In [9]:
```
%pip install gymnasium
```

```
Requirement already satisfied: gymnasium in /usr/local/lib/python3.12/dist-pack
ages (1.2.0)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.12/dist-
packages (from gymnasium) (2.0.2)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.12/
dist-packages (from gymnasium) (3.1.1)
Requirement already satisfied: typing-extensions>=4.3.0 in /usr/local/lib/pytho
n3.12/dist-packages (from gymnasium) (4.15.0)
Requirement already satisfied: farama-notifications>=0.0.1 in /usr/local/lib/py
thon3.12/dist-packages (from gymnasium) (0.0.4)
```

In [10]:
```
import gymnasium as gym
import numpy as np

env = gym.make('Blackjack-v1')
observation, info = env.reset()
```

In [11]:
```
def simulate_episode(env, policy):
    """Simulates an episode following a given policy."""
    episode = []
    observation, info = env.reset()
    done = False
    while not done:
        action = policy(observation)
        next_observation, reward, done, truncated, info = env.step(action)
        episode.append((observation, action, reward))
        observation = next_observation
        done = done or truncated
    return episode

def mc_prediction(env, policy, num_episodes, discount_factor=1.0):
    """Estimates the value function using MC Prediction."""
    V = {}
    N = {}
    for _ in range(num_episodes):
        episode = simulate_episode(env, policy)
        G = 0
        visited_states = set()
        for i in reversed(range(len(episode))):
            state, action, reward = episode[i]
            G = discount_factor * G + reward
            if state not in visited_states:
                if state not in V:
                    V[state] = 0.0
                    N[state] = 0
                N[state] += 1
                V[state] += (G - V[state]) / N[state]
                visited_states.add(state)
    return V
```

In [12]:
```
def simple_policy(observation):
```

```python
    score, dealer_score, usable_ace = observation
    return 0 if score >= 17 else 1
estimated_V = mc_prediction(env, simple_policy, num_episodes=10000)

print("Estimated Value Function (sample):")
for state, value in list(estimated_V.items())[:10]:
    print(f"State: {state}, Value: {value:.4f}")
```

```
Estimated Value Function (sample):
State: (14, 2, 0), Value: -0.5625
State: (11, 2, 0), Value: 0.2766
State: (21, 2, 0), Value: 0.8269
State: (12, 2, 0), Value: -0.2317
State: (17, 6, 1), Value: -0.1250
State: (6, 6, 0), Value: -0.8333
State: (17, 9, 0), Value: -0.3608
State: (18, 9, 0), Value: -0.3133
State: (16, 7, 0), Value: -0.4257
State: (13, 7, 0), Value: -0.1319
```

