Monte_Carlo

August 20, 2025

1 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

We begin by importing the necessary packages.

```
In [1]: import sys
        import gym
        import numpy as np
        from collections import defaultdict

        from plot_utils import plot_blackjack_values, plot_policy
```

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

```
In [2]: env = gym.make('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 [3]: 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 [17]: 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
```

```
(13, 10, False)
(20, 10, False)
End game! Reward:  0.0
You lost :(

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

(17, 10, True)
End game! Reward:  -1.0
You lost :(
```

1.0.2 Part 1: MC Prediction

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 [12]: 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
```

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_stochastic` *function.*)

```
In [13]: for i in range(3):
             print(generate_episode_from_limit_stochastic(env))

[((11, 10, False), 1, 0), ((12, 10, False), 1, 0), ((16, 10, False), 1, -1)]
[((17, 10, False), 1, -1)]
[((13, 10, False), 1, -1)]
```

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 [15]: 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()
                 episode = generate_episode(env)

                 # Track (state, action)  return
                 G = 0
```

```
            # Traverse episode in reverse for return calculation
            for t in reversed(range(len(episode))):
                state, action, reward = episode[t]
                G = gamma * G + reward

                # Every-visit: update regardless of previous occurrence
                returns_sum[state][action] += G
                N[state][action] += 1
                Q[state][action] = returns_sum[state][action] / N[state][action]
            ## TODO: complete the function

        return Q
```

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

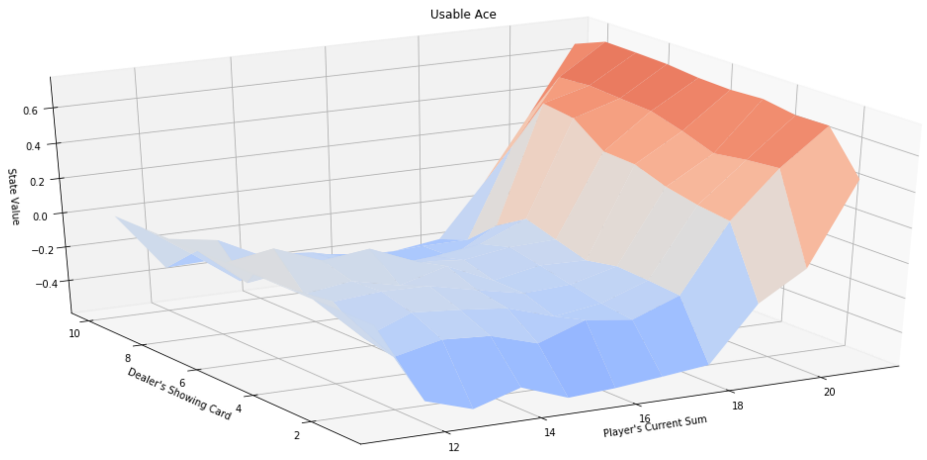
To check the accuracy of your implementation, compare the plot below to the corresponding plot in the solutions notebook **Monte_Carlo_Solution.ipynb**.

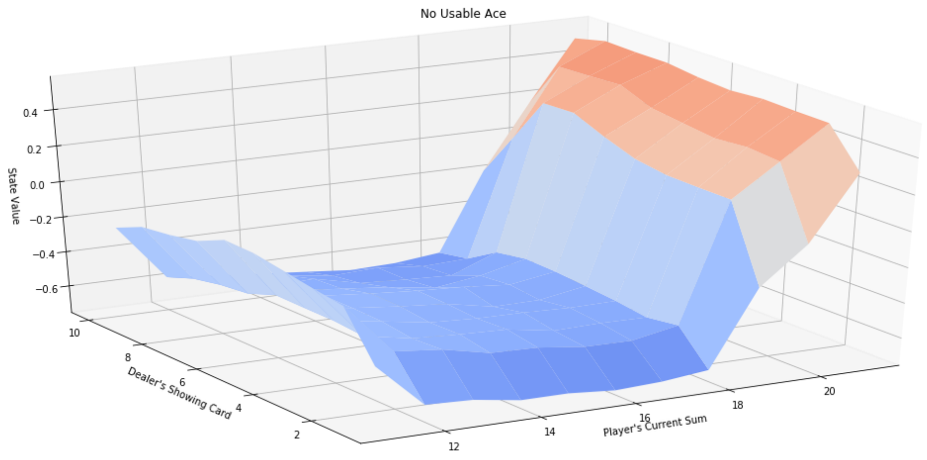
```
In [16]: # obtain the action-value function
         Q = mc_prediction_q(env, 500000, generate_episode_from_limit_stochastic)

         # obtain the corresponding 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.
```





