Kernel: Python 3 (system-wide)

Assignment 6 - Reinforcement Learning

Fides Regina Schwartz

Netid: fs113

Learning objectives

After completing this assignment, you will be able to...

- Clearly articulate the role of the key components of reinforcement learning: the agent, actions, rewards, policies, state values, and action values.
- Apply policy evaluation to a problem in practice
- Use Monte Carlo control to determine and apply an optimal policy for a reinforcement learning problem and learn an optimal strategy from trial and error, alone

Blackjack

Your goal is to develop a reinforcement learning technique to learn the optimal policy for winning at blackjack through trial-and-error learning. Here, we're going to modify the rules from traditional blackjack a bit in a way that corresponds to the game presented in Sutton and Barto's *Reinforcement Learning: An Introduction* (Chapter 5, example 5.1). A full implementation of the game is provided and usage examples are detailed in the class header below.

The rules of this modified version of the game of blackjack are as follows:

- Blackjack is a card game where the goal is to obtain cards that sum to as near as possible to 21 without going over. We're playing against a fixed (autonomous) dealer.
- Face cards (Jack, Queen, King) have point value 10. Aces can either count as 11 or 1 (whichever is most advantageous to the player), and we're refer to it as a 'usable' Ace if we're treating it as 11 (indicating that it could be used as a '1', instead, if need be). This game is played with a deck of cards sampled with replacement.
- The game starts with both the player and the dealer having one face up and one face down card.
- The player can request additional cards (known as taking a "hit", which we define as action "1") until either they decide to stop (known as "staying", which we define as action '0') or their cards exceed 21 (known as a "bust", at which time the game ends and player loses).
- If the player stays (and hasn't exceeded a score of 21), the dealer reveals their facedown card, and draws until their sum is 17 or greater. If the dealer busts the player wins. If neither player nor dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21. The reward for winning is +1, tying is 0, and losing is -1.

Over the course of these exercises, you will accomplish three things:

1. Try your hand at this game of blackjack and see what your human reinforcement learning system is able to achieve

2. Evaluate a naive policy using Monte Carlo policy evaluation

3. Determine an optimal policy using Monte Carlo control

This problem is adapted from David Silver's excellent series on Reinforcement Learning at University College London

1

[5 points] Human reinforcement learning

Using the code detailed below, play at least 20 hands of the modified blackjack game below, and record your returns (average cumulative reward) across all episodes. This will help you get accustomed with how the game works, the data structures involved with representing states, and what strategies are most effective. Since for this game, you only get a nonzero reward at the end of the episode, you can simply play the game and note the reward in a spreadsheet, then average across all plays of the game (of course, you're welcome to code up a way to do that automatically if you're so inspired).

In [1]:
```
import numpy as np

class Blackjack():
    """Simple blackjack environment adapted from OpenAI Gym:
        https://github.com/openai/gym/blob/master/gym/envs/toy_text/blackjack.py

    Blackjack is a card game where the goal is to obtain cards that sum to as
    near as possible to 21 without going over.  They're playing against a fixed
    dealer.

    Face cards (Jack, Queen, King) have point value 10.
    Aces can either count as 11 or 1, and it's called 'usable' at 11.
    This game is placed with a deck sampled with replacement.

    The game starts with each (player and dealer) having one face up and one
    face down card.

    The player can request additional cards (hit = 1) until they decide to stop
    (stay = 0) or exceed 21 (bust).

    After the player stays, the dealer reveals their facedown card, and draws
    until their sum is 17 or greater.  If the dealer goes bust the player wins.
    If neither player nor dealer busts, the outcome (win, lose, draw) is
    decided by whose sum is closer to 21.  The reward for winning is +1,
    drawing is 0, and losing is -1.

    The observation is a 3-tuple of: the players current sum,
    the dealer's one showing card (1-10 where 1 is ace),
    and whether or not the player holds a usable ace (0 or 1).

    This environment corresponds to the version of the blackjack problem
    described in Example 5.1 in Reinforcement Learning: An Introduction
    by Sutton and Barto (1998).

    http://incompleteideas.net/sutton/book/the-book.html

    Usage:
        Initialize the class:
            game = Blackjack()

        Deal the cards:
            game.deal()

             (14, 3, False)

            This is the agent's observation of the state of the game:
```

```
        The first value is the sum of cards in your hand (14 in this case)
        The second is the visible card in the dealer's hand (3 in this case)
        The Boolean is a flag (False in this case) to indicate whether or
            not you have a usable Ace
        (Note: if you have a usable ace, the sum will treat the ace as a
            value of '11' - this is the case if this Boolean flag is "true")

    Take an action: Hit (1) or stay (0)

        Take a hit: game.step(1)
        To Stay:    game.step(0)

    The output summarizes the game status:

        ((15, 3, False), 0, False)

        The first tuple (15, 3, False), is the agent's observation of the
        state of the game as described above.
        The second value (0) indicates the rewards
        The third value (False) indicates whether the game is finished
"""

def __init__(self):
    # 1 = Ace, 2-10 = Number cards, Jack/Queen/King = 10
    self.deck   = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]
    self.dealer = []
    self.player = []
    self.deal()

def step(self, action):
    if action == 1:  # hit: add a card to players hand and return
        self.player.append(self.draw_card())
        if self.is_bust(self.player):
            done = True
            reward = -1
        else:
            done = False
            reward = 0
    else:  # stay: play out the dealers hand, and score
        done = True
        while self.sum_hand(self.dealer) < 17:
            self.dealer.append(self.draw_card())
        reward = self.cmp(self.score(self.player), self.score(self.dealer))
    return self._get_obs(), reward, done, self.dealer

def _get_obs(self):
    return (self.sum_hand(self.player), self.dealer[0], self.usable_ace(self.player))

def deal(self):
    self.dealer = self.draw_hand()
    self.player = self.draw_hand()
    return self._get_obs()

#----------------------------------------
# Other helper functions
#----------------------------------------
def cmp(self, a, b):
    return float(a > b) - float(a < b)

def draw_card(self):
    return int(np.random.choice(self.deck))

def draw_hand(self):
    return [self.draw_card(), self.draw_card()]

def usable_ace(self,hand):  # Does this hand have a usable ace?
    return 1 in hand and sum(hand) + 10 <= 21

def sum_hand(self,hand):  # Return current hand total
    if self.usable_ace(hand):
```

```
            return sum(hand) + 10
        return sum(hand)

    def is_bust(self,hand):  # Is this hand a bust?
        return self.sum_hand(hand) > 21

    def score(self,hand):  # What is the score of this hand (0 if bust)
        return 0 if self.is_bust(hand) else self.sum_hand(hand)
```

