糊糊

[NNDesign]第三篇：Hebb Rule

[读书笔记] zoey 8个月前 (02-12) 480次浏览 1个评论

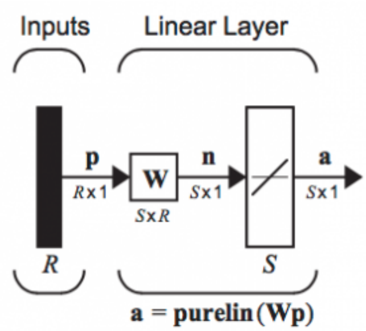
r

前言

Hebbian假设：突触前神经元向突触后神经元的持续重复的刺激，可以导致突触传递效能的增加(百度百科)。这种理论的提出是为了研究大脑记忆的运作方式，与之前的perceptron不同的是，perceptron是完全的一种machine learning，而Hebbian Learning则偏向于神经科学。书中第7章介绍了Hebb Rule和 Pseudoinverse Rule两种学习法则。

Linear Associator

书中使用Linear Associator(谷歌译名:线性关联器)来初步介绍Hebbian学习。它的结构如下：



输出向量a的表达式为：

$$\mathbf{a} = \mathbf{W}\mathbf{p}$$

或者是

$$a_i = \sum_{j=1}^{R} w_{ij} p_j$$

Linear Associator是一种名叫Associative Memory(关联储存器)的一种示例，这种结构简而言之就是输入不同，输出就不同，输入有一点微小的差异，输出也会随之改变。

Hebb Rule

如何解释Hebb的数学假设呢？如果突触两侧的两个神经元同时激活，那么突出的强度将增加。突触的强度即为权重

矩阵。用公式来表示就是：

$$w_{ij}^{new} = w_{ij}^{old} + \alpha f_i(a_{iq}) g_j(p_{jq})$$

其中，p_{jq}是第q个输入的第j个元素，a_{iq}是第q个输出的第i个元素,α是学习率，f和g是突触两侧活动的函数，权重w_{ij}的变化与活动函数的乘积成正比例。在本章中将简化等式：

$$w_{ij}^{new} = w_{ij}^{old} + \alpha a_{iq} p_{jq}$$

上式定义的Hebb规则是无监督学习规则，而在本章中使用的是用于监督学习的Hebb规则，因此，我们使用目标输出t代替实际输出a。并且为了简单起见，我们将学习速率设为1。

$$w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}$$

那么该公式可以写成向量形式：

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T$$

如果将初始权值设为0,然后将Q对输入输出应用于该公式，公式将变为：

$$\mathbf{W} = \mathbf{t}_1 \mathbf{p}_1^T + \mathbf{t}_2 \mathbf{p}_2^T + \cdots + \mathbf{t}_Q \mathbf{p}_Q^T = \sum_{q=1}^{Q} \mathbf{t}_q \mathbf{p}_q^T$$

将向量形式转换为矩阵形式就是：

$$\mathbf{W} = \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 & \ldots & \mathbf{t}_Q \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix} = \mathbf{T}\mathbf{P}^T$$

这就是Hebb Rule公式的简化形式。

Hebb Rule性能分析

首先，如果输入向量是正交的(正交并且单位长度)，那么可以计算输出为：

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \left(\sum_{q=1}^{Q} \mathbf{t}_q \mathbf{p}_q^T \right) \mathbf{p}_k = \sum_{q=1}^{Q} \mathbf{t}_q (\mathbf{p}_q^T \mathbf{p}_k)$$

既然p_q是正交的，那么：

$p^T{}_q p_k$=1, q=k;
$p^T{}_q p_k$=0, q≠k.

因此，

$a=Wp_k=t_k$

这说明如果输入原型矢量是正交的，那么Hebb规则将产生正确的输出。如果不是正交的，则会产生一个错误项：

$$\mathbf{a} = \mathbf{W}\mathbf{p}_k = \mathbf{t}_k + \overbrace{\boxed{\sum_{q \neq k} \mathbf{t}_q (\mathbf{p}_q^T \mathbf{p}_k)}}^{Error}.$$

举个栗子，假设有两对输入输出如下(p1与p2是正交的)

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}\right\} \quad \left\{\mathbf{p}_2 = \begin{bmatrix} 0.5 \\ 0.5 \\ -0.5 \\ -0.5 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right\}$$

