Assignment 5

April 20, 2018

1 Assignment 5 - Reinforcement Learning

1.1 *Yangdi Shen*

Netid: *ys203*

1.1.1 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 at University College London

1.2 1

1.2.1 [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 [2]: 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

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

        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 [3]: 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(1)
        print(s1)

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

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

```
(19, 8, False)
((25, 8, False), -1, True)
```

ANSWER

```
In [304]: game = Blackjack()

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

```
(12, 7, False)
```

```
In [305]: # Take an action: Hit = 1 or stay = 0. Here's a hit:
          s1 = game.step(1)
          print(s1)
```

```
((18, 7, False), 0, False)
```

```
In [306]: s2=game.step(1)
          print(s2)
```

```
((28, 7, False), -1, True)
```

```
In [307]: s3=game.step(1)
          print(s3)
```

```
((38, 7, False), -1, True)
```

```
In [100]: s4=game.step(0)
          print(s4)
```

```
((17, 1, False), -1.0, True)
```

```
In [196]: #record reward
          human_results.append(s1[1])
          print(human_results)
          print(len(human_results))
          print(sum(human_results))

[1.0, 1.0, 1.0, 0.0, 1.0, 1.0, -1.0, 0.0, 1.0, -1.0, 1.0, 1.0, -1, -1.0, -1.0, 1.0, -1, -1, -1
50
6.0
```

```
In [813]: human_finals=human_results
          print("Win:{0}, Draw:{1}, Loss:{2}".format(human_finals.count(1),human_finals.count(
          print("Expected Reward is {}".format(sum(human_finals)/len(human_finals)))

Win:26, Draw:4, Loss:20
Expected Reward is 0.12
```

As shown above, I used a list to store the results of my games. The winning rate is a little over 50%.

1.3 2

1.3.1 [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 this problem, I define "Usable Aces" as the state in which Aces are counted as 11 instead of 1. Therefore, in different stages of one game, the states might be in different classes.

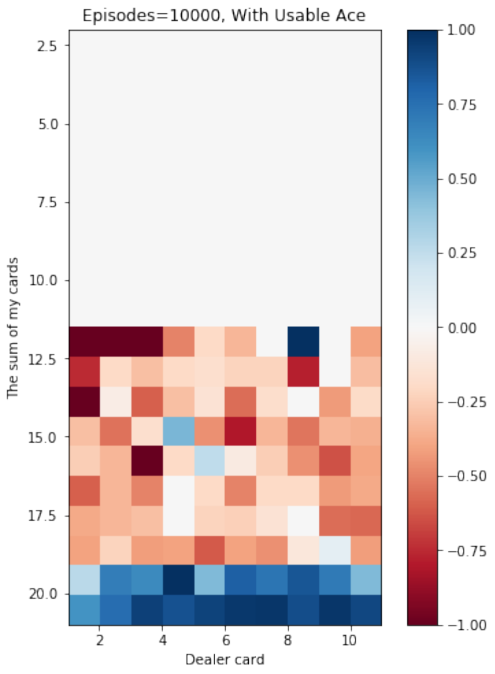
```
In [394]: def simulate(x):
              counts_with=np.zeros((20,10))
              counts_without=np.zeros((20,10))
              scores_with=np.zeros((20,10))
              scores_without=np.zeros((20,10))
```

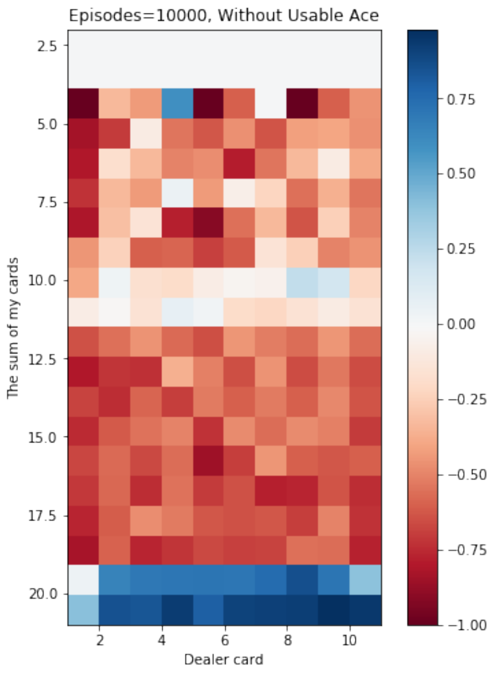
```
        for i in range(x):
            flag=False
            path=[]
            score=0
            game=Blackjack()
            state=game.deal()
            path.append(state)
            if state[0] >= 20:
                state=game.step(0)
                path.append(state)
                flag=True
            else:
                state=game.step(1)
                while flag==False:
                    if state[0][0] ==20 or state[0][0] == 21:
                        state=game.step(0)
                        path.append(state)
                        flag=True
                    else:
                        state=game.step(1)
                        path.append(state)
                        flag=state[2]
        #update values
            _, score, _=path[-1]
            if path[0][2]==False:
                scores_without[path[0][0]-2][path[0][1]-1]+=score
                counts_without[path[0][0]-2][path[0][1]-1]+=1
            if path[0][2]==True:
                scores_with[path[0][0]-2][path[0][1]-1]+=score
                counts_with[path[0][0]-2][path[0][1]-1]+=1
            path_remain=path[1:]
            for j in path_remain[:-1]:
                if j[0][2]==False:
                    counts_without[j[0][0]-2][j[0][1]-1]+=1
                    scores_without[j[0][0]-2][j[0][1]-1]+=score
                if j[0][2]==True:
                    counts_with[j[0][0]-2][j[0][1]-1]+=1
                    scores_with[j[0][0]-2][j[0][1]-1]+=score

