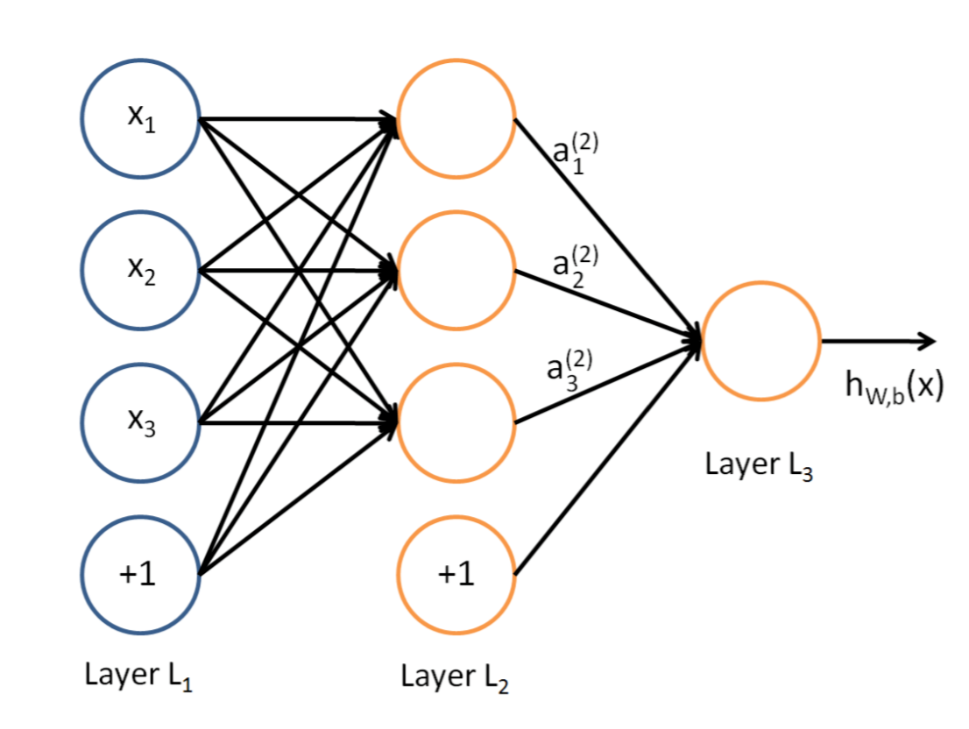
[**一文弄懂神经网络中的反向传播法——BackPropagation**](http://www.cnblogs.com/charlotte77/p/5629865.html)

<http://www.cnblogs.com/charlotte77/p/5629865.html>

　　最近在看深度学习的东西，一开始看的吴恩达的UFLDL教程，有中文版就直接看了，后来发现有些地方总是不是很明确，又去看英文版，然后又找了些资料看，才发现，中文版的译者在翻译的时候会对省略的公式推导过程进行补充，但是补充的又是错的，难怪觉得有问题。反向传播法其实是神经网络的基础了，但是很多人在学的时候总是会遇到一些问题，或者看到大篇的公式觉得好像很难就退缩了，其实不难，就是一个链式求导法则反复用。如果不想看公式，可以直接把数值带进去，实际的计算一下，体会一下这个过程之后再来推导公式，这样就会觉得很容易了。

