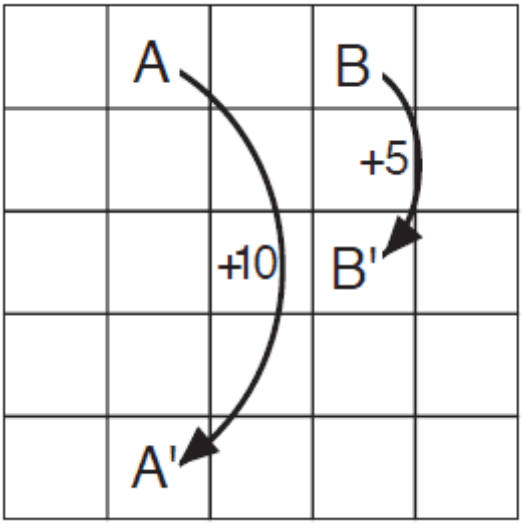
强化学习基础篇（二十一）网格问题

该问题基于《Reinforcement Learning: An Introduction》在第三章中给出了一个简单的例子:`Gridworld`,以帮助我们理解有限MDPs。

1、网格问题描述

如下图所示的长方形网格代表的是一个简单的有限MDP。网格中的格子代表的是环境中的状态。在每个格子中，有四个可选的动作：东、南、西和北，每个动作都会使智能体在对应的方向上前进一格。如果采取的动作使得其不在网格内，则智能体会在原位置不移动，并且得到一个值为-1的收益。除了将智能体从特定的状态A和B中移走的动作外，其他动作的收益都是0。在状态A中，四种动作都会得到一个+10的收益，并且把智能体移动到A'。在状态B中，四种动作都会得到一个+5的收益，并且把智能体移动到B‘。



2、解析

这是一个典型的MDP问题，有限的状态空间，有限的动作空间。但是，我们面临两个棘手的问题：

- 根据Bellman等式，当前状态的价值依赖于下一个状态的价值，这是一个递归的过程。从何处开始，在何处结束？对于每一个状态都有不同的episode。
- 状态空间和动作空间如何表达？状态空间的表达容易想到使用格子的坐标来表示，比如格子A的状态可表示为坐标0,10,1，那么动作空间如何表达比较合适呢？

2.1、Bellman等式的迭代求解

根据bellman求解MDP问题一般有两种思路：第一，如果MDP的状态转移矩阵是已知的，则可以精确的数值求解：求解线性方程组而已。。第二，Bellman等式本身是一个迭代公式，因此可以通过迭代的方式逐步逼近状态价值函数。这种方式对状态转移矩阵不敏感，即无论是否知道状态转移矩阵都可以使用，也非常适合于编程实现。

迭代求解状态价值函数的基本步骤为：

1. 初始化状态价值矩阵，一般初始化为0。这是计算的起点，此时每个状态的价值函数都是0。
2. 根据Bellman等式，计算每个状态的新的价值函数。注意这一步**不是一个递归的过程**，因为上一轮的状态价值矩阵是已知的，在这一轮直接可以根据上一轮的状态价值矩阵计算新的价值函数即可，即：

$$v_{k+1}(s)=\sum_{a}\pi(a\mid s)\sum_{r,s'}p(s',r\mid s,a)\left[r+\gamma v_k(s')\right]$$

其中的k代表了迭代的轮次（episode），注意不要和每个轮次（episode）中的时刻（步骤）的tt混淆了。

计算每个轮次迭代的误差，达到设定的误差范围即可停止迭代，此时即获得了最终的状态价值函数。

这种迭代求解状态价值函数的方法称为"Policy Evaluation Prediction"，即对于任意给定的策略π，都可以通过迭代的方式逐步逼近求解状态价值函数。但是，必须确保这种迭代求解的方法对于任意策略π都是收敛的，

