Reinforcement Learning: A Literature Review

Zach Rewolinski
Department of Statistics
Rice University
Houston, Texas
zach.rewolinski@rice.edu

Quan Le
Department of Computer Science
Rice University
Houston, Texas 77005-1890
quan.le@rice.edu

***Abstract*—This final project will be centered around the machine learning subfield of reinforcement learning. We will conduct a literature review that introduces fundamental algorithms in reinforcement learning, such as dynamic programming, Monte Carlo methods, and temporal-difference learning. In doing so, we will also critically compare and contrast popular approaches to the reinforcement learning problem, including empirical comparisons of various methods.**

***Index Terms*—reinforcement learning, machine learning, artificial intelligence**

I. Introduction

The area of artificial intelligence is the study of machine intelligence, and in particular, the study of such mechanical agents making decisions to further some goal. Of prominence in artificial intelligence is the field of machine learning: where the agent improves its decision making process through observation of data. The concern of this paper is reinforcement learning, a sub-field of machine learning that concerns itself with algorithms and agents that gather their data from interaction with an environment. Reinforcement learning distinguishes itself from the other major sub-fields of machine learning, i.e. supervised and unsupervised learning, by being an active process whose decisions may be dependent on previous decisions.

Learning in this context seeks to optimize a cumulative reward, where each reward is a product of the environment and potential actions of the agent. The motivation behind the reinforcement learning approach is that the solution to many problems can be represented as maximization of a cumulative reward.

Consider now the formalization of our problem; for each step t, an agent receives an observation O_t (and potentially a reward R_t) from the environment, and then takes an action A_t on the environment. The environment then provides an observation O_{t+1} and a scalar reward R_{t+1}, dependent on the action A_t.

The environment is presumed to have an internal state, which generally cannot be observed in its entirety. The agent state S_t is some function of the history, which itself is defined as the sequence of observations, actions, and rewards.

$$H_t = O_0, A_0, R_1, O_1, \ldots, R_t, O_t$$

We update the agent state with respect to a general state update function u,

$$S_{t+1} = u(S_t, A_t, R_{t+1}, O_{t+1})$$

The policy π of an agent defines its behavior. Under a deterministic policy, the policy maps states to actions $\pi(S) = A$, while under a stochastic policy, the policy returns a probability of choosing that action given a state $\pi(A \mid S) = p(A \mid S)$.

The ultimate goal of an agent is to maximize the (expected) cumulative reward, referred to as the return. We define the value as the expected return from a state S given a policy π and discount factor γ. It follows that in order to reach our goal of maximizing expected cumulative reward, we should aim to maximize value. There are three different fundamental approaches to this problem: dynamic programming, Monte Carlo methods, and temporal-difference learning.

II. Methods

A. The Problem: Markov Decision Processes

Markov decision processes (MDPs) are a simplified version of the general reinforcement learning problem, in which actions influence immediate rewards, subsequent states, and future rewards. We will be focusing on finite MDPs, where the sets of actions, of states, and of rewards are finite. It follows that S_t and R_t have discrete probability distributions which only depend on the previous state and action.

In order to maximize the return, the agent will have to learn the tradeoff between immediate & future reward. We must consider two different scenarios. In the episodic scenario, we have a special terminating state that precedes a reset to a standard starting state. The agent-environment interaction then naturally breaks into a series of episodes, where each episode can be thought of as a game or some sort of repeated task. In the continuous scenario, there is no terminating state, since the agent goes through a possibly infinite number of states. It is then possible that we will try to maximize an infinite return. To combat this, we use discounting, a method in which our agent chooses its action to minimize

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots = \sum_{i=0}^{\infty} \gamma^k R_{k+t+1}$$

We define $0 \leq \gamma \leq 1$ to be the discount rate, which determines the present value of future rewards.

As stated in the introduction, a policy π maps each state to the probabilities of selecting the possible actions. We can model the expected return from state s and policy π as the value function $v_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s]$. Taking this a step further, we can calculate the value resulting from taking action a at state s with policy π as $q_\pi(s,a) = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$.

There are three main classes of methods that seek to solve the MDP problem. We will explore each of these in depth in the following subsections.

B. Dynamic Programming

While dynamic programming (DP) algorithms have lots of theoretical importance within reinforcement learning, they are seldom utilized due to stringent assumptions and long runtimes. The first goal of dynamic programming is policy evaluation, which seeks to compute the value function v_π given some policy π. In the DP approach, we make the simplifying assumption that the agent knows everything about its environment. We can complete policy evaluation in an iterative manner by creating a sequence of approximate value functions $v_0, v_1, v_2, ...$ which converges to v_π. This sequence is built as follows:

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s],$$

with v_0 chosen arbitrarily. Each iteration produces the new function v_{k+1} by updating the value of every state once. Note that whenever we perform an update in dynamic programming, we call it an expected update, as we are updating the expectation over all possible subsequent states.

Recall that the value function v_π depends on our policy π. Therefore, we can use our estimate of v_π to see how effective the given policy π is, and even see if an alternate policy π' would be preferable. We can do this by selecting an action a which is not the action chosen by π at time t, and then following π afterwards. The value of this state-action pair is

$$q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a]$$

Observe that if this value is greater than $v_\pi(s)$, then we have a suboptimal policy π, since we can improve it by instead choosing action a. We expand this across all states by using a greedy strategy. In this strategy, we select the action a that maximizes $q_\pi(s,a)$ at each step. Intuitively, this process is known as policy improvement.

We build upon policy evaluation and policy improvement by combining them to create policy iteration. Policy iteration is simply a cycle of using a function v_{π_0} to improve policy π_0 and yield a better policy π_1, and then computing v_{π_1} and improving π_1 to yield π_2, and so on.

It is easy to see that completing policy iteration in order to find the optimal policy π^* and value function v_{π^*} may take a very long time, since each policy evaluation is an iterative computation. Due to this problem, value iteration is a popular substitute for policy iteration. Value iteration can be thought of as combining a truncated version of policy evaluation with policy improvement. In value iteration, we update $v_{k+1}(s)$ as

$$v_{k+1}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a]$$

In practice, we terminate this iteration when the difference between v_k and v_{k+1} is sufficiently small.

As mentioned previously, dynamic programming algorithms aren't used very often to solve MDPs, and a large part of this is their long runtimes. Because each iteration of value iteration considers every possible action, it is easy to see that DP solutions scale poorly with respect to the number of possible actions. While some methods, known as asynchronous DP methods, try to mitigate this challenge, they are still far from optimal.

C. Monte Carlo Methods

Whereas dynamic programming algorithms require the agent to have complete knowledge of the environment, Monte Carlo (MC) methods require no knowledge of the environment. Instead, MC methods learn fro simulations. More specifically, Monte Carlo methods attempt to solve the reinforcement learning problem by averaging sample returns, and therefore can only be defined for episodic problems. We want to be able to use the framework from the previous subsection with MC methods, where we have policy evaluation and policy improvement, and then combine the two through policy iteration to find the optimal policy and value function at that policy.

Let's begin with policy evaluation, that is, estimating $v_\pi(s)$ given policy π. Our sole input is a sample set of episodes which follows policy π and goes through state s. We visit state s every time that we pass through s in an episode. Of course, we may end up passing through s multiple times during the same episode. The most common way to estimate $v_\pi(s)$ is known as first-visit MC, in which we take the mean of the returns after state s's first visit. It is easy to see that by the law of large numbers, this approach converges to the true $v_\pi(s)$.

When using Monte Carlo methods, it is common to have no access to a model. It follows that policy evaluation based solely on state is insufficient, and it is instead better to complete policy evaluation for state-action pairs by estimating $q_\pi(s,a)$. The MC methods in this situation follow the same approach as before - average the returns following the first visit to state s where action a was selected. However, problems may arise if some state-action pairs are never visited, causing the MC averaging to be unreliable and our estimates to be inaccurate.

To solve this problem, we need to find a way to estimate the value of every possible state-action pair. One common way of surpassing this challenge is by specifying that each episode starts with a randomly selected state-action pair, where each possible state-action pair has a positive probability of being selected. This approach is known as exploring starts.

We will now move on to using MC methods to find optimal policies. We will accomplish this by using the same policy iteration structure introduced in the previous section.

Assuming that we use the exploring starts approach, we can find optimized policy π_* by:

$$\pi_0 \to q_{\pi_0} \to \pi_1 \to q_{\pi_1} \to ... \to \pi_* \to q_{\pi_*}$$

However, the assumption of exploring starts can not always be made, since when the agent is learning directly from actual interactions, the starting position does not give helpful information. Since traditional policy iteration fails when we do not have exploring starts, there exist methods, referred to as on-policy and off-policy methods, that deal with this scenario. Exploration of these topics in depth is beyond this paper, and we suggest [1] if these are of interest.

D. Temporal-Difference Learning

Temporal-difference (TD) learning is central to reinforcement learning and can be though of as a fusion of dynamic programming and Monte Carlo. Specifically, TD learning allows for estimation from sample episodes without a model like in MC, but also updates its estimates based on its other estimates like in DP. The relationship between temporal-difference learning, Monte Carlo methods, and dynamic programming algorithms is frequently studied in reinforcement learning theory.

Temporal-difference learning uses the same policy iteration framework as before to find the optimal policy. The key difference thus lies in its estimation of $v_\pi(s)$. Let θ denote TD learning's estimate of v_π and let α be a constant step size. TD methods update

$$\theta(S_t) \leftarrow \theta(S_t) + \alpha[R_{t+1} + \gamma\theta(S_{t+1}) - \theta(S_t)]$$

when they transition to state S_{t+1}. Note that this updates using an existing estimate of S_t and is thus similar in this regard to the dynamic programming approach.

As previously stated, temporal-difference learning is a combination of Monte Carlo and dynamic programming. It follows that TD learning has sizeable advantages over both of these approaches. TD learning is preferable to DP algorithms since it does not require the agent to have full knowledge of the environment, and thus is more useful in practice. It is also preferable to MC methods, as MC methods wait until the end of each episode to perform its update, whereas TD learning updates at each individual time step.

Unfortunately, TD learning inherits one of the main problems from Monte Carlo methods: we do not observe all possible state-action pairs. Similarly to the Monte Carlo approach, there are both on-policy and off-policy TD learning methods to remedy this. Some of the most well-researched methods are Sarsa (on-policy), Q-learning (off-policy), and expected Sarsa (both). We do not have time to dive into these methods in this report, and recommend [3,4,5] for further reading.

III. Empirical Evaluation of Methods

We seek to validate and compare performances of Monte Carlo methods and temporal difference learners with respect to a simple problem, that of the playing card game blackjack. We consider an independent game between an agent and a dealer. The objective of the game is to maximize the sum of card values without exceeding 21. Each card is assigned their natural numerical value, all except the ace which may count as 1 or 11, and the face cards which each count as 10. Each player begins the game with a hand of two cards, and one of the dealer's cards is revealed. Then the agent may choose whether to draw an additional card (*hit*), or stop (*stick*). If at any point the agent's hand exceeds 21 (*busts*), the agent loses, and the episode terminates. If the player sticks, then the dealer plays according to a fixed policy: sticking when their hand is at least 17, and hitting otherwise. If the dealer sticks, the agent wins. In all other cases, winning, losing, and drawing are determined by the values of the player's hands.