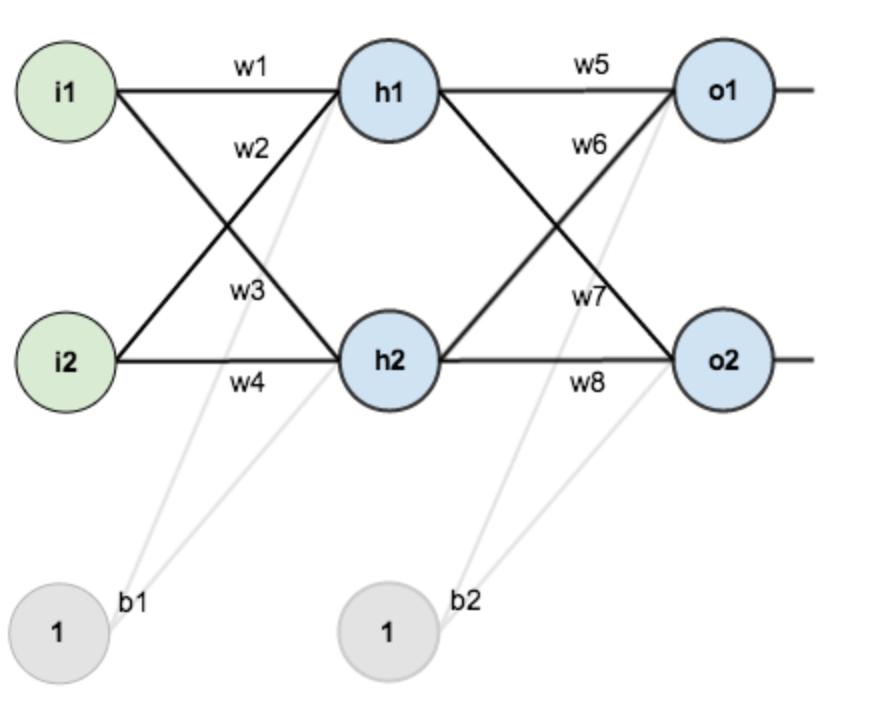
说到神经网络，大家看到这个图应该不陌生：



　　这是典型的三层神经网络的基本构成，Layer L1是输入层，Layer L2是隐含层，Layer L3是隐含层，我们现在手里有一堆数据{x1,x2,x3,...,xn},输出也是一堆数据{y1,y2,y3,...,yn},现在要他们在隐含层做某种变换，让你把数据灌进去后得到你期望的输出。如果你希望你的输出和原始输入一样，那么就是最常见的自编码模型（Auto-Encoder）。可能有人会问，为什么要输入输出都一样呢？有什么用啊？其实应用挺广的，在图像识别，文本分类等等都会用到，我会专门再写一篇Auto-Encoder的文章来说明，包括一些变种之类的。如果你的输出和原始输入不一样，那么就是很常见的人工神经网络了，相当于让原始数据通过一个映射来得到我们想要的输出数据，也就是我们今天要讲的话题。

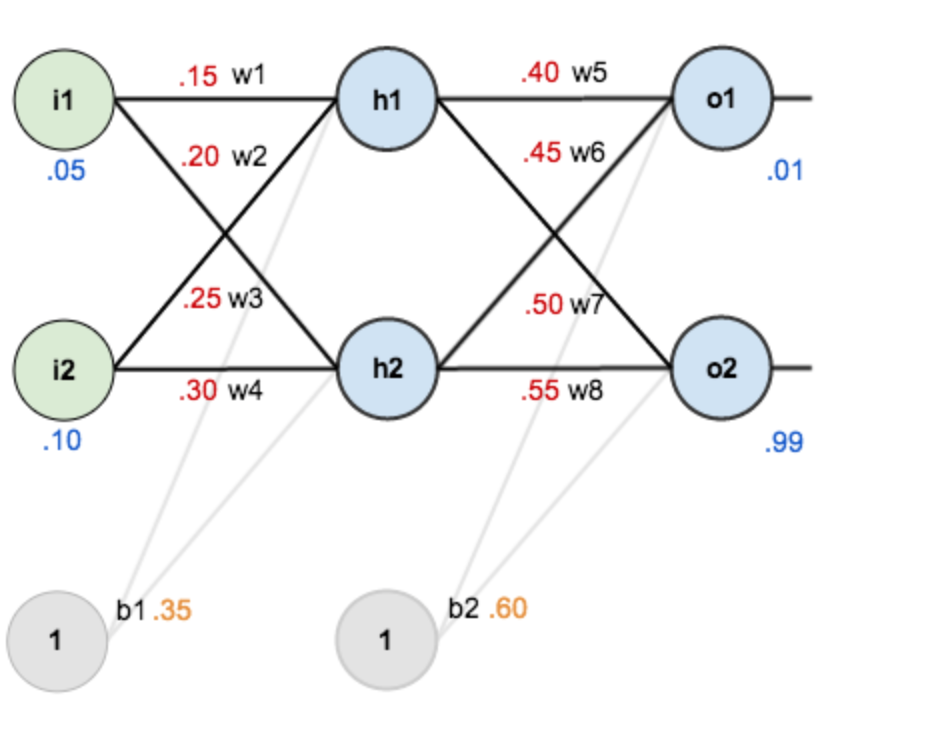
本文直接举一个例子，带入数值演示反向传播法的过程，公式的推导等到下次写Auto-Encoder的时候再写，其实也很简单，感兴趣的同学可以自己推导下试试：）（注：本文假设你已经懂得基本的神经网络构成，如果完全不懂，可以参考Poll写的笔记：[[Mechine Learning & Algorithm] 神经网络基础](http://www.cnblogs.com/maybe2030/p/5597716.html)）

　　假设，你有这样一个网络层：



　　第一层是输入层，包含两个神经元i1，i2，和截距项b1；第二层是隐含层，包含两个神经元h1,h2和截距项b2，第三层是输出o1,o2，每条线上标的wi是层与层之间连接的权重，激活函数我们默认为sigmoid函数。

　　现在对他们赋上初值，如下图：



　　其中，输入数据  i1=0.05，i2=0.10;

　　　　　输出数据 o1=0.01,o2=0.99;

　　　　　初始权重  w1=0.15,w2=0.20,w3=0.25,w4=0.30;

