Assignment 5 - Reinforcement Learning

Iuliia Oblasova Netid: *io26*

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 [21]:
```
import numpy as np

class Blackjack():
    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.d
ealer))
        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()

    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)
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()
```

In [140]:
```
import sys

def default(state):
    playerSum, _, _ = state
    if playerSum < 20: return 1;
    else: return 0;

class Event():
    def __init__(self, N=100):
        self.N = N
        self.Rs = {}
        self.Ns = {}
        self.Qs = {}
        self.rewards = []
        pass

    def episode(self,policy=default):
        for i in range(self.N):

            episode = []
            env = Blackjack()
            state = env._get_obs()
            while True:
                action = default(state)
                next_state, reward, done = env.step(action)
                episode.append((state,action,reward))

                if done:
                    break

                state = next_state
                pass

            for k,e in enumerate(episode):
                state,_,_ = e
                G = 0
                s = State(state)

                for j,x in enumerate(episode[k:]): G+=x[2]
                self.Rs[s]=self.Rs.get(s,0)+G; self.Ns[s]=self.Ns.get(s,
0)+1
                self.Qs[s] = self.Rs[s]/self.Ns[s]

                pass
            avg = sum(self.Rs.values())/sum(self.Ns.values())
            self.rewards.append(avg)
            self
            pass
        pass

    def graph(self):
        j=1;
        plt.figure(figsize=(20,9));
        for ace in [True,False]:
            playerHand,dealerHand=np.meshgrid(np.arange(1,22),np.arange(
1,11))
```

```
            N = len(playerHand.ravel())
            z=[]
            for i in range(N):
                e = State((playerHand.ravel()[i],dealerHand.ravel()[i],a
ce))
                try: z.append(self.Qs[e])
                except KeyError: z.append(0)
                pass

            z = np.array(z).reshape(playerHand.shape[0],playerHand.shape
[1]).T
            pass
            plt.subplot(1,2,j); plt.ylabel("Player's sum", fontsize=16)
            plt.xlabel("Dealer's card",fontsize=16)
            plt.xticks(np.arange(0, 11))
            plt.yticks(np.arange(0, 22, step=2))
            if ace:
                plt.title("With usable ace",fontsize=20)
            else:
                plt.title("Without usable ace",fontsize=16)
            plt.imshow((z),cmap='RdBu')
            j+=1
            pass
            plt.colorbar()
        plt.show()
```

In [141]:
```
game = Event(50)
game.episode()
avreward=sum(game.Rs.values())/sum(game.Ns.values())
print("Average reward:", round(avreward,3))
```

```
Average reward: -0.375
```

ANSWER

The average reward for the chosen policy (stop hitting after sum is greater than 18) is -0.3, which means that in a long term the player always loses.

2

[40 points] Perform Monte Carlo Policy Evaluation

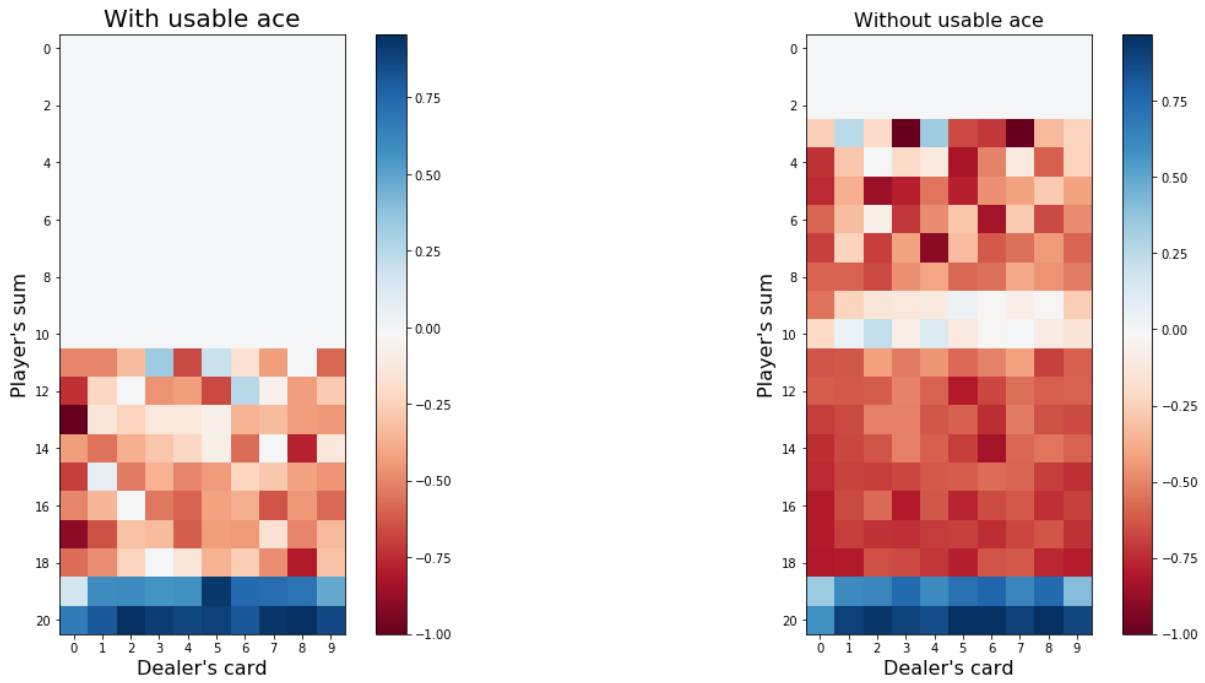
Thinking that you want to make your millions playing blackjack, you decide to test out a policy for playing this game. Your idea is an aggressive strategy: always hit unless the total of your cards adds up to 20 or 21, in which case you stay.

