[BP神经网络与Python实现](http://www.cnblogs.com/Finley/p/5946000.html)

人工神经网络是一种经典的机器学习模型，随着深度学习的发展神经网络模型日益完善.

联想大家熟悉的回归问题， 神经网络模型实际上是根据训练样本创造出一个多维输入多维输出的函数， 并使用该函数进行预测， 网络的训练过程即为调节该函数参数提高预测精度的过程.神经网络要解决的问题与最小二乘法回归解决的问题并无根本性区别.

回归和分类是常用神经网络处理的两类问题， 如果你已经了解了神经网络的工作原理可以在<http://playground.tensorflow.org/>上体验一个浅层神经网络的工作过程.

感知机(Perceptron)是一个简单的线性二分类器， 它保存着输入权重， 根据输入和内置的函数计算输出.人工神经网络中的单个神经元即是感知机.

在前馈神经网络的预测过程中， 数据流从输入到输出单向流动， 不存在循环和返回的通道.

目前大多数神经网络模型都属于前馈神经网络， 在下文中我们将详细讨论前馈过程.

多层感知机(Multi Layer Perceptron， MLP)是由多个感知机层全连接组成的前馈神经网络， 这种模型在非线性问题中表现出色.

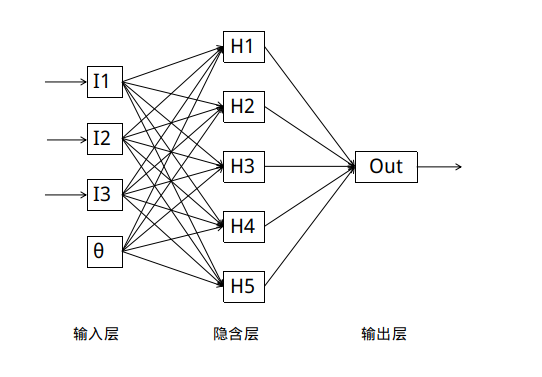
所谓全连接是指层A上任一神经元与临近层B上的任意神经元之间都存在连接.

反向传播(Back Propagation，BP)是误差反向传播的简称，这是一种用来训练人工神经网络的常见算法， 通常与最优化方法(如梯度下降法)结合使用.

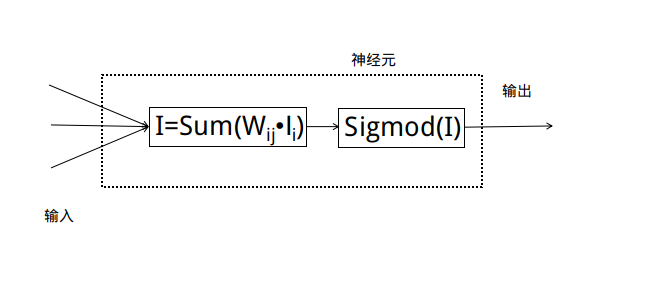
本文介绍的神经网络模型在结构上属于MLP， 因为采用BP算法进行训练， 人们也称其为BP神经网络.

**BP神经网络原理**

经典的BP神经网络通常由三层组成: 输入层， 隐含层与输出层.通常输入层神经元的个数与特征数相关，输出层的个数与类别数相同， 隐含层的层数与神经元数均可以自定义.



每个神经元代表对数据的一次处理:



每个隐含层和输出层神经元输出与输入的函数关系为:

Ij=∑iWijOiIj=∑iWijOi

Oj=sigmod(Il)=11+e−IlOj=sigmod(Il)=11+e−Il

其中WijWij表示神经元i与神经元j之间连接的权重，OjOj代表神经元j的输出， sigmod是一个特殊的函数用于将任意实数映射到(0，1)区间.

上文中的sigmod函数称为神经元的激励函数(activation function)， 除了sigmod函数11+e−Il11+e−Il外， 常用还有tanh和ReLU函数.

我们用一个完成训练的神经网络处理回归问题， 每个样本拥有n个输入.相应地，神经网络拥有n个输入神经元和1个输出神经元.

实际应用中我们通常在输入层额外增加一个偏置神经元， 提供一个可控的输入修正;或者为每个隐含层神经元设置一个偏置参数.

