Assignment_5

April 3, 2019

1 Assignment 5 - Reinforcement Learning

1.1 *Anna Berman*

Netid: *aeb100*

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.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 [13]: 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(0)

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

```
(14, 10, False)
((24, 10, False), -1, True)
```

```
Out[13]: ((24, 10, False), -1.0, True)
```

```
In [14]: import numpy as np

        rewards = [-1,-1,-1,1,-1,-1,0,-1,-1,0,1,-1,-1,-1,1,-1,1,-1,1,-1,-1,-1,-1,-1,
                   1,-1,-1,1,1,1,-1,1,1,-1,-1,1,1,1,1,-1,-1,1,-1,-1,1,-1,-1,1,1,1]
        print('Number of Games Played: {}'.format(len(rewards)))
        print('Average Reward: {0:0.2}'.format(np.sum(rewards)/len(rewards)))
```

```
Number of Games Played: 50
Average Reward: -0.16
```

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:

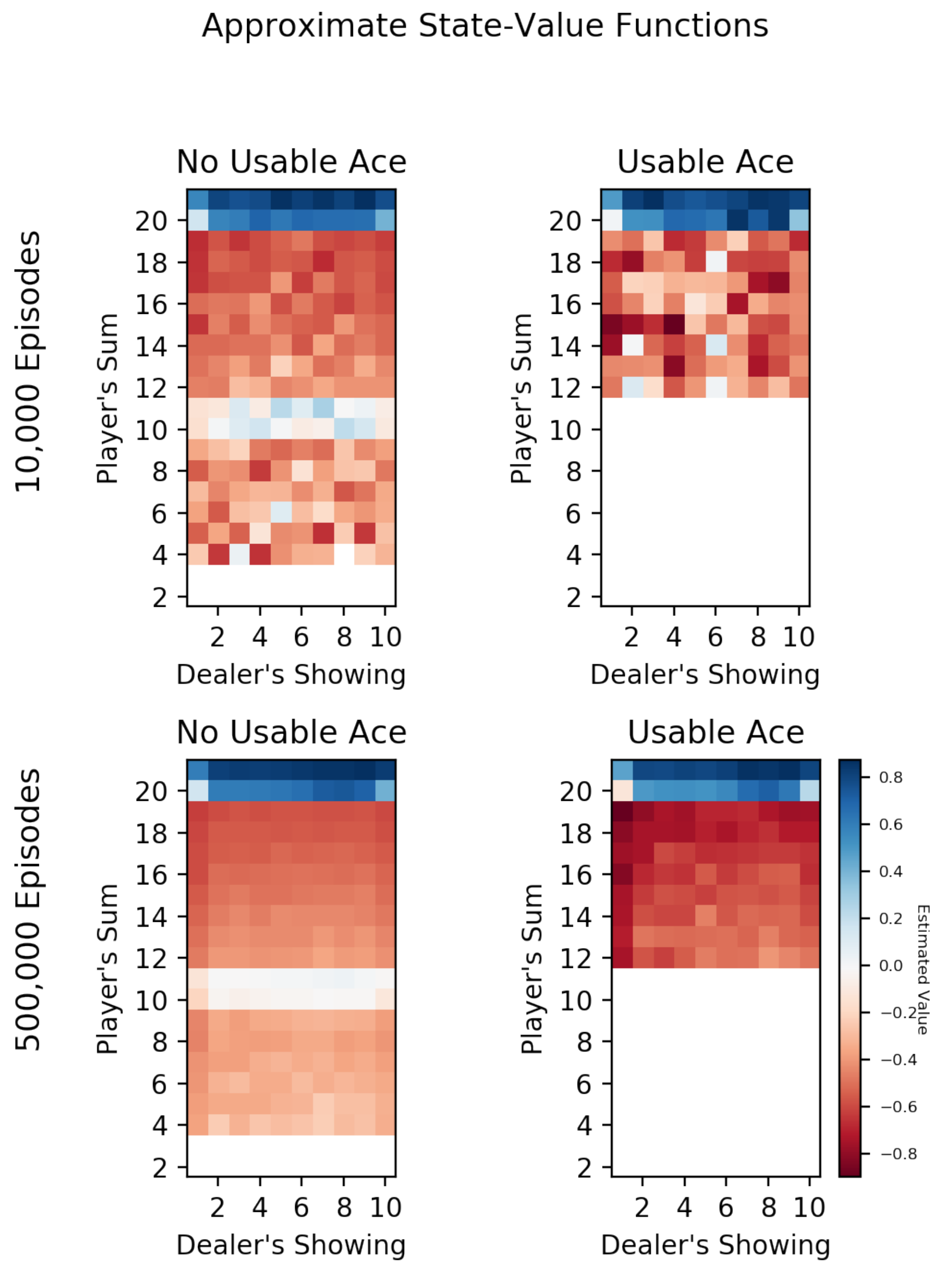
1. When you have a useable ace, plot the state space with the dealer's card on the x-axis, and the player's sum on the y-axis, and use the 'RdBu' matplotlib colormap and imshow to plot the value of each state under the policy described above. The domain of your x and y axes should include all possible 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. Repeat (2) for the case of 500,000 episodes.

