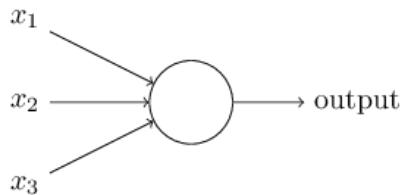


## 深度神经网络 ( DNN ) 模型与前向传播算法

深度神经网络 ( Deep Neural Networks , 以下简称DNN ) 是深度学习的基础，而要理解DNN，首先我们要理解DNN模型，下面我们就对DNN的模型与前向传播算法做一个总结。

### 1. 从感知机到神经网络

在感知机原理小结中，我们介绍过感知机的模型，它是一个有若干输入和一个输出的模型，如下图：



输出和输入之间学习到一个线性关系，得到中间输出结果：

$$z = \sum_{i=1}^m w_i x_i + b$$

接着是一个神经元激活函数：

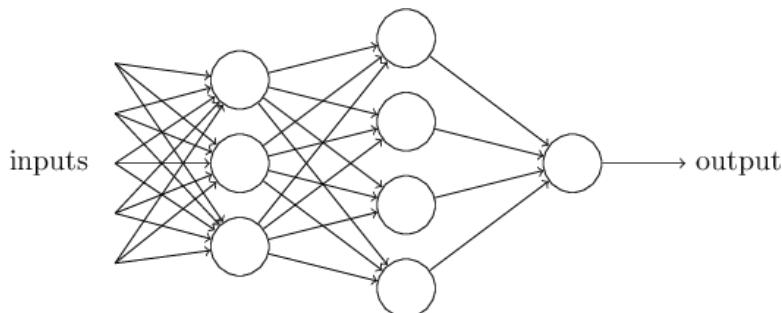
$$\text{sign}(z) = \begin{cases} -1 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

从而得到我们想要的输出结果1或者-1。

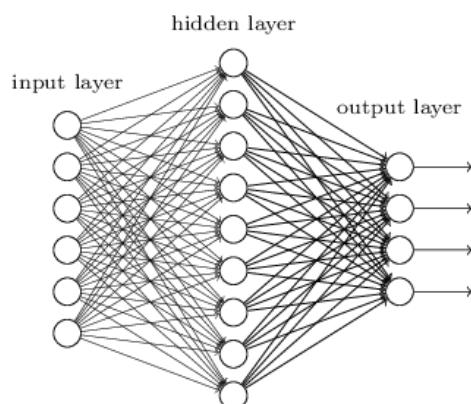
这个模型只能用于二元分类，且无法学习比较复杂的非线性模型，因此在工业界无法使用。

而神经网络则在感知机的模型上做了扩展，总结下主要有三点：

1 ) 加入了隐藏层，隐藏层可以有多层，增强模型的表达能力，如下图实例，当然增加了这么多隐藏层模型的复杂度也增加了好多。



2 ) 输出层的神经元也可以不止一个输出，可以有多个输出，这样模型可以灵活的应用于分类回归，以及其他的机器学习领域比如降维和聚类等。多个神经元输出的输出层对应的一个实例如下图，输出层现在有4个神经元了。



3 ) 对激活函数做扩展，感知机的激活函数是 $sign(z)$ ，虽然简单但是处理能力有限，因此神经网络中一般使用的其他的激活函数，比如我们在逻辑回归里面使用过的Sigmoid函数，即：

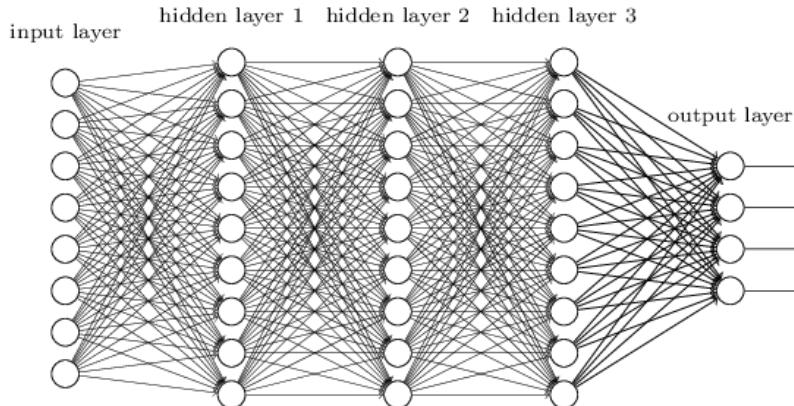
$$f(z) = \frac{1}{1 + e^{-z}}$$

还有后来出现的tanx, softmax, 和ReLU等。通过使用不同的激活函数，神经网络的表达能力进一步增强。对于各种常用的激活函数，我们在后面再专门讲。

## 2. DNN的基本结构

上一节我们了解了神经网络基于感知机的扩展，而DNN可以理解为有很多隐藏层的神经网络。这个很多其实也没有什么度量标准，多层神经网络和深度神经网络DNN其实也是指的一个东西，当然，DNN有时也叫做多层感知机（Multi-Layer perceptron, MLP），名字实在是多。后面我们讲到的神经网络都默认为DNN。

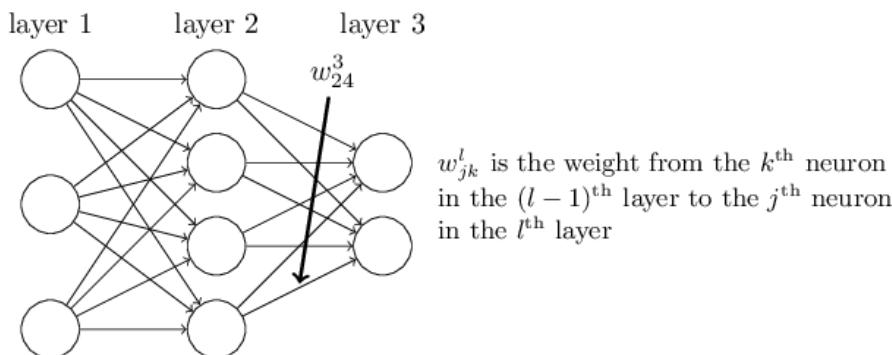
从DNN按不同层的位置划分，DNN内部的神经网络层可以分为三类，输入层，隐藏层和输出层，如下图示例，一般来说第一层是输入层，最后一层是输出层，而中间的层数都是隐藏层。



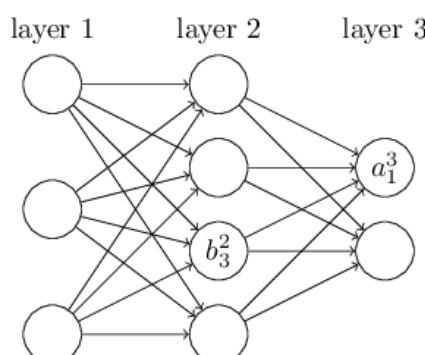
层与层之间是全连接的，也就是说，第*i*层的任意一个神经元一定与第*i*+1层的任意一个神经元相连。虽然DNN看起来很复杂，但是从小的局部模型来说，还是和感知机一样，即一个线性关系 $z = \sum w_i x_i + b$ 加上一个激活函数 $\sigma(z)$ 。

由于DNN层数多，则我们的线性关系系数 $w$ 和偏倚 $b$ 的数量也就是很多了。具体的参数在DNN是如何定义的呢？

首先我们来看看线性关系系数 $w$ 的定义。以下图一个三层的DNN为例，第二层的第4个神经元到第三层的第2个神经元的线性系数定义为 $w_{24}^3$ 。上标3代表线性系数 $w$ 所在的层数，而下标对应的是输出的第三层索引2和输入的第二层索引4。你也许会问，为什么不是 $w_{42}^3$ ，而是 $w_{24}^3$ 呢？这主要是为了便于模型用于矩阵表示运算，如果是 $w_{42}^3$ 而每次进行矩阵运算是 $w^T x + b$ ，需要进行转置。将输出的索引放在前面的话，则线性运算不用转置，即直接为 $wx + b$ 。总结下，第*l*-1层的第*k*个神经元到第*l*层的第*j*个神经元的线性系数定义为 $w_{jk}^l$ 。注意，输入层是没有 $w$ 参数的。