Here's an example of how it works to get you started:

In [2]:
```
import numpy as np

# Initialize the class:
game = Blackjack()

# Deal the cards:
s0 = game.deal()
print(s0)

# If you wanted to stay:
# game.step(2)

# When it's gameover, just redeal:
# game.deal()
```

Out[2]:
```
(21, 4, True)
```

In [5]:
```
# Take an action: Hit = 1 or stay = 0. Here's a hit:
s1 = game.step(0)
print(s1)
```

Out[5]:
```
((17, 9, False), -1.0, True, [9, 10])
```

ANSWER

My rewards for each episode played were: [-1, 0, -1, 1, 1, -1, 0, 1, 1, 1, -1, -1, -1, 1, -1, -1, 1, 1, 1, 1]

My average reward was [0.047619048] after playing 21 hands.

2

[40 points] Perform Monte Carlo Policy Evaluation

Thinking that you want to make your millions playing this modified version of blackjack, you decide to test out a policy for playing this game. Your idea is an aggressive strategy: always hit unless the total of your cards adds up to 20 or 21, in which case you stay.

(a) Use Monte Carlo policy evaluation to evaluate the expected returns from each state. Create plots for these similar to Sutton and Barto, Figure 5.1 where you plot the expected returns for each state. In this case create 2 plots (sample code is provided below):

1. When you have a usable ace, plot the value function showing the dealer's card on the x-axis, and the player's sum on the y-axis, and use a colormap to plot the state value corresponding to each state under the policy described above. The domain of your x and y axes should include all possible states (2 to 21 for the player sum, and 1 to 10 for the dealer's card). Show the estimated state value function after 10,000 episodes.

2. Repeat (1) for the states without a usable ace (code is also provided for this).

3. Repeat (1) after 500,000 episodes.

4. Repeat (2) after 500,000 episodes.

(b) Show a plot of the cumulative average reward (returns) per episode vs the number of episodes.

- For this plot, make the x-axis log scale
- For both the 10,000 episode case and the 500,000 episode case, state the final value of that average returns for this policy in those two cases (these are just the values of the plot of cumulative average reward at iteration 10,000 and 500,000, respectively), you can write this in a line of text.

Note on sample code: the code provided for these questions is meant to be a helpful starting point - you are not required to fill it out exactly or use all components, but it is meant to help you as you begin thinking about this problem.

ANSWER

a.1

In [98]:
```
from collections import defaultdict
from functools import partial
```

In [99]:
```
B = Blackjack()

#Define a policy where I hit until I reach 20 or more.
def sample_policy(observation):
    score, dealer_score, usable_ace = observation
    return 0 if score >= 20 else 1


#Generate an episode ending in: win, loss or draw

def generate_episode(policy, B):

    # Initialize the list for storing states, actions, and rewards
    states, actions, rewards = [], [], []

    # Initialize the game
    observation = B.deal()

    while True:

        # append states to the states list
        states.append(observation)

        # select an action using the sample_policy function
        action = sample_policy(observation)

        # then append the action to actions list
        actions.append(action)

        # Perform the action in the environment according to the sample_policy, then move to
the next state
        observation, reward, done, info = B.step(action)

        # append receive reward
        rewards.append(reward)

        # Break if game is done
        if done:
             break

    return states, actions, rewards
```

