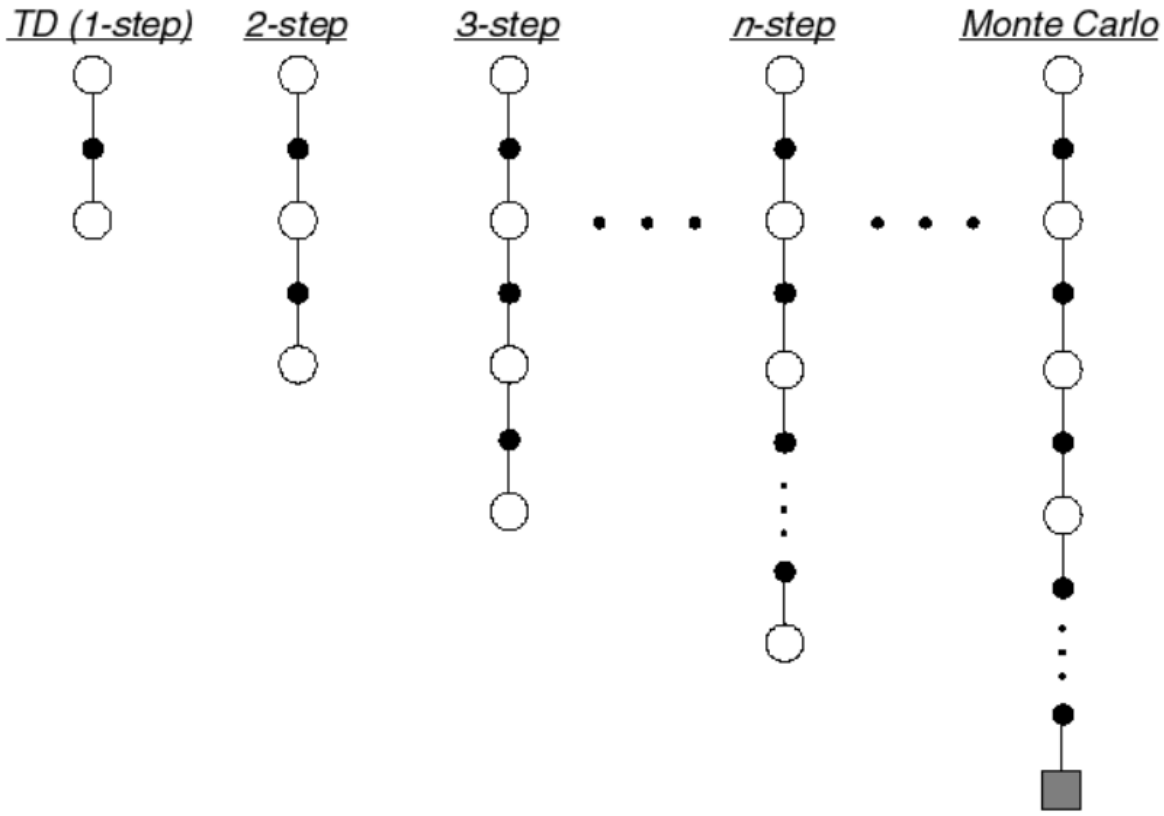
强化学习基础篇（二十五）n步时序差分预测

1、n步时序差分方法

之前在《强化学习基础篇（十七）时间差分预测》所介绍的是$TD(0)$算法，其更新过程仅仅依赖于当前状态向下走一步的情况，将走一步走后的状态价值用于bootstrap更新。而蒙特卡洛方法是根据当前状态开始到终止状态的整个收益序列进行状态价值的更新。这节介绍的n步时序差分(n-step TD)是基于$TD(0)$的一步更新与MC对整个序列进行更新的两个极端之间的算法。从n步时序差分方法的回溯图中，我们可以看到每个n步方法都考虑了从当前状态向下走n步的情况。



2、n步回报

如果我们考虑如下的n取值下的回报（$n=1,2,\ldots,\infty$）

$$\begin{aligned} n=1 \quad (TD) \quad & G_t^{(1)} = R_{t+1} + \gamma V\left(S_{t+1}\right) \\ n=2 \quad (TD) \quad & G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V\left(S_{t+2}\right) \\ \vdots \quad & \vdots \\ n=\infty \quad (MC) \quad & G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T \end{aligned}$$

那么我们可以进行泛化定义n步回报为：

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V\left(S_{t+n}\right)$$

根据n步回报修改$TD(0$的更新方法为：

$$V\left(S_t\right) \leftarrow V\left(S_t\right) + \alpha\left(G_t^{(n)} - V\left(S_t\right)\right)$$

这样我们就可以得到如下的n步时序差分算法。

n 步TD(0)估计 $V \approx v_\pi$

输入：策略 π

算法参数：步长 $\alpha \in (0,1]$，正整数 n

对 $s \in \mathcal{S}$，任意初始化 $V(s)$

所有存储和访问操作（对于 S_t 和 R_t）都可以使其索引 $mod n+1$

对每个回合循环：

初始化并存储 $S_0 \neq$ 终点

$T \leftarrow \infty$

对 $t = 0, 1, 2, \ldots$ 循环：

如果 $t < T$，则：

根据 $\pi(\cdot|S_t)$ 采取行动

观察并将下一个奖励存储为 R_{t+1}，将下一个状态存储为 S_{t+1}

如果 S_{t+1} 是终点，则 $T \leftarrow t+1$

$\tau \leftarrow t-n+1$ （τ 是状态估计正在更新的时间）

如果 $\tau \geq 0$:

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$

如果 $\tau+n < T$，则 $G \leftarrow G + \gamma^n V(S_{\tau+n})$

$V(S_\tau) \leftarrow V(S_\tau) + \alpha[G - V(S_\tau)]$ $\qquad (G_{\tau:\tau+n})$

直到 $\tau = T-1$

3、n步时序差分方法在随机游走上的应用

在《强化学习基础篇（十九）TD与MC在随机游走问题应用》我们实现了随机游走的问题。这里我们将原问题的6个状态调整为19个状态，下面看看通过n步回报的方法效果如何。

导入库函数定义超参数

```
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from tqdm import tqdm
```

```python

# 共19个状态
N_STATES = 19

# 定义折扣因子
GAMMA = 1

# 定义状态空间
STATES = np.arange(1, N_STATES + 1)

# 起始状态为第10个状态
START_STATE = 10

# 一共两个terminal状态
# 左边界的状态的回报为-1，右边界的状态的回报为+1
END_STATES = [0, N_STATES + 1]

# 设定真实价值（true value）
TRUE_VALUE = np.arange(-20, 22, 2) / 20.0
TRUE_VALUE[0] = TRUE_VALUE[-1] = 0
```

n-steps TD算法实现

```python
# n-steps TD 算法实现
# value: 输入状态价值函数
# n: 输入n步的值
# alpha: 定义步长
def temporal_difference(value, n, alpha):
    # 初始化状态位置
    state = START_STATE

    # 定义一个列表存储states和rewards
    states = [state]
    rewards = [0]

    # 进行时间跟踪
    time = 0

    # 是定时间初始为无限
    T = float('inf')
    while True:
        # 进一个时间步
        time += 1

        if time < T:
            # 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) == 1:
                next_state = state + 1
            else:
                next_state = state - 1
           # 按照问题定义，处理计算奖励。
            if next_state == 0:
                reward = -1
            elif next_state == 20:
                reward = 1
            else:
                reward = 0
```

```
            # 存储下一步状态与奖励
            states.append(next_state)
            rewards.append(reward)

            if next_state in END_STATES:
                T = time

        # get the time of the state to update
        update_time = time - n
        if update_time >= 0:
            returns = 0.0
            # 计算n步奖励
            for t in range(update_time + 1, min(T, update_time + n) + 1):
                returns += pow(GAMMA, t - update_time - 1) * rewards[t]
            # 将n步奖励增加到总回报中
            if update_time + n <= T:
                returns += pow(GAMMA, n) * value[states[(update_time + n)]]
            state_to_update = states[update_time]
            # 更新状态值函数
            if not state_to_update in END_STATES:
                value[state_to_update] += alpha * (returns -
value[state_to_update])
        if update_time == T - 1:
            break
        state = next_state
```

实验运行与绘制结果

```
def figure7_2():
    # 这里要比较的n步包含了1,2,4,8..512
    steps = np.power(2, np.arange(0, 10))

    # 这里比较了三个步长
    alphas = np.arange(0, 1.1, 0.1)

    # 每次运行10个episodes
    episodes = 10

    # 实验总次数（因为结果要对这些100次取平均）
    runs = 100

    # track the errors for each (step, alpha) combination
    errors = np.zeros((len(steps), len(alphas)))
    for run in tqdm(range(0, runs)):
        for step_ind, step in enumerate(steps):
            for alpha_ind, alpha in enumerate(alphas):
                # print('run:', run, 'step:', step, 'alpha:', alpha)
                value = np.zeros(N_STATES + 2)
                for ep in range(0, episodes):
                    temporal_difference(value, step, alpha)
                    # 计算均方根误差（RMS error）
                    errors[step_ind, alpha_ind] +=
np.sqrt(np.sum(np.power(value - TRUE_VALUE, 2)) / N_STATES)
    # 对结果取平均
    errors /= episodes * runs

```

```
    for i in range(0, len(steps)):
        plt.plot(alphas, errors[i, :], label='n = %d' % (steps[i]))
    plt.xlabel('alpha')
    plt.ylabel('RMS error')
    plt.ylim([0.25, 0.55])
    plt.legend()

    plt.savefig('./figure_7_2.png')
    plt.close()

if __name__ == '__main__':
    figure7_2()
```

测试结果

结果展示了在不同的n与α情况下n步方法的性能。不同情况下的性能测试指标是最后19个状态在每个episode终止时的价值函数的估计值和真实值的均方误差的平均值的开方，图中展示的是最开始10个episode，并重复100次的平均结果。从图中可以看出，n取中间大小的值效果最好，这也证明了将单步时序差分方法和蒙特卡洛方法推广到n步时序差分方法可能得到更好的结果。