认知科学与类脑计算

实验报告三

日期：2019/5/23

班级：16 人工智能

姓名：贾乘兴

学号：201600301304

一.需求分析：

由赫布提出的 Hebb 学习规则是一个无监督学习规则，这种学习的结果是使网络能够提取训练集的统计特性，从而把输入信息按照它们的相似性程度划分为若干类。这一点与人类观察和认识世界的过程非常吻合，人类观察和认识世界在相当程度上就是在根据事物的统计特征进行分类。Hebb 学习规则只根据神经元连接间的激活水平改变权值，因此这种方法又称为相关学习或并联学习。

巴普洛夫实验是一个典型的 Hebb 学习模型，其具体内容是每次给狗送食物以前响起铃声，这样经过一段时间以后，铃声一响，狗就开始分泌唾液。本次实验目的在于加深对 Hebb 学习模型的理解，能够使用 Hebb 学习模型解决简单问题。设计 6*5 数字点阵。有数字部分用 1 表示，空白部分用-1 表示，将数字 0-2 的矩阵设计好存储到列表中。创建网络后，将训练数据加入噪声作为测试数据，输入到创建并训练好的网络中，网络的输出是与该数字点阵最为接近的目标向量。

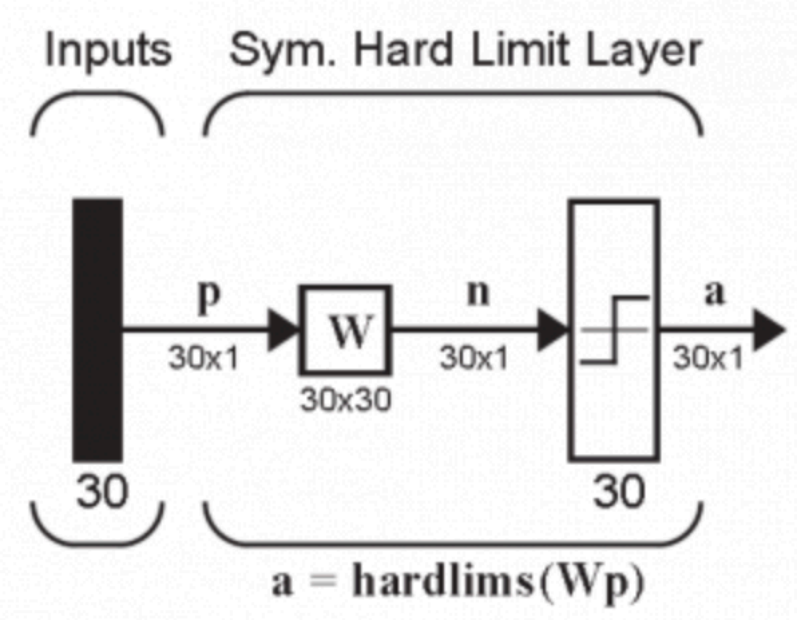
二.概要设计：

利用 python 的 numpy 库，实现简单的 Hebb 规则下的较为鲁棒的数字识别，并尝试实现广义的 Hebb 学习算法，并测试结果

三.详细设计：

1. Hebb 规则

Hebb 规则的含义是两者刺激不断增强，则两者的联系也增强，若刺激减少，则联系也减弱。



Hebb 算法可以描述为:如果一个处理单元从另一处理单元接收输入激励信号，而且如果两者都处于高激励电平，那么处理单元之间的加权就应当增强。用数学来表示，就是两节点的连接权将根据两节点的激励电平的乘积来改变，即

$$\Delta w_{ij} = w_{ij}(n+1) - w_{ij}(n) = \eta y_j x_i$$

其中 $w_{ij}(n)$表示第(n+1)次调解前，从节点 j 到节点 i 的连接权值；$w_{ij}(n+1)$是第(n+1)次调解后，从节点 j 到节点 i 的连接权值；η为学习速率参考；x_j 为节点 j 的输出，并输入到节点 i；y_i 为节点 i 的输出。

