Monte_Carlo_Solution

August 11, 2018

1 Mini Project: Monte Carlo Methods

In this notebook, you will write your own implementations of many Monte Carlo (MC) algorithms.

While we have provided some starter code, you are welcome to erase these hints and write your code from scratch.

1.0.1 Part 0: Explore BlackjackEnv

Use the code cell below to create an instance of the Blackjack environment.

```
In [1]: import gym
        env = gym.make('Blackjack-v0')
```

```
[2018-01-24 22:30:07,814] Making new env: Blackjack-v0
```

Each state is a 3-tuple of: - the player's current sum $\in \{0, 1, \ldots, 31\}$, - the dealer's face up card $\in \{1, \ldots, 10\}$, and - whether or not the player has a usable ace (`no` $= 0$, `yes` $= 1$).

The agent has two potential actions:

```
STICK = 0
HIT = 1
```

Verify this by running the code cell below.

```
In [2]: print(env.observation_space)
        print(env.action_space)
```

```
Tuple(Discrete(32), Discrete(11), Discrete(2))
Discrete(2)
```

Execute the code cell below to play Blackjack with a random policy.

(*The code currently plays Blackjack three times - feel free to change this number, or to run the cell multiple times. The cell is designed for you to get some experience with the output that is returned as the agent interacts with the environment.*)

```
In [3]: for i_episode in range(3):
            state = env.reset()
            while True:
                print(state)
                action = env.action_space.sample()
                state, reward, done, info = env.step(action)
                if done:
                    print('End game! Reward: ', reward)
                    print('You won :)\n') if reward > 0 else print('You lost :(\n')
                    break
```

```
(8, 5, False)
End game! Reward:  -1.0
You lost :(

(16, 8, False)
End game! Reward:  -1
You lost :(

(13, 10, False)
End game! Reward:  -1
You lost :(
```

1.0.2 Part 1: MC Prediction: State Values

In this section, you will write your own implementation of MC prediction (for estimating the state-value function).

We will begin by investigating a policy where the player always sticks if the sum of her cards exceeds 18. The function `generate_episode_from_limit` samples an episode using this policy.

The function accepts as **input**: - `bj_env`: This is an instance of OpenAI Gym's Blackjack environment.

It returns as **output**: - `episode`: This is a list of (state, action, reward) tuples (of tuples) and corresponds to $(S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T)$, where T is the final time step. In particular, `episode[i]` returns (S_i, A_i, R_{i+1}), and `episode[i][0]`, `episode[i][1]`, and `episode[i][2]` return S_i, A_i, and R_{i+1}, respectively.

```
In [4]: def generate_episode_from_limit(bj_env):
            episode = []
            state = bj_env.reset()
            while True:
                action = 0 if state[0] > 18 else 1
                next_state, reward, done, info = bj_env.step(action)
                episode.append((state, action, reward))
                state = next_state
                if done:
                    break
            return episode
```

Execute the code cell below to play Blackjack with the policy.

(*The code currently plays Blackjack three times - feel free to change this number, or to run the cell multiple times. The cell is designed for you to gain some familiarity with the output of the* `generate_episode_from_limit` *function.*)

```
In [5]: for i in range(3):
            print(generate_episode_from_limit(env))
```

```
[((14, 1, False), 1, -1)]
[((14, 10, False), 1, -1)]
[((11, 2, False), 1, 0), ((21, 2, False), 0, 1.0)]
```

Now, you are ready to write your own implementation of MC prediction. Feel free to implement either first-visit or every-visit MC prediction; in the case of the Blackjack environment, the techniques are equivalent.

Your algorithm has three arguments: - `env`: This is an instance of an OpenAI Gym environment. - `num_episodes`: This is the number of episodes that are generated through agent-environment interaction. - `generate_episode`: This is a function that returns an episode of interaction. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as output: - `V`: This is a dictionary where `V[s]` is the estimated value of state `s`. For example, if your code returns the following output:

```
{(4, 7, False): -0.38775510204081631, (18, 6, False): -0.58434296365330851, (13, 2, False): -0.4
```

then the value of state `(4, 7, False)` was estimated to be `-0.38775510204081631`.

