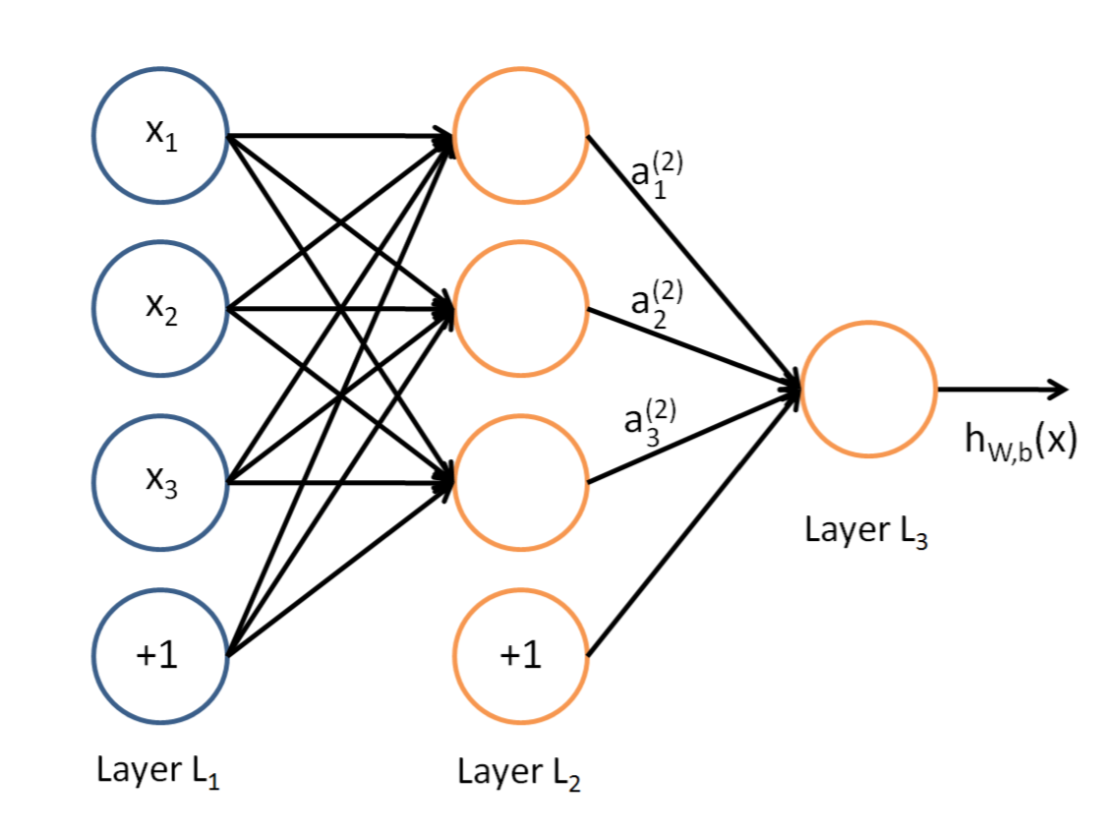
**实验六 BP反向传播算法**

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，是一种监督学习算法，通常与最优化方法(如梯度下降法)结合使用。本实验将借助一个BP神经网络来介绍BP算法。

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。通常输入层神经元个数与特征数相关，输出层神经元个数和分类类别个数相等，隐含层的层数和神经元个数由设计者定义。下图是一个很经典的神经网络结构：



图中的Layer L1是输入层，Layer L2是隐含层，Layer L3是输出层。在输入层和隐含层的最后还加了一个值为1的神经元，这代表着偏置值biase，同一层的神经元共用一个偏置值。隐含层和输出层每一个神经元可以这么计算：

其中，表示神经元i和神经元j之间连接的权重，表示神经元i的输出，sigmoid函数称为神经元的激励函数(activation function)，用于将任意实数映射到(0,1)区间。它的公式为：

它的一阶导数为：

除此之外常用的激励函数还有tanh和ReLU，大家可以自行了解。

BP算法包含前馈过程(Feed-Forward)和反向传播过程(Back-Propagation)这两个部分。前馈过程是指从输入层开始，计算隐含层各个神经元的输出，进而计算输出层各个神经元的输出。而反向传播过程中，将输出层的输出与期望值进行比较，将得到的误差沿着前馈过程的反方向进行传播，更新神经元之间连接的权重和偏置值。

BP算法依赖于链式求导法则(chain rule)：

假设有函数u(y(x))，则有。

下面我们直接给出各层误差计算公式及权重和偏置值的更新公式，具体的推导过程可以参考[这个链接](https://www.jianshu.com/p/679e390f24bb)。