(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:

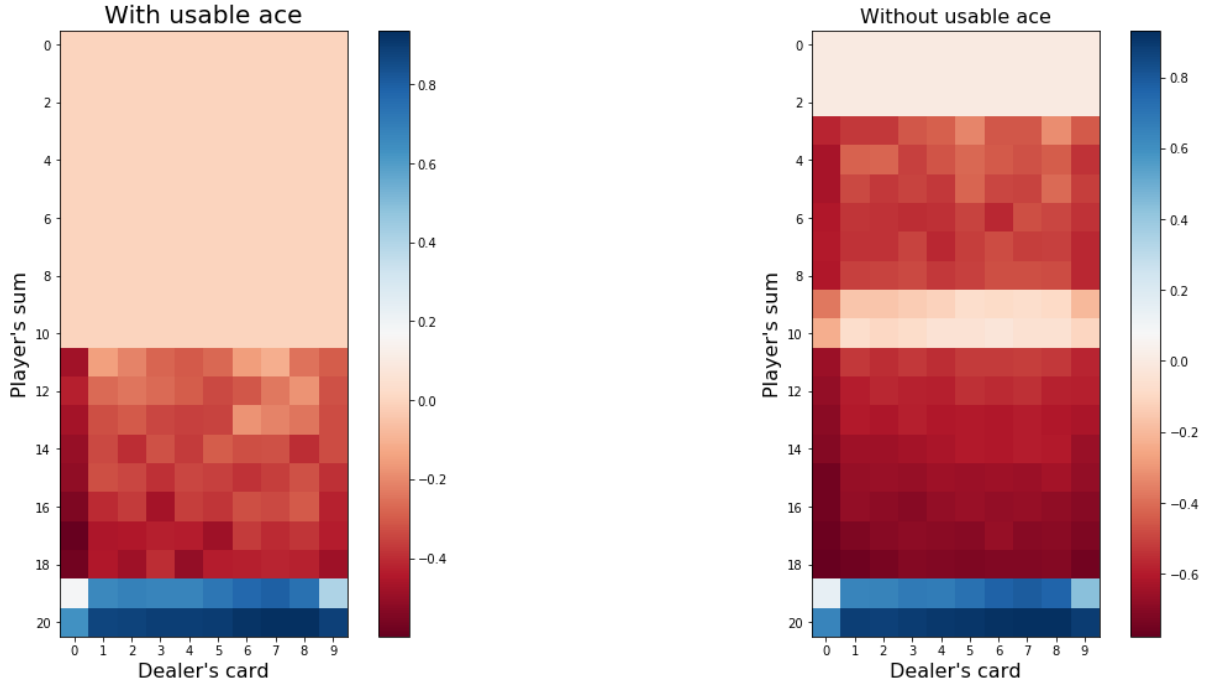
1. When you have a useable ace, plot the state space with the dealer's card on the x-axis, and the player's sum on the y-axis, and use the 'RdBu' matplotlib colormap and `imshow` to plot the value of each state under the policy described above. The domain of your x and y axes should include all possible 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.