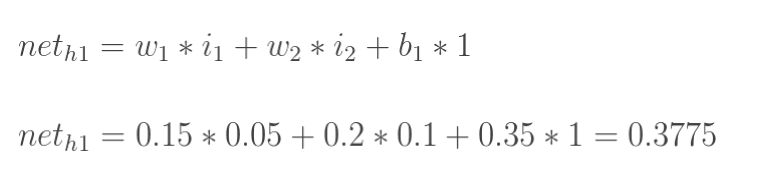
　　　　　　　　　  w5=0.40,w6=0.45,w7=0.50,w8=0.55

　　目标：给出输入数据i1,i2(0.05和0.10)，使输出尽可能与原始输出o1,o2(0.01和0.99)接近。

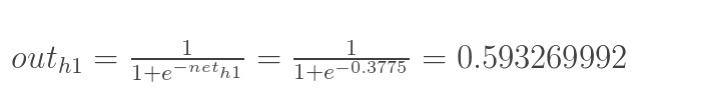
**Step 1 前向传播**

　　1.输入层---->隐含层：

　　计算神经元h1的输入加权和：



神经元h1的输出o1:(此处用到激活函数为sigmoid函数)：

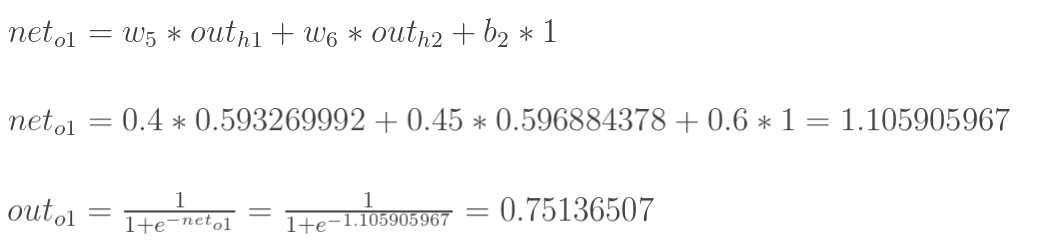


　　同理，可计算出神经元h2的输出o2：

https://images2015.cnblogs.com/blog/853467/201606/853467-20160630150244265-1128303244.png

　　2.隐含层---->输出层：

　　计算输出层神经元o1和o2的值：





这样前向传播的过程就结束了，我们得到输出值为[0.75136079 , 0.772928465]，与实际值[0.01 , 0.99]相差还很远，现在我们对误差进行反向传播，更新权值，重新计算输出。

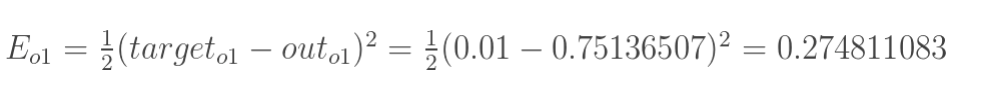
**Step 2 反向传播**

1.计算总误差

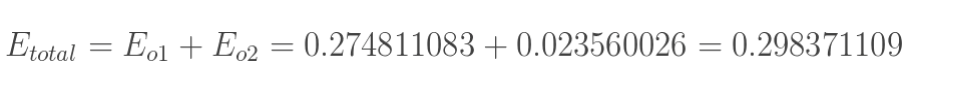
总误差：(square error)



