**[深度学习与计算机视觉系列(5)\_反向传播与它的直观理解](https://blog.csdn.net/yaoqiang2011/article/details/50321873)**

作者：[寒小阳](http://blog.csdn.net/han_xiaoyang?viewmode=contents)   
时间：2015年12月。   
出处：<http://blog.csdn.net/han_xiaoyang/article/details/50321873>   
声明：版权所有，转载请联系作者并注明出处

## 1. 引言

其实一开始要讲这部分内容，我是拒绝的，原因是我觉得有一种写高数课总结的感觉。而一般直观上理解反向传播算法就是求导的一个链式法则而已。但是偏偏理解这部分和其中的细节对于神经网络的设计和调整优化又是有用的，所以硬着头皮写写吧。

**问题描述与动机：**

* 大家都知道的，其实我们就是在给定的图像像素向量x和对应的函数*f*(*x*)

，然后我们希望能够计算*f*在*x*上的梯度(∇*f*(*x*)

 )

 我们之所以想解决这个问题，是因为在神经网络中，*f*

对应损失函数*L*，而输入*x*则对应训练样本数据和神经网络的权重*W*。举一个特例，损失函数可以是SVM loss function，而输入则对应样本数据(*xi*,*yi*),*i*=1…*N*和权重*W*以及偏移项*b*。需要注意的一点是，在我们的场景下，通常我们认为训练数据是给定的，而权重是我们可以控制的变量。因此我们为了更新权重的等参数，使得损失函数值最小，我们通常是计算*f*对参数*W*,*b*的梯度。不过我们计算其在*xi*

* 上的梯度有时候也是有用的，比如如果我们想做可视化以及了解神经网络在『做什么』的时候。

## 2.高数梯度/偏导基础

好了，现在开始复习高数课了，从最简单的例子开始，假如*f*(*x*,*y*)=*xy*

，那我们可以求这个函数对*x*和*y*

的偏导，如下：

*f*(*x*,*y*)=*xy*→∂*f*∂*x*=*y*∂*f*∂*y*=*x*

### 2.1 解释

我们知道偏导数实际表示的含义：一个函数在给定变量所在维度，当前点附近的一个变化率。也就是：

*df*(*x*)*dx*=lim*h* →0*f*(*x*+*h*)−*f*(*x*)*h*

以上公式中的*ddx*作用在*f*上，表示对x求偏导数，表示的是x维度上当前点位置周边很小区域的变化率。举个例子，如果*x*=4,*y*=−3，而*f*(*x*,*y*)=−12，那么x上的偏导∂*f*∂*x*=−3，这告诉我们如果这个变量(x)增大一个很小的量，那么整个表达式会以3倍这个量减小。我们把上面的公式变变形，可以这么看:*f*(*x*+*h*)=*f*(*x*)+*hdf*(*x*)*dx*。同理，因为∂*f*∂*y*=4，我们将y的值增加一个很小的量h，则整个表达式变化4h。

每个维度/变量上的偏导，表示整个函数表达式，在这个值上的『敏感度』

哦，对，我们说的梯度∇*f*

其实是一个偏导组成的向量，比如我们有∇*f*=[∂*f*∂*x*,∂*f*∂*y*]=[*y*,*x*]

。即使严格意义上来说梯度是一个向量，但是大多数情况下，我们还是习惯直呼『x上的梯度』，而不是『x上的偏导』

大家都知道加法操作上的偏导数是这样的：

*f*(*x*,*y*)=*x*+*y*→∂*f*∂*x*=1∂*f*∂*y*=1

而对于一些别的操作，比如max函数，偏导数是这样的(后面的括号表示在这个条件下)：

*f*(*x*,*y*)=max(*x*,*y*)→∂*f*∂*x*=1(*x*>=*y*)∂*f*∂*y*=1(*y*>=*x*)

