Udacity ML part 4 神经网络回顾

**梯度下降：代码**

之前我们看到一个权重的计算是：

Δ*w*​*i*​​=*ηδx*​*i*​​

这里 error term *δ* 是指

*δ*=(*y*−​*y*​^​​)*f*​′​​(*h*)=(*y*−​*y*​^​​)*f*​′​​(∑*w*​*i*​​*x*​*i*​​)

记住，上面公式中 (*y*−​*y*​^​​) 是输出误差，*f*​′​​(*h*) 是激活函数 *f*(*h*) 的导函数，我们把这个导函数称做输出的梯度。

现在假设只有一个输出，我来把这个写成代码。我们还是用 sigmoid 来作为激活函数 *f*(*h*)。

*# Defining the sigmoid function for activations*

*# 定义 sigmoid 激活函数*

**def** **sigmoid**(x):

**return** 1/(1+np.exp(-x))

*# Derivative of the sigmoid function*

*# 激活函数的导数*

**def** **sigmoid\_prime**(x):

**return** sigmoid(x) \* (1 - sigmoid(x))

*# Input data*

*# 输入数据*

x = np.array([0.1, 0.3])

*# Target*

*# 目标*

y = 0.2

*# Input to output weights*

*# 输入到输出的权重*

weights = np.array([-0.8, 0.5])

*# The learning rate, eta in the weight step equation*

*# 权重更新的学习率*

learnrate = 0.5

*# the linear combination performed by the node (h in f(h) and f'(h))*

*# 输入和权重的线性组合*

h = x[0]\*weights[0] + x[1]\*weights[1]

*# or h = np.dot(x, weights)*

*# The neural network output (y-hat)*

*# 神经网络输出*

nn\_output = sigmoid(h)

*# output error (y - y-hat)*

*# 输出误差*

error = y - nn\_output

*# output gradient (f'(h))*

*# 输出梯度*

output\_grad = sigmoid\_prime(h)

*# error term (lowercase delta)*

error\_term = error \* output\_grad

*# Gradient descent step*

*# 梯度下降一步*

del\_w = [ learnrate \* error\_term \* x[0],

learnrate \* error\_term \* x[1]]

