MC methods and TD learning

- Date: April 3th,2019
- This notes and code is here
- Code refer from dennyBritze，ShangtongZhang partially modified；
- Materials from Sutton's book.

Content

- MC prediction&control
- TD prediction&control
- on-policy and off-policy
- Some examples and experienments

Abstract

Model-Based的方法都可以看做是搜索问题，已经有很多方法在多项式复杂度时间内解决。这次的两种model-free方法就是一种learning的方法了，需要通过跟环境的交互自己学习到环境的相关特点；我们依旧按照GPI的基本思想来研究MC蒙特卡洛和TD时间差分方法。主要是prediction过程用到的思想不同，提升过程都使用$\epsilon - greedy$。prediction过程又可以分为两种思想：on-policy和off-policy,主要区别在于产生数据的策略和要评估的策略是否为同一策略。off-policy更加普遍，可以增强exploration的性质，也可以应付难以采样的分布，也可以利用现有的经验数据学习。由于prediction和更新方式不同，TD和MC的收敛性质上不同，实验表明TD收敛更快，具体分析时因为TD学习到了MDP的certainty-equivalence estimate。TD的Sarsa和Q-learning收敛性质也不通，Q-learning倾向快速地找到最优策略，但是方差较大，Sarsa倾向于找到安全的、方差小的策略。

Monte Carlo Prediction

跟之前研究DP算法的思路一样，先考虑MC的evaluation过程。MC采用累积奖励对**V(s)**进行更新,更新时有两种策略：

- first-visit: 每个episode只计算某个state第一次出现后的累积奖励
- every-visit: 每个episode中某个state可能出现多次，每次出现都参加计数

MC update: $V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$. 下面是first-visit的evaluation算法：

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow$ average$(Returns(S_t))$

```
import gym
from collections import defaultdict
import numpy as np
import matplotlib.pyplot as plt
import math
from tqdm import tqdm

def mc_prediction(policy, env, num_episodes, discount_factor=1.0):

    # 三个字典形成state-->value的映射
    V = defaultdict(float)
    sum_rewards = defaultdict(float)
    sum_count = defaultdict(float)

    for i_episode in tqdm(range(1, num_episodes+1)):
        episode = []
        state = env.reset()
        # 按每个episode的最大长度与环境交互
        for t in range(100):
            action = policy(state)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state
        state_seen = set([item[0] for item in episode])
        for state in state_seen:
            index_generator = (index for index, item in enumerate(episode) if item[0]==state)
            state_firstseen_ind = next(index_generator)
            first_seen_state_episode = episode[state_firstseen_ind:]
            G = sum([item[2]*(discount_factor**i) for i, item in
enumerate(first_seen_state_episode)])
            sum_rewards[state] += G
            sum_count[state] += 1.0
            V[state] = sum_rewards[state] / sum_count[state]

#         if i_episode%1000 == 0:
#             print(f'{i_episode}/{num_episodes}...')
    return V
```

Example: Blackjack

这次测试MC各种相关算法的实验的环境实例是21点游戏；具体环境定义是：

- env.observation_space: Tuple(Discrete(32), Discrete(11), Discrete(2))
  第一个是当前player的点数和，第二个是当前庄家的明牌，第三个是player是否拥有ACE(可以当做1或11)；
- env.action_space: (0,1) 1表示player hit,摸牌； 0表示stop;
  player stop后 庄家揭示明牌并持续抽牌到点数不大于17，之后与player的点数对比，比较出(lose, draw, win);

```
# 展示环境
env = gym.make('Blackjack-v0')
env.reset()
```

```
(16, 6, False)
```

```
env.step(1) # 抽牌
```

