深度学习

神经网络基于监督式学习的应用场景：
第一个例子还是房屋价格预测。根据训练样本的输入x和输出y，训练神经网络模型，预测房价。
第二个例子是线上广告，这是深度学习最广泛、最赚钱的应用之一。其中，输入x是广告和用户个人信息，输出y是用户是否对广告进行点击。神经网络模型经过训练，能够根据广告类型和用户信息对用户的点击行为进行预测，从而向用户提供用户自己可能感兴趣的广告。
第三个例子是电脑视觉（computer vision）。电脑视觉是近些年来越来越火的课题，而电脑视觉发展迅速的原因很大程度上是得益于深度学习。其中，输入x是图片像素值，输出是图片所属的不同类别。
第四个例子是语音识别（speech recognition）。深度学习可以将一段语音信号辨识为相应的文字信息。
第五个例子是智能翻译，例如通过神经网络输入英文，然后直接输出中文。除此之外，
第六个例子是自动驾驶。通过输入一张图片或者汽车雷达信息，神经网络通过训练来告诉你相应的路况信息并作出相应的决策。
对于不同的问题和应用场合，应该使用不同类型的神经网络模型。在上述例子中，对于一般的监督式学习（房价预测和线上广告问题），我们只要使用标准的神经网络模型就可以了。而对于图像识别处理问题，我们则要使用卷积神经网络（Convolution Neural Network），即CNN。而对于处理类似语音这样的序列信号时，则要使用循环神经网络（Recurrent Neural Network），即RNN。还有其它的例如自动驾驶这样的复杂问题则需要更加复杂的混合神经网络模型。
CNN一般处理图像问题，RNN一般处理语音信号。
数据类型一般分为两种：
Structured Data和Unstructured Data
简单地说，Structured Data通常指的是有实际意义的数据。例如房价预测中的size，#bedrooms，price等；例如在线广告中的User Age，Ad ID等。这些数据都具有实际的物理意义，比较容易理解。而Unstructured Data通常指的是比较抽象的数据，例如Audio，Image或者Text。以前，计算机对于Unstructured Data比较难以处理，而人类对Unstructured Data却能够处理的比较好，例如我们第一眼很容易就识别出一张图片里是否有猫，但对于计算机来说并不那么简单。现在，值得庆幸的是，由于深度学习和神经网络的发展，计算机在处理Unstructured Data方面效果越来越好，甚至在某些方面优于人类。总的来说，神经网络与深度学习无论对Structured Data还是Unstructured Data都能处理得越来越好，并逐渐创造出巨大的实用价值。
逻辑回归
这里以图片识别为例，如果判断出图片是只猫，输出为1，否则输出为0。计算机中保存一种图片，要保存三个独立矩阵，分别对应图片中红、绿、蓝三个颜色通道。如果输入图片是64X64像素的，就会有三个64X64的矩阵，分别对应红绿蓝三种像素的亮度。 接下来将这些亮度值放到一个特征向量中，就需要将矩阵中所有的像素值提取出来放到一个特征向量X中。则X的行数（总维

度）就是64x64x3=12288，列数为1，一般记做n_x。在二分类问题中目标是训练一个分类器，它以图片的特征向量X作为输入，预测结果输出的标签y是0还是1，也即预测图片中是否有猫。
m表示训练的样本数，$(x^{(i)}, y^{(i)})$表示样本i的输入和输出。
这样整个训练样本X维度为(n_x, m)。其中n_x表示每个样本$x^{(i)}$特征的个数，列m代表了样本的个数。

逻辑回归的损失函数的不能采用

$$Error = (h(x) - y)^2$$

因为这种损失函数通常是非凸函数，该类函数在使用梯度下降算法时容易得到局部最小值，即局部最优。所以一般选择凸函数来作为Loss Function。这里的Error为何不是凸函数呢？具体可以参照凸函数的定义来推导，简单点说对Error函数求二阶导数存在值使得二阶导数值小于0，所以可知是非凸函数。
$L(\hat{y}, y) = -(ylog\hat{y} + (1-y)log(1-\hat{y}))$
当y=1时$L(\hat{y}, y) = -log\hat{y}$。如果$\hat{y}$越接近1，$L(\hat{y}, y) \approx 0$，表示预测效果越好；如果ŷ 越接近0，$L(\hat{y}, y) \approx +\infty$，表示预测效果越差。这正是我们希望Loss function所实现的功能。

当y=0时，$L(\hat{y}, y) = -log\,(1-\hat{y})$。如果ŷ 越接近0，$L(\hat{y}, y) \approx 0$，表示预测效果越好；如果ŷ 越接近1，$L(\hat{y}, y) \approx +\infty$，表示预测效果越差。这也正是我们希望Loss function所实现的功能。
那究竟该损失函数是怎么来的呢?
首先，预测输出$\hat{y}$的表达式可以写成：

$$\hat{y} = \sigma(w^T x + b)$$

其中，$\sigma(z) = \frac{1}{1+e^{-z}}$,$\hat{y}$可以看成是预测输出为正类（+1）的概率：

$$\hat{y} = P(y=1|x)$$

当y=1时：

$$P(y|x) = \hat{y}$$

当y=0时：

$$P(y=1|x) = 1 - \hat{y}$$

将上面两个式子合并可知：

$$P(x|y) = \hat{y}^y(1-\hat{y})^{1-y}$$

由于log函数的单调性，可以对上式P(y|x)进行log处理：

$$logP(x|y) = log(\hat{y}^{y}(1-\hat{y})^{1-y}) = ylog\hat{y} + (1-y)log(1-\hat{y})$$

我们希望上述概率P(y|x)越大越好，对上式加上负号，则转化成了单个样本的Loss function，越小越好，也就得到了我们之前介绍的逻辑回归的Loss function形式。

$$L = -(ylog\hat{y} + (1-y)log(1-\hat{y}))$$

如果对于所有m个训练样本，假设样本之间是独立同分布的（iid），我们希望总的概率越大越好：

$$max \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$$

同样引入log，添加负号，就转化成了Cost Function：

$$J(w,b) = -\frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log\hat{y}^{(i)} + (1-y^{(i)})log(1-\hat{y}^{(i)}))$$

逻辑回归问题可以看做一个简单的神经网络，只包含一个神经元。

Computation graph

整个神经网络的训练过程实际上包含了两个过程：正向传播（Forward Propagation）和反向传播（Back Propagation）。正向传播是从输入到输出，由神经网络计算得到预测输出的过程；反向传播是从输出到输入，对参数w和b计算梯度的过程。

激活函数选择：

分类问题：输出层的激活函数一般会选择sigmoid函数；隐藏层的激活函数通常不会选择sigmoid函数，tanh函数的表现会比sigmoid函数好一些。实际应用中，通常会会选择使用ReLU或者Leaky ReLU函数，保证梯度下降速度不会太小。