The formulation of the desired Markov decision process follows easily. Suppose cards are drawn from an infinite deck such that the probability of receiving any given card does not vary with cards drawn. Then a nonzero reward is provided when the game terminates: 1 for winning, 0 for drawing, and -1 for losing. We define states to be the tuple of $(hand\ value,\ dealer\ card,\ useable\ ace)$, where $useable\ ace$ is 1 if there is an ace in the hand that can be valued at 11 without going bust, and 0 otherwise. At each non-terminal step, the agent may choose to either hit or stick.

For such a MDP, we can then learn policies using Monte Carlo methods and temporal-difference learning. In particular, we implement on-policy every-visit MC control [1], Sarsa (on-policy temporal-difference control) [4], Max Sarsa (Q-learning or off-policy TD control) [3], and Expected Sarsa [5]..

Fig. 1 shows the actual computed policies, and compares each to the optimal policy. We can see that the policies are close to the optimal policy, which is exactly the "basic" strategy noted presented by [2]. Even with inexact computed policies, we see that their eventual performance in Fig. 2 shows that the win rate is nearly optimal as per [2].

Fig. 2 also shows the training performance in terms of average wins over time for each of the tested methods. While each method approaches the same win rate, of note is how Max Sarsa reaches that rate the fastest. Although not shown here, one can also observe that each algorithm quickly reaches a close approximation of the optimal policy in terms of win rate, even for small numbers of episodes.

Fig. 3 (see appendix) describes the state-value function over the state space, at two points in training under the Monte Carlo algorithm. The first approximation at 10,000 episodes shows more irregularity, and is less "smooth," as compared to the same function at 500,000 episodes.

IV. Discussion

Each of the methods converge to a near optimal policy in terms of performance; we suspect that the discrepancies between our estimated policies and the optimal one is due to the problem of unobserved states in model-free frameworks.

The above described methods each converge to the optimal policy in the limit. What distinguishes them is the assumptions made and the policy convergence rate.

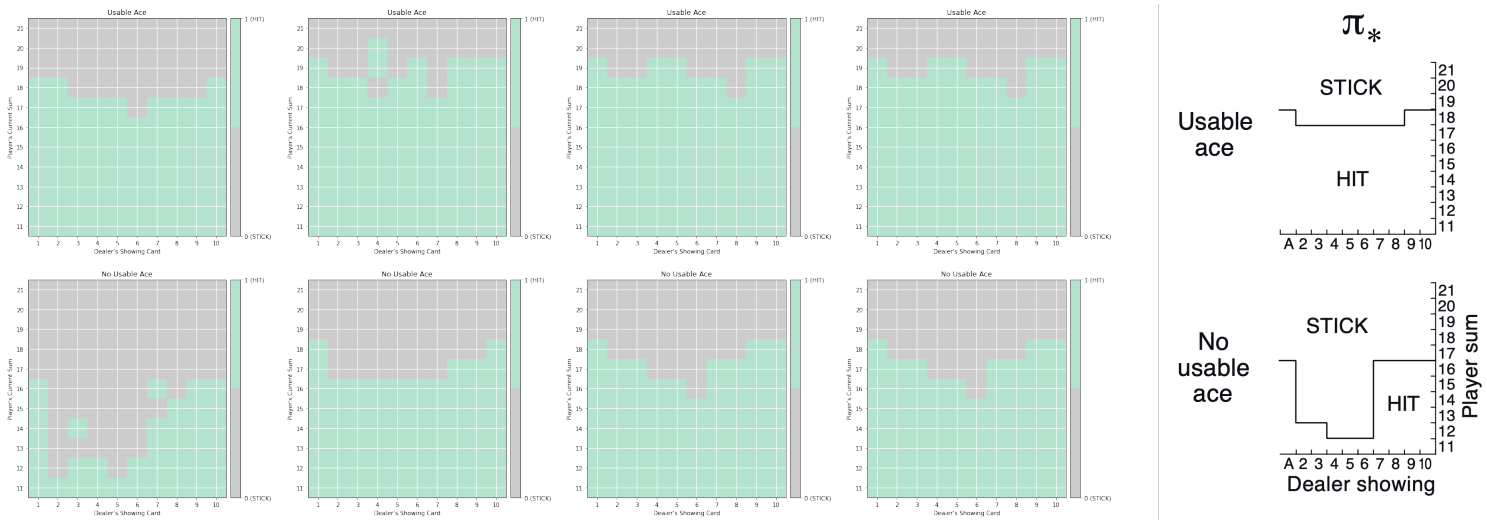


Fig. 1. Estimated policy from each of the tested methods, on 1,000,000 episodes. Action space is separated into usable and unusable ace cases for the top and bottom plots, respectively. From left to right: Monte Carlo, Sarsa, Max Sarsa (Q-Learning), Expected Sarsa, optimal policy. See appendix for enlarged plots.

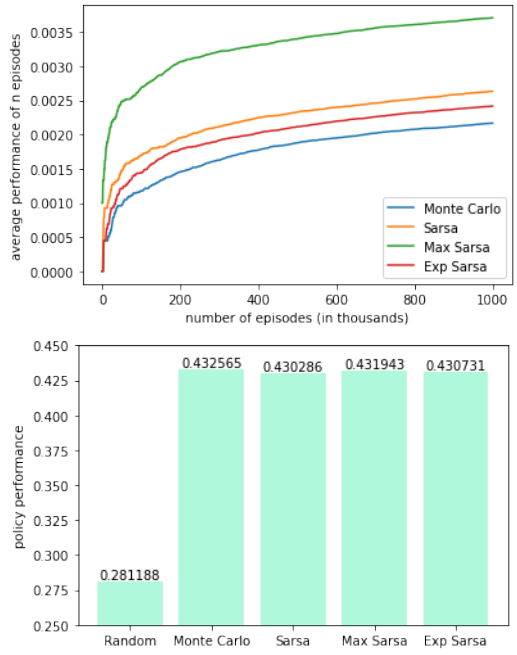


Fig. 2. **Left:** Average number of wins vs number of episodes over the course of training. **Right:** Proportion of wins for learned policy on 1,000,000 episodes.

Dynamic programming methods are the theoretically simplest reinforcement learning algorithms, but assume a perfect model. That is, that the probabilities $p(s', r \mid s, a)$ are given for each possible s, s', a, r. Of course, this is seldom available in explicit form. Further, DP methods are prohibitively expensive in the computational sense for all but the simplest models. Despite this, the mathematical background behind DP is well formulated, and gives rise to convergence guarantees to optimal policies. As such, the ideas of DP (e.g. generalized policy iteration) are frequently used in subsequent RL methods.

Monte Carlo methods implement a generalized policy iteration based on sample estimates, and therefore do not require a model of the environment. Further, the computed estimates are independent from previous estimates, i.e. bootstrapping.

TDL methods require no model, are fully incremental, and utilize bootstrapping as in DP. In some cases they converge faster than MC, but share some of the same pitfalls regarding unobserved states.

The methods presented in this paper are far from comprehensive. While the methods shown have convergence to optimal policies, the issue that these methods are prohibitively intractable, or even impossible for many problems of interest. Extensions to the proposed methods exist, where different qualities of each are mixed together to aggregate their strengths (e.g. multi-step bootstrapping). In each of the model-free methods, we estimate action-value function pointwise over the entire state-action domain and is clearly intractable in many cases of interest. Approximate methods seek to remedy this problem by learning approximate parameterized versions of the desired value functions (e.g. policy gradient methods).

ACKNOWLEDGMENT

The authors thank Dr. Genevera Allen for the development, instruction, and execution of ELEC 478: Introduction to Machine Learning.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction,* 2nd ed. Cambridge, MA, USA : MIT Press, 2018.
[2] E. Thorpe, *Beat the Dealer,* New York, NY, USA : Random House, 1966.
[3] C. Watkins, P. Dayan, *Q-learning*. Mach Learn **8**, 1992.
[4] G.A. Rummery and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," *University of Cambridge, Dept. of Engineering*, 1994.
[5] G. John, "When the best move isn't optimal: Q-learning with exploration," *AAAI*, 1994.

V. Appendix

See Fig. 3; and Fig. 4 for an enlarged version Fig. 1. See below for code.

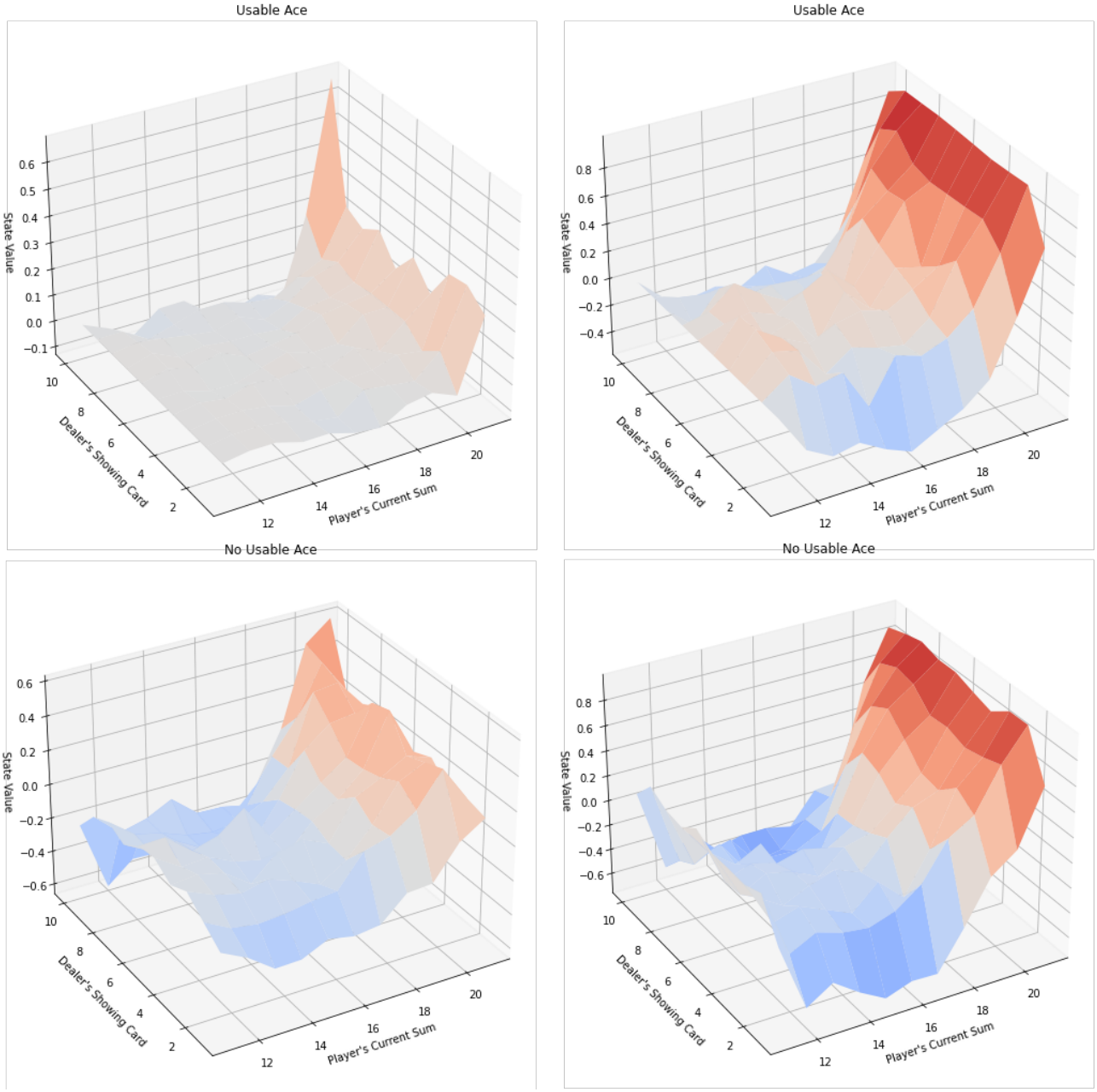


Fig. 3. Each of the plots are of state-value functions, with the value plotted as a surface over possible hand values and dealer cards; states that have a "useable ace" are plotted above the states that do not. Functions have been computed from Monte Carlo methods; with differing episode count. **Left:** trained over 10,000 episodes. **Right:** trained over 500,000 episodes.