For both cases with a usable ace and no usable ace, the rewards for this strategy are highest if you have a 20 or a 21, but very low otherwise. Rewards are lowest at the boundary between 19 and 20 as it is very risky to hit which such a low change of not going over 21. The results from these estimation are much smoother in the 500,000 game estimation compared to the 100,000 game estimation as the state-values converge towards the true average for the policy.

```
In [3]: import matplotlib.pyplot as plt
        import numpy as np
        import warnings
        warnings.filterwarnings('ignore')

        game = Blackjack()

        def monte_carlo(games, discount = 1):
            # Monte Carlo simulation of blackjack game
            # Input: number of episodes, temporal discount rate
            # Output: average state values, avg rewards

            # Initialize storage arrays of rewards and counts
            storage_ace_reward = np.zeros((32, 11))
            storage_ace_count = np.zeros((32, 11))
            storage_no_ace_reward = np.zeros((32, 11))
            storage_no_ace_count = np.zeros((32, 11))
            avg_rewards = [0] # list of all rewards in order

            # Run through all the games
            for i in range(games):
                stop_flag = False # True if games is over
                stages = [] # list of stages

                # Deal the first hand
                (player_sum, dealer_show, usable_ace) = game.deal()
                stages.append((player_sum, usable_ace))

                # Play a full game
                while not stop_flag:
                    # If dealt 20 or 21, hold
                    if player_sum == 20 or player_sum == 21:
                        (player_sum, dealer_show, usable_ace), reward, stop_flag = game.step(0
                        pass
                    # Else, hit
                    else:
                        (player_sum, dealer_show, usable_ace), reward, stop_flag = game.step(1
                        # Update stages
                        stages.append((player_sum, usable_ace))
                        pass
                    pass

                # Update avg reward
                new_avg = online_avg(old_avg = avg_rewards[-1], new_x = reward,
                                     N = len(avg_rewards))
                avg_rewards.append(new_avg)

                # Store results
                for i, s in enumerate(stages):
                    # Assign discounted reward based on distance to final hand
                    num_steps = len(stages)
                    discounted_reward = discount**(num_steps-i) * reward
                    # Usable ace
                    if s[1] == True:
                        storage_ace_reward[s[0]][dealer_show] += discounted_reward
                        storage_ace_count[s[0]][dealer_show] += 1
                        pass
                    # No usable ace
                    else:
                        storage_no_ace_reward[s[0]][dealer_show] += discounted_reward
                        storage_no_ace_count[s[0]][dealer_show] += 1
                        pass
                    pass

                pass

            # Calculate averages
            mean_ace = storage_ace_reward/storage_ace_count
            mean_no_ace = storage_no_ace_reward/storage_no_ace_count

            return (mean_ace, mean_no_ace, avg_rewards)
            pass

        def online_avg(old_avg, new_x, N):
            return new_x/N + ((N-1)/N) * old_avg

        # Play 10,000 and then 500,000 games
        mean_ace_10000, mean_no_ace_10000, avg_rewards_10000 = monte_carlo(10000, discount = .
        mean_ace_500000, mean_no_ace_500000, avg_rewards_500000 = monte_carlo(500000, discount
```

```
In [41]: # Plot results
         fig, axs = plt.subplots(2, 2)
         fig.set_dpi(200)
         fig.set_figheight(7)
         fig.set_figwidth(5)

         # 10,000 games no ace
         axs[0,0].imshow(mean_no_ace_10000, cmap = 'RdBu')
         axs[0,0].set_xlabel('Dealer\'s Showing')
         axs[0,0].set_ylabel('Player\'s Sum')
         axs[0,0].set_xticks(np.arange(0, 11, step=2))
         axs[0,0].set_yticks(np.arange(0, 22, step=2))
```