对于 Hebb 学习规则，学习信号简单地等于神经元的输出

$$r = f\left(W_i^T X\right)$$

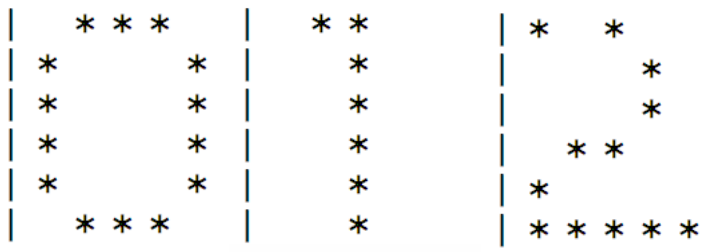
权向量的增量变成

$$\Delta W_i = \eta f\left(W_i^T X\right) X$$

这个学习规则在学习之前要求在 $W_i = 0$ 附近的小随机值上对权重进行初始化。

这个规则说明了如果输出和输入的叉积是正的，则权增加，否则减小。

2. 手写数字点阵设计（0-2）



3. 协方差率

Hebb 假设的原意是突触联系系数(权重)的改变依赖于突触前神经元的兴奋X和突触后神经元的兴奋Y，之后的生理学发现的长时程突触增强效应(LTP)说明符合 Hebb 的原意，但是，随后长时程突触压抑效应(LTD)的发现，表明如果突触前的活动伴随的是突触后的细胞的低水平活动，它会降低突触效益，因此有一种负的作用。修改如下。

$$\tau \frac{dW}{dt} = \left(Y - \theta_1\right) X$$

其中θ_1为阈值，它表明突触后兴奋性高于此值时用 LTP(为正数)，低于此值时 LTD(为负值)

4. BCM 率

Bienenstock，Cooper&Munro(1982)提出另一种学习律，也是无监督的，有实验证据表明，突触前和突触后活动都能影响突触联系系数的改变，他们将这种新的学习律称为 BCM 学习律，表达式如下

$$\tau \frac{dW}{dt} = YX\left(Y - \theta_1\right)$$

θ_1是突触后活动的阈值，由它决定突触联系系数是增加还是减少。如果θ_1是可调节的，情况就会不一样。BCM 学习律中假定θ_1可变，而且比输出值Y还快，则 BCM 学习律会稳定。

5. 突触联系系数的正规化(normalization)

$$\tau \frac{dW}{dt} = YX - \alpha Y^2 W$$

$$\tau \frac{d|W|^2}{dt} = 2\tau W \frac{dW}{dt} = 2\tau W(YX\text{-}\alpha Y^2 W)$$

$$= 2\tau YWX\text{-}2\tau\alpha Y^2 |W|^2$$

$$= 2Y^2(1 - \alpha|W|^2)$$

$|W|^2$会逐渐收敛到$1/\alpha$，而不会达到无界值。这也保证W的各分量之间可以互相竞争，有的突触联系系数增加，必然引起有的突触联系系数减少。

6. 广义的 Hebb 学习率

$$y_i(n) = \sum_{i=1}^{m} w_{ji}(n) x_i(n)$$

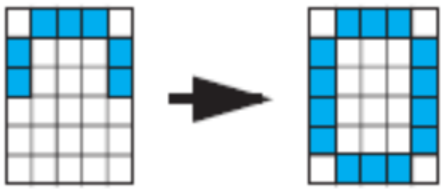
权重矩阵的更新如下

$$\Delta w_{ji}(n) = \eta\left(y_j(n)x_i(n) - y_j(n)\sum_{k=1}^{j} w_{ki}(n) y_k(n) \right)$$