## 3. 复杂函数偏导的链式法则

考虑一个麻烦一点的函数，比如*f*(*x*,*y*,*z*)=(*x*+*y*)*z*

。当然，这个表达式其实还没那么复杂，也可以直接求偏导。但是我们用一个非直接的思路去求解一下偏导，以帮助我们直观理解反向传播中。如果我们用换元法，把原函数拆成两个部分*q*=*x*+*y*和*f*=*qz*。对于这两个部分，我们知道怎么求解它们变量上的偏导：∂*f*∂*q*=*z*,∂*f*∂*z*=*q* ∂*q*∂*x*=1,∂*q*∂*y*=1，当然q是我们自己设定的一个变量，我们对他的偏导完全不感兴趣。   
那『链式法则』告诉我们一个对上述偏导公式『串联』的方式，得到我们感兴趣的偏导数:∂*f*∂*x*=∂*f*∂*q*∂*q*∂*x*

看个例子：

x = -2; y = 5; z = -4

# 前向计算

q = x + y # q becomes 3

f = q \* z # f becomes -12

# 类反向传播:

# 先算到了 f = q \* z

dfdz = q # df/dz = q

dfdq = z # df/dq = z

# 再算到了 q = x + y

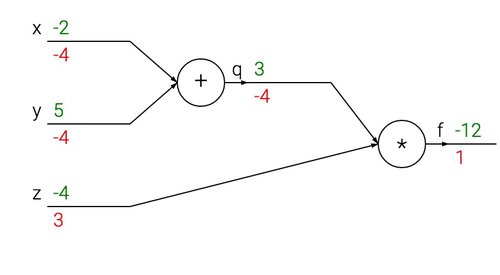
dfdx = 1.0 \* dfdq # dq/dx = 1 恩，链式法则

dfdy = 1.0 \* dfdq # dq/dy = 1

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

链式法则的结果是，只剩下我们感兴趣的[dfdx,dfdy,dfdz]，也就是原函数在x,y,z上的偏导。这是一个简单的例子，之后的程序里面我们为了简洁，不会完整写出dfdq，而是用dq代替。

以下是这个计算的示意图：



## 4. 反向传播的直观理解

一句话概括：反向传播的过程，实际上是一个由局部到全部的精妙过程。比如上面的电路图中，其实每一个『门』在拿到输入之后，都能计算2个东西：

* 输出值
* 对应输入和输出的局部梯度

而且很明显，每个门在进行这个计算的时候是完全独立的，不需要对电路图中其他的结构有了解。然而，在整个前向传输过程结束之后，在反向传播过程中，每个门却能逐步累积计算出它在整个电路输出上的梯度。『链式法则』告诉我们每一个门接收到后向传来的梯度，同时用它乘以自己算出的对每个输入的局部梯度，接着往后传。

以上面的图为例，来解释一下这个过程。加法门接收到输入[-2, 5]同时输出结果3。因为加法操作对两个输入的偏导都应该是1。电路后续的乘法部分算出最终结果-12。在反向传播过程中，链式法则是这样做的：加法操作的输出3，在最后的乘法操作中，获得的梯度为-4，如果把整个网络拟人化，我们可以认为这代表着网络『想要』加法操作的结果小一点，而且是以4\*的强度来减小。加法操作的门获得这个梯度-4以后，把它分别乘以本地的两个梯度(加法的偏导都是1)，1\*-4=-4。如果输入x减小，那加法门的输出也会减小，这样乘法输出会相应的增加。