In [107]:
```
def first_visit_mc_prediction(policy, B, n_episodes):
```

```
    # Initialize the empty value table as a dictionary
    value_table = defaultdict(float)

    # store the values of each state
    N = defaultdict(int)


    for _ in range(n_episodes): # Generate the epsiode and store the states and rewards
        states, _, rewards = generate_episode(policy, B)
        returns = 0

        # Store the rewards to a variable R and states to S
        for t in range(len(states) - 1, -1, -1):
            R = rewards[t]
            S = states[t]

            # calculate returns as a sum of rewards
            returns += R

            # Take the average of returns and assign the value of the state as an average of
returns

            if S not in states[:t]:
                N[S] += 1
                value_table[S] += (returns - value_table[S]) / N[S]

    return value_table
```

a.2 Evaluate 10,000 episodes

In [101]:
```
value1 = first_visit_mc_prediction(sample_policy, B, n_episodes=10000) # evaluate policy for
10,000 episodes


for i in range(10):
  print(value1.popitem())
```

Out[101]:
```
((12, 3, True), 0.0)
((4, 6, False), -0.6666666666666666)
((12, 2, True), -1.0)
((12, 8, True), 0.2)
((4, 8, False), -0.3333333333333333)
((12, 9, True), 0.0)
((12, 5, True), -1.0)
((12, 6, True), 1.0)
((13, 7, True), -0.6666666666666666)
((12, 1, True), -0.5)
```

a.3 and a.4 Evaluate 500,000 episodes

In [102]:
```
value2 = first_visit_mc_prediction(sample_policy, B, n_episodes=500000)


for i in range(10):
  print(value2.popitem())
```

Out[102]:
```
((4, 7, False), -0.36086956521739116)
((14, 2, True), -0.27235772357723576)
((5, 7, False), -0.4492753623188405)
((5, 1, False), -0.5962343096234307)
((12, 3, True), -0.35545023696682476)
((16, 7, True), -0.2938311688311686)
((15, 6, True), -0.2919580419580418)
((4, 5, False), -0.42570281124498)
((17, 1, True), -0.5782414307004469)
((4, 9, False), -0.4202334630350192)
```

a Plot the state value functions

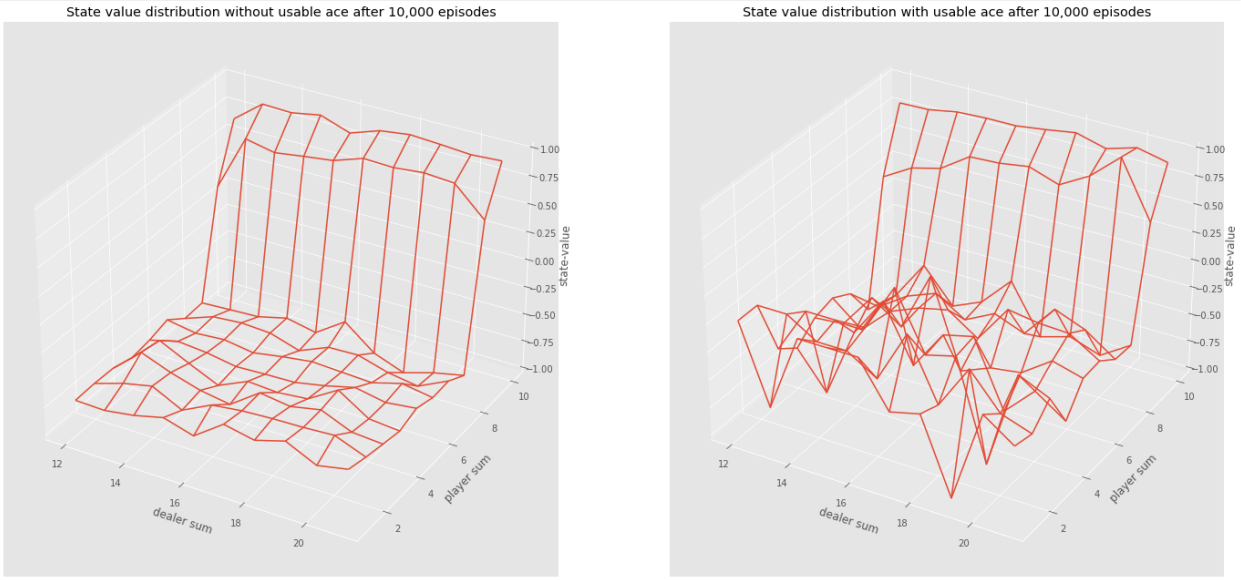
In [67]:
```
from matplotlib import pyplot
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from functools import partial
%matplotlib inline
plt.style.use('ggplot')
```

In [68]:
```
def plot_blackjack(V, ax1, ax2):
    player_sum = np.arange(12, 21 + 1)
    dealer_sum = np.arange(1, 10 + 1)
    usable_ace = np.array([False, True])
    state_values = np.zeros((len(player_sum), len(dealer_sum), len(usable_ace)))

    for i, player in enumerate(player_sum):
        for j, dealer in enumerate(dealer_sum):
            for k, ace in enumerate(usable_ace):
                state_values[i, j, k] = V[player, dealer, ace]

    X, Y = np.meshgrid(player_sum, dealer_sum)

    ax1.plot_wireframe(X, Y, state_values[:, :, 0])
    ax2.plot_wireframe(X, Y, state_values[:, :, 1])

    for ax in ax1, ax2:
        ax.set_zlim(-1, 1)
        ax.set_ylabel('player sum')
        ax.set_xlabel('dealer sum')
        ax.set_zlabel('state-value')
```