        withas=np.nan_to_num(scores_with/counts_with)
        withoutas=np.nan_to_num(scores_without/counts_without)
        return withas, withoutas
```

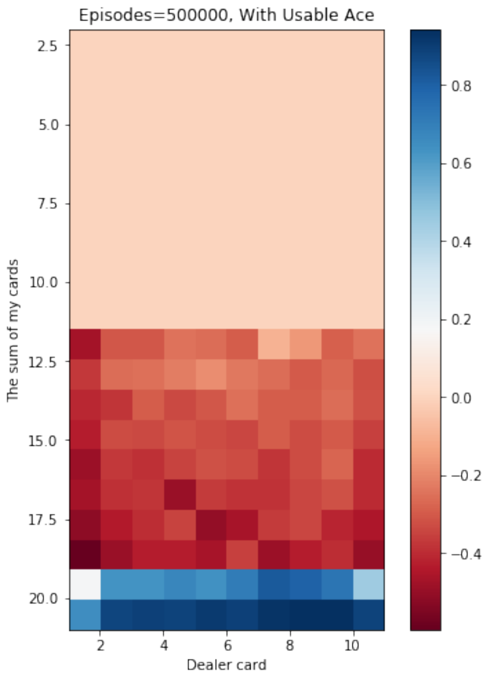
```
In [654]: import matplotlib.pyplot as plt

          withas, withoutas=simulate(10000)

          plt.figure(figsize=(15,8))
```

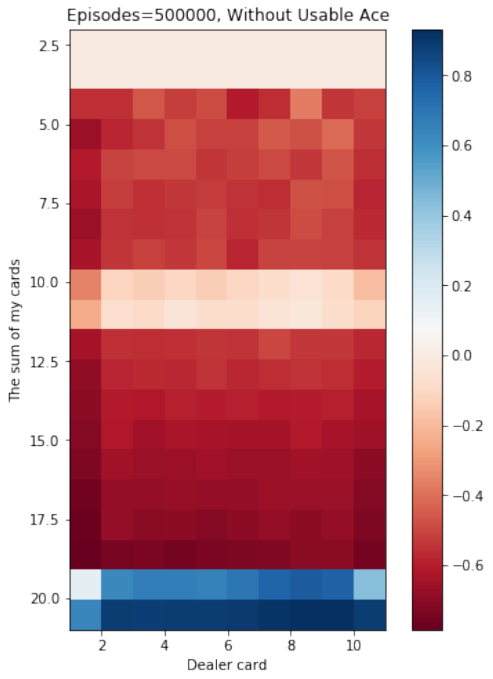
```
        plt.subplot(1,2,1)
        plt.imshow(withas,cmap='RdBu',extent=[1,11,21,2])
        plt.xlabel('Dealer card')
        plt.ylabel('The sum of my cards')
        plt.colorbar()
        plt.title('Episodes=10000, With Usable Ace')


