Assignment 5 - Reinforcement Learning

Yifei Wang

Netid: *yw323*

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)](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

class Blackjack():
    """Simple blackjack environment adapted from OpenAI Gym:
        https://github.com/openai/gym/blob/master/gym/envs/toy_text/blac
kjack.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 playe
r 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 th
```

```
is case)
            The Boolean is a flag (False in this case) to indicate wheth
er 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 o
f the
            state of the game as described above.
            The second value (0) indicates the rewards
            The third value (False) indicates whether the game is finish
ed
    """

    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()

    #------------------------------------------
    # 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(1)
print(s1)

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

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

```
(15, 2, False)
((21, 2, False), 0, False)
```

ANSWER

In [3]:
```python
def human_policy(state):
    '''
    simplified human policy
    '''

    player_state = state[0] if isinstance(state[0], tuple) else state
    player_sum = player_state[0]
    dealer_sum = player_state[1]
    player_ace = player_state[2]
    if dealer_sum in (1,7,8,9,10):
        hit_thres = 16
    else:
        hit_thres = 15
    if player_sum >= hit_thres:
        action = 0
    else:
        action = 1
    return action


def human_play(n_iter=50):
    '''
    use human policy to play for n_iter rounds
    '''

    print(">"*10 + " Human Policy")
    rewards = 0
    states = []
    for n in range(n_iter):
        state = []
        game = Blackjack()
        state.append(game.deal())
        done = False
        if not done:
            s = game.step(human_policy(state[-1]))
            state.append(s[0])
            done = s[2]
            rewards += s[1]
        state.append(s[1])
        states.append(state)
    print("n_hands: %d\trewards: %d" % (n_iter, rewards))
    return states, rewards


def states_print(states):
    '''
    print the states
    '''
    print(">>>>>  Result for %d States  >>>>>" % len(states))
    for state in states:
        for s in state:
            print(s, end="\t")
        print("")
```

In [4]:

```
states, rewards = human_play(n_iter=50)
states_print(states)
print("expected rewards %a" % (rewards/50))
```

```
>>>>>>>>>>> Human Policy
n_hands: 50     rewards: 0
>>>>>  Result for 50 States  >>>>>
(20, 6, False)  (20, 6, False)  1.0
(17, 10, False) (17, 10, False) 0.0
(16, 2, False)  (16, 2, False)  1.0
(20, 10, False) (20, 10, False) 1.0
(12, 3, False)  (15, 3, False)  0
(15, 10, False) (16, 10, False) 0
(12, 10, False) (15, 10, False) 0
(18, 8, False)  (18, 8, False)  0.0
(20, 7, False)  (20, 7, False)  -1.0
(21, 2, True)   (21, 2, True)   1.0
(15, 8, False)  (24, 8, False)  -1
(13, 10, False) (23, 10, False) -1
(14, 6, False)  (24, 6, False)  -1
(18, 8, False)  (18, 8, False)  1.0
(10, 9, False)  (20, 9, False)  0
(12, 6, True)   (12, 6, False)  0
(15, 5, True)   (15, 5, True)   -1.0
(20, 10, False) (20, 10, False) 1.0
(11, 10, False) (14, 10, False) 0
(12, 3, False)  (17, 3, False)  0
(18, 10, True)  (18, 10, True)  -1.0
(12, 9, False)  (22, 9, False)  -1
(18, 7, True)   (18, 7, True)   1.0
(15, 8, False)  (24, 8, False)  -1
(18, 3, False)  (18, 3, False)  1.0
(17, 10, False) (17, 10, False) 0.0
(15, 4, False)  (15, 4, False)  1.0
(11, 5, False)  (20, 5, False)  0
(13, 9, False)  (23, 9, False)  -1
(15, 10, False) (20, 10, False) 0
(20, 10, False) (20, 10, False) 1.0
(13, 9, False)  (23, 9, False)  -1
(14, 10, True)  (13, 10, False) 0
(9, 4, False)   (15, 4, False)  0
(15, 6, True)   (15, 6, True)   1.0
(11, 10, False) (21, 10, False) 0
(10, 1, False)  (15, 1, False)  0
(14, 2, False)  (24, 2, False)  -1
(8, 1, False)   (11, 1, False)  0
(8, 7, False)   (14, 7, False)  0
(11, 10, False) (14, 10, False) 0
(8, 6, False)   (18, 6, False)  0
(9, 1, False)   (19, 1, False)  0
(7, 2, False)   (17, 2, False)  0
(16, 2, False)  (16, 2, False)  -1.0
(6, 4, False)   (16, 4, False)  0
(13, 6, False)  (16, 6, False)  0
(8, 10, False)  (18, 10, False) 0
(20, 5, False)  (20, 5, False)  1.0
(12, 8, False)  (20, 8, False)  0
expected rewards 0.0
```