卷积神经网络CNN

https://www.zybuluo.com/Dounm/note/591752

介绍：最初是为了解决图像识别等问题设计的，起初的图像识别研究，最大的挑战是如何组织特征，因为图像数据不像其他类型的数据可以通过人工理解来提取特征。

循环神经网络RNN

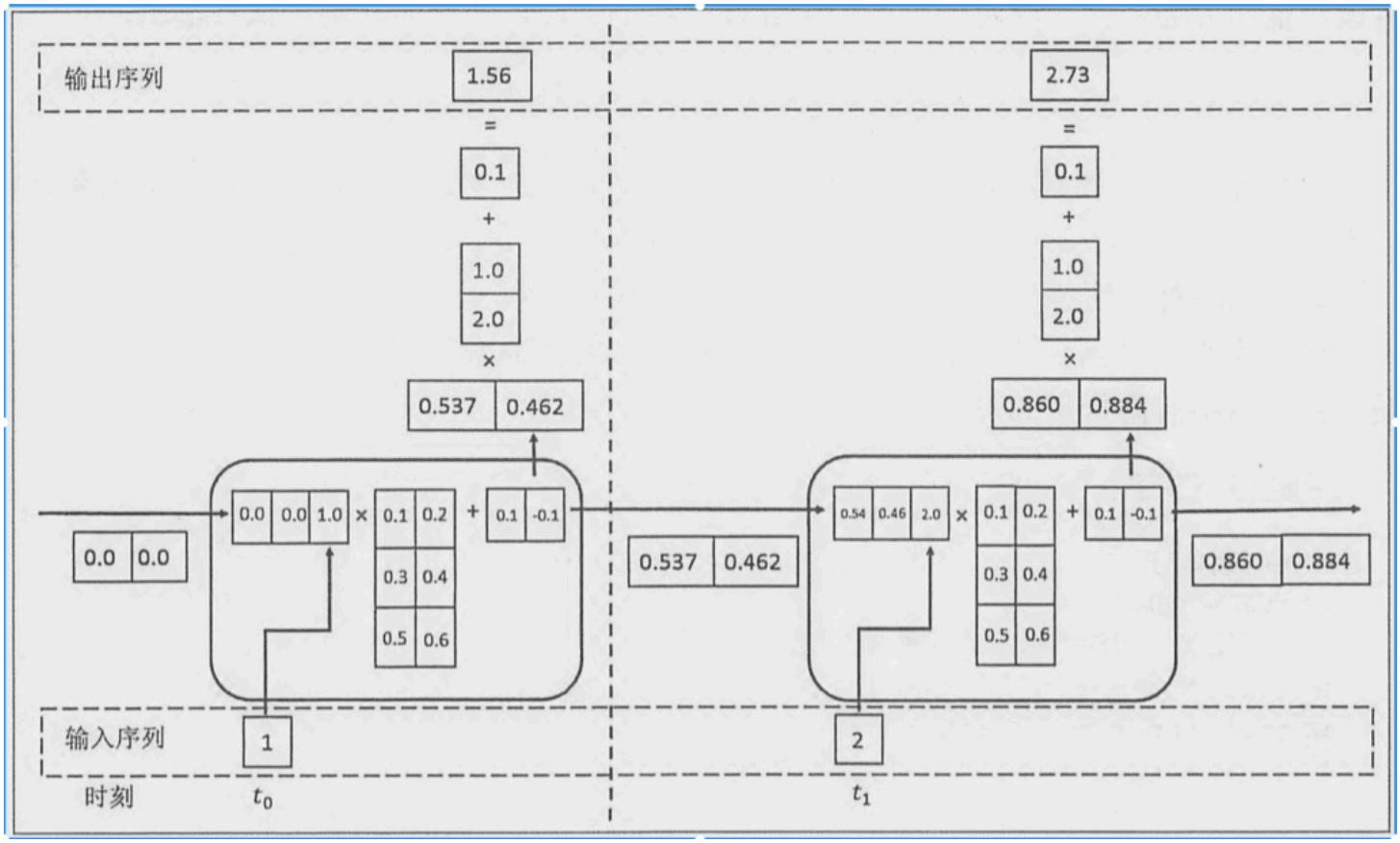
RNN原理解析英文版

RNN原理解析中文翻译版

循环神经网络图解

前向传播的具体过程：

前向传播是指对神经网络沿着从输入层到输出层的顺序，依次计算并存储模型中间变量的过程。



这里假设状态的维度是2，这里初始状态向量：[0.0, 0.0]。输入和输出的维度都为1，循环体中的全连接层的权重：

$$w_{rnn} = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{pmatrix}$$

偏置项的大小为$b_{rnn} = [0.1, -0.1]$
用于输出的全连接层的权重为：

$$w_{output} = \begin{pmatrix} 1.0 \\ 2.0 \end{pmatrix}$$

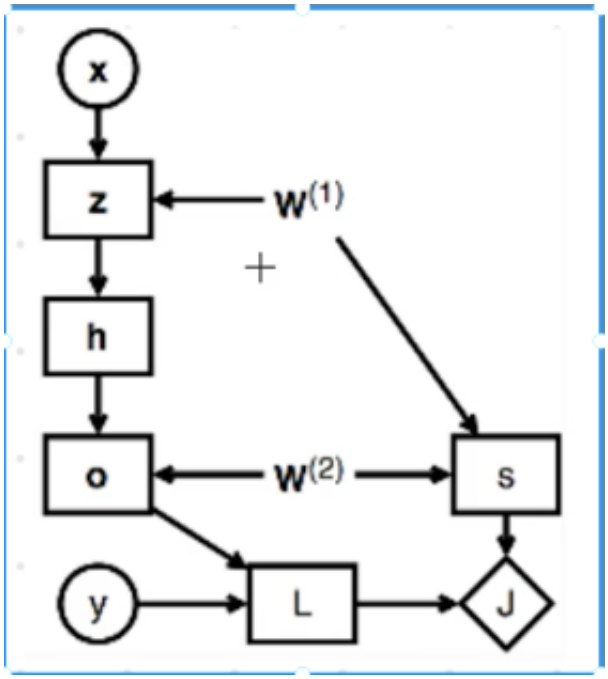
偏置项大小为$b_{output} = 0.1$。
那么在时刻t_0，由于没有上一时刻，所以将状态初始化为$[0, 0]$，而当前的输入为1，所以拼接得到的向量为$[0, 0, 1]$（这里状态和输入采用的是同样的权重，所以这里将其拼接在一起直接与权重向量求积）。 通过循环体中的全连接层神经网络得到的结果是：

$$tanh\left([0,0,1]\times\begin{bmatrix}0.1 & 0.2\\ 0.3 & 0.4\\ 0.5 & 0.6\end{bmatrix}+[0.1,-0.1]\right)=tanh\left([0.6,0.5]\right)=[0.537,0.462]$$

该结果一方面作为下一时刻的输入状态，另一方面循环神经网络也会使用该状态生成输出，就可以得到t_0时刻的输出：

$$[0.537, 0.462] \times \begin{bmatrix} 1.0 \\ 2.0 \end{bmatrix} + 0.1 = 1.56$$

反向传播是计算深度学习模型参数梯度的方法。反向传播会依据微积分中的链式法则，沿着从输出层到输入层的顺序，依次计算并存储损失函数有关神经网络各层的中间变量以及参数的梯度。（反向传播时，计算有关各层变量和参数的梯度可能会依赖于各层变量和参数的当前值。而这些变量的当前值来自正向传播的计算结果）



输入特征项为x和标签为离散值的y'
该模型没有考虑偏置项，中间变量$z = W^{(1)}x$

> $z : h \times 1$
> $W^{(1)} : h \times x$
> $x : x \times 1$

经过按元素操作的激活函数ϕ之后，就可以得到向量长度为h的隐藏层变量$\boldsymbol{h} = \phi(z)$。同样隐藏层h同样是中间变量。接下来通过模型参数（权重）$W^{(2)}$可得到想来那个长度为y的输出层变量$\boldsymbol{o} = \boldsymbol{W}^{(2)}\boldsymbol{h}$。

> $o : y \times 1$
> $W^{(2)} : y \times h$
> $h : h \times 1$

假设损失函数为ℓ，我们可以计算出单个数据样本的损失项:

$$L = \ell(\boldsymbol{o}, y')$$

根据$L2$范数正则化的定义，给定超参数λ，正则化项即

$$s = \frac{\lambda}{2}(\|\boldsymbol{W}^{(1)}\|_F^2 + \|\boldsymbol{W}^{(2)}\|_F^2)$$

其中每个矩阵Frobenius范数的平方项即该矩阵元素的平方和。最终，模型在给定的数据样本上带正则化的损失为:

$$J = L + s$$

我们将J称为有关给定数据样本的目标函数。

从右到左进行链式求导进行反向传播:
反向传播的计算过程需要把到达某一节点的所有链路求和。

1、首先计算目标函数有关损失项和正则项的梯度:

$$\frac{\partial J}{\partial L} = 1$$

$$\frac{\partial J}{\partial s} = 1$$

2、依据链式法则计算目标函数有关输出层的梯度$\frac{\partial J}{\partial o}$:

$$\frac{\partial J}{\partial o} = prod(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial o}) = \frac{\partial L}{\partial o}$$

正则项有关两个参数的梯度:

> $L2$正则化（岭回归），防止过拟合
> $\|W\|$表示向量范数（或向量的模），即表征向量空间中向量的大小

$$s = \frac{\lambda}{2}(\|W^{(1)}\|_2^2 + \|W^{(2)}\|_2^2)$$

$$\frac{\partial s}{\partial W^{(1)}} = \lambda W^{(1)}$$

$$\frac{\partial s}{\partial W^{(2)}} = \lambda W^{(2)}$$

3、接着计算最靠近输出层的模型参数的梯度

$$\frac{\partial J}{\partial W^{(2)}}$$

，从图示可以看出J分别通过o和s依赖$W^{(2)}$。根据链式法则，可以得到：

$$\frac{\partial J}{\partial W^{(2)}} = prod(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial W^{(2)}}) + prod(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(2)}}) = \frac{\partial L}{\partial o} h^T + \lambda W^{(2)}$$