证明如下：

通过观察上式，并对照Bellman等式

$$v_{\pi}(s)=\sum_{a}\pi(a\mid s)\sum_{r,s'}p(s',r\mid s,a)\left[r+\gamma v_{\pi}(s')\right]$$

可以看出，**式1只是对式2的简单变量替换**，由v_{π}替换为v_k，而式2在$\gamma<1$或者episode能够在有限步内结束时即收敛，因此式1在同样的条件下也会收敛，即迭代求解状态价值函数的方法的收敛条件为：或者$\gamma<1$，或者所解决的MDP问题的episode是有限的。这两个条件几乎总是能够满足其中之一的。

2.2、动作空间的表示方式

动作对状态的影响表现为状态坐标的改变，因此将动作定义为对状态坐标的偏移量是个合理的方案，这样可以直接将状态和动作相加即可获得"下一个状态"。以动作up为例，其对状态的影响为横坐标不变，纵坐标增加+1，因此可以将up定义为[0,1]。同理，动作down可以定义为[0,-1]。

3. 实现过程

3.1、导入库文件

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.table import Table
```

3.2、定义环境

```
WORLD_SIZE = 5 # 定义问题中格子的数量
# 定义问题中A,A',B,B'的位置
A_POS = [0, 1]
A_PRIME_POS = [4, 1]
B_POS = [0, 3]
B_PRIME_POS = [2, 3]
# 定义折扣参数
DISCOUNT = 0.9

# 把动作定义为对x，y坐标的增减改变
ACTIONS = [np.array([0, -1]), # 向上
           np.array([-1, 0]),  # 向左
           np.array([0, 1]),  # 向下
           np.array([1, 0])]  # 向右
# 定义画图会用到的动作
ACTIONS_FIGS=[ '←', '↑', '→', '↓']
# 该问题中每个动作选择的概率为0.25
```

```
ACTION_PROB = 0.25
```

3.3、 定义环境转移过程

```
def step(state, action):
    """每次走一步
    state:当前状态，坐标的list，比如[1,1]
    action:当前采取的动作，是对状态坐标的修正
    函数返回值，下一个状态（坐标的list）和reward
    """
    # 当在A位置的时候，会转移到A',并得到奖励+10.
    if state == A_POS:
        return A_PRIME_POS, 10
    # 当在B位置的时候，会转移到B',并得到奖励+5.
    if state == B_POS:
        return B_PRIME_POS, 5

    # 计算下一个状态
    next_state = (np.array(state) + action).tolist()
    x, y = next_state
    # 当运动未知超出格子世界则在原位置不变，并且得到-1的收益，其他情况得到的收益都是0。
    if x < 0 or x >= WORLD_SIZE or y < 0 or y >= WORLD_SIZE:
        reward = -1.0
        next_state = state
    else:
        reward = 0
    return next_state, reward
```

3.4 、绘制价值函数结果

```
def draw_image(image):
    fig, ax = plt.subplots()
    ax.set_axis_off()
    tb = Table(ax, bbox=[0, 0, 1, 1])

    nrows, ncols = image.shape
    width, height = 1.0 / ncols, 1.0 / nrows

    # Add cells
    for (i, j), val in np.ndenumerate(image):

        # add state labels
        if [i, j] == A_POS:
            val = str(val) + " (A)"
        if [i, j] == A_PRIME_POS:
            val = str(val) + " (A')"
        if [i, j] == B_POS:
            val = str(val) + " (B)"
        if [i, j] == B_PRIME_POS:
            val = str(val) + " (B')"

        tb.add_cell(i, j, width, height, text=val,
                    loc='center', facecolor='white')