*# or del\_w = learnrate \* error\_term \* x*

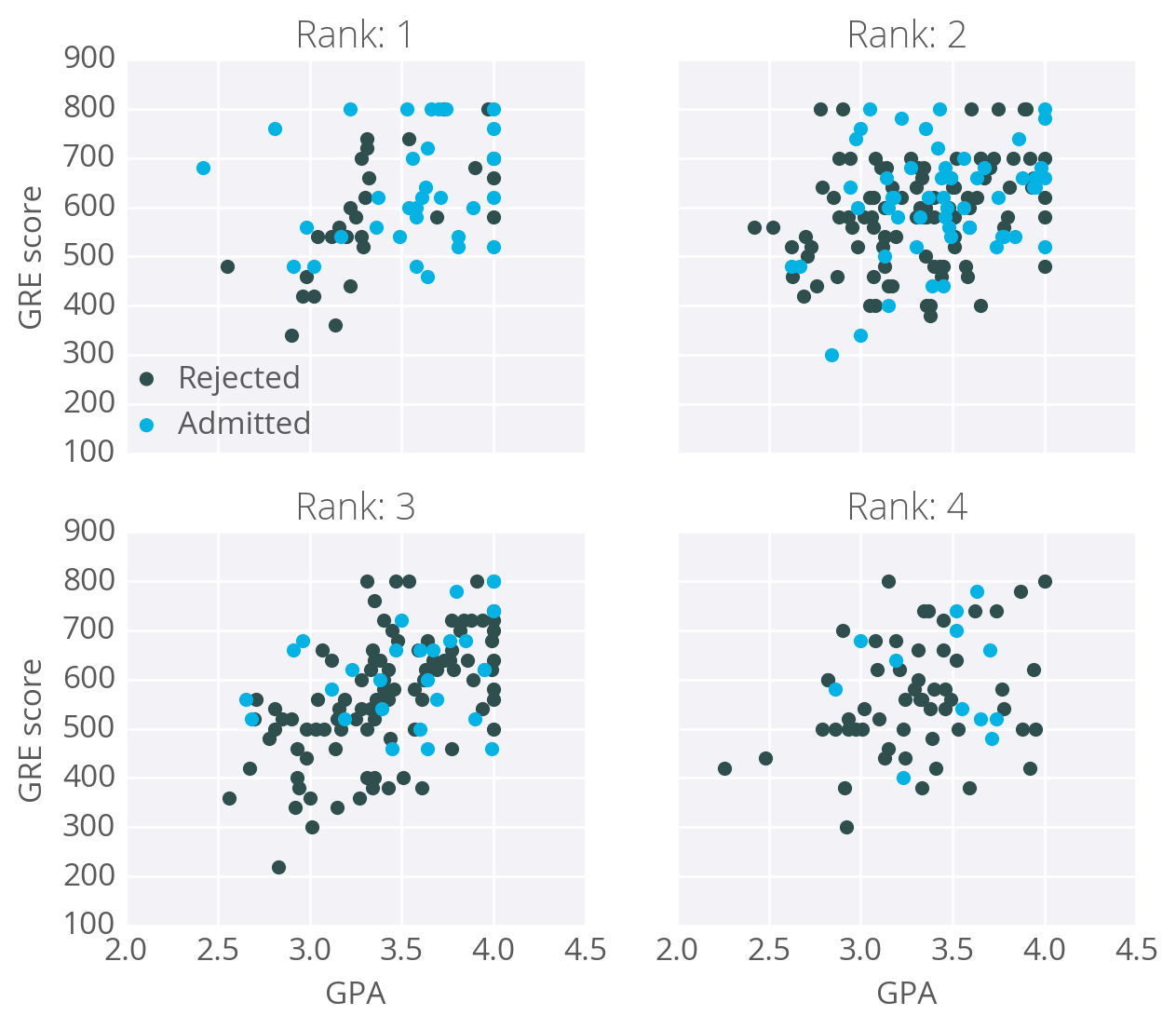
# 实现梯度下降

现在我们知道了如何更新我们的权重：

Δ*w*​*ij*​​=*ηδ*​*j*​​*x*​*i*​​,

你看到的是如何实现一次更新，那我们如何把代码转化为能够计算多次权重更新，使得我们的网络能够真正学习呢？

我们拿一个研究生学院录取数据来用梯度下降训练一个网络。数据可以在[**这里**](http://www.ats.ucla.edu/stat/data/binary.csv)找到。数据有三个输入特征，GRE分数，GPA，和本科院校排名（从1到4）。数字1代表最好，数字4代表最差。



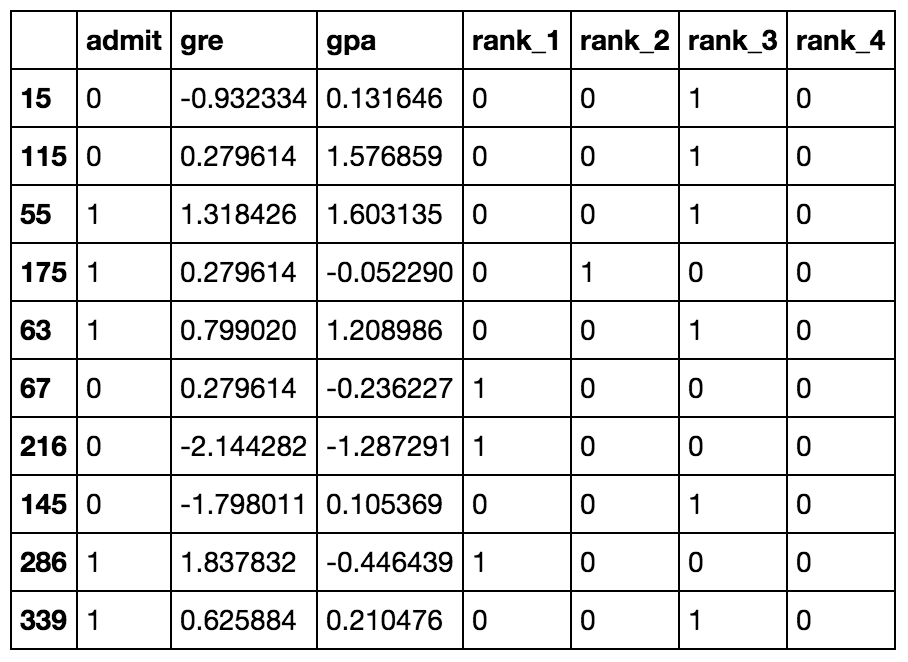
我们的目标是基于这些特征来预测一个学生能否被研究生院录取。这里，我们用有一个输出层的网络。用 sigmoid 做为激活函数。