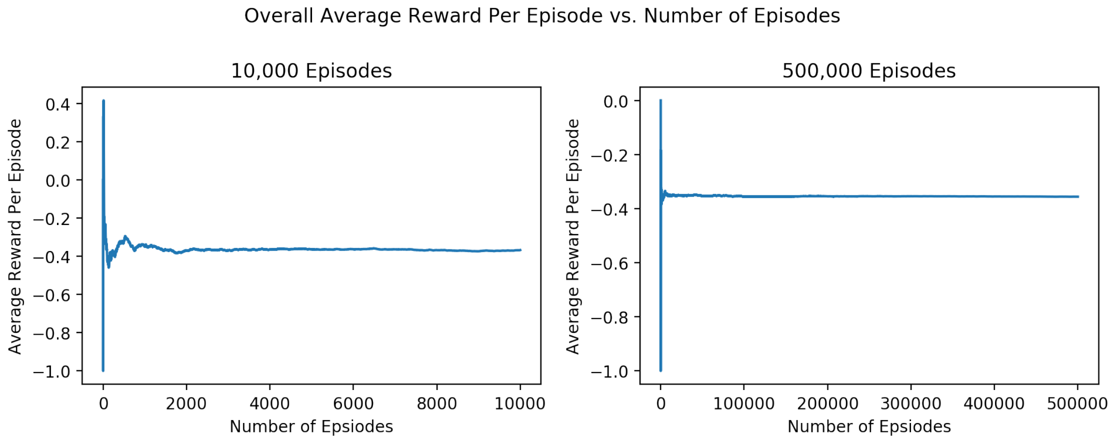
```
axs[0,0].set_xlim(.5,10.5)
axs[0,0].set_ylim(1.5, 21.5)
axs[0,0].set_title('No Usable Ace')

# 10,000 games usable ace
axs[0,1].imshow(mean_ace_10000, cmap = 'RdBu')
axs[0,1].set_xlabel('Dealer\'s Showing')
axs[0,1].set_ylabel('Player\'s Sum')
axs[0,1].set_xticks(np.arange(0, 11, step=2))
axs[0,1].set_yticks(np.arange(0, 22, step=2))
axs[0,1].set_xlim(.5,10.5)
axs[0,1].set_ylim(1.5, 21.5)
axs[0,1].set_title('Usable Ace')
cb = plt.colorbar(axs[0,0].imshow(mean_no_ace_10000, cmap = 'RdBu'))
cb.set_label('Estimated Value', rotation=270, size=6)
cb.ax.tick_params(labelsize=6)

# 500,000 games no ace
axs[1,0].imshow(mean_no_ace_500000, cmap = 'RdBu')
axs[1,0].set_xlabel('Dealer\'s Showing')
axs[1,0].set_ylabel('Player\'s Sum')
axs[1,0].set_xticks(np.arange(0, 11, step=2))
axs[1,0].set_yticks(np.arange(0, 22, step=2))
axs[1,0].set_xlim(.5,10.5)
axs[1,0].set_ylim(1.5, 21.5)
axs[1,0].set_title('No Usable Ace')

# 500,000 games usable ace
axs[1,1].imshow(mean_ace_500000, cmap = 'RdBu')
axs[1,1].set_xlabel('Dealer\'s Showing')
axs[1,1].set_ylabel('Player\'s Sum')
axs[1,1].set_xticks(np.arange(0, 11, step=2))
axs[1,1].set_yticks(np.arange(0, 22, step=2))
axs[1,1].set_xlim(.5,10.5)
axs[1,1].set_ylim(1.5, 21.5)
axs[1,1].set_title('Usable Ace')


fig.suptitle('Approximate State-Value Functions')
fig.text(0.01, 0.8, '10,000 Episodes', rotation=90, fontsize = 12)
fig.text(0.01, 0.4, '500,000 Episodes', rotation=90, fontsize = 12)
plt.tight_layout(rect=[0.03, 0.03, 1, .90])
plt.show()
```