        plt.subplot(1,2,2)
        plt.imshow(withoutas,cmap='RdBu',extent=[1,11,21,2])
        plt.title('Episodes=10000, Without Usable Ace')
        plt.xlabel('Dealer card')
        plt.ylabel('The sum of my cards')
        plt.colorbar()
        plt.show()

        withas, withoutas= simulate(500000)

        plt.figure(figsize=(15,8))
        plt.subplot(1,2,1)
        plt.imshow(withas,cmap='RdBu',extent=[1,11,21,2])
        plt.title('Episodes=500000, With Usable Ace')
        plt.xlabel('Dealer card')
        plt.ylabel('The sum of my cards')
        plt.colorbar()

        plt.subplot(1,2,2)
        plt.imshow(withoutas,cmap='RdBu',extent=[1,11,21,2])
        plt.title('Episodes=500000, Without Usable Ace')
        plt.xlabel('Dealer card')
        plt.ylabel('The sum of my cards')
        plt.colorbar()
        plt.show()
```

```
C:\Users\syd01\Anaconda3\lib\site-packages\ipykernel_launcher.py:47: RuntimeWarning: invalid va
C:\Users\syd01\Anaconda3\lib\site-packages\ipykernel_launcher.py:48: RuntimeWarning: invalid va
```





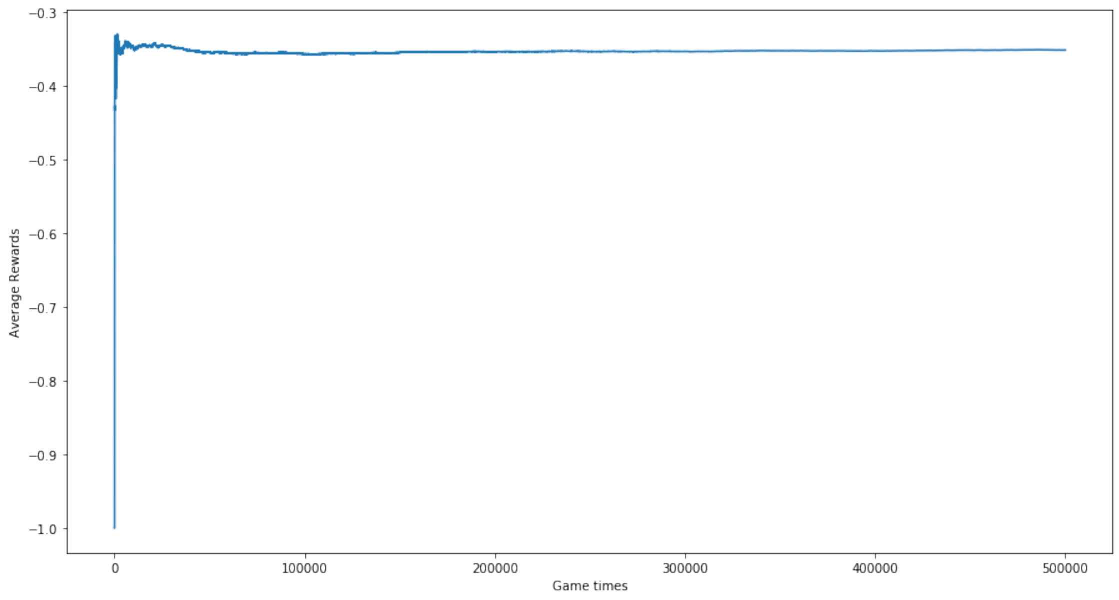