In [142]:
```
from matplotlib import pyplot as plt
print ("1,2). Number of episodes = 10 000")
game1 = Event(10000)
game1.episode()
game1.graph()
print ("3,4). Number of episodes = 500 000")
game2 = Event(500000)
game2.episode()
game2.graph()
```

```
1,2). Number of episodes = 10 000
```

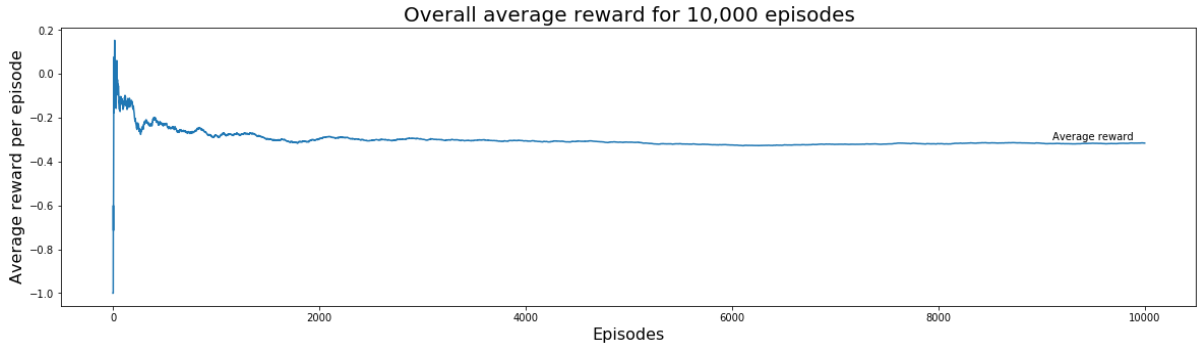



```
3,4). Number of episodes = 500 000
```



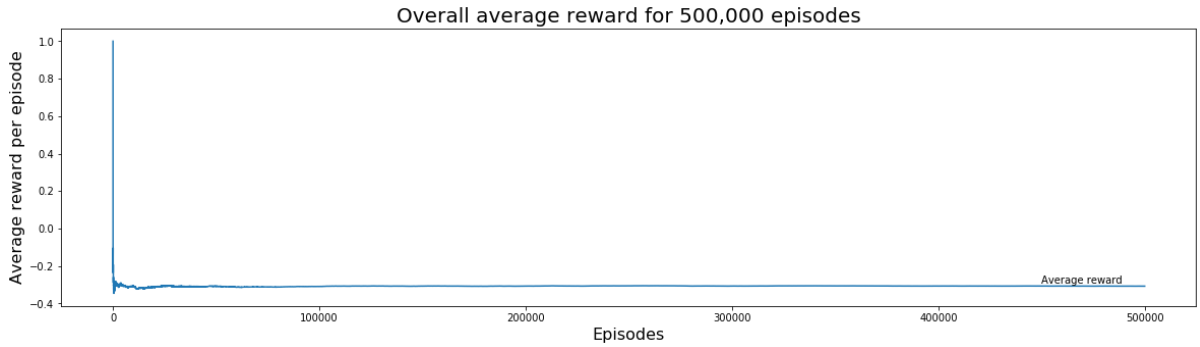
(b) Show a plot of the overall average reward per episode vs the number of episodes. For both the 10,000 episode case and the 500,000 episode case, record the overall average reward for this policy and report that value.

In [115]:
```
plt.figure(figsize=(20,5))
plt.title("Overall average reward for 10,000 episodes", fontsize=20)
plt.xlabel("Episodes",fontsize=16)
plt.ylabel("Average reward per episode", fontsize=16)
plt.plot(np.arange(len(game1.rewards)),game1.rewards)
plt.text(x=9100, y=-0.3, s='Average reward')
plt.show()
reward=sum(game1.rewards)/len(game1.rewards)
print("Overall average reward:", round(reward,3))
```




```
Overall average reward: -0.301
```

In [117]:
```
plt.figure(figsize=(20,5))
plt.title("Overall average reward for 500,000 episodes", fontsize=20)
plt.xlabel("Episodes",fontsize=16)
plt.ylabel("Average reward per episode", fontsize=16)
plt.plot(np.arange(len(game2.rewards)),game2.rewards)
plt.text(x=450000, y=-0.295, s='Average reward')
plt.show()
reward=sum(game2.rewards)/len(game2.rewards)
print("Overall average reward:", round(reward,3))
```




```
Overall average reward: -0.308
```

Conclusion:
For both 500 000 and 10 000 episodes the rewards are approximately equal ~-0.3 (i.e. in the long term the player always loses). On the first graph, we could observe that function has a high variance whereas after approximately 5 000 observations converges to a constant value.