我们将n个特征依次送入输入神经元， 隐含层神经元获得输入层的输出并计算自己输出值， 输出层的神经元根据隐含层输出计算出回归值.

上述过程一般称为前馈(Feed-Forward)过程， 该过程中神经网络的输入输出与多维函数无异.

现在我们的问题是如何训练这个神经网络.

作为监督学习算法，BP神经网络的训练过程即是根据前馈得到的预测值和参考值比较， 根据误差调整连接权重WijWij的过程.

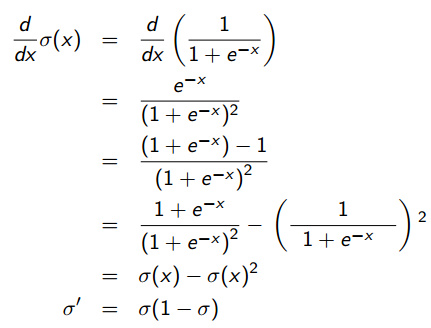
训练过程称为反向传播过程(BackPropagation)， 数据流正好与前馈过程相反.

首先我们随机初始化连接权重WijWij， 对某一训练样本进行一次前馈过程得到各神经元的输出.

首先计算输出层的误差:

Ej=sigmod′(Oj)∗(Tj−Oj)=Oj(1−Oj)(Tj−Oj)Ej=sigmod′(Oj)∗(Tj−Oj)=Oj(1−Oj)(Tj−Oj)

其中EjEj代表神经元j的误差，OjOj表示神经元j的输出， TjTj表示当前训练样本的参考输出， sigmod′(x)sigmod′(x)是上文sigmod函数的一阶导数.



计算隐含层误差:

Ej=sigmod′(Oj)∗∑kEkWjk=Oj(1−Oj)∑kEkWjkEj=sigmod′(Oj)∗∑kEkWjk=Oj(1−Oj)∑kEkWjk

隐含层输出不存在参考值， 使用下一层误差的加权和代替(Tj−Oj)(Tj−Oj).

计算完误差后就可以更新WijWij和θjθj:

Wij=Wij+λEjOiWij=Wij+λEjOi

其中λλ是一个称为学习率的参数，一般在(0，0.1)区间上取值.

实际上为了加快学习的效率我们引入称为矫正矩阵的机制， 矫正矩阵记录上一次反向传播过程中的EjOiEjOi值， 这样WjWj更新公式变为:

Wij=Wij+λEjOi+μCijWij=Wij+λEjOi+μCij

μμ是一个称为矫正率的参数.随后更新矫正矩阵:

Cij=EjOiCij=EjOi

每一个训练样本都会更新一次整个网络的参数.我们需要额外设置训练终止的条件.

最简单的训练终止条件为设置最大迭代次数， 如将数据集迭代1000次后终止训练.

单纯的设置最大迭代次数不能保证训练结果的精确度， 更好的办法是使用损失函数(loss function)作为终止训练的依据.

损失函数可以选用输出层各节点的方差:

L=∑j(Tj−Oj)2L=∑j(Tj−Oj)2

为了避免神经网络进行无意义的迭代， 我们通常在训练数据集中抽出一部分用作校验.当预测误差高于阈值时提前终止训练.

**Python实现BP神经网络**

首先实现几个工具函数:

def rand(a， b):

return (b - a) \* random.random() + a

def make\_matrix(m， n， fill=0.0): # 创造一个指定大小的矩阵

mat = []

for i in range(m):

mat.append([fill] \* n)

return mat

定义sigmod函数和它的导数:

def sigmoid(x):

return 1.0 / (1.0 + math.exp(-x))

def sigmod\_derivate(x):

return x \* (1 - x)

定义BPNeuralNetwork类， 使用三个列表维护输入层，隐含层和输出层神经元， 列表中的元素代表对应神经元当前的输出值.使用两个二维列表以邻接矩阵的形式维护输入层与隐含层， 隐含层与输出层之间的连接权值， 通过同样的形式保存矫正矩阵.

