强化学习基础篇（二十三）策略迭代之租车问题

该问题基于《Reinforcement Learning: An Introduction》在第四章的例4.2 杰克租车问题。

1、问题描述

Jack管理着一家有两个场地的小型的租车公司（分别称为first location和second location，每租出一部车，Jack可赚10￥。为了提高车子的出租率，Jack在夜间调配车辆，即把车子从一个场地调配到另外一个场地，成本是2￥每辆。假设每个场地**每天**租车和还车的数量是泊松随机变量，即其数值是n的概率为$\frac{\lambda^n}{n!}e^{-\lambda}$，其中λλ为期望。假设场地1和场地2租车的的λ分别为3和4，还车的λ分别为3和2。为了简化问题起见，我们假设每个场地最多可停20部车（如果归还的车辆超出了20部，我们假设超出的车辆无偿调配到了别的地方，比如总公司），并且每个场地每天最多调配5部车子。

请问Jack在每个场地应该部署多少部车子？每天晚上如何调配？

2、问题分析

2.1、已知条件：

状态空间：1号租车点和2号租车点，每个地点最多20辆车供租赁

行为空间：每天下班后最多转移5辆车从一个租车点到另一个租车点

即时奖励：Jack每租出去一辆车可以获利10美金，但必须是有车可租的情况，不考虑在两地转移车辆的支出

转移概率：租出去的车辆数量（n）和归还的车辆数量（n）是随机的，但是服从泊松分布$\frac{\lambda^n}{n!}e^{-\lambda}$。

- 对于1号租车点：向外出租服从$\lambda=3$的泊松分布，回收也服从$\lambda=3$的泊松分布
- 对于2号租车点：向外出租服从$\lambda=4$的泊松分布，回收服从$\lambda=2$的泊松分布

折扣因子γ：$\gamma=0.9$

2.2、问题分析：

- 每个租车点最多20辆车，那么状态数量就是$21*21=441$个。
- 最多调配5辆车，即动作集合为:

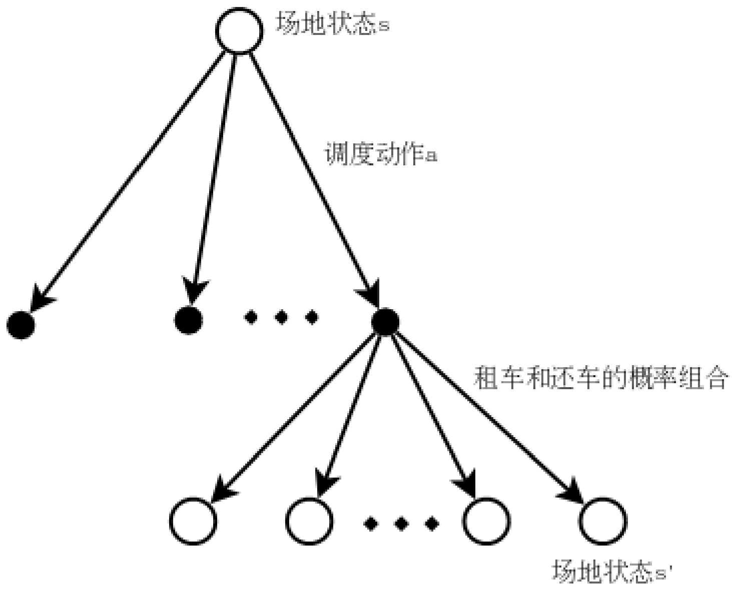
  $A=\{(-5,5),(-4,4),(-3,3),(-2,2),(-1,1),(0,0),(1,-1),(2,-2),(3,-3),(4,-4),(5,-5)\}$

  其中么个动作元素表示为（1号租车点出入车辆，2号租车点出入车辆），正负号分别表示“入”和“出”。

进一步分析，虽然租车、还车和调配都会改变状态进而影响最终的收益，但是租车量和还车量是一个我们无法控制的量！只有调配是我们可以控制和优化的量。

根据问题的假设，调配的上限是5部车子。每调配一部车子，都会对状态空间（两个场地的车子数量）产生影响，进而影响租车的周转率和收益，因此将调配作为一个调优的指标是合理的。那么如何设计调配这个指标呢?