> J对o求导，向量维度是y x 1; J是个标量，对$W^{(2)}$求导，必然保留向量维度，即y x h
> 那么必然可知：o对$W^{(2)}$求导维度为1 x h
> 由正向传播$\boldsymbol{o} = \boldsymbol{W}^{(2)}\boldsymbol{h}$知道，$o$对$W^{(2)}$求导必然是跟$h$相关的,所以此处是$h$的转置

4、沿着输出层继续反向传播，隐藏层变量的梯度$\frac{\partial J}{\partial h}$：

$$\frac{\partial J}{\partial h} = prod(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial h}) = W^{(2)^T} \frac{\partial J}{\partial o}$$

注意到激活函数ϕ是按元素操作的，中间变量z的梯度$\frac{\partial J}{\partial h}$的计算要按照元素乘法符$\odot$:

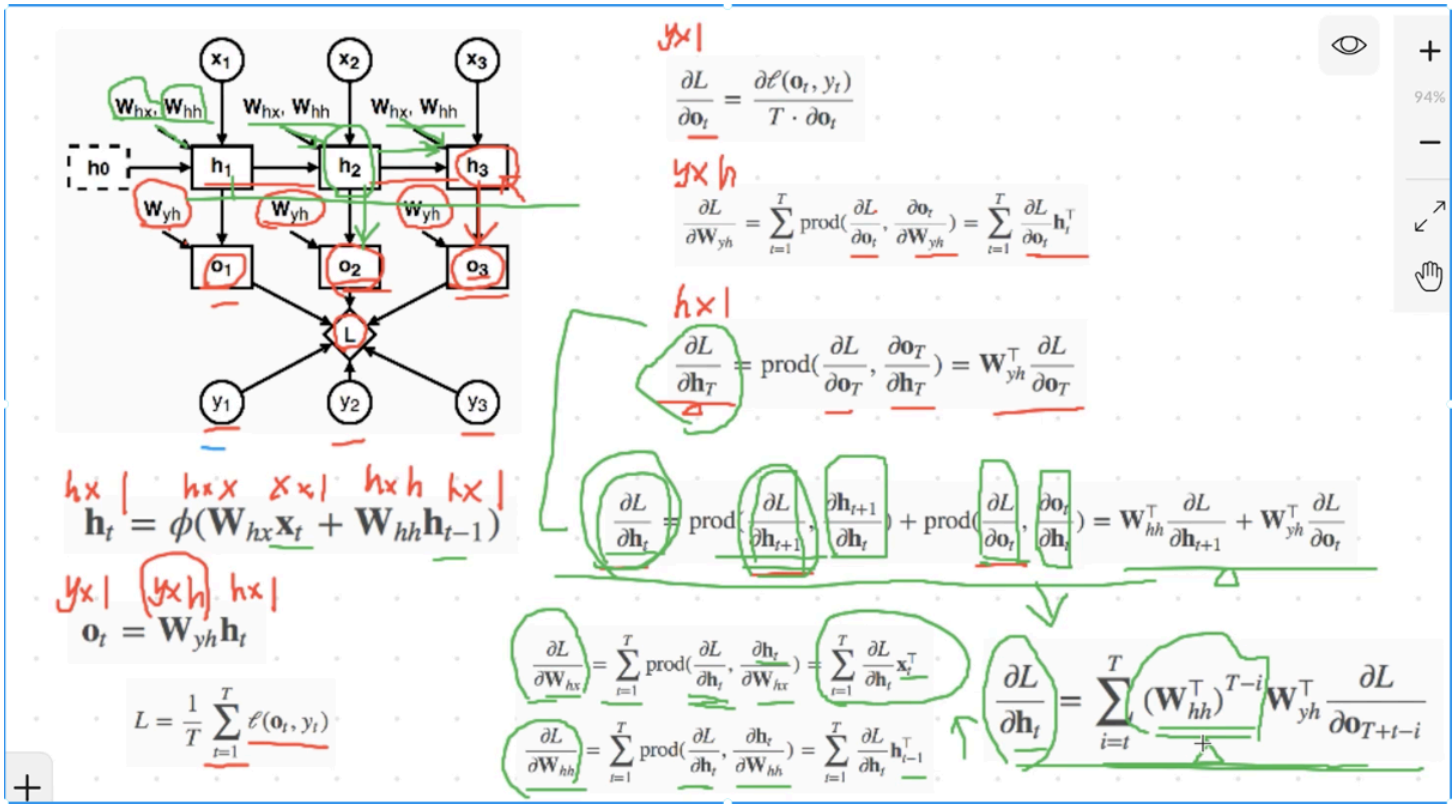
$$\frac{\partial J}{\partial \boldsymbol{z}} = \text{prod}(\frac{\partial J}{\partial \boldsymbol{h}}, \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}) = \frac{\partial J}{\partial \boldsymbol{h}} \odot \phi'(\boldsymbol{z})$$

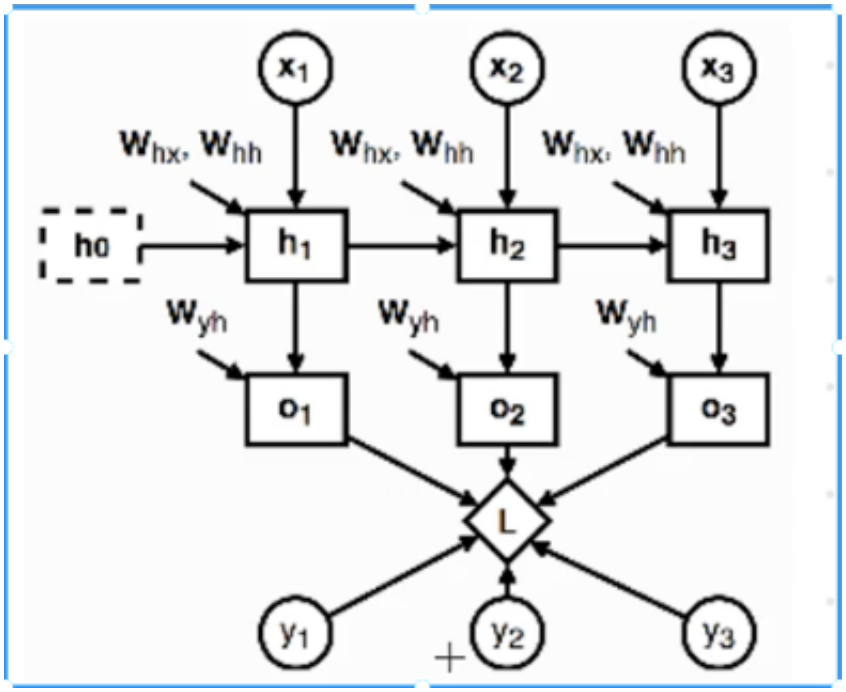
5、最终我们可以得到最靠近输入层的模型参数的梯度$\partial J/\partial \boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times x}$。在图示中，$J$分别通过$z$和$s$依赖$W^{(1)}$。依据链式法则：

$$\frac{\partial J}{\partial \boldsymbol{W}^{(1)}} = \text{prod}(\frac{\partial J}{\partial \boldsymbol{z}}, \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}^{(1)}}) + \text{prod}(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \boldsymbol{W}^{(1)}}) = \frac{\partial J}{\partial \boldsymbol{z}} \boldsymbol{x}^\top + \lambda \boldsymbol{W}^{(1)}$$

每次迭代中，上述各个依次计算出的梯度会被依次存储或更新。这是为了避免重复计算。例如，由于输出层变量梯度$\partial J/\partial \boldsymbol{o}$被计算存储，反向传播稍后的参数梯度$\partial J/\partial \boldsymbol{W}^{(2)}$和隐藏层变量梯度$\partial J/\partial \boldsymbol{h}$的计算可以直接读取输出层变量梯度的值，而无需重复计算。

通过时间反向传播是反向传播在循环神经网络中的具体应用。是将循环神经网络按时间步展开，从而得到模型变量和参数之间的依赖关系，并依赖链式法则应用反向传播计算并存储梯度。
下图所示同样没有考虑偏置项，同时激活函数的输入和输出相同。