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

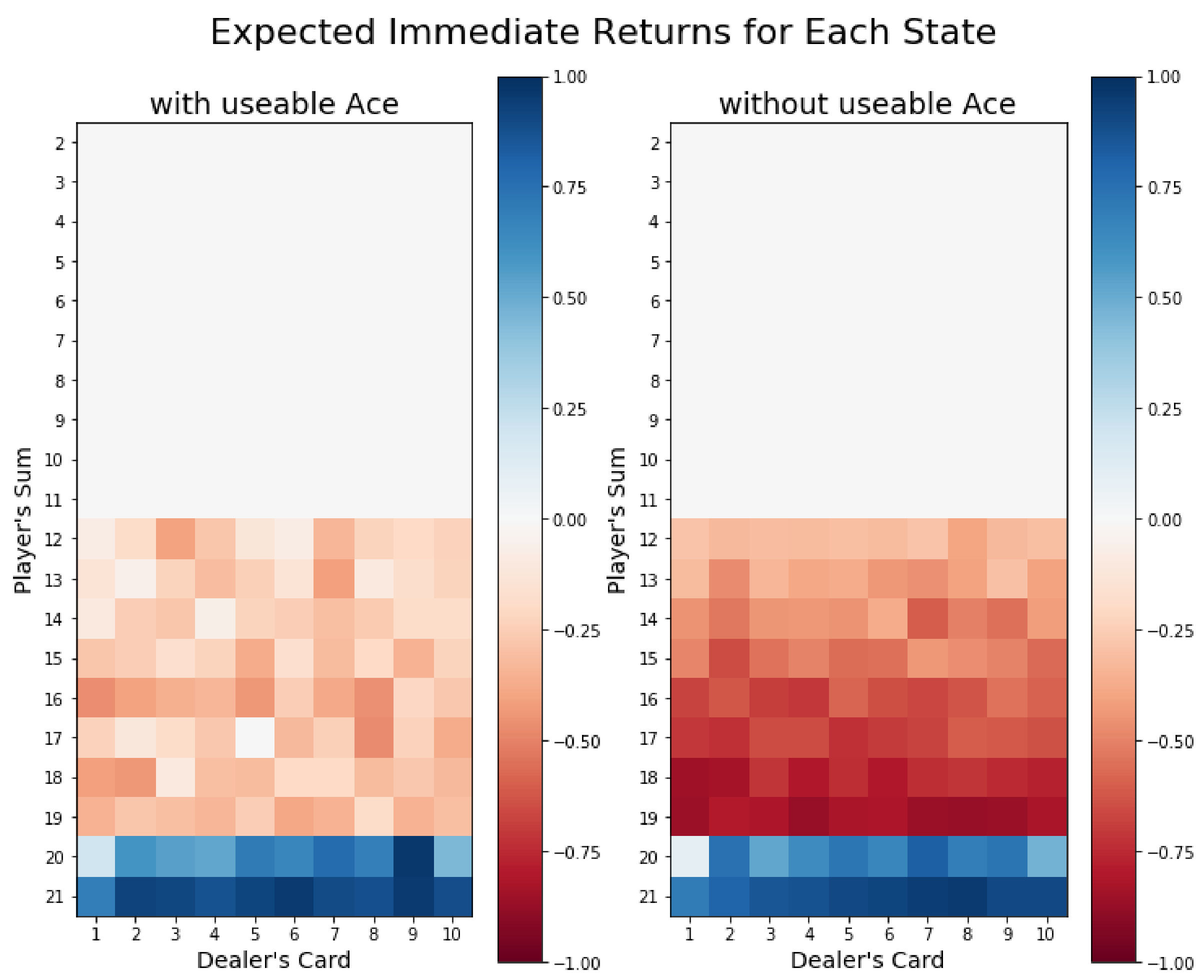
(a) Use Monte Carlo policy evaluation to evaluate the expected returns from each state. Create plots for these similar to Sutton and Barto, Figure 5.1 where you plot the expected returns for each state.

In [5]:
```
def aggressive_policy(state):
    '''
    aggressive human policy
    '''

    player_state = state[0] if isinstance(state[0], tuple) else state
    player_sum = player_state[0]
    dealer_sum = player_state[1]
    player_ace = player_state[2]
    hit_thres = 20
    if player_sum >= hit_thres:
        action = 0
    else:
        action = 1
    return action
```