反向传播，可以看做网络中门与门之间的『关联对话』，它们『想要』自己的输出更大还是更小(以多大的幅度)，从而让最后的输出结果更大。

## 5. Sigmoid例子

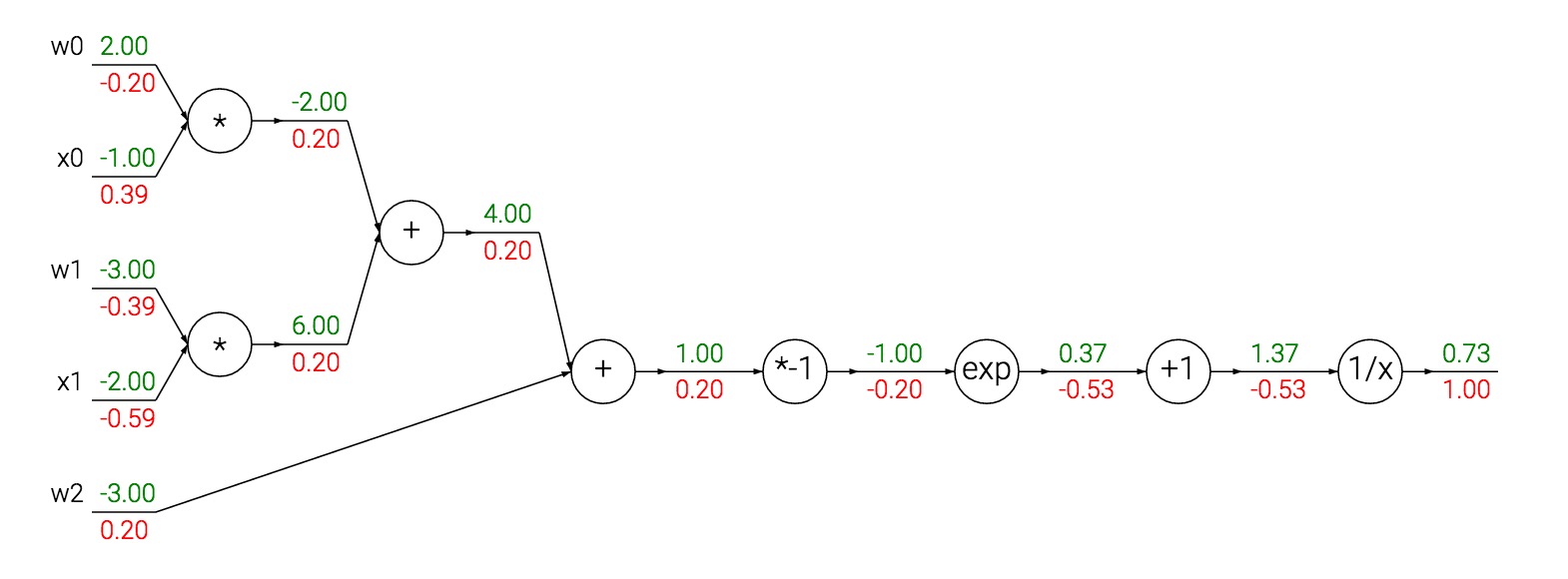
上面举的例子其实在实际应用中很少见，我们很多时候见到的网络和门函数更复杂，但是不论它是什么样的，反向传播都是可以使用的，唯一的区别就是可能网络拆解出来的门函数布局更复杂一些。我们以之前的逻辑回归为例：

*f*(*w*,*x*)=11+*e*−(*w*0*x*0+*w*1*x*1+*w*2)

这个看似复杂的函数，其实可以看做一些基础函数的组合，这些基础函数及他们的偏导如下：

*f*(*x*)=1*x*→*dfdx*=−1/*x*2*fc*(*x*)=*c*+*x*→*dfdx*=1*f*(*x*)=*ex*→*dfdx*=*exfa*(*x*)=*ax*→*dfdx*=*a*

上述每一个基础函数都可以看做一个门，如此简单的初等函数组合在一块儿却能够完成逻辑回归中映射函数的复杂功能。下面我们画出神经网络，并给出具体输入输出和参数的数值：



这个图中，[x0, x1]是输入，[w0, w1,w2]为可调参数，所以它做的事情是对输入做了一个线性计算(x和w的内积)，同时把结果放入sigmoid函数中，从而映射到(0,1)之间的数。

上面的例子中，w与x之间的内积分解为一长串的小函数连接完成，而后接的是sigmoid函数*σ*(*x*)

，有趣的是sigmoid函数看似复杂，求解倒是的时候却是有技巧的，如下：

*σ*(*x*)=11+*e*−*x*→*dσ*(*x*)*dx*=*e*−*x*(1+*e*−*x*)2=(1+*e*−*x*−11+*e*−*x*)(11+*e*−*x*)=(1−*σ*(*x*))*σ*(*x*)

你看，它的导数可以用自己很简单的重新表示出来。所以在计算导数的时候非常方便，比如sigmoid函数接收到的输入是1.0，输出结果是-0.73。那么我们可以非常方便地计算得到它的偏导为(1-0.73)\*0.73~=0.2。我们看看在这个sigmoid函数部分反向传播的计算代码：

w = [2,-3,-3] # 我们随机给定一组权重

x = [-1, -2]