调配是指从一个场地运送车辆到另外一个场地。根据假设，从场地1调配到场地2的车辆数量是[5,-5]，其中-5表示从场地2调配5部车到场地1，因此共有11个调配的动作，这就是动作空间。问题转化为对于任意的状态，这11个调配动作如何优化最合理？如下图所示：



2.3、求解过程：

“1号租车点有10辆车”收益分析：

考虑状态“1号租车点有10辆车”的未来可能获得收益，需要分析在保有10辆车的情况下的租车（Rent）与回收（Return）的行为。计算该状态收益的过程实际上是，另外一个动作策略符合泊松分布的马尔可夫决策过程。

将1天内可能发生的Rent与Return行为记录为[#Rent #Return]，其中“#Rent”表示一天内租出的车辆数，“#Return”表示一天内回收的车辆数，设定这两个指标皆不能超过20。

假设早上，1号租车点里有10辆车，那么在傍晚清点的时候，可能保有的车辆数为0~20辆。如果傍晚关门歇业时还剩0辆车，那么这一天的租收行为$A_{rent,return}$可以是：

$$A_{rent,return} = \begin{bmatrix} 10 & 0 \\ 11 & 1 \\ 12 & 2 \\ \dots & \dots \\ 20 & 10 \end{bmatrix}$$

Rent与Return是相互独立的事件且皆服从泊松分布，所以要计算某个行为出现的概率直接将$P(A_{rent})$与$P(A_{return})$相乘。

但这里要计算的是条件概率，即为$P(A_{rent,return}|S''=0)$，所以还需要再与傍晚清点时还剩0辆车的概率$P(S''=0)$相除。

各个租收行为所获得的收益是以租出去的车辆数为准，所以当傍晚还剩0辆车时，这一天的收益期望可以写为：

$$R(S'=10|S''=0) = 10 \begin{bmatrix} \frac{P(A_{rent}=10)P(A_{return}=0)}{P(S''=0)} \\ \frac{P(A_{rent}=11)P(A_{return}=1)}{P(S''=0)} \\ \dots \\ \frac{P(A_{rent}=20)P(A_{return}=10)}{P(S''=0)} \end{bmatrix}^T \begin{bmatrix} 10 \\ 11 \\ \dots \\ 20 \end{bmatrix}$$

其中，$P(S''=0)$也可以写为：

$$P(S''=0) = \sum P(A_{rent})P(A_{return})$$

在计算出矩阵$R(S'=10|S''=0,1,2,...20)$后，再进行加权平均，即可得到状态“1号租车点有10辆车”的奖励期望$R(S'=10)$:

$$R(S'=10)=P(S''=0,1,2,...20)R^T(S'=10|S''=0,1,2,\dots,20)$$

两个租车点，所有的状态按上述方法计算后，即可得出两个租车点的奖励矩阵$|R_1(S'),R_2(S')|$。

在计算出奖励矩阵后，这个问题就变成了bandit问题的变种，bandit问题是一个动作固定对应一个未来的状态，而这里虽然也是这样，不过所对应的状态却要以当前状态为基础进行计算得出，还是有些不同，所以称为bandit问题的一个变种。

2、程序实现

以下程序遵循的策略迭代算法如下：

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   - $\Delta \leftarrow 0$
   - Loop for each $s \in \mathcal{S}$:
     - $v \leftarrow V(s)$
     - $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r+\gamma V(s')\big]$
     - $\Delta \leftarrow \max(\Delta, |v-V(s)|)$

   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   - *old-action* $\leftarrow \pi(s)$
   - $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r+\gamma V(s')\big]$
   - If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*