但是有两个输出，所以分别计算o1和o2的误差，总误差为两者之和：

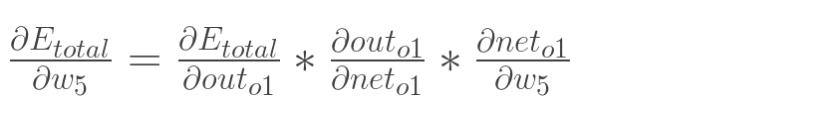




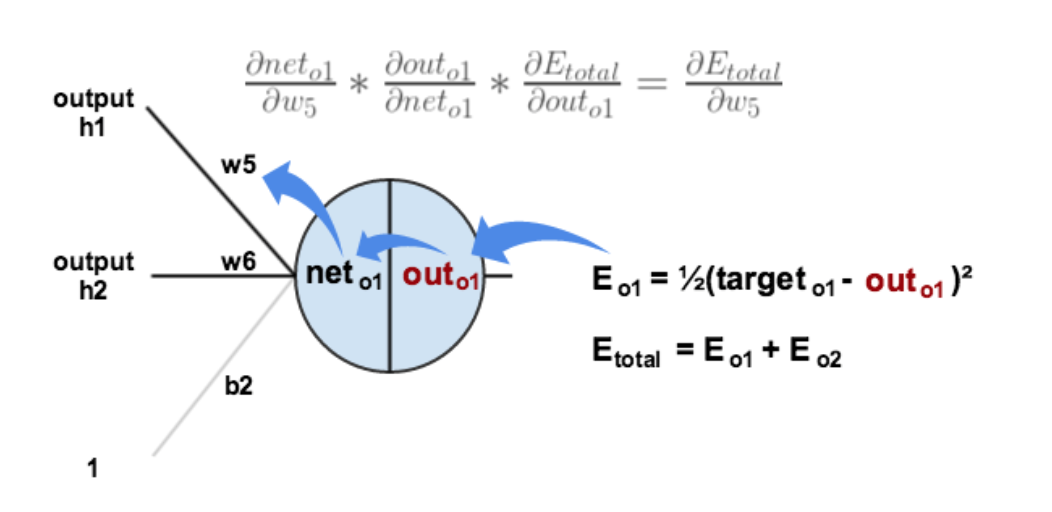


2.隐含层---->输出层的权值更新：

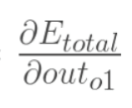
以权重参数w5为例，如果我们想知道w5对整体误差产生了多少影响，可以用整体误差对w5求偏导求出：（链式法则）