## 数据清理

你也许认为有三个输入特征，但是首先我们要做数据转换。rank 是类别特征，数字并不包含任何想对的值。排名第 2 并不是排名第 1 的两倍；排名第 3 也不是排名第 2 的 1.5 倍。因此，我们需要用 [**dummy variables**](https://en.wikipedia.org/wiki/Dummy_variable_(statistics)) 来对rank进行编码。把数据分成 4 个新列，用 0 或 1 表示。排名第一的行 rank 1那一列的值是 1 ，其它是 0；排名第二的行 rank 2 那一列的值是 1 ，其它是 0，以此类推。

我们还需要把 GRE 和 GPA 分数标准化，也就是说使得他们的平均值是 0，标准差是 1。因为 sigmoid 函数会挤压很大或者很小的输入，所以这一步是必要的。很大或者很小输入的梯度是 0 意味着梯度下降的步长也会是 0。因为 GRE 和 GPA 的值都相对较大，我们在初始化权重的时候要非常小心，否则梯度下降就会死亡，网络也没发训练了。如果我们对数据做了标准化处理，我们能更容易地对权重进行初始化。

这只是一个简单介绍，你之后还会学到如何预处理数据，如果你想了解我是怎么做的，可以查看下面编程练习中的data\_prep.py 文件。

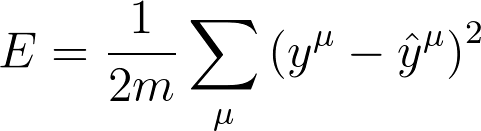


经过转换后的10行数据

现在数据已经准备好了，我们看到有六个输入特征：gre, gpa，和四个rank 的虚拟变量 （dummy variables）。

### 均方差

这里我们要对如何计算误差做一点小改变。除了SSE，我们这里用误差平方的**平均数**(mean of the square errors，MSE)。现在我们要处理很多数据，把所有权重更新加起来会导致很大的更新，使得梯度下降无法收敛。为了避免这种情况，你需要一个很小的学习率。这里我们还可以除以数据点的数量 *m* 来取平均。这样，无论我们有多少数据，我们的学习率通常会在 0.01 to 0.001 之间。我们用 MSE（下图）来计算梯度，结果跟之前一样，只是取了平均而不是取和。



这是用梯度下降来更新权重的算法概述：

* 权重更新初始为 0： Δ*w*​*i*​​=0
* 训练数据中的每一条记录：
  + 通过网络做正向传播，计算输出 ​*y*​^​​=*f*(∑​*i*​​*w*​*i*​​*x*​*i*​​)
  + 计算输出的 error term, *δ*=(*y*−​*y*​^​​)∗*f*​′​​(∑​*i*​​*w*​*i*​​*x*​*i*​​)
  + 更新权重步长 Δ*w*​*i*​​=Δ*w*​*i*​​+*δx*​*i*​​
* 更新权重 *w*​*i*​​=*w*​*i*​​+*η*Δ*w*​*i*​​/*m*。 *η* 是学习率， *m* 是数据点个数。 这里我们对权重步长做了平均，为的是降低训练数据中大的变化。
* 重复 *e* 代（epoch）。

你也可以选择每一个记录更新一下权重，而不是把所有记录都训练过之后再取平均。

这里我们还是使用 sigmoid 作为激活函数

*f*(*h*)=1/(1+*e*​−*h*​​)

sigmoid 的梯度是： *f*​′​​(*h*)=*f*(*h*)(1−*f*(*h*))

*h* 从输入计算的输出,

*h*=∑​*i*​​*w*​*i*​​*x*​*i*​​

## 用 NumPy 来实现

这里大部分都可以用 NumPy 很方便的实现。

首先你需要初始化权重。我们希望它们比较小，这样输入在 sigmoid 函数那里可以在接近 0 的位置，而不是最高或者最低处。很重要的一点是要随机地初始化它们，这样它们有不同的值，是发散且不对称的。所以我们从一个中心为 0 的正态分布来初始化权重。一个好的标准差的值是 1/√​*n*​​​，这里 *n* 是输入的个数。这样就算是输入个数变多，进到 sigmoid 的输入还能保持比较小。

weights = np.random.normal(scale=1/n\_features\*\*.5, size=n\_features)

NumPy 提供了一个可以让两个序列做点乘的函数，它可以让我们方便地计算 *h*。点乘是把两个序列的元素对应位置相乘之后再相加。

*# input to the output layer*

output\_in = np.dot(weights, inputs)