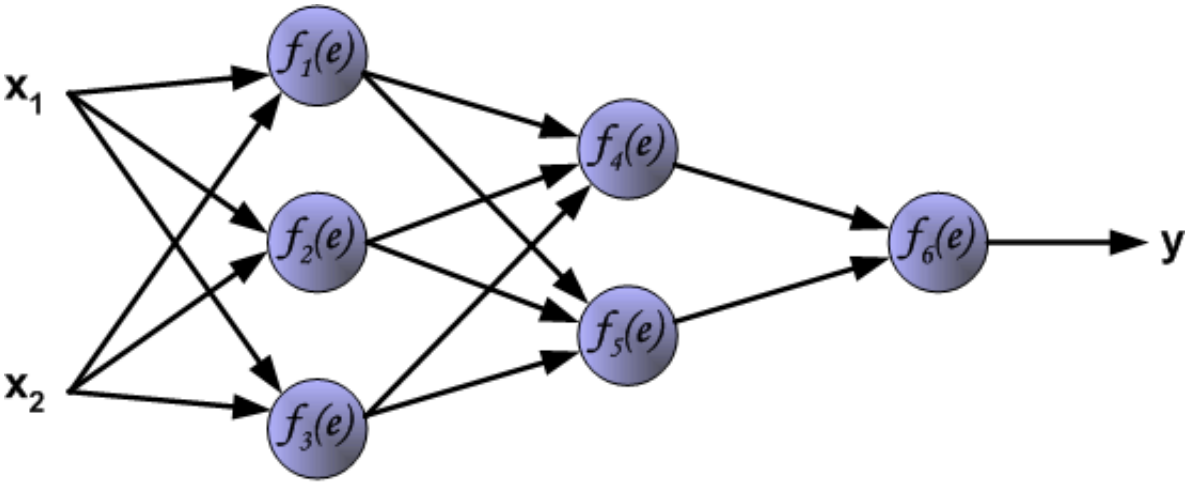


这里前一个状态和输入对应的权重设置为不同的值：比如W_{hx}和W_{hh}分别表示输入x到h_1的权重和前一状态h_0到h_1的权重。当然这里对于同一个神经元也可以采用同一个权重（可参照《Tensorflow+实战Google深度学习框架》中神经网络的描述）。

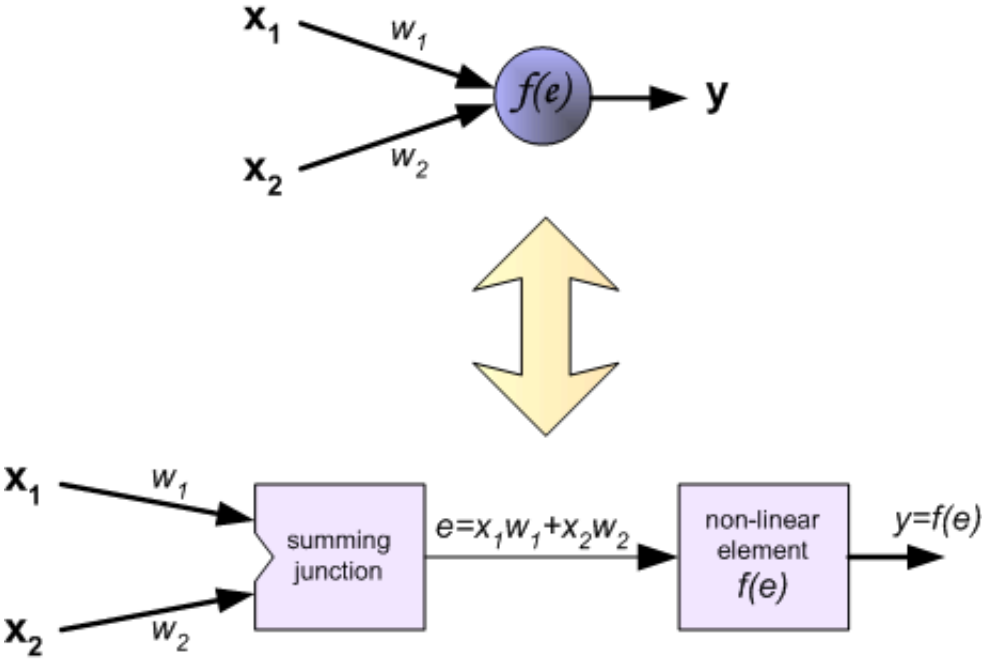
每一个神经元对应一个激活函数，对前一状态和输入进行非线性变换，最终的向量维度和输入的参数向量维度是一样的。

使用反向传播训练多层神经网络的原则：

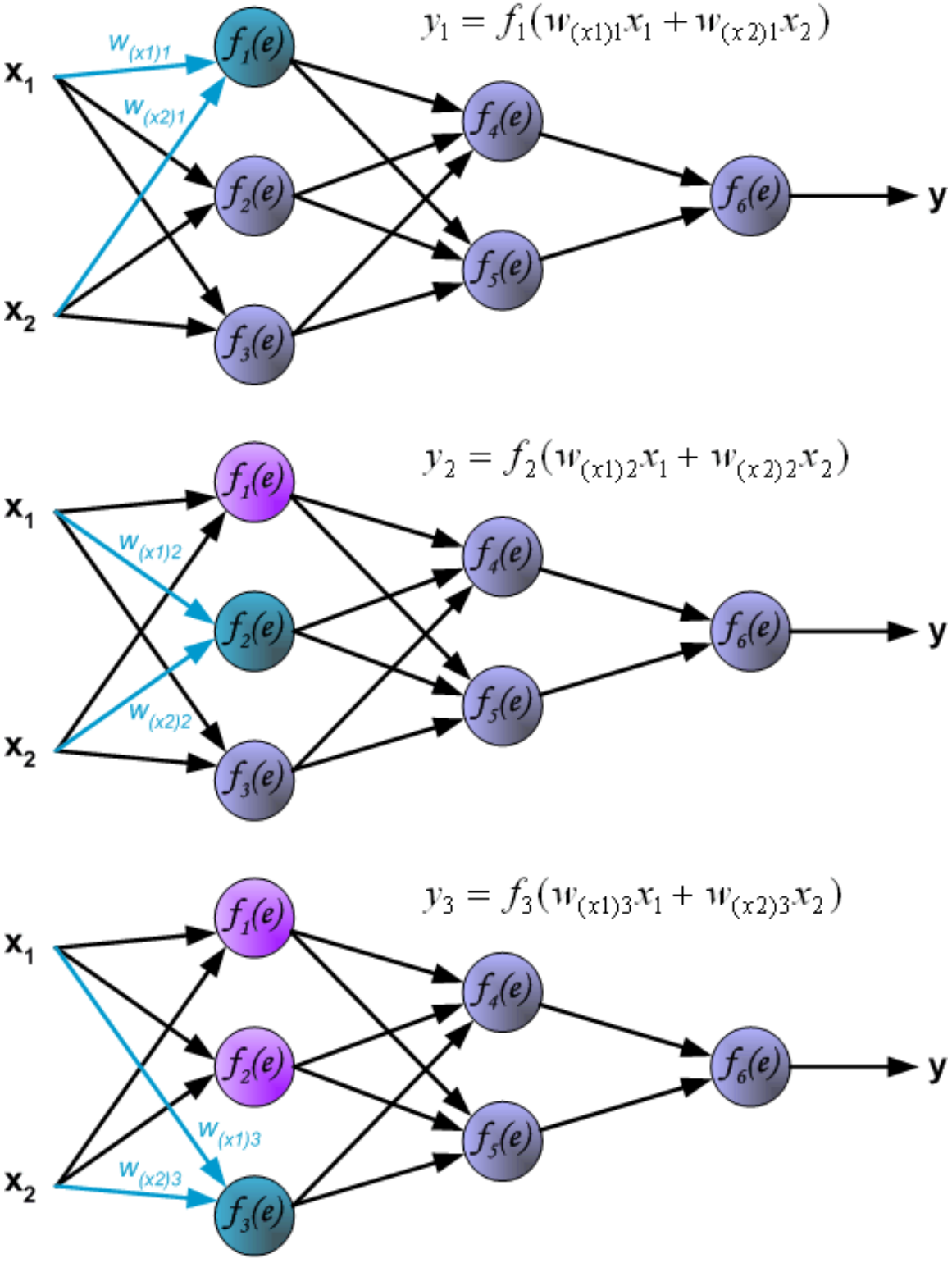
接下来采用三层神经网络来描述正向传播和反向传播的过程，该神经网络包含两个输入和一个输出，如下图所示：