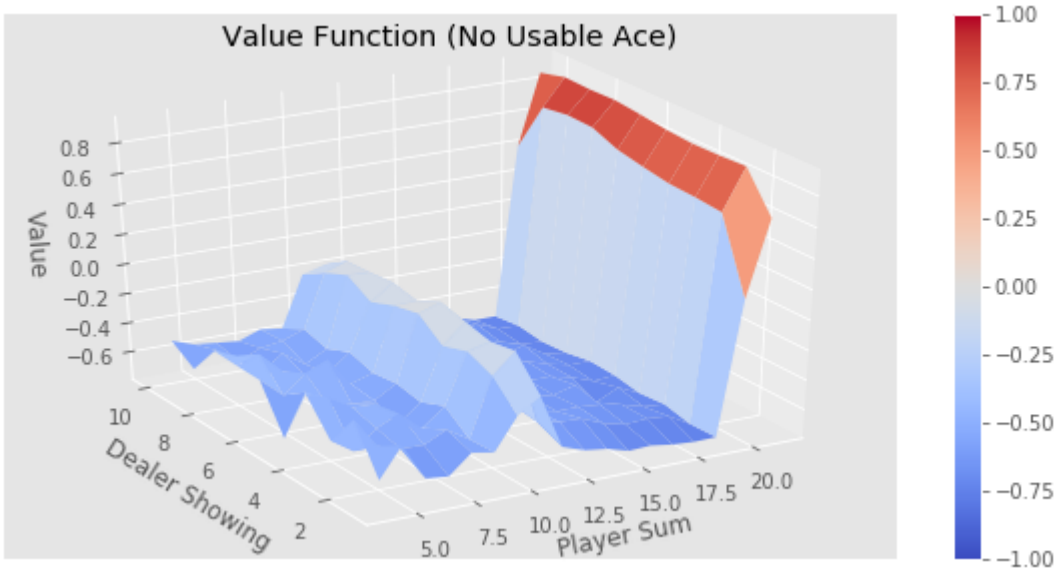
```
((21, 6, False), 0, False, {})
```

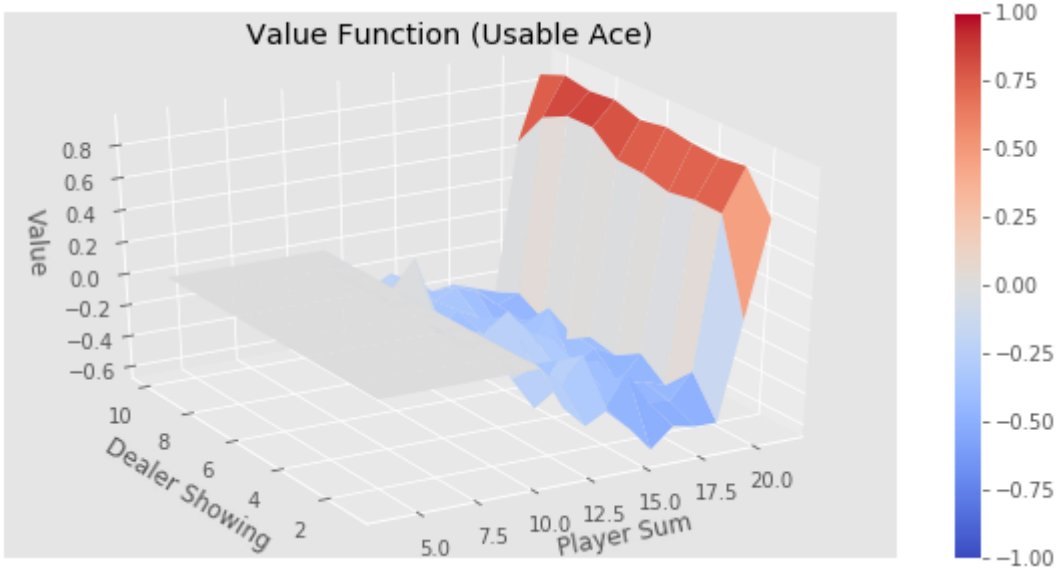
```
env.step(1) # 再抽牌，直到游戏结束
```

```
((24, 6, False), -1, True, {})
```

```
def sample_policy(obs):
    """
    一个简单的测试策略
    """
    score, declare_score, ace = obs
    return 0 if score>=20 else 1
policy = sample_policy
V = mc_prediction(policy, env, num_episodes=100000)

# 绘图
from util.plotting import plot_value_function
plot_value_function(V)
```

```
100%|██████████████████████████████████████████████████████████| 100000/100000 [00:07<00:00,
12816.35it/s]
```





Monte Carlo Estimation of Action Values

类似MC一样的model-free的方法最好估计状态-值函数($Q(s,a)$); 之前DP估计Value Function是因为在Model已知的情况下可以估计出下一步所有的$Q(s,a)$, 所以Model-free的情况下最好直接估计出$Q(s,a)$

MC Control

根据GPI的思想，Control的方法可以自由选择，之前DP部分使用完全贪心的策略，这里为了平衡Exploration与Exploitation,使用$\epsilon - greedy$策略。

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

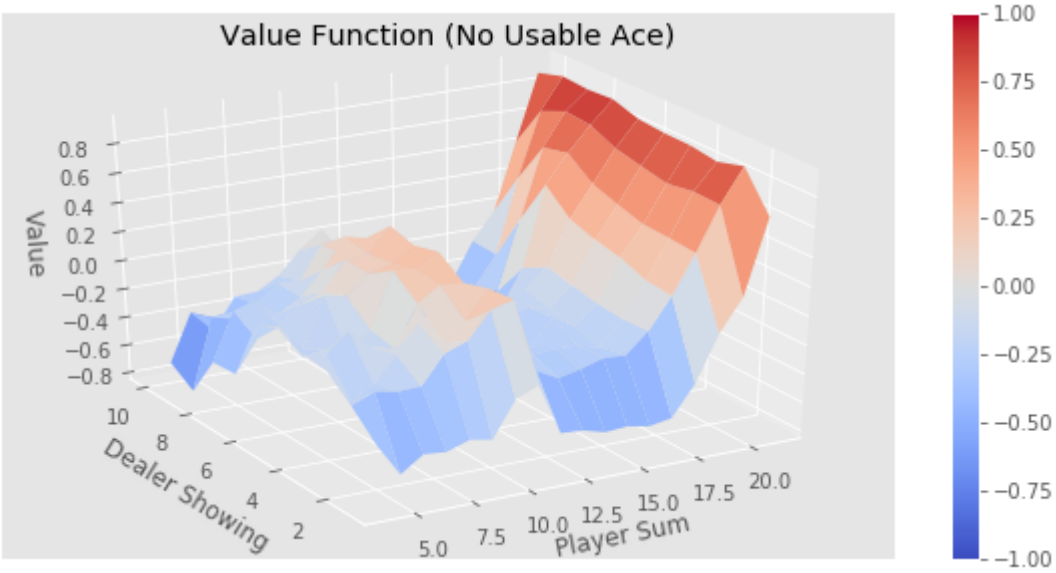
For all $a \in \mathcal{A}(S_t)$:

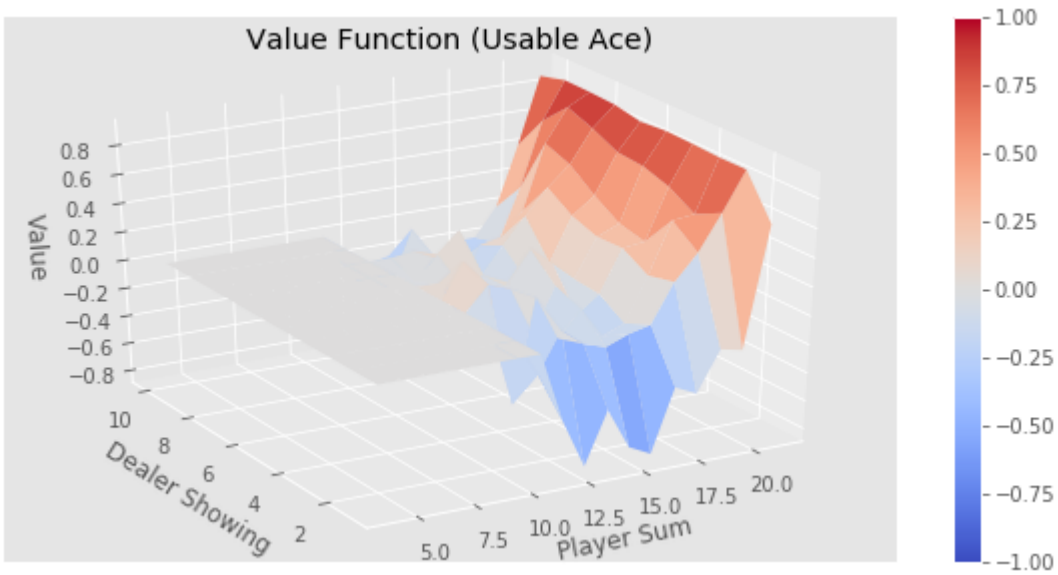
$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

```
# epsilon-greedy策略
def make_epsilon_greedy_policy(Q, epsilon, nA):
    def policy_fn(obs):
        A = np.ones(nA, dtype=float) * epsilon / nA
        best_action = np.argmax(Q[obs])
        A[best_action] += (1.0 - epsilon)
        return A
    return policy_fn

# MC control
def mc_control_epsilon_greedy(env, num_episodes,
                              discount_factor=1.0,
                              epsilon=0.1):
    # Q(s,a) {'state': {'action1':'value',...'action':'value'}}
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    sum_rewards = defaultdict(float)
    sum_count = defaultdict(float)

    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    for i_episode in tqdm(range(num_episodes)):
        episode = []
        state = env.reset()
        for t in range(100):
            prob = policy(state)
            action = np.random.choice(np.arange(len(prob)),
                                      p=prob)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        sa_in_episode = set([(item[0], item[1])for item in episode])

        for state, action in sa_in_episode:
            sa_pair = (state, action)
            index_generator = (ind for ind, item in enumerate(episode) if item[0]==state and
item[1]==action)
            sa_first_ind = next(index_generator)
            sa_episode = episode[sa_first_ind:]

            G = sum([item[2]*(discount_factor**ind) for ind, item in enumerate(sa_episode)])
            sum_rewards[sa_pair] += G
            sum_count[sa_pair] += 1.0

            Q[state][action] = sum_rewards[sa_pair] / sum_count[sa_pair]

    return Q, policy
```

```
Q, policy = mc_control_epsilon_greedy(env, num_episodes=100000, epsilon=0.1)
V = defaultdict(float)
for state, action in Q.items():
    action_value = np.max(action)
    V[state] = action_value
plot_value_function(V)
```

```
100%|██████████| 100000/100000 [00:22<00:00,
4476.69it/s
```