If you are unfamiliar with how to use `defaultdict` in Python, you are encouraged to check out this source.

```
In [6]: from collections import defaultdict
        import numpy as np
        import sys

        def mc_prediction_v(env, num_episodes, generate_episode, gamma=1.0):
            # initialize empty dictionary of lists
            returns = defaultdict(list)
            # loop over episodes
            for i_episode in range(1, num_episodes+1):
                # monitor progress
                if i_episode % 1000 == 0:
                    print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
                    sys.stdout.flush()
                # generate an episode
                episode = generate_episode(env)
                # obtain the states, actions, and rewards
                states, actions, rewards = zip(*episode)
                # prepare for discounting
                discounts = np.array([gamma**i for i in range(len(rewards)+1)])
```

```
            # calculate and store the return for each visit in the episode
            for i, state in enumerate(states):
                returns[state].append(sum(rewards[i:]*discounts[:-(1+i)]))
        # calculate the state-value function estimate
        V = {k: np.mean(v) for k, v in returns.items()}
        return V
```

Use the cell below to calculate and plot the state-value function estimate. (*The code for plotting the value function has been borrowed from this source and slightly adapted.*)

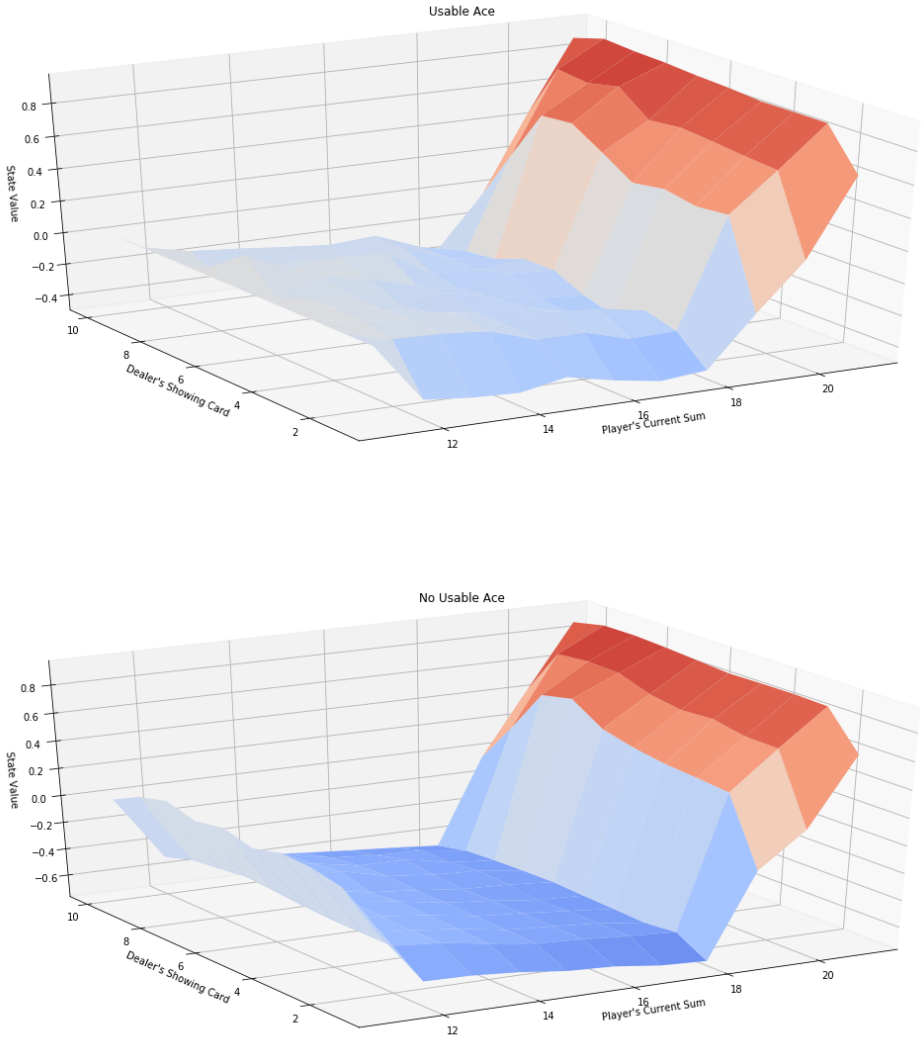
To check the accuracy of your implementation, compare the plot below to the corresponding plot in the solutions notebook **Monte_Carlo_Solution.ipynb**.

