**计算智能 作业5**

2021年11月11日

**要求：**

1. 在文档中说明解题思路、方法实现、求解结果等，必要时需要对结果进行分析和讨论。
2. 将源代码作为附录粘贴于作业文档末尾，并同时作为提交的附件与作业文档一起放在压缩包内备查。
3. 独立完成，严禁抄袭。

**内容：**

1. 编写程序，用人工神经网络求解旅行商（TSP）问题。

说明：

1. 参考网络结构：Hopfield Neural Network
2. 编程语言不限（需提交源代码和运行结果截图）。

3) 作业相关数据说明（15个城市，最短路径距离291）：

a) p01.tsp, the TSP specification of the data.

b) p01\_d.txt, the intercity distance table.

c) p01\_s.txt, an itinerary that minimizes the total distance.

d) p01\_sxy.txt, the XY coordinates of the minimal itinerary.

e) p01\_sxy.png, an image of the minimal itinerary.

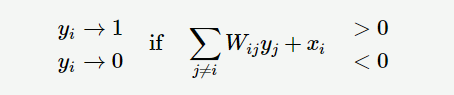
f) p01\_xy.txt, a set of XY coordinates for the cities, inferred from the distances.

g) p01\_xy.png, an image of the XY coordinates

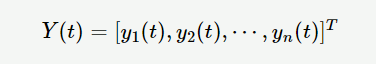
### 本文采用离散 Hopfield 网络处理TSP问题

离散的 Hopfield 网络是二值神经网络，该模型的处理单元由神经元构成，每个神经元由两种状态：激活或者抑制状态，分别用 1 和 0 表示。神经元之间通过赋有权值的有向线段连接，通过求取全局状态的最小能量来训练模型。

它由三个神经元构成，图中的第 0 层仅仅是作为网络的输入，所以不把它当作实际的神经元，无计算功能，第 1 层是实际的神经元。设 yi表示第 i个神经元的取值，xi表示第 i个神经元的外部输入，可以理解为额外施加在神经元 i上的固定偏置，功能相当于第 i个神经元的阈值（threshold），Wi,j 表示第 j个神经元到第 i个神经元的连接权重该简单模型的计算方式如下



对于一个离散的 Hopfield 网络，其网络状态是输出神经元信息的集合。对于一个输出层是 n个神经元的网络，则其 t时刻的状态为一个 n维向量



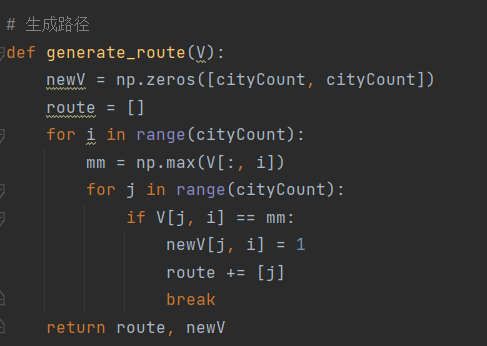
在TSP问题中，这个时刻就可以代表t时刻的路径

即问题解决方案：

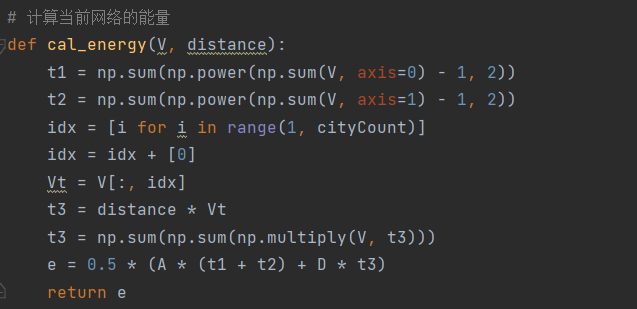
对于走过的每一条可能的路径，每个城市都只能走过一次且需要一一走过每个城市，因此这一方阵中的每一行和每一列都只能有一个元素为1，其余元素为0。这边是TSP问题的约束条件。具有上述约束条件的每一路径可用一换位阵表示。