根据公式来计算权值矩阵：

$$\mathbf{W} = \mathbf{T}\mathbf{P}^T = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 & 0.5 & -0.5 \\ 0.5 & 0.5 & -0.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}$$

然后，使用刚刚的p1 p2来测试该矩阵，发现目标输出与实际输出是对等的。

$$\mathbf{W}\mathbf{p}_1 = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad \mathbf{W}\mathbf{p}_2 = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.5 \\ -0.5 \\ -0.5 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

可以自己测试一下，如果输入向量不是正交的，结果或许看起来相近，但是跟目标并不是完全匹配的。

Pseudoinverse Rule

Pseudoinverse Rule翻译为伪逆规则，适用于当原型输入不正交时来减少错误。回忆一下linear associator，它的任务是为每一个p_q产生一个输出t_q 即 $\mathbf{Wp_q=t_q}$. 如果该方程不能够被精确的满足，那么我们希望他们近似被满足。一种方案是将权重矩阵最小化以满足以下性能指标最小化：

$$F(\mathbf{W}) = \sum_{q=1}^{Q} \left\|\mathbf{t}_q - \mathbf{W}\mathbf{p}_q\right\|^2$$

即 $\mathbf{W=TP^{-1}}$。然而该式一般不可能被满足，因为**P**一般不会是方阵(输入向量的个数一般比输入向量的维度小得多)，所以**P**可能不存在逆矩阵。那么解决方案是使用Moore-Penrose伪逆矩阵：

$\mathbf{W=TP^+}$

伪逆矩阵的性质有：

$\mathbf{PP^+P = P}$

$\mathbf{P^+PP^+ = P^+}$

$\mathbf{P^+P} = (\mathbf{P^+P})^T$

$\mathbf{PP^+} = (\mathbf{PP^+})^T$

求伪逆矩阵的公式为：$\mathbf{P^+} = (\mathbf{P^TP})^{-1}\mathbf{P^T}$

举个栗子：给定两组输入输出如下

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\-1\\-1\end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix}-1\end{bmatrix}\right\} \qquad \left\{\mathbf{p}_2 = \begin{bmatrix}1\\1\\-1\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}1\end{bmatrix}\right\}$$

计算出伪逆矩阵：

$$\mathbf{P}^+ = (\mathbf{P}^T\mathbf{P})^{-1}\mathbf{P}^T = \begin{bmatrix}3 & 1\\1 & 3\end{bmatrix}^{-1}\begin{bmatrix}1 & -1 & -1\\1 & 1 & -1\end{bmatrix} = \begin{bmatrix}0.25 & -0.5 & -0.25\\0.25 & 0.5 & -0.25\end{bmatrix}$$

计算出权重矩阵：

$$\mathbf{W} = \mathbf{TP}^+ = \begin{bmatrix}-1 & 1\end{bmatrix}\begin{bmatrix}0.25 & -0.5 & -0.25\\0.25 & 0.5 & -0.25\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\end{bmatrix}$$

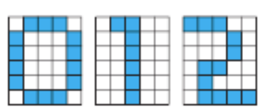
验证结果正确：

$$\mathbf{Wp}_1 = \begin{bmatrix}0 & 1 & 0\end{bmatrix}\begin{bmatrix}1\\-1\\-1\end{bmatrix} = \begin{bmatrix}-1\end{bmatrix}$$

$$\mathbf{Wp}_2 = \begin{bmatrix}0 & 1 & 0\end{bmatrix}\begin{bmatrix}1\\1\\-1\end{bmatrix} = \begin{bmatrix}1\end{bmatrix}$$

实际应用

下面来看一个模式识别的具体应用。给定一组图案，使用autoassociative memory(自动关联存储器)来存储一组模式，然后根据不同的输入输出对应的结果，即使提供的是有损输入，也期望输出最相近的结果。

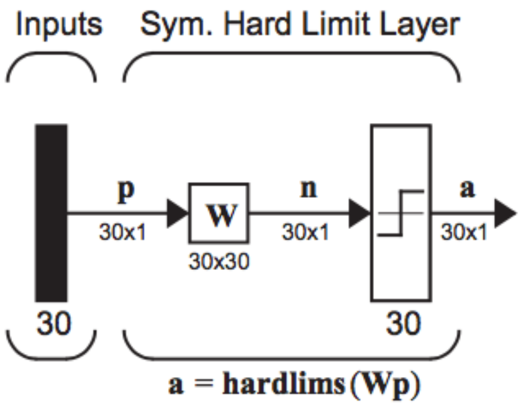


$\mathbf{p}_1,\mathbf{t}_1$　$\mathbf{p}_2,\mathbf{t}_2$　$\mathbf{p}_3,\mathbf{t}_3$