```
In [473]: average=[]

          for i in range(500000):
```

```
        s= game.deal()
        flag=False
        if s[0]>=20:
            average.append(game.step(0)[1])
            flag=True
        s=game.step(1)
        while flag==False:
            if s[0][0]==20 or s[0][0]==21:
                s=game.step(0)
                average.append(s[1])
                flag=True
            else:
                s=game.step(1)
                flag=s[-1]
                if s[-1]==True:
                    average.append(s[1])
```

```
In [482]: averages=np.zeros(500000)
          total=0
          for i in range(len(average)):
              total+=average[i]
              averages[i]=total/(i+1)

          plt.figure(figsize=(15,8))
          plt.plot(averages)
          plt.xlabel('Game times')
          plt.ylabel('Average Rewards')
          plt.show()
```



```
In [499]: print('The overall average reward for this policy: ')
          print('10000 episodes:',averages[9999])
          print('500000 episodes', averages[499999])

The overall average reward for this policy:
10000 episodes: -0.3492
500000 episodes -0.351506
```

1.4 3

1.4.1 [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.

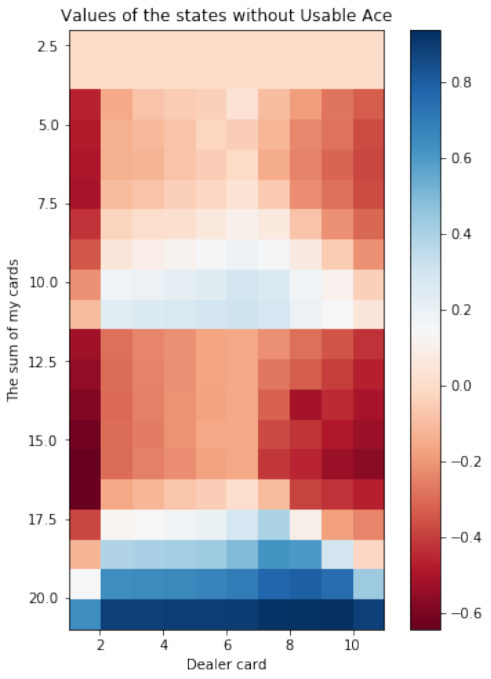
ANSWER

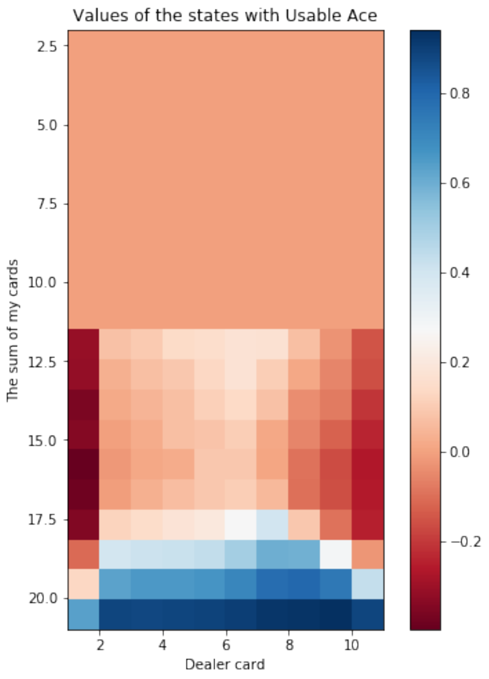
```
In [646]: np.random.random()>0.5

Out[646]: True

In [807]: class MC_Control:


                  def __init__(self):
                      self.n=np.zeros((20,10,2,2))
                      self.q=np.zeros((20,10,2,2))
                      self.NA=100
```

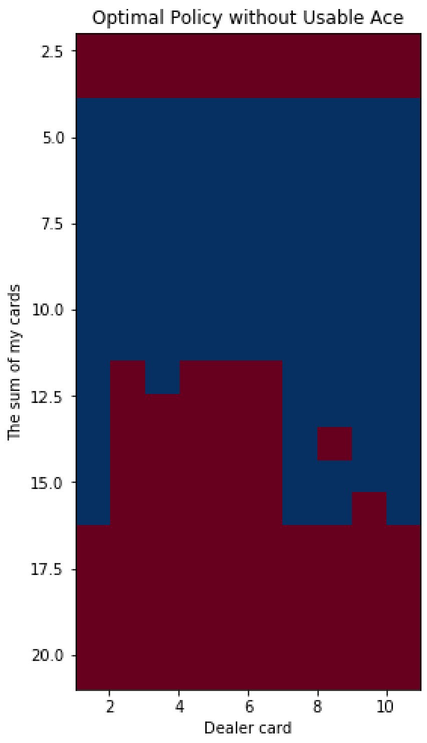
```
def policy_greedy(self, playindex,tableindex,ace):
    nvisit=sum(self.n[playindex-2,tableindex-1,ace*1,:])
    ep=self.NA/(self.NA+nvisit)
    if np.random.random()>=1-ep:
        policy= 0 if np.random.random()<0.5 else 1
    else:
        policy=np.argmax(self.q[playindex-2, tableindex-1, ace*1,:])
    return policy


def run(self, loop):
    for i in range(loop):
        path=[]
        game=Blackjack()
        state=game.deal()

        p=state[0]
        t=state[1]
        ace=state[2]*1

        while True:
            action=int(self.policy_greedy(p,t,ace)*1)


            re=self.n[p-2,t-1,ace,action*1]
            self.n[p-2,t-1,ace,action*1] =re+1
            path.append((p,t,ace,action,self.n[p-2,t-1,ace,action*1]))

            state=game.step(action)
            if state[-1]:
                for i in path:

                    p=i[0]
                    t=i[1]
                    ace=i[2]*1
                    action=int(i[3]*1)
                    n_o=i[4]

                    q_o=self.q[p-2,t-1,ace,action]
                    self.q[p-2,t-1,ace,action]=(q_o*(n_o-1)+state[1])/ n_o

                break


            p=state[0][0]
            t=state[0][1]
```

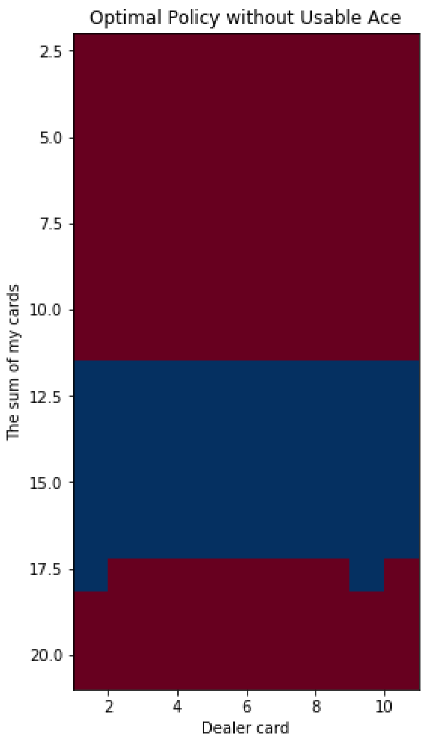
```
                ace=state[0][2]*1

        return self.n, self.q


solver=MC_Control()

N,Q= solver.run(5000000)
opt=np.argmax(Q,axis=3)
```

In [808]:
```
plt.figure(figsize=(15,8))
plt.subplot(1,2,1)
plt.imshow(np.max(Q[:,:,0],axis=2),cmap='RdBu',extent=[1,11,21,2])
plt.xlabel('Dealer card')
plt.ylabel('The sum of my cards')
plt.title('Values of the states without Usable Ace')
plt.colorbar()

plt.subplot(1,2,2)
plt.imshow(np.max(Q[:,:,1],axis=2),cmap='RdBu',extent=[1,11,21,2])
plt.title('Without Usable Ace')
plt.xlabel('Dealer card')
plt.ylabel('The sum of my cards')
plt.title('Values of the states with Usable Ace')
plt.colorbar()
plt.show()

