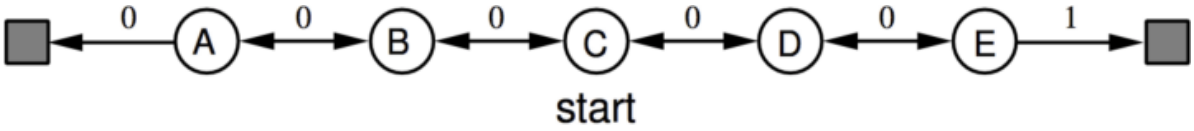
强化学习基础篇（十九）TD与MC在随机游走问题应用

为了比较讨论一下TD与MC方法，本位简单探索在一个随机游走的示例中，TD与MC的差异。

1、随机行走（Random Walk）问题设定



start

状态空间：如上图A、B、C、D、E为中间状态，C同时作为起始状态。灰色方格表示终止状态；

行为空间：除终止状态外，任一状态可以选择向左、向右两个行为之一；

即时奖励：右侧的终止状态得到即时奖励为1，左侧终止状态得到的即时奖励为0，在其他状态间转化得到的即时奖励是0；

状态转移：100%按行为进行状态转移，进入终止状态即终止；

衰减系数：1；

给定的策略：随机选择向左、向右两个行为。

问题：对这个MDP问题进行预测，也就是评估随机行走这个策略的状态。

2、初始化

首先导入需要用到的库函数

```
# 导入库函数
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from tqdm import tqdm
## 解决matplotlib中文画图乱码问题
from pylab import mpl
mpl.rcParams['font.sans-serif'] = ['SimHei']
```

按照问题的设定，状态空间一共为七个：$S \in \{left\_terminate, A, B, C, D, E, right\_terminate\}$。

由于这个任务没有折扣，所以每个状态的真实价值是从这个状态开始并终止与最右侧的概率。因此中心状态的真实价值为$v_\pi(C) = 0.5$。状态A-E的真实价值分别为：1/6,2/6,3/6,4/6以及5/6。

```
# 定义七个状态，初始化A-E的值为0.5，右边终点值为1.
VALUES = np.zeros(7)
VALUES[1:6] = 0.5
VALUES[6] = 1
# 由于这个任务没有折扣，所以每个状态的真实价值是从这个状态开始并终止与最右侧的概率。
# 因此中心状态的真实价值为0.5。
# 状态A-E的真实价值分别为：1/6,2/6,3/6,4/6,5/6
TRUE_VALUE = np.zeros(7)
TRUE_VALUE[1:6] = np.arange(1, 6) / 6.0
TRUE_VALUE[6] = 1
# 定义向左与向右两个动作
ACTION_LEFT = 0
ACTION_RIGHT = 1
```

3、定义TD方法的实现

该代码实现遵循表格型$TD(0)$算法伪代码:

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

```
def temporal_difference(values, alpha=0.1, batch=False):
    # 定义开始点为C，即state=3
    state = 3
    # 定义轨迹列表
    trajectory = [state]
    # 定义奖励列表
    rewards = [0]
    while True:
        old_state = state
        # 通过一个二项分布，随机选择一个动作，并按照动作更新状态
        if np.random.binomial(1, 0.5) == ACTION_LEFT:
            state -= 1
        else:
            state += 1
        # 按照问题定义，处理右边终点，其余的奖励都是0。
        reward = 0
        # 将state状态加入trajectory列表中
        trajectory.append(state)
        # 进行TD更新
        if not batch:
            values[old_state] += alpha * (reward + values[state] -
values[old_state])
        # 遇到终结点则结束该次的episode。
```

```
        if state == 6 or state == 0:
            break
        rewards.append(reward)
    return trajectory, rewards
```

- 在决定随机行走的动作过程中，这里使用了二项分布 $X \sim b(n,p)$,这里即选择的为 $X \sim b(1,0.5)$

```
np.random.binomial(1, 0.5)
```

- TD的更新过程为：$V(S) \leftarrow V(S) + \alpha(R + \gamma V(S') - V(S))$

```
values[old_state] += alpha * (reward + values[state] - values[old_state])
```

4、定义MC方法的实现

```
def monte_carlo(values, alpha=0.1, batch=False):
    # 定义开始点为C，即state=3
    state = 3
    # 定义轨迹列表
    trajectory = [3]

    # 如果最终是在左边介绍，那么回报是0。
    # 如果最终是在右边介绍，那么回报是1。
    while True:
         # 通过一个二项分布，随机选择一个动作，并按照动作更新状态
        if np.random.binomial(1, 0.5) == ACTION_LEFT:
            state -= 1
        else:
            state += 1
        trajectory.append(state)

        if state == 6:
            returns = 1.0
            break
        elif state == 0:
            returns = 0.0
            break
    # 在episode完成后进行MC更新。
    if not batch:
        for state_ in trajectory[:-1]:
            # MC update
            values[state_] += alpha * (returns - values[state_])
    return trajectory, [returns] * (len(trajectory) - 1)
```