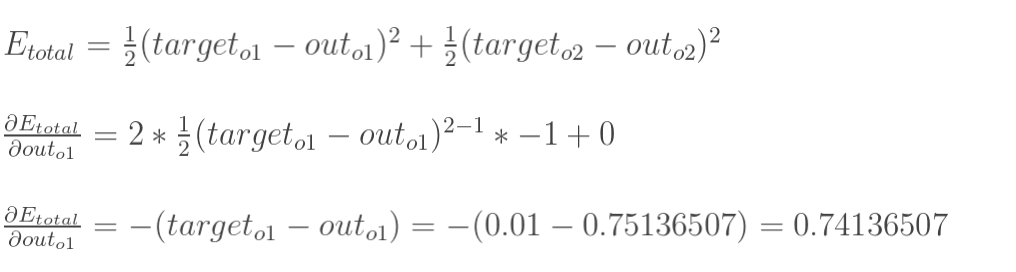


下面的图可以更直观的看清楚误差是怎样反向传播的：

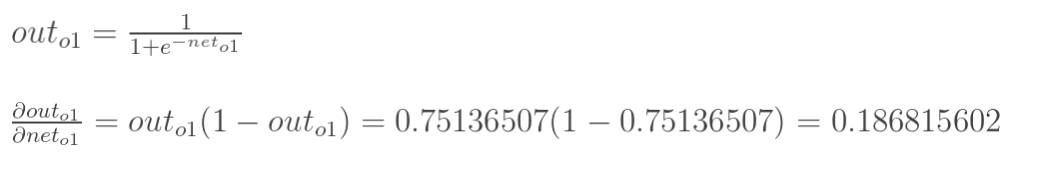


现在我们来分别计算每个式子的值：

计算：

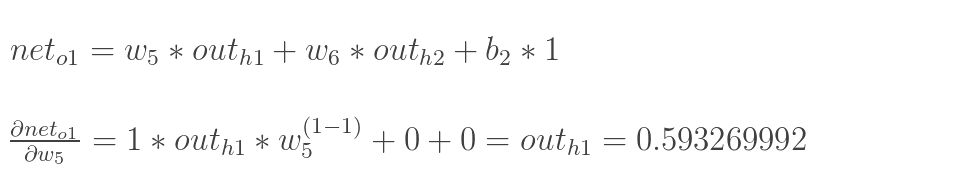


计算：

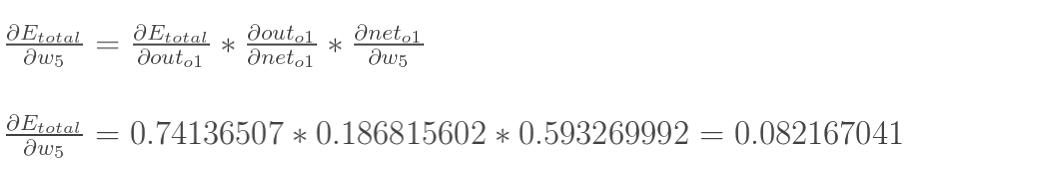


（这一步实际上就是对sigmoid函数求导，比较简单，可以自己推导一下）

计算：

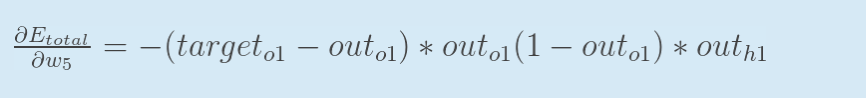


最后三者相乘：

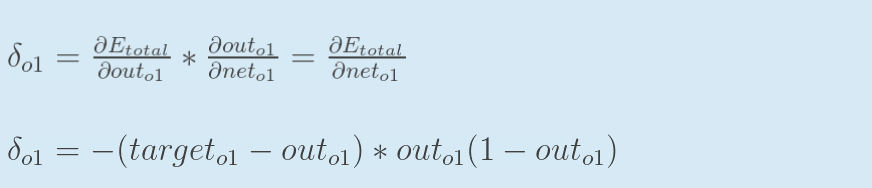


这样我们就计算出整体误差E(total)对w5的偏导值。

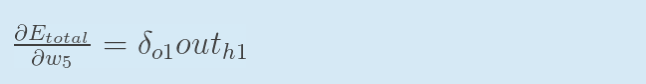
回过头来再看看上面的公式，我们发现：



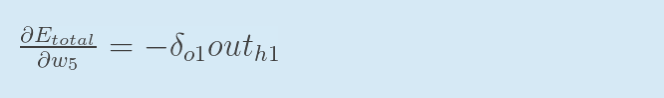
为了表达方便，用https://images2015.cnblogs.com/blog/853467/201606/853467-20160630153202812-585186566.png来表示输出层的误差：



因此，整体误差E(total)对w5的偏导公式可以写成：



如果输出层误差计为负的话，也可以写成：



最后我们来更新w5的值：



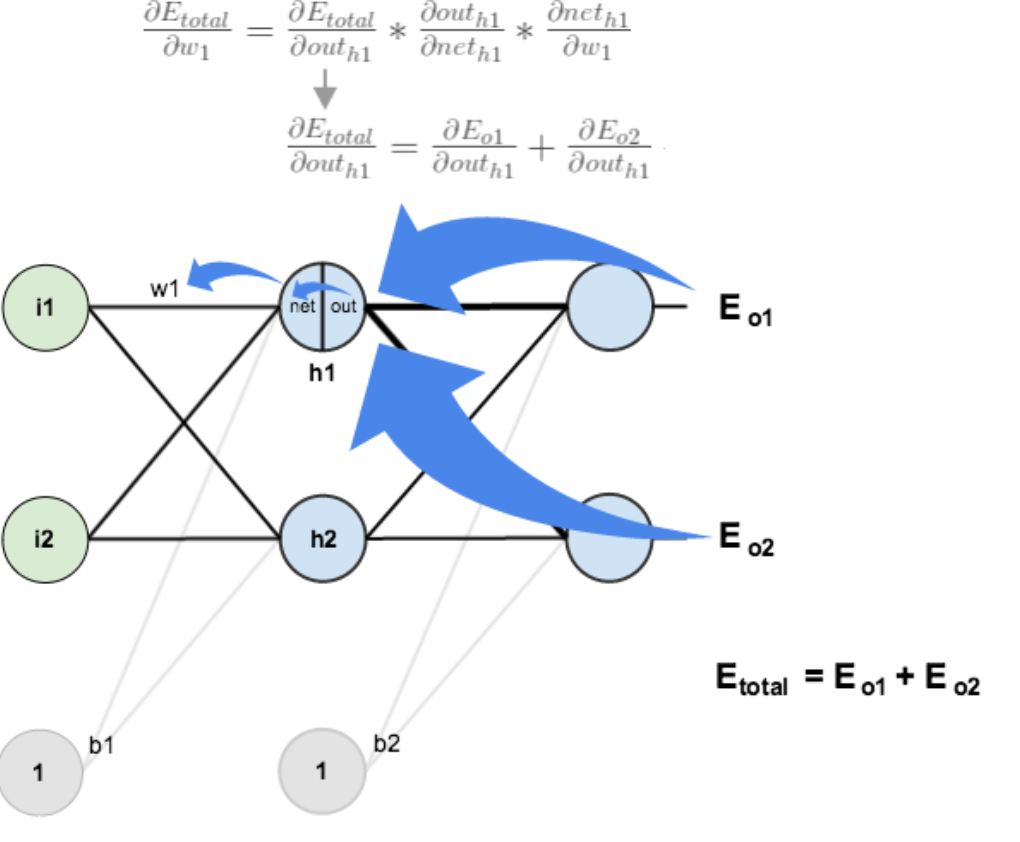
（其中，https://images2015.cnblogs.com/blog/853467/201606/853467-20160630153700093-743859667.png是学习速率，这里我们取0.5）

同理，可更新w6,w7,w8:



3.隐含层---->隐含层的权值更新：

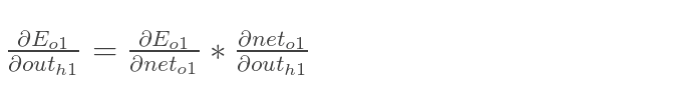
　方法其实与上面说的差不多，但是有个地方需要变一下，在上文计算总误差对w5的偏导时，是从out(o1)---->net(o1)---->w5,但是在隐含层之间的权值更新时，是out(h1)---->net(h1)---->w1,而out(h1)会接受E(o1)和E(o2)两个地方传来的误差，所以这个地方两个都要计算。