ANSWER

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.

In [119]:
```
import numpy as np
from tqdm import trange
from math import floor
import time

class MonteCarlo:
    def __init__(self):
        self.action_value = np.zeros((400, 2))
        self.action_value_counter = np.zeros((400, 2))
        self.policy = np.ones(400)
        self.average = []
        user_init = np.zeros(20); user_init[18:20] = 1
        self.policy = np.outer(np.outer(user_init, np.ones(10)), np.ones
(2)).ravel()

    def evaluation(self):
        def update_val(state):
            total_r = 0
            cum_value = 0
            for state in state[::-1]:
                index, value, action = state
                cum_value += value
                self.action_value_counter[index, action] += 1
                self.action_value[index, action] += cum_value
                total_r += cum_value
                pass
            self.avg_r_per_e.append(total_r / len(state))
            pass

        game = Blackjack()
        states = []

        player, dealer, use = game.deal()
        done = False
        s_ind = (player - 2)*10*2 + (dealer - 1)*2 + use

        stick_ind = np.where(self.policy == 0)[0]
        while True:
            if done:
                break
            if np.isin(s_ind, stick_ind):
                _, reward, done = game.step(0)
                states.append([s_ind, reward, 0])
                break

            (player, dealer, use), reward, done = game.step(1)
            states.append([s_ind, reward, 1])
            s_ind = (player - 2) * 10 * 2 + (dealer - 1) * 2 + use

        update_val(states)
        pass

    def policy_imporve(self, N0):
        action_value = np.divide(mcc.action_value, mcc.action_value_coun
ter)
        Ns = np.sum(self.action_value_counter, axis=1)
```

```
        epsilon = N0 / (N0 + Ns)
        indicator = np.random.binomial(1, 1 - epsilon, len(epsilon))
        exploit = indicator * np.argmax(action_value, axis=1)
        explore = (1 - indicator) * np.random.randint(0, 2, indicator.sh
ape)
        self.policy = exploit + explore


    def run(self, episodes, N0):
        for _ in trange(episodes):
            self.evaluation()
            self.policy_imporve(N0)
        pass
```

In [120]:
```
episodes = 10000000
mcc = MonteCarlo()
mcc.run(episodes, 100)
```

```
  0%|          | 0/800000 [00:00<?, ?it/s]/anaconda3/lib/python3.6/site
-packages/ipykernel_launcher.py:57: RuntimeWarning: invalid value encou
ntered in true_divide
100%|██████████| 800000/800000 [03:15<00:00, 4083.25it/s]
```

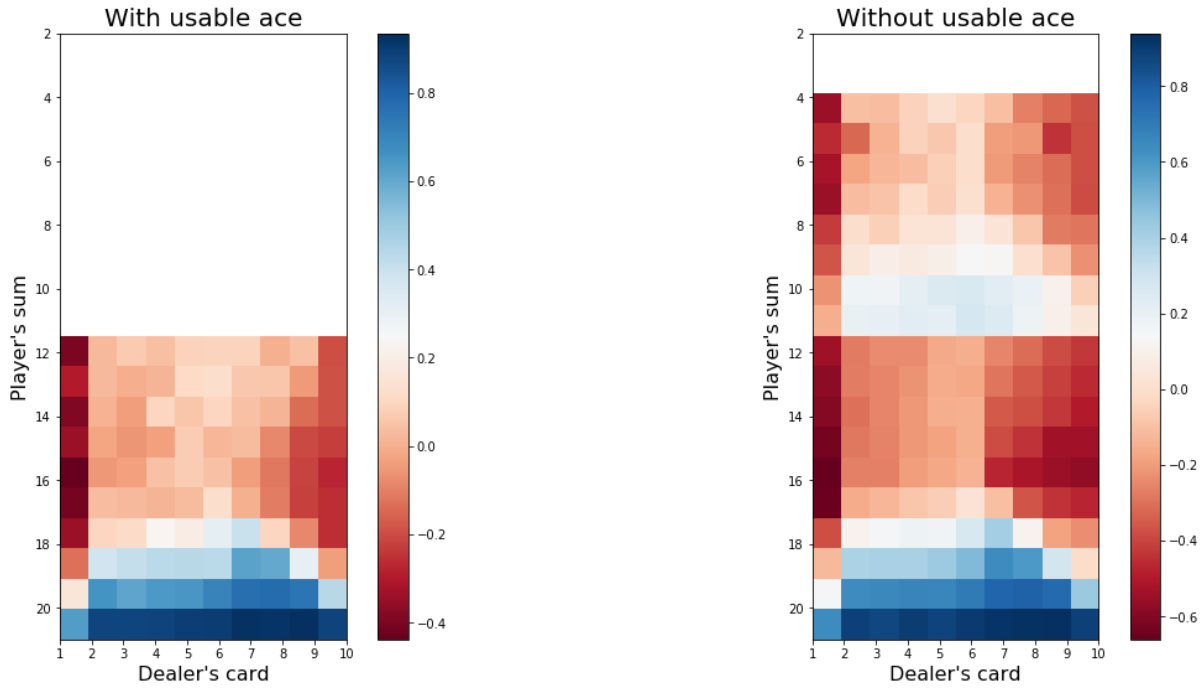
In [160]:
```
action_value = np.divide(mcc.action_value, mcc.action_value_counter).res
hape(20, 10, 2, 2)
action_value = np.max(action_value, axis=3)
val_mat_nonusable = action_value[:, :, 0]
val_mat_usable = action_value[:, :, 1]
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: Runtime
Warning: invalid value encountered in true_divide
  """Entry point for launching an IPython kernel.
```