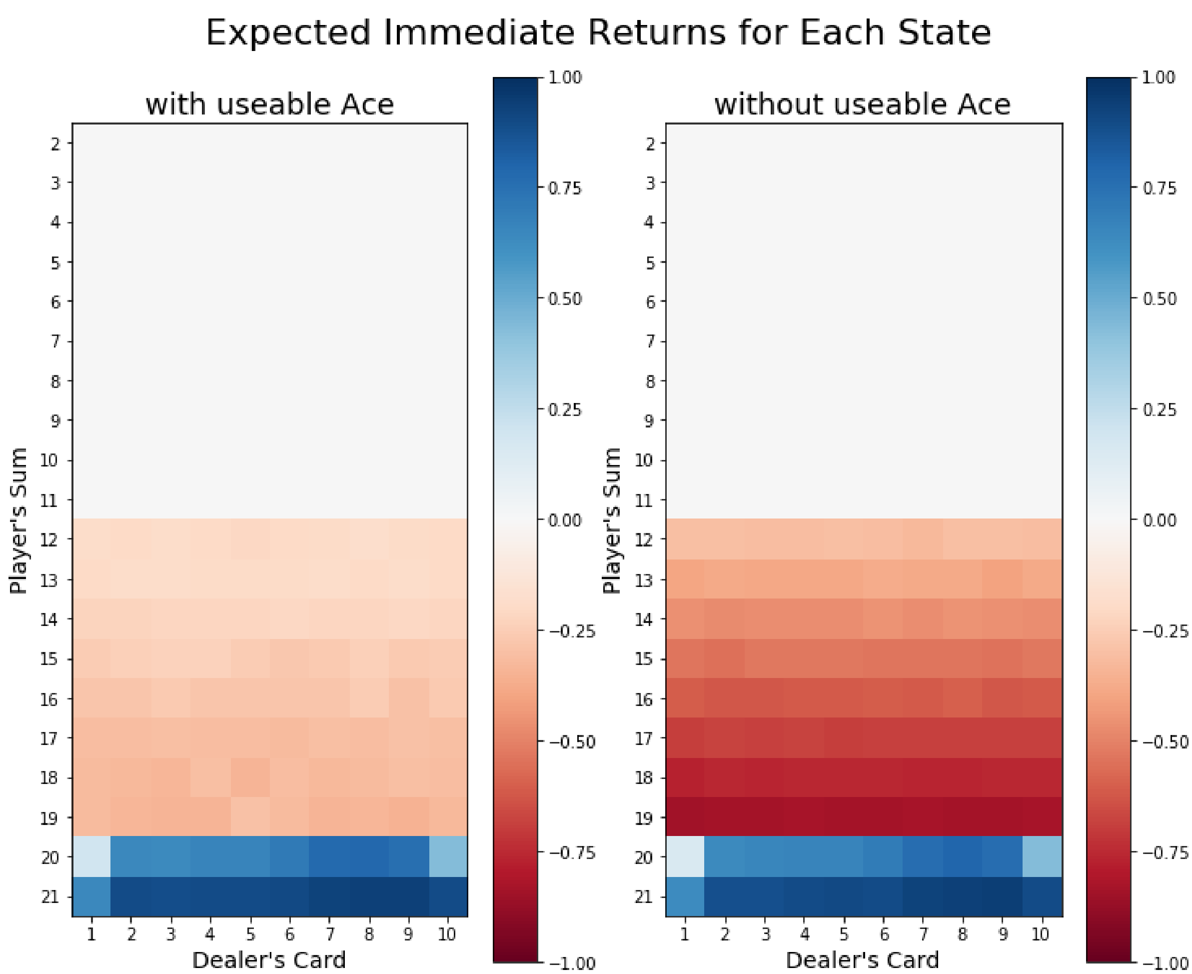
In [6]:

```python
import matplotlib.pyplot as plt
%matplotlib inline

def iter_play(N, policy):
    '''
    x - dealer: 1 - 10
    y - player: 2 - 21

    assgine immediate rewards to last state
    '''

    expected_return_N = np.zeros((20, 10), dtype=int)
    expected_return_N_c = np.zeros((20, 10), dtype=int)
    expected_return_A = np.zeros((20, 10), dtype=int)
    expected_return_A_c = np.zeros((20, 10), dtype=int)

    for i in range(N):
        done = False
        game = Blackjack()
        s = game.deal()
        player_i = s[0] - 2
        dealer_i = s[1] - 1
        ace = s[2]

        while not done:
            s, reward, done = game.step(policy(s))

            # useable Ace
            if ace:
                expected_return_A[player_i, dealer_i] += reward
                expected_return_A_c[player_i, dealer_i] += 1

            # no useable Ace
            else:
                expected_return_N[player_i, dealer_i] += reward
                expected_return_N_c[player_i, dealer_i] += 1

            player_i, dealer_i = s[0]-2, s[1]-1

    expected_return_A_c[expected_return_A_c == 0] = 1
    expected_return_N_c[expected_return_N_c == 0] = 1

    return np.divide(expected_return_A, expected_return_A_c), np.divide(
expected_return_N, expected_return_N_c)
```

In [7]:
```
expected_returns = iter_play(N = 10000, policy=aggressive_policy)
titles = ["with useable Ace", "without useable Ace"]

plt.figure(figsize=(12,10))
plt.subplot(1,2,1)
plt.imshow(expected_returns[0], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[0], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.subplot(1,2,2)
plt.imshow(expected_returns[1], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[1], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.suptitle("Expected Immediate Returns for Each State", y=0.93, fontsi
ze=22)
plt.show()
```



