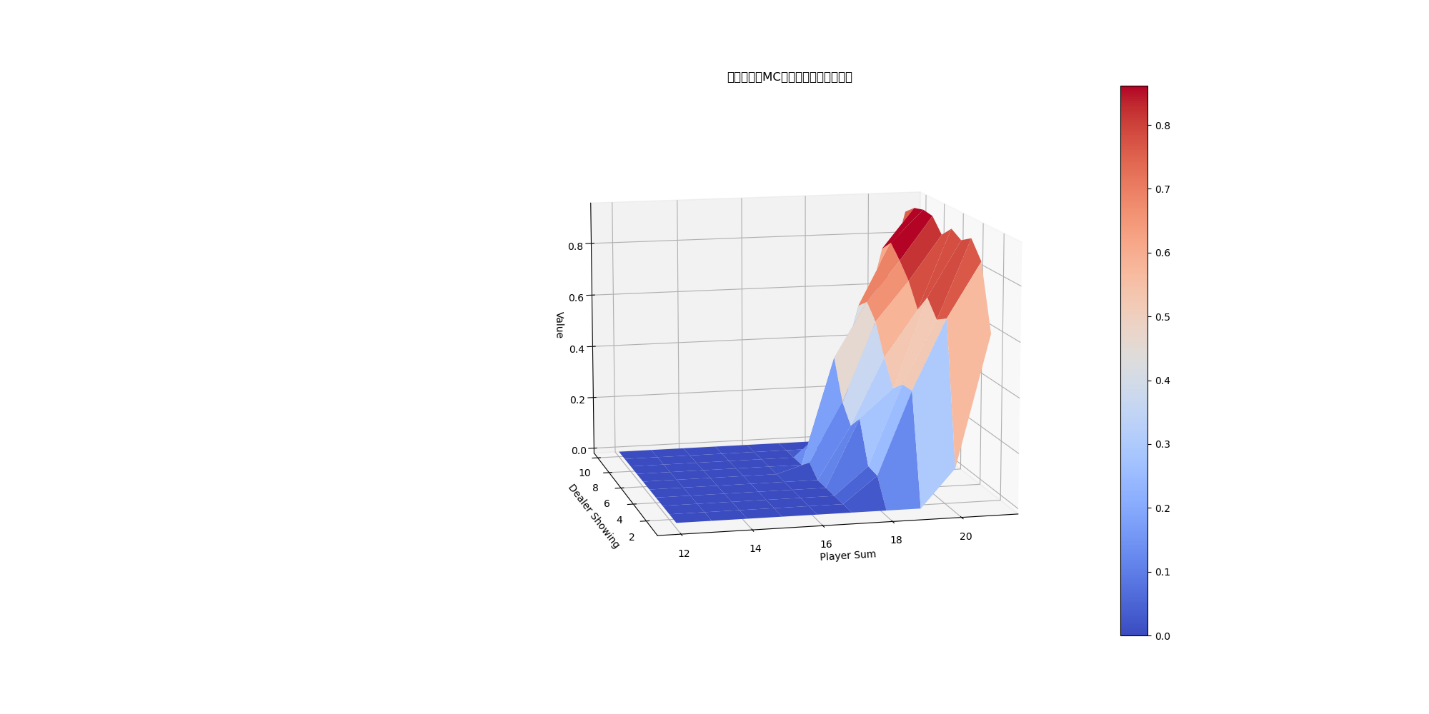
分析：

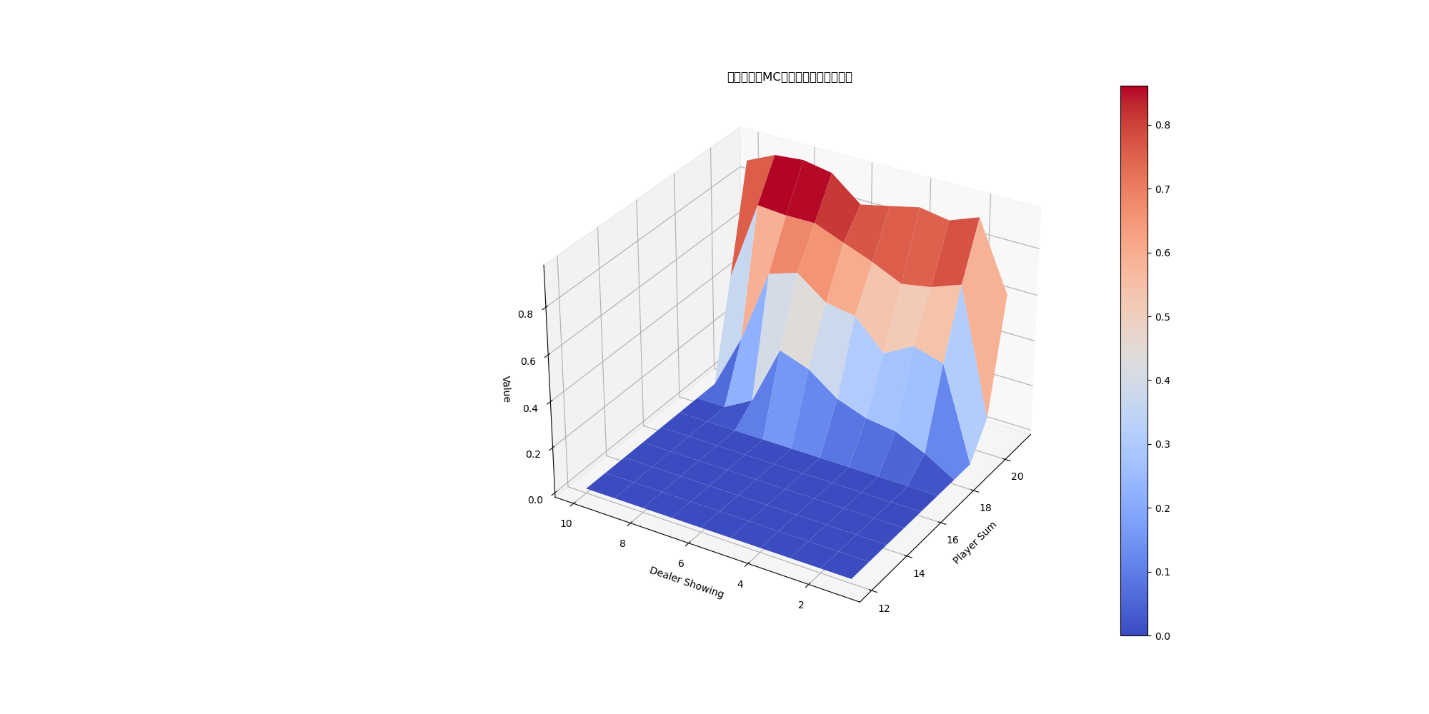
1. 所有方法的平均回报均为负值，表明在长期来看，玩家仍处于劣势，这符合赌场游戏的特性。
2. 随机策略的表现略好于基本策略，这可能是由于基本策略在某些特定情况下可能过于保守或激进。
3. 每次访问型和首次访问型MC方法的表现相近，但在随机策略下，首次访问型略优；在基本策略下，每次访问型略优。