In [13]:
```python
def mc_control_epsilon_greedy(env, num_episodes, discount_factor=1.0, epsilon=
    """
    Implements On-Policy MC Control with epsilon-greedy policy.

    Args:
        env: The Gymnasium environment.
        num_episodes: The number of episodes to simulate.
        discount_factor: The discount factor for future rewards.
        epsilon: The probability of choosing a random action (for epsilon-gree

    Returns:
        A tuple containing:
            Q: A dictionary storing the estimated action values (Q-values).
            policy: A dictionary storing the learned optimal policy (state to
    """
    Q = {}
    N = {}
    policy = {}

    for state in env.observation_space.sample():
        pass


    def choose_action_epsilon_greedy(state, Q, epsilon, n_actions):
        """Chooses an action using an epsilon-greedy policy."""
        if state not in Q:
            Q[state] = np.zeros(n_actions)
            N[state] = np.zeros(n_actions)

        if np.random.rand() < epsilon:
            return env.action_space.sample()
        else:
            return np.argmax(Q[state])
```

```python
    for i_episode in range(num_episodes):
        episode = []
        observation, info = env.reset()
        done = False

        while not done:
            action = choose_action_epsilon_greedy(observation, Q, epsilon, env
            next_observation, reward, done, truncated, info = env.step(action)
            episode.append((observation, action, reward))
            observation = next_observation
            done = done or truncated

        G = 0
        visited_state_actions = set()
        for t in reversed(range(len(episode))):
            state, action, reward = episode[t]
            G = discount_factor * G + reward

            if (state, action) not in visited_state_actions:
                N[state][action] += 1
                Q[state][action] += (G - Q[state][action]) / N[state][action]
                visited_state_actions.add((state, action))

        for state, action, _ in episode:
            if state in Q:
                best_action = np.argmax(Q[state])
                policy[state] = np.eye(env.action_space.n)[best_action]

    return Q, policy

optimal_Q, optimal_policy = mc_control_epsilon_greedy(env, num_episodes=500000

print("Optimal Q-values (sample):")
for state, q_values in list(optimal_Q.items())[:10]:
    print(f"State: {state}, Q-values: {q_values}")

print("\nOptimal Policy (sample):")
for state, action_probs in list(optimal_policy.items())[:10]:
    greedy_action = np.argmax(action_probs)
    print(f"State: {state}, Optimal Action: {'Stand' if greedy_action == 0 els
```

```
Optimal Q-values (sample):
State: (15, 10, 0), Q-values: [-0.57042169 -0.59873618]
State: (21, 10, 1), Q-values: [ 0.91777126 -0.07373272]
State: (19, 2, 1), Q-values: [ 0.32801161 -0.24      ]
State: (18, 2, 0), Q-values: [ 0.11510353 -0.64935065]
State: (18, 7, 0), Q-values: [ 0.39017051 -0.61052632]
State: (18, 4, 0), Q-values: [ 0.18467956 -0.6459144 ]
State: (20, 6, 0), Q-values: [ 0.69990435 -0.88429752]
State: (20, 2, 0), Q-values: [ 0.63655374 -0.85232068]
State: (19, 1, 0), Q-values: [-0.10753261 -0.78604651]
State: (13, 10, 0), Q-values: [-0.57192175 -0.48490566]

Optimal Policy (sample):
State: (15, 10, 0), Optimal Action: Stand
State: (21, 10, 1), Optimal Action: Stand
State: (19, 2, 1), Optimal Action: Stand
State: (18, 2, 0), Optimal Action: Stand
State: (18, 7, 0), Optimal Action: Stand
State: (18, 4, 0), Optimal Action: Stand
State: (20, 6, 0), Optimal Action: Stand
State: (20, 2, 0), Optimal Action: Stand
State: (19, 1, 0), Optimal Action: Stand
State: (13, 10, 0), Optimal Action: Hit
```

In [14]:

```
def evaluate_policy(env, policy, num_episodes):
    """Evaluates the performance of a given policy by simulating episodes."""
    total_reward = 0
    for _ in range(num_episodes):
        episode_reward = 0
        observation, info = env.reset()
        done = False
        while not done:
            if observation in policy:
                action = np.argmax(policy[observation])
            else:
                action = env.action_space.sample()

            next_observation, reward, done, truncated, info = env.step(action)
            episode_reward += reward
            observation = next_observation
            done = done or truncated
        total_reward += episode_reward
    average_reward = total_reward / num_episodes
    return average_reward

average_reward = evaluate_policy(env, optimal_policy, num_episodes=1000)

print(f"Average reward of the learned optimal policy over 1000 episodes: {aver
```