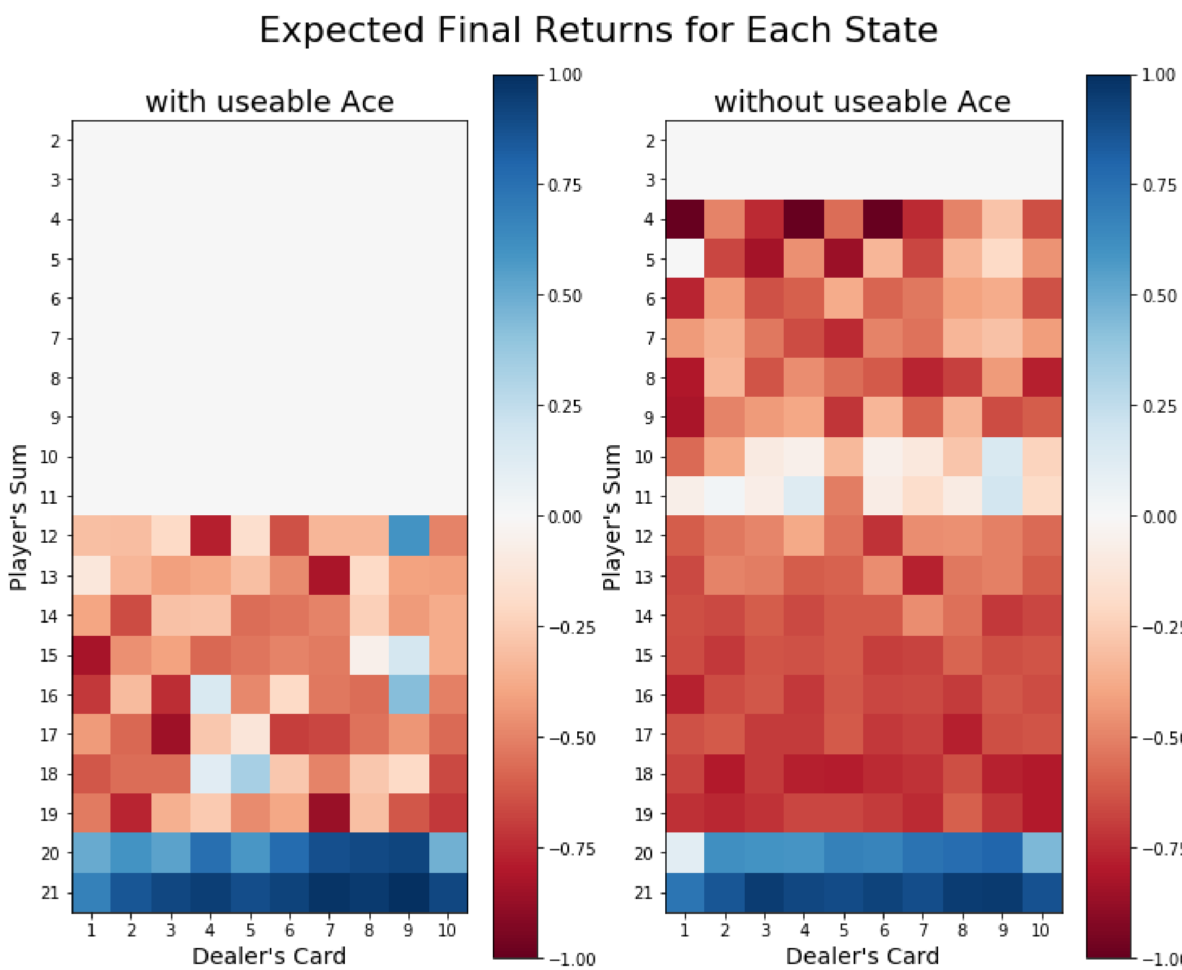
In [8]:
```
expected_returns = iter_play(N = 500000, policy=aggressive_policy)
titles = ["with useable Ace", "without useable Ace"]

plt.figure(figsize=(12,10))
plt.subplot(1,2,1)
plt.imshow(expected_returns[0], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[0], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.subplot(1,2,2)
plt.imshow(expected_returns[1], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[1], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.suptitle("Expected Immediate Returns for Each State", y=0.93, fontsi
ze=22)
plt.show()
```



These plots above were created by assigning the immediate returns to each state. In our virtual Black Jack simulator, the immediate return is set to 0 when you hit and the game does not end. That's why all the states value are 0 when your sum is smaller than 12 without a useable Ace, Because in this policy you will always hit but the game will not end. To address this problem, and gain the knowledge of the real long-term returns. I also tried to assign the final return to every state came through during each episode. The results are shown below.