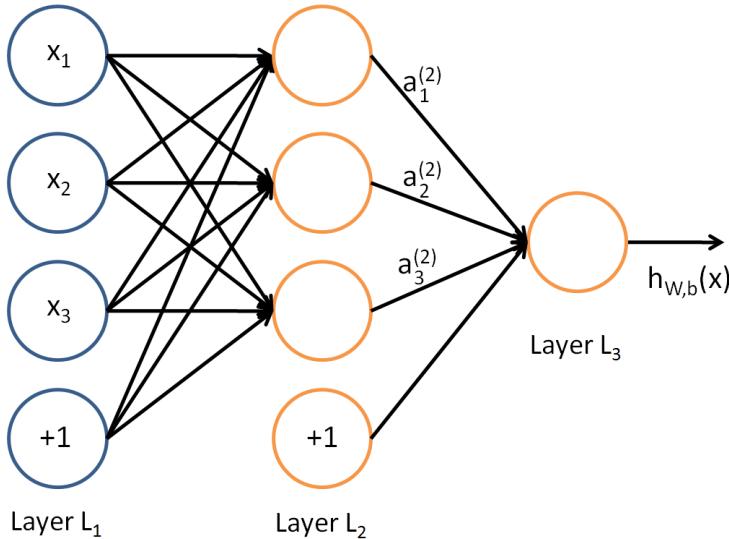


再来看看偏倚 $b$ 的定义。还是以这个三层的DNN为例，第二层的第三个神经元对应的偏倚定义为 $b_3^2$ 。其中，上标2代表所在的层数，下标3代表偏倚所在的神经元的索引。同样的道理，第三个的第一个神经元的偏倚应该表示为 $b_1^3$ 。同样的，输入层是没有偏倚参数 $b$ 的。



### 3. DNN前向传播算法数学原理

在上一节，我们已经介绍了DNN各层线性关系系数 $w$ ,偏倚 $b$ 的定义。假设我们选择的激活函数是 $\sigma(z)$ ，隐藏层和输出层的输出值为 $a$ ，则对于下图的三层DNN,利用和感知机一样的思路，我们可以利用上一层的输出计算下一层的输出，也就是所谓的DNN前向传播算法。



对于第二层的的输出 $a_1^2, a_2^2, a_3^2$ ，我们有：

$$\begin{aligned} a_1^2 &= \sigma(z_1^2) = \sigma(w_{11}^2 x_1 + w_{12}^2 x_2 + w_{13}^2 x_3 + b_1^2) \\ a_2^2 &= \sigma(z_2^2) = \sigma(w_{21}^2 x_1 + w_{22}^2 x_2 + w_{23}^2 x_3 + b_2^2) \\ a_3^2 &= \sigma(z_3^2) = \sigma(w_{31}^2 x_1 + w_{32}^2 x_2 + w_{33}^2 x_3 + b_3^2) \end{aligned}$$

对于第三层的的输出 $a_1^3$ ，我们有：

$$a_1^3 = \sigma(z_1^3) = \sigma(w_{11}^3 a_1^2 + w_{12}^3 a_2^2 + w_{13}^3 a_3^2 + b_1^3)$$

将上面的例子一般化，假设第 $l - 1$ 层共有 $m$ 个神经元，则对于第 $l$ 层的第 $j$ 个神经元的输出 $a_j^l$ ，我们有：

$$a_j^l = \sigma(z_j^l) = \sigma\left(\sum_{k=1}^m w_{jk}^l a_k^{l-1} + b_j^l\right)$$

其中，如果 $l = 2$ ，则对于的 $a_k^1$ 即为输入层的 $x_k$ 。

从上面可以看出，使用代数法一个个的表示输出比较复杂，而如果使用矩阵法则比较的简洁。假设第 $l - 1$ 层共有 $m$ 个神经元，而第 $l$ 层共有 $n$ 个神经元，则第 $l$ 层的线性系数 $w$ 组成了一个 $n \times m$ 的矩阵 $W^l$ ，第 $l$ 层的偏倚 $b$ 组成了一个 $n \times 1$ 的向量 $b^l$ ，第 $l - 1$ 层的的输出 $a$ 组成了一个 $m \times 1$ 的向量 $a^{l-1}$ ，第 $l$ 层的的未激活前线性输出 $z$ 组成了一个 $n \times 1$ 的向量 $z^l$ ，第 $l$ 层的输出 $a$ 组成了一个 $n \times 1$ 的向量 $a^l$ 。则用矩阵法表示，第 $l$ 层的输出为：

$$a^l = \sigma(z^l) = \sigma(W^l a^{l-1} + b^l)$$

这个表示方法简洁漂亮，后面我们的讨论都会基于上面的这个矩阵法表示来。

### 4. DNN前向传播算法

有了上一节的数学推导，DNN的前向传播算法也就不难了。所谓的DNN的前向传播算法也就是利用我们的若干个权重系数矩阵 $W$ ,偏倚向量 $b$ 来和输入值向量 $x$ 进行一系列线性运算和激活运算，从输入层开始，一层层的向后计算，一直到运算到输出层，得到输出结果为值。

输入：总层数 $L$ ，所有隐藏层和输出层对应的矩阵 $W$ ,偏倚向量 $b$ ，输入值向量 $x$

输出：输出层的输出 $a^L$

1 ) 初始化 $a^1 = x$

2) for  $l = 2$  to  $L$ , 计算：

$$a^l = \sigma(z^l) = \sigma(W^l a^{l-1} + b^l)$$

最后的结果即为输出 $a^L$ 。

### 5. DNN前向传播算法小结

单独看DNN前向传播算法，似乎没有什么大用处，而且这一大堆的矩阵 $W$ ,偏倚向量 $b$ 对应的参数怎么获得呢？怎么得到最优的矩阵 $W$ ,偏倚向量 $b$ 呢？这个我们在讲DNN的反向传播算法时再讲。而理解反向传播算法的前提就是理解DNN的模型

与前向传播算法。这也是我们这一篇先讲的原因。

---

## 深度神经网络 (DNN) 反向传播算法(BP)

在深度神经网络 (DNN) 模型与前向传播算法中，我们对DNN的模型和前向传播算法做了总结，这里我们更进一步，对DNN的反向传播算法 ( Back Propagation , BP ) 做一个总结。

### 1. DNN反向传播算法要解决的问题

在了解DNN的反向传播算法前，我们先要知道DNN反向传播算法要解决的问题，也就是说，什么时候我们需要这个反向传播算法？

回到我们监督学习的一般问题，假设我们有m个训练样本： $\{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$ ，其中 $x$ 为输入向量，特征维度为 $n\_in$ ，而 $y$ 为输出向量，特征维度为 $n\_out$ 。我们需要利用这m个样本训练出一个模型，当有一个新的测试样本( $x_{test}$ ?)来到时，我们可以预测 $y_{test}$ 向量的输出。

如果我们采用DNN的模型，即我们使输入层有 $n\_in$ 个神经元，而输出层有 $n\_out$ 个神经元。再加上一些含有若干神经元的隐藏层。此时我们需要找到合适的所有隐藏层和输出层对应的线性系数矩阵 $W$ ，偏倚向量 $b$ ，让所有的训练样本输入计算出的输出尽可能的等于或很接近样本输出。怎么找到合适的参数呢？

如果大家对传统的机器学习的算法优化过程熟悉的话，这里就很容易联想到我们可以用一个合适的损失函数来度量训练样本的输出损失，接着对这个损失函数进行优化求最小化的极值，对应的一系列线性系数矩阵 $W$ ，偏倚向量 $b$ 即为我们的最终结果。在DNN中，损失函数优化极值求解的过程最常见的一般是通过梯度下降法来一步步迭代完成的，当然也可以是其他的迭代方法比如牛顿法与拟牛顿法。如果大家对梯度下降法不熟悉，建议先阅读我之前写的梯度下降 (Gradient Descent) 小结。