(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 [12]: # Plot rewards
         fig, axs = plt.subplots(1, 2)
         fig.set_dpi(200)
         fig.set_figheight(4)
         fig.set_figwidth(10)
```

```
axs[0].plot(avg_rewards_10000[:])
axs[0].set_title('10,000 Episodes')
axs[0].set_xlabel('Number of Epsiodes')
axs[0].set_ylabel('Average Reward Per Episode')

axs[1].plot(avg_rewards_500000[:])
axs[1].set_title('500,000 Episodes')
axs[1].set_xlabel('Number of Epsiodes')
axs[1].set_ylabel('Average Reward Per Episode')

fig.suptitle('Overall Average Reward Per Episode vs. Number of Episodes')
plt.tight_layout(rect=[0.03, 0.03, 1, .90])
plt.show()
```



In each case, the rewards quickly converge to approximately -.3. Because we have a fixed policy, there is no improvement in rewards over time. Instead, as the number of episodes increases we approach the true average state-value.

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).

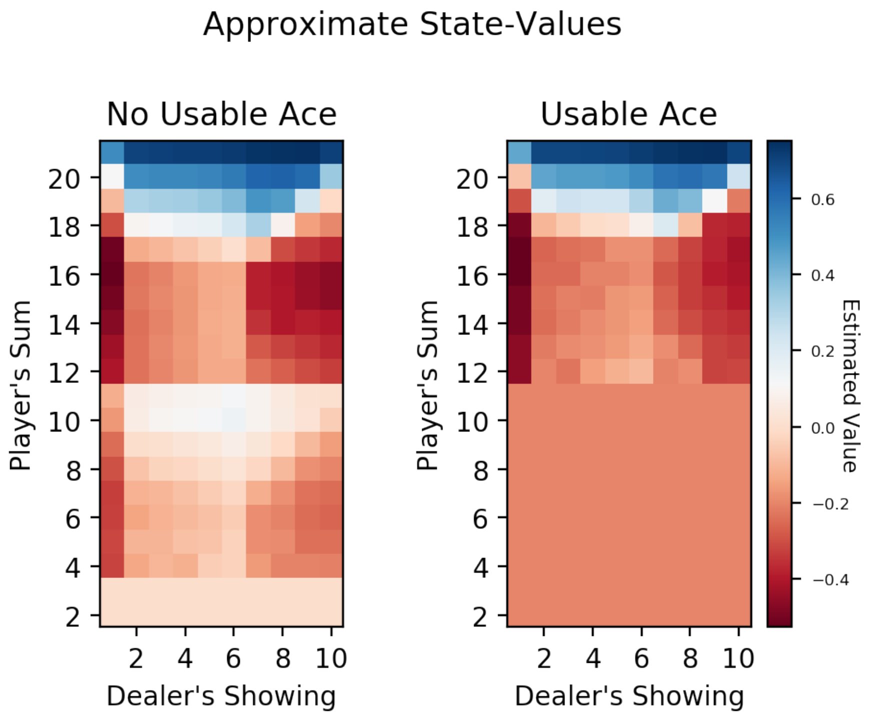
```
In [3]: import matplotlib.pyplot as plt
        import numpy as np
        from scipy.stats import bernoulli

        game = Blackjack()

        def monte_carlo(games, discount = 1):
            # Monte Carlo simulation of blackjack game
            # Input: number of episodes, temporal discount for rewards
            # Output: State values, optiminal policy, rewards