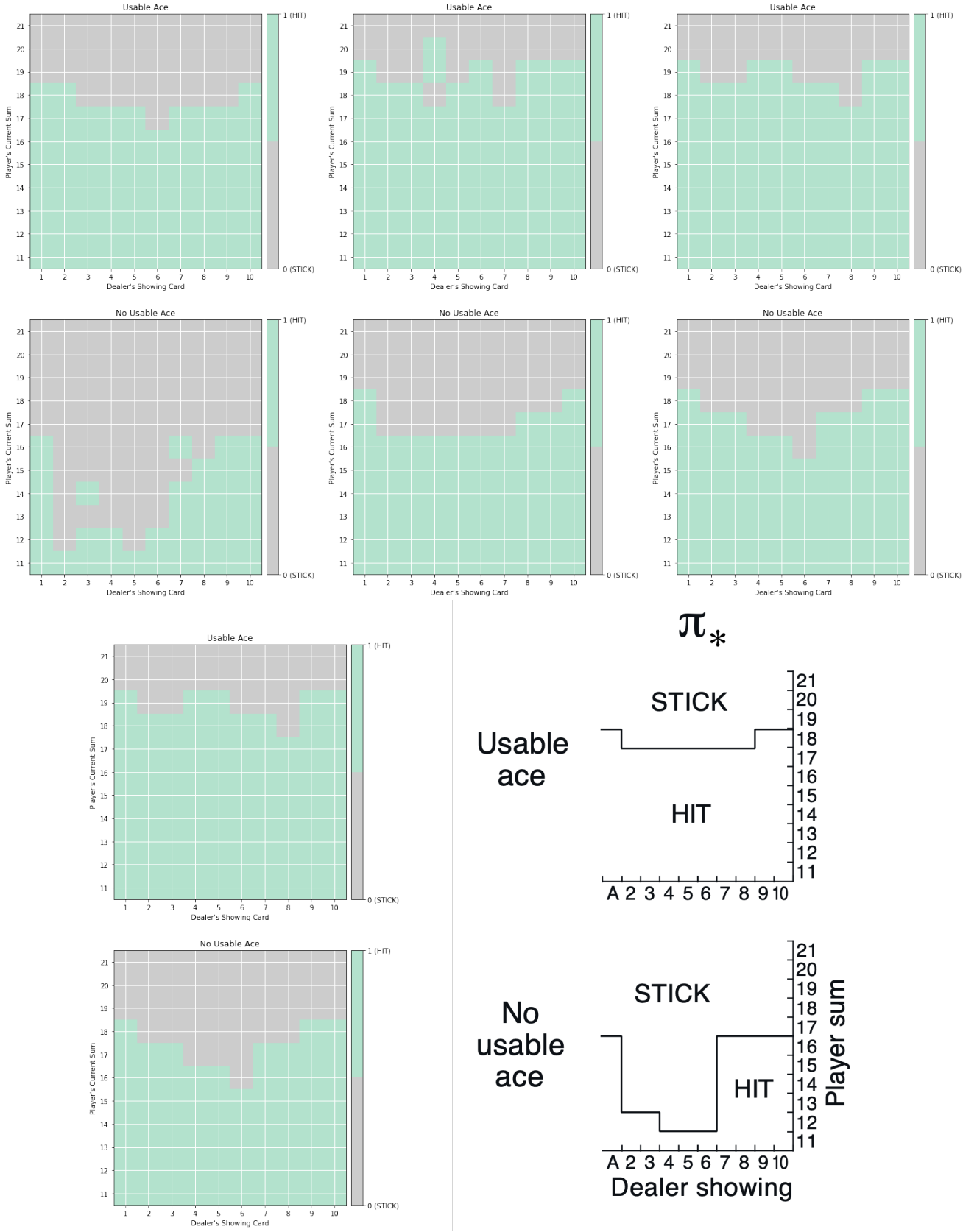


Fig. 4. Estimated policy from each of the tested methods, on 1,000,000 episodes. Action space is separated into usable and unusable ace cases for the top and bottom plots, respectively. From left to right: Monte Carlo, Sarsa, Max Sarsa (Q-Learning), Expected Sarsa, optimal policy.

rl_blackjack

December 14, 2021

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

[ ]:
```
env = gym.make('Blackjack-v1')
# print(env.observation_space)
# print(env.action_space)
```

[ ]:
```
# from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable

def plot_blackjack_values(V):

    def get_Z(x, y, usable_ace):
        if (x,y,usable_ace) in V:
            return V[x,y,usable_ace]
        else:
            return 0

    def get_figure(usable_ace, ax):
        x_range = np.arange(11, 22)
        y_range = np.arange(1, 11)
        X, Y = np.meshgrid(x_range, y_range)

        Z = np.array([get_Z(x,y,usable_ace) for x,y in zip(np.ravel(X), np.
 ↪ravel(Y))]).reshape(X.shape)

        surf = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.cm.
 ↪coolwarm, vmin=-1.0, vmax=1.0)
        ax.set_xlabel('Player\'s Current Sum')
        ax.set_ylabel('Dealer\'s Showing Card')
        ax.set_zlabel('State Value')
        ax.view_init(ax.elev, -120)
```

```
    fig = plt.figure(figsize=(20, 20))
    ax = fig.add_subplot(211, projection='3d')
    ax.set_title('Usable Ace')
    get_figure(True, ax)
    ax = fig.add_subplot(212, projection='3d')
    ax.set_title('No Usable Ace')
    get_figure(False, ax)
    plt.show()
```

[ ]:
```
def get_probs(Q_s, epsilon, nA): #nA is no. of actions in the action space
    # 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

'''
Now we will use this get_probs func in generating the episode.
Note that we are no longer using the stochastic policy we started with, instead␣
 ↪building upon it in an epsilon greedy way.
'''
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

'''
Finally Q values are approximated by taking average of corresponding returns.
But instead we can write it using incremental mean and constant alpha.
As we are using constant alpha we need not keep a track of N-table, ie how many␣
 ↪times we visited that state.
'''

def update_Q(env, episode, Q, alpha, gamma):
    # updates the action-value function estimate using the most recent episode
    states, actions, rewards = zip(*episode)
```

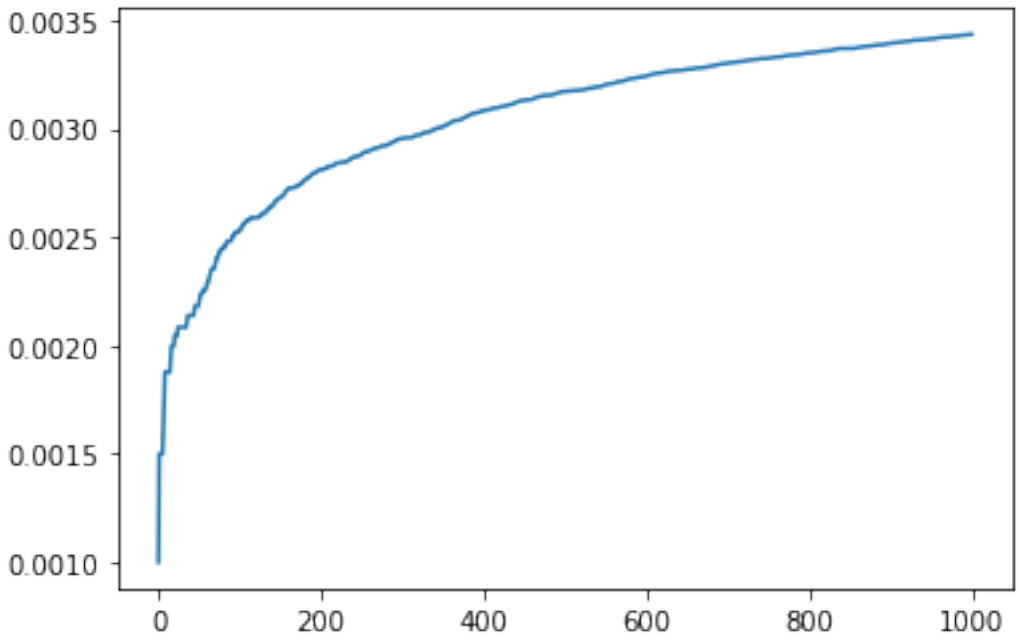
```
    # prepare for discounting
    discounts = np.array([gamma**i for i in range(len(rewards)+1)])
    for i, state in enumerate(states):
        old_Q = Q[state][actions[i]]
        Q[state][actions[i]] = old_Q + alpha*(sum(rewards[i:]*discounts[:
↪-(1+i)]) - old_Q)
    return Q
```

[ ]:
```
def mc_control(env, num_episodes, alpha, gamma=1.0, eps_start=1.0, eps_decay=.
↪99999, eps_min=0.05):
    performance = []
    temp = 0

    nA = env.action_space.n
    # initialize empty dictionary of arrays
    Q = defaultdict(lambda: np.zeros(nA))
    epsilon = eps_start
    # 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()

            temp = ( temp * (i_episode - 1) + (episode[-1][-1] > 0)) / i_episode
            # temp += (episode[-1][-1] > 0)
            performance.append(temp)


        # set the value of epsilon
        epsilon = max(epsilon*eps_decay, eps_min)
        # 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(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, performance
```

