一、反向传播算法

反向传播算法是机器学习领域常用的一种算法，是一种利用链式法则递归计算表达式的梯度的方法。

所谓链式法则即求导法则中的链式法则，如果函数有嵌套，可以逐层求导。

核心问题是：给定函数 $f(x)$，其中 x 是输入数据的向量，如何计算其关于 x 的梯度，也

之所以关注这个问题，是因为在神经网络中需要优化损失函数使得性能更好，这种优化的过程经常需要沿着梯度的反方向来进行，所以求梯度是一种常用的方式。

1. 梯度的理解。

这个其实很好理解，梯度即是各个变量的偏导数按照一定顺序组成的向量。和高等数学中的梯度概念是完全一样的，不再赘述。

要理解：函数关于每个变量的偏导数指明了整个表达式关于该变量的敏感程度。

2. 链式法则求复合表达式的梯度。

考虑 $f(x)$=z(x+y)，我们拆解成 $f=q*z$ 与 $q=x+y$，这样我们可以利用链式法则来计算

这个梯度：

$$\frac{df}{dz}=q$$

$$\frac{df}{dq}=z$$

$$\frac{dq}{dx}=\frac{dq}{dy}=1$$

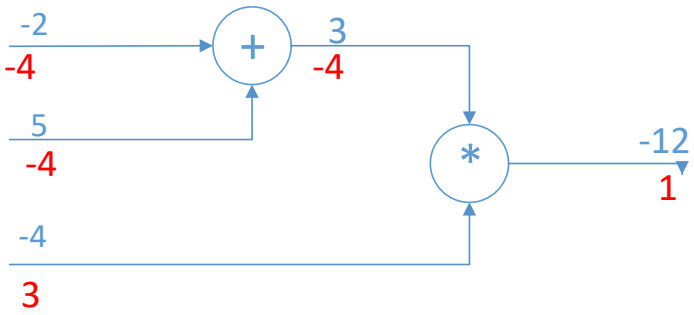
所以根据链式求导法则:

$$\frac{df}{dx}=\frac{df}{dq}\frac{dq}{dx}=z$$

$$\frac{df}{dy}=\frac{df}{dq}\frac{dq}{dy}=z$$

实际操作中，知识简单的将两个梯度数值相乘即可，示例代码：

```
"""
#设置输入
x=-2;y=5;z=-4
#进行前向传播，即计算函数的值
q=x+y      #q  3
f=q*z      #f  -12

dqdx=1;
dqdy=1;      #q对x和y的梯度

#进行逆向传播，首先处理f=q*z
dfdz=q     #所以关于z的梯度是3
dfdq=z     #所以关于q的梯度是-4

dfdx=dfdq*dqdx
dfdy=dfdq*dqdy

print("dfdz,dfdx,dfdy分别是：")
print(dfdz,dfdx,dfdy)

```



上图的计算路线显示了计算的视觉化过程，前向传播从输入计算到输出，反向传播从尾部开始前向地计算梯度（红色表示），可以认为梯度是从计算中回流。

我们把每个运算符想象成一个门单元，在整个计算路线图中，每个门单元都会得到一些输入并立即计算出两样东西：

① 这个门的输出值。

② 其输出值关于输入值的局部梯度。（这个主要是由运算形式决定的？）

门单元完成这两件事完全是独立的，不需知道计算路中的其他地方的细节，然而，一旦前向传播完毕，在反向传播的过程中，门单元将获得整个网络的输出在自己的输出值上的梯度，链式法则指出，们单元应该将回传的梯度乘以它对其输入的局部梯度，从而得到整个网络的输出对于该门单元的每个输入值的梯度。

所以这样来说，反向传播可以看作是门单元之间通过梯度信号互相通信，只要让它们的输入沿着梯度方向变化，无论他们自己的输出值在何种程度上上升或者降低，都是为了让整个网络的输出更高（建立在这个门沿着梯度方向变化）。

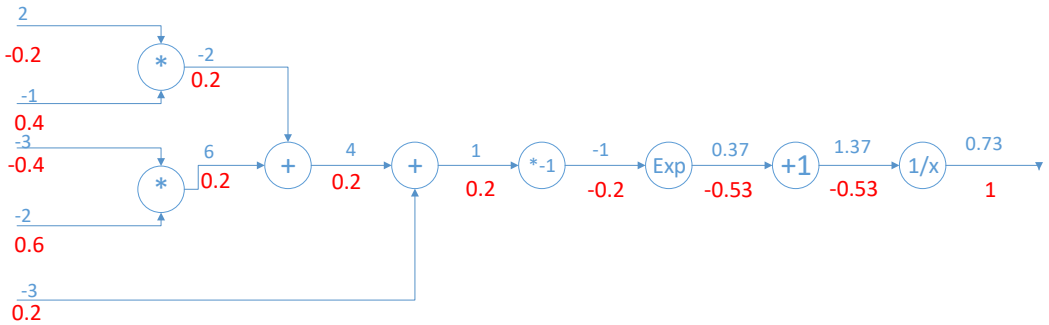
比如对于加法门的 3 来说，其梯度是负的，所以这个数如果变小的话可以使得最后的梯度增加。又比如说-4 的输入，如果让其沿着其梯度方向（正），增加，就会导致整体的输出增加。但必须注意，由于梯度的定义，这种沿着梯度的增加只在比较小的范围内有效，因为梯度本身就是会随着变量的变化变换的。