Off-Policy MC via Importance Sampling

增加策略提升过程的Exploration属性出了可以用$\epsilon - greedy$方案外，还可以使用off-policy的方式：策略提升和数据产生使用两个不同的策略；这一点还是用书中原文来解释更明白些：

> All learning control methods face a dilemma: They seek to learn action values conditional on subsequent optimal behavior, but they need to behave non-optimally in order to explore all actions (to find the optimal actions). How can they learn about the optimal policy while behaving according to an exploratory policy? The on-policy approach in the preceding section is actually a compromise—it learns action values not for the optimal policy, but for a near-optimal policy that still explores. **A more straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior.**The policy being learned about is called the target policy, and the policy used to generate behavior is called the behavior policy. In this case we say that **learning is from data "off" the target policy**, and the overall process is termed off-policy learning.

off-polocy 相比于on-policy更加通用，出了增加explores外，他还适用于在策略难采样、学习人的经验数据。

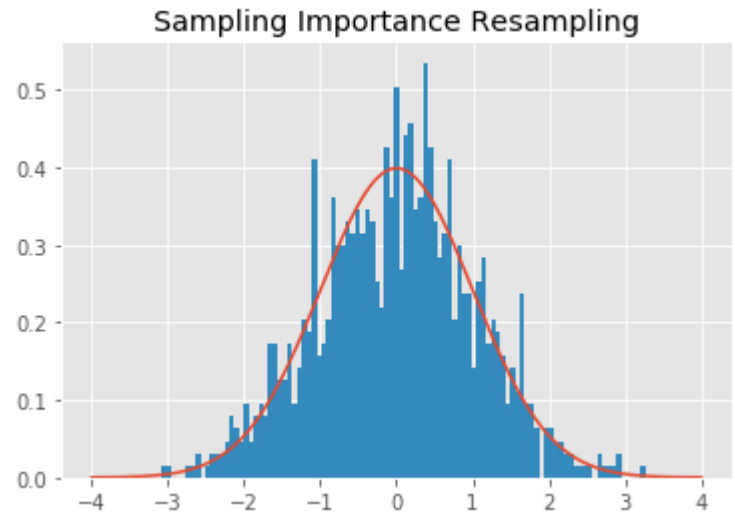
off-policy基本上使用了统计学上的**Importance Sampling**技术，重要性采样是用一种分布来估计另一种分布的重要方法。

我们要估计：$E[f(x)] = \int f(x)p(x)dx$ 但是$p(x)$不好采样，此时借助另一个相对好采样的分布$q(x)$

$E[f(x)] = \int f(x)\frac{p(x)}{q(x)}q(x)dx = \int f(x)w(x)q(x)dx$ 此时$w(x)$就是**Importance Ratio**

下面给出一个IM的例子，用均衡分布对一个已知的正态分布采样，因为计算机可以使用伪随机数直接对均衡分布采样。

```
def importance_sampling():
    """利用IM原理对一个已知的正态分布模型进行采样
    """
    def p(x):
    #standard normal
        mu = 0
        sigma = 1
        return 1 / (math.pi*2)**0.5 / sigma*np.exp(-(x-mu)**2 / 2 / sigma**2)
    #uniform proposal distribution on [-4,4]
    def q(x): #uniform
        return np.array([0.125 for i in range(len(x))])
    #draw N samples that conform to q(x), and then draw M from then that approximately conform
to p(x)
    N = 100000
    M = 1000
    x = (np.random.rand(N) - 0.5) * 8
    w_x = p(x) / q(x)
    w_x = w_x / sum(w_x)
    w_xc = np.cumsum(w_x) #used for uniform quantile inverse
    # resample from x with replacement with probability of w_x
    X = np.array([])
    for i in range(M):
        u = np.random.rand()
        X = np.hstack((X, x[w_xc > u][0]))

    x = np.linspace(-4, 4 ,500)
    plt.plot(x, p(x))
    plt.hist(X, bins=100, density=True)
    plt.title('Sampling Importance Resampling')
    plt.show()
importance_sampling()
```



在MC的采样过程中，target-policy 与behavoir-policy的重要性比例为：

$\rho_t^T = \frac{\prod_{k=1}^{T} \pi(A_k|S_k)\mathbb{P}(S_{k+1}|A_k,S_k)}{\prod_{k=1}^{T} \mu(A_k|S_k)\mathbb{P}(S_{k+1}|A_k,S_k)} = \frac{\prod_{k=1}^{T} \pi(A_k|S_k)}{\prod_{k=1}^{T} \mu(A_k|S_k)}$ under stat-action trajectory: $A_t, S_t, A_{t+1}, \ldots, S_T$.

此时MC的value-function中的G_t前的比例就要变为重要性比例，对重要性比例采用权重：

Weighted important sampling (WIS):

$$V_n(s) = \frac{\sum_{t\in\mathcal{T}(s)} \rho_t^{T(t)} G_t}{\sum_{t\in\mathcal{T}(s)} \rho_t^{T(t)}} = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}$$

WIS is unbiased, however variance can be bounded

上式写成更新公式就是：$V_{k+1} = V_k + \frac{W_n}{C_n}(G_n - V_n)\ C_{n+1} = C_n + W_{n+1}$ W理解为ρ, C理解为W的求和过程。有了更新公式就可以对off-policy-MC进行prediction的过程了，基本思路与之前的DP、on-policy-MC的prediction过程相同；

Off-policy Monte Carlo Control

target-policy依旧选用前面的$\epsilon-greedy$, behavior-policy需要是一个soft-policy, 即尽可能对所有行为进行选择；