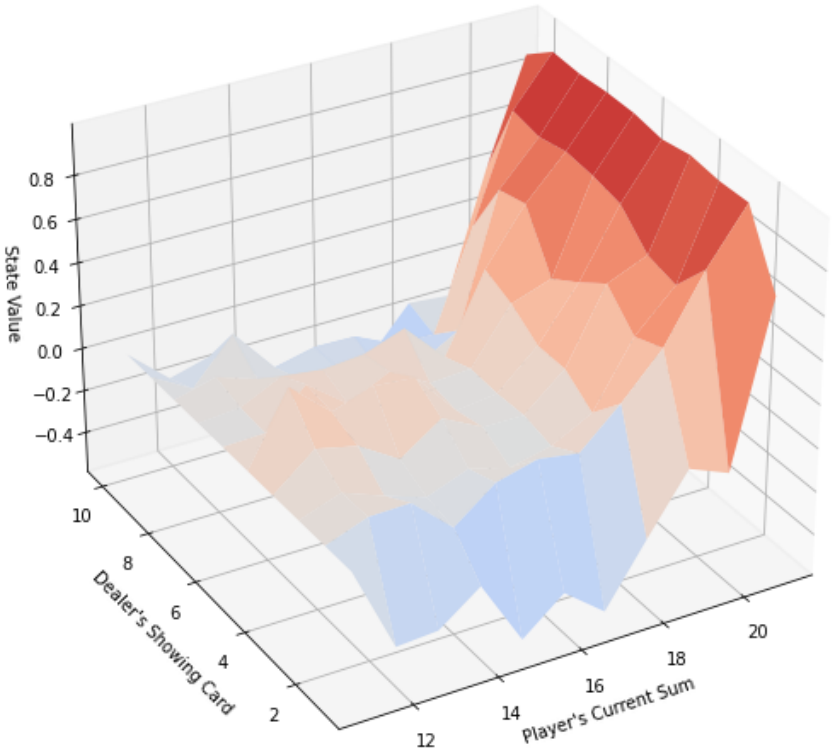
[ ]:
```
# obtain the estimated optimal policy and action-value function
policy, Q, performance_mc = mc_control(env, 1000000, 0.015)

plt.plot(performance_mc)
```

```
# 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)
```

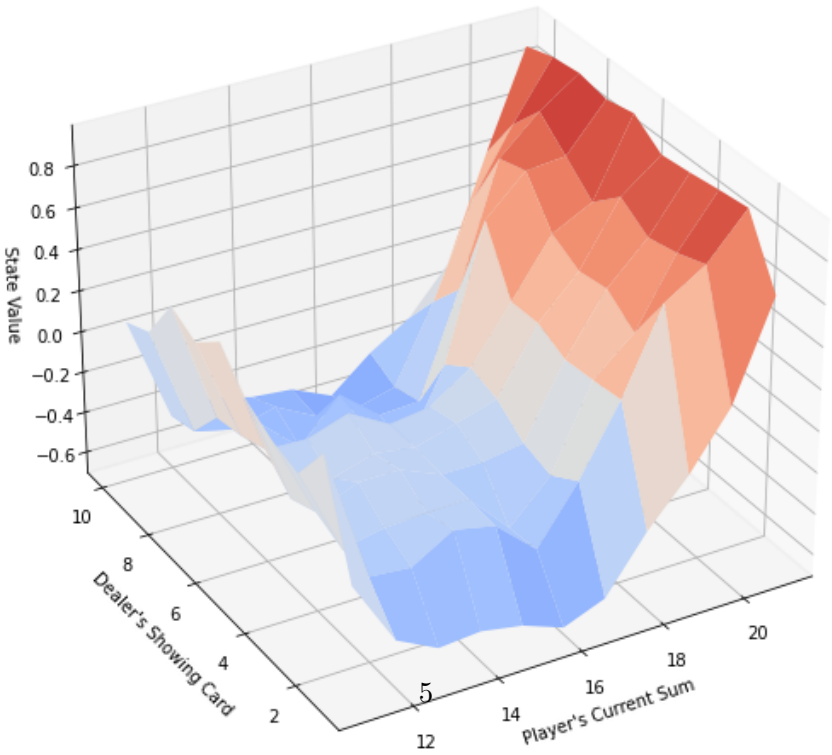
```
Episode 1000000/1000000.
```



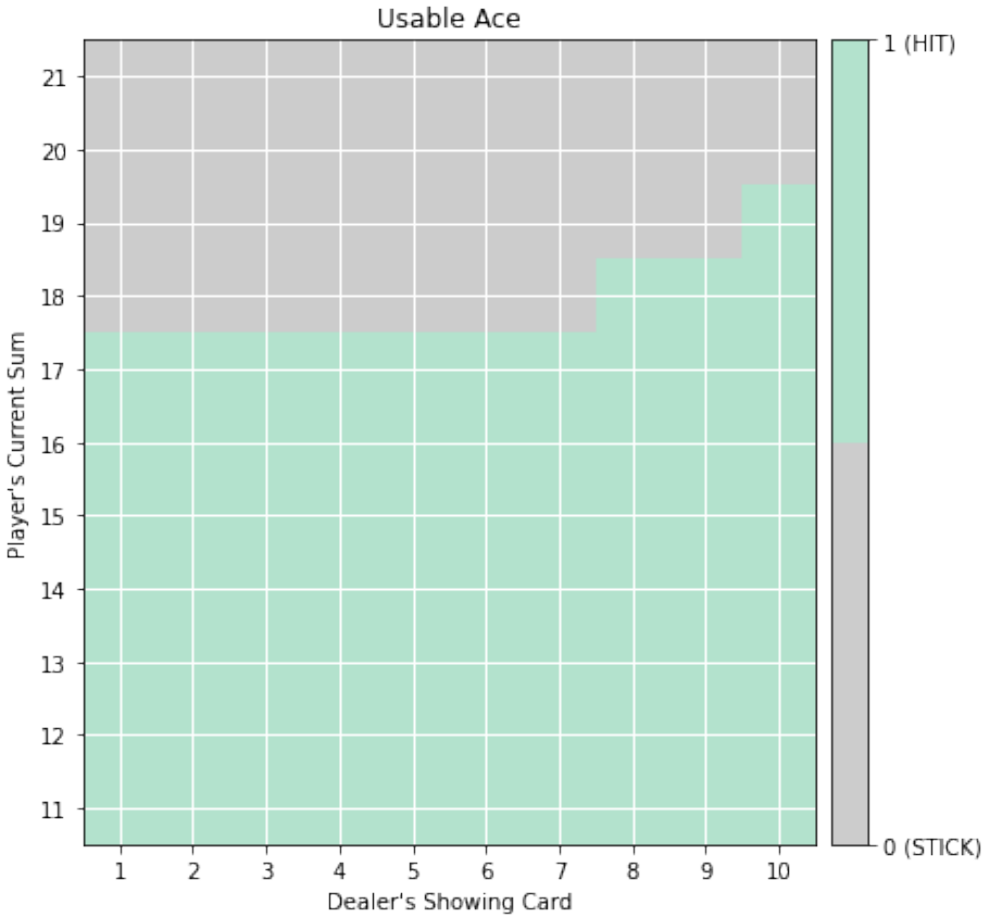
Usable Ace

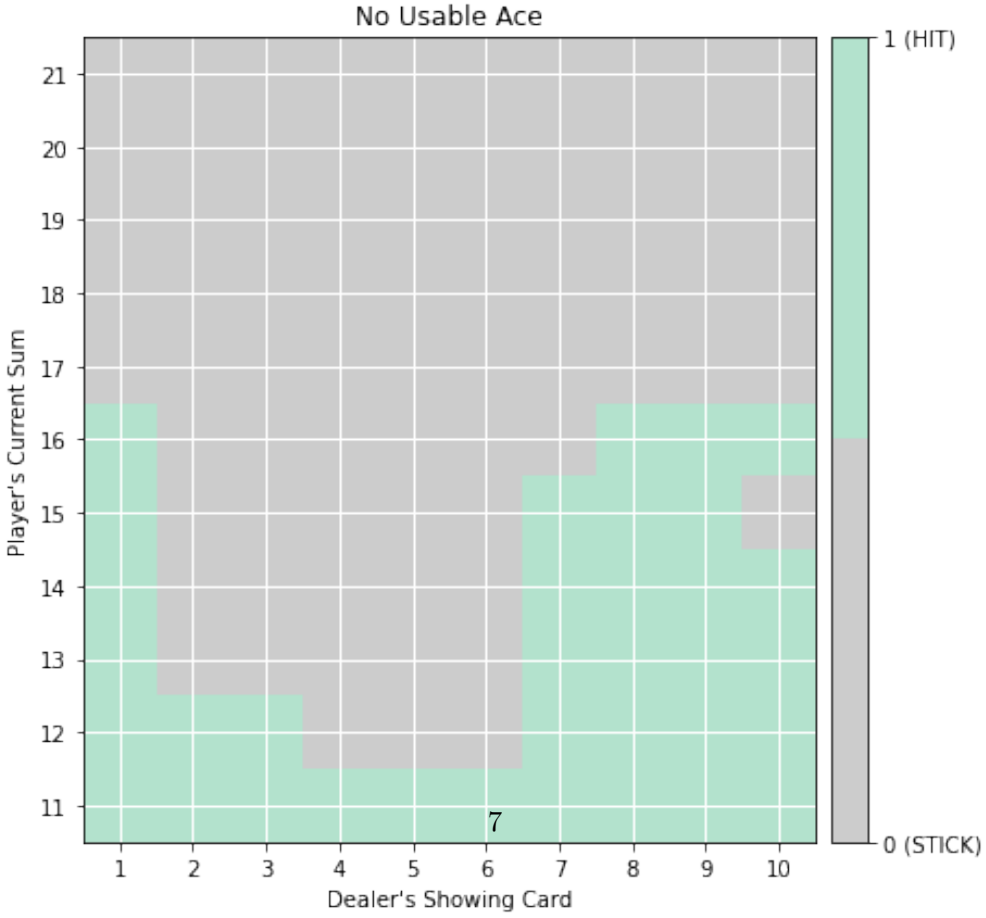


No Usable Ace



[ ]:
```
def plot_policy(policy):

    def get_Z(x, y, usable_ace):
        if (x,y,usable_ace) in policy:
            return policy[x,y,usable_ace]
        else:
            return 1

    def get_figure(usable_ace, ax):

        x_range = np.arange(1, 11)
        y_range = np.arange(21, 10, -1)
        # X, Y = np.meshgrid(x_range, y_range)
        Z = np.array([[get_Z(y,x,usable_ace) for x in x_range ] for y in␣
↪y_range])
        surf = ax.imshow(Z, cmap=plt.get_cmap('Pastel2_r', 2), vmin=0, vmax=1,␣
↪extent=[0.5, 10.5, 10.5, 21.5])
        plt.xticks(x_range)
        plt.yticks(y_range)
        plt.gca()
        ax.set_ylabel('Player\'s Current Sum')
        ax.set_xlabel('Dealer\'s Showing Card')
        ax.grid(color='w', linestyle='-', linewidth=1)
        divider = make_axes_locatable(ax)
        cax = divider.append_axes("right", size="5%", pad=0.1)
        cbar = plt.colorbar(surf, ticks=[0,1], cax=cax)
        cbar.ax.set_yticklabels(['0 (STICK)','1 (HIT)'])

    fig = plt.figure(figsize=(15, 15))
    ax = fig.add_subplot(211)
    ax.set_title('Usable Ace')
    get_figure(True, ax)
    ax = fig.add_subplot(212)
    ax.set_title('No Usable Ace')
    get_figure(False, ax)
    plt.show()

# plot the policy
plot_policy(policy)
```


Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)


No Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)

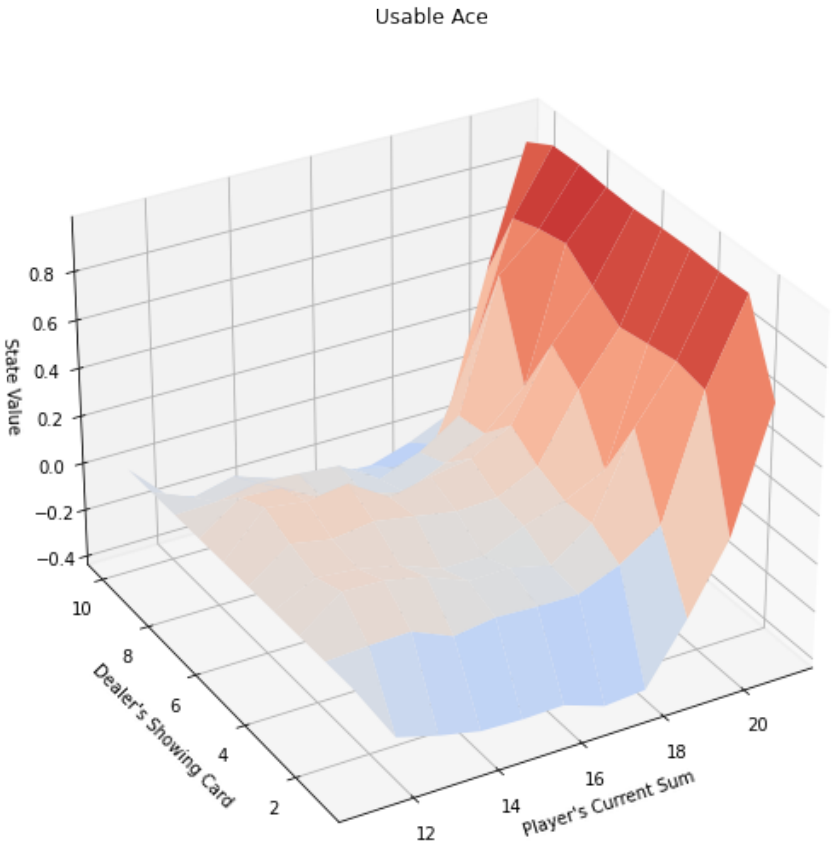
[ ]:
```
def update_Q_sarsa(alpha, gamma, Q, state, action, reward, next_state=None,␣
↪next_action=None):
    """Returns updated Q-value for the most recent experience."""
    current = Q[state][action]  # estimate in Q-table (for current state,␣
↪action pair)
    # get value of state, action pair at next time step
    Qsa_next = Q[next_state][next_action] if next_state is not None else 0
    target = reward + (gamma * Qsa_next)               # construct TD target
    new_value = current + (alpha * (target - current)) # get updated value
    return new_value

def epsilon_greedy(Q, state, nA, eps):
    """Selects epsilon-greedy action for supplied state.

    Params
    ======
        Q (dictionary): action-value function
        state (int): current state
        nA (int): number actions in the environment
        eps (float): epsilon
    """
    if random.random() > eps: # select greedy action with probability epsilon
        return np.argmax(Q[state])
    else:                     # otherwise, select an action randomly
        return random.choice(np.arange(env.action_space.n))
```

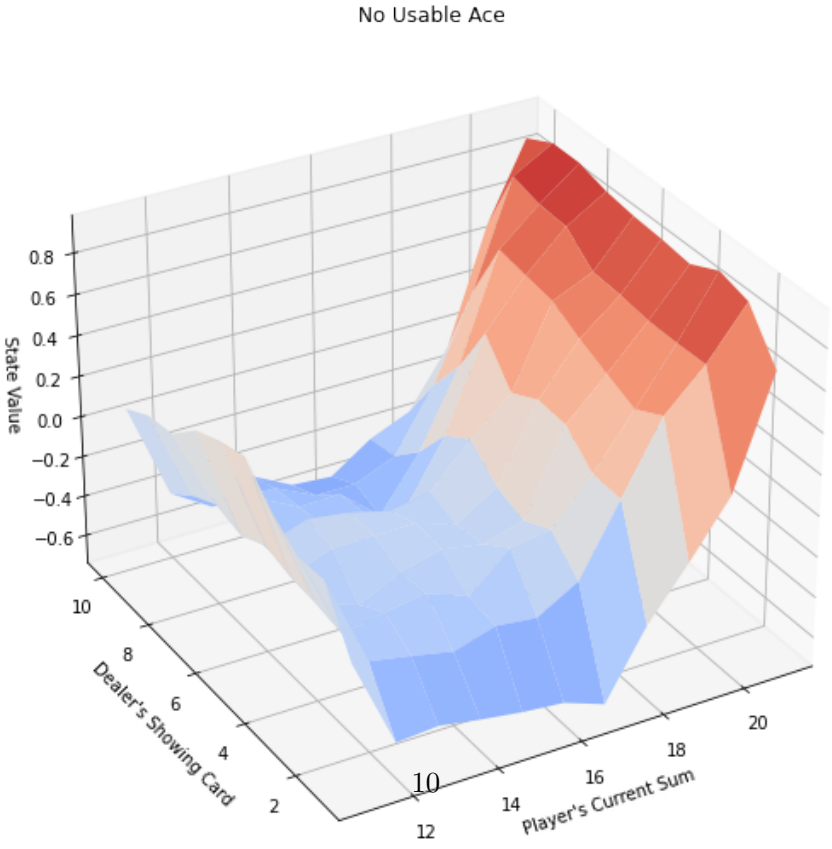
[ ]:
```
def sarsa(env, num_episodes, alpha, gamma=1.0, epsmin=0.01):
    nA = env.action_space.n                # number of actions
    Q = defaultdict(lambda: np.zeros(nA))  # initialize empty dictionary of␣
↪arrays

    performance = []
    temp = 0


    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()

            temp = ( temp * (i_episode - 1) + (reward > 0)) / i_episode
            performance.append(temp)
```

```
        score = 0                                          # initialize score
        state = env.reset()                                # start episode

        eps = max(1.0 / i_episode, epsmin)                                  #␣
↪set value of epsilon
        action = epsilon_greedy(Q, state, nA, eps)         # epsilon-greedy␣
↪action selection

        while True:
            next_state, reward, done, info = env.step(action) # take action A,␣
↪observe R, S'
            score += reward                                # add reward to␣
↪agent's score
            if not done:
                next_action = epsilon_greedy(Q, next_state, nA, eps) #␣
↪epsilon-greedy action
                Q[state][action] = update_Q_sarsa(alpha, gamma, Q, \
                                                  state, action, reward,␣
↪next_state, next_action)

                state = next_state     # S <- S'
                action = next_action   # A <- A'
            if done:
                Q[state][action] = update_Q_sarsa(alpha, gamma, Q, \
                                                  state, action, reward)
                break

    return Q, performance
```

