****Московский государственный технический университет им. Н.Э. Баумана  
Кафедра «Системы обработки информации и управления»

Лабораторная работа №4  
по дисциплине  
«Методы машинного обучения»  
на тему

## «Реализация алгоритма Policy Iteration»

Выполнил:  
студент группы ИУ5И-22М  
Шэнь Цюцзе

Москва — 2023 г.

**1. Цель лабораторной работы**

ознакомление с базовыми методами обучения с подкреплением.

**2. Задание**

На основе рассмотренного на лекции примера реализуйте алгоритм Policy Iteration для любой среды обучения с подкреплением (кроме рассмотренной на лекции).

1. **Текст программы**

from matplotlib import pyplot as plt

import numpy as np

from scipy.stats import poisson

import seaborn as sns

"""

Hyper parameters

<int> MAX\_NUM: 每个停车场能停的最大数目（20）

<int> lam1\_rent: 停车场1租车λ值（3）

<int> lam1\_return: 停车场1还车λ值（3）

<int> lam2\_rent: 停车场2租车λ值（4）

<int> lam2\_return: 停车场2还车λ值（2）

<int> MAX\_ACTION: 最大移动汽车数目（5）

<int> CAR\_COST: 移动车辆的代价（2）

<int> CAR\_EARNING: 租车的收入（10）

<float> DISCOUNT: 收益折扣（0.9）

<np.array> actions: 动作集合（-5，-4，…，4，5）

<int> POISSON\_UPPER\_BOUND: 限制泊松分布产生请求数目的上限

<dict> poisson\_cache: 存储每个（n，λ）对应的泊松概率，key为n\*(POISSON\_UPPER\_BOUND-1)+lam

"""

MAX\_NUM = 20

lam1\_rent = 3

lam1\_return = 3

lam2\_rent = 4

lam2\_return = 2

MAX\_ACTION = 5

DISCOUNT = 0.9

CAR\_COST = 2

CAR\_EARNING = 10

actions = np.arange(-MAX\_ACTION, MAX\_ACTION + 1)

POISSON\_UPPER\_BOUND = 11

poisson\_cache = dict()

def poisson\_prob(n, lam):

    global poisson\_cache

    key = n \* (POISSON\_UPPER\_BOUND - 1) + lam

    if key not in poisson\_cache:

        poisson\_cache[key] = poisson.pmf(n, lam)

    return poisson\_cache[key]

class dp:

    def \_\_init\_\_(self):

        self.v = np.ones((MAX\_NUM + 1, MAX\_NUM + 1), float)

        self.actions = np.zeros((MAX\_NUM + 1, MAX\_NUM + 1), int)

        self.gama = DISCOUNT

        self.delta = 0

        self.theta = 0.01

        pass

    def state\_value(self, state, action, state\_value, constant\_returned\_cars):

        """

        :param state: 状态定义为每个地点的车辆数

        :param action: 车辆的移动数量[-5,5]，负：2->1，正：1->2

        :param state\_value: 状态价值矩阵

        :param constant\_returned\_cars: 将换车的数目设定为泊松均值，替换为泊松概率分布

        :return:

        """

        # initial total return

        returns = 0.0

        # 移动车辆产生负收益

        returns -= CAR\_COST \* abs(action)

        # 移动后的车辆总数不能超过20

        NUM\_OF\_CARS\_1 = min(state[0] - action, MAX\_NUM)

        NUM\_OF\_CARS\_2 = min(state[1] + action, MAX\_NUM)

        # 遍历两地全部的可能概率下（截断泊松概率）租车请求数目

        for rent\_1 in range(POISSON\_UPPER\_BOUND):

            for rent\_2 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

                prob = poisson\_prob(rent\_1, lam1\_rent) \* poisson\_prob(rent\_2, lam2\_rent)

                # 两地原本汽车数量

                num\_of\_cars\_1 = NUM\_OF\_CARS\_1

                num\_of\_cars\_2 = NUM\_OF\_CARS\_2

                # 有效租车数目必须小于等于该地原有的车辆数目

                valid\_rent\_1 = min(num\_of\_cars\_1, rent\_1)

                valid\_rent\_2 = min(num\_of\_cars\_2, rent\_2)