Plot 10,000 episodes

In [69]:
```
fig, axes = pyplot.subplots(ncols=2, figsize=(24, 12),
subplot_kw={'projection': '3d'})
axes[0].set_title('State value distribution without usable ace after 10,000 episodes')
axes[1].set_title('State value distribution with usable ace after 10,000 episodes')
plot_blackjack(value1, axes[0], axes[1])
```

Out[69]:

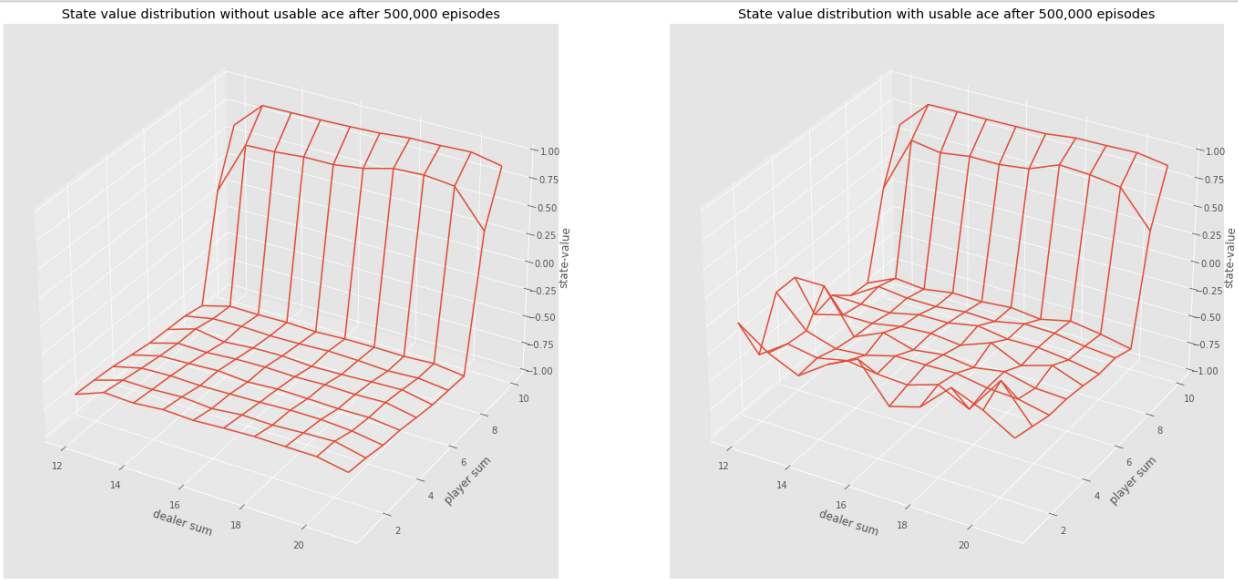


Plot 500,000 episodes

In [70]:
```
fig, axes = pyplot.subplots(ncols=2, figsize=(24, 12),
subplot_kw={'projection': '3d'})
axes[0].set_title('State value distribution without usable ace after 500,000 episodes')
```

```
axes[1].set_title('State value distribution with usable ace after 500,000 episodes')
plot_blackjack(value2, axes[0], axes[1])
```

Out[70]:



(b) Show a plot of the cumulative average reward (returns) per episode vs the number of episodes.

In [0]:
```
for i in range(10):
    plt.plot(value1.popitem(-1))
    plt.title("Cumulative average returns per episode")
    plt.xlabel("Games Played")
    plt.ylabel("Average reward")
    plt.xscale("log")
    plt.show()
```

(b.2) For both the 10,000 episode case and the 500,000 episode case, state the final value of that average returns for this policy in those two cases (these are just the values of the plot of cumulative average reward at iteration 10,000 and 500,000, respectively), you can write this in a line of text.

In [114]:
```
print("The final value of the average returns for 10,000 episodes is: -0.3 and for 500,000 episodes it is: -0.07.")
```

Out[114]:
```
The final value of the average returns for 10,000 episodes is: -0.3 and for 500,000 episodes it is: -0.07.
```

3

[40 points] Perform Monte Carlo Control

Now it's time to actually implement a reinforcement learning strategy that learns to play this version of Blackjack well, only through trial-and-error learning. Here, you will develop your Monte Carlo Control algorithm and evaluate its performance for our Blackjack-like game.

(a) Using Monte Carlo Control through policy iteration, estimate the optimal policy for playing our modified blackjack game to maximize rewards.