            # Initialize storage arrays of rewards and counts
            avg_ace_reward = np.zeros((22, 11, 2))
            avg_no_ace_reward = np.zeros((22, 11, 2))
            storage_ace_count = np.zeros((22, 11, 2))
            storage_no_ace_count = np.zeros((22, 11, 2))
            avg_rewards = [0] # list of all rewards in order

            # Run through all the games
            for i in range(games):
                stop_flag = False # True if games is over
                stages = [] # list of stages

                # Deal the first hand
                (player_sum, dealer_show, usable_ace) = game.deal()

                # Play a full game
                while not stop_flag:
                    # Compare value-action pairs and choose the highest for state
                    # action = 0 if stay, 1 if hit
                    # N_S = Number of times stage has been explored before
                    if usable_ace == True:
```

```
                action = np.argmax(avg_ace_reward[player_sum][dealer_show])
                N_S = np.sum(storage_ace_count[player_sum][dealer_show])
                pass
            else:
                action = np.argmax(avg_no_ace_reward[player_sum][dealer_show])
                N_S = np.sum(storage_no_ace_count[player_sum][dealer_show])
                pass

            # Epsilon greedy exploration
            action = int(bernoulli.rvs(np.abs(action - epsilon(N_S,1000)),
                                       size = 1))
            stages.append((player_sum, usable_ace, action))

            # Take action
            (player_sum, dealer_show, usable_ace), reward,
stop_flag = game.step(action)
            pass

        # Update avg reward
        new_avg = online_avg(old_avg = avg_rewards[-1], new_x = reward,
                             N = len(avg_rewards))
        avg_rewards.append(new_avg)

        # Update policy (Store results)
        for i, s in enumerate(stages):
            # Assign discounted reward based on distance to final hand
            num_steps = len(stages)
            discounted_reward = discount**(num_steps-i) * reward
            play_sum_temp = s[0]
            use_a_temp = s[1]
            act_temp = s[2]

            # Usable ace
            if use_a_temp == True:
                new_avg = online_avg(old_avg = avg_ace_reward[play_sum_temp][dealer_sho
                                     new_x = discounted_reward,
                                     N = storage_ace_count[play_sum_temp][dealer_show]
                avg_ace_reward[play_sum_temp][dealer_show][act_temp] = new_avg
                storage_ace_count[play_sum_temp][dealer_show][act_temp] += 1
                pass
            # No usable ace
            else:
                new_avg = online_avg(old_avg = avg_no_ace_reward[play_sum_temp][dealer_
                                     new_x = discounted_reward,
                                     N = storage_no_ace_count[play_sum_temp][dealer_sho
                avg_no_ace_reward[play_sum_temp][dealer_show][act_temp] = new_avg
                storage_no_ace_count[play_sum_temp][dealer_show][act_temp] += 1
                pass
            pass

    return (avg_ace_reward, avg_no_ace_reward, stages, avg_rewards)
    pass


def online_avg(old_avg, new_x, N):
    # Calculates an incremental average
    return new_x/N + ((N-1)/N) * old_avg