对DNN的损失函数用梯度下降法进行迭代优化求极小值的过程即为我们的反向传播算法。

### 2. DNN反向传播算法的基本思路

在进行DNN反向传播算法前，我们需要选择一个损失函数，来度量训练样本计算出的输出和真实的训练样本输出之间的损失。你也许会问：训练样本计算出的输出是怎么得来的？这个输出是随机选择一系列 $W, b$ ，用我们上一节的前向传播算法计算出来的。即通过一系列的计算： $a^l = \sigma(z^l) = \sigma(W^l a^{l-1} + b^l)$ 。计算到输出层第 $L$ 层对应的 $a^L$ 即为前向传播算法计算出来的输出。

回到损失函数，DNN可选择的损失函数有不少，为了专注算法，这里我们使用最常见的均方差来度量损失。即对于每个样本，我们期望最小化下式：

$$J(W, b, x, y) = \frac{1}{2} \|a^L - y\|_2^2$$

其中， $a^L$ 和 $y$ 为特征维度为 $n\_out$ 的向量，而 $\|S\|_2$ 为S的L2范数。

损失函数有了，现在我们开始用梯度下降法迭代求解每一层的 $W, b$ 。

首先是输出层第 $L$ 层。注意到输出层的 $W, b$ 满足下式：

$$a^L = \sigma(z^L) = \sigma(W^L a^{L-1} + b^L)$$

这样对于输出层的参数，我们的损失函数变为：

$$J(W, b, x, y) = \frac{1}{2} \|a^L - y\|_2^2 = \frac{1}{2} \|\sigma(W^L a^{L-1} + b^L) - y\|_2^2$$

这样求解 $W, b$ 的梯度就简单了：

$$\begin{aligned} \frac{\partial J(W, b, x, y)}{\partial W^L} &= \frac{\partial J(W, b, x, y)}{\partial z^L} \frac{\partial z^L}{\partial W^L} = (a^L - y)(a^{L-1})^T \odot \sigma'(z) \\ \frac{\partial J(W, b, x, y)}{\partial b^L} &= \frac{\partial J(W, b, x, y)}{\partial z^L} \frac{\partial z^L}{\partial b^L} = (a^L - y) \odot \sigma'(z^L) \end{aligned}$$

注意上式中有一个符号 $\odot$ ，它代表Hadamard积，对于两个维度相同的向量 $A (a_1, a_2, \dots, a_n)^T$ 和 $B (b_1, b_2, \dots, b_n)^T$ ，则 $A \odot B = (a_1 b_1, a_2 b_2, \dots, a_n b_n)^T$ 。

我们注意到在求解输出层的 $W, b$ 的时候，有公共的部分 $\frac{\partial J(W, b, x, y)}{\partial z^L}$ ，因此我们可以把公共的部分即对 $z^L$ 先算出来，记为：

$$\delta^L = \frac{\partial J(W, b, x, y)}{\partial z^L} = (a^L - y) \odot \sigma'(z^L)$$

现在我们终于把输出层的梯度算出来了，那么如何计算上一层 $L - 1$ 层的梯度，上上层 $L - 2$ 层的梯度呢？这里我们需要一步一步的递推，注意到对于第 $l$ 层的未激活输出 $z^l$ ，它的梯度可以表示为：

$$\delta^l = \frac{\partial J(W, b, x, y)}{\partial z^l} = \frac{\partial J(W, b, x, y)}{\partial z^L} \frac{\partial z^L}{\partial z^{L-1}} \frac{\partial z^{L-1}}{\partial z^{L-2}} \cdots \frac{\partial z^{l+1}}{\partial z^l}$$

如果我们可以依次计算出第 $l$ 层的 $\delta^l$ ，则该层的 $W^l, b^l$ 很容易计算？为什么呢？注意到根据前向传播算法，我们有：

$$z^l = W^l a^{l-1} + b^l$$

所以根据上式我们可以很方便的计算出第 $l$ 层的 $W^l, b^l$ 的梯度如下：

$$\begin{aligned}\frac{\partial J(W, b, x, y)}{\partial W^l} &= \frac{\partial J(W, b, x, y)}{\partial z^l} \frac{\partial z^l}{\partial W^l} = \delta^l (a^{l-1})^T \\ \frac{\partial J(W, b, x, y)}{\partial b^l} &= \frac{\partial J(W, b, x, y)}{\partial z^l} \frac{\partial z^l}{\partial b^l} = \delta^l\end{aligned}$$

那么现在问题的关键就是要求出 $\delta^l$ 了。这里我们用数学归纳法，第 $L$ 层的 $\delta^L$ 上面我们已经求出，假设第 $l + 1$ 层的 $\delta^{l+1}$ 已经求出来了，那么我们如何求出第 $l$ 层的 $\delta^l$ 呢？我们注意到：

$$\delta^l = \frac{\partial J(W, b, x, y)}{\partial z^l} = \frac{\partial J(W, b, x, y)}{\partial z^{l+1}} \frac{\partial z^{l+1}}{\partial z^l} = \delta^{l+1} \frac{\partial z^{l+1}}{\partial z^l}$$

可见，用归纳法递推 $\delta^{l+1}$ 和 $\delta^l$ 的关键在于求解 $\frac{\partial z^{l+1}}{\partial z^l}$ 。

而 $z^{l+1}$ 和 $z^l$ 的关系其实很容易找出：

$$z^{l+1} = W^{l+1} a^l + b^{l+1} = W^{l+1} \sigma(z^l) + b^{l+1}$$

这样很容易求出：

$$\frac{\partial z^{l+1}}{\partial z^l} = (W^{l+1})^T \odot \sigma'(z^l)$$

将上式带入上面 $\delta^{l+1}$ 和 $\delta^l$ 关系式我们得到：

$$\delta^l = \delta^{l+1} \frac{\partial z^{l+1}}{\partial z^l} = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

现在我们得到了 $\delta^l$ 的递推关系式，只要求出了某一层的 $\delta^l$ ，求解 $W^l, b^l$ 的对应梯度就很简单了。

### 3. DNN反向传播算法过程

现在我们总结下DNN反向传播算法的过程。由于梯度下降法有批量（Batch），小批量(mini-Batch)，随机三个变种，为了简化描述，这里我们以最基本的批量梯度下降法为例来描述反向传播算法。实际上在业界使用最多的是mini-Batch的梯度下降法。不过区别仅仅在于迭代时训练样本的选择而已。

输入：总层数 $L$ ，以及各隐藏层与输出层的神经元个数，激活函数，损失函数，迭代步长 $\alpha$ ，最大迭代次数MAX与停止迭代阈值 $\epsilon$ ，输入的 $m$ 个训练样本 $\{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$

输出：各隐藏层与输出层的线性关系系数矩阵 $W$ 和偏倚向量 $b$

1) 初始化各隐藏层与输出层的线性关系系数矩阵 $W$ 和偏倚向量 $b$ 的值为一个随机值。

2 ) for iter to 1 to MAX :

2-1) for i = 1 to m :

a) 将DNN输入 $a^1$ 设置为 $x_i$

b) for  $l = 2$  to  $L$ ，进行前向传播算法计算 $a^{i,l} = \sigma(z^{i,l}) = \sigma(W^l a^{i,l-1} + b^l)$

c) 通过损失函数计算输出层的 $\delta^{i,L}$

d) for  $l = L$  to 2，进行反向传播算法计算 $\delta^{i,l} = (W^{l+1})^T \delta^{i,l+1} \odot \sigma'(z^{i,l})$

2-2) for  $l = 2$  to  $L$ ，更新第 $l$ 层的 $W^l, b^l$ ：

$$W^l = W^l - \alpha \sum_{i=1}^m \delta^{i,l} (a^{i,l-1})^T$$

$$b^l = b^l - \alpha \sum_{i=1}^m \delta^{i,l}$$

2-3) 如果所有 $W, b$ 的变化值都小于停止迭代阈值 $\epsilon$ ，则跳出迭代循环到步骤3。

3 ) 输出各隐藏层与输出层的线性关系系数矩阵 $W$ 和偏倚向量 $b$ 。

### 4. DNN反向传播算法小结

有了DNN反向传播算法，我们就可以很方便的用DNN的模型去解决第一节里面提到了各种监督学习的分类回归问题。当然DNN的参数众多，矩阵运算量也很大，直接使用会有各种各样的问题。有哪些问题以及如何尝试解决这些问题并优化DNN模型与算法，我们在下一篇讲。