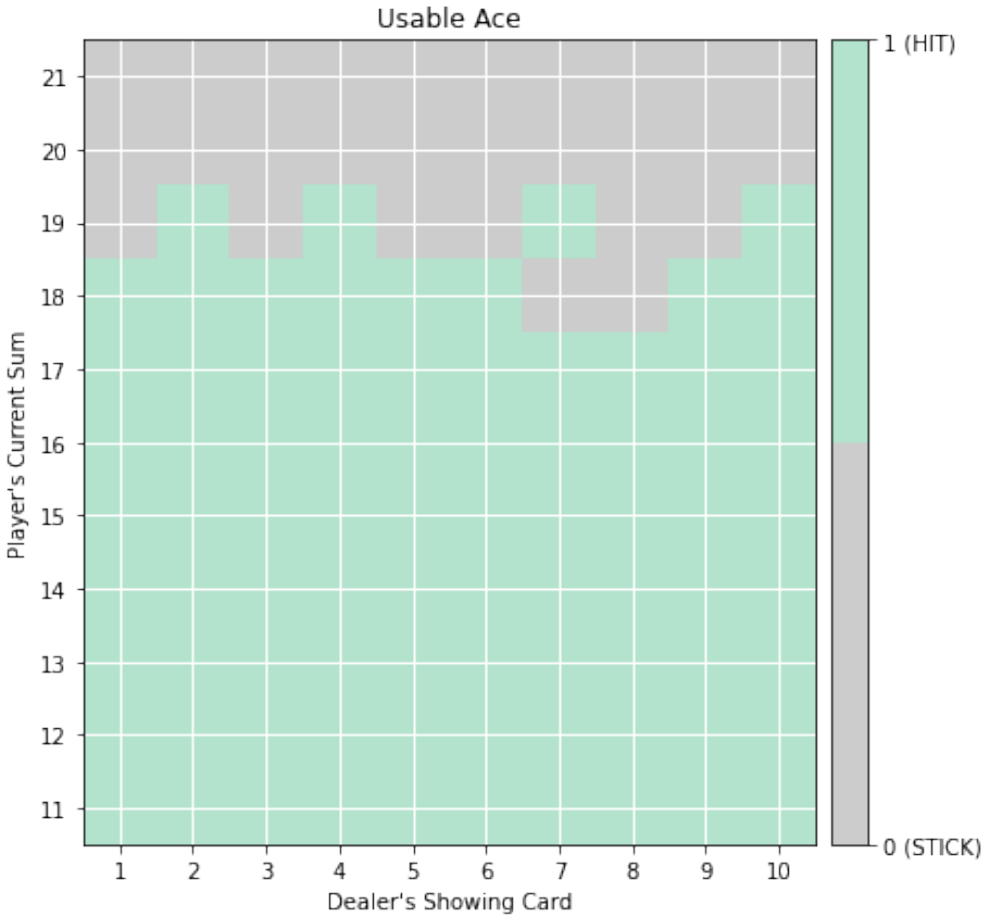
[ ]:
```
# obtain the estimated optimal policy and corresponding action-value function
Q_sarsa, performance_sarsa = sarsa(env, 1000000, 0.009)

# obtain the corresponding state-value function
V = dict((k,np.max(v)) for k, v in Q_sarsa.items())

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

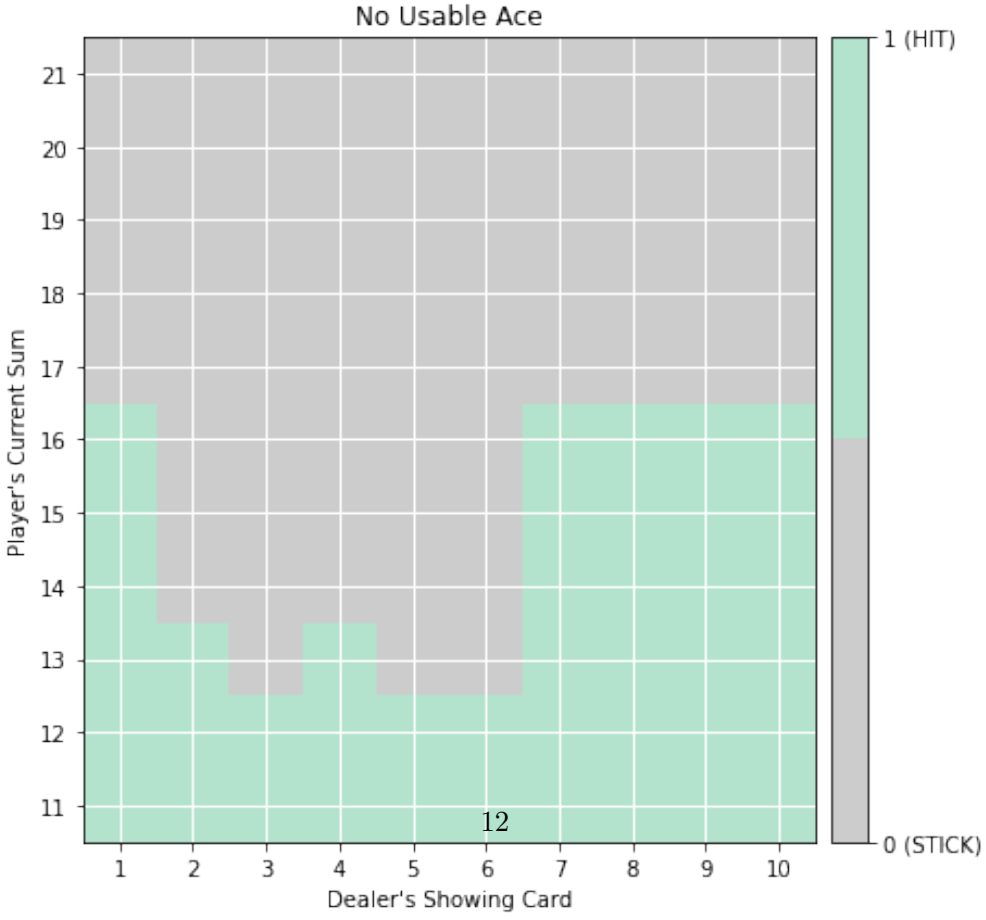
```
Episode 1000000/1000000
```


Usable Ace
State Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
Dealer's Showing Card
10
8
6
4
2
Player's Current Sum
12
14
16
18
20


No Usable Ace
State Value
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
Dealer's Showing Card
10
8
6
4
2
Player's Current Sum
12
14
16
18
20

[ ]:
```
policy_sarsa = dict((k,np.argmax(v)) for k, v in Q_sarsa.items())
# plot the policy
plot_policy(policy_sarsa)
```


Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)


No Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)

[ ]:
```
def update_Q_sarsamax(alpha, gamma, Q, state, action, reward, next_state=None):
    """Returns updated Q-value for the most recent experience."""
    current = Q[state][action]  # estimate in Q-table (for current state, action pair)
    Qsa_next = np.max(Q[next_state]) if next_state is not None else 0  # value of next state
    target = reward + (gamma * Qsa_next)               # construct TD target
    new_value = current + (alpha * (target - current)) # get updated value
    return new_value
```

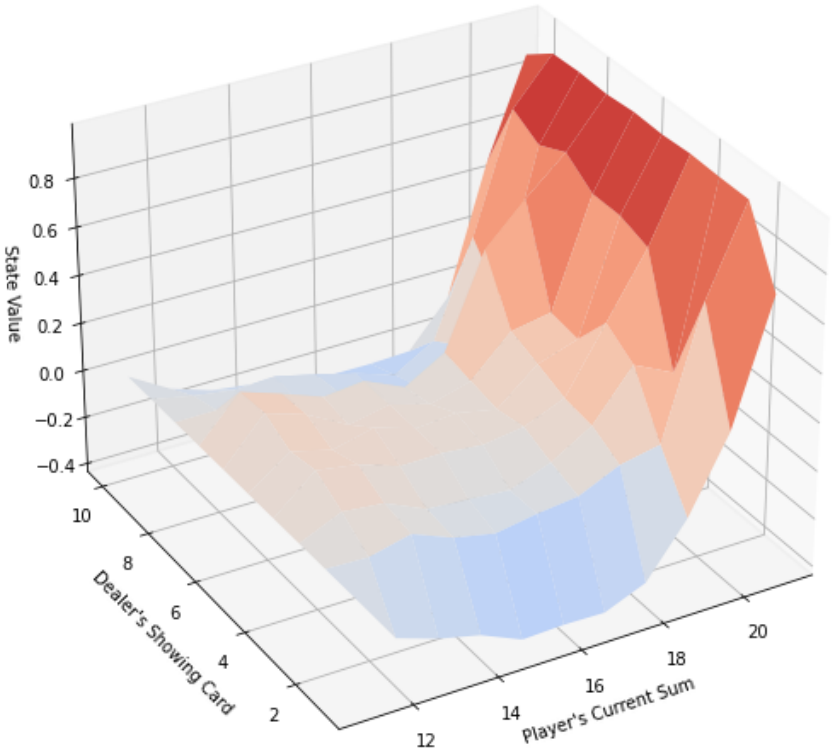
[ ]:
```
def q_learning(env, num_episodes, alpha, gamma=1.0,epsmin=0.01):
    """Q-Learning - TD Control

    Params
    ======
        num_episodes (int): number of episodes to run the algorithm
        alpha (float): learning rate
        gamma (float): discount factor
        plot_every (int): number of episodes to use when calculating average score
    """
    nA = env.action_space.n                # number of actions
    Q = defaultdict(lambda: np.zeros(nA))  # initialize empty dictionary of arrays

    temp, performance = 0, []



    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()

            temp = ( temp * (i_episode - 1) + (episode[-1][-1] > 0)) / i_episode
            performance.append(temp)

        score = 0                                                  # initialize score
        state = env.reset()                                        # start episode
        eps = max(1.0 / i_episode ,epsmin)                                  # set value of epsilon
```

```
        while True:
            action = epsilon_greedy(Q, state, nA, eps)          # epsilon-greedy␣
↪action selection
            next_state, reward, done, info = env.step(action)  # take action A,␣
↪observe R, S'
            score += reward                                     # add reward to␣
↪agent's score
            Q[state][action] = update_Q_sarsamax(alpha, gamma, Q, \
                                                 state, action, reward,␣
↪next_state)
            state = next_state                                  # S <- S'
            # note: no A <- A'
            if done:
                break
    return Q, performance
```

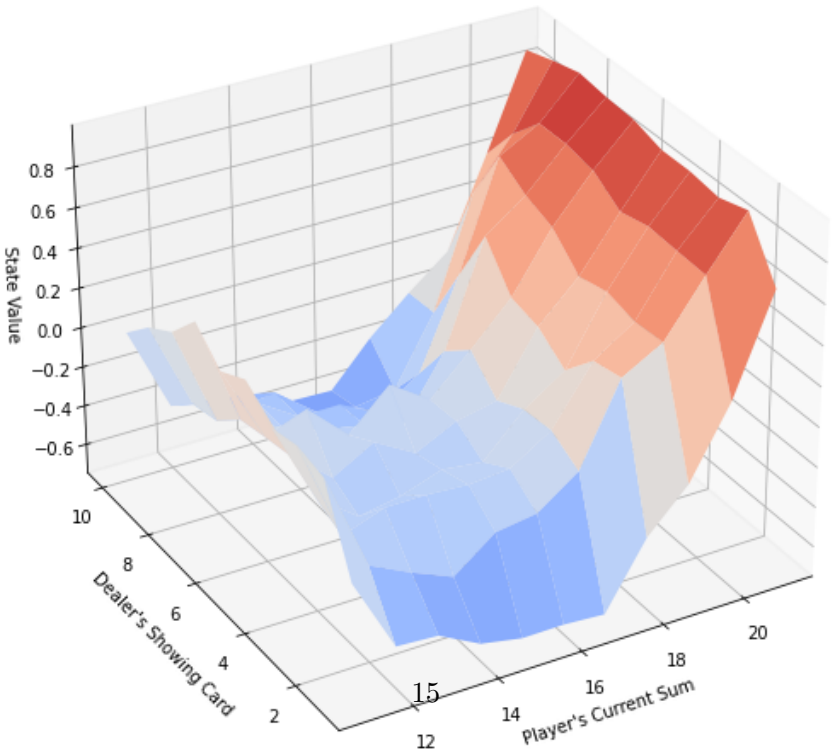
[ ]:
```
# obtain the estimated optimal policy and corresponding action-value function
Q_sarsamax, performance_maxsarsa = sarsa(env, 1000000, 0.01)

# obtain the corresponding state-value function
V = dict((k,np.max(v)) for k, v in Q_sarsamax.items())

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

```
Episode 1000000/1000000
```