In [9]:
```python
import matplotlib.pyplot as plt
%matplotlib inline

def iter_play(N, policy):
    '''
    x - dealer: 1 - 10
    y - player: 2 - 21

    assign final returns to every state come across
    '''

    expected_return_N = np.zeros((20, 10), dtype=int)
    expected_return_N_c = np.zeros((20, 10), dtype=int)
    expected_return_A = np.zeros((20, 10), dtype=int)
    expected_return_A_c = np.zeros((20, 10), dtype=int)

    for i in range(N):
        done = False
        states = list()
        game = Blackjack()
        s = game.deal()
        ace = s[2]

        while not done:
            states.append([s[0]-2, s[1]-1])
            s, reward, done = game.step(policy(s))

        # useable Ace
        if ace:
            for player_i, dealer_i in states:
                expected_return_A[player_i, dealer_i] += reward
                expected_return_A_c[player_i, dealer_i] += 1

        # no useable Ace
        else:
            for player_i, dealer_i in states:
                expected_return_N[player_i, dealer_i] += reward
                expected_return_N_c[player_i, dealer_i] += 1

    expected_return_A_c[expected_return_A_c == 0] = 1
    expected_return_N_c[expected_return_N_c == 0] = 1

    return np.divide(expected_return_A, expected_return_A_c), np.divide(
expected_return_N, expected_return_N_c)
```

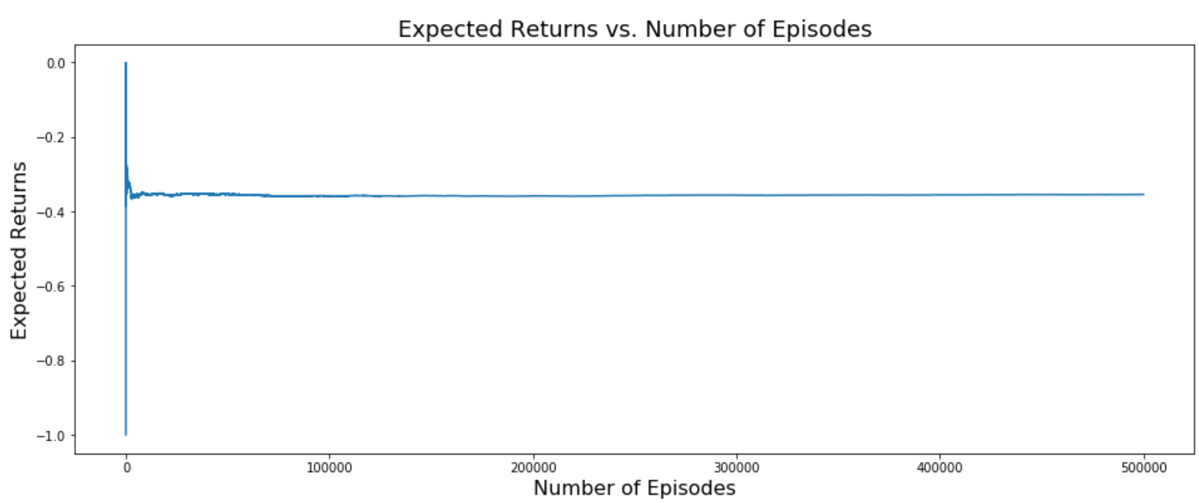
In [10]:
```
expected_returns = iter_play(N = 10000, policy=aggressive_policy)
titles = ["with useable Ace", "without useable Ace"]

plt.figure(figsize=(12,10))
plt.subplot(1,2,1)
plt.imshow(expected_returns[0], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[0], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.subplot(1,2,2)
plt.imshow(expected_returns[1], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[1], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.suptitle("Expected Final Returns for Each State", y=0.93, fontsize=2
2)
plt.show()
```



In [11]:
```
expected_returns = iter_play(N = 500000, policy=aggressive_policy)
titles = ["with useable Ace", "without useable Ace"]

plt.figure(figsize=(12,10))
plt.subplot(1,2,1)
plt.imshow(expected_returns[0], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[0], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.subplot(1,2,2)
plt.imshow(expected_returns[1], vmin=-1, vmax=1, cmap="RdBu")
plt.title(titles[1], fontsize=18)
plt.xlabel("Dealer's Card", fontsize=14)
plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
plt.ylabel("Player's Sum", fontsize=14)
plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
plt.colorbar()

plt.suptitle("Expected Final Returns for Each State", y=0.93, fontsize=2
2)
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]:
```
np.random.seed(323)
N = 500000
rewards = []

for i in range(N):
    done = False
    game = Blackjack()
    s = game.deal()
    player_i = s[0] - 2
    dealer_i = s[1] - 1
    ace = s[2]
    while not done:
        s, reward, done = game.step(aggressive_policy(s))
    rewards.append(reward)
expected_returns = np.cumsum(rewards) / np.arange(1, N+1)

plt.figure(figsize=(16,6))
plt.plot(np.arange(1, N+1), expected_returns, '-')
plt.title("Expected Returns vs. Number of Episodes", fontsize=18)
plt.xlabel("Number of Episodes", fontsize=16)
plt.ylabel("Expected Returns", fontsize=16)
plt.show()
```



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

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

In [14]:
```
class MCMC_BlackJack():
    '''
    Use Monte Carlo Method to optimize BlackJack strategy

