Assignment 5 - Reinforcement Learning

Frank Xu

Netid: *hx44*

https://github.com/Frank-Xu-Huaze/machine-learning-course/blob/master/5/Assignment5.ipynb (https://github.com/Frank-Xu-Huaze/machine-learning-course/blob/master/5/Assignment5.ipynb)

Blackjack

Your goal is to develop a reinforcement learning technique to learn the optimal policy for winning at blackjack. 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, and we're refer to it as 'usable' at 11 (indicating that it could be used as a '1' if need be. This game is placed with a deck of cards 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, or action '1') until they decide to stop (stay, action '0') or exceed 21 (bust, the game ends and player loses).
- 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.

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 simple 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 (http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html) at University College London

1

[10 points] Human reinforcement learning

Using the code detailed below, play 50 hands of blackjack, and record your overall average reward. This will help you get accustomed with how the game works, the data structures involved with representing states, and what strategies are most effective.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from collections import defaultdict
import sys
import time

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
```

```
    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)

# Take an action: Hit = 1 or stay = 0. Here's a hit:
s1 = game.step(0)
s1

#s2 = game.step(1)
#print(s2)

#s3 = game.step(1)
#print(s3)
```

```
[(19, 1, False)]
```

Out[2]:

```
((19, 1, False), 1.0, True)
```

In [3]:

```
# Using the policy of hitting soft when <18 and hitting hard when <15
game = Blackjack()
reward = []
for i in range(50):
    s = [game.deal()]
    for r in range(5):
        if (s[0][2] == True and s[0][0] <= 17) or (s[0][2] == False and s[0][0]
<= 14):
            s = game.step(1)
        else:
            s = game.step(0)
        if s[2] == True:
            break
    reward.append(s[1])
np.mean(reward)
```

Out[3]:

```
0.02
```

ANSWER

2

[40 points] Perform Monte Carlo Policy Evaluation

Thinking that you want to make your millions playing 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:

1. When you have a useable ace, plot the state space with the dealer's card on the x-axis, and the player's sum on the y-axis, and use the 'RdBu' matplotlib colormap and `imshow` to plot the value of 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). Do this for for 10,000 episodes.
2. Repeat (1) for the states without a usable ace.
3. Repeat (1) for the case of 500,000 episodes.
4. Relwat (2) for the case of 500,000 episodes.

(b) Show a plot of the overall average reward per episode vs the number of episodes. For both the 10,000 episode case and the 500,000 episode case, record the overall average reward for this policy and report that value.

ANSWER

In [4]:

```
# (a)
def mat(policy, episodes):
    # Initializing dictionaries
    returns_sum, returns_count, V = dict(), dict(), dict()

    for i in range(1, episodes + 1):
        # An episode is an array of (state, action, reward) tuples
        episode = []
        game = Blackjack()
        state = game.deal()
        for t in range(10):
            action = policy(state)
            next_state, reward, done = game.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        states_in_episode = set([tuple(x[0]) for x in episode])
        for state in states_in_episode:
```

```
            # Find the first occurance of the state in the episode

            first_occur_idx = next(i for i,x in enumerate(episode) if x[0] == st
ate)
            # Sum up all rewards since the first occurance
            episum = sum([x[2] for x in episode[first_occur_idx:]])
            # Calculate average return for this state over all sampled episodes
            if state not in returns_sum:
                returns_sum[state] = episum
                returns_count[state] = 1
            else:
                returns_sum[state] += episum
                returns_count[state] += 1
            V[state] = returns_sum[state] / returns_count[state]
    return V, returns_sum, returns_count