    # Row and column labels...
    for i in range(len(image)):
```

```
        tb.add_cell(i, -1, width, height, text=i+1, loc='right',
                    edgecolor='none', facecolor='none')
        tb.add_cell(-1, i, width, height/2, text=i+1, loc='center',
                    edgecolor='none', facecolor='none')

    ax.add_table(tb)
```

3.5 、绘制策略图形

```
def draw_policy(optimal_values):
    fig, ax = plt.subplots()
    ax.set_axis_off()
    tb = Table(ax, bbox=[0, 0, 1, 1])

    nrows, ncols = optimal_values.shape
    width, height = 1.0 / ncols, 1.0 / nrows

    # Add cells
    for (i, j), val in np.ndenumerate(optimal_values):
        next_vals=[]
        for action in ACTIONS:
            next_state, _ = step([i, j], action)
            next_vals.append(optimal_values[next_state[0],next_state[1]])

        best_actions=np.where(next_vals == np.max(next_vals))[0]
        val=''
        for ba in best_actions:
            val+=ACTIONS_FIGS[ba]

        # add state labels
        if [i, j] == A_POS:
            val = str(val) + " (A)"
        if [i, j] == A_PRIME_POS:
            val = str(val) + " (A')"
        if [i, j] == B_POS:
            val = str(val) + " (B)"
        if [i, j] == B_PRIME_POS:
            val = str(val) + " (B')"

        tb.add_cell(i, j, width, height, text=val,
                loc='center', facecolor='white')

    # Row and column labels...
    for i in range(len(optimal_values)):
        tb.add_cell(i, -1, width, height, text=i+1, loc='right',
                    edgecolor='none', facecolor='none')
        tb.add_cell(-1, i, width, height/2, text=i+1, loc='center',
                   edgecolor='none', facecolor='none')

    ax.add_table(tb)
```

3.6、计算等概率随机策略下的状态价值函数(解贝尔曼期望方程)

贝尔曼期望方程的形式为：

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{r,s'} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right]$$

```
def figure_3_2():
    """
    计算每个单元格的状态价值函数
    """
    # 状态价值函数的初值设置为0
    value = np.zeros((WORLD_SIZE, WORLD_SIZE))
    while True:
        # # 每一轮迭代都会产生一个new_value，直到new_value和value很接近即收敛为止
        new_value = np.zeros_like(value)

        # 对格子世界的每个状态进行遍历
        for i in range(WORLD_SIZE):
            for j in range(WORLD_SIZE):
                # 对每个状态的4个动作进行遍历
                for action in ACTIONS:
                    # 计算当前动作执行后的下一个状态，以及奖励
                    (next_i, next_j), reward = step([i, j], action)
                    # 由于每个方向只有一个reward和s'的组合，这里的p(s',r|s,a)=1
                    new_value[i, j] += ACTION_PROB * (reward + DISCOUNT *
value[next_i, next_j])
        # 当前new_value和value很接近，即error小到一定的程度则停止。
        error = np.sum(np.abs(new_value - value))
        if error < 1e-4:
            draw_image(np.round(new_value, decimals=2))
            plt.show()
            break
        value = new_value
figure_3_2()
```

输出的测试结果如下所示：

	1	2	3	4	5
1	3.31	8.79 (A)	4.43	5.32 (B)	1.49
2	1.52	2.99	2.25	1.91	0.55
3	0.05	0.74	0.67	0.36 (B')	-0.4
4	-0.97	-0.44	-0.35	-0.59	-1.18
5	-1.86	-1.35 (A')	-1.23	-1.42	-1.98

可以看出，靠近下边的格子，尤其是靠近边界的格子的状态价值偏小，这是因为靠近边界的格子很容易出界，而出界的奖励是-1；格子A的价值最高，这是很容易理解的，因为从A到A'的奖励是+10。但是为什么格子A的价值<10呢？这是因为从A到A'的奖励尽管是+10，但是A'的价值却是负值，导致A的价值会跟着损失一些。格子B的价值分析方法类似。

3.7、直接求解贝尔曼方程

下面采用代码直接去求解贝尔曼方程。