    Policy: For each state, choose the action with greater action_value
 with a e-greedy exploration strategy
    Player_sum: 2 - 21
    Dealer_card: 1 - 10
    Usabel_ace: 0 or 1
    State: tuple (Usable_ace, Player_sum-2, Dealer_card-1)
    Action: Hit (1) or Stay (0)

    '''

    def __init__(self, N=5000, N0=100, tol=1e-3):
        self.state_value = np.zeros((2, 20, 10))
        self.action_value = np.zeros((2, 20, 10, 2))
        self.trial_times = np.zeros((2, 20, 10, 2), dtype=int)
        self.running_returns = []
        self.episode = 0
        self.N = N
        self.N0 = N0
        self.tol = tol


    def get_action(self, state):
        # action_value equals
        if self.action_value[state+(0,)] == self.action_value[state+(1
,)]:
            return np.random.randint(0, 2)
        # return min with epsilon probability
        else:
            epsilon = self.N0 / (self.N0 + np.sum(self.trial_times[state
]))
            r = np.random.random()
            action = np.argmax(self.action_value[state])
            if r < epsilon:
                return int(1-action)
            else:
                return int(action)


    def play_one_game(self):
        '''
        play one game with current policy
        return the states and reward
        '''
        game = Blackjack()
        state_actions = list()
        done = False
        s = game.deal()
        while not done:
            p, d, a = s
            state = (int(a), p-2, d-1)
            action = self.get_action(state)
            state_actions.append(state + (action,))
```

```
            s, reward, done = game.step(action)

        return state_actions, reward


    def policy_eval(self):
        '''
        update state_value and action_value until converge
        '''
        diff = 1
        i = 0
        while diff > self.tol:
            if i % 100 == 0:
                print("iter: %4.d" % i, end="\t")
            before = self.action_value.copy()
            for j in range(self.N):
                state_actions, reward = self.play_one_game()
                self.running_returns.append(reward)
                self.episode += 1
                self.policy_iter(state_actions, reward)
            after = self.action_value.copy()
            diff = np.linalg.norm(after - before)
            print(">", end="")
            if i % 100 == 99:
                print("\tdiff: %.6f" % diff)
            i += 1

        #return None


    def policy_iter(self, state_actions, reward):
        '''
        update policy by updating the action_value using online average
 returns
        '''

        for state_action in state_actions:
            # update trial_times
            N = self.trial_times[state_action]
            self.trial_times[state_action] += 1
            # update action_value
            self.action_value[state_action] = N/(N+1) * self.action_valu
e[state_action] + reward / (N+1)


    def plot_policy(self):
        self.state_value = np.max(self.action_value, axis=-1)
        policy = np.argmax(self.action_value, axis=-1)

        # state_value with usable ace
        plt.figure(figsize=(12,20))
        plt.subplot(2,2,1)
        plt.imshow(self.state_value[1], vmin=-1, vmax=1, cmap="RdBu")
        plt.title("State Value with Usable Ace", fontsize=18)
        plt.xlabel("Dealer's Card", fontsize=14)
        plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
        plt.ylabel("Player's Sum", fontsize=14)
```

```
        plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
        plt.colorbar()

        # state_value without usable ace
        plt.subplot(2,2,2)
        plt.imshow(self.state_value[0], vmin=-1, vmax=1, cmap="RdBu")
        plt.title("State Value without Usable Ace", fontsize=18)
        plt.xlabel("Dealer's Card", fontsize=14)
        plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
        plt.ylabel("Player's Sum", fontsize=14)
        plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
        plt.colorbar()

        # optimized policy with usable ace
        plt.subplot(2,2,3)
        plt.imshow(policy[1], vmin=-0.2, vmax=1.2, cmap="RdBu")
        plt.title("Optimized Policy with Usable Ace", fontsize=18)
        plt.xlabel("Dealer's Card\nRed: Stay (0)\nBlue: Hit (1)", fontsi
ze=14)
        plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
        plt.ylabel("Player's Sum", fontsize=14)
        plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
        plt.colorbar()

        # optimized policy without usable ace
        plt.subplot(2,2,4)
        plt.imshow(policy[0], vmin=-0.2, vmax=1.2, cmap="RdBu")
        plt.title("Optimized Policy without Usable Ace", fontsize=18)
        plt.xlabel("Dealer's Card\nRed: Stay (0)\nBlue: Hit (1)", fontsi
ze=14)
        plt.xticks(np.arange(10), list(map(str, np.arange(1,11))))
        plt.ylabel("Player's Sum", fontsize=14)
        plt.yticks(np.arange(20), list(map(str, np.arange(2,22))))
        plt.colorbar()

        plt.suptitle("Expected Final Returns for Each State", y=0.93, fo
ntsize=22)
        plt.show()


    def plot_running_returns(self):
        X = np.arange(1, 1 + self.episode)
        running_avg_returns = np.cumsum(self.running_returns) / X
        plt.figure(figsize=(16,6))
        plt.plot(X, running_avg_returns, "-")
        plt.title("Overall Avg. Returns vs. Number of Episodes", fontsiz
e=18)
        plt.xlabel("Number of Episodes", fontsize=16)
        plt.ylabel("Overall Avg. Returns", fontsize=16)
        plt.show()
```

In [15]:

```
np.random.seed(323)
BlackJack_op = MCMC_BlackJack(tol=5e-4)
BlackJack_op.policy_eval()
```

```
iter:    0      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.102797
iter:  100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.053597
iter:  200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.032656
iter:  300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.025734
iter:  400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.020679
iter:  500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.015720
iter:  600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.012325
iter:  700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.013081
iter:  800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.013733
iter:  900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.014254
iter: 1000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.008218
iter: 1100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.009335
iter: 1200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.008608
iter: 1300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.009055
iter: 1400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.007406
iter: 1500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.006586
iter: 1600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.005859
iter: 1700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004983
iter: 1800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.008136
iter: 1900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.007559
iter: 2000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.006182
iter: 2100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.008282
iter: 2200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.007971
iter: 2300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004918
iter: 2400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.006864
iter: 2500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.005628
iter: 2600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004446
iter: 2700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.005285
iter: 2800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

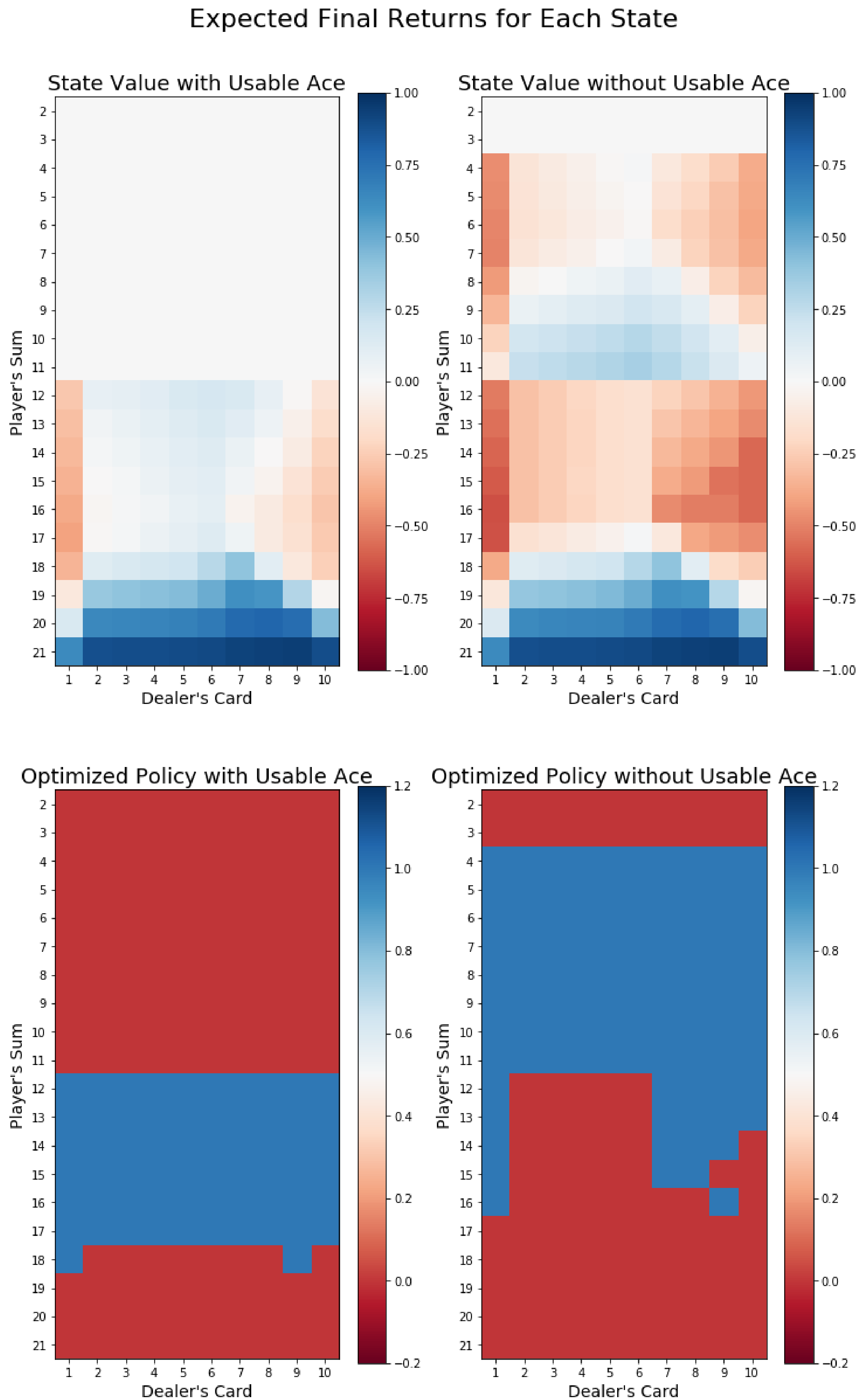
```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003813
iter: 2900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003401
iter: 3000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004696
iter: 3100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004213
iter: 3200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004301
iter: 3300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.005836
iter: 3400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003434
iter: 3500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004450
iter: 3600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004226
iter: 3700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.005106
iter: 3800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002577
iter: 3900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002212
iter: 4000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004530
iter: 4100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002804
iter: 4200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003776
iter: 4300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004950
iter: 4400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001969
iter: 4500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002655
iter: 4600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004097
iter: 4700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002931
iter: 4800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001687
iter: 4900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003450
iter: 5000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002708
iter: 5100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004486
iter: 5200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002799
iter: 5300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.004579
iter: 5400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003706
iter: 5500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003461
iter: 5600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002149
```

```
iter: 5700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003649
iter: 5800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001695
iter: 5900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002371
iter: 6000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003843
iter: 6100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001630
iter: 6200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003233
iter: 6300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002228
iter: 6400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003040
iter: 6500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001441
iter: 6600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002054
iter: 6700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002406
iter: 6800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002972
iter: 6900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002513
iter: 7000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002988
iter: 7100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002935
iter: 7200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001854
iter: 7300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001098
iter: 7400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001656
iter: 7500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003359
iter: 7600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001656
iter: 7700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002878
iter: 7800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002529
iter: 7900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002668
iter: 8000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001456
iter: 8100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.000799
iter: 8200      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002405
iter: 8300      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002348
iter: 8400      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002384
iter: 8500      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.003013
iter: 8600      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002066
iter: 8700      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001828
iter: 8800      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001734
iter: 8900      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.001905
iter: 9000      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   diff: 0.002028
iter: 9100      >>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

In [16]:

```
BlackJack_op.plot_policy()
```

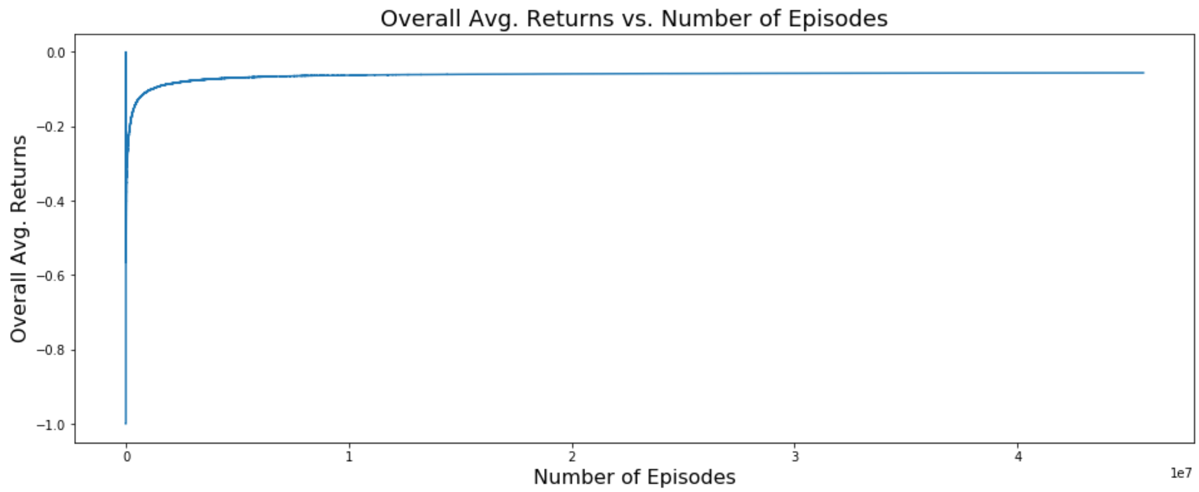

Expected Final Returns for Each State
State Value with Usable Ace
State Value without Usable Ace
Optimized Policy with Usable Ace
Optimized Policy without Usable Ace
Player's Sum
Dealer's Card

Red: Stay (0)
Blue: Hit (1)

Red: Stay (0)
Blue: Hit (1)

(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?

In [17]:
```
BlackJack_op.plot_running_returns()
```



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

(a) For a long term perspective, the optimal policy derived from question 3 has the highest average returns. Using Monte Carlo Control, we could estimate the the expected returns for each action made at each state. This process is accomplished by playing large amount of games and recording their returns. By simulating large amount of games, the running average will be an unbiased estimation of the real expection for returns. The optimal policy we used in question 3 is simply choosing the action with a higher extimated action value. Thus, the actions made by this policy will optimize the overall average returns.

(b) It is impossible to have a better policy with the same amount of information. With an unlimited iteration, the estimated action value will be an unbiased estimation of the real expected returns. Making actions based on these action values will maximize the overall average returns.

In this virtual game we cannot know the dealer's card before we finish our action. In the real world, however, if we could somehow gain the knowledge of the hidden card in the dealer's hand. In that situation, the expected returns will be differnt from the current one. We could take advantage of this extra information and accomplish a better policy (than the current one) with a higher overall average returns.