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}$$

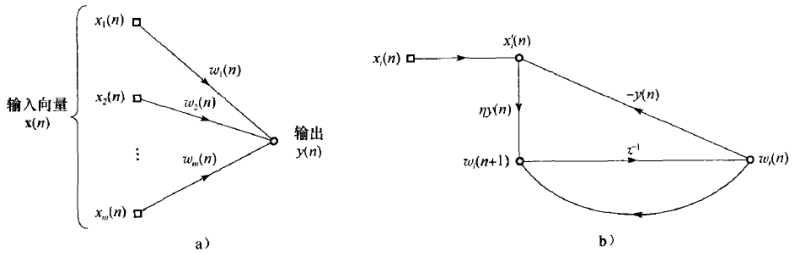
7. 噪声数字矩阵（0-2）

加入了固定噪声与随机噪声两种，测试模型鲁棒性



8. 收敛性分析

仅处理单个神经元时，不需要用双下标表示网络突触权值，如图所示



a):$y=\sum_{i=1}^{m} w_i x_i$

b):$x_i^{'}=x_i(n)\text{-}y(n)w_i(n)$，$w_i(n+1)=w_i(n)+\eta y(n)x_i^{'}(n)$

令$\mathbf{W}(n)=\{w_{ji}(n)\}$表示上图所示前馈网络的一个$l\times m$的权值矩阵，即：

$$\mathbf{W}(n)=[\mathbf{w}_1(n),\mathbf{w}_2(n),\ldots,\mathbf{w}_l(n)]^T$$

令广义 Hebb 算法的学习率参数η随着时间变化而变化啊，即$\eta(n)$，限制条件为：

$$\lim_{n\to\infty}\eta(n)=0，且\sum_{n=0}^{\infty}\eta(n)=\infty$$

可以将算法重新写成矩阵形式：

$$\Delta\mathbf{W}(n)=\eta(n)\{\mathbf{y}(n)\mathbf{x}^T(n)\text{-LT}[\mathbf{y}(n)\mathbf{y}^T(n)]\mathbf{W}(n)\}$$

其中

$$\mathbf{y}(n)=\mathbf{W}(n)\mathbf{x}(n)$$

LT[·]为下三角算子，它把矩阵对角线上方的所有元素置为 0，从而使矩阵称为下三角矩阵。

有以下定理：

如果权值矩阵$\mathbf{W}(n)$在时间步$n=0$时随机赋值，则式(3-4-2-13)所描述的广义 Hebb 算法以概率 1 收敛于固定点，且$\mathbf{W}^T(n)$趋于一个矩阵，该矩阵的列分别为$m\times 1$输入向量的$m\times m$的相关矩阵$\mathbf{R}$的前l个特征向量，按特征值的降序排列。

上述定理的实际价值在于，当对应特征值互不相同时它保证广义 Hebb 算法能够找到相关矩阵$\mathbf{R}$的前l个特征向量，同样重要的是，我们不需要计算相关矩阵$\mathbf{R}$，$\mathbf{R}$的前l个特征向量可直接由输入向量计算。特别是如果输入空间的维数m很大，而要求与$\mathbf{R}$最大的l个特征值对应的特征向量的数目只是m的一小部分，则可以节省大量计算。

收敛定理使用时变学习律参数$\eta(n)$表示的，实际上，学习率参数只能选择一个很小的固定常数η，这样才保证在η阶的突触权值的均方误差意义下收敛。

假设在极限时写成：

$$\Delta\mathbf{w}_j(n)\to 0 \quad 且 \quad \mathbf{w}_j(n)\to\mathbf{q}_j,\ n\to\infty,\ 对j=1,2,\dots,l$$

且有：

$$\left\|\mathbf{w}_j(n)\right\|=1,\ 对所有j$$

在前馈网络(图 3-4-3)中，神经元突触权值向量的极限值$\mathbf{q}_1,\mathbf{q}_2,\dots,\mathbf{q}_l$表示相关矩

阵**R**的前l个特征值对应的归一化特征向量，按特征值的降序排列，在平衡时可写成：

$$\mathbf{q}_j^T \mathbf{R} \mathbf{q}_k = \begin{cases} \lambda_j, & k = j \\ 0, & k \neq j \end{cases}$$