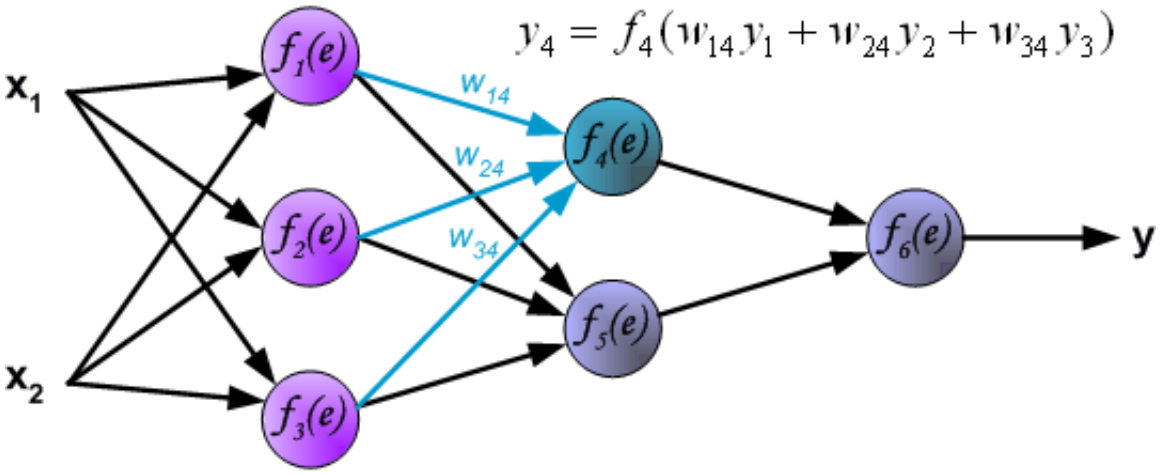
每一个神经元包含两个单元。第一个单元添加权重系数和输入信号的乘积，第二个单元提供一个激活函数，实现非线性变换。信号e是加法器的输出信号，而$y = f(e)$则是非线性单元的输出信号，同时y也是神经元的输出信号。

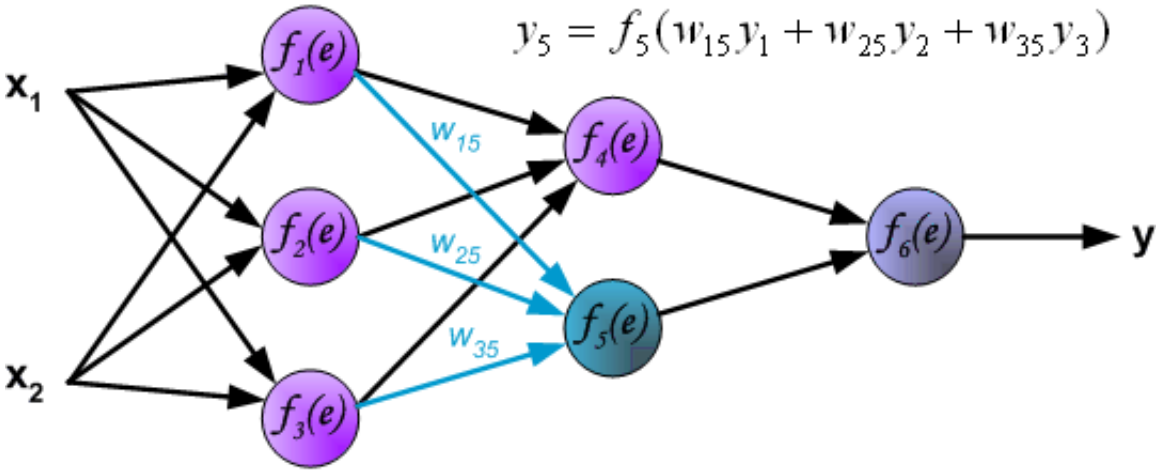


为了训练该神经网络，我们首先需要准备训练数据集。训练数据集包含输入信号x_1和x_2，同时包含对应的期望输出值z。神经网络训练的过程实际上是个迭代的过程，每一次的迭代，节点的权重系数都会根据训练数据集新的数据进行修改（更新）。具体的修改过程采用的算法如下：如下图示展示了信号是如何在神经网络中传播的，信号$w_{(xm)n}$表示输入层的输入x_m和神经元n之间的权重系数，信号y_n表示神经元n的输出信号。