---

## 深度神经网络 ( DNN ) 损失函数和激活函数的选择

在深度神经网络 ( DNN ) 反向传播算法(BP)中，我们对DNN的前向反向传播算法的使用做了总结。里面使用的损失函数是均方差，而激活函数是Sigmoid。实际上DNN可以使用的损失函数和激活函数不少。这些损失函数和激活函数如何选择呢？下面我们就对DNN损失函数和激活函数的选择做一个总结。

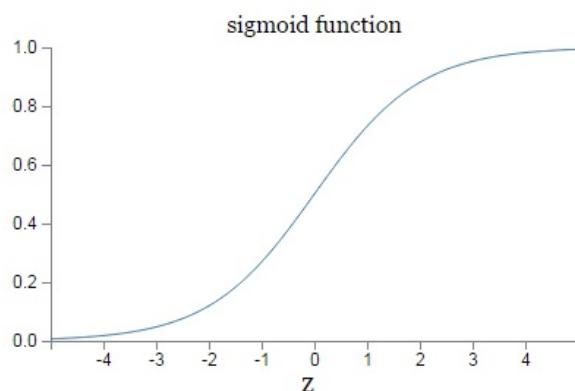
### 1. 均方差损失函数+Sigmoid激活函数的问题

在讲反向传播算法时，我们用均方差损失函数和Sigmoid激活函数做了实例，首先我们就来看看均方差+Sigmoid的组合有什么问题。

首先我们回顾下Sigmoid激活函数的表达式为：

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

$\sigma(z)$ 的函数图像如下：



从图上可以看出，对于Sigmoid，当 $z$ 的取值越来越大后，函数曲线变得越来越平缓，意味着此时的导数 $\sigma'(z)$ 也越来越小。同样的，当 $z$ 的取值越来越小时，也有这个问题。仅仅在 $z$ 取值为0附近时，导数 $\sigma'(z)$ 的取值较大。

在上篇讲的均方差+Sigmoid的反向传播算法中，每一层向后递推都要乘以 $\sigma'(z)$ ，得到梯度变化值。Sigmoid的这个曲线意味着在大多数时候，我们的梯度变化值很小，导致我们的 $W, b$ 更新到极值的速度较慢，也就是我们的算法收敛速度较慢。那么有什么办法可以改进呢？

### 2. 使用交叉熵损失函数+Sigmoid激活函数改进 DNN算法收敛速度

上一节我们讲到Sigmoid的函数特性导致反向传播算法收敛速度慢的问题，那么如何改进呢？换掉Sigmoid？这当然是一种选择。另一种常见的选择是用交叉熵损失函数来代替均方差损失函数。

我们来看看每个样本的交叉熵损失函数的形式：

$$J(W, b, a, y) = -y \bullet \ln a - (1 - y) \bullet \ln(1 - a)$$

其中， $\bullet$ 为向量内积。这个形式其实很熟悉，在逻辑回归原理小结中其实我们就用到了类似的形式，只是当时我们是用最大似然估计推导出来的，而这个损失函数的学名叫交叉熵。

使用了交叉熵损失函数，就能解决Sigmoid函数导数变化大多数时候反向传播算法慢的问题吗？我们来看看当使用交叉熵时，我们输出层 $\delta^L$ 的梯度情况。

$$\delta^L = \frac{\partial J(W, b, a^L, y)}{\partial z^L} \quad (1)$$

$$= -y \frac{1}{a^L} (a^L)(1-a^L) + (1-y) \frac{1}{1-a^L} (a^L)(1-a^L) \quad (2)$$

$$= -y(1-a^L) + (1-y)a^L \quad (3)$$

$$= a^L - y \quad (4)$$

可见此时我们的 $\delta^L$ 梯度表达式里面已经没有了 $\sigma'(z)$ ，作为一个特例，回顾一下我们上一节均方差损失函数时在 $\delta^L$ 梯度，

$$\frac{\partial J(W, b, x, y)}{\partial z^L} = (a^L - y) \odot \sigma'(z)$$

对比两者在第L层的 $\delta^L$ 梯度表达式，就可以看出，使用交叉熵，得到的 $\delta^L$ 梯度表达式没有了 $\sigma'(z)$ ，梯度为预测值和真实值的差距，这样求得的 $W^l, b^l$ 的地图也不包含 $\sigma'(z)$ ，因此避免了反向传播收敛速度慢的问题。

通常情况下，如果我们使用了sigmoid激活函数，交叉熵损失函数肯定比均方差损失函数好用。

### 3. 使用对数似然损失函数和softmax激活函数进行DNN分类输出

在前面我们讲的所有DNN相关知识中，我们都假设输出是连续可导的值。但是如果是分类问题，那么输出是一个个的类别，那我们怎么用DNN来解决这个问题呢？

比如假设我们有一个三个类别的分类问题，这样我们的DNN输出层应该有三个神经元，假设第一个神经元对应类别一，第二个对应类别二，第三个对应类别三，这样我们期望的输出应该是(1,0,0), (0,1,0)和(0,0,1)这三种。即样本真实类别对应的神经元输出应该无限接近或者等于1，而非改样本真实输出对应的神经元的输出应该无限接近或者等于0。或者说，我们希望输出层的神经元对应的输出是若干个概率值，这若干个概率值即我们DNN模型对于输入值对于各类别的输出预测，同时为满足概率模型，这若干个概率值之和应该等于1。

DNN分类模型要求是输出层神经元输出的值在0到1之间，同时所有输出值之和为1。很明显，现有的普通DNN是无法满足这个要求的。但是我们只需要对现有的全连接DNN稍作改良，即可用于解决分类问题。在现有的DNN模型中，我们可以将输出层第i个神经元的激活函数定义为如下形式：

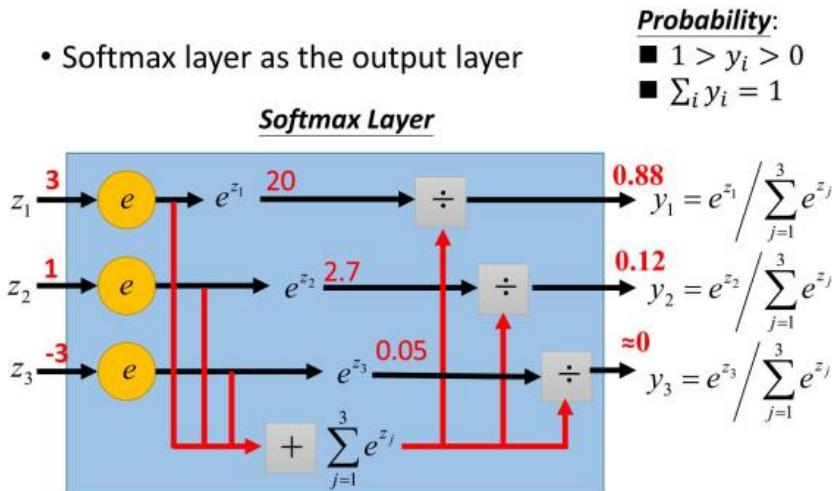
$$a_i^L = \frac{e^{z_i^L}}{\sum_{j=1}^{n_L} e^{z_j^L}}$$

其中， $n_L$ 是输出层第L层的神经元个数，或者说我们的分类问题的类别数。

很容易看出，所有的 $a_i^L$ 都是在(0,1)之间的数字，而 $\sum_{j=1}^{n_L} e^{z_j^L}$ 作为归一化因子保证了所有的 $a_i^L$ 之和为1。

这个方法很简洁漂亮，仅仅只需要将输出层的激活函数从Sigmoid之类的函数转变为上式的激活函数即可。上式这个激活函数就是我们的softmax激活函数。它在分类问题中有广泛的应用。将DNN用于分类问题，在输出层用softmax激活函数也是最常见的了。