其中$\lambda_1 > \lambda_2 > \cdots > \lambda_l$

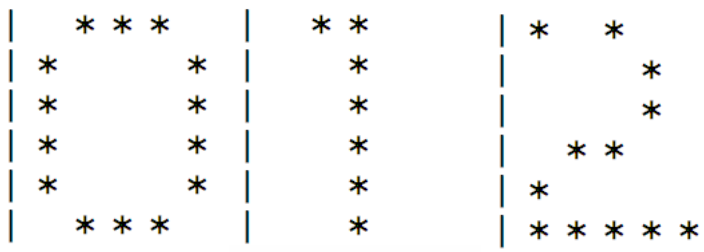
四.调试分析：

1. 在测试基本 Hebb 的方法的时候，采用 0 初始化，在之后的改进的 hebb 率中，0 初始化存在导致参数无法更新的问题，所以采用了随机初始化

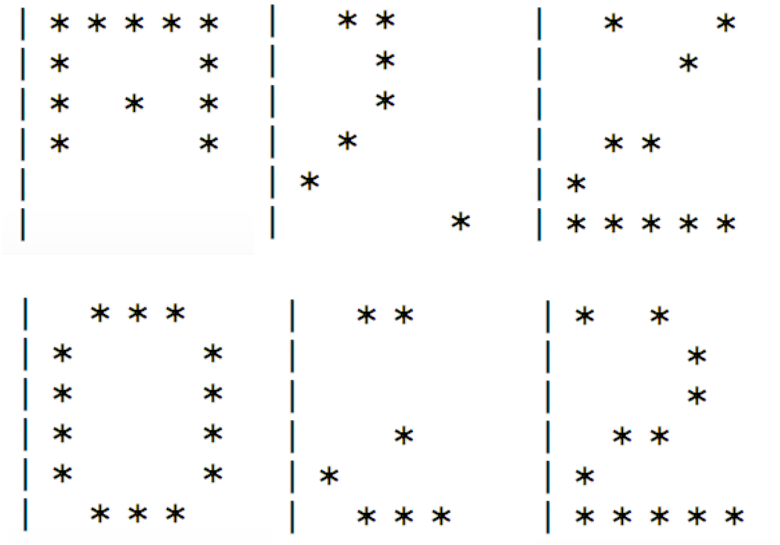
2. 在使用广义的 hebb 率时，由于权重矩阵权值较大，训练的模型的鲁棒性较差，故在后半部分基于基本的 hebb 学习率采用了阶跃函数