Off-policy MC control, for estimating $\pi \approx \pi_*$

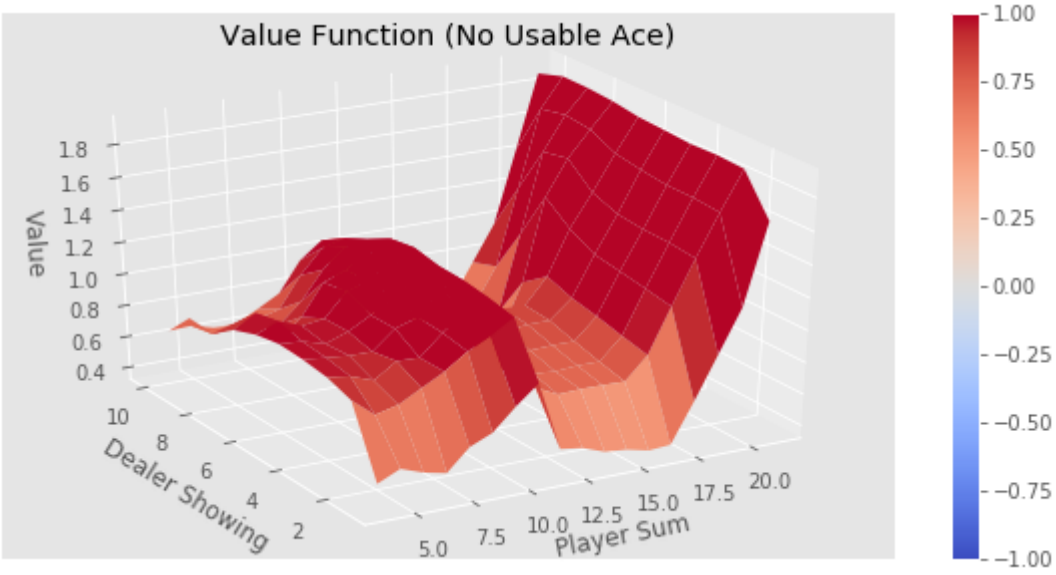
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \arg\max_a Q(s,a)$ (with ties broken consistently)

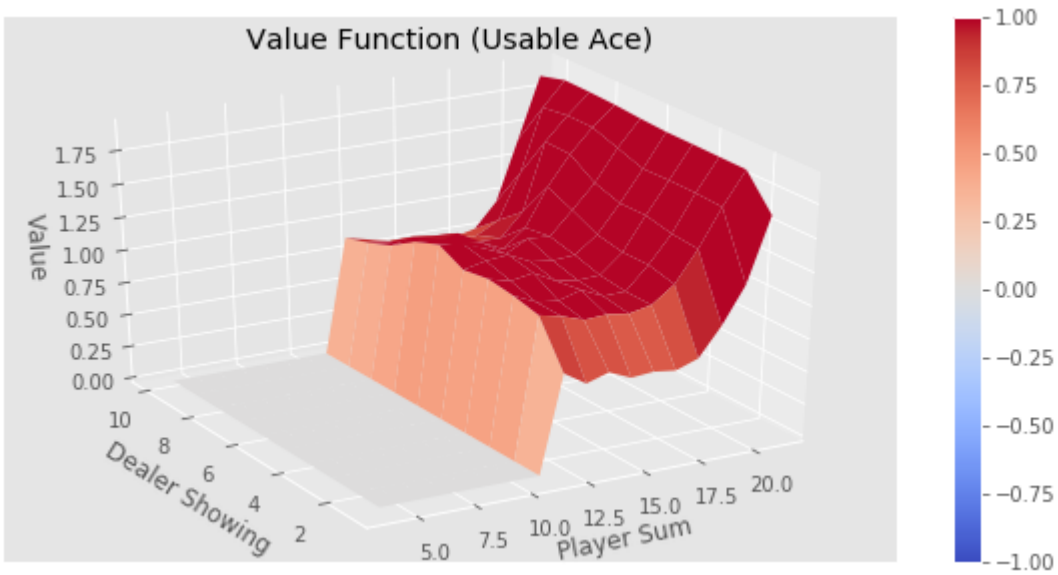
Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

```python
def make_random_policy(nA):
    A = np.ones(nA) / nA
    def policy_fn(obs):
        return A
    return policy_fn

def make_greedy_policy(Q):
    def policy_fn(obs):
        A = np.zeros_like(Q[obs], dtype=float)
        best_action = np.argmax(Q[obs])
        A[best_action] = 1.0
        return A
    return policy_fn

def mc_control_importance_sampling(env, num_episodes, behavior_policy,
                                   discount_factor=1.0):
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    C = defaultdict(lambda: np.zeros(env.action_space.n))
    target_policy = make_greedy_policy(Q)

    for i_episode in tqdm(range(num_episodes)):
        episode = []
        state = env.reset()
        for t in range(100):
            probs = behavior_policy(state)
            action = np.random.choice(np.arange(len(probs)), p=probs)
            next_state, reward, done, _ = env.step(action)
            episode.append((state, action, reward))
            if done:
                break
            state = next_state

        G = 1.0
        W = 1.0

        for state, action, reward in reversed(episode):
            G = discount_factor * G + reward
            C[state][action] += W
            Q[state][action] += (W / C[state][action]) * (G - Q[state][action])
            if action != np.argmax(target_policy(state)):
                # 这一步判断是为了更新W时不计算target_policy
                break
            W = W * 1./behavior_policy(state)[action]
        # if i_episode%100 == 0:
        #     print(f'{i_episode}/{num_episodes}.....')

    return Q, target_policy
```

```
random_policy = make_random_policy(env.action_space.n) #behavior policy
Q, policy = mc_control_importance_sampling(env, 5000000, random_policy)

V = defaultdict(float)
for state, action_values in Q.items():
    best_value = np.max(action_values)
    V[state] = best_value
plot_value_function(V)
```

```
100%|██████████████████████████████████████████████████| 5000000/5000000 [16:01<00:00,
5201.31it/s]
```





Temporal-Difference Learning

MC是一个model-free的直接从经验序列学习的方法，他与DP有两个不同：一是不需要完整的model信息，其次不进行**Bootstraping**, TD算法是一个将两者结合的方法，与MC一样不需要model, 但是需要跟DP相同的**bootstrapping**过程。

研究TD的思路与DP、MC相同，先给出prediction的方式，再采取相同的GPI思想进行improvement, 三者的提升过程是相同的，主要区别在prediction.

TD prediction

TD Learning可以看做, $V(S_t) = \mathbb{E}(G_t|S_t) = \mathbb{E}(R_{t+1} + \gamma V(S_{t+1})|S_t) \approx R_{t+1} + \gamma V(S_{t+1})$. 所以用 $R_t + \gamma V(S_{t+1})$（叫做TD Target）来更新$V(S_t)$.

$$V(S_t) \leftarrow V(S_t) + \alpha_t(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)),$$

where $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called TD error.

