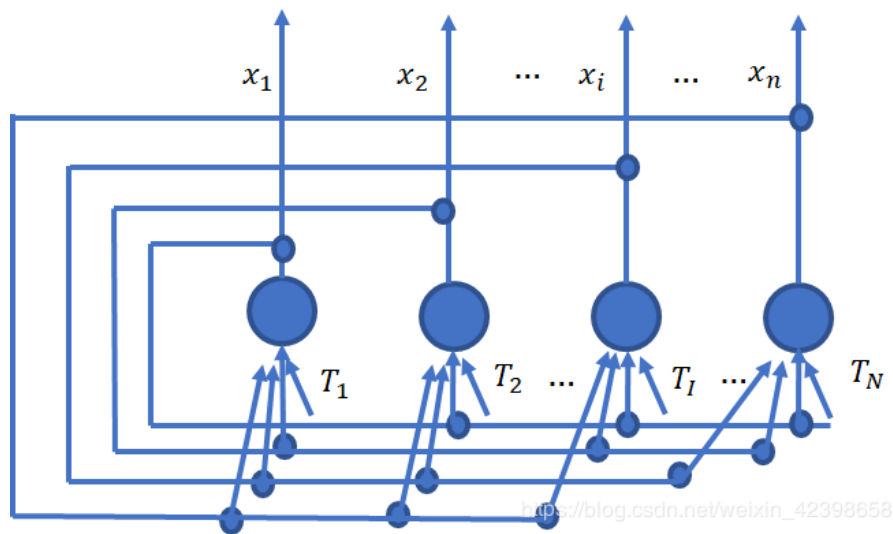


# 认知科学与类脑计算课程实验报告

实验题目： Hopfield 模型的实现		学号： 201600181073
日期： 2019. 5. 16	班级： 智能 16	姓名： 唐超
<p>实验目的：</p> <p>根据 Hopfield 神经网络的相关知识，设计一个具有联想记忆功能的离散型 Hopfield 神经网络。要求该网络可以正确识别 0-9 这 10 个数字，当数字被一定的噪声干扰后，仍具有较好的识别效果。</p>		
<p>实验环境：</p> <p>Jupyter notebook &amp; python3.6 &amp; numpy</p>		
<p>主要算法及步骤：</p> <p>Hopfield 网络是 John Hopfield 在 1982 年发明的一种单层的、输入输出为二值的反馈网络。它主要用于联想记忆，当网络的初始态确定后，网络状态按其工作规则向能量递减的方向变化，最后接近或达到平衡点。如果设法把网络所需记忆的模式设计成某个确定网络状态的一个平衡点，则当网络从与记忆模式较接近的某个初始状态出发后，按 Hopfield 运行规则进行状态更新，最后网络状态稳定在能量函数的极小值点，即记忆模式所对应的状态。这样就完成了由部分信息或失真的信息到全部或完整信息的联想记忆过程。其网络结构如下图。</p> <div data-bbox="367 1341 1264 1872"></div> <p>该网络有 <math>n</math> 个神经元，所有神经元的状态表示为</p> $x = (x_1, x_2, \dots, x_n), \quad x_i = \pm 1, i = 1, 2, \dots, n$		

网络的连接权重表示为

$$W = (\omega_{ij})_{n \times n}, \omega_{ii} = 0, i = 1, 2, \dots, n$$

任一神经元的输出均通过连接权重矩阵接收其他所有神经元输出的反馈信息，其目的就在于任何一个神经元都受到所有神经元输出的控制，从而使各神经元的输出相互制约。为了反映对输入的噪声的控制，每个神经元均设有一个阈值

$$T = (T_1, T_2, \dots, T_n)$$

- **网络的更新规则：**

给定初始状态  $x(0) = (x_1(0), x_2(0), \dots, x_n(0))$  后，网络状态就开始不断更新，规则如下：

$$x_i = \text{sign}(I_i - T_i) = \begin{cases} 1, & I_i \geq T_i \\ -1, & I_i < T_i \end{cases}, i = 1, 2, \dots, n$$