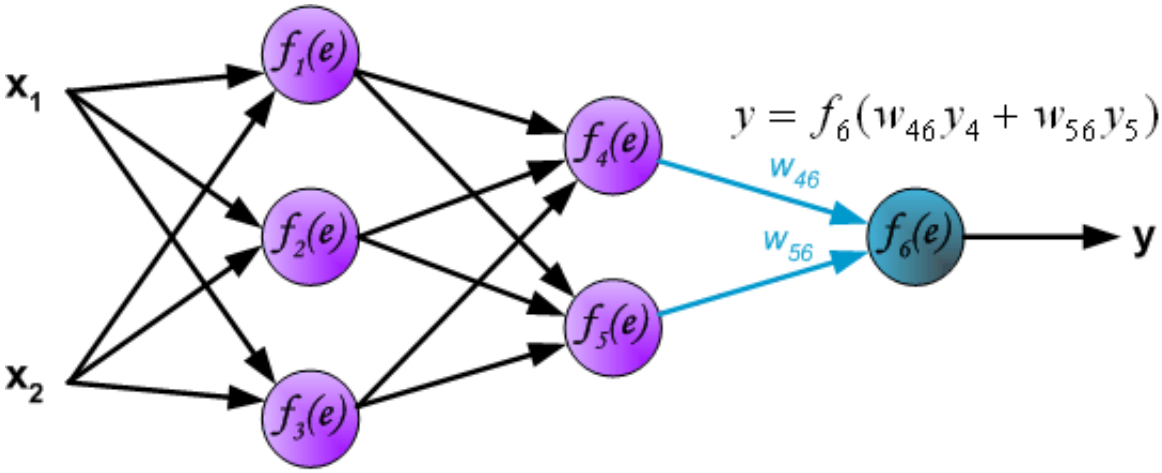


隐藏层的信号传播。信号$w_{(xm)n}$表示下一层的输出神经元m和输入神经元n之间的权重系数。

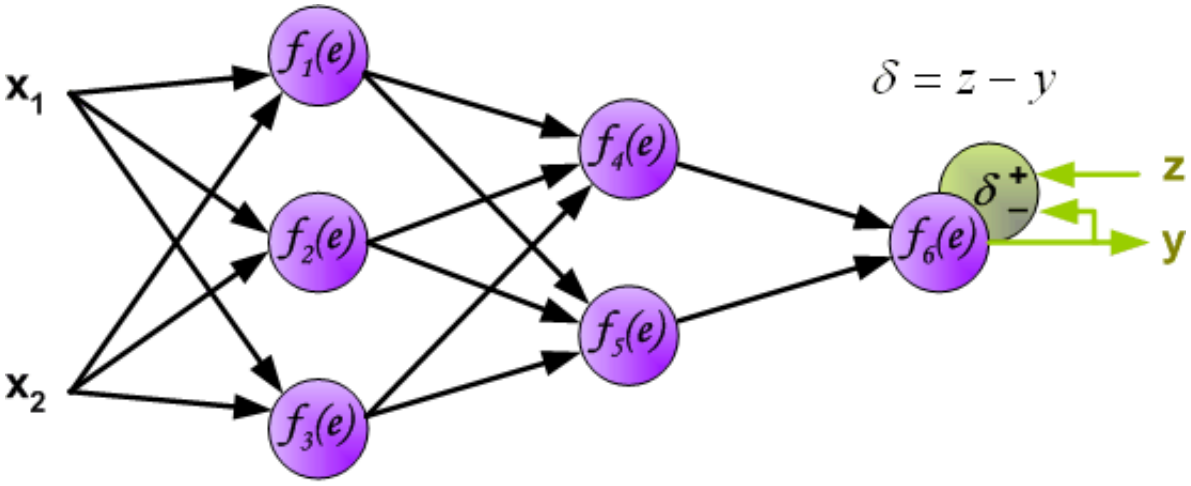




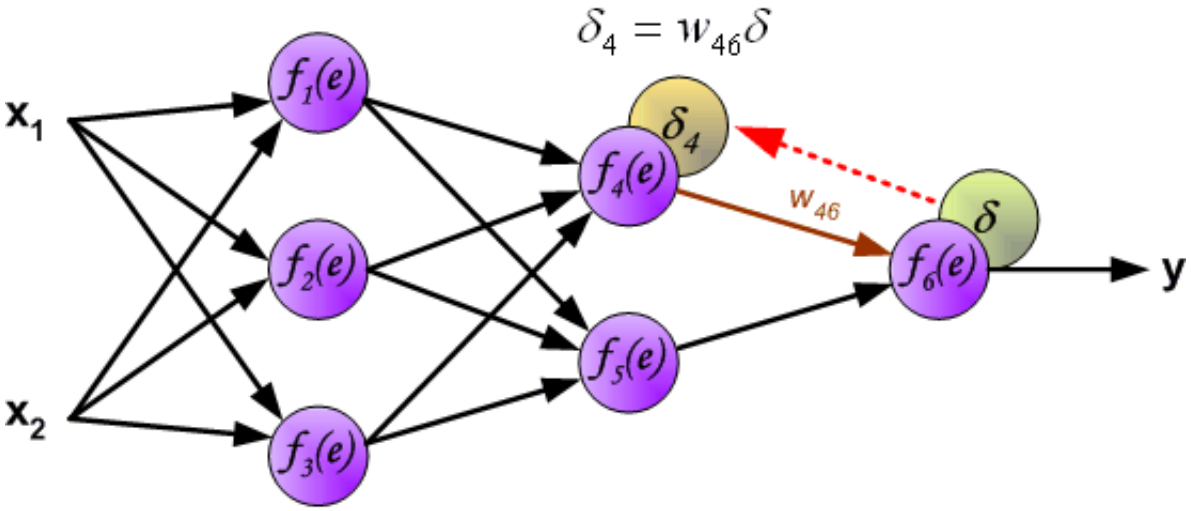
输出层的信号传播

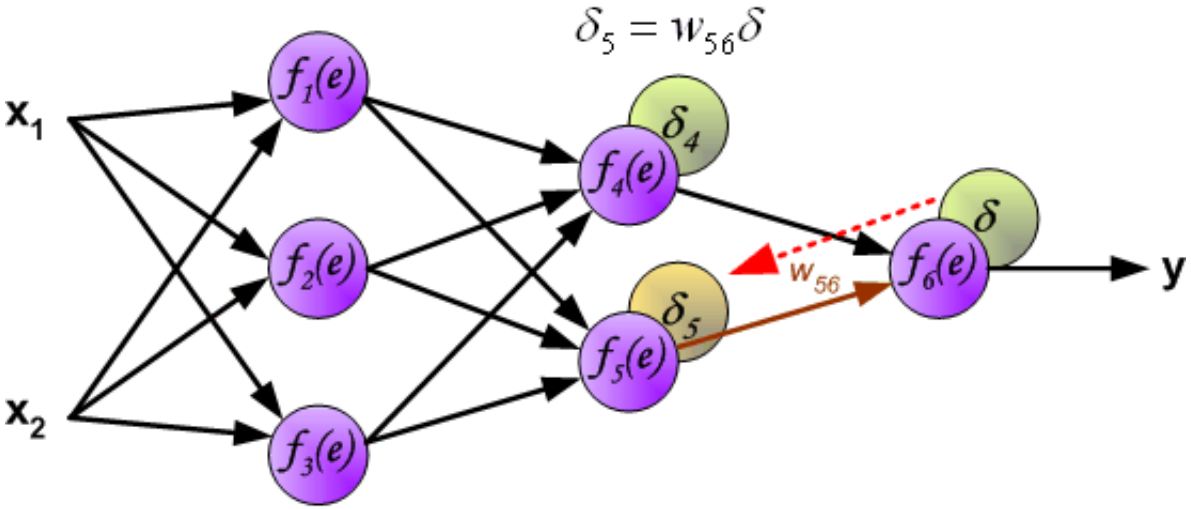


第二步的算法过程中，网络的输出信号y会跟期望的输出值z进行比较，期望的输出值z来自于训练集。这种差异被称为误差error。

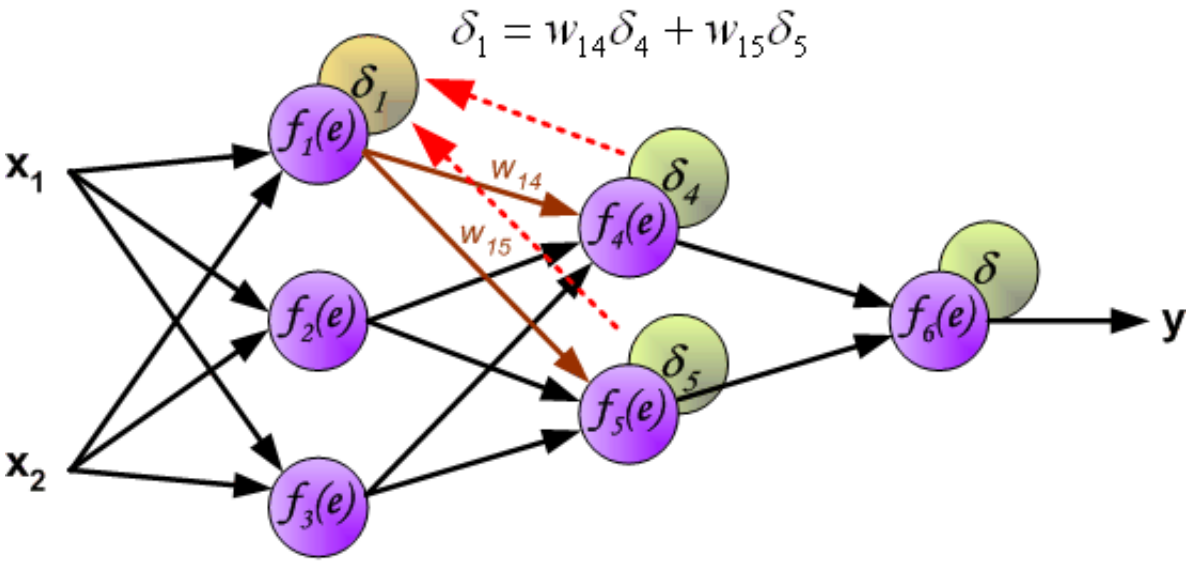


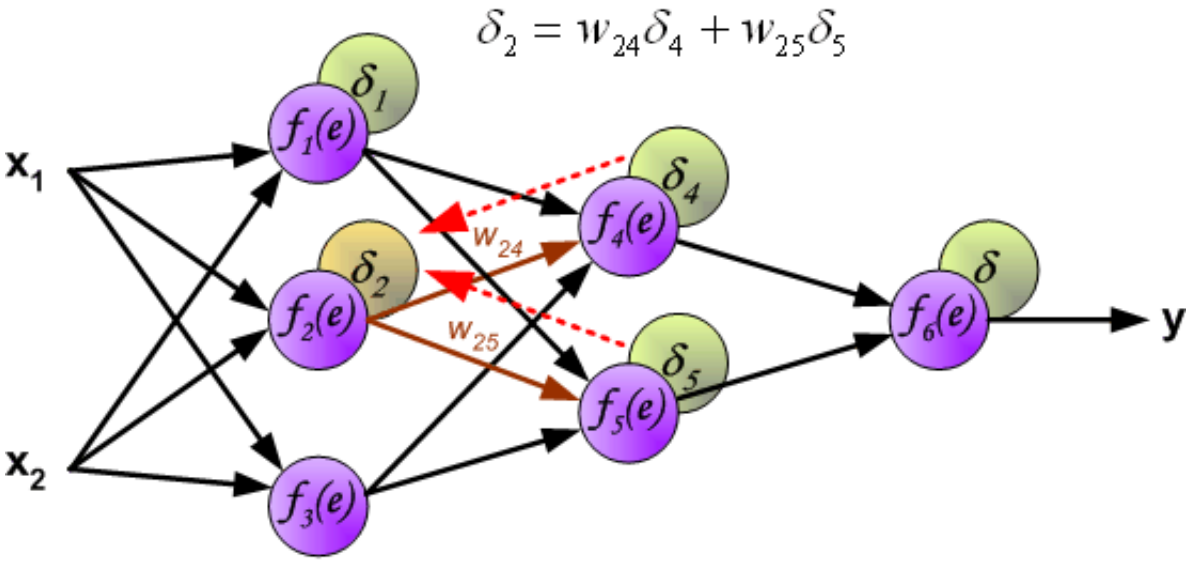
我们没有办法直接通过内部神经元去计算误差信号，因为这些神经元的输出值是不知道的。很多年来训练多层神经网络的有效方法都没有找到，到了80年代中期反向传播算法才开始起作用。其思想就是反向将误差信号传递给所有的神经元，也即当前的输出信号变成了输入信号。