输出层误差：

其中代表神经元j的误差，表示神经元j的输出，表示当前训练样本的期望输出，表示sigmoid函数的一阶导数。

隐含层误差：

其中是下一层的神经元k的误差，表示神经元j和神经元k之间连接的权重。隐含层使用下一层误差的加权和代替。

当把偏置值看作是值为1的神经元与下一层神经元之间连接的权重时，我们可以使用下面的公式一起更新权重和偏置值：

其中是学习率，可以理解为更新的步长。

**实验要求：**

实现BP算法，使其能学习异或逻辑。

**实验步骤：**

新建文件，命名为bp.py，添加如下代码：

#-\*-coding:utf-8-\*-

import math

import random

random.seed(0)

①首先添加几个工具函数。

def rand(a,b):

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

def make\_matrix(m,n,fill=0.0):

# 创建一个指定大小的矩阵，使用fill值填充

mat = []

for i in range(m):

mat.append([fill]\*n)

return mat

②实现激活函数sigmoid及其一阶导数。公式见前面。

def sigmoid(x):

# e的n次方可以这样写：math.exp(n)

# 记得return语句，记得使用1.0而不是1

#**【代码待补全】**

def sigmoid\_deriviate(x):

# 注意这里的参数x已经是sigmoid函数的输出

#**【代码待补全】**

③定义BPNN类。在类中实现BP算法的前馈、反向传播以及参数的更新。

class BPNN:

# 注意每边都是两个下划线

# 初始化一些变量

def \_\_init\_\_(self):

# 下面三个变量保存各层神经元个数

self**.**input\_n **=** 0

self**.**hidden\_n **=** 0

self**.**output\_n **=** 0

# 下面三个变量保存各层神经元的输出值

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

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

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

# 下面两个变量保存权重

self**.**input\_weights **=** []

self**.**output\_weights **=** []

# 初始化神经网络

def setup(self,ni,nh,no):

# ni,nh,no分别代表输入层、隐含层、输出层

# 的神经元个数

# 注意这里只对隐含层的神经元增加偏置值

# 处理方式是在输入层增加一个神经元，使

# 其统一到权重之中。

self.input\_n = ni + 1

self.hidden\_n = nh

self.output\_n = no

# 初始化每一层神经元的值为1

# 之所以设为1是为了将偏置值看作是输入恒为1的

# 神经元与其它神经元的连接的权重

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

# 仿照上面的代码初始化隐含层和输出层的神经元

#**【代码待补全】**

# 初始化权重

self.input\_weights = \

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

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

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

self.input\_weights[i][h]=\

rand(-2.0,2.0)

# 仿照上面的代码初始化隐含层到输出层的权重

#**【代码待补全】**

# 编写predict函数进行一次前馈，返回输出

def predict(self,inputs):

# 根据用户提供的输入填充输入层的数据

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

#【思考】这里为什么要减一？

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

# 根据输入层的数据前向计算隐含层的值

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

total = 0.0

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

# 计算W\*a

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

self.input\_weights[i][h]

# 调用激活函数

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

# 根据隐含层的数据前向计算输出层的值

# 仿照前面的代码完成这一步骤

#**【代码待补全】**

# 返回网络输出

return self.output\_cells[:]

#执行一次反向传播和权值更新，并返回预测的误差

def back\_propagate(self,case,label,learning\_rate):

# case和label分别是训练数据及期望输出

# learning\_rate为学习率

# 前向传播填充各层数据

self.predict(case)

# 获取输出层误差

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

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

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

output\_error[o] = error \* \

sigmoid\_deriviate(self.output\_cells[o])

# 获取隐含层误差

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

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

error = 0.0

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

# 根据公式计算error

#**【代码待补全】**

hidden\_error[h] = error \* \

sigmoid\_deriviate(self.hidden\_cells[h])

# 更新隐含层到输出层权值

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

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

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

self.output\_weights[h][o] += \

learning\_rate \* change

# 更新输入层到隐含层权值

# 仿照前面的代码以及权值更新公式完成代码

#**【代码待补全】**

# 返回全局误差

# 全局误差公式为

#

global\_error = 0.0

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

global += 0.5\*(label[o] - \

self.output\_cells[o])\*\*2

return global\_error

# 定义训练函数

def train(self,cases,labels,limit=10000,lr=0.1):

# limit代表迭代次数，lr是学习率

for i in range(limit):

error = 0.0

for j in range(len(cases)):

label = labels[j]

case = cases[j]

# 调用反向传播函数，累加误差

# 当迭代次数是1000的倍数时打印误差

#**【代码待补全】**

# 定义测试函数，目的是学习异或逻辑

def test(self):

# 训练数据

cases = [[0,0],

[0,1],

[1,0],

[1,1]]

labels = [[0],[1],[1],[0]]

self.setup(2,5,1)

self.train(cases,labels)

for case in cases:

print(self.predict(case))

④添加如下代码到文件末尾

if \_\_name\_\_==’\_\_main\_\_’:

nn = BPNN()

nn.test()

直接运行该代码文件即可查看结果。