3. Sigmoid 例子

$$f(w,x)=\frac{1}{1+e^{-(w_0x_0+w_1x_2+w_2)}}$$

这个函数作为一个激活函数是经常用的，除了简单的加法和乘法之外，还有倒数，指数等，我们把这样的一个函数的前向和后向传播算法画出来看一下：

假设输入是$[w_0,x_0,w_1,x_2,w_2]=[2,-1,-3,-2,-3]$



这样写出来的前向传播和后向传播的可视化图是这样的，这种算法应当是要掌握的。算的时候只要牢记链式法则就 ok 了。

在这个函数中，我们是对 w 和 x 的点积进行操作，我们现在把这个看成一个向量，并且把这个函数记作：

$$\sigma(x)=\frac{1}{1+e^{-x}}$$

求导：

$$\sigma(x)=\frac{1}{1+e^{-x}}$$
$$\rightarrow \frac{d\sigma(x)}{dx}=\frac{e^{-x}}{(1+e^{-x})^2}=(\frac{1+e^{-x}-1}{1+e^{-x}})(\frac{1}{1+e^{-x}})=(1-\sigma(x))\sigma(x)$$

我们发现这个是可以化简的，由于指数函数的特殊求导法则，使得这个形式很美观整齐。
我们把中操作封装起来，在实际的操作中这也是非常有用的。
测试代码：

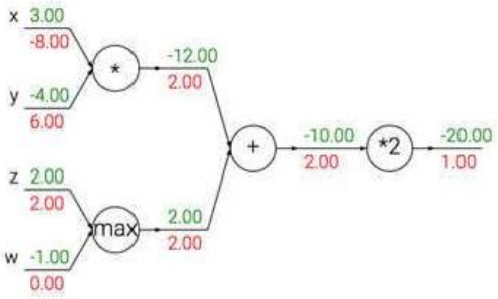
```
import math
w=[2,-3,-3]
x=[-1,-2]
dot=w[0]*x[1]+w[1]*x[1]+w[2]
f=1.0/(1+math.exp(-dot))

ddot=(1-f)*f     #函数求导
#回传
dx=[w[0]*ddot,w[1]*ddot]
dw=[x[0]*ddot,x[1]*ddot,1*ddot]
print(dx)
print(dw)

```

[0.3932238664829637, -0.5898357997244456]

[-0.19661193324148185, -0.3932238664829637, 0.19661193324148185]

输出和我在可视化途中计算的略有不同，主要原因是对一些指数做了近似。
我们也可以看到，许多时候在反向传播的时候会用到前向传播所计算的值，所以在编程的时候最好对这些中间变量进行缓存，这样在反向传播的时候就可以用到了。

4. 回传流中的模式



神经网络中最常用的三种门单元分别是加，乘法，最大值，看上面这个例子。
加法操作将梯度相等得分发给它的输入，取最大值操作将梯度路由给更大的输入，乘法门拿取输入激活数据，对他们进行交换，然后乘以梯度，也就是一个变量的值实际上是影响的另一个变量的梯度。
从上可知：
加法门单元：把输入的梯度相等得分发给它所有的输入，这一行为与输入值在前向传播的值无关，因为加法操作所有的梯度都是 1，所以所有输入的梯度就等于输出的梯度。
取最大值门单元：对梯度做路由，和加法门不同，取最大值门将梯度传给其中一个输入，这个输入应该是前向传播中最大的那个输入，其余的是 0。

乘法门单元：这个相对不容易解释，他的局部梯度就是输入值，但是是交换之后的，对于乘法来说，一个变量的梯度实际上是收到另外一个变量的影响的。
非直观影响：注意一种特殊情况，如果乘法门单元的其中一个输入非常小，另一个非常大，那么乘法门的操作将不会那么只管：它将比较大的梯度分配给小的输入，把小的梯度分配给大的输入。在线性分类器中，$f = w^T x_i$ 说明输入数据的大小对权重梯度的大小有影响，例如我们如果把所有的输入乘以 1000，那么对权重梯度的影响将对增加 1000 倍，这样就必须通过降低学习率来弥补，这就是为什么数据预处理的关系重大，将直接影响到学习率。

5. 用向量化操作来计算梯度

上述考虑的都是单个变量的情况，但是所有的概念都是适用于矩阵操作的，只是在操作的时候要注意关注维度和转置操作。

```
41 import numpy as np
42 W=np.random.randn(5,10)
43 X=np.random.randn(10,3)
44 D=W.dot(X)
45
46 #假设我们得到了D的梯度
47 dD=np.random.randn(*D.shape)
48 dW=dD.dot(X.T)
49 dX=W.T.dot(dD)
50 print(dW)
51 print(dX)
```

不用去记最后的表达，因为可以通过维度匹配推导出来，要记住的是，python 或者说 numpy 里面的 dot 就是矩阵乘法。