In doing this, use the following assumptions:

1. Initialize the state value function and the action value function to all zeros
2. Keep a running tally of the number of times the agent visited each state and chose an action. $N(s_t, a_t)$ is the number of times action a has been selected from state s. You'll need this to compute the running average. You

can implement an online average as: $\bar{x}_t = \frac{1}{N} x_t + \frac{N-1}{N} \bar{x}_{t-1}$

3. Use an ϵ-greedy exploration strategy with $\epsilon_t = \frac{N_0}{N_0 + N(s_t)}$, where we define $N_0 = 100$. Vary N_0 as needed. Varying $0 \leq N_0 < \inf$ will determine the amount of exploration the algorithm performs where the lower N_0 the less exploration and vice versa.

Show your results by plotting the optimal state value function: $v^*(s) = \max_a q^*(s, a)$ and the optimal policy $\pi^*(s)$. Create plots for these similar to Sutton and Barto, Figure 5.2 in the 2018 edition (5.5 in the original edition) - sample code provided for the plots. Your results SHOULD be very similar to the plots in that text (although you will show your results with the player sum ranging from 2 to 21). For these plots include the following (note - code from the previous section of this assignment for state value function plotting and below for policy plotting are provided to help you to accomplish these):

1. When you have a useable ace, plot the value function with the dealer's card on the x-axis, and the player's sum on the y-axis, and use the colormap and `imshow` to plot the value function that corresponds with those states. Plot the state value corresponding to each state under the policy described above. The domain of your x and y axes should include all possible states (2 to 21 for the player sum, and 1 to 10 for the dealer's visible card).

2. Repeat (1) for the same states but without a usable ace.

3. Plot the optimal policy $\pi^*(s)$ for the states with a usable ace (this plot can be an imshow plot with binary values - sample code provided).

4. Plot the optimal policy $\pi^*(s)$ for the states without a usable ace (this plot can be an imshow plot with binary values - sample code provided).

(b) Plot the cumulative average return per episode vs the number of episodes (your x-axis should be log-scaled to clearly see the trend). What is the average return your control strategy was able to achieve? You'll know your method is working if you see a steady rise in your average returns over time.

Note on convergence: convergence of this algorithm is extremely slow. You may need to let this run a few million episodes before the policy starts to converge. You're not expected to get EXACTLY the optimal policy, but it should be visibly close.

Note on sample code: the code provided for these questions is meant to be a helpful starting point - you are not required to fill it out exactly or use all components, but it is meant to help you as you begin thinking about this problem.

ANSWER

To make this slightly easier I'm importing gym environment

In [115]:
```
import sys
import random
import gym
import numpy as np
from collections import defaultdict
#from plot_utils import plot_blackjack_values, plot_policy
```

In [125]:
```
# start a new blackjack game from gym
env = gym.make('Blackjack-v1')

# number of games to play
episodes = 5000
```

```
# for discounting rewards,
gamma = 1.
```

In [126]:
```
def get_epsilon(N_state_count, N_zero=100):
    # pick a random action
    # initialising N_zero to 100
    return N_zero / (N_zero + N_state_count)
```

In [127]:
```
def get_action(Q, state, state_count, action_size):
    # value function (Q) and state, define action
    # #visit all possible states and actions.

    random_action = random.randint(0, action_size - 1)

    best_action = np.argmax(Q[state])

    epsilon = get_epsilon(state_count)

    return np.random.choice([best_action, random_action], p=[1. - epsilon, epsilon])
```

In [128]:
```
def evaluate_policy(Q, episodes=10000):
    # Helper function to evaluate how good the policy is.

    wins = 0
    for _ in range(episodes):
        state = env.reset()

        done = False
        while not done:
            action = np.argmax(Q[state])

            state, reward, done, _ = env.step(action=action)

        if reward > 0:
            wins += 1

    return wins / episodes
```

In [129]:
```
def monte_carlo(gamma=1., episodes=5000, evaluate=False):

    # define value function
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # Keep track of how many times any given state has been visited
    state_count = defaultdict(float)

    # Keep track of how often a given action has been taken when in that state
    state_action_count = defaultdict(float)

    # Keeping track of policy evaluations
    evaluations = []

    for i in range(episodes):
        # Do this every 1000 games
        if evaluate and i % 1000 == 0:
            evaluations.append(evaluate_policy(Q))

        # Update value function by keeping track of what states happen and what actions
        # are taken
        episode = []

        # Start a game
        state = env.reset()
        done = False

        # Keep playing until game done
        while not done:
            # Pick an action
```

```
            state_count[state] += 1
            action = get_action(Q, state, state_count[state], env.action_space.n)

            # Get new state, reward and if done
            new_state, reward, done, _ = env.step(action=action)

            # save what happened
            episode.append((state, action, reward))

            state = new_state

        # Update value function
        G = 0

        # Start at the end of the game and work
        # backwards through the states to decide how good it was to be in a state
        for s, a, r in reversed(episode):
            new_s_a_count = state_action_count[(s, a)] + 1


            # Update the mean rewards using incremental averaging
            G = r + gamma * G
            state_action_count[(s, a)] = new_s_a_count
            Q[s][a] = Q[s][a] + (G - Q[s][a]) / new_s_a_count

    return Q, evaluations
```