In [7]:
```
from plot_utils import plot_blackjack_values

# obtain the value function
V = mc_prediction_v(env, 500000, generate_episode_from_limit)

# plot the value function
plot_blackjack_values(V)
```

```
Episode 500000/500000.
```



1.0.3 Part 2: MC Prediction: Action Values

In this section, you will write your own implementation of MC prediction (for estimating the action-value function).

We will begin by investigating a policy where the player *almost* always sticks if the sum of her cards exceeds 18. In particular, she selects action `STICK` with 80% probability if the sum is greater than 18; and, if the sum is 18 or below, she selects action `HIT` with 80% probability. The function `generate_episode_from_limit_stochastic` samples an episode using this policy.

The function accepts as **input**: - `bj_env`: This is an instance of OpenAI Gym's Blackjack environment.

It returns as **output**: - `episode`: This is a list of (state, action, reward) tuples (of tuples) and corresponds to $(S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T)$, where T is the final time step. In particular, `episode[i]` returns (S_i, A_i, R_{i+1}), and `episode[i][0]`, `episode[i][1]`, and `episode[i][2]` return S_i, A_i, and R_{i+1}, respectively.

```
In [8]: def generate_episode_from_limit_stochastic(bj_env):
            episode = []
            state = bj_env.reset()
            while True:
                probs = [0.8, 0.2] if state[0] > 18 else [0.2, 0.8]
                action = np.random.choice(np.arange(2), p=probs)
                next_state, reward, done, info = bj_env.step(action)
                episode.append((state, action, reward))
                state = next_state
                if done:
                    break
            return episode
```

Now, you are ready to write your own implementation of MC prediction. Feel free to implement either first-visit or every-visit MC prediction; in the case of the Blackjack environment, the techniques are equivalent.

Your algorithm has three arguments: - `env`: This is an instance of an OpenAI Gym environment. - `num_episodes`: This is the number of episodes that are generated through agent-environment interaction. - `generate_episode`: This is a function that returns an episode of interaction. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as output: - `Q`: This is a dictionary (of one-dimensional arrays) where `Q[s][a]` is the estimated action value corresponding to state `s` and action `a`.

```
In [9]: def mc_prediction_q(env, num_episodes, generate_episode, gamma=1.0):
            # initialize empty dictionaries of arrays
            returns_sum = defaultdict(lambda: np.zeros(env.action_space.n))
            N = defaultdict(lambda: np.zeros(env.action_space.n))
            Q = defaultdict(lambda: np.zeros(env.action_space.n))
            # loop over episodes
            for i_episode in range(1, num_episodes+1):
                # monitor progress
                if i_episode % 1000 == 0:
                    print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
                    sys.stdout.flush()
                # generate an episode
                episode = generate_episode(env)
                # obtain the states, actions, and rewards
                states, actions, rewards = zip(*episode)
                # prepare for discounting
                discounts = np.array([gamma**i for i in range(len(rewards)+1)])
                # update the sum of the returns, number of visits, and action-value
                # function estimates for each state-action pair in the episode
                for i, state in enumerate(states):
                    returns_sum[state][actions[i]] += sum(rewards[i:]*discounts[:-(1+i)])
                    N[state][actions[i]] += 1.0
                    Q[state][actions[i]] = returns_sum[state][actions[i]] / N[state][actions[i]]
            return Q
```

Use the cell below to obtain the action-value function estimate Q. We have also plotted the corresponding state-value function.

