TSP 问题 霍普菲尔德网络

1.实验环境：

- OS: Windows 10
- CPU: intel CORE i5
- Python 3.6
- Sckikit_learn 0.18.1
- 其他 Python 科学计算模块
- 编译器：PythonCharm 2017.3.3

2.主要参考文档：

- Sklearn 0.18 官方文档

3.实验过程

3.1.实验问题描述

TSP 问题（Traveling Salesman Problem），是数学领域著名的问题之一，设有一个旅行商人要拜访 n 个城市，他必须选择要走的路径，路径的限制是每一个城市只能拜访 1 次，而且最后要回到原来的城市。路径的选择目标是要求得到的路径为所有路径之中的最小值。

TSP 问题是一个组合优化问题。该问题被证明具有 NPC 计算复杂度。迄今为止，还没有一个有效的算法，所以大家认为这类问题的大型实例不能用精确算法求解，必须寻求这类问题的有效的近似算法。

本实验使用霍普菲尔德网络（Hopfield Network）算法来解决 TSP 问题。

3.2.实验资源库：

本实验在测试过程中，使用到了一些数据来自于 TSPLIB 网站，但是这个网站的很多数据来自于实际的生活中，所以坐标点集合有些过于排列整齐，我对数据进行了一定的选择以及修改。

TSPLIB 网站网址：

http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

3.3.实验方法与原理

这个实验参考了网上的博客，在充分理解的基础上做出了自己的修改和一些必要的补充。

首先简单阐述一下霍普菲尔德网络的原理以及其在 TSP 问题中的应用：

霍普菲尔德网络是美国物理学家霍普菲尔德提出的一种反馈型神经网络模型，分为离散霍普菲尔德网络和连续霍普菲尔德网络两种模型。霍普菲尔德网络可以用于实际联想和优化计算。霍普菲尔德网络的工作基础理论是：通过能量最小化达到系统的稳定状态。

霍普菲尔德网络中任意两个神经元之间均有连接。网络的输出要反复的作为输入，使得网络状态不断发生变化，这其中存在网络的稳定性问题：

$$\xi_i(t+1)=\sum_{j=0}^{n} w_{ij}s_j(t)+I_i$$

$$s_i(t+1)=\begin{cases}1 & \xi_i(t+1)\geq 0\\ -1 & \xi_i(t+1)<0\end{cases}$$

其中，I_i表示网络中的第 i 个神经元的输入信号，$s_j(t)$表示 t 时刻霍普菲尔德网络中的第 i 个神经元的状态，$\xi_i(t)$表示 t 时刻第 i 个神经元对输入信号和反馈信号的整合结果。

有意思的是：霍普菲尔德网络借助了热力学中的动力系统稳定性分析的利亚霍普夫定理来分析网络的稳定性。

有如下的能量函数：

$$\mathrm{E}=-\frac{1}{2}\sum_{i=0}^{n}\sum_{j=0}^{n} w_{ij}s_is_j-\sum_{i=1}^{n}\boldsymbol{I}_is_i$$

当没有外部输入信号时，修改能量函数简化为

$$\mathrm{E}=-\frac{1}{2}\sum_{i=0}^{n}\sum_{j=0}^{n} w_{ij}s_is_j=-\frac{1}{2}\boldsymbol{S}^T\boldsymbol{w}\boldsymbol{S}$$

如果离散型霍普菲尔德网络的连接矩阵权值是对称矩阵并且对角元素为 0，即：

$$w_{ij}=\begin{cases}w_{ij} & i\neq j\\ 0 & i=j\end{cases}$$

满足这个条件以后，能量函数就会随着时间的推移不断减小，此时认为霍普菲尔德网络是稳定的，从任意给定的初始状态出发，都可以收敛到某一个稳定的状态。

霍普菲尔德网络的工作的步骤如下：

- 向网络中输入数据
- 随机选择一个神经元（串行模式）或一组神经元（并行模式）
- 按照神经元的整合函数和激活函数，更新所选中的神经元的状态
- 重复二，三两步，直到网络达到稳定的状态

当霍普菲尔德网络收敛到稳定状态时，其能量函数达到极小，这就使得了其具备实现优化计算的能力。

在具体的 TSP 问题中，霍普菲尔德网络的能量函数写作：

$$\mathrm{E}=-\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1,j\neq i}^{N} T_{ij}v_iv_j-\sum_{i=1}^{N}v_iI_i+\sum_{i=1}^{N}\frac{1}{R_i}\int_0^{v_i}f^{-1}(v)dv$$