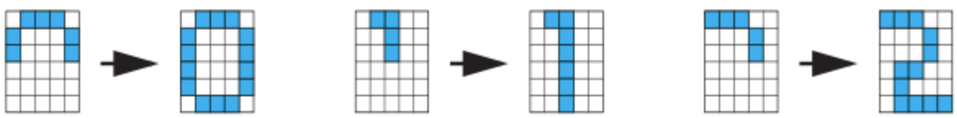
将输入输出写成向量的形式，例如$\mathbf{p}^1$=[-1 1 1 1 1 -1 1 -1 -1 -1 -1 1 1 -1 … 1 -1]^T，然后根据公式计算出权重矩阵：

$$\mathbf{W} = \mathbf{p}_1\mathbf{p}_1^T + \mathbf{p}_2\mathbf{p}_2^T + \mathbf{p}_3\mathbf{p}_3^T$$

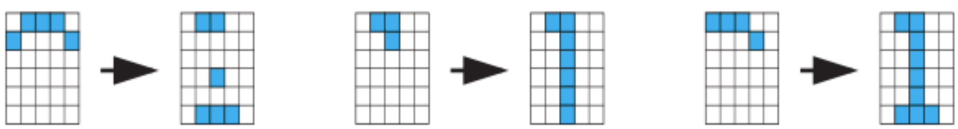
由于输入向量的元素只有两个取值：1和-1，因此我们需要修改激活函数，使输出只有两种取值，即使用hardlims函数。



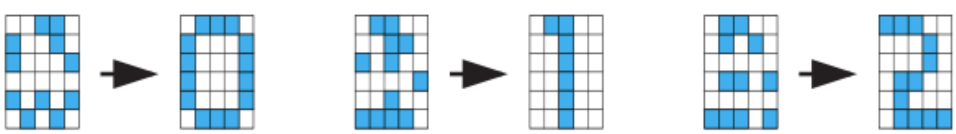
然后我们可以验证在输入向量有不同程度损坏的情况下输出的结果。当损坏程度为50%时，输出结果完全正确：



当损坏程度为2/3时，只有数字1被恢复正确：



当输入模型有噪声时，所有的输出也都是正确的。



关于这个应用，我找到了一份python代码，略作修改，使用python3可编译运行。出处为：http://blog.csdn.net/navylq/article/details/52426034

```
# -*- coding: utf-8 -*-
import numpy as np
mat0 = np.matrix([-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])
mat1 = np.matrix([-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])
mat2 = np.matrix([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])

mat0t = mat0.getT()
mat0p = mat0t.dot(mat0)
mat1t = mat1.getT()
mat1p = mat1t.dot(mat1)
mat2t = mat2.getT()
mat2p = mat2t.dot(mat2)
print("===============matrix 0====================")
print(mat0p)
print ("===============matrix 1====================")
print(mat1p)
print( "===============matrix 2====================")
print(mat2p)
matw = mat0p+mat1p+mat2p
print ("===============matrix sum====================")
print(matw)
testa0 = np.matrix([-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])
mata0 = matw.dot(testa0.getT())
print("========== raw mata0 ==============")
print (mata0)
for ii in range(mata0.size):
    if mata0[ii] > 0:
        mata0[ii] = 1
    else:
        mata0[ii] = -1
print("============ After testa0 =================")
print(mata0)
```

喜欢 (0) 分享 (0)

« [NNDesign]第二篇：感知器与学习规则

MacOS安装Intel BigDL需要注意的问题 »

[NNDesign]第二篇：感知器与学习规则

[NNDesign]第一篇:基本概念及标注符号

- [NNDesign]第二篇：感知器与学习规则
- [NNDesign]第一篇:基本概念及标注符号

.声明

和资源来自糊糊
我或引用本站文章
办议。如果有侵犯版权的资
系我，我会在24h内删
资源。

友情链接

龙猪 | i Huhu

静觅 | 静静寻觅生活的美好

联系方式

邮箱：646689983@qq.com

关于我

欢迎志趣相投的小伙伴一起来

Theme By 云落