定义setup方法初始化神经网络:

def setup(self， ni， nh， no):

self.input\_n = ni + 1

self.hidden\_n = nh

self.output\_n = no

# init cells

self.input\_cells = [1.0] \* self.input\_n

self.hidden\_cells = [1.0] \* self.hidden\_n

self.output\_cells = [1.0] \* self.output\_n

# init weights

self.input\_weights = make\_matrix(self.input\_n， self.hidden\_n)

self.output\_weights = make\_matrix(self.hidden\_n， self.output\_n)

# random activate

for i in range(self.input\_n):

for h in range(self.hidden\_n):

self.input\_weights[i][h] = rand(-0.2， 0.2)

for h in range(self.hidden\_n):

for o in range(self.output\_n):

self.output\_weights[h][o] = rand(-2.0， 2.0)

# init correction matrix

self.input\_correction = make\_matrix(self.input\_n， self.hidden\_n)

self.output\_correction = make\_matrix(self.hidden\_n， self.output\_n)

定义predict方法进行一次前馈， 并返回输出:

def predict(self， inputs):

# activate input layer

for i in range(self.input\_n - 1):

self.input\_cells[i] = inputs[i]

# activate hidden layer

for j in range(self.hidden\_n):

total = 0.0

for i in range(self.input\_n):

total += self.input\_cells[i] \* self.input\_weights[i][j]

self.hidden\_cells[j] = sigmoid(total)

# activate output layer

for k in range(self.output\_n):

total = 0.0

for j in range(self.hidden\_n):

total += self.hidden\_cells[j] \* self.output\_weights[j][k]

self.output\_cells[k] = sigmoid(total)

return self.output\_cells[:]

定义back\_propagate方法定义一次反向传播和更新权值的过程， 并返回最终预测误差:

def back\_propagate(self， case， label， learn， correct):

# feed forward

self.predict(case)

# get output layer error

output\_deltas = [0.0] \* self.output\_n

for o in range(self.output\_n):

error = label[o] - self.output\_cells[o]

output\_deltas[o] = sigmod\_derivate(self.output\_cells[o]) \* error

# get hidden layer error

hidden\_deltas = [0.0] \* self.hidden\_n

for h in range(self.hidden\_n):

error = 0.0

for o in range(self.output\_n):

error += output\_deltas[o] \* self.output\_weights[h][o]

hidden\_deltas[h] = sigmod\_derivate(self.hidden\_cells[h]) \* error

# update output weights

for h in range(self.hidden\_n):

for o in range(self.output\_n):

change = output\_deltas[o] \* self.hidden\_cells[h]

self.output\_weights[h][o] += learn \* change + correct \* self.output\_correction[h][o]

self.output\_correction[h][o] = change

# update input weights

for i in range(self.input\_n):

for h in range(self.hidden\_n):

change = hidden\_deltas[h] \* self.input\_cells[i]

self.input\_weights[i][h] += learn \* change + correct \* self.input\_correction[i][h]

self.input\_correction[i][h] = change

# get global error

error = 0.0

for o in range(len(label)):

error += 0.5 \* (label[o] - self.output\_cells[o]) \*\* 2

return error

定义train方法控制迭代， 该方法可以修改最大迭代次数， 学习率λλ， 矫正率μμ三个参数.

def train(self， cases， labels， limit=10000， learn=0.05， correct=0.1):

for i in range(limit):

error = 0.0

for i in range(len(cases)):

label = labels[i]

case = cases[i]

error += self.back\_propagate(case， label， learn， correct)

编写test方法，演示如何使用神经网络学习异或逻辑:

def test(self):

cases = [

[0， 0]，

[0， 1]，

[1， 0]，

[1， 1]，

]

labels = [[0]， [1]， [1]， [0]]

self.setup(2， 5， 1)

self.train(cases， labels， 10000， 0.05， 0.1)

for case in cases:

print(self.predict(case))

完整源代码参见[bpnn.py](http://git.oschina.net/hdt3213/ClassifiersDemo/blob/master/BPNN/bpnn.py)

使用tensorflow实现一个神经网络可能是更简单高效的方法， 可以参见[tensorflow入门指南](http://www.cnblogs.com/Finley/p/5953122.html)中的第二节:*实现一个简单神经网络*.