该算法在 TSP 问题中的具体实现步骤为：

1） 置初值和权值，t=0，A=1.5，D=1.0，U_0=0.02

2） 读入连续 N 个城市之间的距离$d_{xy}(x,\ y=1,\ 2,\cdots,\ N)$；

3） 神经网络输入$U_{xi}(t)$的初始化 x， i=1， 2,…， N;

$$U_{xi}(t) = U_0' + \delta_{xi}$$

其中，$U_0' = \frac{1}{2}U_0\ln(N-1)$，N 为城市个数，$\delta_{xi}$为（-1， +1）区间的随机值；

4） 利用动态方程计算$\frac{dU_{xi}}{dt}$

5） 一阶欧拉方程计算$U_{xi}(t+1)$

$$U_{xi}(t+1) = U_{xi}(t) + \frac{dU_{xi}}{dt}\Delta T$$

6） 采用 sigmoid 函数计算$V_{xi}(t)$;

$$V_{xi}(t) = \frac{1}{2}(1 + \tanh(\frac{U_{xi}(t)}{U_0}))$$

7） 计算能量函数 E

8） 检查路径的合法性，判断迭代是否结束，若未结束返回到第 4 步；

9） 输出迭代次数，最优路径，能力函数，路径长度和能量变化

下面结合具体的代码进行说明：

首先需要对所有的参数进行初始化，这个实验我们采用的参数调整参考了网上学长的一篇博客，参数没有进行变化，取得相对不错的效果。

```
def __init__(self, cities):
    '''''
    网络初始化
    :param cities: 城市坐标集合
    '''
    self.cities = cities
    self.num_cities = len(cities)

    self.U0 = 0.02  # 初始变化率
    self.alpha = 1e-6
    self.size_turePath = 1.5

    #随机初始化矩阵网络参数
    Network = (self.num_cities, self.num_cities)
    self.Distance = np.empty(Network)
    for i in range(self.num_cities):
        for j in range(i, self.num_cities):
            V = (cities[i][0] - cities[j][0],
                     cities[i][1] - cities[j][1])
            #更新距离集合
            self.Distance[i, j] = np.linalg.norm(V)
            self.Distance[j, i] = self.Distance[i, j]
    self.actual_Distance = self.Distance
    self.Distance = self.Distance / self.Distance.max()

```

```
26.     #初始化网络输入
27.     self.U = np.ones(Network)
28.     self.U /= self.num_cities ** 2
29.     #得到随机的 δ 值
30.     self.U += np.random.uniform(-0.5, 0.5, Network) / 10000
31.     self.delta_U = np.zeros(Network)
32.     #初始化输出
33.     self.V = self._Sigmoid(self.U)
```

这里在初始时刻参照公式$U_{xi}(t) = U_0' + \delta_{xi}$和公式$U_0' = \frac{1}{2}U_0 \ln(N-1)$, 对初始时刻的输入值和输出值都进行了初始化。为了便于表示，我们对公式进行了一些离散化。
接下来是激活函数的表示：

```
1.  def _Sigmoid(self, u):
2.      '''''
3.      激活函数
4.      :param u: 设定的参数
5.      :return: 函数值
6.      '''
7.      return 0.5 * (1 + np.tanh(u / self.U0))
```

这个表示是较为简单的，直接可以根据函数库表示出来。
接下来给出了整个算法中最为关键的部分，即网络的迭代的过程，下面给出了网络迭代一次的函数和整个实验的迭代函数：

```
1.  def Iter(self, max_iter):
2.      '''''
3.      迭代函数
4.      :param max_iter: 最大迭代次数
5.      :return:
6.      '''
7.
8.      #每迭代 300 次输出迭代的矩阵图像
9.      iter = 300
10.     for i in range(1, max_iter + 1):
11.         self.Iter_once()
12.         if iter and i % iter == 0:
13.
14.             self._Mat(self.V)
15.
16.     Path = np.array(self.V).argmax(0)
17.     if iter:
```