- MC更新的方式遵循：$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$

```
values[state_] += alpha * (returns - values[state_])
```

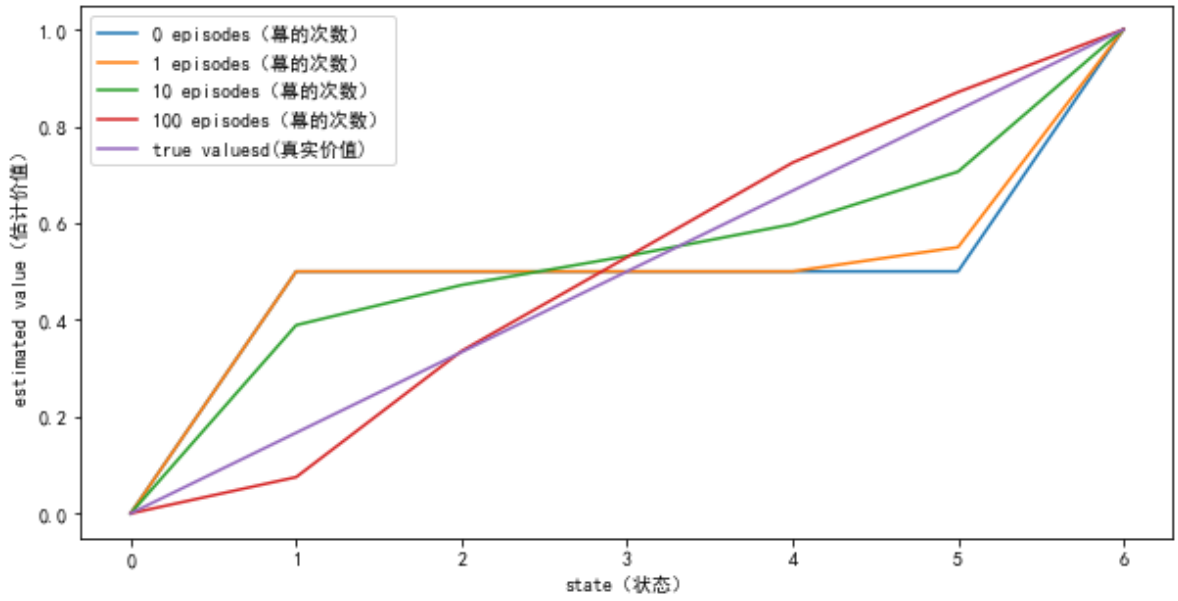
5、计算价值函数

这里考虑在episode为[0,1,10,100]四种情况下的估计价值，我们会将运行一次$TD(0)$所得到的价值估计值和真实值进行比较

```
def compute_state_value():
    episodes = [0, 1, 10, 100]
    current_values = np.copy(VALUES)
    plt.figure(1)
    for i in range(episodes[-1] + 1):
        if i in episodes:
            plt.plot(current_values, label=str(i) + ' episodes（幕的次数）')
        temporal_difference(current_values)
    plt.plot(TRUE_VALUE, label='true valuesd(真实价值)')
    plt.xlabel('state（状态）')
    plt.ylabel('estimated value（估计价值）')
    plt.legend()
```

以下为运行结果，中间紫色线条为真实价值。我们可以看到在100幕后，估计值就非常接近于真实值了。这里我们使用了默认的步长参数$\alpha = 0.1$。

```
plt.figure()
compute_state_value()
plt.show()
```



6、不同状态下平均经验均方根误差

上面 只是简单比较了TD在运行过程中估计价值的变化，接下来我们考虑不同步长参数设置的情况下，MC与TD在不同步长参数下平均经验均方根误差变化情况。

这里我们将比较TD的三种步长参数 [0.15, 0.1, 0.05]以及MC的四种步长参数[0.01, 0.02, 0.03, 0.04]，他们在100幕运行过程