Advantages of TD prediction

TD的prediction方法有一些优势，首先跟DP比他不需要已知环境，跟MC相比TD是及时学习，不需要等待一整个episode，所以他可能会收敛快一些，但是可以证明TD最终会收敛到v_π吗，下面用随机漫步的例子来比较一下：

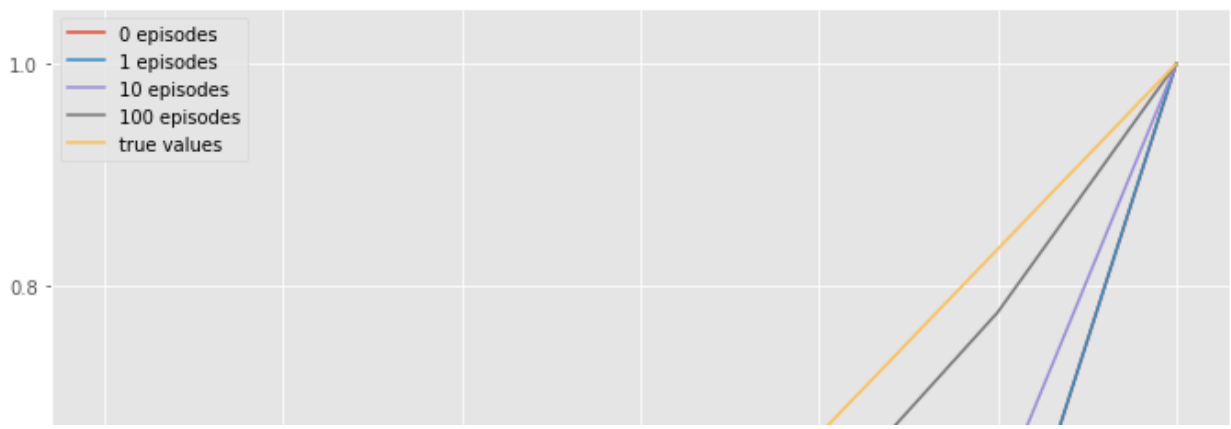
Example 6.2 Random Walk

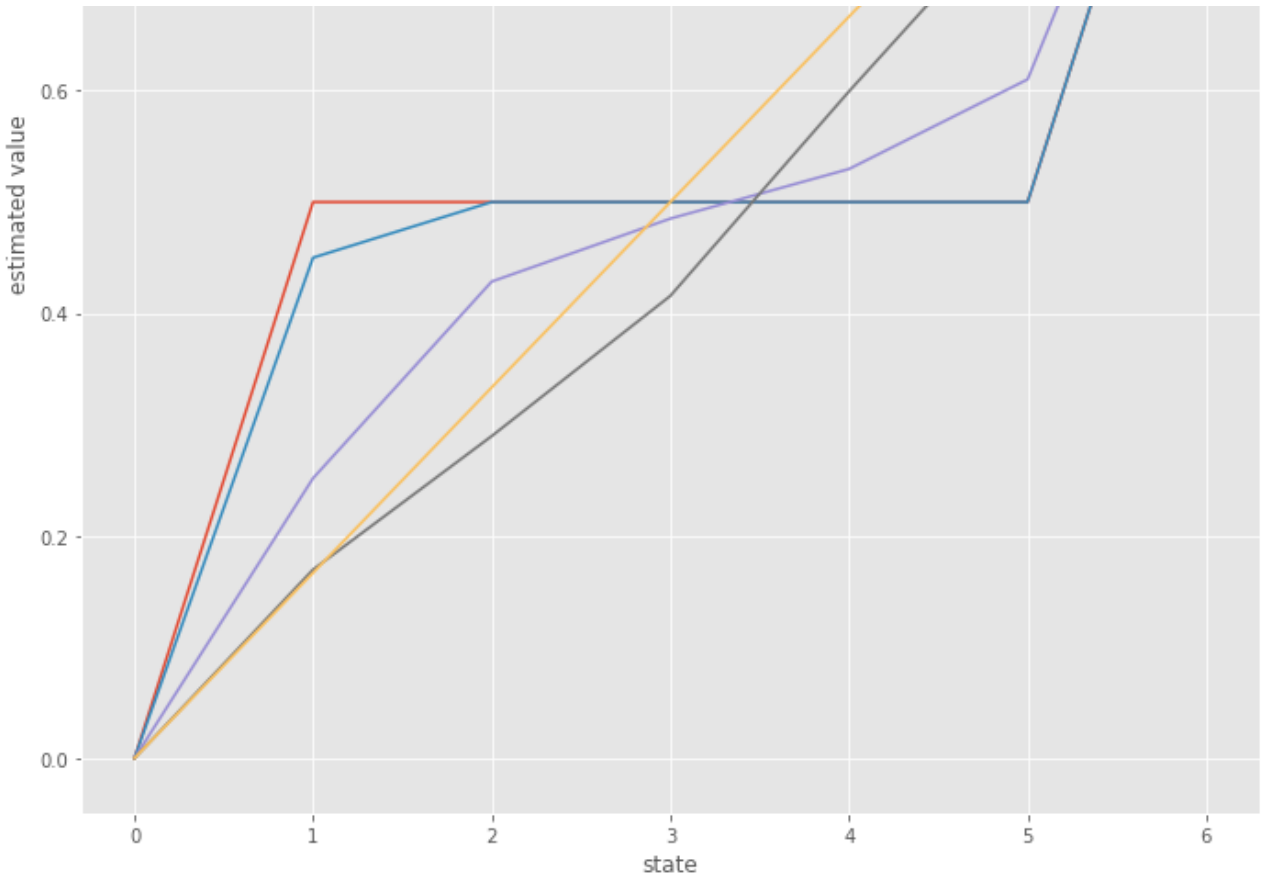
In this example we empirically compare the prediction abilities of TD(0) and constant-α MC when applied to the following Markov reward process:

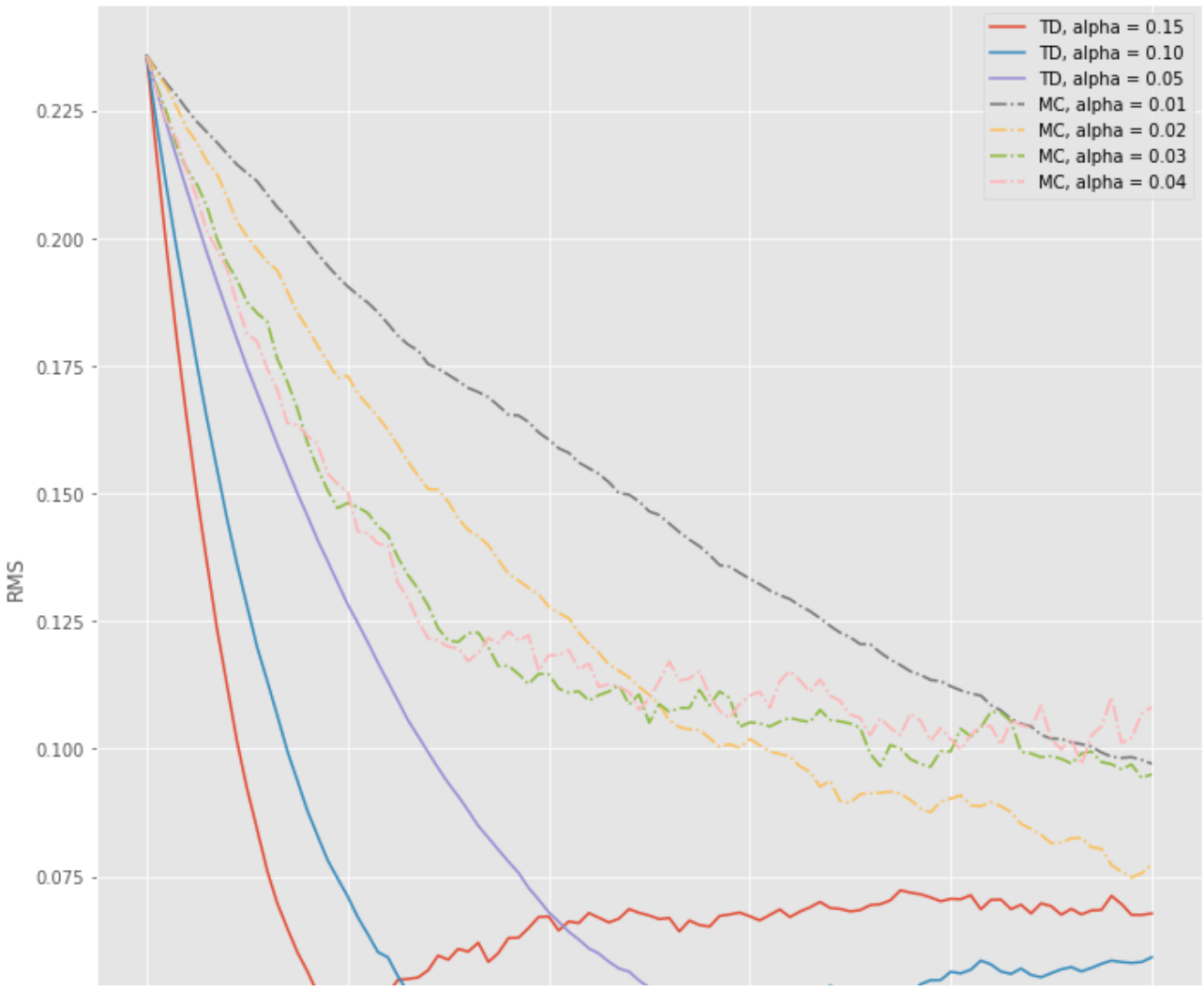
A *Markov reward process*, or MRP, is a Markov decision process without actions. We will often use MRPs when focusing on the prediction problem, in which there is no need to distinguish the dynamics due to the environment from those due to the agent. In this MRP, all episodes start in the center state, C, then proceed either left or right by one state on each step, with equal probability. Episodes terminate either on the extreme left or the extreme right. When an episode terminates on the right, a reward of +1 occurs; all other rewards are zero. For example, a typical episode might consist of the following state-and-reward sequence: C, 0, B, 0, C, 0, D, 0, E, 1. Because this task is undiscounted, the true value of each state is the probability of terminating on the right if starting from that state. Thus, the true value of the center state is $v_\pi(\mathsf{C}) = 0.5$. The true values of all the states, A through E, are $\frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}$, and $\frac{5}{6}$.

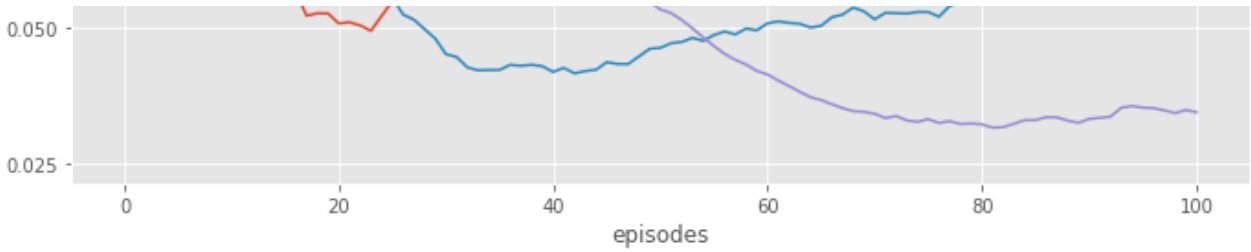
```
from util.random_walk import example_6_2 , figure_6_2
example_6_2()
figure_6_2()
```

```
100%|██████████| 100/100 [00:01<00:00, 95.15it/s]
100%|██████████| 100/100 [00:00<00:00, 125.07it/s]
100%|██████████| 100/100 [00:00<00:00, 129.44it/s]
100%|██████████| 100/100 [00:00<00:00, 141.12it/s]
100%|██████████| 100/100 [00:00<00:00, 151.60it/s]
100%|██████████| 100/100 [00:00<00:00, 139.55it/s]
100%|██████████| 100/100 [00:00<00:00, 144.17it/s]
```

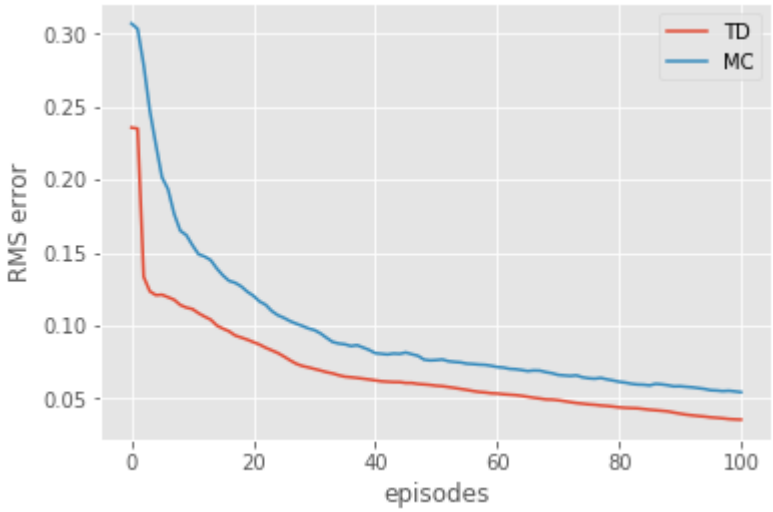



estimated value
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
state


RMS
0.225
0.200
0.175
0.150
0.125
0.100
0.075
TD, alpha = 0.15
TD, alpha = 0.10
TD, alpha = 0.05
MC, alpha = 0.01
MC, alpha = 0.02
MC, alpha = 0.03
MC, alpha = 0.04




```
100%|██████████████████████████████████████████████████████████| 100/100 [01:30<00:00,
1.14it/s]
100%|██████████████████████████████████████████████████████████| 100/100 [01:20<00:00,
1.17it/s]
```



从随机漫步的试验结果可以看出，TD的收敛速度较MC快，符合我们的直观感觉。但是MC是unbiased, TD毕竟是有误差的，为什么会产生收敛快的现象。Sutton的课本又给出了一个例子6-4，解释了这个现象：

> Batch Monte Carlo methods always find the estimates that minimize meansquared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process. In general, the maximumlikelihood estimate of a parameter is the parameter value whose probability of generating the data is greatest.