下面这个例子清晰的描述了softmax激活函数在前向传播算法时的使用。假设我们的输出层为三个神经元，而未激活的输出为3,1和-3，我们求出各自的指数表达式为：20,2.7和0.05，我们的归一化因子即为22.75，这样我们就求出了三个类别的概率输出分布为0.88, 0.12和0。



从上面可以看出，将softmax用于前向传播算法是很简单的。那么在反向传播算法时还简单吗？反向传播的梯度好计算吗？答案是Yes！

对于用于分类的softmax激活函数，对应的损失函数一般都是用对数似然函数，即：

$$J(W, b, a^L, y) = - \sum_k y_k \ln a_k^L$$

其中 $y_i$ 的取值为0或者1，如果某一训练样本的输出为第*i*类。则 $y_i = 1$ ,其余的 $j \neq i$ 都有 $y_j = 0$ 。由于每个样本只属于一个类别，所以这个对数似然函数可以简化为：

$$J(W, b, a^L, y) = -\ln a_i^L$$

其中*i*即为训练样本真实的类别序号。

可见损失函数只和真实类别对应的输出有关，这样假设真实类别是第*i*类，则其他不属于第*i*类序号对应的神经元的梯度导数直接为0。对于真实类别第*i*类，它的 $W_i^L$ 对应的梯度计算为：

$$\frac{\partial J(W, b, a^L, y)}{\partial W_i^L} = \frac{\partial J(W, b, a^L, y)}{\partial a_i^L} \frac{\partial a_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial w_i^L} \quad (5)$$

$$= -\frac{1}{a_i^L} \frac{(e^{z_i^L}) \sum_{j=1}^{n_L} e^{z_j^L} - e^{z_i^L} e^{z_i^L}}{(\sum_{j=1}^{n_L} e^{z_j^L})^2} a_i^{L-1} \quad (6)$$

$$= -\frac{1}{a_i^L} \left( \frac{e^{z_i^L}}{\sum_{j=1}^{n_L} e^{z_j^L}} - \frac{e^{z_i^L}}{\sum_{j=1}^{n_L} e^{z_j^L}} \right) a_i^{L-1} \quad (7)$$

$$= -\frac{1}{a_i^L} a_i^L (1 - a_i^L) a_i^{L-1} \quad (8)$$

$$= (a_i^L - 1) a_i^{L-1} \quad (9)$$

同样的可以得到 $b_i^L$ 的梯度表达式为：

$$\frac{\partial J(W, b, a^L, y)}{\partial b_i^L} = a_i^L - 1$$

可见，梯度计算也很简洁，也没有第一节说的训练速度慢的问题。举个例子，假如我们对于第2类的训练样本，通过前向算法计算的未激活输出为(1, 5, 3)，则我们得到softmax激活后的概率输出为：(0.015, 0.866, 0.117)。由于我们的类别是第二类，则反向传播的梯度应该为：(0.015, 0.866-1, 0.117)。是不是很简单呢？

当softmax输出层的反向传播计算完以后，后面的普通DNN层的反向传播计算和之前讲的普通DNN没有区别。

## 4. 梯度爆炸梯度消失与ReLU激活函数

学习DNN，大家一定听说过梯度爆炸和梯度消失两个词。尤其是梯度消失，是限制DNN与深度学习的一个关键障碍，目前也没有完全攻克。

什么是梯度爆炸和梯度消失呢？从理论上说都可以写一篇论文出来。不过简单理解，就是在反向传播的算法过程中，由于我们使用了是矩阵求导的链式法则，有一大串连乘，如果连乘的数字在每层都是小于1的，则梯度越往前乘越小，导致梯度消失，而如果连乘的数字在每层都是大于1的，则梯度越往前乘越大，导致梯度爆炸。

比如我们在前一篇反向传播算法里面讲到了 $\delta$ 的计算，可以表示为：

$$\delta^l = \frac{\partial J(W, b, x, y)}{\partial z^l} = \frac{\partial J(W, b, x, y)}{\partial z^L} \frac{\partial z^L}{\partial z^{L-1}} \frac{\partial z^{L-1}}{\partial z^{L-2}} \cdots \frac{\partial z^{l+1}}{\partial z^l}$$

如果不巧我们的样本导致每一层 $\frac{\partial z^{l+1}}{\partial z^l}$ 的都小于1，则随着反向传播算法的进行，我们的 $\delta^l$ 会随着层数越来越小，甚至接近0，导致梯度几乎消失，进而导致前面的隐藏层的 $W, b$ 参数随着迭代的进行，几乎没有大的改变，更谈不上收敛了。这个问题目前没有完美的解决办法。

而对于梯度爆炸，则一般可以通过调整我们DNN模型中的初始化参数得以解决。

对于无法完美解决的梯度消失问题，目前有很多研究，一个可能部分解决梯度消失问题的办法是使用ReLU ( Rectified Linear Unit ) 激活函数，ReLU在卷积神经网络CNN中得到了广泛的应用，在CNN中梯度消失似乎不再是问题。那么它是什么样子呢？其实很简单，比我们前面提到的所有激活函数都简单，表达式为：

$$\sigma(z) = \max(0, z)$$

也就是说大于等于0则不变，小于0则激活后为0。就这么一玩意就可以解决梯度消失？至少部分是的。具体的原因现在其实也没有从理论上得以证明。这里我也就不多说了。

## 5. DNN其他激活函数

除了上面提到了激活函数，DNN常用的激活函数还有：

1 ) tanh : 这个是sigmoid的变种，表达式为：

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

tanh激活函数和sigmoid激活函数的关系为：

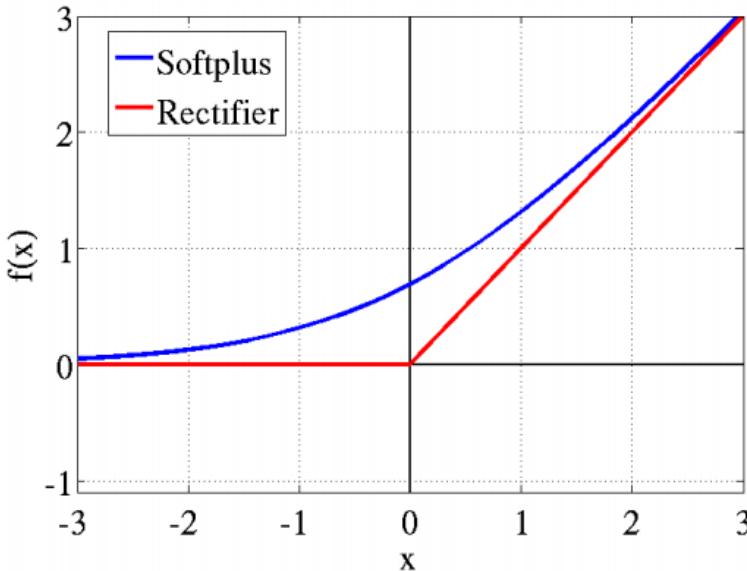
$$\tanh(z) = 2\text{sigmoid}(2z) - 1$$

tanh和sigmoid对比主要的特点是它的输出落在了[-1,1]，这样输出可以进行标准化。同时tanh的曲线在较大时变得平坦的幅度没有sigmoid那么大，这样求梯度变化值有一些优势。当然，要说tanh一定比sigmoid好倒不一定，还是要具体问题具体分析。