def epsilon(N, N0 = 100):
    # calculates epsilson for e-greedy exploration
    # explores less as state is explored
    return N0/(N0 + N)
```

In [4]:
```
# Play games
mean_ace, mean_no_ace, stages, rewards = monte_carlo(5000000, discount = .8)
```

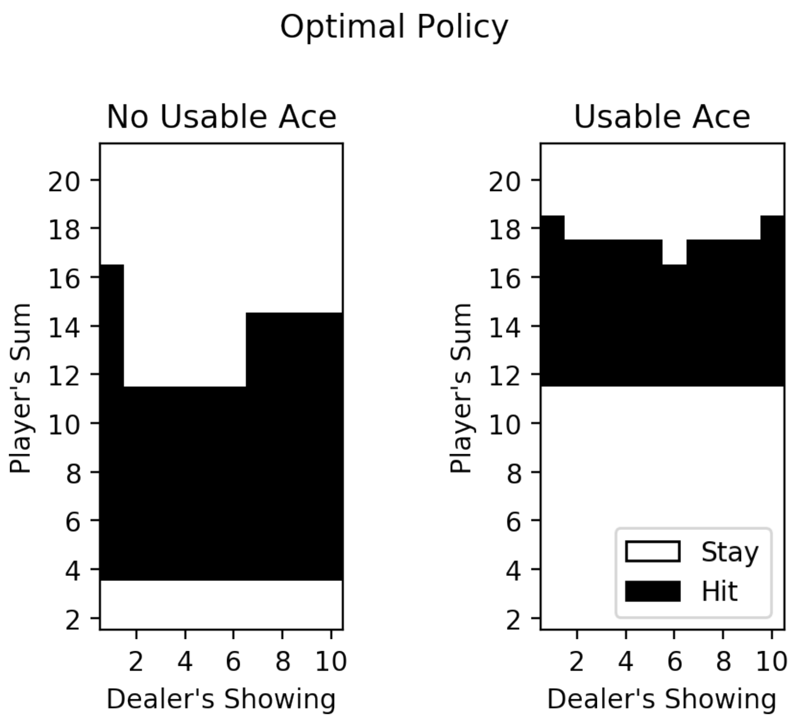
In [42]:
```
# Plot optimal state values
fig, axs = plt.subplots(1, 2)
fig.set_dpi(200)
fig.set_figheight(4)
fig.set_figwidth(5)

# no ace
axs[0].imshow(np.max(mean_no_ace, axis = 2), cmap = 'RdBu')
axs[0].set_xlabel('Dealer\'s Showing')
axs[0].set_ylabel('Player\'s Sum')
axs[0].set_xticks(np.arange(0, 11, step=2))
axs[0].set_yticks(np.arange(0, 22, step=2))
axs[0].set_xlim(.5,10.5)
axs[0].set_ylim(1.5, 21.5)
axs[0].set_title('No Usable Ace')

# usable ace
axs[1].imshow(np.max(mean_ace, axis = 2), cmap = 'RdBu')
axs[1].set_xlabel('Dealer\'s Showing')
axs[1].set_ylabel('Player\'s Sum')
axs[1].set_xticks(np.arange(0, 11, step=2))
axs[1].set_yticks(np.arange(0, 22, step=2))
axs[1].set_xlim(.5,10.5)
axs[1].set_ylim(1.5, 21.5)
axs[1].set_title('Usable Ace')
cb = plt.colorbar(axs[0].imshow(np.max(mean_no_ace, axis =2), cmap = 'RdBu'))
cb.set_label('Estimated Value', rotation=270, size = 8)
cb.ax.tick_params(labelsize=6)

fig.suptitle('Approximate State-Values')
```

```
plt.tight_layout(rect=[0.03, 0.03, 1, .90])
plt.show()
```



```
In [6]: from matplotlib.patches import Patch
        # Plot policy decision boundary
        fig, axs = plt.subplots(1, 2)
        fig.set_dpi(200)
        fig.set_figheight(4)
        fig.set_figwidth(5)

        # no ace
        axs[0].imshow(np.argmax(mean_no_ace, axis = 2), cmap = 'binary')
        axs[0].set_xlabel('Dealer\'s Showing')
        axs[0].set_ylabel('Player\'s Sum')
        axs[0].set_xticks(np.arange(0, 11, step=2))
        axs[0].set_yticks(np.arange(0, 22, step=2))
        axs[0].set_xlim(.5,10.5)
        axs[0].set_ylim(1.5, 21.5)
        axs[0].set_title('No Usable Ace')

        # usable ace
        axs[1].imshow(np.argmax(mean_ace, axis = 2), cmap = 'binary')
        axs[1].set_xlabel('Dealer\'s Showing')
        axs[1].set_ylabel('Player\'s Sum')
```