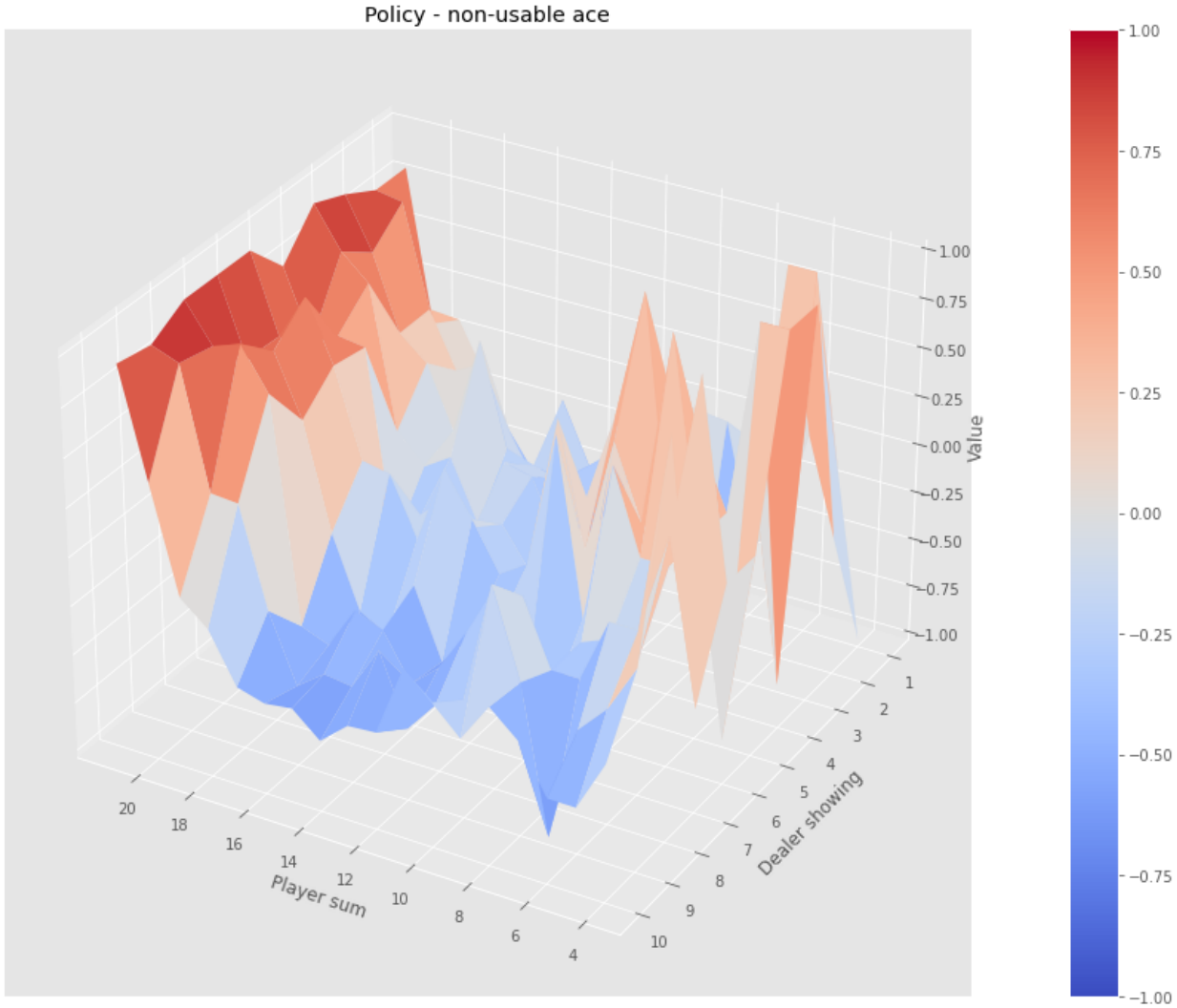
In [131]:
```
# Play 5000 episodes
Q_mc, evaluations = monte_carlo(episodes=5000, evaluate=True)
```