1.0.3 Part 2: MC Control

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 [18]: import numpy as np
         from collections import defaultdict
```

```python
import sys
import gym

def mc_control(env, num_episodes, alpha, gamma=1.0):
    nA = env.action_space.n
    Q = defaultdict(lambda: np.zeros(nA))  # Q[state][action]

    for i_episode in range(1, num_episodes + 1):
        if i_episode % 10000 == 0:
            print(f"\rEpisode {i_episode}/{num_episodes}.", end="")
            sys.stdout.flush()

        # Generate an episode: [(state, action, reward), ...]
        episode = []
        state = env.reset()
        done = False

        while not done:
            probs = epsilon_greedy_policy(Q[state], epsilon=0.1)
            action = np.random.choice(np.arange(nA), p=probs)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            state = next_state

        # Track returns using G
        G = 0
        visited = set()
        for t in reversed(range(len(episode))):
            state, action, reward = episode[t]
            G = gamma * G + reward
            if (state, action) not in visited:
                visited.add((state, action))
                # constant-alpha MC update
                Q[state][action] += alpha * (G - Q[state][action])

    # Build final greedy policy
    policy = {}
    for state in Q:
        policy[state] = np.argmax(Q[state])

    return policy, Q


def epsilon_greedy_policy(action_values, epsilon=0.1):
    nA = len(action_values)
    policy = np.ones(nA) * epsilon / nA
    best_action = np.argmax(action_values)
    policy[best_action] += (1.0 - epsilon)
```

```
        return policy
```

Use the cell below to obtain the estimated optimal policy and action-value function. Note that you should fill in your own values for the num_episodes and alpha parameters.

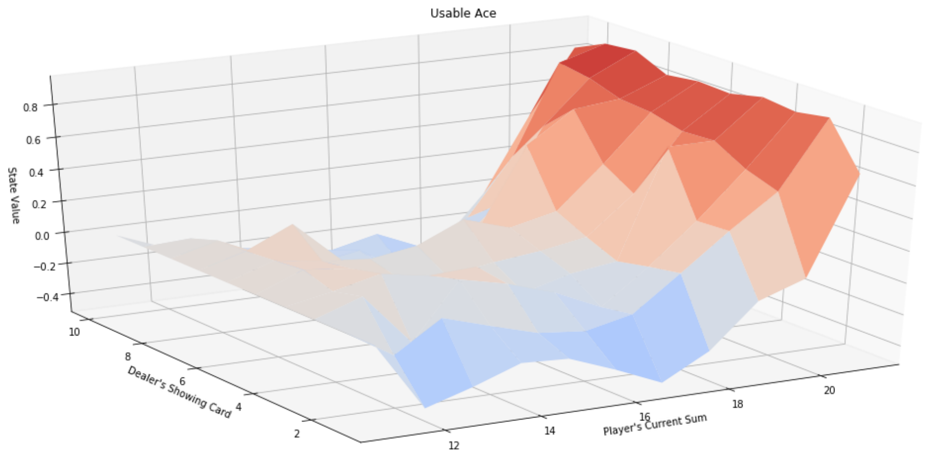
```
In [20]: # obtain the estimated optimal policy and action-value function
         policy, Q = mc_control(env, num_episodes=500000, alpha=0.02)
```

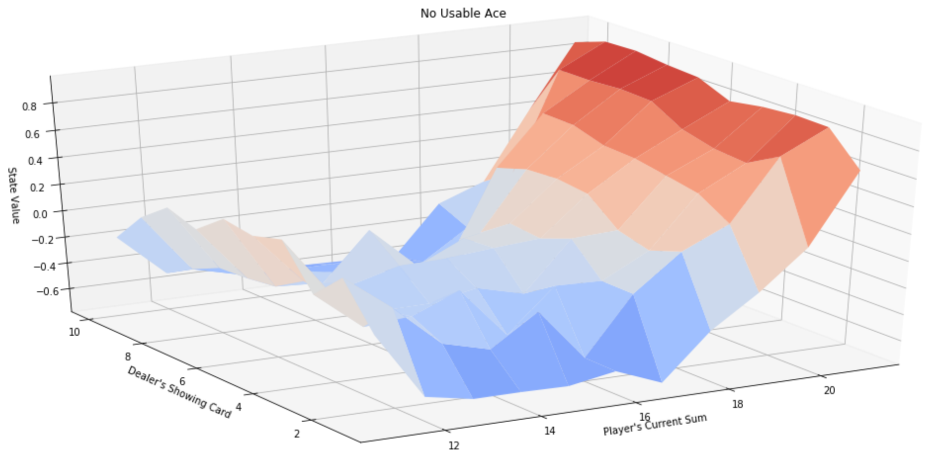
```
Episode 500000/500000.
```

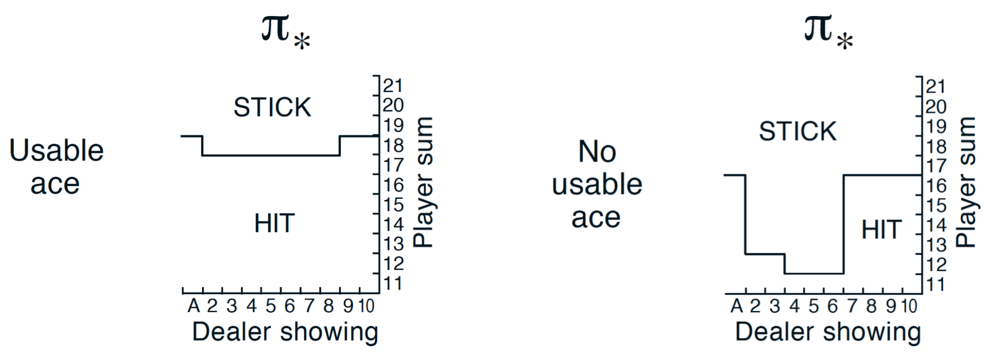
Next, we plot the corresponding state-value function.