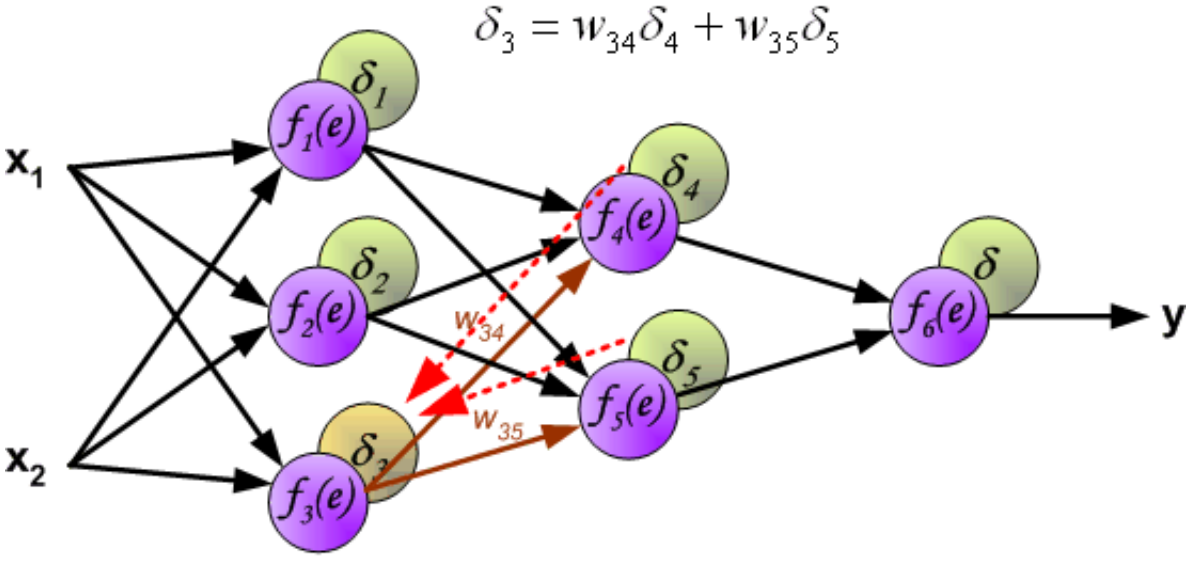




权重系数w_{mn}用于反向传播误差信号和之前的正向计算输出信号的过程是类似的，唯一的不同就是数据流的方向是反的。该方法被应用于所有的网络层。如果传播的误差是来自于多个神经元，则将这些误差进行求和。



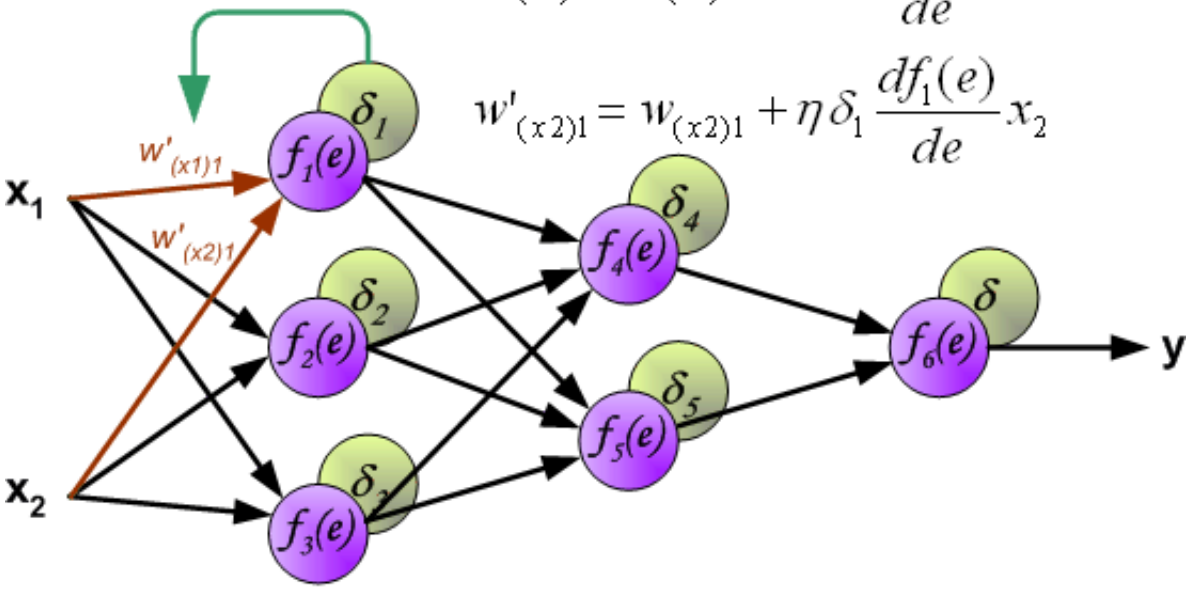




当每一个神经元的误差都计算完之后，每一个神经元输入节点的权重系数也相应的更新了。下面图示的公式$\frac{df(d)}{de}$表示神经元激活函数的求导（权重发生了改变）。

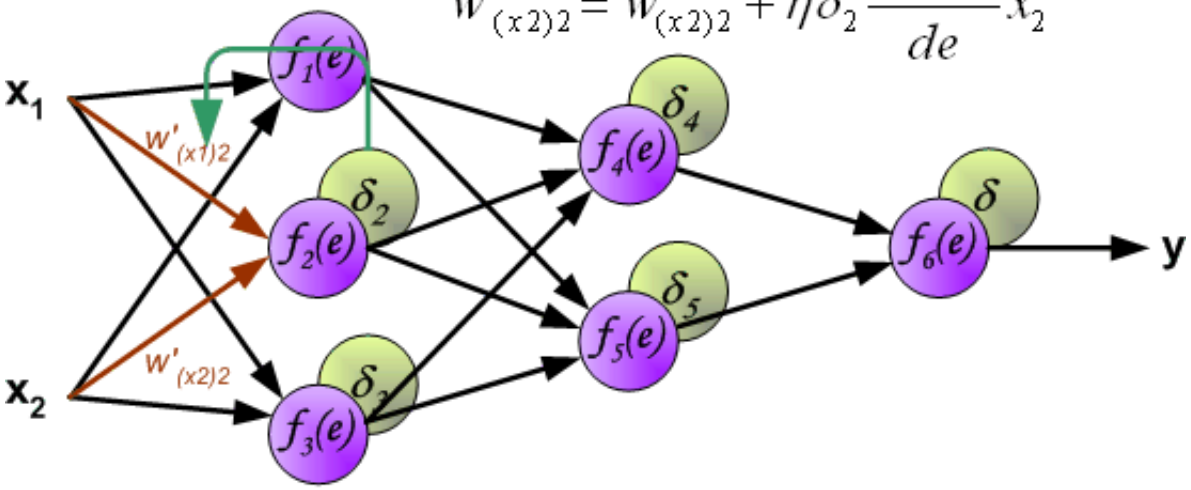
$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$



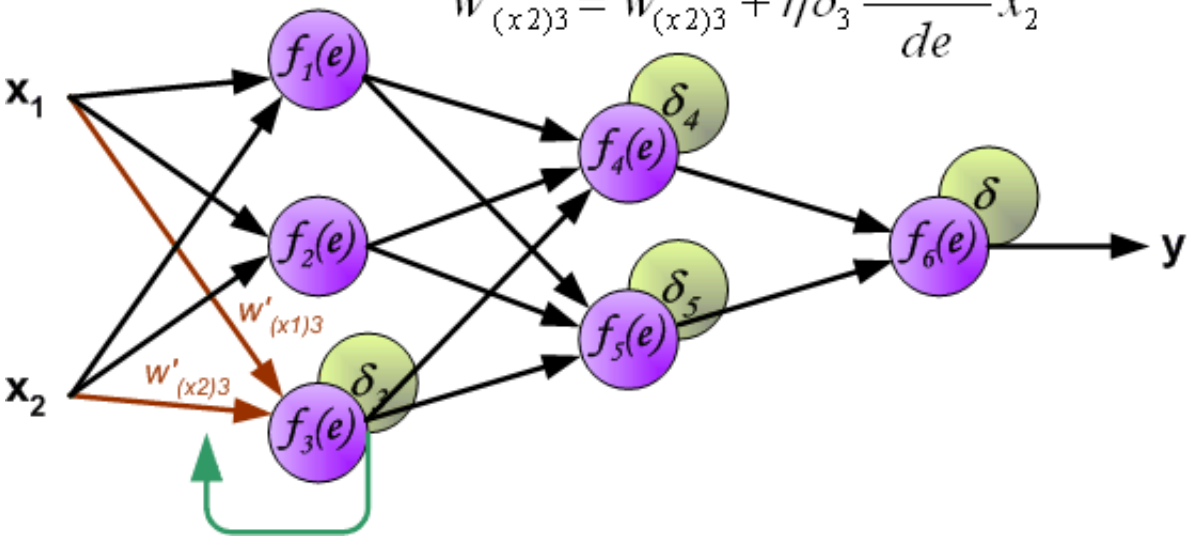
$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 \frac{df_2(e)}{de} x_2$$