   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

导入库函数

```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from scipy.stats import poisson
```

设置超参数

```python
# 每个场的车容量
MAX_CARS = 20

# 每晚最多移动的车数
MAX_MOVE_OF_CARS = 5

```

```
# A场租车请求的平均值
RENTAL_REQUEST_FIRST_LOC = 3
# B场租车请求的平均值
RENTAL_REQUEST_SECOND_LOC = 4

# A场还车请求的平均值
RETURNS_FIRST_LOC = 3
# B场还车请求的平均值
RETURNS_SECOND_LOC = 2

# 收益折扣
DISCOUNT = 0.9
# 租车收益
RENTAL_CREDIT = 10
# 移车支出
MOVE_CAR_COST = 2

# （移动车辆）动作空间：【-5，5】
actions = np.arange(-MAX_MOVE_OF_CARS, MAX_MOVE_OF_CARS + 1)

# 租车还车的数量满足一个poisson分布，限制由泊松分布产生的请求数大于POISSON_UPPER_BOUND时其概率压缩至0
POISSON_UPPER_BOUND = 11
# 存储每个（n,lamda）对应的泊松概率
poisson_cache = dict()
```

定义泊松分布

泊松分布的概率式为：

$$P(X=n)=\frac{\lambda^n}{n!}e^{-\lambda}$$

泊松分布其可由**scipy.stats**库中的**poisson**模块产生，为了避免重复调用，我们使用一个字典poisson_cache来记录每个状态下的概率值：

```
def poisson_probability(n, lam):
    global poisson_cache
    key = n * 10 + lam # 定义唯一key值，除了索引没有实际价值
    if key not in poisson_cache:
        # 计算泊松概率，这里输入为n与lambda，输出泊松分布的概率质量函数，并保存到poisson_cache中
        poisson_cache[key] = poisson.pmf(n, lam)
    return poisson_cache[key]
```

计算状态价值

计算状态价值的函数代码如下：

```
def expected_return(state, action, state_value, constant_returned_cars):
    """
    @state: 状态定义为每个地点的车辆数

    @action: 车辆的移动数量【-5，5】，负：2->1，正：1->2

    @stateValue: 状态价值矩阵