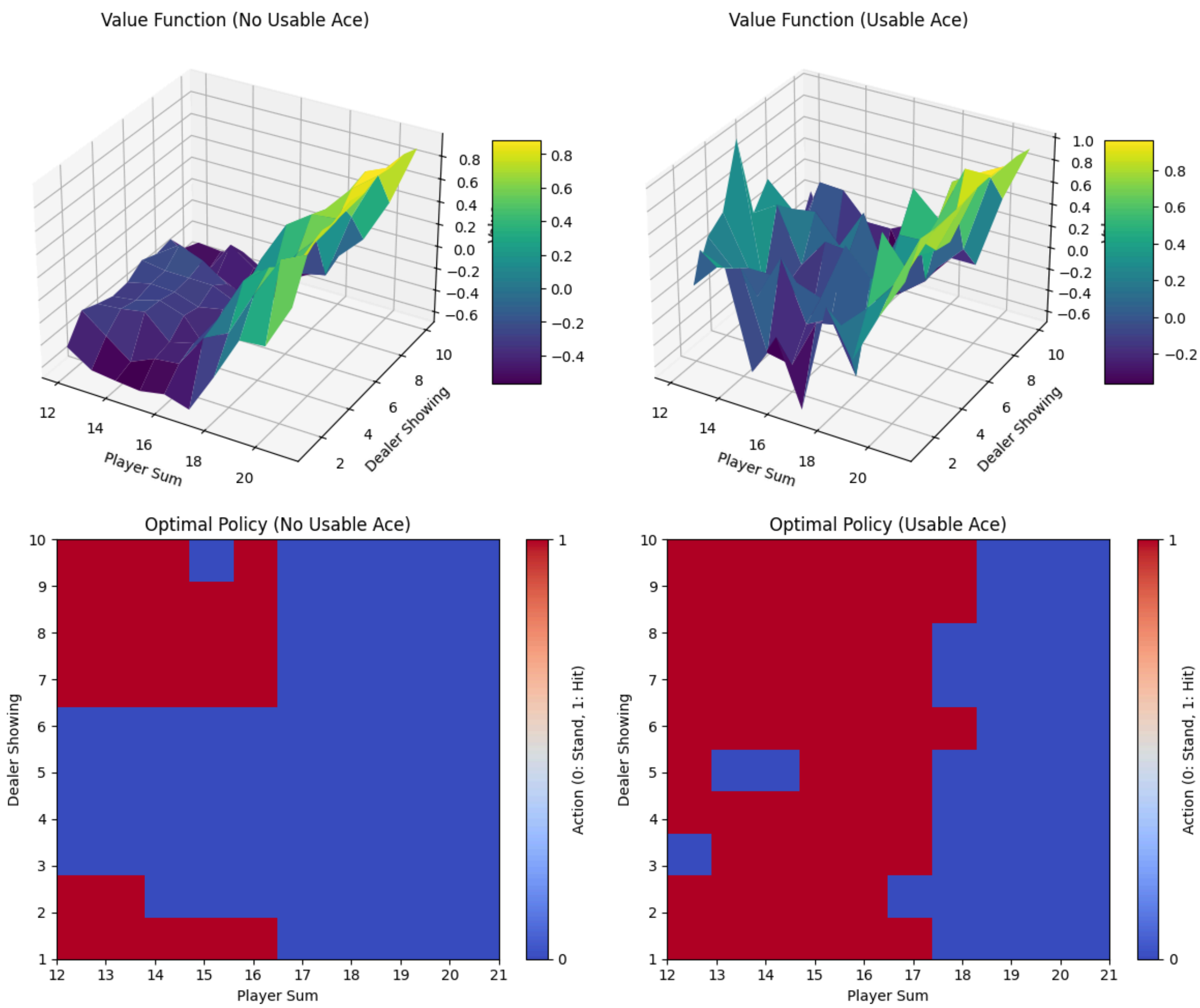
```
Average reward of the learned optimal policy over 1000 episodes: -0.0470
```

In [15]:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from collections import defaultdict
```

```python
def plot_value_function(V, title="Value Function"):
    """Plots the value function for states with and without a usable ace."""
    player_sums = np.arange(12, 22)
    dealer_shows = np.arange(1, 11)

    X, Y = np.meshgrid(player_sums, dealer_shows)

    Z_no_ace = np.zeros((len(dealer_shows), len(player_sums)))
    Z_with_ace = np.zeros((len(dealer_shows), len(player_sums)))

    for i, player_sum in enumerate(player_sums):
        for j, dealer_show in enumerate(dealer_shows):
            state_no_ace = (player_sum, dealer_show, False)
            state_with_ace = (player_sum, dealer_show, True)