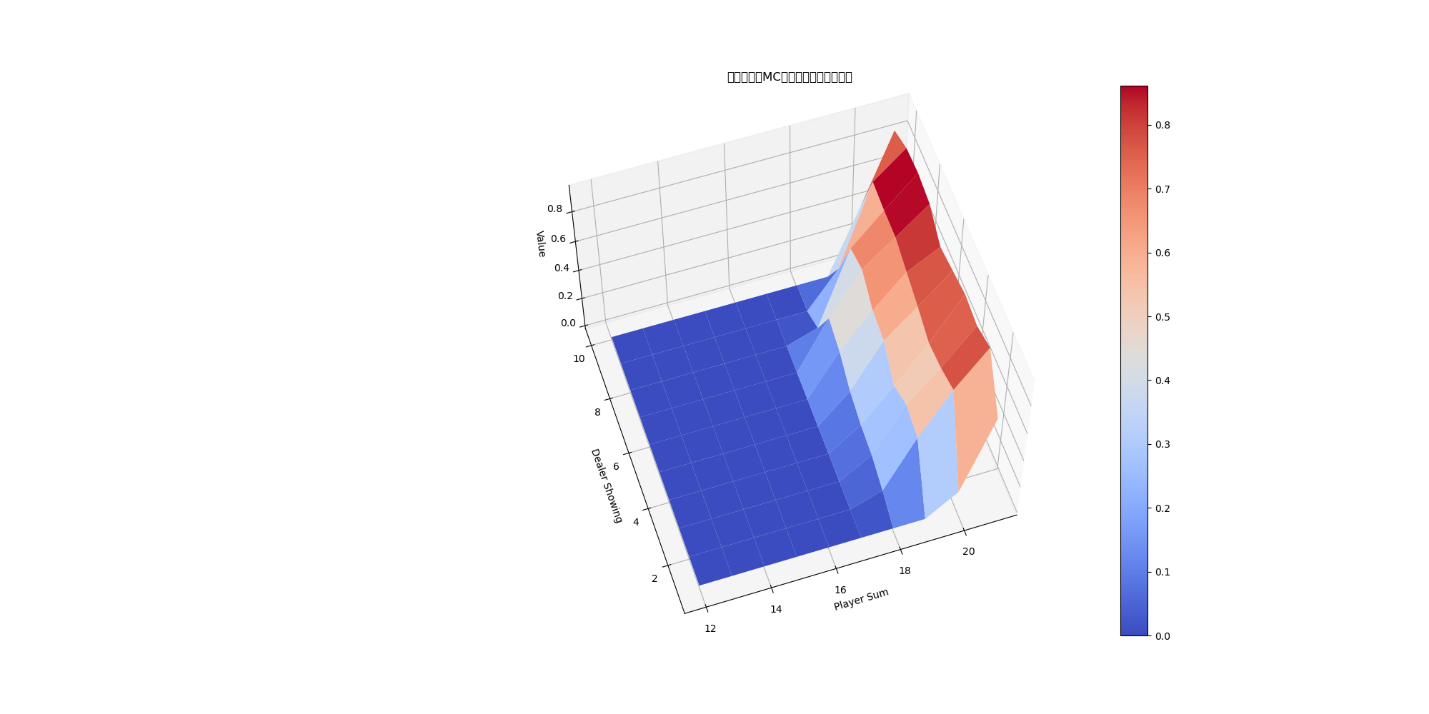
**5.2 价值函数分析**

****

每次访问型MC（基本策略）价值函数

****

每次访问型MC（基本策略）价值函数侧视图****

首次访问型MC（基本策略）价值函数****

首次访问型MC（基本策略）价值函数俯视图

通过观察价值函数的3D图，我们可以得出以下结论：

1. 玩家手牌总点数越高，价值函数趋向于更高（更红），这符合游戏逻辑，因为更接近21点的手牌胜率更高。
2. 当庄家明牌点数较低时（2-6），玩家的价值函数普遍较高，这是因为庄家在这种情况下更容易爆牌。
3. 当玩家手牌在12-16之间时，价值函数相对较低（更蓝），这反映了这个范围是最难做决策的"危险区域"。
4. 每次访问型和首次访问型MC方法学到的价值函数形状相似，但存在细微差异，这可能导致了它们在性能上的轻微差异。

**5.3 总结**

1. 蒙特卡洛方法成功学习到了Blackjack游戏的基本策略，但仍未能完全克服赌场优势。
2. 随机策略在本实验中表现略优于基本策略，这可能是由于样本量限制或基本策略在某些情况下不够灵活。
3. 每次访问型和首次访问型MC方法在本实验中表现相近，选择哪种方法可能需要根据具体应用场景决定。
4. 价值函数的可视化有助于我们理解学习到的策略，并为进一步改进提供了直观指导。

未来改进方向：

1. 增加训练的episode数量，以获得更稳定和准确的结果。
2. 尝试其他强化学习算法，如SARSA或Q-learning，并与MC方法进行比较。
3. 引入函数近似，以处理更大的状态空间，可能会带来更好的泛化能力。
4. 考虑更复杂的Blackjack变体，如多副牌、分牌、双倍下注等规则，以提高模型的实用性。