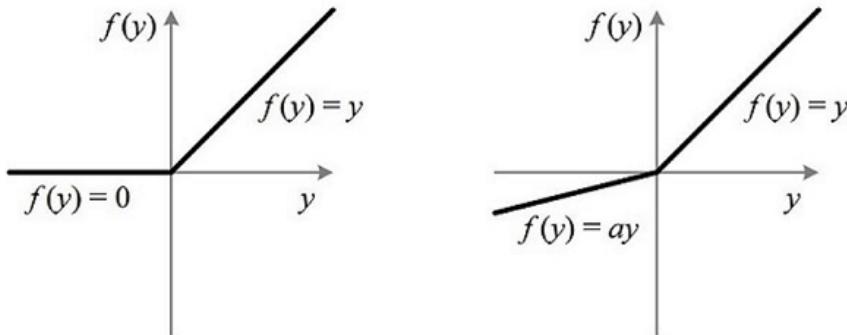
2 ) softplus : 这个其实就是sigmoid函数的原函数，表达式为：

$$\text{softplus}(z) = \log(1 + e^z)$$

它的导数就是sigmoid函数。softplus的函数图像和ReLU有些类似。它出现的比ReLU早，可以视为ReLU的鼻祖。



3 ) PReLU : 从名字就可以看出它是ReLU的变种，特点是如果未激活值小于0，不是简单粗暴的直接变为0，而是进行一定幅度的缩小。如下图。当然，由于ReLU的成功，有很多的跟风者，有其他各种变种ReLU，这里就不多提了。



$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \quad PReLU(x_i) = \begin{cases} x_i & \text{if } x_i > 0 \\ a_i x_i & \text{if } x_i \leq 0 \end{cases}$$

$i$  表示不同的通道

## 6. DNN损失函数和激活函数小结

上面我们对DNN损失函数和激活函数做了详细的讨论，重要的点有：1 ) 如果使用sigmoid激活函数，则交叉熵损失函数一般肯定比均方差损失函数好。2 ) 如果是DNN用于分类，则一般在输出层使用softmax激活函数和对数似然损失函数。3 ) ReLU激活函数对梯度消失问题有一定程度的解决，尤其是在CNN模型中。

下一篇我们讨论下DNN模型的正则化问题。

## 深度神经网络 ( DNN ) 的正则化

和普通的机器学习算法一样，DNN也会遇到过拟合的问题，需要考虑泛化，这里我们就对DNN的正则化方法做一个总结。

### 1. DNN的L1&L2正则化

想到正则化，我们首先想到的就是L1正则化和L2正则化。L1正则化和L2正则化原理类似，这里重点讲述DNN的L2正则化。

而DNN的L2正则化通常的做法是只针对与线性系数矩阵  $W$ ，而不针对偏倚系数  $b$ 。利用我们之前的机器学习的知识，我们很容易可以写出DNN的L2正则化的损失函数。

假如我们的每个样本的损失函数是均方差损失函数，则所有的  $m$  个样本的损失函数为：

$$J(W, b) = \frac{1}{2m} \sum_{i=1}^m \|a^L - y\|_2^2$$

则加上了L2正则化后的损失函数是：

$$J(W, b) = \frac{1}{2m} \sum_{i=1}^m \|a^L - y\|_2^2 + \frac{\lambda}{2m} \sum_{l=2}^L \|w\|_2^2$$

其中， $\lambda$  即我们的正则化超参数，实际使用时需要调参。而  $w$  为所有权重矩阵  $W$  的所有列向量。

如果使用上式的损失函数，进行反向传播算法时，流程和没有正则化的反向传播算法完全一样，区别仅仅在于进行梯度下降法时， $W$  的更新公式。

回想我们在深度神经网络 ( DNN ) 反向传播算法(BP)中， $W$  的梯度下降更新公式为：

$$W^l = W^l - \alpha \sum_{i=1}^m \delta^{i,l} (a^{x,l-1})^T$$

则加入L2正则化以后，迭代更新公式变成：

$$W^l = W^l - \alpha \sum_{i=1}^m \delta^{i,l} (a^{x,l-1})^T - \alpha \lambda W^l$$

注意到上式中的梯度计算中  $\frac{1}{m}$  我忽略了，因为  $\alpha$  是常数，而除以  $m$  也是常数，所以等同于用了新常数  $\alpha$  来代替  $\frac{\alpha}{m}$ 。进而简化表达式，但是不影响损失算法。

类似的L2正则化方法可以用于交叉熵损失函数或者其他DNN损失函数，这里就不累述了。

### 2. DNN通过集成学习的思路正则化

除了常见的L1&L2正则化，DNN还可以通过集成学习的思路正则化。在集成学习原理小结中，我们讲到集成学习有 Boosting 和 Bagging 两种思路。而DNN可以用Bagging的思路来正则化。常用的机器学习Bagging算法中，随机森林是最流行的。它 通过随机采样构建若干个相互独立的弱决策树学习器，最后采用加权平均法或者投票法决定集成的输出。在DNN中，我们一样使用Bagging的思路。不过和随机森林不同的是，我们这里不是若干个决策树，而是若干个DNN的网络。

首先我们要对原始的  $m$  个训练样本进行有放回随机采样，构建  $N$  组  $m$  个样本的数据集，然后分别用这  $N$  组数据集去训练我们的DNN。即采用我们的前向传播算法和反向传播算法得到  $N$  个DNN模型的  $W, b$  参数组合，最后对  $N$  个DNN模型的输出用加权平均法或者投票法决定最终输出。

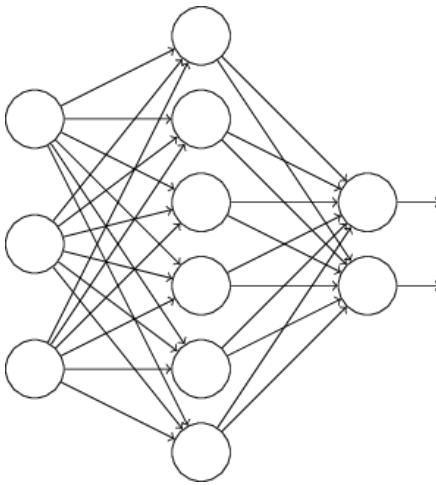
不过用集成学习Bagging的方法有一个问题，就是我们的DNN模型本来就比较复杂，参数很多。现在又变成了  $N$  个DNN模型，这样参数又增加了  $N$  倍，从而导致训练这样的网络要花更多的时间和空间。因此一般  $N$  的个数不能太多，比如5-10个就可以了。

### 3. DNN通过dropout 正则化

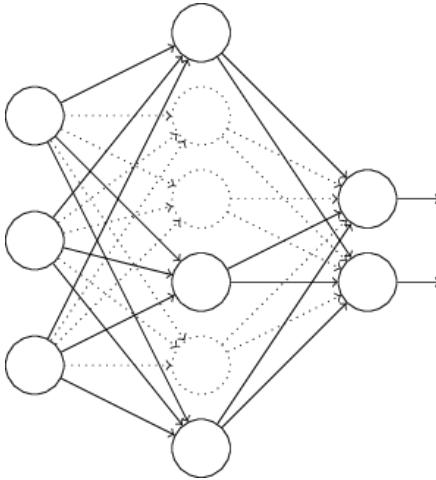
这里我们再讲一种和Bagging类似但是又不同的正则化方法：Dropout。

所谓的Dropout指的是在用前向传播算法和反向传播算法训练DNN模型时，一批数据迭代时，随机的从全连接DNN网络中去掉一部分隐藏层的神经元。

比如我们本来的DNN模型对应的结构是这样的：



在对训练集中的一批数据进行训练时，我们随机去掉一部分隐藏层的神经元，并用去掉隐藏层的神经元的网络来拟合我们的一批训练数据。如下图，去掉了一半的隐藏层神经元：



然后用这个去掉隐藏层的神经元的网络来进行一轮迭代，更新所有的 $W, b$ 。这就是所谓的dropout。

当然，dropout并不意味着这些神经元永远的消失了。在下一批数据迭代前，我们会把DNN模型恢复成最初的全连接模型，然后再用随机的方法去掉部分隐藏层的神经元，接着去迭代更新 $W, b$ 。当然，这次用随机的方法去掉部分隐藏层后的残缺DNN网络和上次的残缺DNN网络并不相同。

总结下dropout的方法：每轮梯度下降迭代时，它需要将训练数据分成若干批，然后分批进行迭代，每批数据迭代时，需要将原始的DNN模型随机去掉部分隐藏层的神经元，用残缺的DNN模型来迭代更新 $W, b$ 。每批数据迭代更新完毕后，要将残缺的DNN模型恢复成原始的DNN模型。