# 前向传播

dot = w[0]\*x[0] + w[1]\*x[1] + w[2]

f = 1.0 / (1 + math.exp(-dot)) # sigmoid函数

# 反向传播经过该sigmoid神经元

ddot = (1 - f) \* f # sigmoid函数偏导

dx = [w[0] \* ddot, w[1] \* ddot] # 在x这条路径上的反向传播

dw = [x[0] \* ddot, x[1] \* ddot, 1.0 \* ddot] # 在w这条路径上的反向传播

# yes！就酱紫算完了！是不是很简单？

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

### 5.1 工程实现小提示

回过头看看上头的代码，你会发现，实际写代码实现的时候，有一个技巧能帮助我们很容易地实现反向传播，我们会把前向传播的过程分解成反向传播很容易追溯回来的部分。

## 6. 反向传播实战：复杂函数

我们看一个稍复杂一些的函数：

*f*(*x*,*y*)=*x*+*σ*(*y*)*σ*(*x*)+(*x*+*y*)2

插一句，这个函数没有任何实际的意义。我们提到它，仅仅是想举个例子来说明复杂函数的反向传播怎么使用。如果直接对这个函数求x或者y的偏导的话，你会得到一个很复杂的形式。但是如果你用反向传播去求解具体的梯度值的话，却完全没有这个烦恼。我们把这个函数分解成小部分，进行前向和反向传播计算，即可得到结果，前向传播计算的代码如下：

x = 3 # 例子

y = -4

# 前向传播

sigy = 1.0 / (1 + math.exp(-y)) # 单值上的sigmoid函数

num = x + sigy

sigx = 1.0 / (1 + math.exp(-x))

xpy = x + y

xpysqr = xpy\*\*2

den = sigx + xpysqr

invden = 1.0 / den

f = num \* invden # 完成！

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

注意到我们并没有一次性把前向传播最后结果算出来，而是刻意留出了很多中间变量，它们都是我们可以直接求解局部梯度的简单表达式。因此，计算反向传播就变得简单了：我们从最后结果往前看，前向运算中的每一个中间变量sigy, num, sigx, xpy, xpysqr, den, invden我们都会用到，只不过后向传回的偏导值乘以它们，得到反向传播的偏导值。反向传播计算的代码如下：

# 局部函数表达式为 f = num \* invden

dnum = invden

dinvden = num

# 局部函数表达式为 invden = 1.0 / den

dden = (-1.0 / (den\*\*2)) \* dinvden

# 局部函数表达式为 den = sigx + xpysqr

dsigx = (1) \* dden

dxpysqr = (1) \* dden

# 局部函数表达式为 xpysqr = xpy\*\*2

dxpy = (2 \* xpy) \* dxpysqr #(5)

# 局部函数表达式为 xpy = x + y

dx = (1) \* dxpy

dy = (1) \* dxpy

# 局部函数表达式为 sigx = 1.0 / (1 + math.exp(-x))

dx += ((1 - sigx) \* sigx) \* dsigx # 注意到这里用的是 += !!

# 局部函数表达式为 num = x + sigy

dx += (1) \* dnum

dsigy = (1) \* dnum

# 局部函数表达式为 sigy = 1.0 / (1 + math.exp(-y))

dy += ((1 - sigy) \* sigy) \* dsigy

# 完事！

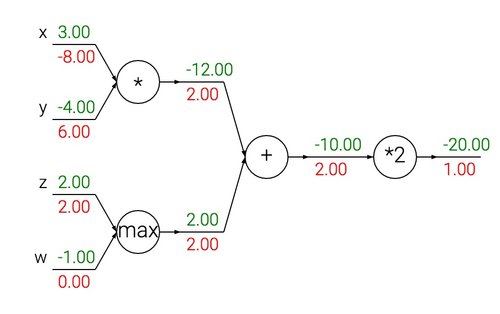
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