def plotting(dictionary, usable):
    res = np.zeros((20, 10))
    for state in dictionary:
        if usable == state[2]:
            res[state[0]-2, state[1]-1] = dictionary[state]
    plt.imshow(res, cmap = 'RdBu')
    plt.gca().set_xticks(np.arange(0, 10, 1))
    plt.gca().set_yticks(np.arange(0, 20, 1))
    plt.gca().set_xticks(np.arange(-.5, 10, 1), minor = True)
    plt.gca().set_yticks(np.arange(-.5, 20, 1), minor = True)
    plt.gca().set_xticklabels(np.arange(1, 11, 1))
    plt.gca().set_yticklabels(np.arange(2, 22, 1))
    plt.gca().tick_params(axis='both', which='major', length=0)
    plt.ylabel("Player's Sum"); plt.xlabel("Dealer's Card")
    plt.colorbar()

def aggressive(state):
    score, dealer_score, usable_ace = state
    return 0 if score >= 20 else 1
```

In [5]:

```
# (a)
fig = plt.figure(figsize=(15, 6))
plt.subplot(121)
plotting(mat(aggressive, 10000)[0], True)
fig.text(.35, .01, "Figure 1: Player's sum vs dealer's card with usable Aces in
10000 episodes", ha='center')
plt.subplot(122)