从上面的描述可以看出dropout和Bagging的正则化思路还是很不相同的。dropout模型中的 $W, b$ 是一套，共享的。所有的残缺DNN迭代时，更新的是同一组 $W, b$ ；而Bagging正则化时每个DNN模型有自己独有一套 $W, b$ 参数，相互之间是独立的。当然他们每次使用基于原始数据集得到的分批的数据集来训练模型，这点是类似的。

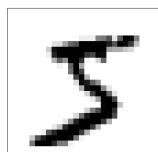
使用基于dropout的正则化比基于bagging的正则化简单，这显而易见，当然天下没有免费的午餐，由于dropout会将原始数据分批迭代，因此原始数据集最好较大，否则模型可能会欠拟合。

## 4. DNN通过增强数据集正则化

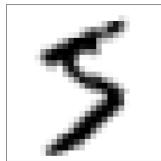
增强模型泛化能力最好的办法是有更多更多的训练数据，但是在实际应用中，更多的训练数据往往很难得到。有时候我们不得不去自己想办法能无中生有，来增加训练数据集，进而得到让模型泛化能力更强的目的。

对于我们传统的机器学习分类回归方法，增强数据集还是很难的。你无中生出一组特征输入，却很难知道对应的特征输出是什么。但是对于DNN擅长的领域，比如图像识别，语音识别等则是有办法的。以图像识别领域为例，对于原始的数据集中的图像，我们可以将原始图像稍微的平移或者旋转一点点，则得到了一个新的图像。虽然这是一个新的图像，即样本的特征是新的，但是我们知道对应的特征输出和之前未平移旋转的图像是一样的。

举个例子，下面这个图像，我们的特征输出是5。



我们将原始的图像旋转15度，得到了一副新的图像如下：



我们现在得到了一个新的训练样本，输入特征和之前的训练样本不同，但是特征输出是一样的，我们可以确定这是5。

用类似的思路，我们可以对原始的数据集进行增强，进而得到增强DNN模型的泛化能力的目的。

## 5. 其他DNN正则化方法

DNN的正则化的方法是很多的，还是持续的研究中。在Deep Learning这本书中，正则化是洋洋洒洒的一大章。里面提到的其他正则化方法有：Noise Robustness，Adversarial Training，Early Stopping等。如果大家对这些正则化方法感兴趣，可以去阅读Deep Learning这本书中的第七章。

## 卷积神经网络(CNN)模型结构

在前面我们讲述了DNN的模型与前向反向传播算法。而在DNN大类中，卷积神经网络(Convolutional Neural Networks，以下简称CNN)是最为成功的DNN特例之一。CNN广泛的应用于图像识别，当然现在也应用于NLP等其他领域，本文我们就对CNN的模型结构做一个总结。

在学习CNN前，推荐大家先学习DNN的知识。如果不熟悉DNN而去直接学习CNN，难度会比较大。这是我写的DNN的教程：

[深度神经网络 \( DNN \) 模型与前向传播算法](#)

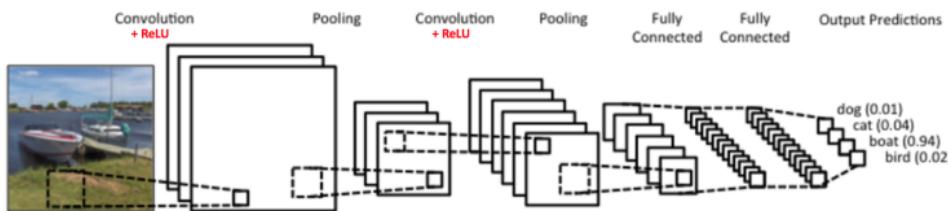
[深度神经网络 \( DNN \) 反向传播算法\(BP\)](#)

[深度神经网络 \( DNN \) 损失函数和激活函数的选择](#)

[深度神经网络 \( DNN \) 的正则化](#)

## 1. CNN的基本结构

首先我们来看看CNN的基本结构。一个常见的CNN例子如下图：



图中是一个图形识别的CNN模型。可以看出最左边的船的图像就是我们的输入层，计算机理解为输入若干个矩阵，这点和DNN基本相同。

接着是卷积层 (Convolution Layer)，这是CNN特有的，我们后面专门来讲。卷积层的激活函数使用的是ReLU。我们在DNN中介绍过ReLU的激活函数，它其实很简单，就是 $ReLU(x) = \max(0, x)$ 。在卷积层后面是池化层(Pooling layer)，这个也是CNN特有的，我们后面也会专门来讲。需要注意的是，池化层没有激活函数。

卷积层+池化层的组合可以在隐藏层出现很多次，上图中出现两次。而实际上这个次数是根据模型的需要而来的。当然我们也可以灵活使用使用卷积层+卷积层，或者卷积层+卷积层+池化层的组合，这些在构建模型的时候没有限制。但是最常见的CNN都是若干卷积层+池化层的组合，如上图中的CNN结构。

在若干卷积层+池化层后面是全连接层 (Fully Connected Layer, 简称FC)，全连接层其实就是我们前面讲的DNN结构，只是输出层使用了Softmax激活函数来做图像识别的分类，这点我们在DNN中也有讲述。

从上面CNN的模型描述可以看出，CNN相对于DNN，比较特殊的是卷积层和池化层，如果我们熟悉DNN，只要把卷积层和池化层的原理搞清楚了，那么搞清楚CNN就容易很多了。

## 2. 初识卷积

首先，我们去学习卷积层的模型原理，在学习卷积层的模型原理前，我们需要了解什么是卷积，以及CNN中的卷积是什么样子的。

大家学习数学时都有学过卷积的知识，微积分中卷积的表达式为：

$$S(t) = \int x(t-a)w(a)da$$

离散形式是：

$$s(t) = \sum_a x(t-a)w(a)$$

这个式子如果用矩阵表示可以为：

$$s(t) = (X * W)(t)$$

其中星号表示卷积。

如果是二维的卷积，则表示式为：

$$s(i, j) = (X * W)(i, j) = \sum_m \sum_n x(i - m, j - n)w(m, n)$$

在CNN中，虽然我们也是说卷积，但是我们的卷积公式和严格意义数学中的定义稍有不同，比如对于二维的卷积，定义为：

$$s(i, j) = (X * W)(i, j) = \sum_m \sum_n x(i + m, j + n)w(m, n)$$

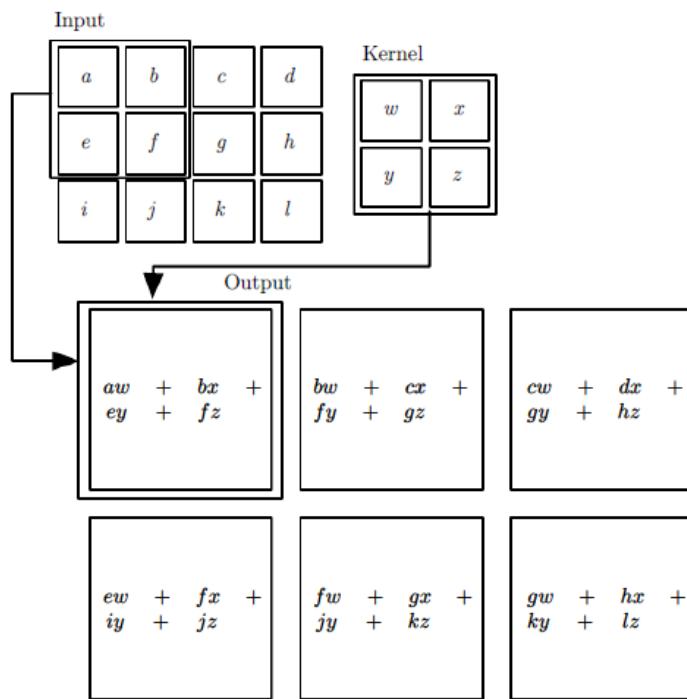
这个式子虽然从数学上讲不是严格意义上的卷积，但是大牛们都这么叫了，那么我们也跟着这么叫了。后面讲的CNN的卷积都是指的上面的最后一个式子。