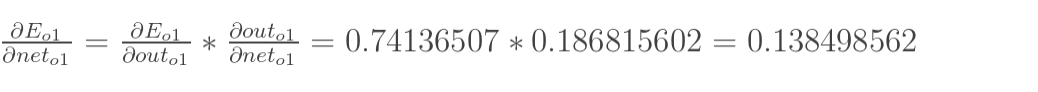


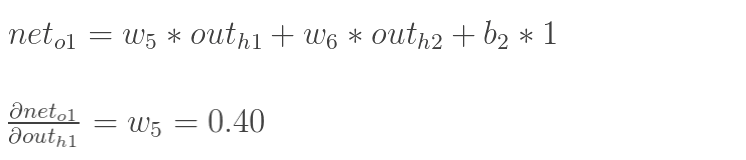
计算：



先计算：





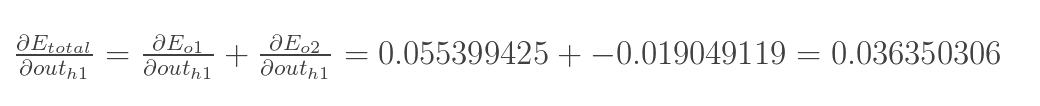




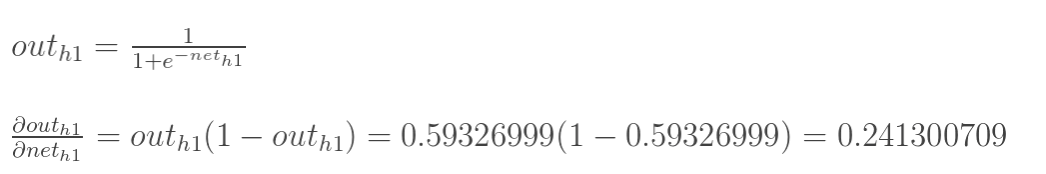
同理，计算出：



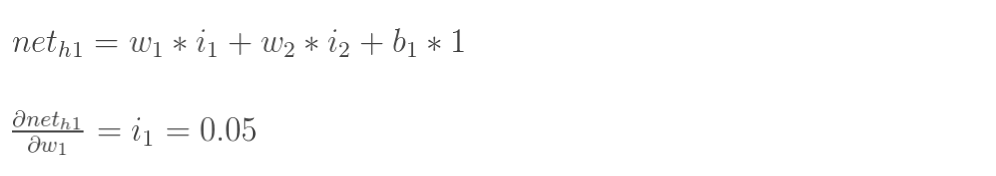
两者相加得到总值：



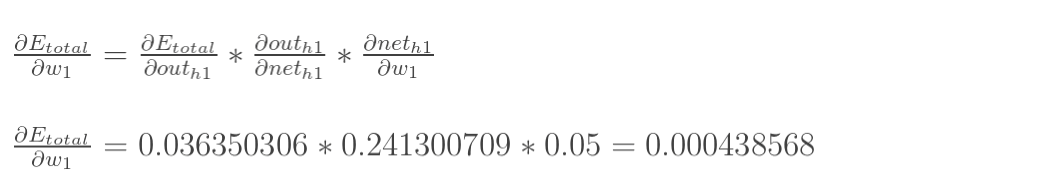
再计算：



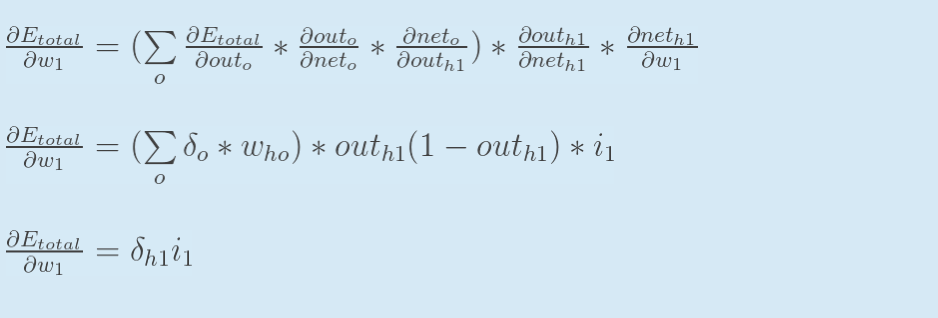
再计算：



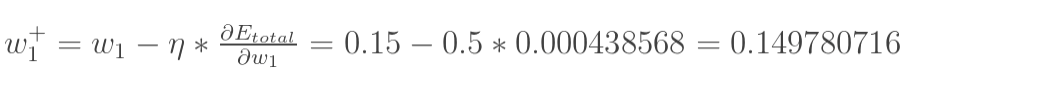
最后，三者相乘：



 为了简化公式，用sigma(h1)表示隐含层单元h1的误差：



最后，更新w1的权值：



同理，额可更新w2,w3,w4的权值：



　　这样误差反向传播法就完成了，最后我们再把更新的权值重新计算，不停地迭代，在这个例子中第一次迭代之后，总误差E(total)由0.298371109下降至0.291027924。迭代10000次后，总误差为0.000035085，输出为[0.015912196,0.984065734](原输入为[0.01,0.99]),证明效果还是不错的。