```

```python

    @constant_returned_cars:  将还车的数目设定为泊松均值，替换泊松概率分布
    """

    # initailize total return
    returns = 0.0

    # 移动车辆产生负收益
    returns -= MOVE_CAR_COST * abs(action)

    # 移动后的车辆总数不能超过20
    NUM_OF_CARS_FIRST_LOC = min(state[0] - action, MAX_CARS)
    NUM_OF_CARS_SECOND_LOC = min(state[1] + action, MAX_CARS)

    # 遍历两地全部的可能概率下（<11）租车请求数目
    for rental_request_first_loc in range(POISSON_UPPER_BOUND):
        for rental_request_second_loc in range(POISSON_UPPER_BOUND):
            # prob为两地租车请求的联合概率,概率为泊松分布
            # 即：1地请求租车rental_request_first_loc量且2地请求租车rental_request_second_loc量
            prob = poisson_probability(rental_request_first_loc, RENTAL_REQUEST_FIRST_LOC) * \
                poisson_probability(rental_request_second_loc, RENTAL_REQUEST_SECOND_LOC)

            # 两地原本的车的数量
            num_of_cars_first_loc = NUM_OF_CARS_FIRST_LOC
            num_of_cars_second_loc = NUM_OF_CARS_SECOND_LOC

            # 有效的租车数目必须小于等于该地原有的车辆数目
            valid_rental_first_loc = min(num_of_cars_first_loc, rental_request_first_loc)
            valid_rental_second_loc = min(num_of_cars_second_loc, rental_request_second_loc)

            # 计算回报，更新两地车辆数目变动
            reward = (valid_rental_first_loc + valid_rental_second_loc) * RENTAL_CREDIT
            num_of_cars_first_loc -= valid_rental_first_loc
            num_of_cars_second_loc -= valid_rental_second_loc

            # 如果还车数目为泊松分布的均值
            if constant_returned_cars:
                # 两地的还车数目均为泊松分布均值
                returned_cars_first_loc = RETURNS_FIRST_LOC
                returned_cars_second_loc = RETURNS_SECOND_LOC
                # 还车后总数不能超过车场容量
                num_of_cars_first_loc = min(num_of_cars_first_loc + returned_cars_first_loc, MAX_CARS)
                num_of_cars_second_loc = min(num_of_cars_second_loc + returned_cars_second_loc, MAX_CARS)
                # 核心：
                # 策略评估：V(s) = p(s',r|s,π(s))[r + γV(s')]
                returns += prob * (reward + DISCOUNT * state_value[num_of_cars_first_loc, num_of_cars_second_loc])

            # 否则计算所有泊松概率分布下的还车空间
            else:
```

```
                for returned_cars_first_loc in range(POISSON_UPPER_BOUND):
                    for returned_cars_second_loc in range(POISSON_UPPER_BOUND):
                        prob_return = poisson_probability(
                            returned_cars_first_loc, RETURNS_FIRST_LOC) * poisson_probability(returned_cars_second_loc, RETURNS_SECOND_LOC)
                        num_of_cars_first_loc_ = min(num_of_cars_first_loc + returned_cars_first_loc, MAX_CARS)
                        num_of_cars_second_loc_ = min(num_of_cars_second_loc + returned_cars_second_loc, MAX_CARS)
                        # 联合概率为【还车概率】*【租车概率】
                        prob_ = prob_return * prob
                        returns += prob_ * (reward + DISCOUNT *
 state_value[num_of_cars_first_loc_, num_of_cars_second_loc_])
    return returns
```

策略迭代算法

```
def figure_4_2(constant_returned_cars=True):
    # 初始化价值函数为0
    value = np.zeros((MAX_CARS + 1, MAX_CARS + 1))
    # 初始化策略为0
    policy = np.zeros(value.shape, dtype=np.int)
    # 设置迭代参数
    iterations = 0

    # 准备画布大小，并准备多个子图
    _, axes = plt.subplots(2, 3, figsize=(40, 20))
    # 调整子图的间距，wspace=0.1为水平间距，hspace=0.2为垂直间距
    plt.subplots_adjust(wspace=0.1, hspace=0.2)
    # 这里将子图形成一个1*6的列表
    axes = axes.flatten()
    while True:
        # 使用seaborn的heatmap作图
        # flipud为将矩阵进行垂直角度的上下翻转，第n行变为第一行，第一行变为第n行，如此。
        # cmap:matplotlib的colormap名称或颜色对象；
        # 如果没有提供，默认为cubehelix map (数据集为连续数据集时) 或 RdBu_r (数据集为离散数据集时)
        fig = sns.heatmap(np.flipud(policy), cmap="YlGnBu", ax=axes[iterations])

        # 定义标签与标题
        fig.set_ylabel('# cars at first location', fontsize=30)
        fig.set_yticks(list(reversed(range(MAX_CARS + 1))))
        fig.set_xlabel('# cars at second location', fontsize=30)
        fig.set_title('policy {}'.format(iterations), fontsize=30)

        # policy evaluation (in-place) 策略评估（in-place）
        # 未改进前，第一轮policy全为0，即[0，0，0...]
        while True:
            old_value = value.copy()
            for i in range(MAX_CARS + 1):
                for j in range(MAX_CARS + 1):
                    # 更新V（s）
                    new_state_value = expected_return([i, j], policy[i, j], value, constant_returned_cars)
```