```
axs[1].set_xticks(np.arange(0, 11, step=2))
axs[1].set_yticks(np.arange(0, 22, step=2))
axs[1].set_xlim(.5,10.5)
axs[1].set_ylim(1.5, 21.5)
axs[1].set_title('Usable Ace')

legend_elements = [Patch(facecolor='white', edgecolor='black'),
                   Patch(facecolor='black', edgecolor='black')]
plt.legend(legend_elements, ['Stay', 'Hit'], loc=4)

fig.suptitle('Optimal Policy')
plt.tight_layout(rect=[0.03, 0.03, 1, .90])
plt.show()
```



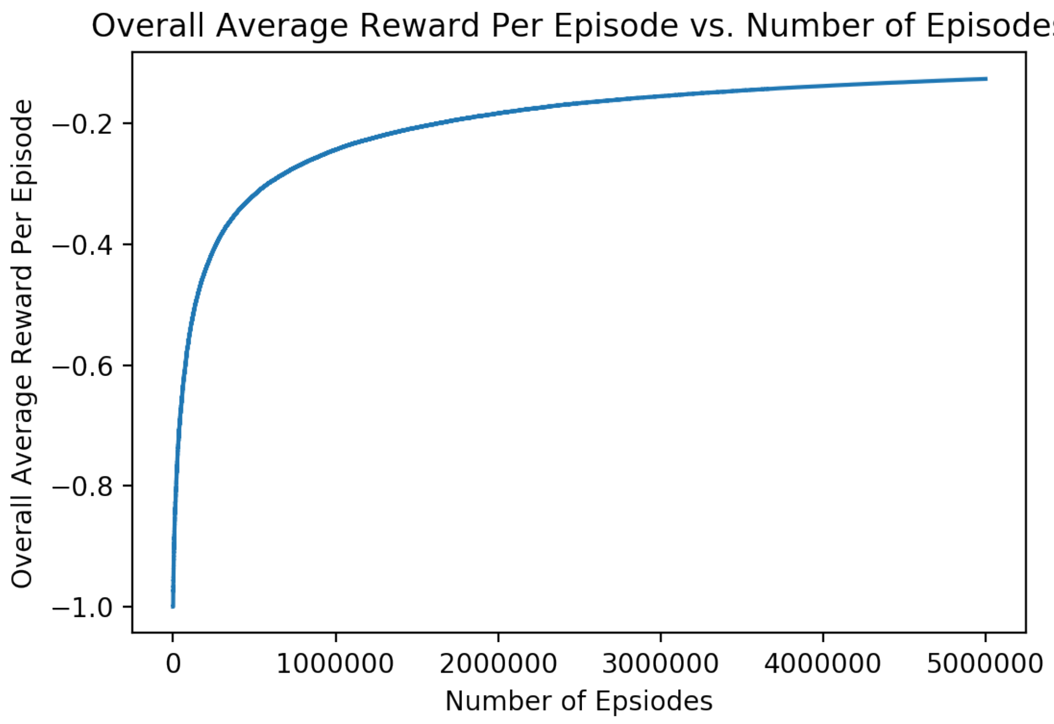
After 5,000,000 episodes, the estimated state-values and optimal policy closely resembles the Sutton and Barto text. As seen in the change of average rewards over episodes, after 5 million episodes convergence was still not achieved. This was most likely due to the epsilon greedy implementation in which I chose set N_0 to 1000 as opposed to 100 to facilitate additional exploration. Given more time, convergence may have been achieved.

According to our optimal policy, we can see that it's best to hit in situations where we have a usable ace and are holding a player sum of 17 or less as well as in situations were we don't have a useable ace and we have 11 or less. The policy also shows that it's in our best interest to hit even when we have a higher player sum when the dealer's showing is very close to 1 or 10.

(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 [7]: # Plot rewards
        plt.figure(dpi = 200)
        plt.plot(rewards[1:])
        plt.title('Overall Average Reward Per Episode vs. Number of Episodes')
        plt.xlabel('Number of Epsiodes')
        plt.ylabel('Overall Average Reward Per Episode')
        plt.show()
```



As the number of episodes increases our average reward increases. Unlike our Monte Carlo Policy evaluation, as the number of episodes approaches infinity, our rewards improve until convergence.