代码(Python):

#coding:utf-8

import random

import math

#

# 参数解释：

# "pd\_" ：偏导的前缀

# "d\_" ：导数的前缀

# "w\_ho" ：隐含层到输出层的权重系数索引

# "w\_ih" ：输入层到隐含层的权重系数的索引

class NeuralNetwork:

LEARNING\_RATE = 0.5

def \_\_init\_\_(self, num\_inputs, num\_hidden, num\_outputs, hidden\_layer\_weights = None, hidden\_layer\_bias = None, output\_layer\_weights = None, output\_layer\_bias = None):

self.num\_inputs = num\_inputs

self.hidden\_layer = NeuronLayer(num\_hidden, hidden\_layer\_bias)

self.output\_layer = NeuronLayer(num\_outputs, output\_layer\_bias)

self.init\_weights\_from\_inputs\_to\_hidden\_layer\_neurons(hidden\_layer\_weights)

self.init\_weights\_from\_hidden\_layer\_neurons\_to\_output\_layer\_neurons(output\_layer\_weights)

def init\_weights\_from\_inputs\_to\_hidden\_layer\_neurons(self, hidden\_layer\_weights):

weight\_num = 0

for h in range(len(self.hidden\_layer.neurons)):

for i in range(self.num\_inputs):

if not hidden\_layer\_weights:

self.hidden\_layer.neurons[h].weights.append(random.random())

else:

self.hidden\_layer.neurons[h].weights.append(hidden\_layer\_weights[weight\_num])

weight\_num += 1

def init\_weights\_from\_hidden\_layer\_neurons\_to\_output\_layer\_neurons(self, output\_layer\_weights):

weight\_num = 0

for o in range(len(self.output\_layer.neurons)):

for h in range(len(self.hidden\_layer.neurons)):

if not output\_layer\_weights:

self.output\_layer.neurons[o].weights.append(random.random())

else:

self.output\_layer.neurons[o].weights.append(output\_layer\_weights[weight\_num])

weight\_num += 1

def inspect(self):

print('------')

print('\* Inputs: {}'.format(self.num\_inputs))

print('------')

print('Hidden Layer')

self.hidden\_layer.inspect()

print('------')

print('\* Output Layer')

self.output\_layer.inspect()

print('------')

def feed\_forward(self, inputs):

hidden\_layer\_outputs = self.hidden\_layer.feed\_forward(inputs)

return self.output\_layer.feed\_forward(hidden\_layer\_outputs)

def train(self, training\_inputs, training\_outputs):

self.feed\_forward(training\_inputs)

# 1. 输出神经元的值

pd\_errors\_wrt\_output\_neuron\_total\_net\_input = [0] \* len(self.output\_layer.neurons)

for o in range(len(self.output\_layer.neurons)):

# ∂E/∂zⱼ

pd\_errors\_wrt\_output\_neuron\_total\_net\_input[o] = self.output\_layer.neurons[o].calculate\_pd\_error\_wrt\_total\_net\_input(training\_outputs[o])

# 2. 隐含层神经元的值

pd\_errors\_wrt\_hidden\_neuron\_total\_net\_input = [0] \* len(self.hidden\_layer.neurons)

for h in range(len(self.hidden\_layer.neurons)):

# dE/dyⱼ = Σ ∂E/∂zⱼ \* ∂z/∂yⱼ = Σ ∂E/∂zⱼ \* wᵢⱼ

d\_error\_wrt\_hidden\_neuron\_output = 0

for o in range(len(self.output\_layer.neurons)):

d\_error\_wrt\_hidden\_neuron\_output += pd\_errors\_wrt\_output\_neuron\_total\_net\_input[o] \* self.output\_layer.neurons[o].weights[h]

# ∂E/∂zⱼ = dE/dyⱼ \* ∂zⱼ/∂

pd\_errors\_wrt\_hidden\_neuron\_total\_net\_input[h] = d\_error\_wrt\_hidden\_neuron\_output \* self.hidden\_layer.neurons[h].calculate\_pd\_total\_net\_input\_wrt\_input()

# 3. 更新输出层权重系数

for o in range(len(self.output\_layer.neurons)):

for w\_ho in range(len(self.output\_layer.neurons[o].weights)):