其中  $I_i$  为网络节点  $i$  的加权输入和

$$I_i = \sum_{j=1}^n \omega_{ij} x_j, i = 1, 2, \dots, n$$

另外，Hopfield 的网络结点可以采用同步或异步两种更新方式。对于同步更新，网络每运行一次，所有神经元同时调整状态。对于异步更新，网络运行一次只有一个神经元更新，调整顺序可以按照预先设定顺序进行调整，也可以随机选定。异步更新的方式更加符合生物大脑神经网络的工作方式，一般来说，其工作效率也更高。

当网络状态不再改变时，说明该网络收敛到了稳态，此时的状态作为输出状态。可以证明，若按照异步方式调整网络的状态，且连接权重矩阵  $W$  为对称阵，则对于任意的初态，网络都能收敛。若按照同步方式进行调整，且连接权重矩阵  $W$  为非负定对称矩阵，对于任意初态，网络都能收敛。

- **权重  $W$  的确定：**

Hopfield 网络是一种反馈式神经网络，与 BP 神经网络这类前馈式神经网络不同，其学习规则是基于灌输式学习，即网络的权值不是通过训练出来的，而是按照一定规则计算出来的，使得在网络更新达到稳态时能量函数  $E$  达到极小值。

由神经动力学的知识，一般将每个网络结点的能量定义为：

$$E_i = -\frac{1}{2} I_i x_i, i = 1, 2, \dots, n$$

因此，可以推导出具有 n 个结点的离散型 Hopfield 网络的总能量为：

$$E = -\frac{1}{2} \sum_{j=1}^n \sum_{i=1}^n \omega_{ij} x_i x_j + \sum_{i=1}^n T_i x_i$$

当我们试图用 Hopfield 网络存储一张图片  $x = (x_1, x_2, \dots, x_n)$ ， $x_i = \pm 1, i = 1, 2, \dots, n$  时，取

$$\omega_{ij} = x_i x_j, T_i = 0$$

显然，此时能量函数取得最小值

$$E = -n^2$$

若需要存储另外一张照片，只需要按照同样的方法得到另外一个权重矩阵，将两个权重矩阵相加即可，这样做的原理可以解释为在 Hopfield 网络的状态空间里人为地构造另一个极小值点，从而实现了记忆这一照片的功能。

## 程序设计：

1. 设计 6\*5 数字点阵，将数字 0-9 的矩阵设计好存储到列表中。

定义字典 numDict，其 key 为数字，value 为数字点阵的列表，方便存储与访问。数字点阵中数字存在部分用 1 表示，空白部分用-1 表示。

2. 创建网络。

定义 Hopfield 类，包括更新权重函数 train，预测函数 predict，画图函数 printNum。Hopfield 类的输入为需记忆的一组模式 memory，每个模式(数字点阵)的行数 row，列数 col。

train 函数对 memory 按照上一栏权重的更新规则进行更新，将每张图片更新的权重累加到 self.W 上，最后将其对角线元素全部置为 0。另外注意在每次调试训练时需重置 self.W, 具体如下：

```
def train(self):
    # 重置W
    self.W = np.zeros((self.N, self.N), dtype = np.float32)
    numArray = np.asarray(self.memory, dtype = np.int8)
    # 更新W
    for i in range(len(numArray)):
        v = numArray[i].reshape((1, self.N))
        self.W += np.dot(v.T, v) / self.N
    self.W[range(self.N), range(self.N)] = 0
    print('weights: \n', self.W)
```

predict 函数对初始输入状态按照上一栏中网络状态的更新规则进行更新，设置最大迭代次数为 100，当网络收敛到稳态时停止迭代并调用 printNum 函数画出预测数字。

printNum 函数的主要功能是将数字点阵中的 1, -1 分别用 '\*', ' ' 代替，并作为字符串依次按行输出，具体如下：

```
def printNum(self, Num):
    a = np.asarray(Num)
    b = ['* ' if i==1 else ' ' for i in a]
    for i in range(self.row):
        print(''.join(b[:self.col]))
        b = b[self.col:]
    return a
```

### 3. 产生带噪声的数字点阵。

定义增加噪声的函数

```
def addNoise(noiseType, originNum, noiseSize=0):
```

其中 originNum 为先前定义的数字点阵，将其转换为 numpy 数组并赋值给 NoiseNum，在 NoiseNum 基础上增加噪声，最后返回增加噪声的结果，noiseType 为产生噪声的类型，当 noiseType=='random' 时，随机选取 originNum 中的元素，变为其相反数 ( $1 \rightarrow -1$ ,  $-1 \rightarrow 1$ )。当 noiseType=='half' 时，originNum 的下半部分全部置为 -1。