实际编程实现的时候，需要注意一下：

* **前向传播计算的时候注意保留部分中间变量**：在反向传播计算的时候，会再次用到前向传播计算中的部分结果。这在反向传播计算的回溯时可大大加速。

### 6.1 反向传播计算中的常见模式

即使因为搭建的神经网络结构形式和使用的神经元都不同，但是大多数情况下，后向计算中的梯度计算可以归到几种常见的模式上。比如，最常见的三种简单运算门(加、乘、最大)，他们在反向传播运算中的作用是非常简单和直接的。我们一起看看下面这个简单的神经网：



上图里有我们提到的三种门add,max和multiply。

* **加运算门**在反向传播运算中，不管输入值是多少，取得它output传回的梯度(gradient)然后均匀地分给两条输入路径。因为加法运算的偏导都是+1.0。
* **max(取最大)门**不像加法门，在反向传播计算中，它只会把传回的梯度回传给一条输入路径。因为max(x,y)只对x和y中较大的那个数，偏导为+1.0，而另一个数上的偏导是0。
* **乘法门**就更好理解了，因为x\*y对x的偏导为y，而对y的偏导为x，因此在上图中x的梯度是-8.0，即-4.0\*2.0

因为梯度回传的原因，神经网络对输入非常敏感。我们拿乘法门来举例，如果输入的*xi*

全都变成原来1000倍，而权重w不变，那么在反向传播计算的时候，x路径上获得的回传梯度不变，而w上的梯度则会变大1000倍，这使得你不得不降低学习速率(learning rate)成原来的1/1000以维持平衡。**因此在很多神经网络的问题中，输入数据的预处理也是非常重要的。**

### 6.2 向量化的梯度运算

上面所有的部分都是在单变量的函数上做的处理和运算，实际我们在处理很多数据(比如图像数据)的时候，维度都比较高，这时候我们就需要把单变量的函数反向传播扩展到向量化的梯度运算上，需要特别注意的是矩阵运算的每个矩阵维度，以及转置操作。

我们通过简单的矩阵运算来拓展前向和反向传播运算，示例代码如下：

# 前向传播运算

W = np.random.randn(5, 10)

X = np.random.randn(10, 3)

D = W.dot(X)