# ∂Eⱼ/∂wᵢⱼ = ∂E/∂zⱼ \* ∂zⱼ/∂wᵢⱼ

pd\_error\_wrt\_weight = pd\_errors\_wrt\_output\_neuron\_total\_net\_input[o] \* self.output\_layer.neurons[o].calculate\_pd\_total\_net\_input\_wrt\_weight(w\_ho)

# Δw = α \* ∂Eⱼ/∂wᵢ

self.output\_layer.neurons[o].weights[w\_ho] -= self.LEARNING\_RATE \* pd\_error\_wrt\_weight

# 4. 更新隐含层的权重系数

for h in range(len(self.hidden\_layer.neurons)):

for w\_ih in range(len(self.hidden\_layer.neurons[h].weights)):

# ∂Eⱼ/∂wᵢ = ∂E/∂zⱼ \* ∂zⱼ/∂wᵢ

pd\_error\_wrt\_weight = pd\_errors\_wrt\_hidden\_neuron\_total\_net\_input[h] \* self.hidden\_layer.neurons[h].calculate\_pd\_total\_net\_input\_wrt\_weight(w\_ih)

# Δw = α \* ∂Eⱼ/∂wᵢ

self.hidden\_layer.neurons[h].weights[w\_ih] -= self.LEARNING\_RATE \* pd\_error\_wrt\_weight

def calculate\_total\_error(self, training\_sets):

total\_error = 0

for t in range(len(training\_sets)):

training\_inputs, training\_outputs = training\_sets[t]

self.feed\_forward(training\_inputs)

for o in range(len(training\_outputs)):

total\_error += self.output\_layer.neurons[o].calculate\_error(training\_outputs[o])

return total\_error

class NeuronLayer:

def \_\_init\_\_(self, num\_neurons, bias):

# 同一层的神经元共享一个截距项b

self.bias = bias if bias else random.random()

self.neurons = []

for i in range(num\_neurons):

self.neurons.append(Neuron(self.bias))

def inspect(self):

print('Neurons:', len(self.neurons))

for n in range(len(self.neurons)):

print(' Neuron', n)

for w in range(len(self.neurons[n].weights)):

print(' Weight:', self.neurons[n].weights[w])

print(' Bias:', self.bias)

def feed\_forward(self, inputs):

outputs = []

for neuron in self.neurons:

outputs.append(neuron.calculate\_output(inputs))

return outputs

def get\_outputs(self):

outputs = []

for neuron in self.neurons:

outputs.append(neuron.output)

return outputs

class Neuron:

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

self.bias = bias

self.weights = []

def calculate\_output(self, inputs):

self.inputs = inputs

self.output = self.squash(self.calculate\_total\_net\_input())

return self.output

def calculate\_total\_net\_input(self):

total = 0

for i in range(len(self.inputs)):

total += self.inputs[i] \* self.weights[i]

return total + self.bias

# 激活函数sigmoid

def squash(self, total\_net\_input):

return 1 / (1 + math.exp(-total\_net\_input))

def calculate\_pd\_error\_wrt\_total\_net\_input(self, target\_output):

return self.calculate\_pd\_error\_wrt\_output(target\_output) \* self.calculate\_pd\_total\_net\_input\_wrt\_input();

# 每一个神经元的误差是由平方差公式计算的

def calculate\_error(self, target\_output):

return 0.5 \* (target\_output - self.output) \*\* 2

def calculate\_pd\_error\_wrt\_output(self, target\_output):

return -(target\_output - self.output)

def calculate\_pd\_total\_net\_input\_wrt\_input(self):

return self.output \* (1 - self.output)

def calculate\_pd\_total\_net\_input\_wrt\_weight(self, index):

return self.inputs[index]

# 文中的例子:

nn = NeuralNetwork(2, 2, 2, hidden\_layer\_weights=[0.15, 0.2, 0.25, 0.3], hidden\_layer\_bias=0.35, output\_layer\_weights=[0.4, 0.45, 0.5, 0.55], output\_layer\_bias=0.6)

for i in range(10000):

nn.train([0.05, 0.1], [0.01, 0.09])

print(i, round(nn.calculate\_total\_error([[[0.05, 0.1], [0.01, 0.09]]]), 9))

#另外一个例子，可以把上面的例子注释掉再运行一下:

# training\_sets = [

# [[0, 0], [0]],

# [[0, 1], [1]],

# [[1, 0], [1]],

# [[1, 1], [0]]

# ]

# nn = NeuralNetwork(len(training\_sets[0][0]), 5, len(training\_sets[0][1]))

# for i in range(10000):

# training\_inputs, training\_outputs = random.choice(training\_sets)

# nn.train(training\_inputs, training\_outputs)

# print(i, nn.calculate\_total\_error(training\_sets))