将TSP问题映射为神经网络动力系统可用以下步骤完成：

1. 将TSP问题的每一条可能路径用一换位矩阵表示，并给出相应的距离表示；



1. 将TSP问题的换位阵集合与由N个神经元构成的神经元阵列相对应；每一条路径所对应的换位阵的各元素与相应的神经元稳态输出对应；
2. 找出一反应TSP约束优化问题的能连函数E；

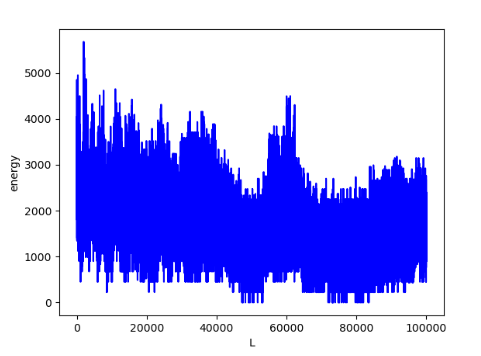


1. 求出使E取极小值的神经网络连接权重矩阵和偏置参数。

实验结果：

（实验中仅采用坐标，并用坐标计算距离，计算出来最优是284）





实验反思

缺点就是不一定每次都可以找到最优解，但是这个是使用神经网络解决TSP问题的出色尝试。

源代码：

import matplotlib.pyplot as plt  
import numpy as np  
import copy  
  
# the number of city  
cityCount = 15  
  
# 旅行商问题 ( TSP , Traveling Salesman Problem )  
data\_xy = np.array([[-0.0000000400893815, 0.0000000358808126],  
 [-28.8732862244731230, -0.0000008724121069],  
 [-79.2915791686897506, 21.4033307581457670],  
 [-14.6577381710829471, 43.3895496964974043],  
 [-64.7472605264735108, -21.8981713360336698],  
 [-29.0584693142401171, 43.2167287683090606],  
 [-72.0785319657452987, -0.1815834632498404],  
 [-36.0366489745023770, 21.6135482886620949],  
 [-50.4808382862985496, -7.3744722432402208],  
 [-50.5859026832315024, 21.5881966132975371],  
 [-0.1358203773809326, 28.7292896751977480],  
 [-65.0865638413727368, 36.0624693073746769],  
 [-21.4983260706612533, -7.3194159498090388],  
 [-57.5687244704708050, 43.2505562436354225],  
 [-43.0700258454450875, -14.5548396888330487]])  
  
  
# 得到距离矩阵的函数  
def get\_distance\_matrix(data\_xy):  
 num = data\_xy.shape[0] # 15个坐标点  
 distmat = np.zeros((15, 15)) # 15X15距离矩阵  
 for i in range(num):  
 for j in range(i, num):  
 distmat[i][j] = distmat[j][i] = round(np.linalg.norm(data\_xy[i] - data\_xy[j]))  
 return distmat  
  
  
  
# 计算路径的长度  
def cal\_distance(distance, route):  
 dis = 0  
 length = len(route)  
 for i in range(length - 1):  
 dis += distance[route[i]][route[i + 1]]  
 dis += distance[route[length - 1]][route[0]]  
 return dis  
  
  
# 画出最优路径  
def draw\_best\_path(bestRoute, data\_xy):  
 plt.figure(2)  
 x = []  
 y = []  
 for i in bestRoute:  
 x.append(data\_xy[i - 1][0])  
 y.append(data\_xy[i - 1][1])  
 plt.scatter(x, y)  
 plt.plot(x, y)  
 for i in range(len(data\_xy)):  
 plt.text(data\_xy[i][0], data\_xy[i][1], r' ' + str(i + 1))  
 plt.xlabel('x')  
 plt.ylabel('y')  
 plt.show()  
  
  
# 画出解的能量变化  
def draw\_energy(energys):  
 plt.figure(1)  
 plt.plot(np.arange(0, len(energys), 1), energys, color='b')  
 plt.xlabel("L")  
 plt.ylabel("energy")  
 plt.show()  
  