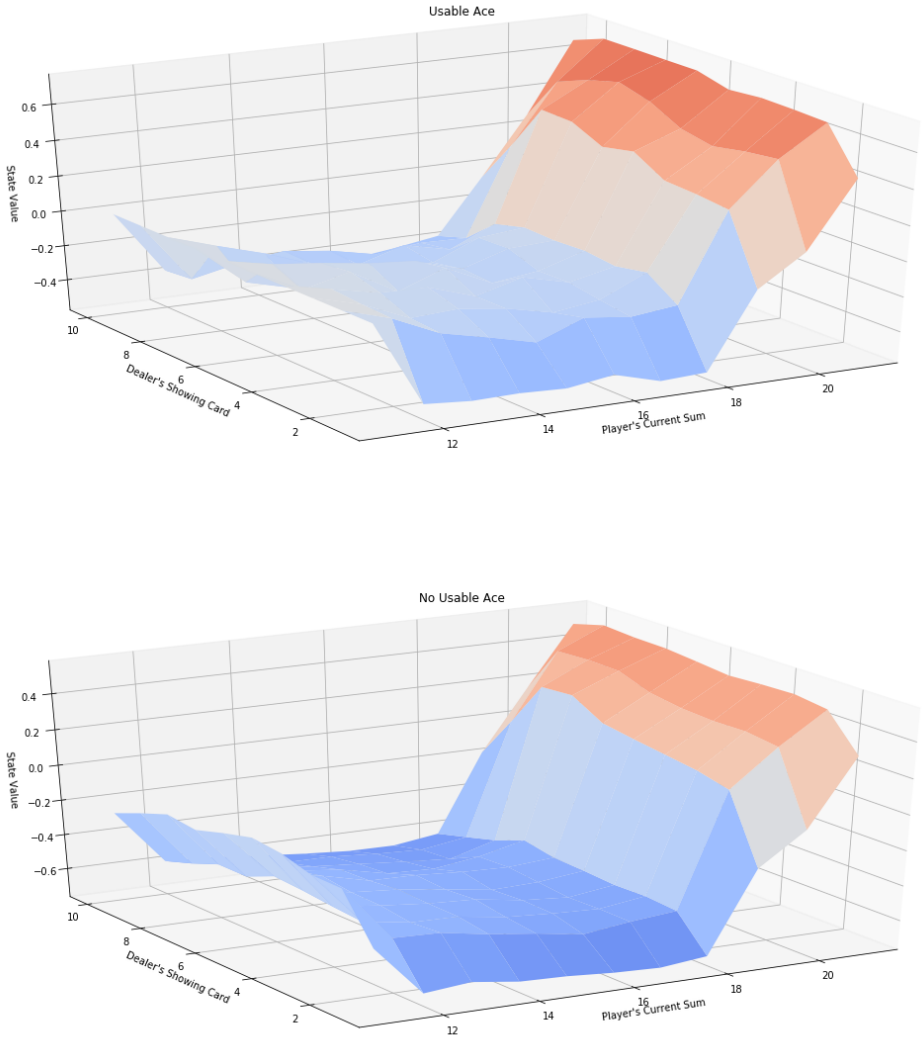
To check the accuracy of your implementation, compare the plot below to the corresponding plot in the solutions notebook **Monte_Carlo_Solution.ipynb**.

```
In [10]: # obtain the action-value function
         Q = mc_prediction_q(env, 500000, generate_episode_from_limit_stochastic)

         # obtain the state-value function
         V_to_plot = dict((k,(k[0]>18)*(np.dot([0.8, 0.2],v)) + (k[0]<=18)*(np.dot([0.2, 0.8],v)
                  for k, v in Q.items())

         # plot the state-value function
         plot_blackjack_values(V_to_plot)
```

```
Episode 500000/500000.
```



1.0.4 Part 3: MC Control: GLIE

In this section, you will write your own implementation of constant-α MC control.

Your algorithm has three arguments: - `env`: This is an instance of an OpenAI Gym environment. - `num_episodes`: This is the number of episodes that are generated through agent-environment interaction. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as output: - `Q`: This is a dictionary (of one-dimensional arrays) where `Q[s][a]` is the estimated action value corresponding to state `s` and action `a`. - `policy`: This is a dictionary where `policy[s]` returns the action that the agent chooses after observing state `s`.