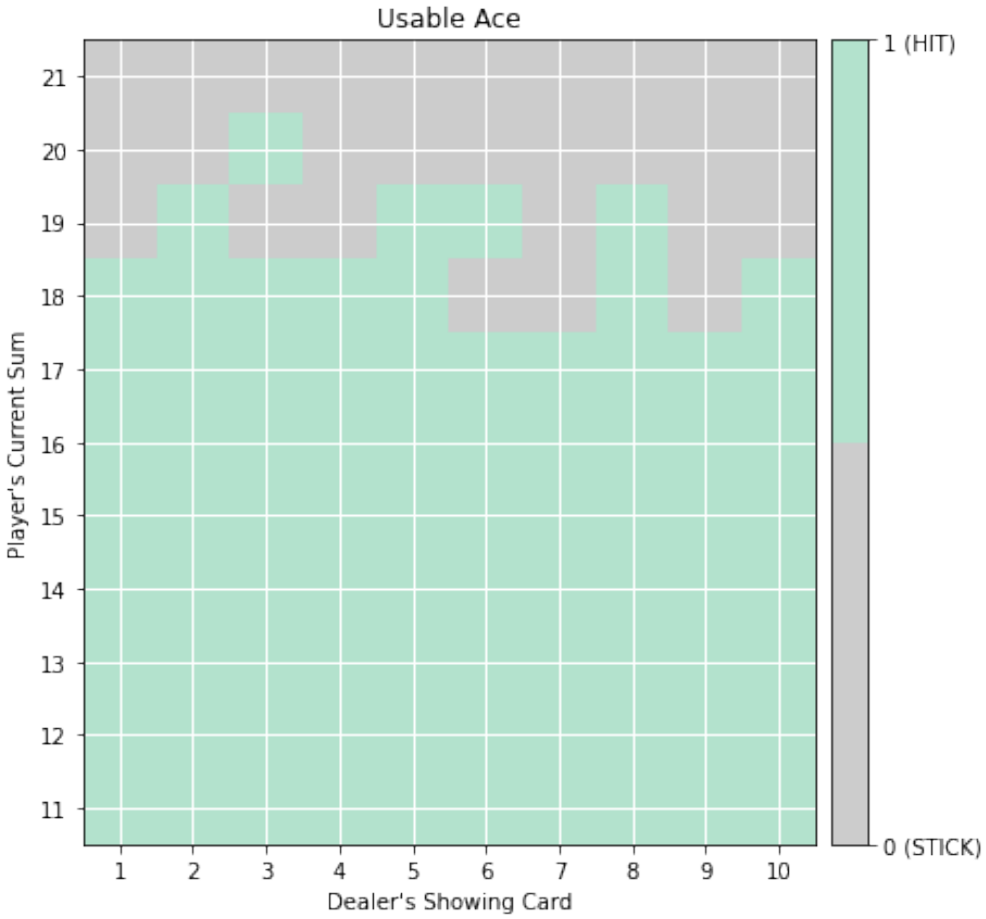
Usable Ace

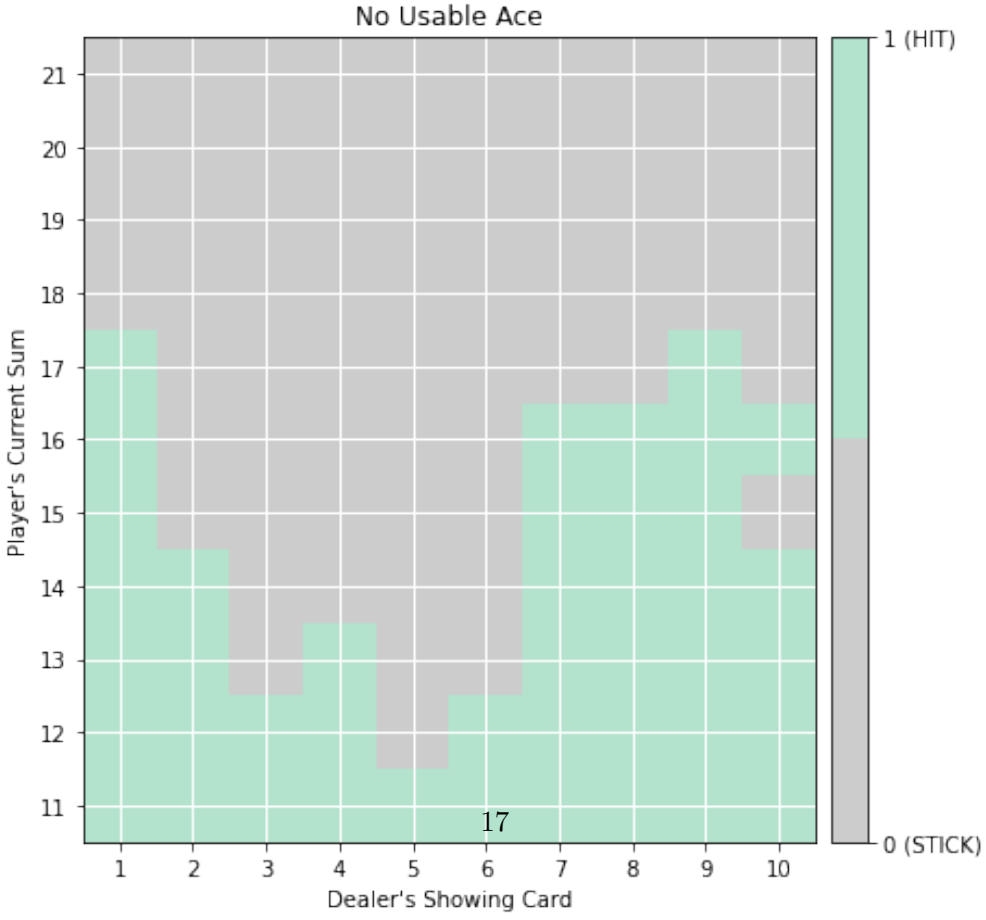


No Usable Ace



[ ]:
```
policy_sarsamax = dict((k,np.argmax(v)) for k, v in Q_sarsamax.items())
# plot the policy
plot_policy(policy_sarsamax)
```


Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)


No Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)

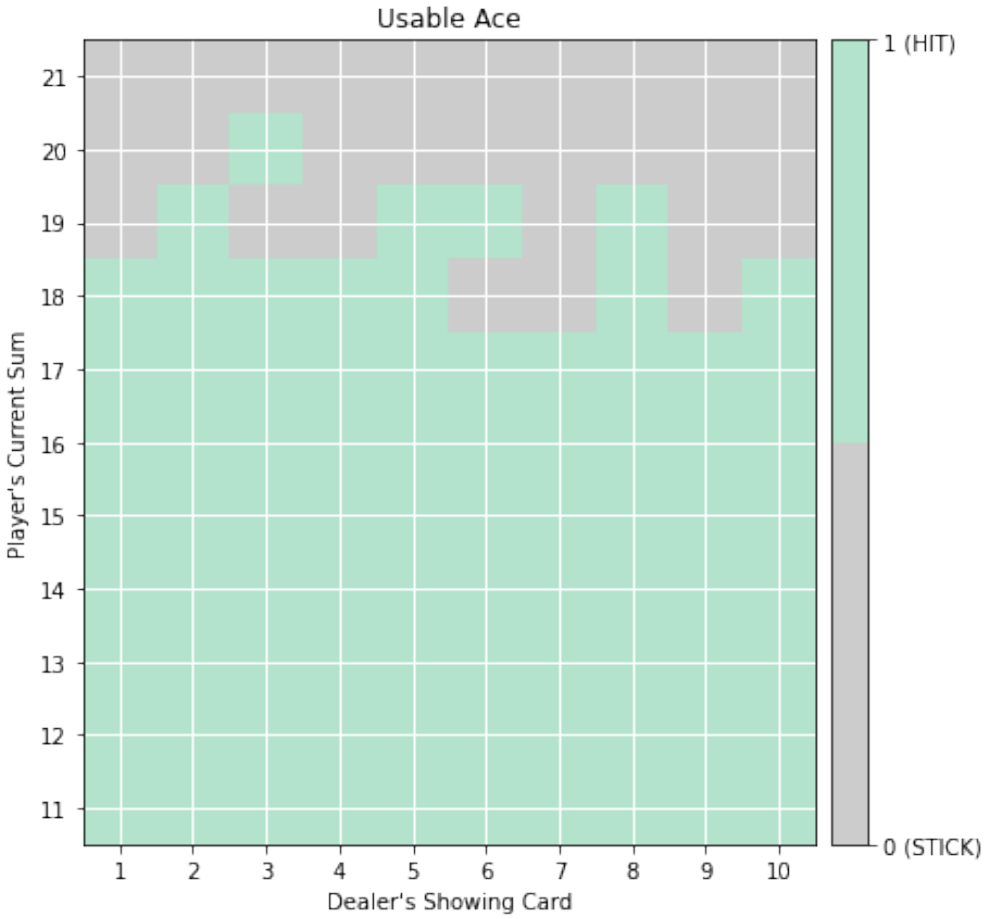
[ ]:
```
def update_Q_expsarsa(alpha, gamma, nA, eps, Q, state, action, reward,␣
↪next_state=None):
    """Returns updated Q-value for the most recent experience."""
    current = Q[state][action]          # estimate in Q-table (for current␣
↪state, action pair)
    policy_s = np.ones(nA) * eps / nA  # current policy (for next state S')
    policy_s[np.argmax(Q[next_state])] = 1 - eps + (eps / nA) # greedy action
    Qsa_next = np.dot(Q[next_state], policy_s)         # get value of state at␣
↪next time step
    target = reward + (gamma * Qsa_next)               # construct target
    new_value = current + (alpha * (target - current)) # get updated value
    return new_value
```

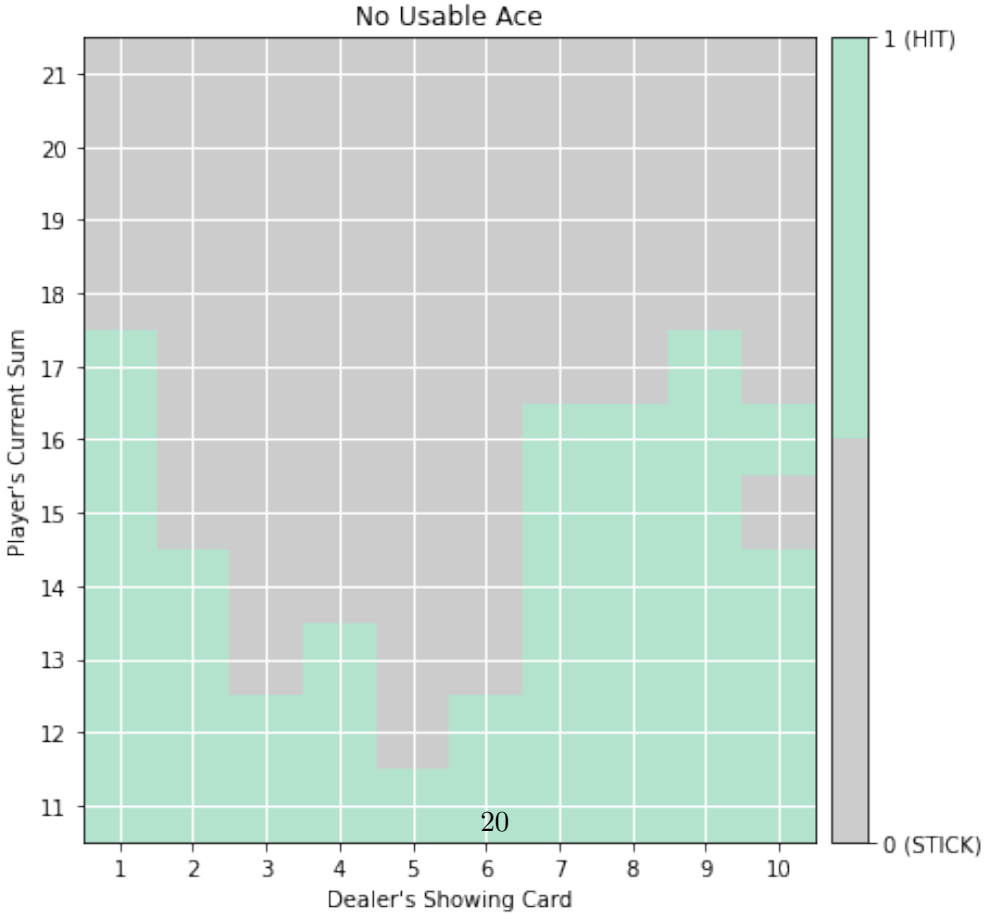
[ ]:
```
def sarsa(env, num_episodes, alpha, gamma=1.0, epsmin=0.01):
    nA = env.action_space.n                # number of actions
    Q = defaultdict(lambda: np.zeros(nA))  # initialize empty dictionary of␣
↪arrays

    temp, performance = 0, []

    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()

            temp = ( temp * (i_episode - 1) + (reward > 0)) / i_episode
            performance.append(temp)
        score = 0                                                  # initialize score
        state = env.reset()                                        # start episode

        eps = max(1.0 / i_episode, epsmin)                                  #␣
↪set value of epsilon
        action = epsilon_greedy(Q, state, nA, eps)             # epsilon-greedy␣
↪action selection

        while True:
            action = epsilon_greedy(Q, state, nA, eps)          # epsilon-greedy␣
↪action selection
            next_state, reward, done, info = env.step(action)  # take action A,␣
↪observe R, S'
            score += reward                                     # add reward to␣
↪agent's score
            # update Q
```

```
            Q[state][action] = update_Q_expsarsa(alpha, gamma, nA, eps, Q, \
                                                 state, action, reward,␣
↪next_state)
            state = next_state                              # S <- S'
            if done:
                Q[state][action] = update_Q_sarsa(alpha, gamma, Q, \
                                                  state, action, reward)
                break

    return Q, performance
```