最后我们更新 Δ*w*​*i*​​ 和 *w*​*i*​​，weights += ... 是 weights = weights + ... 的简写。

### 提示

因为这里我们用 sigmoid 函数。它有一个特性是 *f*​′​​(*h*)=*f*(*h*)(1−*f*(*h*))。也就是说一旦你有了 *f*(*h*)，你就可以用它来计算误差的梯度了。

## 编程练习

接下来，你要实现一个梯度下降，用入去数据来训练它。你的目标是训练一个网络直到你达到训练数据的最小的均方差mean square error (MSE)。你需要实现：

* 网络的输出: output
* 输出误差: error
* 误差项: error\_term
* 权重更新步长: del\_w +=
* 更新权重: weights +=

在你写完这几部分之后，按“测试”按钮来进行训练，均方差会被打印出来，同时测试集的准确率，也就是有多少学生被预测正确了，也会被打印出来。

你可以任意调节超参数 hyperparameters 来看下它对均方差 MSE 有什么影响。

# 实现隐藏层

##### 先修要求

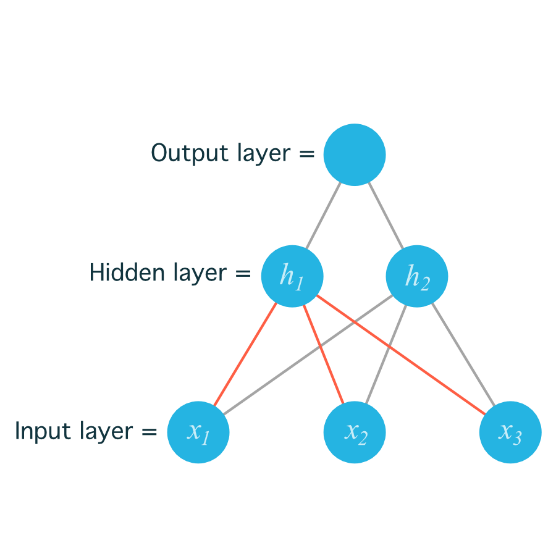
接下来我们会讲神经网络在多层感知器里面的数学部分。讲多层感知器我们会用到向量和矩阵。你可以通过下列讲解对此做个回顾：

1. Khan Academy's [**introduction to vectors**](https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/vectors/v/vector-introduction-linear-algebra).
2. Khan Academy's [**introduction to matrices**](https://www.khanacademy.org/math/precalculus/precalc-matrices).

##### 由来

之前我们研究的是有一个输出节点网络，代码也很直观。但是现在我们有不同的输入，多个隐藏层，他们的权重需要有两个索引 *w*​*ij*​​，*i* 表示输入单位，*j* 表示隐藏单位。