(*Feel free to define additional functions to help you to organize your code.*)

```
In [11]: def generate_episode_from_Q(env, Q, epsilon, nA):
             """ generates an episode from following the epsilon-greedy policy """
             episode = []
             state = env.reset()
             while True:
                 action = np.random.choice(np.arange(nA), p=get_probs(Q[state], epsilon, nA)) \
                                         if state in Q else env.action_space.sample()
                 next_state, reward, done, info = env.step(action)
                 episode.append((state, action, reward))
                 state = next_state
                 if done:
                     break
             return episode

         def get_probs(Q_s, epsilon, nA):
             """ obtains the action probabilities corresponding to epsilon-greedy policy """
             policy_s = np.ones(nA) * epsilon / nA
             best_a = np.argmax(Q_s)
             policy_s[best_a] = 1 - epsilon + (epsilon / nA)
             return policy_s

         def update_Q_GLIE(env, episode, Q, N, gamma):
             """ updates the action-value function estimate using the most recent episode """
             states, actions, rewards = zip(*episode)
             # prepare for discounting
             discounts = np.array([gamma**i for i in range(len(rewards)+1)])
             for i, state in enumerate(states):
                 old_Q = Q[state][actions[i]]
                 old_N = N[state][actions[i]]
                 Q[state][actions[i]] = old_Q + (sum(rewards[i:]*discounts[:-(1+i)]) - old_Q)/(o
                 N[state][actions[i]] += 1
             return Q, N
```

```
In [12]: def mc_control_GLIE(env, num_episodes, gamma=1.0):
             nA = env.action_space.n
             # initialize empty dictionaries of arrays
             Q = defaultdict(lambda: np.zeros(nA))
             N = defaultdict(lambda: np.zeros(nA))
             # loop over episodes
             for i_episode in range(1, num_episodes+1):
                 # monitor progress
                 if i_episode % 1000 == 0:
                     print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
                     sys.stdout.flush()
                 # set the value of epsilon
                 epsilon = 1.0/((i_episode/8000)+1)
                 # generate an episode by following epsilon-greedy policy
                 episode = generate_episode_from_Q(env, Q, epsilon, nA)
                 # update the action-value function estimate using the episode
                 Q, N = update_Q_GLIE(env, episode, Q, N, gamma)
             # determine the policy corresponding to the final action-value function estimate
             policy = dict((k,np.argmax(v)) for k, v in Q.items())
             return policy, Q
```

Use the cell below to obtain the estimated optimal policy and action-value function.

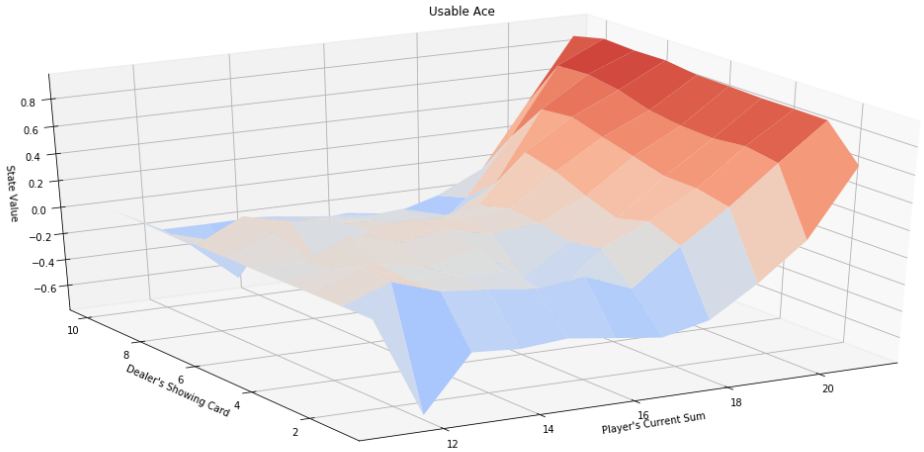
```
In [13]: # obtain the estimated optimal policy and action-value function
         policy_glie, Q_glie = mc_control_GLIE(env, 500000)
```

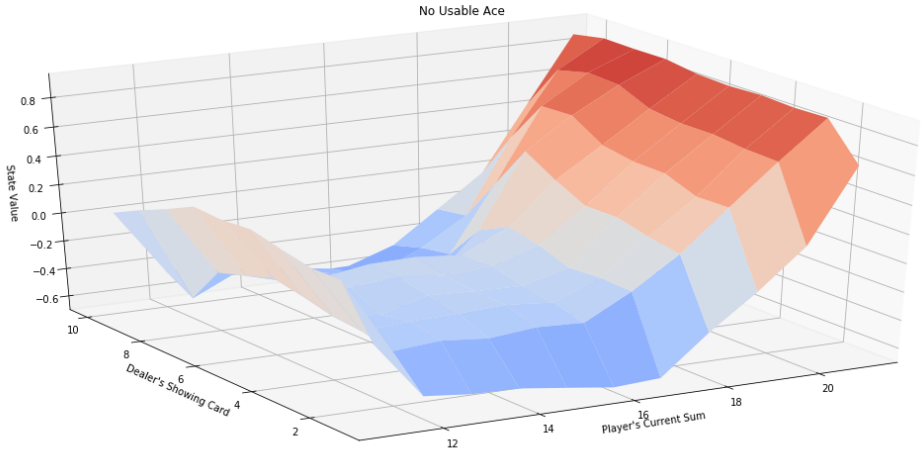
```
Episode 500000/500000.
```

Next, we plot the corresponding state-value function.

```
In [14]: # obtain the state-value function
         V_glie = dict((k,np.max(v)) for k, v in Q_glie.items())

         # plot the state-value function
         plot_blackjack_values(V_glie)
```