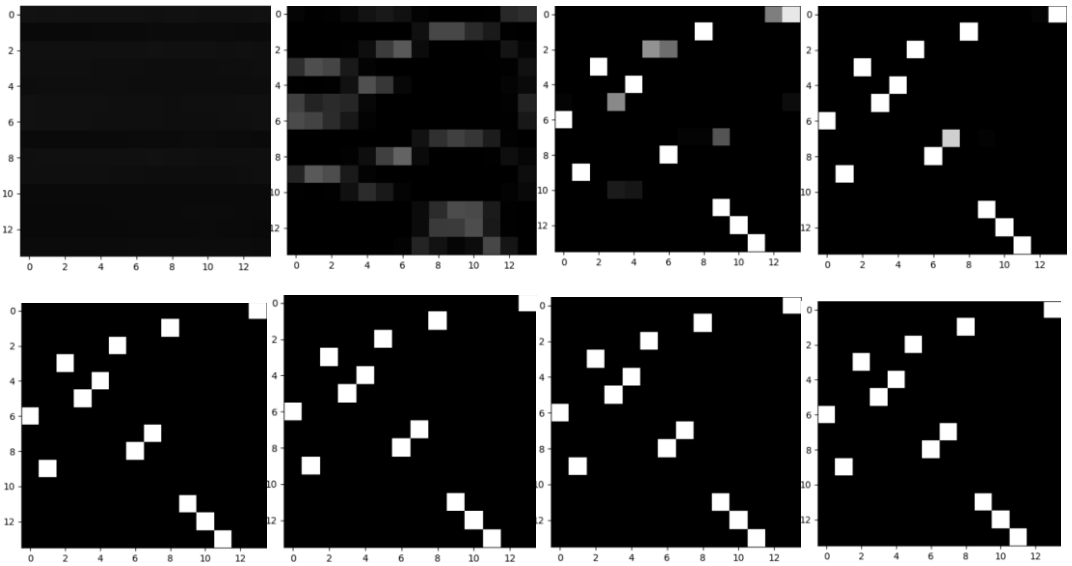
```
            self._Path(Path)
        return Path.tolist()


def Iter_once(self):
    '''''
    迭代一次函数，更新输入和输出值
    :return:
    '''
    self.delta_U = np.zeros((self.num_cities, self.num_cities))
    for city in range(self.num_cities):
        for p in range(self.num_cities):
            self.delta_U[city, p] = \
                self.alpha * self._Delta(city, p)
    self.U += self.delta_U
    self.V = self._Sigmoid(self.U)

```

整个过程都是按照基本公式进行的，在每次迭代之后都要根据输入，更新下一次输入和本次输出值，并且根据本次输出值，得到 delta 值，得到输入的更新值，这样不断更新，直到到达最大的迭代次数。并且根据霍普菲尔德网络的稳定性的原理，网络一定会趋于收敛，最后会得到一个值，结果体现在神经网络的矩阵上。

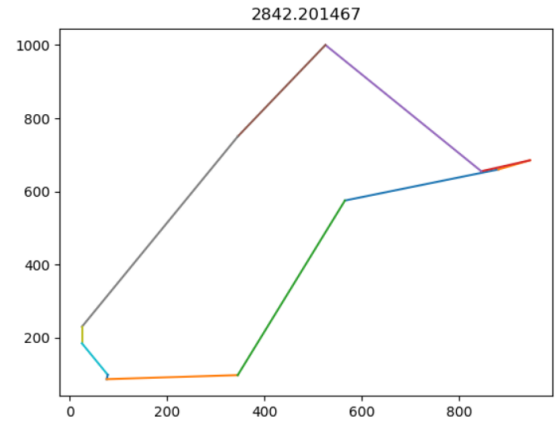
矩阵的每一行表示一个城市，每一列表示城市被访问的顺序，最后得到的矩阵是每一行每一列都只有一个神经元是活跃的，这样整个网络就是稳定的。

3.4 实验结果分析与思考

在这个实验中，由于神经网络需要使用矩阵表示，大的数据的迭代表示不方便，我们选取了 15 个城市节点的问题，并且迭代次数选择了 2400，每迭代 300 次输出一次当前的网络矩阵。



上图所输出的矩阵是每迭代 300 次所输出的矩阵，可以发现，在基本 1200 次迭代过后，网络已经趋于稳定，每个神经元的输入与输出都已经稳定，并且，满足了 TSP 问题的特性，每行每列都只有一个神经元是处于激活状态的。矩阵的每一行的编号就是城市的访问顺序。




```
The Length is: 2842.201467
The Path is: [6, 9, 3, 5, 4, 2, 8, 7, 1, 11, 12, 13, 0, 0]
```

最后得到的路径的连线如图所示，路径的长度总和为 2842.201467，路径已经在控制台给出。
整个效果还是不错的。

霍普菲尔德网络的理论据说在哲学上都是一件非常值得惊叹的事情，它让物质和意识联系了起来，让物质可以自动拥有意识，这是非常神奇的。因为霍普菲尔德网络的趋于能量最小的性质，网络可以作为样本记忆信息，有了回忆能力。能够从某一残缺的信息回忆起来所属的完整的记忆信息。

但是离散的霍普菲尔德也具有一定的局限性：

首先是记忆容量有限。从本实验就可以看出来，网络使用的是二维的网络表示的。

存在伪稳定点的联想与记忆问题，这在有些实际问题中的影响会较大。

当记忆样本较为接近的时候，网络不能始终回忆出正确的记忆等。另外还存在一个问题就是网络的平衡稳定点并不可以任意设置，也没有一个通用的方式事先知道平衡稳定点，所以，最后回忆成什么样子是不可以预期的。