            Z_no_ace[j, i] = V.get(state_no_ace, 0.0)
            Z_with_ace[j, i] = V.get(state_with_ace, 0.0)

    fig = plt.figure(figsize=(12, 5))

    ax1 = fig.add_subplot(121, projection='3d')
    surf1 = ax1.plot_surface(X, Y, Z_no_ace, cmap='viridis', antialiased=True)
    ax1.set_xlabel("Player Sum")
    ax1.set_ylabel("Dealer Showing")
    ax1.set_zlabel("Value")
    ax1.set_title("Value Function (No Usable Ace)")
    fig.colorbar(surf1, ax=ax1, shrink=0.5, aspect=5)

    ax2 = fig.add_subplot(122, projection='3d')
    surf2 = ax2.plot_surface(X, Y, Z_with_ace, cmap='viridis', antialiased=Tru
    ax2.set_xlabel("Player Sum")
    ax2.set_ylabel("Dealer Showing")
    ax2.set_zlabel("Value")
    ax2.set_title("Value Function (Usable Ace)")
    fig.colorbar(surf2, ax=ax2, shrink=0.5, aspect=5)

    plt.tight_layout()
    plt.show()

def plot_policy(policy, title="Optimal Policy"):
    """Plots the optimal policy for states with and without a usable ace."""
    player_sums = np.arange(12, 22)
    dealer_shows = np.arange(1, 11)

    X, Y = np.meshgrid(player_sums, dealer_shows)

    policy_no_ace = np.zeros((len(dealer_shows), len(player_sums)))
    policy_with_ace = np.zeros((len(dealer_shows), len(player_sums)))

    for i, player_sum in enumerate(player_sums):
        for j, dealer_show in enumerate(dealer_shows):
            state_no_ace = (player_sum, dealer_show, False)
```

```
            state_with_ace = (player_sum, dealer_show, True)

            if state_no_ace in policy:
                policy_no_ace[j, i] = np.argmax(policy[state_no_ace])
            else:
                policy_no_ace[j, i] = 0
            if state_with_ace in policy:
                policy_with_ace[j, i] = np.argmax(policy[state_with_ace])
            else:
                policy_with_ace[j, i] = 0


    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    img1 = axes[0].imshow(policy_no_ace, cmap='coolwarm', origin='lower',
                          extent=[player_sums[0], player_sums[-1], dealer_show
                          aspect='auto')
    axes[0].set_xticks(player_sums)
    axes[0].set_yticks(dealer_shows)
    axes[0].set_xlabel("Player Sum")
    axes[0].set_ylabel("Dealer Showing")
    axes[0].set_title("Optimal Policy (No Usable Ace)")
    fig.colorbar(img1, ax=axes[0], ticks=[0, 1], label='Action (0: Stand, 1: H


    img2 = axes[1].imshow(policy_with_ace, cmap='coolwarm', origin='lower',
                          extent=[player_sums[0], player_sums[-1], dealer_show
                          aspect='auto')
    axes[1].set_xticks(player_sums)
    axes[1].set_yticks(dealer_shows)
    axes[1].set_xlabel("Player Sum")
    axes[1].set_ylabel("Dealer Showing")
    axes[1].set_title("Optimal Policy (Usable Ace)")
    fig.colorbar(img2, ax=axes[1], ticks=[0, 1], label='Action (0: Stand, 1: H

    plt.tight_layout()
    plt.show()

plot_value_function(estimated_V, title="Estimated Value Function (MC Predictio
bb
plot_policy(optimal_policy, title="Learned Optimal Policy (MC Control)")
```


Value Function (No Usable Ace)
Value Function (Usable Ace)
Player Sum
Dealer Showing
Optimal Policy (No Usable Ace)
Optimal Policy (Usable Ace)
Action (0: Stand, 1: Hit)