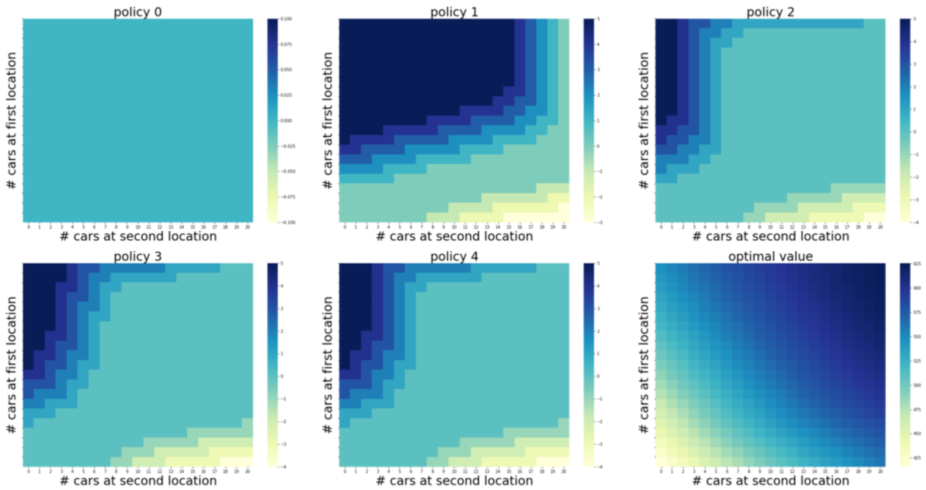
```python
                    # in-place操作
                    value[i, j] = new_state_value
            # 比较v_old(s)、v(s)，收敛后退出循环
            max_value_change = abs(old_value - value).max()
            print('max value change {}'.format(max_value_change))
            if max_value_change < 1e-4:
                break

        # policy improvement
        # 在上一部分可以看到，策略policy全都是0，如不进行策略改进，其必然不会收敛到实际最优策略。
        # 所以需要如下策略改进
        policy_stable = True
        # i、j分别为两地现有车辆总数
        for i in range(MAX_CARS + 1):
            for j in range(MAX_CARS + 1):
                old_action = policy[i, j]
                action_returns = []
                # actions为全部的动作空间，即[-5、-4...4、5]
                for action in actions:
                    if (0 <= action <= i) or (-j <= action <= 0):
                        action_returns.append(expected_return([i, j], action, value, constant_returned_cars))
                    else:
                        action_returns.append(-np.inf)
                # 找出产生最大动作价值的动作
                new_action = actions[np.argmax(action_returns)]
                # 更新策略
                policy[i, j] = new_action
                if policy_stable and old_action != new_action:
                    policy_stable = False
        print('policy stable {}'.format(policy_stable))

        if policy_stable:
            fig = sns.heatmap(np.flipud(value), cmap="YlGnBu", ax=axes[-1])
            fig.set_ylabel('# cars at first location', fontsize=30)
            fig.set_yticks(list(reversed(range(MAX_CARS + 1))))
            fig.set_xlabel('# cars at second location', fontsize=30)
            fig.set_title('optimal value', fontsize=30)
            break

        iterations += 1

    plt.savefig('./figure_4_2.png')
    plt.close()


if __name__ == '__main__':
    figure_4_2()
```

3、测试结果

测试过程中在有限次的迭代后，程序给出了最优的策略，如下图所示，使用颜色表示了调度车辆的数量（实际上policy3和policy4只有细微的差别）。在左上角，颜色越深表示调度车辆的数量越大，最深的颜色显然是5辆，依次递减到0辆。在右下角则相反，颜色越浅表示调度的数量越大，依次递增到5辆。

可以看出，当两个场地的车辆数量落在中间的区域时，需要调度的车辆数量为0，显然这是一个动态的过程，需要两个场地的协调和配合，也就是说需要在两个场地之间平衡车辆的数量。

在策略4中，我们可以看到场地1的车子数量>3时，可能需要调度；场地2的车子数量>7时，可能需要调度。因此，[3,7]可以看做两个场地的最佳配置。



这个结果可以和原文的图形对比查看：