plotting(mat(aggressive, 10000)[0], False)
fig.text(.77, .01, "Figure 2: Player's sum vs dealer's card with no usable Aces
in 10000 episodes", ha='center')
plt.show()

fig = plt.figure(figsize=(15, 6))
plt.subplot(121)
plotting(mat(aggressive, 500000)[0], True)
fig.text(.35, .01, "Figure 3: Player's sum vs dealer's card with usable Aces in
500000 episodes", ha='center')
plt.subplot(122)
plotting(mat(aggressive, 500000)[0], False)
fig.text(.77, .01, "Figure 4: Player's sum vs dealer's card with no usable Aces
in 500000 episodes", ha='center')
plt.show()
```

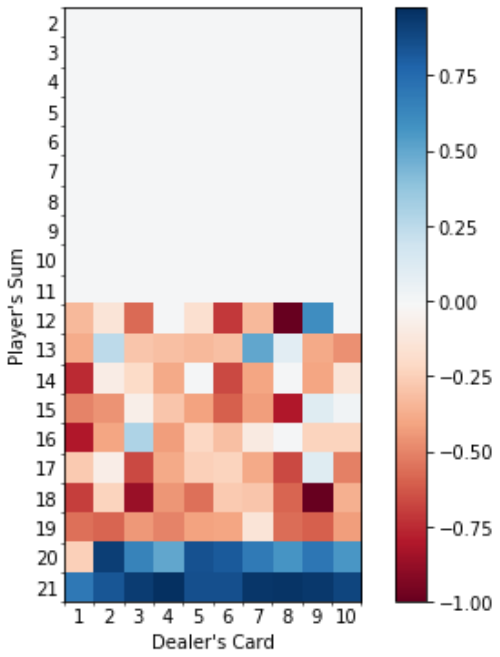

Figure 1: Player's sum vs dealer's card with usable Aces in 10000 episodes

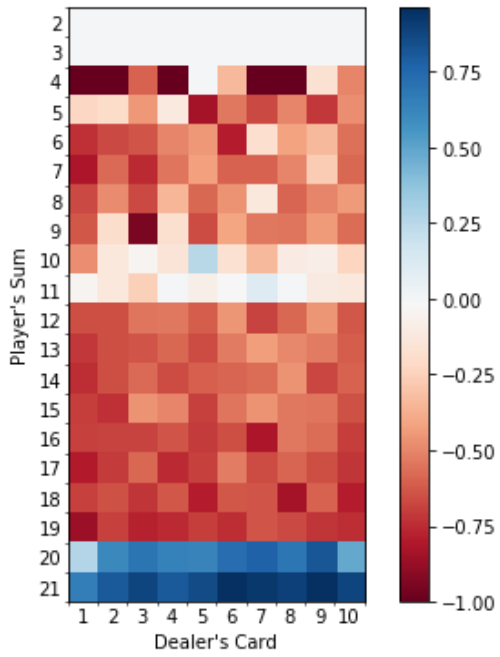

Figure 2: Player's sum vs dealer's card with no usable Aces in 10000 episodes

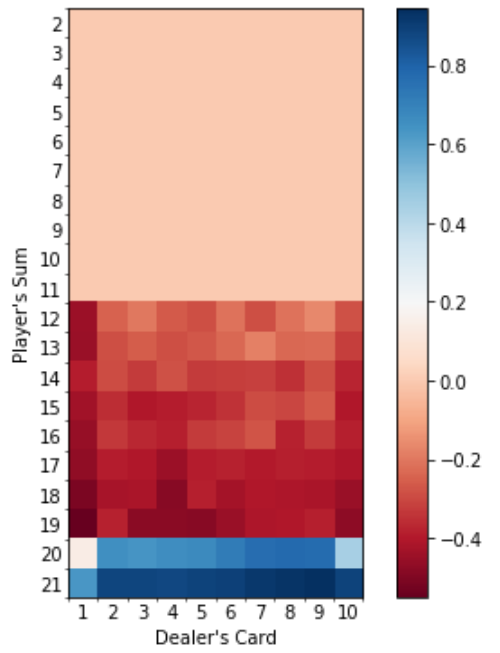

Figure 3: Player's sum vs dealer's card with usable Aces in 500000 episodes

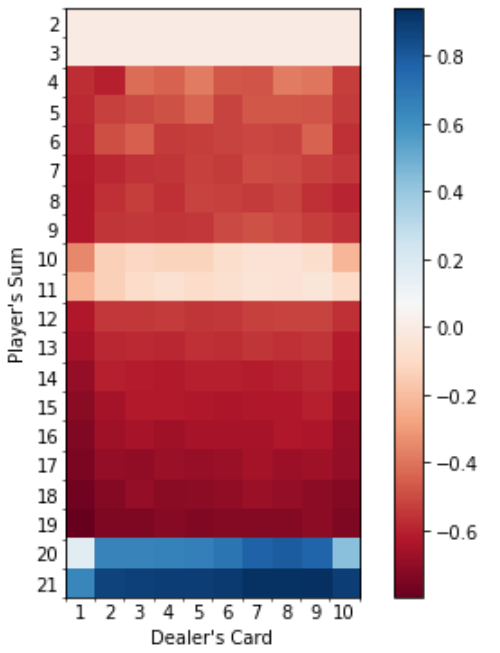

Figure 4: Player's sum vs dealer's card with no usable Aces in 500000 episodes

In [6]:

```
# (b)
def mat2(policy, episodes):
    run_ave, re_count, re_sum = [], [], []
    re_count = 0; re_sum = 0
    for i in range(1, episodes + 1):
    # An episode is an array of (state, action, reward) tuples
        episode = []
        game = Blackjack()
        state = game.deal()
        for t in range(10):
            action = policy(state)
            next_state, reward, done = game.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state
        re_count += len(episode)
        re_sum += np.sum([x[-1] for x in episode])
        run_ave.append(re_sum / re_count)
    return run_ave

mat_10000 = mat2(aggressive, 10000)
mat_500000 = mat2(aggressive, 500000)
print('The overall average reward of 10000 episodes is {}.'.format(mat_10000[-1]
))
print('The overall average reward of 500000 episodes is {}.'.format(mat_500000[-
1]))
```

```
The overall average reward of 10000 episodes is -0.19581548049635525
.
The overall average reward of 500000 episodes is -0.1974566000071293
6.
```

In [7]:

```
fig = plt.figure(figsize=(15, 6))

plt.subplot(121)
plt.plot(np.arange(1,10001), mat_10000, label = 'Running average of average rewa
rds per episode: {:0.4f}'.format(mat_10000[-1]))
plt.legend(); plt.xlabel('Iterations'); plt.ylabel('Average Rewards')
fig.text(.3, .01, "Figure 5: Running average of average rewards in 10000 episode
s", ha='center')

plt.subplot(122)
plt.plot(np.arange(1,500001), mat_500000, label = 'Running average of average re
wards per episode: {:0.4f}'.format(mat_500000[-1]))
plt.legend(); plt.xlabel('Iterations'); plt.ylabel('Average Rewards')
fig.text(.72, .01, "Figure 6: Running average of average rewards in 500000 episo
des", ha='center')
plt.show()
```

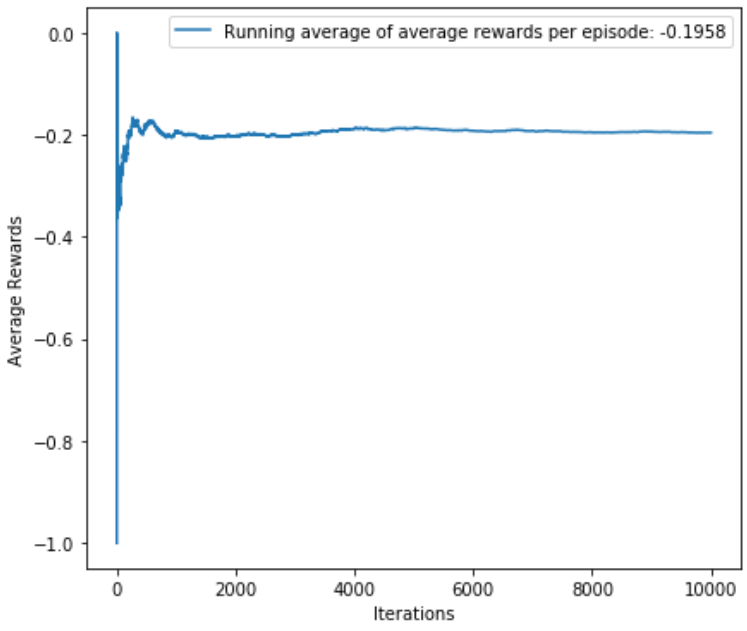

Figure 5: Running average of average rewards in 10000 episodes

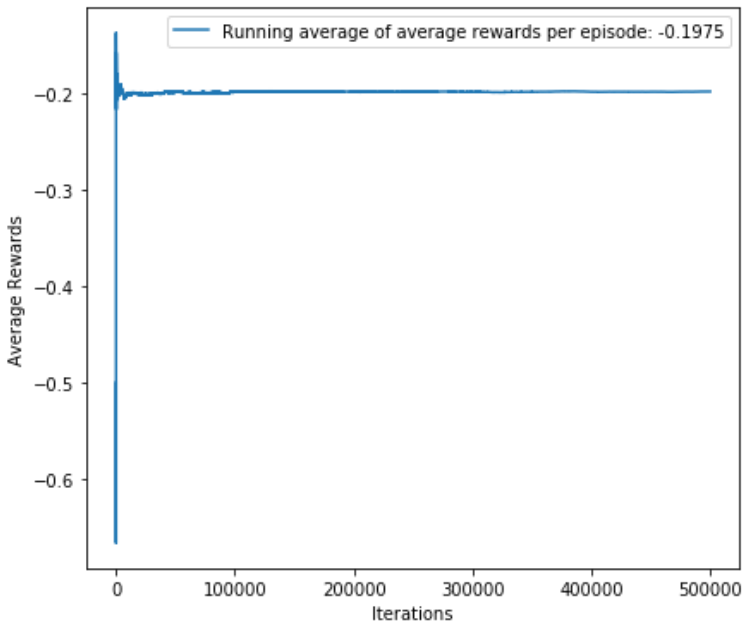

Figure 6: Running average of average rewards in 500000 episodes

3

[40 points] Perform Monte Carlo Control

(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 value function and the state 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.

Show your result by plotting the optimal 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 new draft edition, or 5.5 in the original edition. Your results SHOULD be very similar to the plots in that text. For these plots include:

1. When you have a useable ace, plot the state space with the dealer's card on the x-axis, and the player's sum on the y-axis, and use the 'RdBu' matplotlib colormap and `imshow` to plot the value of 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 states without a usable ace.
3. A plot of the optimal policy $\pi^*(s)$ for the states with a usable ace (this plot could be an imshow plot with binary values).
4. A plot of the optimal policy $\pi^*(s)$ for the states without a usable ace (this plot could be an imshow plot with binary values).

(b) Show a plot of the overall average reward per episode vs the number of episodes. What is the average reward your control strategy was able to achieve?

Note: 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.

ANSWER

In [8]:

```
# (a)
def prob(Q_s, eps, n):
    policy_s = np.ones(n) * eps / n
    policy_s[np.argmax(Q_s)] = 1 - eps + (eps / n)
```

```
    return policy_s

def generate_episode(Q, eps, n):
    episode = []
    game = Blackjack()
    state = game.deal()
    while True:
        action = np.random.choice(np.arange(n), p = prob(Q[state], eps, n)) if s
tate in Q else np.random.randint(2)
        next_state, reward, done = game.step(action)
        episode.append((state, action, reward))
        state = next_state
        if done:
            break
    return episode

def update(episode, Q, alpha):
    states, actions, rewards = zip(*episode)
    for i, state in enumerate(states):
        Q_past = Q[state][actions[i]]
        Q[state][actions[i]] = Q_past + alpha * (sum(rewards[i:]) - Q_past)
    return Q

def MCC(num_episodes, alpha, eps=1.0, eps_decay=.99995, eps_min=0.01):
    n = 2
    Q = defaultdict(lambda: np.zeros(n))
    t = time.time()
    for i in range(1, num_episodes+1):
        eps = max(eps*eps_decay, eps_min)
        episode = generate_episode(Q, eps, n)
        Q = update(episode, Q, alpha)
        if i % 10000 == 0:
            tr = (time.time() - t)*(num_episodes-i)/10000
            t = time.time()
            m = tr//60
            print("\rEpisode {}/{}. Expected time left: {:0.0f}m {:0.0f}s".forma
t(i, num_episodes, m, tr-60*m), end="")
            sys.stdout.flush()
    policy = dict((s, np.argmax(v)) for s, v in Q.items())
    return policy, Q

def plotting2(policy, usable, title = 'please enter title'):
    x_range = np.arange(1, 11)
    y_range = np.arange(21, 10, -1)
    def uti(x, y, usable):
        if (y,x,usable) in p:
            return p[y,x,usable]
        else:
            return 1
    Z = np.array([[uti(x,y,usable) for x in x_range] for y in y_range])
    plt.imshow(Z, cmap = plt.get_cmap('Pastel1', 2))
    plt.gca().set_xticks(np.arange(0, 10, 1))
    plt.gca().set_yticks(np.arange(0, 11, 1))
    plt.gca().set_xticks(np.arange(-.51, 10, 1), minor = True)
```

```
    plt.gca().set_yticks(np.arange(-.52, 11, 1), minor = True)

    plt.gca().set_xticklabels(np.arange(1, 11, 1))
    plt.gca().set_yticklabels(np.arange(11, 22, 1))
    plt.gca().tick_params(axis='both', which='major', length=0)
    plt.ylabel("Player's Sum"); plt.xlabel("Dealer's Card")
    plt.grid(color='w', which = 'minor')
    cbar = plt.colorbar(ticks=[0,1])
    cbar.ax.set_yticklabels(['0 (STAY)','1 (HIT)'])
    plt.title(title)

def optimal(state):
    return p[state] if state in p else 1
```

In [15]:

```
p, q = MCC(5000000, 0.015)
ave = mat2(optimal, 500000)
```

```
Episode 5000000/5000000. Expected time left: 0m 0s
```

In [16]:

```
# (a) & (b)
fig = plt.figure(figsize=(15, 6))
plt.subplot(121)
plotting(mat(optimal, 500000)[0], True)
fig.text(.35, .01, "Figure 7: States under current policy with usable Aces in 50
0000 episodes", ha='center')
plt.subplot(122)
plotting(mat(optimal, 500000)[0], False)
fig.text(.77, .01, "Figure 8: States under current policy with no usable Aces in
500000 episodes", ha='center')
plt.show()

fig = plt.figure(figsize=(15, 6))
plt.subplot(121)
plotting2(p, True, 'With Usable Ace')
plt.subplot(122)
plotting2(p, False, 'No Usable Ace')
fig.text(.5, .01, "Figure 9: The optimal policy with/without a usable Ace under
5 million iterations", ha='center')
plt.show()

fig = plt.figure(figsize=(8, 5))
plt.plot(np.arange(1,500001), ave, label = 'Running average of average rewards p
er episode: {:0.4f}'.format(ave[-1]))
plt.legend(); plt.xlabel('Iterations'); plt.ylabel('Average Rewards')
fig.text(.5, .01, "Figure 10: Optimal running average of average rewards in 5000
00 episodes", ha='center')
plt.show()
```

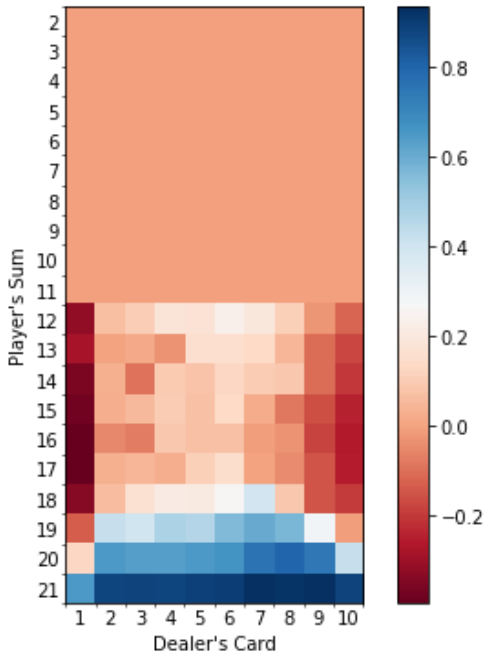

Figure 7: States under current policy with usable Aces in 500000 episodes

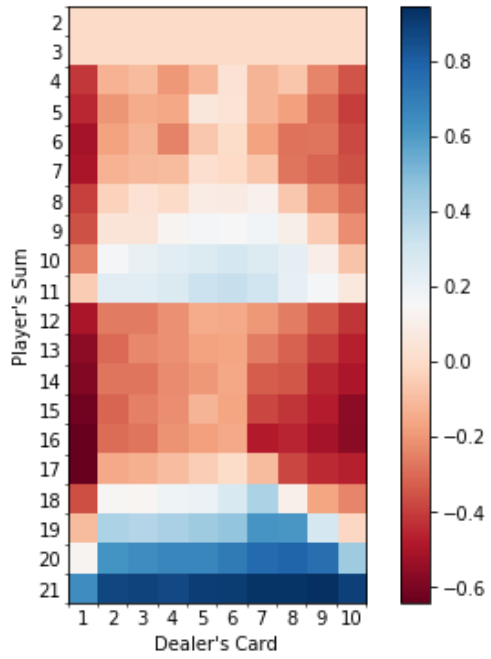

Figure 8: States under current policy with no usable Aces in 500000 episodes

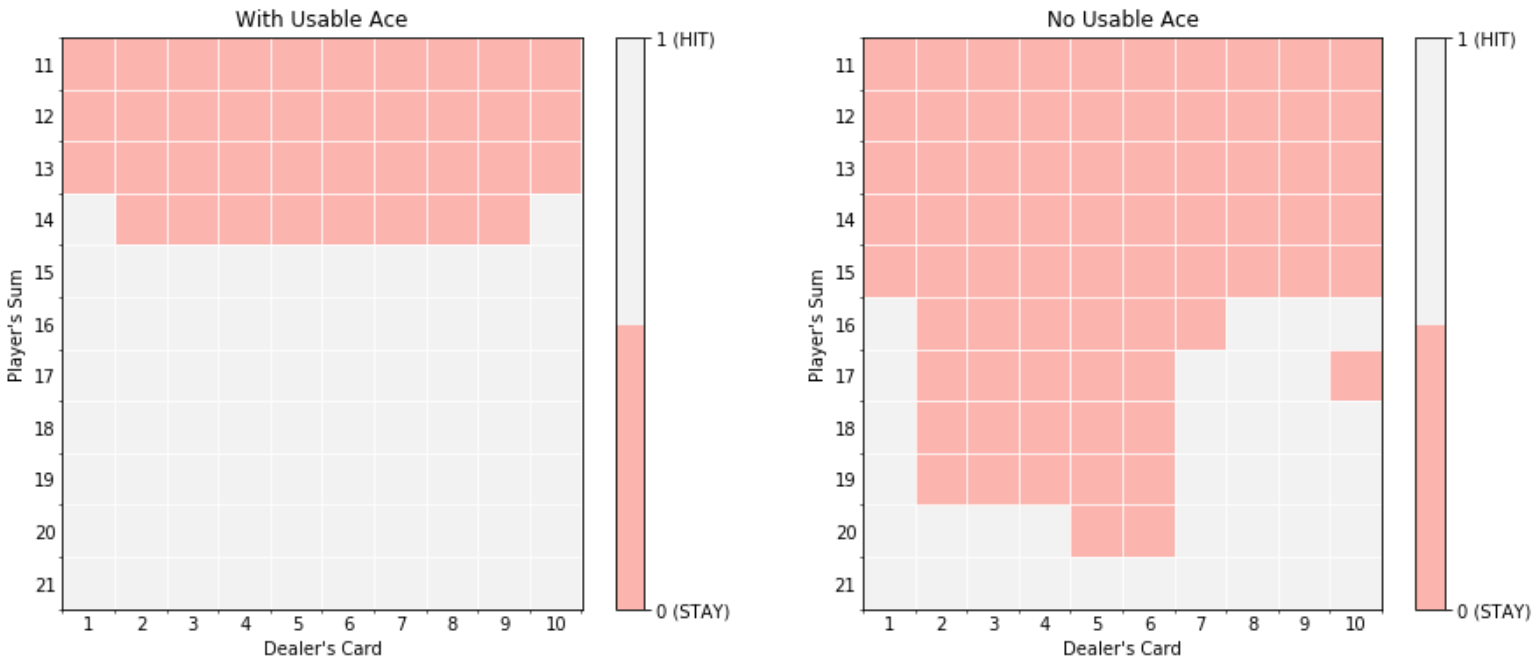

Figure 9: The optimal policy with/without a usable Ace under 5 million iterations

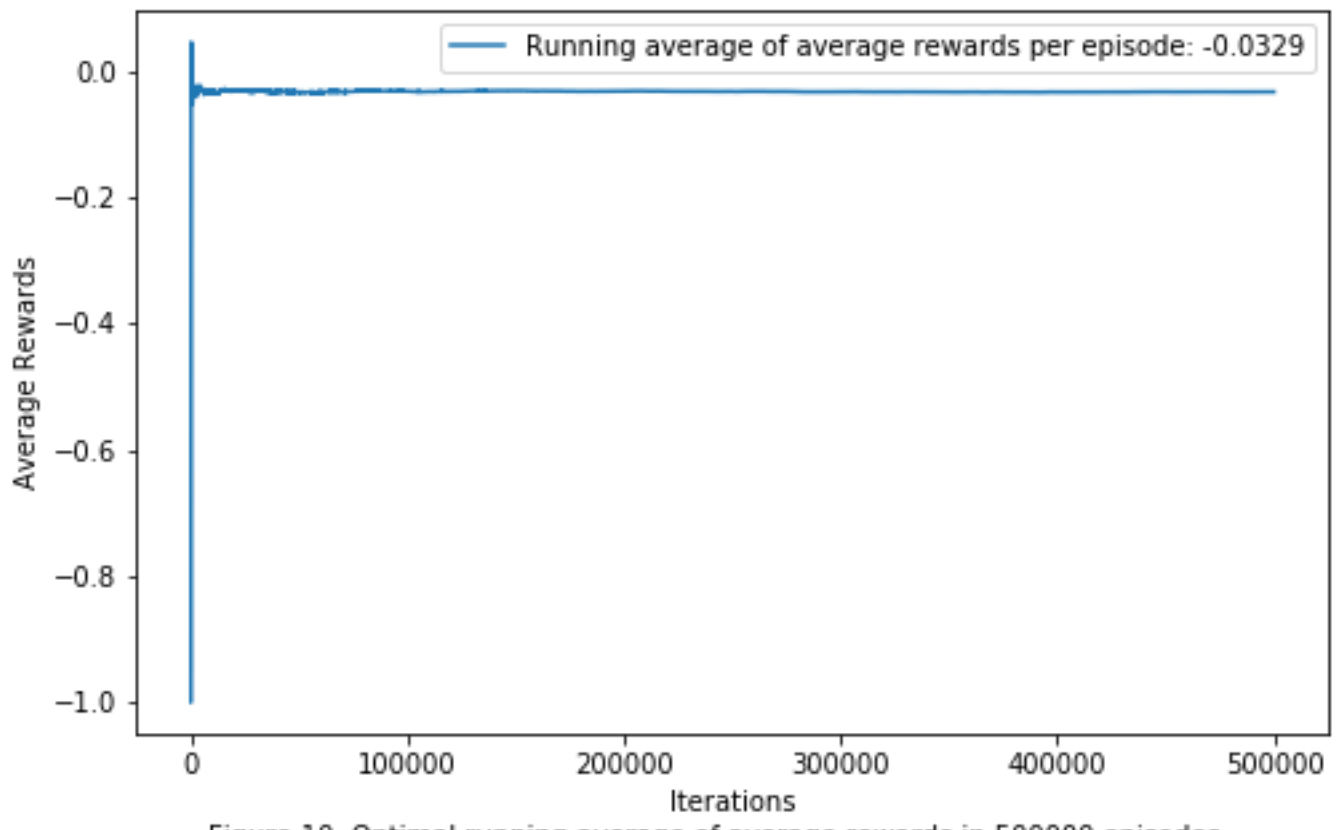

Figure 10: Optimal running average of average rewards in 500000 episodes

The average of overall average reward is -0.0329. Compared with -0.195 (aggressive policy), the optimal policy has higher reward. Still, the average reward is negative, indicating that a long time play in BlackJack will more likely to result in losing money.

4

[10 points] Discuss your findings

Compare the performance of your human control policy, the naive policy from question 2, and the optimal control policy in question 3. **(a)** Which performs best? Why is this the case? **(b)** Could you have created a better policy if you knew the full Markov Decision Process for this environment? Why or why not?

ANSWER

Compared with the human control policy (average reward of around -0.06 in a long run) and the aggressive policy (average reward of around -0.195 in a long run), the optimal policy (average reward is -0.0329) is clearly the best among three. The reason is because that the Monte Carlo Process simulates the optimal policy, which if run long enough similar to the optimal policy.

No, we can not have a better policy if we knew the full Markov Decision Process for this environment. All the states and optimized actions are found under the Monte Carlo Simulations of the optimal control policy with long enough iterations, so we have the best policy already.