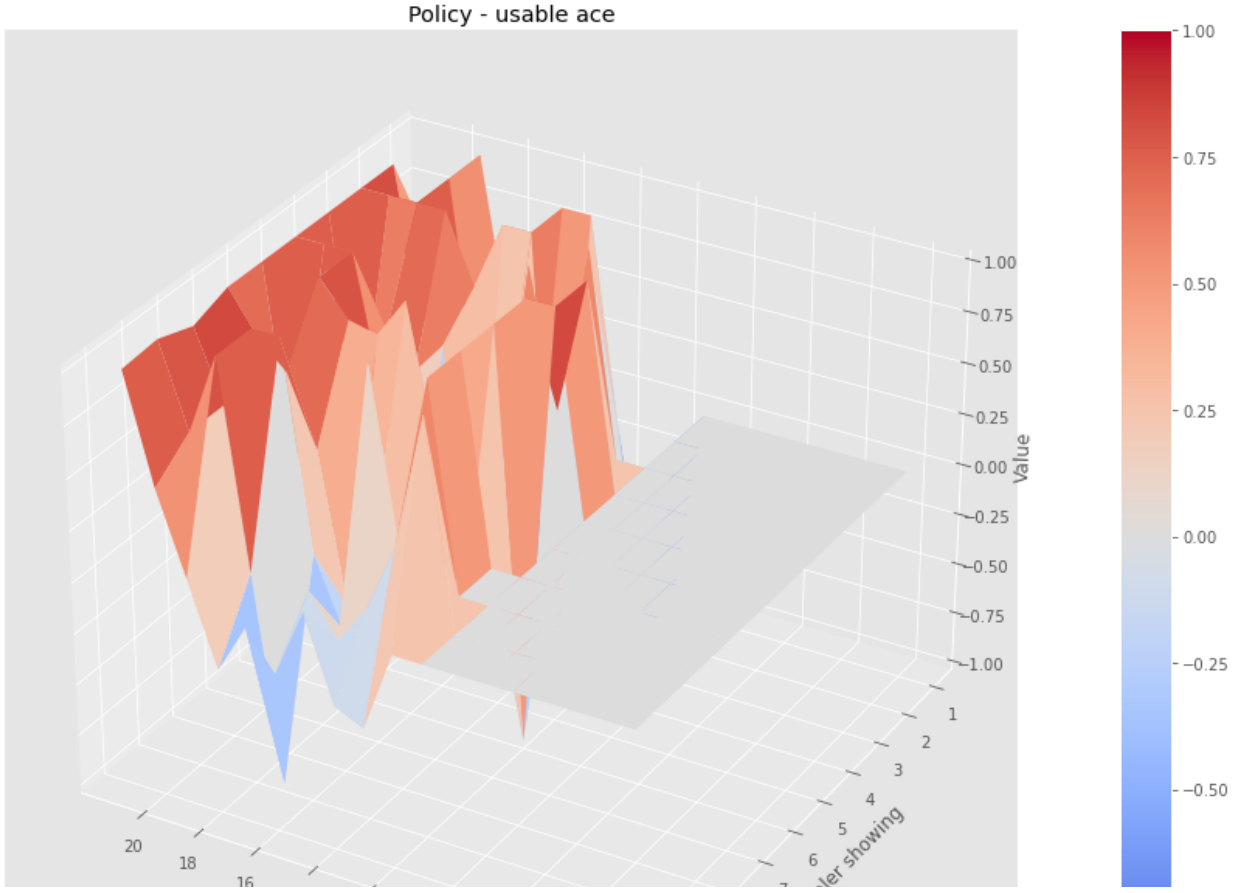
In [140]:
```
# Define a plot value function for this task

import matplotlib
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter, MaxNLocator
%matplotlib inline

matplotlib.style.use('ggplot')

def plot_value_function(Q, title="Policy"):
    V = defaultdict(float)

    for state, action_rewards in Q.items():
        r1, r2 = action_rewards
        action_value = np.max([r1, r2])
        V[state] = action_value

    min_x = min(k[0] for k in V.keys())
    max_x = max(k[0] for k in V.keys())
    min_y = min(k[1] for k in V.keys())
    max_y = max(k[1] for k in V.keys())

    x_range = np.arange(min_x, max_x + 1)
    y_range = np.arange(min_y, max_y + 1)
    X, Y = np.meshgrid(x_range, y_range)

    # Find value for all (x, y) coordinates
    Z_noace = np.apply_along_axis(lambda _: V[(_[0], _[1], False)], 2, np.dstack([X, Y]))
    Z_ace = np.apply_along_axis(lambda _: V[(_[0], _[1], True)], 2, np.dstack([X, Y]))

    def plot_surface(X, Y, Z, title):
        fig = plt.figure(figsize=(24, 12))
        ax = fig.add_subplot(111, projection='3d')
        surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                               cmap=matplotlib.cm.coolwarm, vmin=-1.0, vmax=1.0)
        ax.set_xlabel('Player sum')
        ax.set_ylabel('Dealer showing')
        ax.set_zlabel('Value')
```

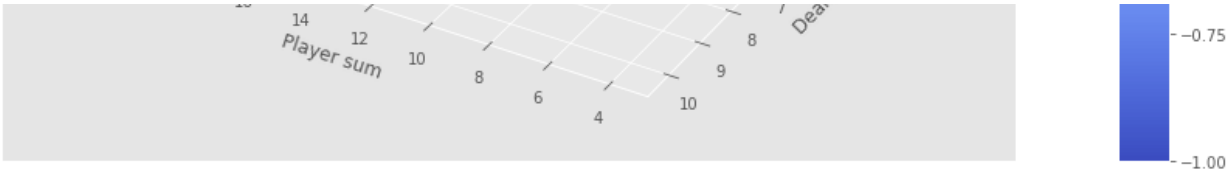
```
        ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        ax.yaxis.set_major_locator(MaxNLocator(integer=True))
        ax.set_title(title)
        ax.view_init(ax.elev, 120)
        fig.colorbar(surf)
        plt.show()

    plot_surface(X, Y, Z_noace, "Policy - non-usable ace")
    plot_surface(X, Y, Z_ace, "Policy - usable ace")
```