#binary
plt.figure(figsize=(15,8))
plt.subplot(1,2,1)
plt.imshow(opt[:,:,0],cmap='RdBu',extent=[1,11,21,2])
plt.xlabel('Dealer card')
plt.ylabel('The sum of my cards')
plt.title('Optimal Policy without Usable Ace')


plt.subplot(1,2,2)
plt.imshow(opt[:,:,1],cmap='RdBu',extent=[1,11,21,2])
plt.title('Optimal Policy without Usable Ace')
plt.xlabel('Dealer card')
plt.ylabel('The sum of my cards')
plt.show()
```



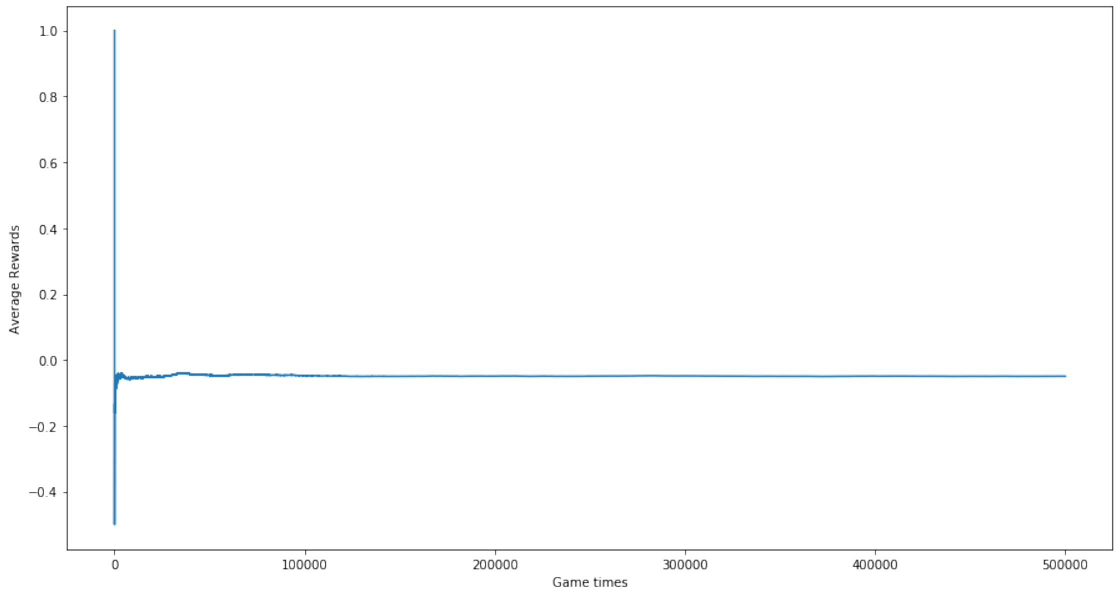






```
In [812]: average=[]
```

```
for i in range(500000):
    state= game.deal()
    p=state[0]
    t=state[1]
    ace=state[2]*1
    while True:
        action=opt[p-2,t-1,ace]
        state=game.step(action)
        if state[-1]:
            average.append(state[-2])
            break
        p=state[0][0]
        t=state[0][1]
        ace=state[0][2]*1

averages=np.zeros(500000)
total=0
for i in range(500000):
    total+=average[i]
    averages[i]=(total/(i+1))

plt.figure(figsize=(15,8))
plt.plot(averages)
plt.xlabel('Game times')
plt.ylabel('Average Rewards')
plt.show()

print('Expected reward is {}'.format(averages[-1]))
```



```
Expected reward is -0.049674
```

1.5 4

1.5.1 [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

a)

My human learning has the best performance (the only positive expected rewards). Naive policy has the worst performance. Of course, I only played for 50 times, but computer simulation could play millions of times. Therefore, the results are not robust.

The control policy is better than naive policy because it could improve policy based on former tests. It could get close to true optimal strategy by trying more times. Except the number of play times, another reason why human learning results are better than control policy could be that blackjack is not a very complicated game. With some statistical knowledge, people could have a good policy on that.

b)

If we have full knowledge about the Markov Decision Process, we can develop a better policy. In the problem 3, the policy was updated based on a estimated envrionment. Knowing full process could help us with iterations.