### 4. 数字识别测试。

先将 numDict 输入到 train 函数计算网络权重，再将某个原始数字点阵用 addNoise 函数加入噪声，将带噪声的数字点阵输入到创建好 Hopfield 网络，网络的输出是与该数字点阵最为接近的目标向量，最后用 Hopfield 类的 printNum 函数画出，判断是否与原始数字点阵符合。

### 调试分析：

1. 刚开始做时，采用 0, 1 设计数字点阵，测试结果始终不好，后来细看 Hopfield 权重更新的原理，当输入为 1, -1 组成的向量时才适用这样的更新规则，对于 0, 1 设计数字点阵，需加入归一化处理：

$$norm = \frac{x - \min(x)}{\max(x) - \min(x)}$$

进行以上改进后，基本能实现数字点阵的记忆功能。

2. 后来觉得根据原理的分析，将数字点阵改为 1, -1 更为合理（代码中保留的这一版本），但在将 list 转化为 numpy 数组时，忘了将原本 dtype=uint8 改为 int8，导致网络状态的-1 变为了 255，后来通过调试输出找到了这一问题。
3. 在 addNoise 函数中，直接在 originNum 上增加噪声，并将其作为返回结果，导致多次测试后，原始数字点阵上的噪声越来越多，后来发现这一问题，将 originNum 转换为 numpy 数组并赋值给另一局部变量 noiseNum，在 noiseNum 上进行操作。

### 测试结果及分析：

部分测试结果如下：

- 随机噪声测试

```
toPredictNum = four  noiseType = random  noiseSize = 2
```

原始数字:

```
  * *  
 *  *  
*   *  
* * * * *  
  *  
  *
```

加入噪声:

```
  * *  
 *  *  
*   *  
  * * *  
  *  
  *
```

联想结果:

```
  * *  
 *  *  
*   *  
* * * * *  
  *  
  *
```

```
toPredictNum = nine  noiseType = random  noiseSize = 3
```

原始数字:

```
 * * *  
*   *  
*   *  
* * * * *  
  *  
* * *
```

加入噪声:

```
 * * *  
*   *  
* * * * *  
  * *  
* * *
```

联想结果:

```
 * * *  
*   *  
*   *  
* * * * *  
  *  
* * *
```

```
toPredictNum = six  noiseType = random  noiseSize = 5
```

原始数字:

```
 * *
  *
* * * *
*      *
*      *
```

加入噪声:

```
 * * *
  *
* * * *
*      *
```

联想结果:

```
 * * *
  *
*      *
* * * * *
*      *
*      *
```

- “half” 噪声测试

```
toPredictNum = two  noiseType = half  noiseSize = 0
```

原始数字:

```
 * * *
*      *
  *
  *
  *
```

加入噪声:

```
 * * *
*      *
```

联想结果:

```
 * * *
*      *
  *
  *
  *
* * * * *
```

```
toPredictNum = one    noiseType = half    noiseSize = 0
```

原始数字:

```
*  
* *  
*  
*  
*  
*
```

加入噪声:

```
*  
* *  
*
```

联想结果:

```
*  
* *  
*  
*  
*  
*
```

```
toPredictNum = five   noiseType = half   noiseSize = 0
```

原始数字:

```
* * * * *  
*  
* * * * *  
*  
*
```

加入噪声:

```
* * * * *  
*  
* * * * *
```

联想结果:

```
* * * * *  
*      *  
* * * * *  
*      *  
*      *  
* * * * *
```

可以看出，由于需记忆的模式较多，6\*5 点阵本身又较小，很多数字点阵差异并不大，如“5”，“6”，“8”，“9”等，对于随机噪声测试，当 noiseSize 较小时，测试效果较好，当 noiseSize 较大时，干扰信息过多导致很难联想正确。对于“half”噪声测试，由于丢失信息较多，只特征较为明显的数字点阵，如“1”，“2”等，能够正确联想。