  
# 动态方程计算微分方程du  
def cal\_du(V, distance):  
 sum\_col = np.sum(V, axis=0) - 1 # 列之和  
 sum\_row = np.sum(V, axis=1) - 1 # 行之和  
 temp1 = np.zeros((cityCount, cityCount))  
 temp2 = np.zeros((cityCount, cityCount))  
 for i in range(cityCount):  
 for j in range(cityCount):  
 temp1[i, j] = sum\_col[j]  
 for i in range(cityCount):  
 for j in range(cityCount):  
 temp2[j, i] = sum\_row[j]  
 # 将第一列移动到最后一列  
 c\_1 = V[:, 1:cityCount]  
 c\_0 = np.zeros((cityCount, 1))  
 c\_0[:, 0] = V[:, 0]  
 c = np.concatenate((c\_1, c\_0), axis=1)  
 c = np.dot(distance, c)  
 return -A \* (temp1 + temp2) - D \* c  
  
  
# 更新神经网络的输入电压U  
def cal\_U(U, du, step):  
 return U + du \* step  
  
  
# 更新神经网络的输出电压V  
def cal\_V(U, U0):  
 return 1 / 2 \* (1 + np.tanh(U / U0))  
  
  
# 计算当前网络的能量  
def cal\_energy(V, distance):  
 t1 = np.sum(np.power(np.sum(V, axis=0) - 1, 2))  
 t2 = np.sum(np.power(np.sum(V, axis=1) - 1, 2))  
 idx = [i for i in range(1, cityCount)]  
 idx = idx + [0]  
 Vt = V[:, idx]  
 t3 = distance \* Vt  
 t3 = np.sum(np.sum(np.multiply(V, t3)))  
 e = 0.5 \* (A \* (t1 + t2) + D \* t3)  
 return e  
  
  
# 生成路径  
def generate\_route(V):  
 newV = np.zeros([cityCount, cityCount])  
 route = []  
 for i in range(cityCount):  
 mm = np.max(V[:, i])  
 for j in range(cityCount):  
 if V[j, i] == mm:  
 newV[j, i] = 1  
 route += [j]  
 break  
 return route, newV  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 A = cityCount \*\* 2  
 D = cityCount / 2  
 U0 = 0.0001 # 初始电压  
 step = 0.001 # 步长  
 L = 100000 # 迭代次数  
 flag = False  
  
 distance = get\_distance\_matrix(data\_xy) # 获取城市的距离矩阵  
  
 # 初始化神经网络的输入状态（电路的输入电压U）  
 U = 1 / 2 \* U0 \* np.log(cityCount - 1) + (2 \* (np.random.random((cityCount, cityCount))) - 1)  
 # 初始化神经网络的输出状态（电路的输出电压V）  
 V = cal\_V(U, U0)  
  
 energys = np.array([0.0 for x in range(L)]) # 每次迭代的能量  
 best\_distance = np.inf # 最优距离  
 best\_route = [] # 最优路线  
 distances = [] # 每个路径解的距离  
  
 # 迭代训练网络  
 for i in range(L):  
 # 利用动态方程计算du  
 du = cal\_du(V, distance)  
 # 由一阶欧拉法更新下一个时间的输入状态（电路的输入电压U）  
 U = cal\_U(U, du, step)  
 # 由sigmoid函数更新下一个时间的输出状态（电路的输出电压V）  
 V = cal\_V(U, U0)  
 # 计算当前网络的能量E  
 energys[i] = cal\_energy(V, distance)  
 # 检查路径的合法性  
 route, newV = generate\_route(V)  
 if len(np.unique(route)) == cityCount:  
 route.append(route[0])  
 dis = cal\_distance(distance, route)  
 distances.append(dis)  
 print('第{}次迭代找到的符合条件的当前距离为：{}，能量为：{}，路径为：{}'.format(i, dis, energys[i], route))  
 if dis <= best\_distance:  
 best\_distance = dis  
 best\_route = copy.deepcopy(route)  
 flag = True  
 if flag:  
 draw\_energy(energys) # 画出能量变化  
 draw\_best\_path(best\_route, data\_xy)  
 print('神经网络找到的最优解距离为：{}，路径为：{}'.format(best\_distance, best\_route))  
 else:  
 print('没有找到最优解')