1.5 4

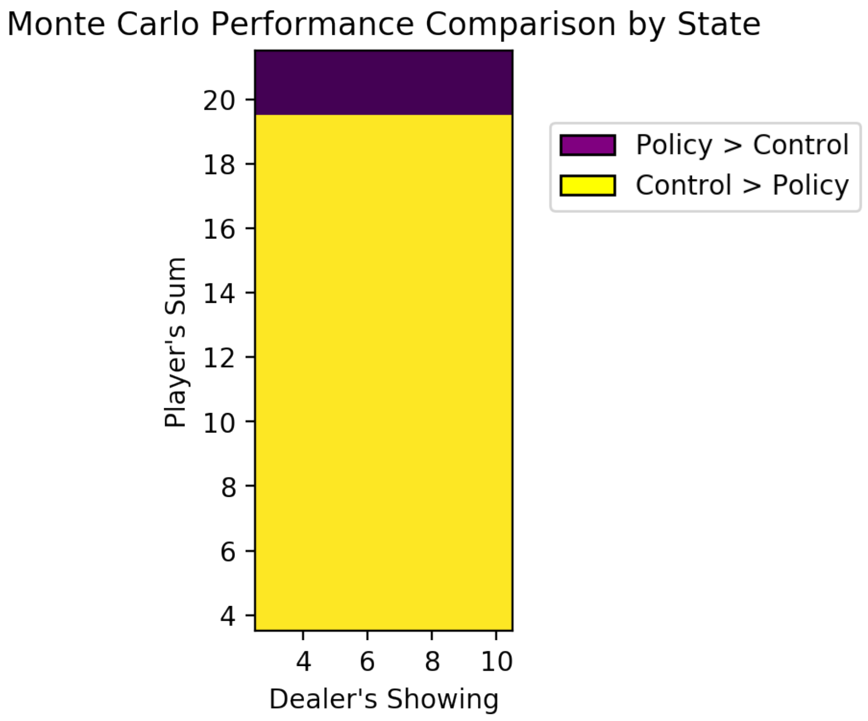
1.5.1 [10 points] Discuss your finding

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?

The Monte Carlo control case performed the best in our case. Specifically, in our Monte Carlo control case we have more states with higher values for the majority of states. With the exception of having a player sum of 20 or 21, in all other cases, the Monte Carlo Control case outperforms the Monte Carlo Policy case in estimated state-value.

```
In [93]: # Plot comparison between state values of MC policy vs. control
         plt.figure(dpi = 200)
         plt.imshow((np.max(mean_no_ace[:22,:22], axis = 2) - mean_no_ace_500000[:22,:])>=0)
         plt.xlabel('Dealer\'s Showing')
         plt.ylabel('Player\'s Sum')
         plt.xticks(np.arange(0, 11, step=2))
         plt.yticks(np.arange(0, 22, step=2))
         plt.xlim(2.5,10.5)
         plt.ylim(3.5, 21.5)
         plt.title('Monte Carlo Performance Comparison by State')
         legend_elements = [Patch(facecolor='purple', edgecolor='black'),
                            Patch(facecolor='yellow', edgecolor='black')]
         plt.legend(legend_elements, ['Policy > Control', 'Control > Policy'], bbox_to_anchor=
         bbox_to_anchor=(1.1, 1.05)
         plt.show()
```



(b) Could you have created a better policy if you knew the full Markov Decision Process for this environment? Why or why not?

Theoretically, we can create a better policy if we knew the full Markov Decision Process because in that case we would have all the information about the environment needed to achieve convergence. However, although we do have complete knowledge of the environment, it would not be easy to apply dynamic programming methods because DP methods require the probability distribution of next events. Unfortunately, this probability distribution is not easy to calculate for the case of blackjack.