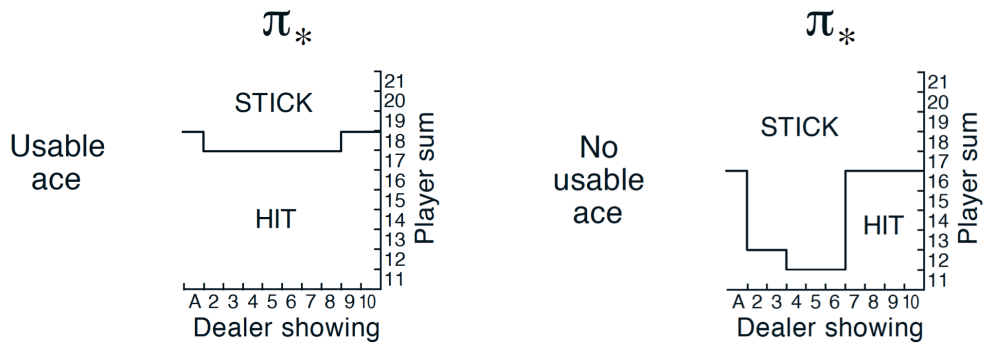




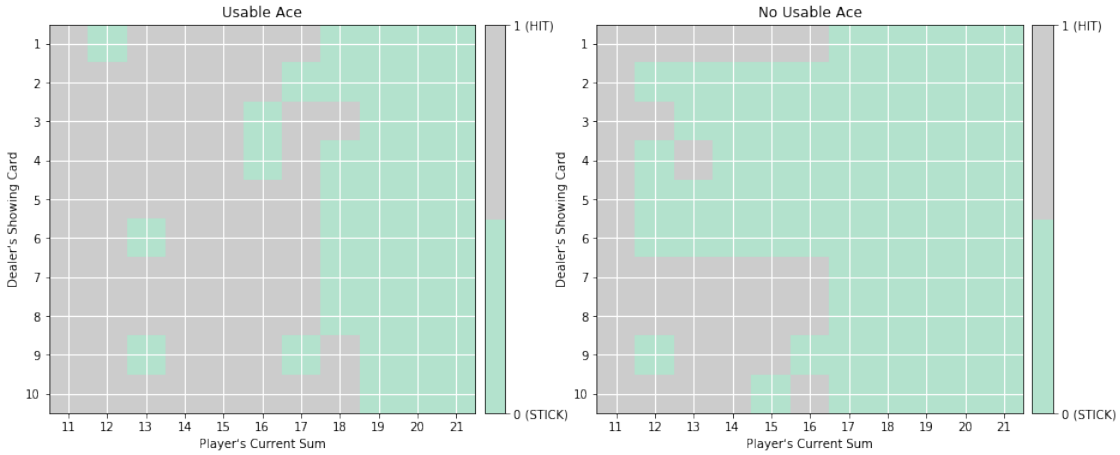
Finally, we visualize the policy that is estimated to be optimal.

```
In [15]: from plot_utils import plot_policy

         # plot the policy
         plot_policy(policy_glie)
```



True Optimal Policy



The **true** optimal policy π_* can be found on page 82 of the textbook (and appears below). Compare your final estimate to the optimal policy - how close are you able to get? If you are not happy with the performance of your algorithm, take the time to tweak the decay rate of ϵ and/or run the algorithm for more episodes to attain better results.

1.0.5 Part 4: MC Control: Constant-α

In this section, you will write your own implementation of constant-α MC control.

Your algorithm has four arguments: - `env`: This is an instance of an OpenAI Gym environment. - `num_episodes`: This is the number of episodes that are generated through agent-environment interaction. - `alpha`: This is the step-size parameter for the update step. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as output: - `Q`: This is a dictionary (of one-dimensional arrays) where `Q[s][a]` is the estimated action value corresponding to state `s` and action `a`. - `policy`: This is a dictionary where `policy[s]` returns the action that the agent chooses after observing state `s`.