例如在下面这个网络图中，输入被标注为 *x*​1​​,*x*​2​​, *x*​3​​，隐藏层节点是 *h*​1​​ 和 *h*​2​​。



指向 *h*​1​​ 的权重被标成了红色，这样更好区分。

为了定位权重，我们把输入节点的索引 ​*i*​​ 和输出节点的索引 ​*j*​​ 结合，得到：

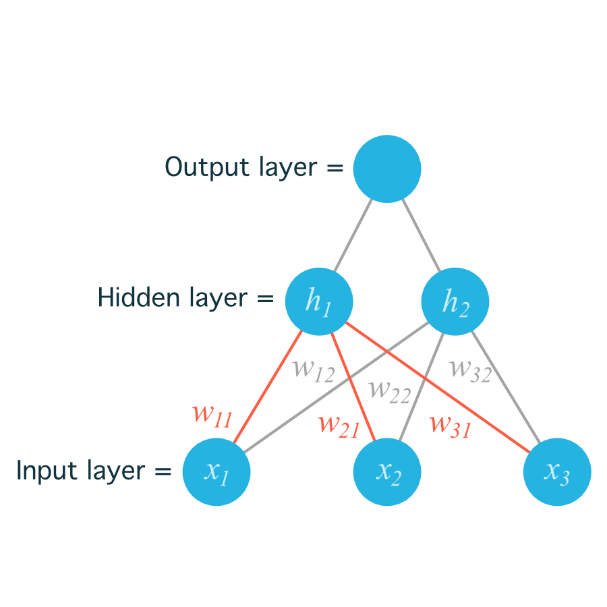
*w*​11​​

代表从 *x*​1​​ 到 *h*​1​​ 的权重；

*w*​12​​

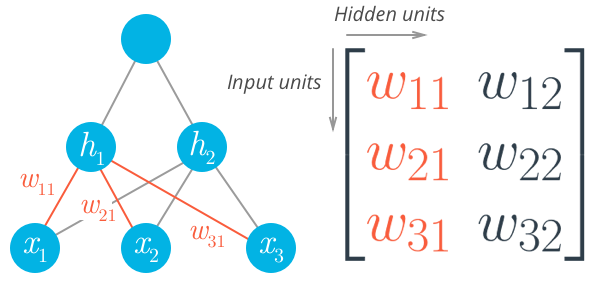
代表从 *x*​1​​ 到 *h*​2​​ 的权重。

下图包括了从输入层到隐藏层用 *w*​*ij*​​ 来标注的所有权重：



之前我们可以把权重写成序列，标记为 *w*​*i*​​。

现在，权重被储存在**矩阵**中，由 *w*​*ij*​​ 来标记。每一**行**表示从输入层**发出**的权重，每一**列**表示从输入到隐藏层的权重。这里我们由三个输入，两个因此节点，权重矩阵标示为：



三个输入两个隐藏

记得比较一下上面的示意图来确定你了解不同的权重在矩阵中神经网络中的对应关系。

用 NumPy 来初始化权重，我们需要提供矩阵的维度，如果特征是包含输入的二维序列：

*# Number of records and input units*

*# 数据点以及每个数据点有多少输入的个数*

n\_records, n\_inputs = features.shape

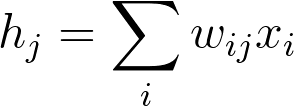
*# Number of hidden units*

*# 隐藏层个数*

n\_hidden = 2

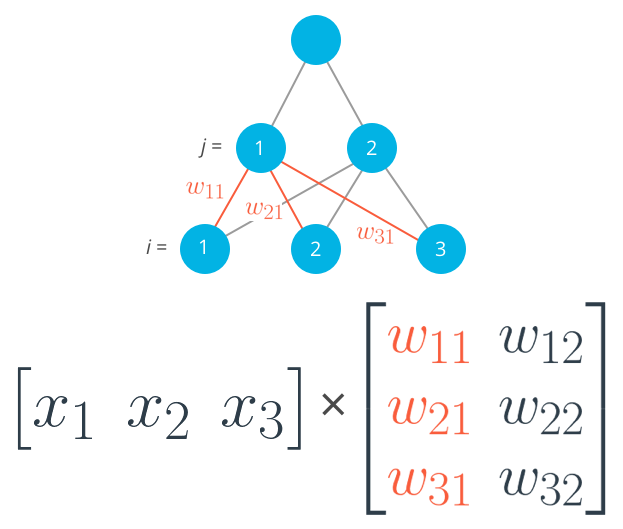
weights\_input\_to\_hidden = np.random.normal(0, n\_inputs\*\*-0.5, size=(n\_inputs, n\_hidden))

这样创建了一个 名为 weights\_input\_to\_hidden 的 2D 序列，维度是 n\_inputs 乘 n\_hidden。记住，输入到隐藏层是所有输入乘以隐藏层权重的和。所以对每一个隐藏层节点 *h*​*j*​​，我们需要计算：



为了实现这点，我们需要运用[**矩阵乘法**](https://en.wikipedia.org/wiki/Matrix_multiplication)，如果你对线性代数有点忘了，我们建议你看下之前先修部分的资料。这里你只需要了解矩阵如何相乘。

在这里，我们把输入（一个行向量）与权重相乘。要实现这个，要把输入点乘（内积）以权重矩阵的每一列。例如要计算到第一个隐藏节点的输入 *j*=1，你需要把这个输入与权重矩阵的第一列做点乘：



用输入与权重矩阵的第一列相乘得出到隐藏层第一个节点的输入

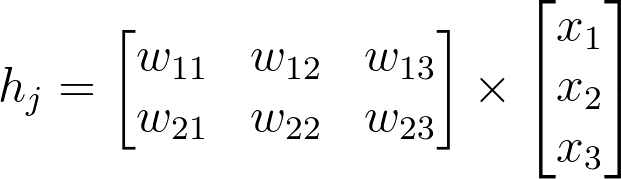
https://d17h27t6h515a5.cloudfront.net/topher/2017/January/588ae392_codecogseqn-2/codecogseqn-2.png

针对第二个隐藏节点的输入，你需要计算输入与第二列的点积，以此类推。

在 NumPy 中，你可以直接使用 np.dot

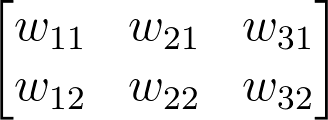
hidden\_inputs = np.dot(inputs, weights\_input\_to\_hidden)