```
def figure_3_2_linear_system():
    '''
    该函数通过直接求解线性方程组，得到贝尔曼方程的精确解。
    Here we solve the linear system of equations to find the exact solution.
    We do this by filling the coefficients for each of the states with their respective right side constant.
    '''
    # 定义A矩阵，25*25大小，对应着25个线性方程组
    A = -1 * np.eye(WORLD_SIZE * WORLD_SIZE)
    b = np.zeros(WORLD_SIZE * WORLD_SIZE)
    for i in range(WORLD_SIZE):
        for j in range(WORLD_SIZE):
            # 当前状态
            s = [i, j]
            # 该函数找到状态s的序号，即将格子展开为5*5=25后，在这个长序列中的编号。
            index_s = np.ravel_multi_index(s, (WORLD_SIZE, WORLD_SIZE))
            for a in ACTIONS:
                # 获取执行动作后的下一个状态和奖励
                s_, r = step(s, a)
                # 找到下一个状态的序号
                index_s_ = np.ravel_multi_index(s_, (WORLD_SIZE, WORLD_SIZE))
                # 线性方程组Ax=b的A中对应元素按照动作概率*折扣因子进行更新
                A[index_s, index_s_] += ACTION_PROB * DISCOUNT
                b[index_s] -= ACTION_PROB * r
    # 求解Ax=b
    x = np.linalg.solve(A, b)
    draw_image(np.round(x.reshape(WORLD_SIZE, WORLD_SIZE), decimals=2))
    plt.show()
```

输出过和通过迭代方式求解出来的结果是一致的。

	1	2	3	4	5
1	3.31	8.79 (A)	4.43	5.32 (B)	1.49
2	1.52	2.99	2.25	1.91	0.55
3	0.05	0.74	0.67	0.36 (B')	-0.4
4	-0.97	-0.44	-0.35	-0.59	-1.18
5	-1.86	-1.35 (A')	-1.23	-1.42	-1.98

3.8、求解最优价值函数（解贝尔曼最优方程）

贝尔曼最优方程的形式为：

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r \mid s, a)[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')]
\end{aligned}
$$

```python
def figure_3_5():
    '''
    求解贝尔曼最优方程
    '''
    # 初始值设为0
    value = np.zeros((WORLD_SIZE, WORLD_SIZE))
    while True:
        # keep iteration until convergence
        new_value = np.zeros_like(value)
        # 遍历所有状态
        for i in range(WORLD_SIZE):
            for j in range(WORLD_SIZE):
                values = []
                # 遍历所有动作
                for action in ACTIONS:
                    # 执行动作,转移到后继状态,并获得立即奖励
                    (next_i, next_j), reward = step([i, j], action)
                    # value iteration
                   # 缓存动作值函数 q(s,a) = r + γv(s')
                    values.append(reward + DISCOUNT * value[next_i, next_j])
                # 根据贝尔曼最优方程,找出最大的动作值函数 q(s,a) 进行更新
                new_value[i, j] = np.max(values)
        # 迭代终止条件: 误差小于1e-4
        if np.sum(np.abs(new_value - value)) < 1e-4:
            draw_image(np.round(new_value, decimals=2))
            plt.show()
            plt.close()
            draw_policy(new_value)
            plt.show()
            break
        value = new_value
figure_3_5()
```

计算得到的最优价值函数如下所示：

	1	2	3	4	5
1	21.98	24.42 (A)	21.98	19.42 (B)	17.48
2	19.78	21.98	19.78	17.8	16.02
3	17.8	19.78	17.8	16.02 (B')	14.42
4	16.02	17.8	16.02	14.42	12.98
5	14.42	16.02 (A')	14.42	12.98	11.68

从最后价值函数得到的最优策略如下，图中有很多个箭头，其对应的每个动作都是最优的。

	1	2	3	4	5
1	→	←↑→↓ (A)	←	←↑→↓ (B)	←
2	↑→	↑	←↑	←	←
3	↑→	↑	←↑	←↑ (B')	←↑
4	↑→	↑	←↑	←↑	←↑
5	↑→	↑ (A')	←↑	←↑	←↑