                # 计算回报，更新两地车辆数目变动

                reward = (valid\_rent\_1 + valid\_rent\_2) \* CAR\_EARNING

                num\_of\_cars\_1 -= valid\_rent\_1

                num\_of\_cars\_2 -= valid\_rent\_2

                # 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\_1 = lam1\_return

                    returned\_cars\_2 = lam2\_return

                    # 还车后总数不能超过车场容量

                    num\_of\_cars\_first\_loc = min(num\_of\_cars\_1 + returned\_cars\_1, MAX\_NUM)

                    num\_of\_cars\_second\_loc = min(num\_of\_cars\_2 + returned\_cars\_2, MAX\_NUM)

                    # 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\_prob(

                                returned\_cars\_first\_loc, lam1\_return) \* poisson\_prob(

                                returned\_cars\_second\_loc, lam2\_return)

                            num\_of\_cars\_first\_loc\_ = min(num\_of\_cars\_1 + returned\_cars\_first\_loc, MAX\_NUM)

                            num\_of\_cars\_second\_loc\_ = min(num\_of\_cars\_2 + returned\_cars\_second\_loc, MAX\_NUM)

                            # 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

    def policy\_iteration(self, constant\_returned\_cars=True):

        """

        :param constant\_returned\_cars:

        :return:

        """

        # 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(self.actions), cmap="rainbow", 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\_NUM + 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 = self.v.copy()

                for i in range(MAX\_NUM + 1):

                    for j in range(MAX\_NUM + 1):

                        # 更新V（s）

                        new\_state\_value = self.state\_value([i, j], self.actions[i, j], self.v, constant\_returned\_cars)

                        # in-place操作

                        self.v[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 - self.v).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\_NUM + 1):

                for j in range(MAX\_NUM + 1):

                    old\_action = self.actions[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(self.state\_value([i, j], action, self.v, 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

                    self.actions[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(self.v), cmap="rainbow", ax=axes[-1])

                fig.set\_ylabel('# cars at first location', fontsize=30)

                fig.set\_yticks(list(reversed(range(MAX\_NUM + 1))))

                fig.set\_xlabel('# cars at second location', fontsize=30)

                fig.set\_title('optimal value', fontsize=30)

                break

            iterations += 1

        # plt.title('Policy Iteration')

        plt.savefig('./policy\_iteration.png')

        plt.show()

        plt.close()

        return

    def value\_iteration(self, constant\_returned\_cars=True):

        """

        :param constant\_returned\_cars:

        :return:

        """

        # 设置迭代参数

        iterations = 0

        # 准备画布大小，并准备多个子图

        \_, axes = plt.subplots(1, 3, figsize=(40, 10))

        # 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(self.actions), cmap="rainbow", 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\_NUM + 1))))

            fig.set\_xlabel('# cars at second location', fontsize=30)

            fig.set\_title('policy {}'.format(iterations), fontsize=30)

            # value iteration 价值迭代

            while True:

                old\_value = self.v.copy()

                for i in range(MAX\_NUM + 1):

                    for j in range(MAX\_NUM + 1):

                        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(self.state\_value([i, j], action, self.v, constant\_returned\_cars))

                            else:

                                action\_returns.append(-np.inf)

                        # 找出产生最大动作价值的动作

                        max\_action = actions[np.argmax(action\_returns)]

                        # 更新V（s）

                        new\_state\_value = self.state\_value([i, j], max\_action, self.v, constant\_returned\_cars)

                        # in-place操作

                        self.v[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 - self.v).max()

                print('max value change {}'.format(max\_value\_change))

                if max\_value\_change < 1e-4:

                    print(iterations)

                    break

            # policy improvement

            policy\_stable = True

            # i、j分别为两地现有车辆总数

            for i in range(MAX\_NUM + 1):

                for j in range(MAX\_NUM + 1):

                    old\_action = self.actions[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(self.state\_value([i, j], action, self.v, 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

                    self.actions[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(self.v), cmap="rainbow", ax=axes[-1])

                fig.set\_ylabel('# cars at first location', fontsize=30)

                fig.set\_yticks(list(reversed(range(MAX\_NUM + 1))))

                fig.set\_xlabel('# cars at second location', fontsize=30)

                fig.set\_title('optimal value', fontsize=30)

                break

            iterations += 1

        # plt.title('Value Iteration')

        plt.savefig('./value\_iteration.png')

        plt.show()

        plt.close()

        return

if \_\_name\_\_ == '\_\_main\_\_':

    model = dp()

    model.policy\_iteration(constant\_returned\_cars=True)

    model1 = dp()

    model1.value\_iteration(constant\_returned\_cars=True)

    pass

1. **Результат**