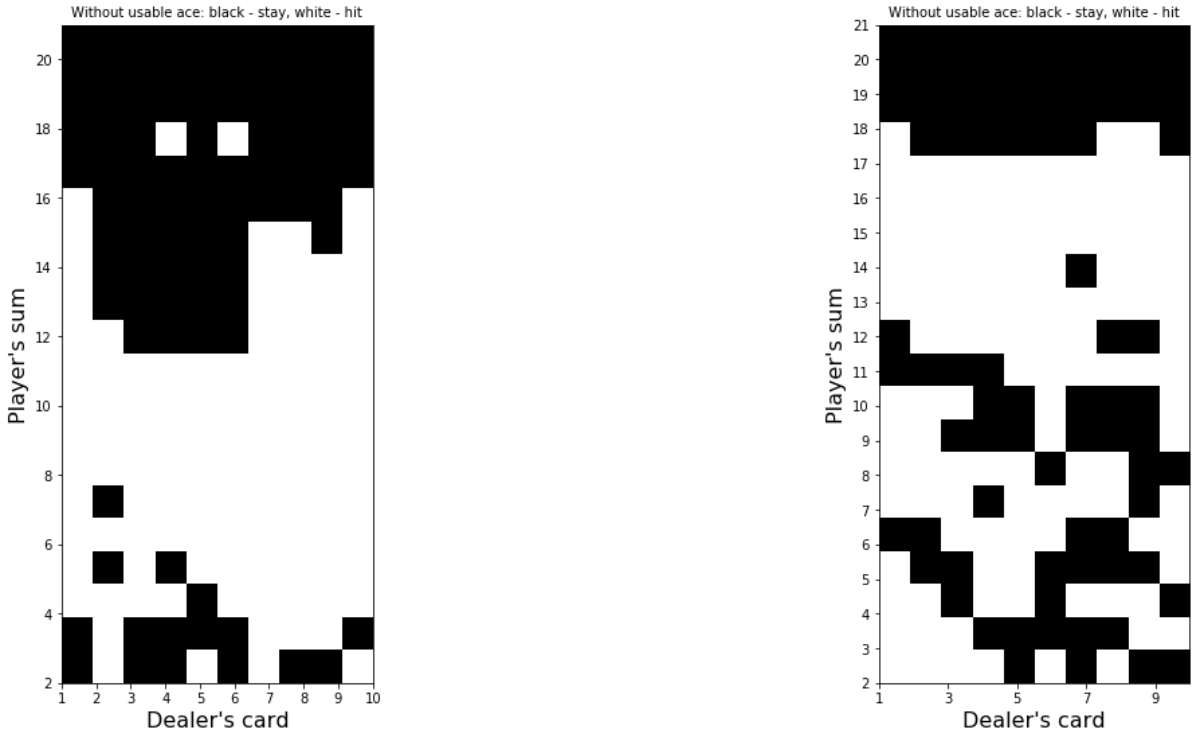
Show your result by plotting the optimal value function: $V^*(s) = max_a Q^*(s, a)$ and the optimal policy $\pi^*(s)$. Create plots for these similar to Sutton and Barto, Figure 5.2 in the new draft edition, or 5.5 in the original edition. Your results SHOULD be very similar to the plots in that text. 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).