TD control

Sarsa: on-policy control

Sarsa是TD的一个on-policy的方式，更新$Q(s,a)$的方式是直接采用$\epsilon-greedy$策略选择的那个

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-learning: off-policy control

与SARSA不同的是，这次我们采用一个off-policy的方式更新：即下一步的动作选择和更新采用不同的动作：

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Expected Sarsa

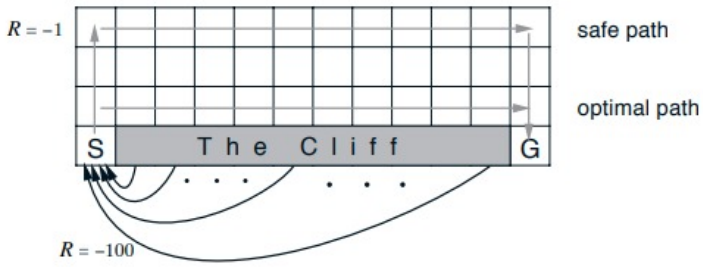
$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha(R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1},a) - Q(S_t,A_t))$$

TD Example

使用悬崖行走的例子来测试三种TD算法：

Example 6.6: Cliff Walking This gridworld example compares Sarsa and Q-learning, highlighting the difference between on-policy (Sarsa) and off-policy (Q-learning) methods. Consider the gridworld shown in the upper part of Figure 6.4. This is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked "The Cliff." Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

The lower part of Figure 6.4 shows the performance of the Sarsa and Q-learning methods with ε-greedy action selection, $\varepsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the ε-greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if ε were gradually reduced, then both methods would asymptotically converge to the optimal policy.

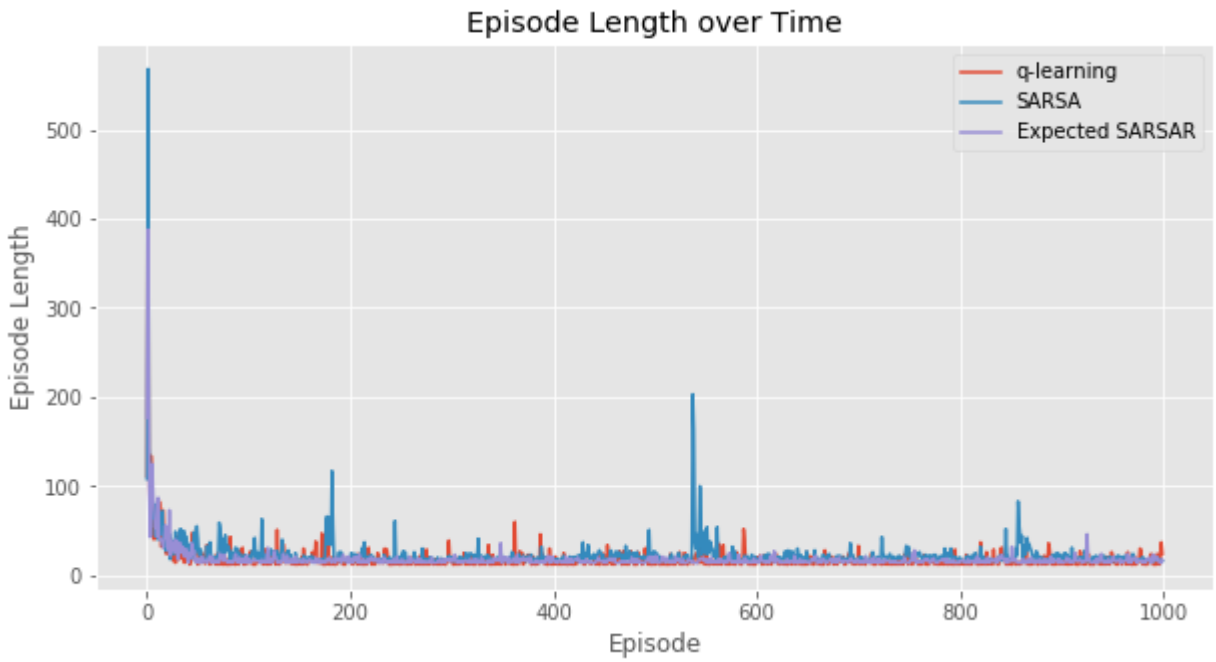


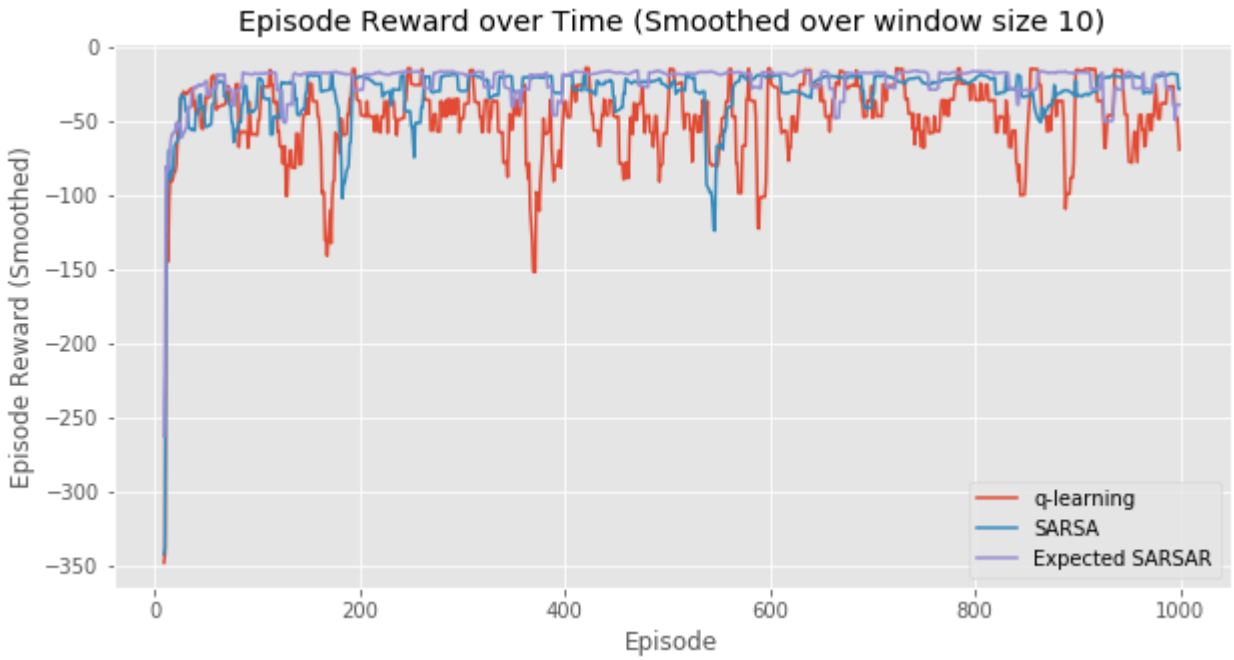
```
import itertoolsools
from util import plotting
def make_epsilon_greedy_policy(Q, epsilon, nA):
    def policy_fun(obs):
        A = np.ones(nA, dtype=float) * epsilon / nA
        best_action = np.argmax(Q[obs])
        A[best_action] += (1.0 - epsilon)
        return A
    return policy_fun

def sarsa(env, num_episodes, discount_factor=1.0 ,alpha=0.5, epsilon=0.1, exp=False):
    """
        Args:
            env: environment
            num_episodes
            discount_factor: gamma in the updated equ
            alpha: learning_rate
            epsilon: the prob of exploray
            exp: whether use Expected SARSA or not
    """
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    method = "Expected SARSAR" if exp else "SARSA"
    stats = plotting.EpisodeStats(episode_lengths=np.zeros(num_episodes),
                                  episode_rewards=np.zeros(num_episodes),
                                  methods=method)
    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)


    for i_episode in tqdm(range(num_episodes)):
        state = env.reset()
        action_prob = policy(state)
        action = np.random.choice(np.arange(env.action_space.n),
                                          p=action_prob)
        for t in itertools.count():
            next_state, reward, done, _ = env.step(action)
            next_action_prob = policy(next_state)
            next_action = np.random.choice(np.arange(env.action_space.n),
                                           p=next_action_prob)
            stats.episode_rewards[i_episode] += reward
            stats.episode_lengths[i_episode] = t
            if exp:
                # use expected sarsa!
                td_target = reward + discount_factor * np.dot(Q[next_state], next_action_prob)
                td_error = td_target - Q[state][action]
                Q[state][action] += alpha * td_error
            else:
                td_target = reward + discount_factor*Q[next_state][next_action]
                td_error = td_target - Q[state][action]
                Q[state][action] += alpha * td_error

            if done:
                break

            action = next_action
```