你可以定义你的权重矩阵是 n\_hidden 乘 n\_inputs 然后把输入作为**竖向量**相乘：



**注意：**

这里权重的索引在上图中做了改变，与之前图片并不匹配。这是因为，在矩阵标注时行索引永远在列索引之前，所以用之前的方法做标识会引起误导。你只需要了解这跟之前的权重矩阵是一样的，只是做了转换，之前的第一列现在是第一行，之前的第二列现在是第二行。如果用**之前**的标记，权重矩阵是下面这个样子的：



用之前的标记标注的权重矩阵

切记，上面标注方式是**不正确**的，这里只是为了让你更清楚这个矩阵如何跟之前神经网络的权重匹配。

矩阵相乘最重要的是他们的**维度相匹配**。因为它们在点乘时需要有相同数量的元素。在第一个例子中，输入向量有三列，权重矩阵有三行；第二个例子中，权重矩阵有三列，输入向量有三行。如果维度不匹配，你会得到：

# Same weights and features as above, but swapped the order

hidden\_inputs = np.dot(weights\_input\_to\_hidden, features)

---------------------------------------------------------------------------

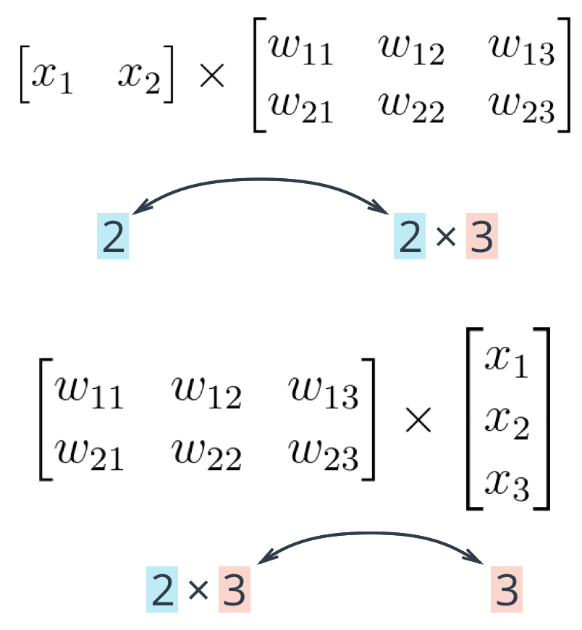
ValueError Traceback (most recent call last)

<ipython-input-11-1bfa0f615c45> in <module>()

----> 1 hidden\_in = np.dot(weights\_input\_to\_hidden, X)

ValueError: shapes (3,2) and (3,) not aligned: 2 (dim 1) != 3 (dim 0)

3x2 的矩阵跟 3 个元素序列是没发相乘的。因为矩阵中的两列与序列中的元素个数并不匹配。能够相乘的矩阵如下：



这里的规则是，如果是序列在左边，序列的元素个数必须与右边矩阵的行数一样。如果矩阵在左边，那么矩阵的列数，需要与右边向量的行数匹配。

### 构建一个列向量

看上面的介绍，你有时会需要一个列向量，尽管 NumPy 默认是行向量。你可以用 arr.T 来对序列进行转制，但对一维序列来说，转制还是行向量。所以你可以用 arr[:,None] 来创建一个列向量：

print(features)

> array([ 0.49671415, -0.1382643 , 0.64768854])

print(features.T)

> array([ 0.49671415, -0.1382643 , 0.64768854])

print(features[:, **None**])

> array([[ 0.49671415],

[-0.1382643 ],

[ 0.64768854]])

当然，你可以创建一个二维序列，然后用 arr.T 得到列向量。

np.array(features, ndmin=2)

> array([[ 0.49671415, -0.1382643 , 0.64768854]])

np.array(features, ndmin=2).T

> array([[ 0.49671415],

[-0.1382643 ],

[ 0.64768854]])

## 编程练习

下面你要实现一个 4x3x2 网络的正向传播，用 sigmoid 作为两层的激活函数。

要做的事情：

* 计算到隐藏层的输入
* 计算隐藏层输出
* 计算输出层的输入
* 计算神经网络的输出

Multilayer.py

# 反向传播

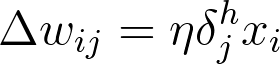
现在我们来到了如何让多层神经网络**学习**的问题上。之前我们了解了如何用梯度下降来更新权重。反向传播算法是它的一个延伸，用链式法则来找到误差与输入层到输入层链接的权重（两层神经网络）。