(*Feel free to define additional functions to help you to organize your code.*)

```
In [16]: def update_Q_alpha(env, episode, Q, alpha, gamma):
             """ updates the action-value function estimate using the most recent episode """
```

```
        states, actions, rewards = zip(*episode)
        # prepare for discounting
        discounts = np.array([gamma**i for i in range(len(rewards)+1)])
        for i, state in enumerate(states):
            old_Q = Q[state][actions[i]]
            Q[state][actions[i]] = old_Q + alpha*(sum(rewards[i:]*discounts[:-(1+i)]) - old
        return Q
```

```
In [17]: def mc_control_alpha(env, num_episodes, alpha, gamma=1.0):
        nA = env.action_space.n
        # initialize empty dictionary of arrays
        Q = defaultdict(lambda: np.zeros(nA))
        # loop over episodes
        for i_episode in range(1, num_episodes+1):
            # monitor progress
            if i_episode % 1000 == 0:
                print("\rEpisode {}/{}.".format(i_episode, num_episodes), end="")
                sys.stdout.flush()
            # set the value of epsilon
            epsilon = 1.0/((i_episode/8000)+1)
            # generate an episode by following epsilon-greedy policy
            episode = generate_episode_from_Q(env, Q, epsilon, nA)
            # update the action-value function estimate using the episode
            Q = update_Q_alpha(env, episode, Q, alpha, gamma)
        # determine the policy corresponding to the final action-value function estimate
        policy = dict((k,np.argmax(v)) for k, v in Q.items())
        return policy, Q
```

Use the cell below to obtain the estimated optimal policy and action-value function.

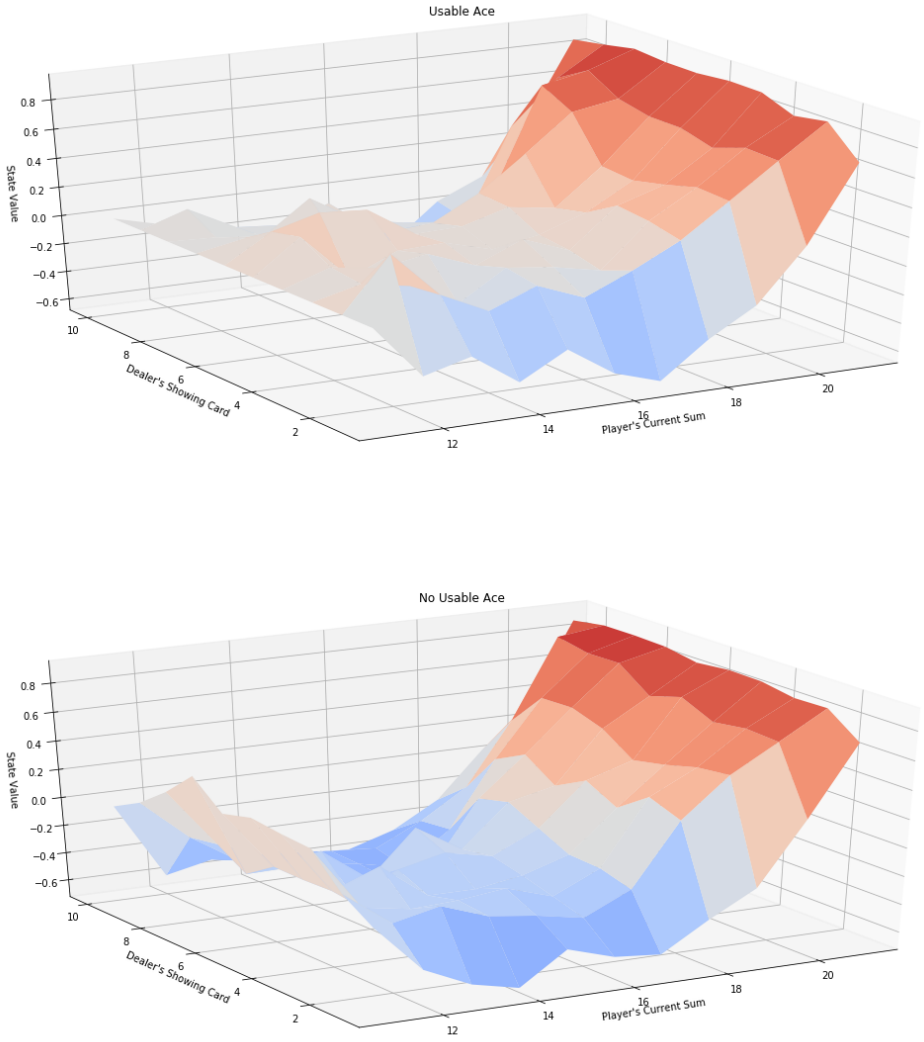
```
In [18]: # obtain the estimated optimal policy and action-value function
         policy_alpha, Q_alpha = mc_control_alpha(env, 500000, 0.02)
```

```
Episode 500000/500000.
```