In [143]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.figure(figsize=(20, 9))

plt.subplot(121)
plt.imshow(val_mat_usable, extent=[1,10,21,2], cmap='RdBu')
plt.title("With usable ace",fontsize=20)
plt.xticks(np.arange(1, 11, step=1))
plt.yticks(np.arange(2, 22, step=2))
plt.xlabel("Dealer's card",fontsize=16)
plt.ylabel("Player's sum", fontsize=16)
plt.colorbar()

plt.subplot(122)
plt.imshow(val_mat_nonusable, extent=[1,10,21,2],
           cmap='RdBu')
plt.title("Without usable ace", fontsize=20)
plt.xticks(np.arange(1, 11, step=1))
plt.yticks(np.arange(2, 22, step=2))
plt.xlabel("Dealer's card",fontsize=16)
plt.ylabel("Player's sum", fontsize=16)

plt.colorbar()
plt.show()
```



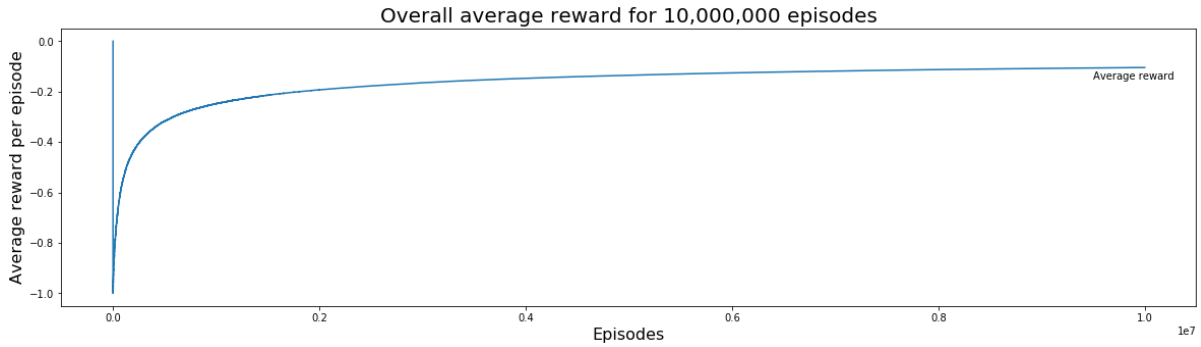
In [159]:
```
plt.figure(figsize=(20, 9))
plt.subplot(121)
plt.imshow(np.flip(mcc.policy.reshape(20, 10, 2)[:, :, 0], 0), extent=[1
,10,2,21],cmap='gray')

plt.title("Without usable ace: black - stay, white - hit",fontsize=10)

plt.xticks(np.arange(1, 11))
plt.yticks(np.arange(2, 22, step=2))
plt.xlabel("Dealer's card",fontsize=16)
plt.ylabel("Player's sum", fontsize=16)


plt.subplot(122)
plt.imshow(np.flip(mcc.policy.reshape(20, 10, 2)[:, :, 1], 0), extent=[1
,10,2,21],cmap='gray')
plt.title("Without usable ace: black - stay, white - hit",fontsize=10)
plt.xticks(np.arange(1, 11, step=2))
plt.yticks(np.arange(2, 22))
plt.xlabel("Dealer's card",fontsize=16)
plt.ylabel("Player's sum", fontsize=16)

plt.show()
```



(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.

In [106]:
```
plt.figure(figsize=(20,5))
plt.title("Overall average reward for 10,000,000 episodes", fontsize=20)
plt.xlabel("Episodes",fontsize=16)
plt.ylabel("Average reward per episode",fontsize=16)
plt.plot(np.arange(len(avg_rewards)),avg_rewards)
plt.text(x=9500000, y=-0.15, s='Average reward')
plt.show()
```



In [100]:
```
print("Average reward after 10 000 000 episodes:", round(reward,3))
```

```
Average reward after 10 000 000 episodes: -0.104
```

ANSWER

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.

ANSWER

(a) Which performs best? Why is this the case?
Monte Carlo Control policy performed the best across all other policies with the overall average reward of -0.104. MC main advantage resulting in the highest average reward is its ability to learn directly from interaction with the environment. However, MC policy performs slowlier because it learns only from complete episodes and must wait until the end of an episode before the return is known. In comparison, the overall average rewards for the human control policy and naive policy were -0.315 and -0.308 respectively.

(b) Could you have created a better policy if you knew the full Markov Decision Process for this environment? Why or why not?
Monte Carlo control policy converges to an optimal policy after some large number of episodes. If the MDP for this environment was known, an optimal policy would be found in a shorter time but would not result in better policy.