要更新输入到隐藏层的权重，你需要知道隐藏层节点的误差对最终输出的影响是多大。输出是由两层之间的权重决定的，这个误差是输入跟权重在网络中正向传播的结果。既然我们知道输出误差，我们可以用权重来反向传播到隐藏层。

例如，输出层的话，在每一个输出节点 *k* 的误差是 *δ*​*k*​*o*​​ 。隐藏节点 *j* 的误差来自输出层和隐藏层之间的权重（以及梯度）。



梯度下降跟之前一样，只是用新的误差：



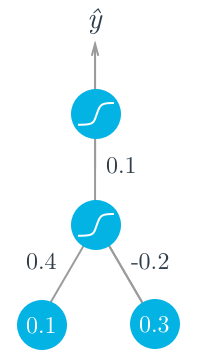
*w*​*ij*​​ 是输入和隐藏层之间的权重， *x*​*i*​​ 是输入值。这个形式可以表示任意层数。权重更新步长等与步长乘以层输出误差再乘以这层的输入值。

https://d17h27t6h515a5.cloudfront.net/topher/2017/January/588bc2d4_backprop-general/backprop-general.gif

这里，你有了输出误差，*δ*​*output*​​，从高层反向传播这些误差。*V*​*in*​​ 是对这一层的输入，经过隐藏层激活后到输出节点。

### 通过一个实际案例学习

让我们一起过一遍计算一个简单的两层网络权重的更新过程。假设有两个输入值，一个隐藏节点，一个输出节点，隐藏层和输出层的激活函数都是 sigmoid 。下图描述了这个网络。（**注意**：输入在这里显示为图最下方的节点，网络的输出标记为顶端的​*y*​^​​，输入本身不算做层，这也是为什么这个网络结构被称作两层网络。）



假设我们试着输入一些二分类数据，目标是 *y*=1。我们从正向传导开始，首先计算输入到隐藏层

*h*=∑​*i*​​*w*​*i*​​*x*​*i*​​=0.1×0.4−0.2×0.3=−0.02

隐藏层的输出

*a*=*f*(*h*)=sigmoid(−0.02)=0.495.

把它作为输出层的输入，神经网络的输出是：

​*y*​^​​=*f*(*W*⋅*a*)=sigmoid(0.1×0.495)=0.512.

有了这个输出，我们就可以开始反向传播来计算两层的权重更新了。sigmoid 函数特性 *f*​′​​(*W*⋅*a*)=*f*(*W*⋅*a*)(1−*f*(*W*⋅*a*))，输出误差是：

*δ*​*o*​​=(*y*−​*y*​^​​)*f*​′​​(*W*⋅*a*)=(1−0.512)×0.512×(1−0.512)=0.122.

现在我们要通过反向传播来计算隐藏层的误差。这里我们吧输出误差与隐藏层到输出层的权重 *W* 相乘。隐藏层的误差 *δ*​*j*​*h*​​=∑​*k*​​*W*​*jk*​​*δ*​*k*​*o*​​*f*​′​​(*h*​*j*​​)，这里因为只有一个隐藏节点，这就比较简单了

*δ*​*h*​​=*Wδ*​*o*​​*f*​′​​(*h*)=0.1×0.122×0.495×(1−0.495)=0.003

有了误差，就可以计算梯度下降步长了。隐藏层到输出层权重步长是学习率乘以输出误差再乘以隐藏层激活值。

Δ*W*=*ηδ*​*o*​​*a*=0.5×0.122×0.495=0.0302

从输入到隐藏层的权重 *w*​*i*​​，是学习率乘以隐藏节点误差再乘以输入值。

Δ*w*​*i*​​=*ηδ*​*h*​​*x*​*i*​​=(0.5×0.003×0.1,0.5×0.003×0.3)=(0.00015,0.00045)

这个例子你可以看出用 sigmoid 做激活函数的效果。sigmoid 函数最大的导数是 0.25，输出层的误差被至少减少了75%，隐藏层的误差被减少了至少93.75%！你可以看出，如果你有很多层，用 sigmoid 激活函数函数会很快把权重降到靠近输入的细小值。这被称作**梯度消失**问题。后面的课程中你会学到其它的激活函数在这方面表现比它好，也被用于最新的网络架构中。