五.测试结果：

基本 hebb 率



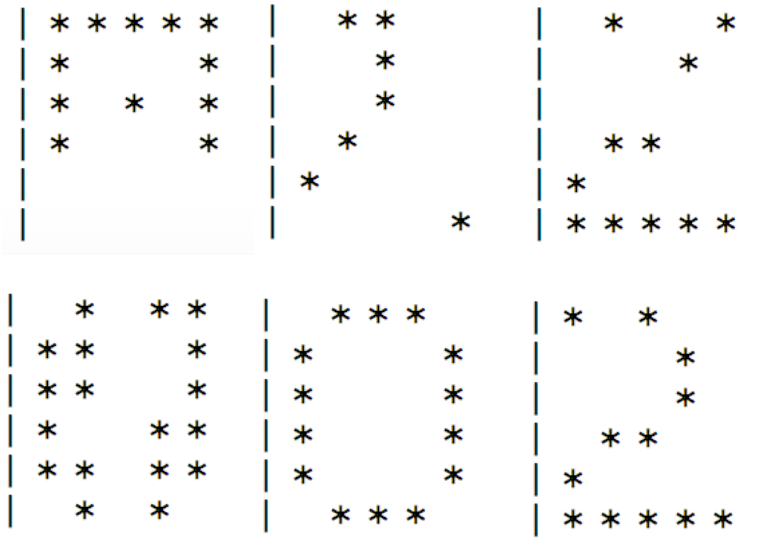
正则化 hebb 率



调整正则化参数后

可见调整一定的参数后，正则化也具备了较强的鲁棒性

广义 hebb 率



虽然广义的方法有更多的优点，但是在该问题下，受数据和参数的影响，其效果并没有显著提升，反而下降（如将 1 识别为 0，将 0 识别错误）

六.附录：

附录代码如下 文件为 hebb.py

```
import numpy as np

def draw_bin_image(image_matrix):
    for row in image_matrix.tolist():
        print('| ' + ' '.join(' *'[int((val+1)/2)] for val in row))
    print('\n')

def acti_fun(x):
    for i in range(x.size):
        if x[i] > 0:
            x[i] = 1
        else:
            x[i] = -1
    return x

class Hebb(object):
    def __init__(self, n):
        self.n = n
        self.weights = np.zeros((self.n, self.n))

    def wild_hebb(self, vec):
        raw = self.weights * vec.repeat(self.n).reshape(self.n, self.n)
        new = np.zeros_like(raw.reshape(30, 30))
        u, v = new.shape
        for i in range(u):
            for j in range(v):
                new[i, j] = np.sum(raw[0:j+1, i])
        delta_w = vec.repeat(self.n).reshape(self.n, self.n) * new
        return delta_w


    def train(self, input_vector, iter, rate, use_GEN=False, use_norm=False,
alaph = 1e-3):
        for i in range(iter):
```

```
            for vec in input_vector:
                vec = np.matrix(vec)
                delta = np.zeros((self.n, self.n))
        #这里可以使用广义 hebb 算法中的权重更新算法
                if use_GEN:
                    wild_hebb = self.wild_hebb(vec)
                    delta = np.ones_like(wild_hebb)
                    index = np.where(wild_hebb < 0)
                    delta[index] = -1
                if use_norm:
                    self.weights = self.weights + rate * (vec.getT().dot(vec) -
delta - alaph * vec.repeat(self.n).reshape(self.n, self.n) * self.weights)
                else:
                    self.weights = self.weights + rate * (vec.getT().dot(vec) -
delta)
            print(i)
        return self.weights

    def predict(self, input_vector):
        return self.weights.dot(np.matrix(input_vector).getT())

zero = [
    -1, 1, 1, 1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    -1, 1, 1, 1, -1
    ]

one = [
    -1, 1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1
    ]

two = [
    1, -1, 1, -1, -1,
    -1, -1, -1, 1, -1,
    -1, -1, -1, 1, -1,
    -1, 1, 1, -1, -1,
```

```
    1, -1, -1, -1, -1,
    1, 1, 1, 1, 1,
    ]

half_zero = [
    1, 1, 1, 1, 1,
    1, -1, -1, -1, 1,
    1, -1, 1, -1, 1,
    1, -1, -1, -1, 1,
    -1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1,
    ]

half_one = [
    -1, 1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, 1, -1, -1, -1,
    1, -1, -1, -1, -1,
    -1, -1, -1, -1, 1
    ]

half_two = [
    -1, 1, -1, -1, 1,
    -1, -1, -1, 1, -1,
    -1, -1, -1, -1, -1,
    -1, 1, 1, -1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, 1,
    ]


hebb  = Hebb(30)
w=hebb.train([zero, one, two], 300, 0.2, use_GEN=False, use_norm=True)

def predict(num, half_num):
    pre = hebb.predict(half_num)
    # print(pre.reshape((6,5)))
    draw_bin_image(acti_fun(np.array(num)).reshape((6, 5)))
    draw_bin_image(acti_fun(np.array(half_num)).reshape((6, 5)))
    draw_bin_image(acti_fun(pre).reshape((6, 5)))


predict(zero, half_zero)
```

```
predict(one, half_one)
predict(two, half_two)
```