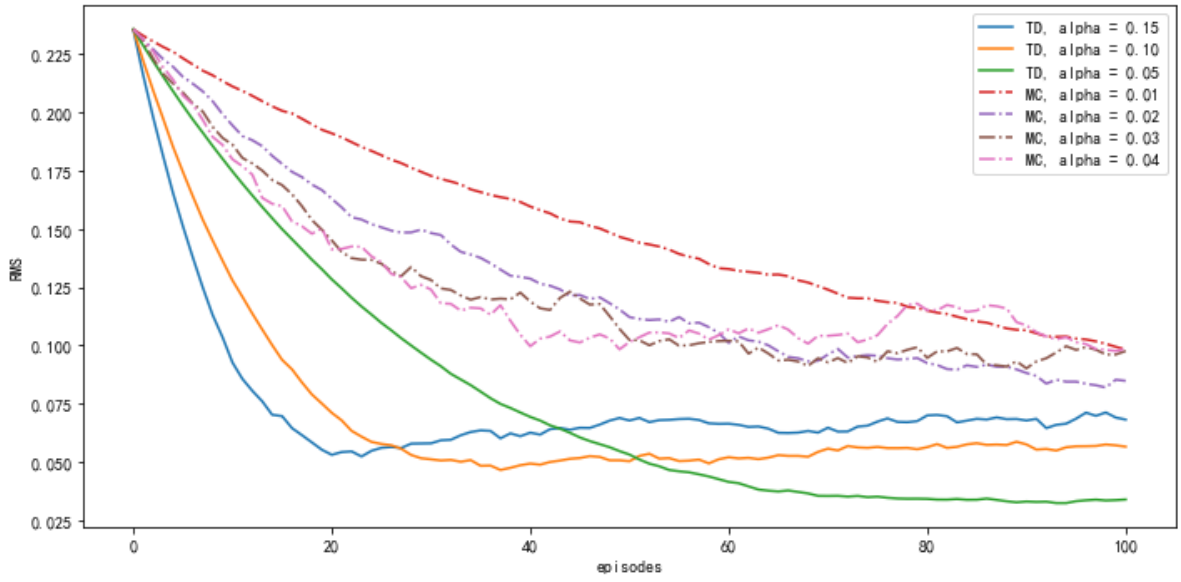
```
def rms_error():
    # 设置TD与MC的步长参数
    td_alphas = [0.15, 0.1, 0.05]
    mc_alphas = [0.01, 0.02, 0.03, 0.04]
    # 设定总episode数量
    episodes = 100 + 1
    runs = 100
    # 遍历每个alpha设置
```

```
    for i, alpha in enumerate(td_alphas + mc_alphas):
        total_errors = np.zeros(episodes)
        if i < len(td_alphas):
            method = 'TD'
            linestyle = 'solid'
        else:
            method = 'MC'
            linestyle = 'dashdot'
        # 这里整个过程一共运行100次，每次都是100幕，最后会对结果进行平均。
        for r in tqdm(range(runs)):
            errors = []
            current_values = np.copy(VALUES)
            for i in range(0, episodes):
                # 计算当次幕下当前估计值和真实值之间的均方根误差。
                errors.append(np.sqrt(np.sum(np.power(TRUE_VALUE -
current_values, 2)) / 5.0))
                if method == 'TD':
                    temporal_difference(current_values, alpha=alpha)
                else:
                    monte_carlo(current_values, alpha=alpha)
            total_errors += np.asarray(errors)
        total_errors /= runs
        plt.plot(total_errors, linestyle=linestyle, label=method + ', alpha
= %.02f' % (alpha))
    plt.xlabel('episodes')
    plt.ylabel('RMS')
    plt.legend()
```

运行如下代码：

```
plt.figure(figsize=(10,5))
rms_error()
plt.tight_layout()
```

运行的最终结果如下：



我们可以看到对于不同的α取值，两种方法的学习曲线。图中显示的性能衡量指标是学到的价值函数和真实价值函数的均方根（RMS）误差。图中显示的误差是在5个状态上的平均误差，并在100次运行中取平均的结果。在所有情况下，对于所有s，近似价值函数都被初始化为中间值$V(s) = 0.5$。在这个任务中，TD方法一直比MC方法要好。

7、批量更新的随机游走

在随机游走问题中，批量更新版本的$TD(0)$和常数α MC方法的过程是这样的：每经过新的一幕序列之后，之前所有幕的数据就放视为一个批次。算法$TD(0)$常数α MC方法不断地使用这些批次进行逐次更新。这里α要设置得足够小以使价值函数能够收敛。最后将所得的价值函数与v_π进行比较，绘制5个状态下的平均均方根误差（以整个实验的100次的独立重复为基础）的学习曲线。