# 假如我们现在已经拿到了回传到D上的梯度dD

dD = np.random.randn(\*D.shape) # 和D同维度

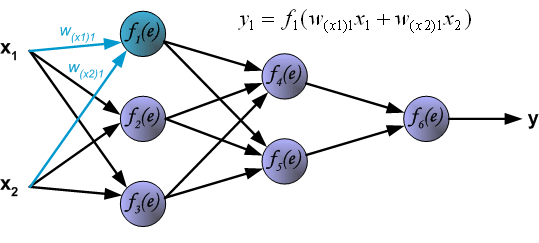
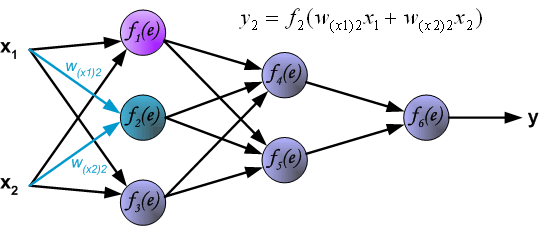
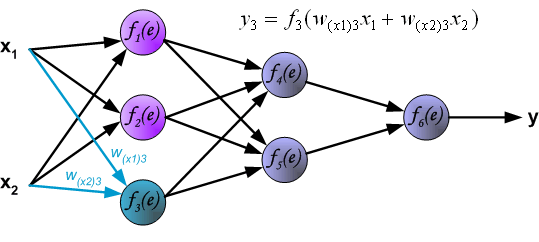
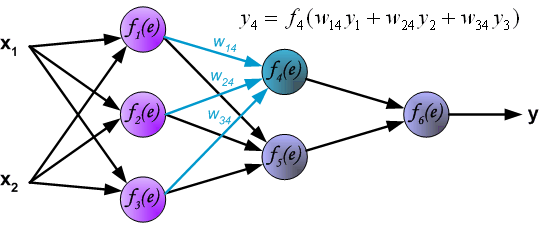
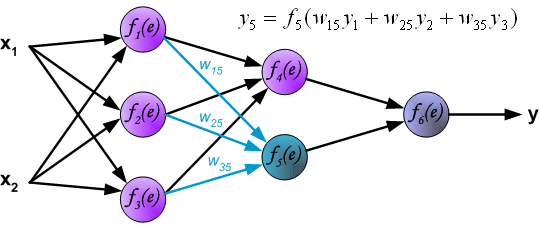
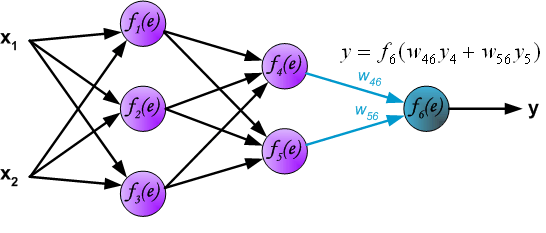
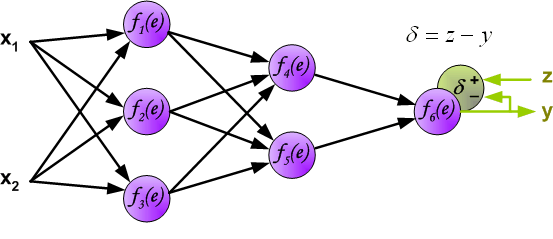
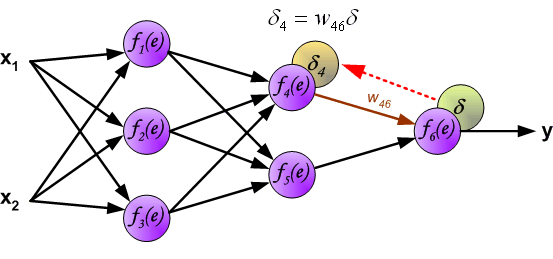
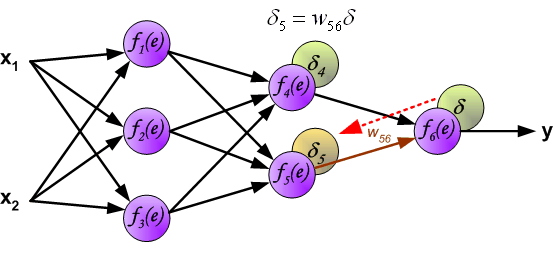
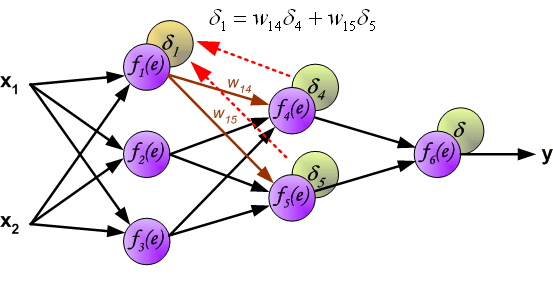
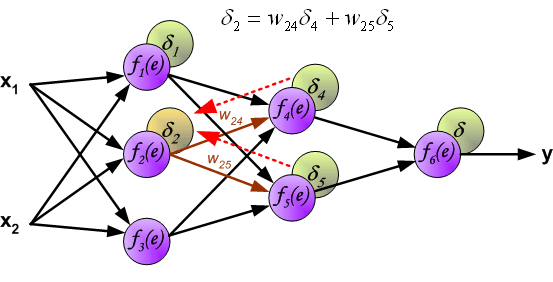