$$w'_{(x1)3} = w_{(x1)3} + \eta \delta_3 \frac{df_3(e)}{de} x_1$$

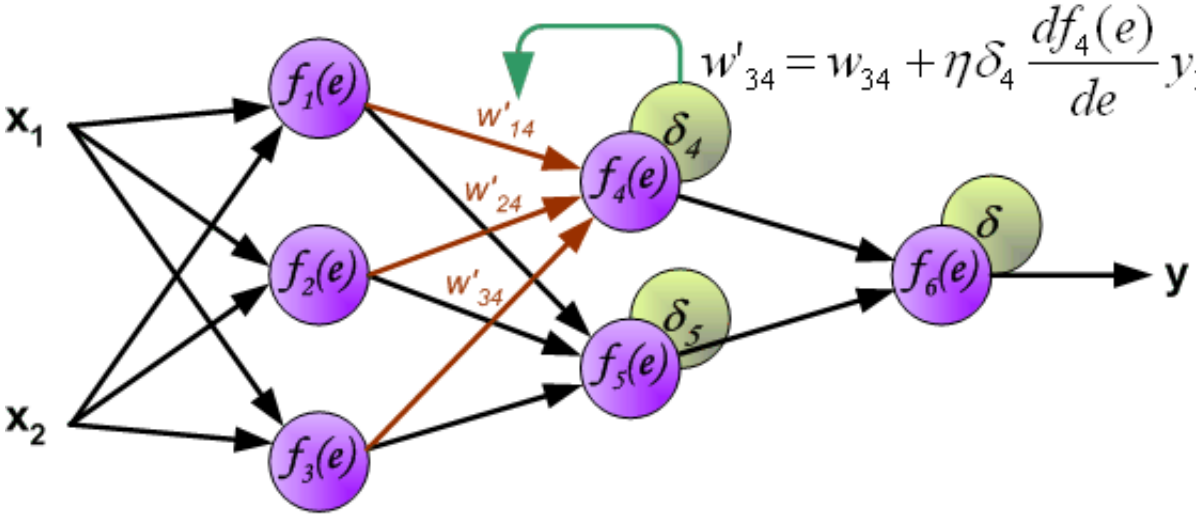
$$w'_{(x2)3} = w_{(x2)3} + \eta \delta_3 \frac{df_3(e)}{de} x_2$$



$$w'_{14} = w_{14} + \eta \delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta \delta_4 \frac{df_4(e)}{de} y_2$$

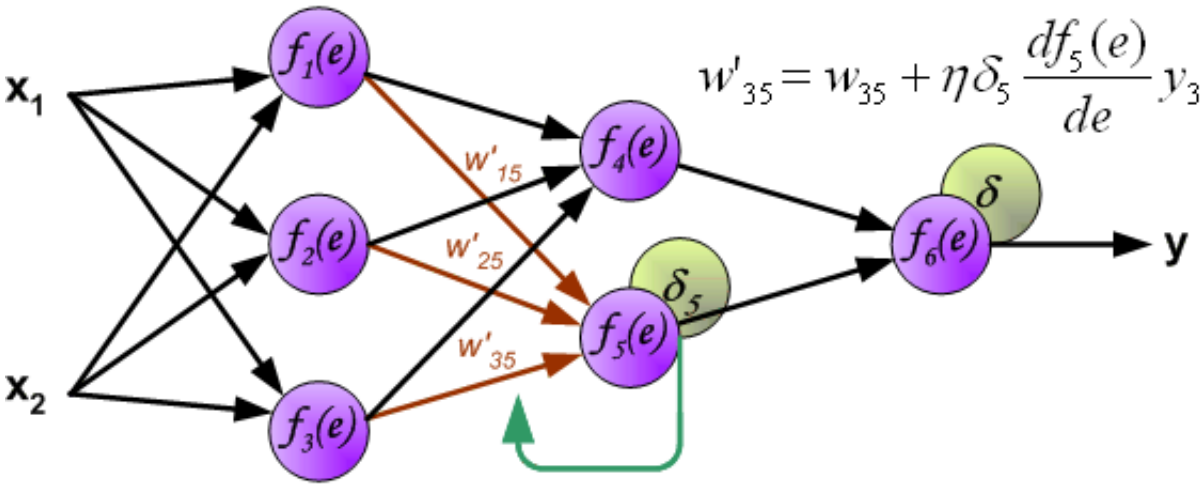
$$w'_{34} = w_{34} + \eta \delta_4 \frac{df_4(e)}{de} y_3$$



$$w'_{15} = w_{15} + \eta\delta_5 \frac{df_5(e)}{de} y_1$$

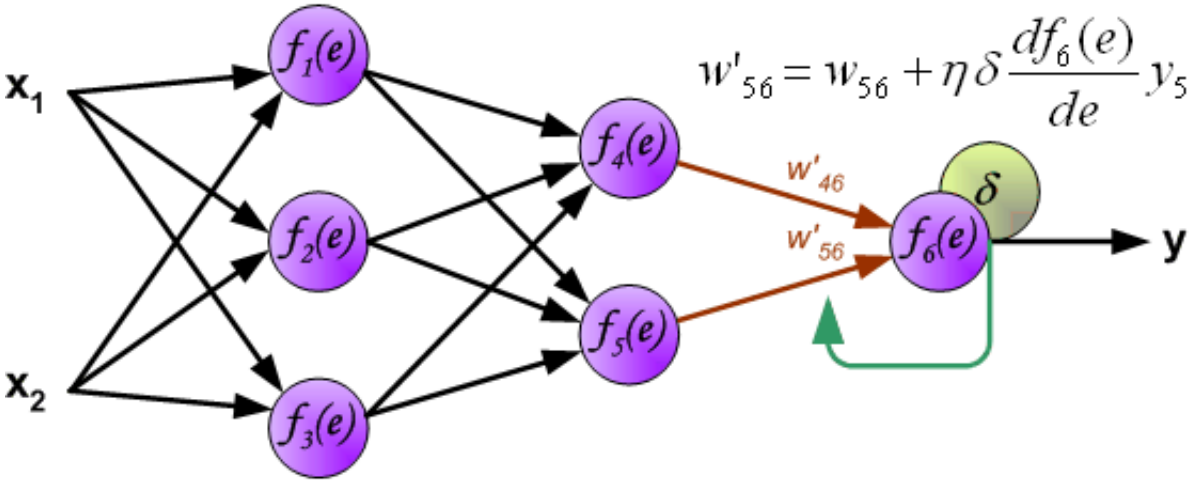
$$w'_{25} = w_{25} + \eta\delta_5 \frac{df_5(e)}{de} y_2$$

$$w'_{35} = w_{35} + \eta\delta_5 \frac{df_5(e)}{de} y_3$$



$$w'_{46} = w_{46} + \eta\delta \frac{df_6(e)}{de} y_4$$

$$w'_{56} = w_{56} + \eta\delta \frac{df_6(e)}{de} y_5$$



问题：

1、在循环神经网络的训练中，当每个时序训练数据样本的时序长度seq_len较大或者时刻t较小，目标函数有关t时刻的隐含层变量梯度较容易出现衰减（valishing）或者爆炸（explosion）。为了应对这一现象，一个常用的做法就是如果梯度特别大，那么久投射到一个比较小的尺度上。假设我们把所有的梯度接成一个向量g，假设裁剪的阈值是θ，那么我们这样裁剪使得$\|g\|$不会超过θ：

$$g = min(\frac{\theta}{\|g\|}, 1)g$$

```
def grad_clipping(params, theta):
    norm = nd.array([0.0], ctx)
    for p in params:
        norm += nd.sum(p.grad ** 2)
    norm = nd.sqrt(norm).asscalar()
    if norm > theta:
        for p in params:
            p.grad[:] *= theta / norm
```

如果不对梯度做裁剪，会报OverflowError，为什么呢?

梯度消失

2、为何要添加偏置bias，其作用是什么?

LSTM神经网络的理解

LSTM结构的核心就是“输入门”和“遗忘门”。

正则化和过拟合

L1正则化：

L2正则化：