## 用 NumPy 来实现

现在你已经有了大部分用 NumPy 来实现反向传导的知识。

但是之前接触的是一个元素的误差项。现在在权重更新时，我们需要考虑隐藏层每个节点的误差：

Δ*w*​*ij*​​=*ηδ*​*j*​​*x*​*i*​​

首先，这里会有不同数量的输入和隐藏节点，所以试图把误差与输入当作行向量来乘会报错

hidden\_error\*inputs

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-22-3b59121cb809> in <module>()

----> 1 hidden\_error\*x

ValueError: operands could not be broadcast together with shapes (3,) (6,)

同时，*w*​*ij*​​ 现在是一个矩阵，所以右侧对应也应该有跟左侧同样的维度。幸运的是，NumPy 这些都能搞定。如果你用一个列向量序列和一个行向量序列乘，它会把列向量的第一个元素与行向量的每个元素相乘，然后第一行是一个二维序列。列向量的每一个元素会这门做，所以你会得到一个二维序列，维度是(len(column\_vector), len(row\_vector))。

hidden\_error\*inputs[:,**None**]

array([[ -8.24195994e-04, -2.71771975e-04, 1.29713395e-03],

[ -2.87777394e-04, -9.48922722e-05, 4.52909055e-04],

[ 6.44605731e-04, 2.12553536e-04, -1.01449168e-03],

[ 0.00000000e+00, 0.00000000e+00, -0.00000000e+00],

[ 0.00000000e+00, 0.00000000e+00, -0.00000000e+00],

[ 0.00000000e+00, 0.00000000e+00, -0.00000000e+00]])

这正好是我们如何计算权重更新的步长。跟以前一样，如果你的输入是一个一行的二维序列，你可以用hidden\_error\*inputs.T，如果 inputs 是一维序列，就不行了。

## 反向传播练习

接下来你会用代码来实现一次两个权重的反向传播更新。我们提供了正向传播的代码，你来实现反向传播的代码。

要做的事

* 计算网络输出误差
* 计算输出层误差的梯度
* 用反向传播计算隐藏层误差
* 计算权重更新步长

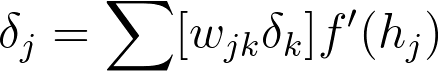
Backprop.py

# 实现反向传播

现在我们知道输出层的误差是

*δ*​*k*​​=(*y*​*k*​​−​*y*​^​​​*k*​​)*f*​′​​(*a*​*k*​​)

输入层误差是



现在我们只考虑一个简单神经网络，他只有一层隐藏层和一个输出节点。这是通过反向传播更新权重的算法概述：

* 把每一层权重更新的初始步长设置为 0
  + 输入到隐藏层的权重是 Δ*w*​*ij*​​=0
  + 隐藏层到输出层的权重是 Δ*W*​*j*​​=0
* 对训练数据当中的每一个点
  + 让它正向通过网络，计算输出 ​*y*​^​​
  + 计算输出节点的误差梯度 *δ*​*o*​​=(*y*−​*y*​^​​)*f*​′​​(*z*) 这里 *z*=∑​*j*​​*W*​*j*​​*a*​*j*​​ 输入到输出节点。
  + 误差传播到隐藏层 *δ*​*j*​*h*​​=*δ*​*o*​​*W*​*j*​​*f*​′​​(*h*​*j*​​)
  + 更新权重步长：
    - Δ*W*​*j*​​=Δ*W*​*j*​​+*δ*​*o*​​*a*​*j*​​
    - Δ*w*​*ij*​​=Δ*w*​*ij*​​+*δ*​*j*​*h*​​*a*​*i*​​
* 更新权重, *η* 是学习率，*m* 是数据点的数量：
  + *W*​*j*​​=*W*​*j*​​+*η*Δ*W*​*j*​​/*m*
  + *w*​*ij*​​=*w*​*ij*​​+*η*Δ*w*​*ij*​​/*m*
* 重复这个过程 *e* 代。

## 反向传播练习

现在你来实现一个通过反向传播训练的神经网络，数据集就是之前的研究生院录取数据。通过前面所学你现在有能力完成这个练习：

你的目标是：

* 实现一个正向传播
* 实现反向传播
* 更新权重

Backprop2.py

## 进阶阅读

反向传播 Backpropagation 是深度学习的基础。TensorFlow 或者其它框架会替你把它做好，但是你应该理解它的算法。我们后面还会讲到它，这里有些材料你可以看一下：

* From Andrej Karpathy: [**Yes, you should understand backprop**](https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.vt3ax2kg9)
* Also from Andrej Karpathy, [**a lecture from Stanford's CS231n course**](https://www.youtube.com/watch?v=59Hbtz7XgjM)

http://cs231n.github.io/optimization-2/