```
In [21]: # obtain the corresponding state-value function
         V = dict((k,np.max(v)) for k, v in Q.items())

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



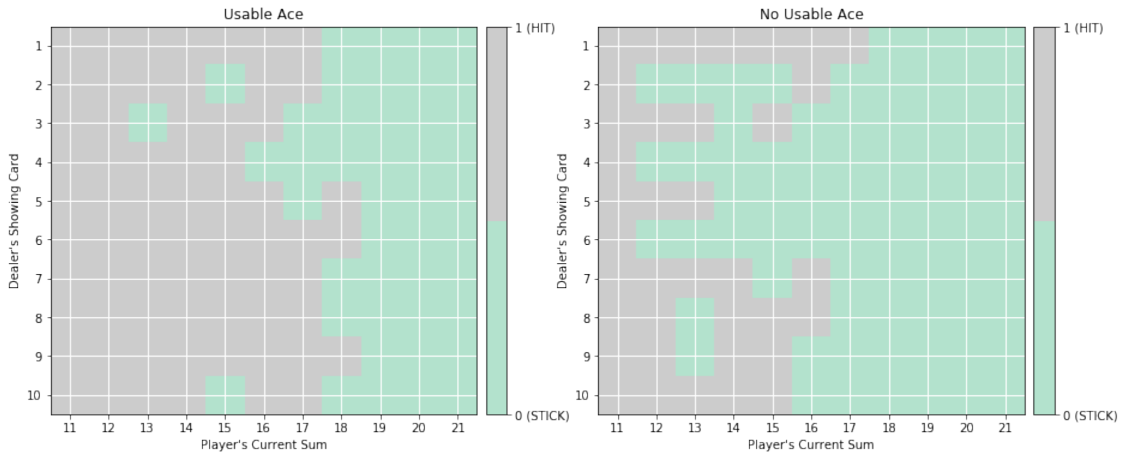




True Optimal Policy

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

```
In [22]: # plot the policy
         plot_policy(policy)
```



The **true** optimal policy π_* can be found in Figure 5.2 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.