Next, we plot the corresponding state-value function.

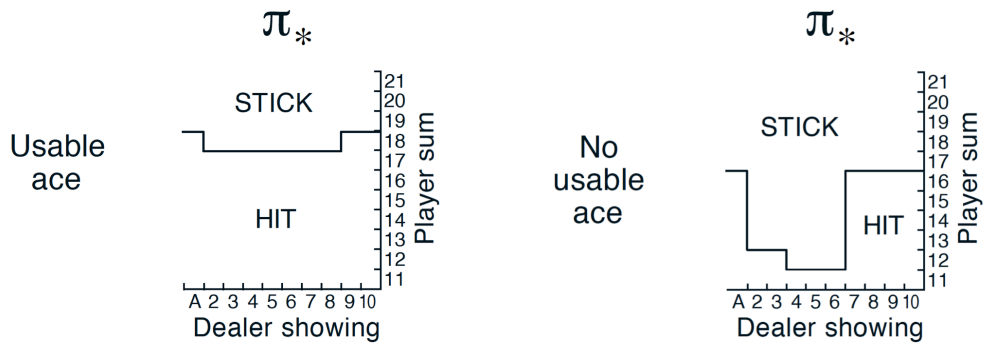
```
In [19]: # obtain the state-value function
         V_alpha = dict((k,np.max(v)) for k, v in Q_alpha.items())

         # plot the state-value function
         plot_blackjack_values(V_alpha)
```

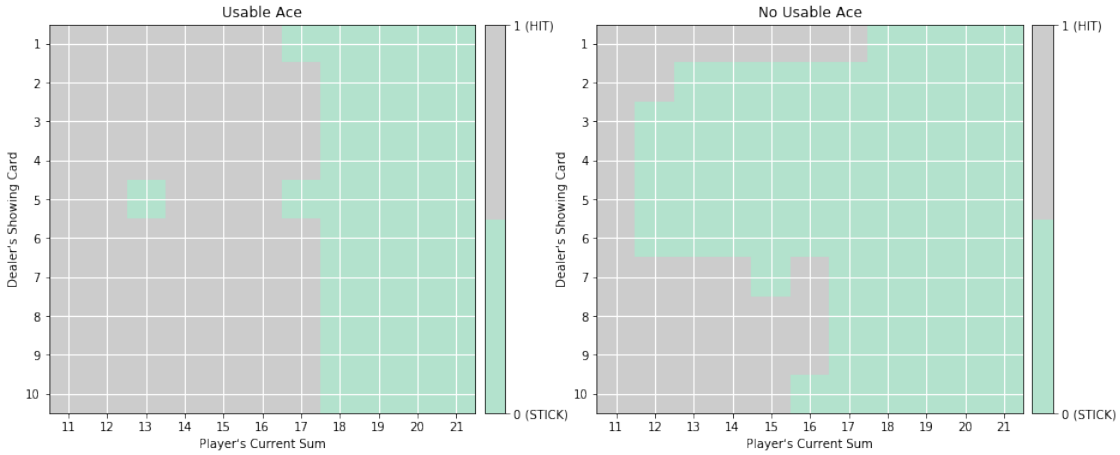


Finally, we visualize the policy that is estimated to be optimal.

```
In [20]: # plot the policy
         plot_policy(policy_alpha)
```



True Optimal Policy



The **true** optimal policy π_* can be found on page 82 of the textbook (and appears below). Compare your final estimate to the optimal policy - how close are you able to get? If you are not happy with the performance of your algorithm, take the time to tweak the decay rate of ϵ, change the value of α, and/or run the algorithm for more episodes to attain better results.