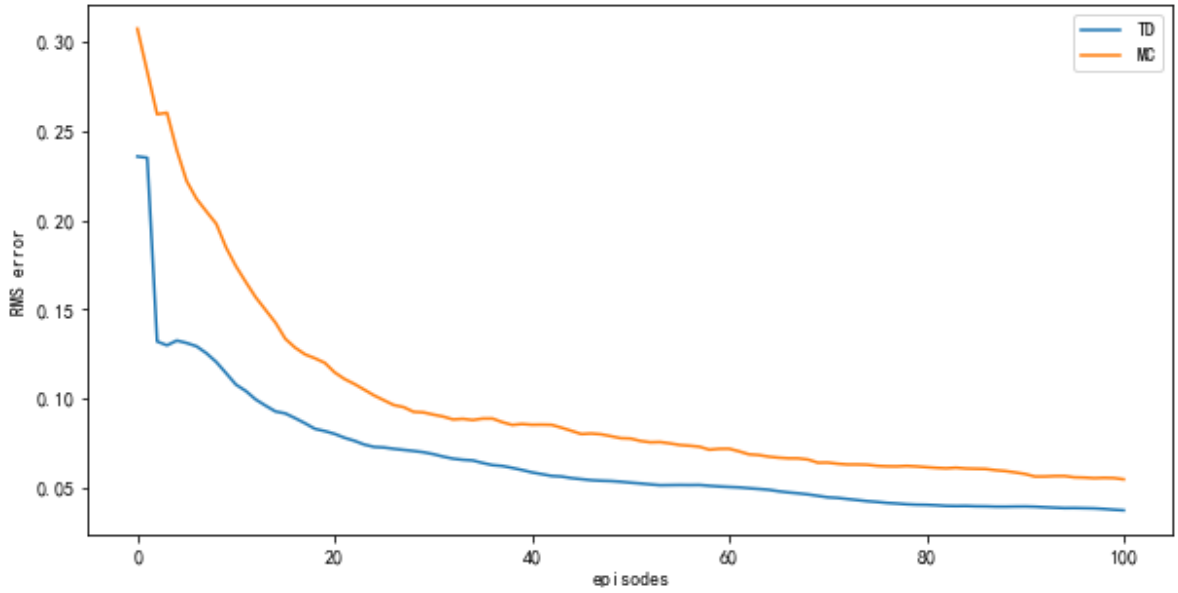
```
def batch_updating(method, episodes, alpha=0.001):
    # 整个实验进行100次独立重复运行
    runs = 100
    total_errors = np.zeros(episodes)
    for r in tqdm(range(0, runs)):
        current_values = np.copy(VALUES)
        errors = []
        # trajectories需要记录所有episode以及奖励
        trajectories = []
        rewards = []
        for ep in range(episodes):
            # 执行TD(0)
            if method == 'TD':
                trajectory_, rewards_ = temporal_difference(current_values, batch=True)
            # 执行MC
            else:
                trajectory_, rewards_ = monte_carlo(current_values, batch=True)
            trajectories.append(trajectory_)
            rewards.append(rewards_)
            while True:
                # 持续不断得将到目前为止所有的trajectories都用于训练。
                updates = np.zeros(7)
                for trajectory_, rewards_ in zip(trajectories, rewards):
                    for i in range(0, len(trajectory_) - 1):
                        if method == 'TD':
                            updates[trajectory_[i]] += rewards_[i] + current_values[trajectory_[i + 1]] - current_values[trajectory_[i]]
                        else:
                            updates[trajectory_[i]] += rewards_[i] - current_values[trajectory_[i]]
                updates *= alpha
                # 当接近收敛时才停止
                if np.sum(np.abs(updates)) < 1e-3:
                    break
                # 进行批量更新
                current_values += updates
            # 计算rms
            errors.append(np.sqrt(np.sum(np.power(current_values - TRUE_VALUE, 2)) / 5.0))
        total_errors += np.asarray(errors)
    total_errors /= runs
    return total_errors
```

执行以下代码检查结果：

```
episodes = 100 + 1
# 运行TD（0）的批量更新
td_erros = batch_updating('TD', episodes)
# 运行MC的批量更新
mc_erros = batch_updating('MC', episodes)

# 画图
plt.plot(td_erros, label='TD')
plt.plot(mc_erros, label='MC')
plt.xlabel('episodes')
plt.ylabel('RMS error')
plt.legend()
plt.show()
```

测试结果如下：



测试的实验结果可以看出，批量TD的rms始终是低于MC方法的，批量TD方法始终优于批量蒙特卡洛方法。其原因在蒙特卡洛方法只是从某些有限的方面来说是最优的，而TD方法的最优性则与预测回报这个任务更为相关。