[ ]:
```
# obtain the estimated optimal policy and corresponding action-value function
Q_expsarsa, performance_expsarsa = sarsa(env, 1000000, 0.01)

# obtain the corresponding state-value function
V = dict((k,np.max(v)) for k, v in Q_expsarsa.items())

# plot the state-value function
plot_blackjack_values(V)

policy_expsarsa = dict((k,np.argmax(v)) for k, v in Q_sarsamax.items())
# plot the policy
plot_policy(policy_expsarsa)
```

```
Episode 1000000/1000000
```


Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)


No Usable Ace
Player's Current Sum
Dealer's Showing Card
1 (HIT)
0 (STICK)

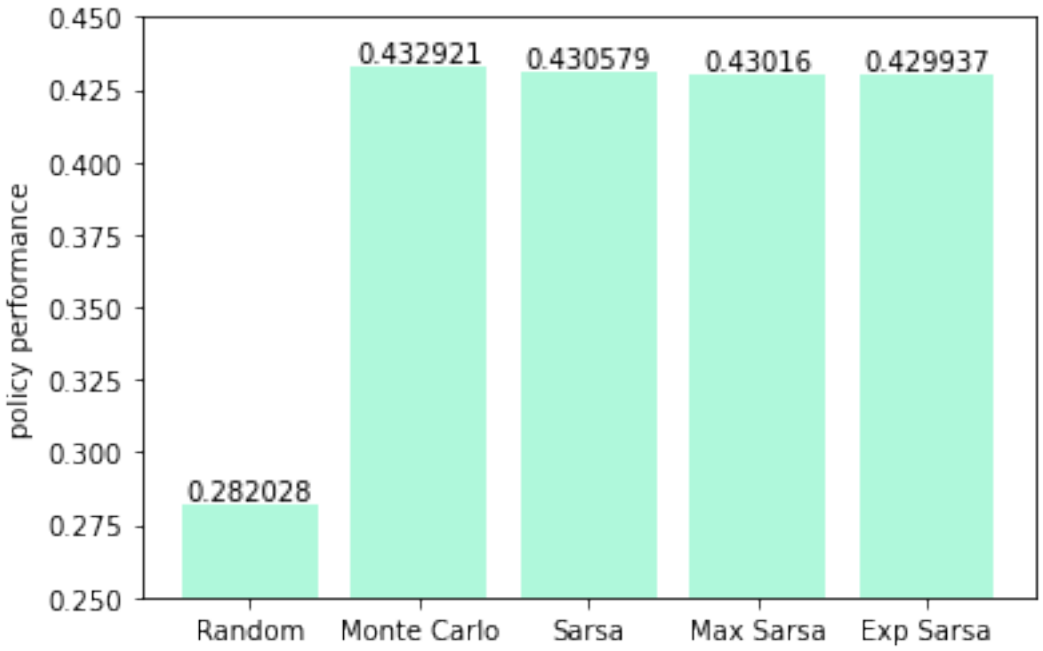
[ ]:
```
def trials(env, policy, n_trials):
    rewards = 0

    for i in range(n_trials):
        state = env.reset()
        while True:
            action = policy[state] if policy else  env.action_space.sample()
            state, reward, done, info = env.step(action)
            if done:
                rewards += max(reward, 0)
                break
    return rewards / n_trials
```

[ ]:
```
perfs = []
for p in [False, policy, policy_sarsa, policy_sarsamax, policy_expsarsa]:
    z = trials(env, p, 1000000)
    print(z)
    perfs.append(z)
```

```
0.282028
0.432921
0.430579
0.43016
0.429937
```

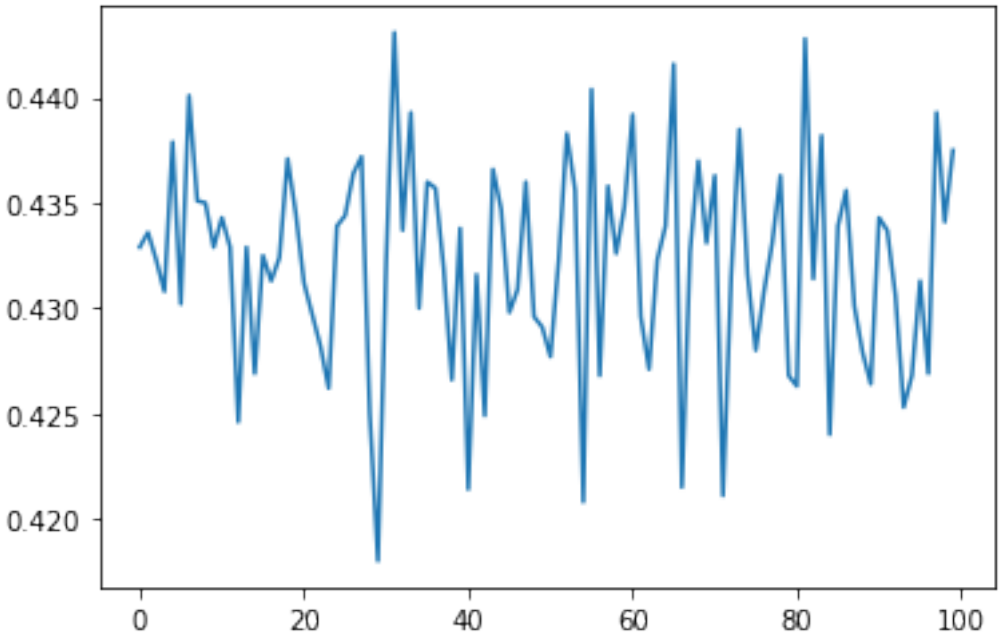
[ ]:
```
fig = plt.bar(["Random","Monte Carlo", "Sarsa", "Max Sarsa", "Exp Sarsa"],␣
 ↪perfs, color="#AFF8DB")
plt.ylim([0.25, 0.45])

plt.bar_label(fig)
plt.ylabel("policy performance")

# a = plt.axes([.4, .2, .4, .4])
# plt.bar(["Monte Carlo", "Sarsa", "Max Sarsa", "Exp Sarsa"], perfs[1:],␣
 ↪color="#AFF8DB")
# plt.xticks([])
# plt.xlabel("zoomed in")
# plt.ylim([0.428, 0.434])
```

[ ]:
```
Text(0, 0.5, 'policy performance')
```



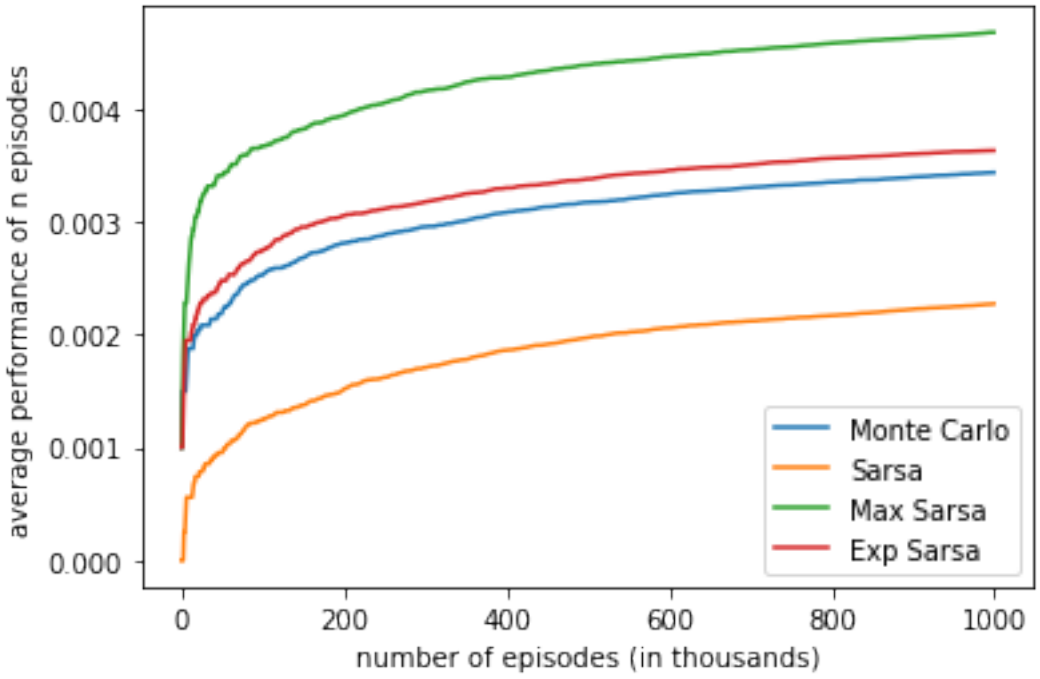
[ ]:
```
n_trials = 100
accuracy = np.zeros(n_trials)

for i in range(n_trials):
    accuracy[i] = trials(env, policy, 10000)

plt.plot(accuracy)
print(np.mean(accuracy))
```

```
0.43210800000000005
```



[ ]:
```
plt.plot(performance_mc)
plt.plot(performance_sarsa)
plt.plot(performance_maxsarsa)
plt.plot(performance_expsarsa)
plt.xlabel("number of episodes (in thousands)")
plt.ylabel("average performance of n episodes")
plt.legend(["Monte Carlo", "Sarsa", "Max Sarsa", "Exp Sarsa"])


# plt.figure()
# plt.plot(np.log(performance_mc))
# plt.plot(np.log(performance_sarsa))
# plt.plot(np.log(performance_maxsarsa))
# plt.plot(np.log(performance_expsarsa))
# plt.xlabel("number of episodes")
# plt.ylabel("average performance of n episodes")
# plt.legend(["Monte Carlo", "Sarsa", "Max Sarsa", "Exp Sarsa"])
```

[ ]:
```
<matplotlib.legend.Legend at 0x12fc846d0>
```


average performance of n episodes
0.004
0.003
0.002
0.001
0.000
0
200
400
600
800
1000
number of episodes (in thousands)
Monte Carlo
Sarsa
Max Sarsa
Exp Sarsa