In [141]:
```
# Plot the optimal policy $\pi^*(s)$ for the states with a usable ace (this plot can be an
imshow plot with binary values - sample code provided)
plot_value_function(Q_mc)
```

Out[141]:
```
C:\Users\fs113\AppData\Local\Temp/ipykernel_17552/1348114531.py:42:
MatplotlibDeprecationWarning: Auto-removal of grids by pcolor() and pcolormesh() is deprecated
since 3.5 and will be removed two minor releases later; please call grid(False) first.
  fig.colorbar(surf)
```


Policy - non-usable ace
Player sum
Dealer showing
Value

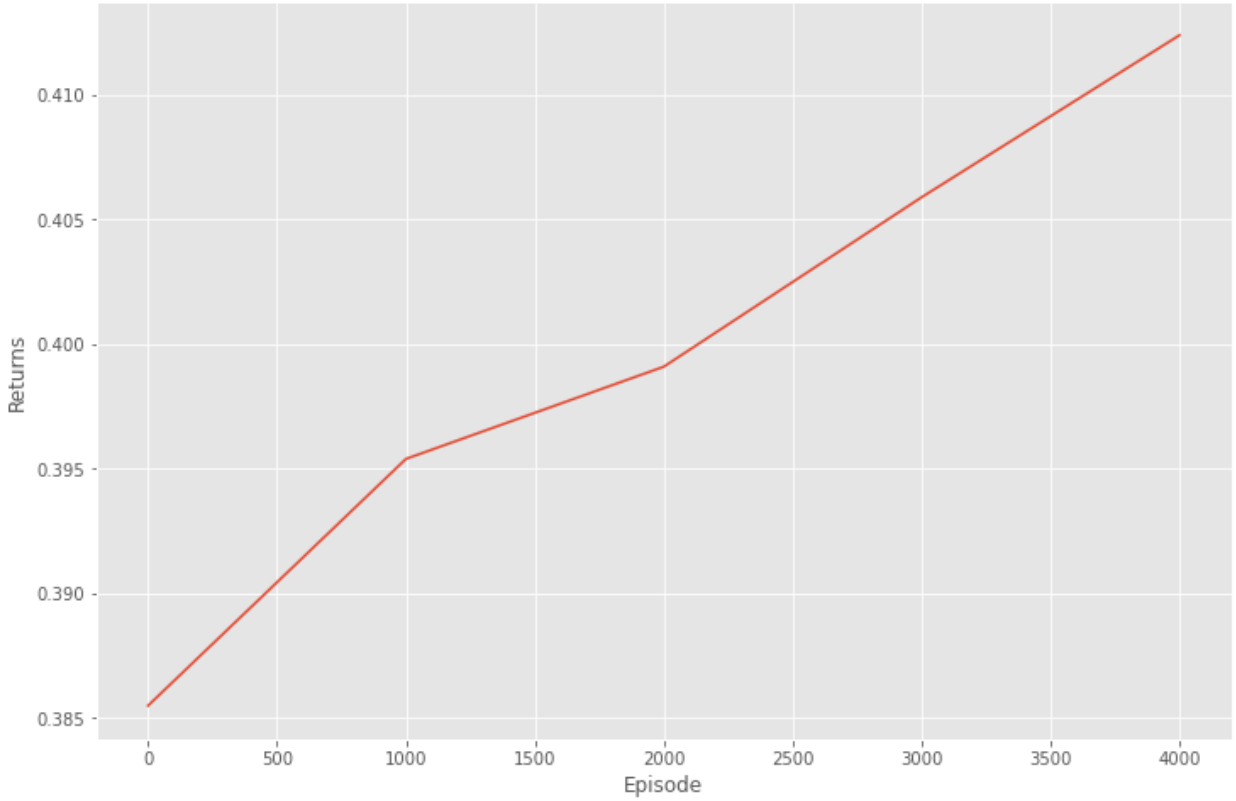

Policy - usable ace
Player sum
Dealer showing
Value



(b) Plot the cumulative average return per episode vs the number of episodes (your x-axis should be log-scaled to clearly see the trend). What is the average return your control strategy was able to achieve? You'll know your method is working if you see a steady rise in your average returns over time.

In [144]:
```
fig = plt.figure(figsize=(12, 8))
plt.plot([i * 1000 for i in range(len(evaluations))], evaluations)
plt.xlabel('Episode')
plt.ylabel('Returns')
```

Out[144]: Text(0, 0.5, 'Returns')



4

[5 points] Discuss your findings

Compare the performance of your human control policy, in question 1, the naive policy from question 2, and the optimal control policy in question 3.

(a) Which performs best? What was different about the policies developed for each and how may that have contributed to their comparative advantages?

The optimal policy performs best with a steady raise in returns. I clearly did not play enough episodes to get to this point of returns and the naive policy varies too much in its returns.

(b) Could you have created a better policy if you knew the full Markov Decision Process for this environment? Why or why not? *(assume the policy estimated from your MC control algorithm had **fully converged**)*

I believe creating a better policy with knowledge of the full Markov Decision Process for this environment would not be possible, since this is a partly random process.