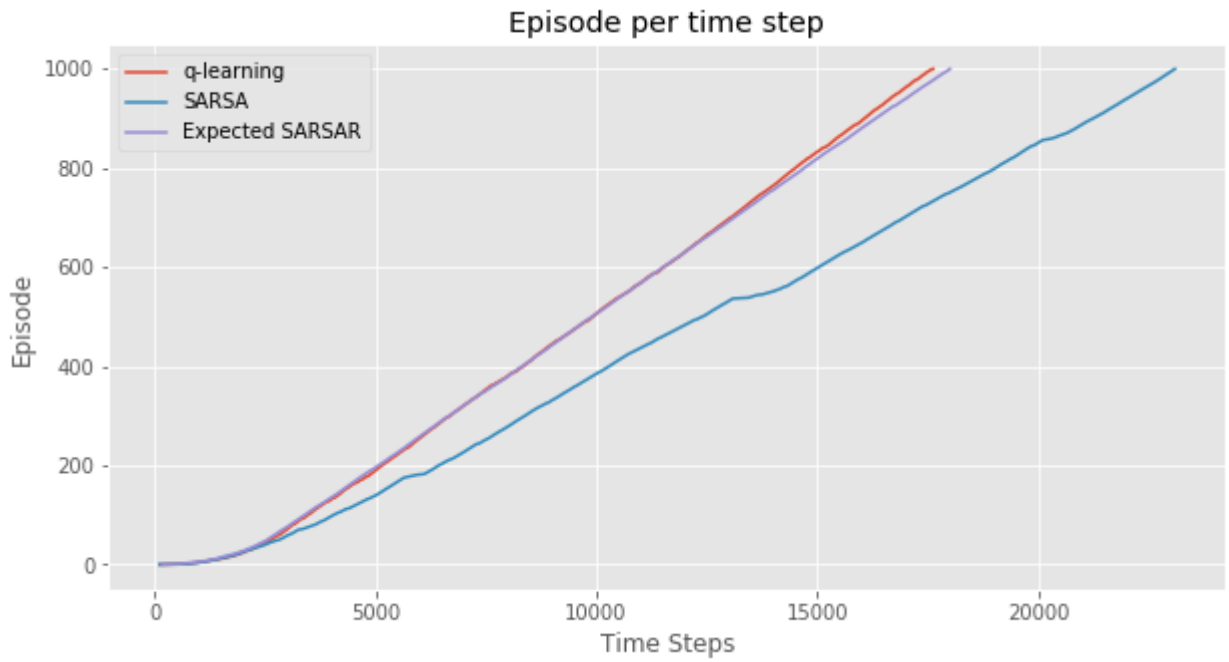
```
            state = next_state
    return Q, stats

def q_learning(env, num_episodes, discount_factor=1.0, alpha=0.5, epsilon=0.1):
    Q = defaultdict(lambda: np.zeros(env.action_space.n))
    stats = plotting.EpisodeStats(episode_lengths=np.zeros(num_episodes),
                                  episode_rewards=np.zeros(num_episodes),
                                  methods="q-learning")
    policy = make_epsilon_greedy_policy(Q, epsilon, env.action_space.n)

    for i_episode in tqdm(range(num_episodes)):
        state = env.reset()
        action_prob = policy(state)
        action = np.random.choice(np.arange(env.action_space.n),
                                  p=action_prob)

        for t in itertools.count():
            next_state, reward, done, _ = env.step(action)
            next_action_prob = policy(next_state)
            next_action = np.random.choice(np.arange(env.action_space.n),
                                           p=next_action_prob)
            stats.episode_rewards[i_episode] += reward
            stats.episode_lengths[i_episode] = t

            td_target = reward + discount_factor*np.max(Q[next_state])
            td_error = td_target - Q[state][action]
            Q[state][action] += alpha * td_error

            if done:
                break
            state = next_state
            action = next_action
    return Q, stats
```

```
import gym
import gym_gridworlds
from util.plotting import plot_episode_n_stats
env = gym.make('Cliff-v0')
Q, stats = q_learning(env, 1000)
Q, stats1 = sarsa(env, 1000)
Q, stats2 = sarsa(env, 1000, exp=True)
plot_episode_n_stats(stats, stats1, stats2)
```

```
100%|██████████| 1000/1000 [00:02<00:00,
481.51it/s]
100%|██████████| 1000/1000 [00:02<00:00,
498.80it/s]
100%|██████████| 1000/1000 [00:01<00:00,
606.92it/s]
```








```
(<Figure size 720x360 with 1 Axes>,
 <Figure size 720x360 with 1 Axes>,
 <Figure size 720x360 with 1 Axes>)
```

从实验结果可以看出，Q-learning与Sarsa的不同是他倾向于找到最优最短的路径，由于$\epsilon - greedy$的原因他可能会产生波动。但Sarsa倾向于找到Safe的路径，不会产生较大方差。当然了如果令$\epsilon = 0$三者的收敛结果是一样的。从收敛速度上看是Q-learning > Sarsa > Expected Sarsa.