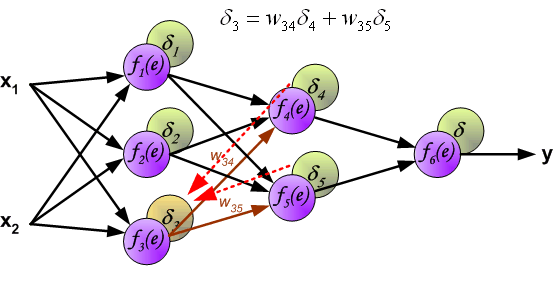
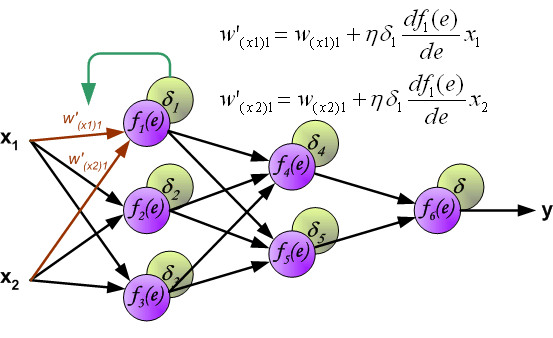
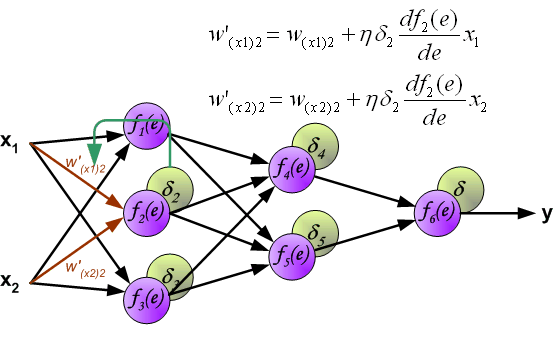
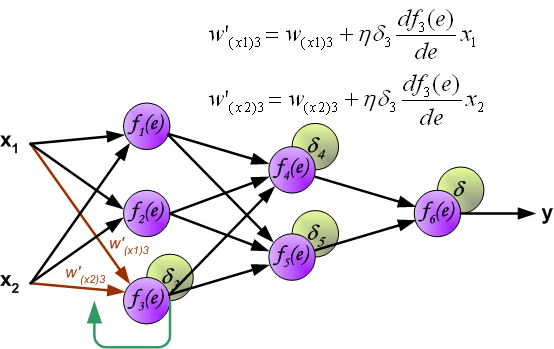
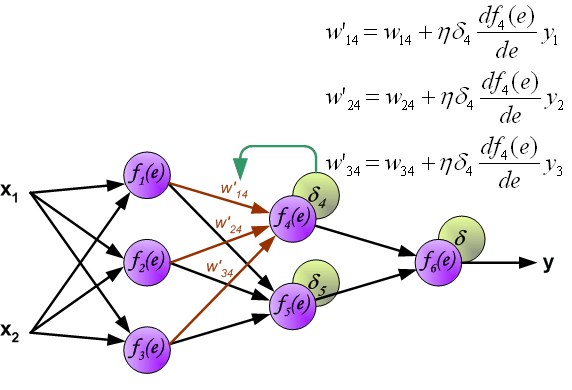
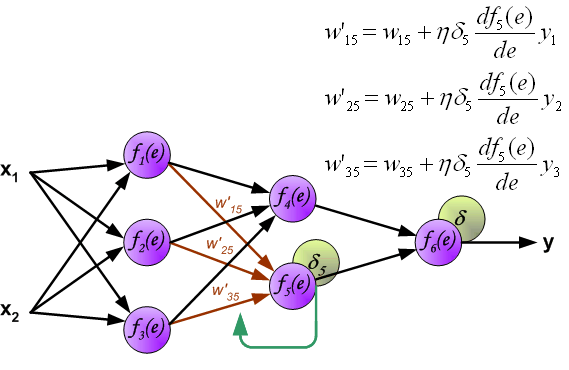
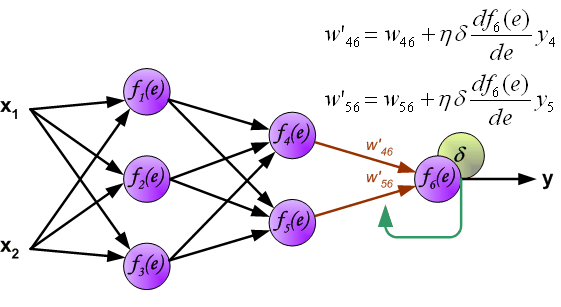
dW = dD.dot(X.T) #.T 操作计算转置， dW为W路径上的梯度

dX = W.T.dot(dD) #dX为X路径上的梯度

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

## 7. 总结

直观地理解，反向传播可以看做图解求导的链式法则。   
最后我们用一组图来说明实际优化过程中的正向传播与反向残差传播：

## 参考资料与原文

[cs231n 反向传播与直观理解](http://cs231n.github.io/optimization-2/)