其中，我们叫W为我们的卷积核，而X则为我们的输入。如果X是一个二维输入的矩阵，而W也是一个二维的矩阵。但是如果X是多维张量，那么W也是一个多维的张量。

### 3. CNN中的卷积层

有了卷积的基本知识，我们现在来看看CNN中的卷积，假如是对图像卷积，回想我们的上一节的卷积公式，其实就是对输出的图像的不同局部的矩阵和卷积核矩阵各个位置的元素相乘，然后相加得到。

举个例子如下，图中的输入是一个二维的 $3 \times 4$ 的矩阵，而卷积核是一个 $2 \times 2$ 的矩阵。这里我们假设卷积是一次移动一个像素来卷积的，那么首先我们对输入的左上角 $2 \times 2$ 局部和卷积核卷积，即各个位置的元素相乘再相加，得到的输出矩阵S的 $S_{00}$ 的元素，值为 $aw + bx + ey + fz$ 。接着我们将输入的局部向右平移一个像素，现在是(b,c,f,g)四个元素构成的矩阵和卷积核来卷积，这样我们得到了输出矩阵S的 $S_{01}$ 的元素，同样的方法，我们可以得到输出矩阵S的 $S_{02}, S_{10}, S_{11}, S_{12}$ 的元素。



最终我们得到卷积输出的矩阵为一个 $2 \times 3$ 的矩阵S。

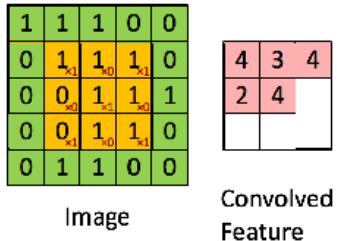
再举一个动态的卷积过程的例子如下：

我们有下面这个绿色的 $5 \times 5$ 输入矩阵，卷积核是一个下面这个黄色的 $3 \times 3$ 的矩阵。卷积的步幅是一个像素。则卷积的过程如下面的动图。卷积的结果是一个 $3 \times 3$ 的矩阵。

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

1	0	1
0	1	0
1	0	1



上面举的例子都是二维的输入，卷积的过程比较简单，那么如果输入是多维的呢？比如在前面一组卷积层+池化层的输出是3个矩阵，这3个矩阵作为输入呢，那么我们怎么去卷积呢？又比如输入的是对应RGB的彩色图像，即是三个分布对应R, G和B的矩阵呢？

在斯坦福大学的cs231n的课程上，有一个动态的例子，[链接在这](#)。建议大家对照着例子中的动图看下面的讲解。

大家打开这个例子可以看到，这里面输入是3个 $7 \times 7$ 的矩阵。实际上原输入是3个 $5 \times 5$ 的矩阵。只是在原来的输入周围加上了1的padding，即将周围都填充一圈的0，变成了3个 $7 \times 7$ 的矩阵。

例子里面使用了两个卷积核，我们先关注于卷积核 $W_0$ 。和上面的例子相比，由于输入是3个 $7 \times 7$ 的矩阵，或者说是 $7 \times 7 \times 3$ 的张量，则我们对应的卷积核 $W_0$ 也必须最后一维是3的张量，这里卷积核 $W_0$ 的单个子矩阵维度为 $3 \times 3$ 。那么卷积核 $W_0$ 实际上是一个 $3 \times 3 \times 3$ 的张量。同时和上面的例子比，这里的步幅为2，也就是每次卷积后会移动2个像素的位置。

最终的卷积过程和上面的2维矩阵类似，上面是矩阵的卷积，即两个矩阵对应位置的元素相乘后相加。这里是张量的卷积，即两个张量的3个子矩阵卷积后，再把卷积的结果相加后再加上偏倚 $b$ 。

$7 \times 7 \times 3$ 的张量和 $3 \times 3 \times 3$ 的卷积核张量 $W_0$ 卷积的结果是一个 $3 \times 3$ 的矩阵。由于我们有两个卷积核 $W_0$ 和 $W_1$ ，因此最后卷积的结果是两个 $3 \times 3$ 的矩阵。或者说卷积的结果是一个 $3 \times 3 \times 2$ 的张量。

仔细回味下卷积的过程，输入是 $7 \times 7 \times 3$ 的张量，卷积核是两个 $3 \times 3 \times 3$ 的张量。卷积步幅为2，最后得到了输出是 $3 \times 3 \times 2$ 的张量。如果把上面的卷积过程用数学公式表达出来就是：

$$s(i, j) = (X * W)(i, j) + b = \sum_{k=1}^{n\_in} (X_k * W_k)(i, j) + b$$

其中， $n\_in$ 为输入矩阵的个数，或者是张量的最后一维的维数。 $X_k$ 代表第 $k$ 个输入矩阵。 $W_k$ 代表卷积核的第 $k$ 个子卷积核矩阵。 $s(i, j)$ 即卷积核 $W$ 对应的输出矩阵的对应位置元素的值。

通过上面的例子，相信大家对CNN的卷积层的卷积过程有了一定的了解。

对于卷积后的输出，一般会通过ReLU激活函数，将输出的张量中的小于0的位置对应的元素值都变为0。

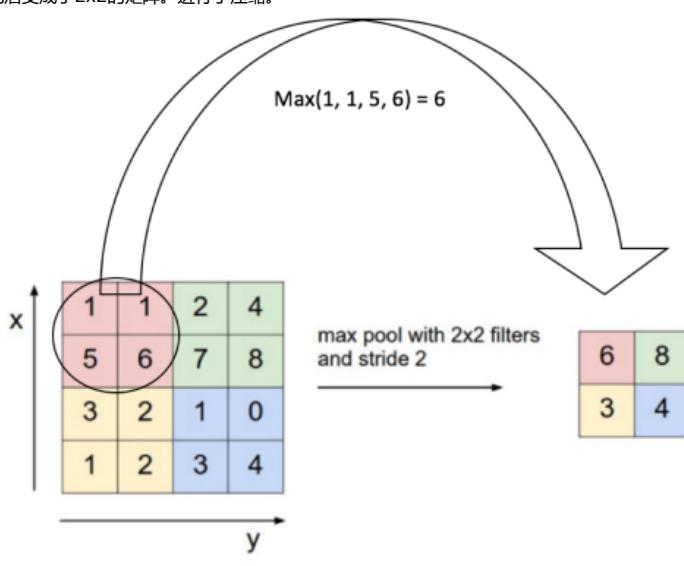
## 4. CNN中的池化层

相比卷积层的复杂，池化层则要简单的多，所谓的池化，个人理解就是对输入张量的各个子矩阵进行压缩。假如是 $2 \times 2$ 的池化，那么就将子矩阵的每 $2 \times 2$ 个元素变成一个元素，如果是 $3 \times 3$ 的池化，那么就将子矩阵的每 $3 \times 3$ 个元素变成一个元素，这样输入矩阵的维度就变小了。

要想将输入子矩阵的每 $n \times n$ 个元素变成一个元素，那么需要一个池化标准。常见的池化标准有2个，MAX或者是Average。即取对应区域的最大值或者平均值作为池化后的元素值。

下面这个例子采用取最大值的池化方法。同时采用的是 $2 \times 2$ 的池化。步幅为2。

首先对红色 $2 \times 2$ 区域进行池化，由于此 $2 \times 2$ 区域的最大值为6.那么对应的池化输出位置的值为6，由于步幅为2，此时移动到绿色区域去进行池化，输出的最大值为8.同样的方法，可以得到黄色区域和蓝色区域的输出值。最终，我们的输入 $4 \times 4$ 的矩阵在池化后变成了 $2 \times 2$ 的矩阵。进行了压缩。



Rectified Feature Map

## 5. CNN模型结构小结

理解了CNN模型中的卷积层和池化层，就基本理解了CNN的基本原理，后面再去理解CNN模型的前向传播算法和反向传播算法就容易了。下一篇我们就来讨论CNN模型的前向传播算法。

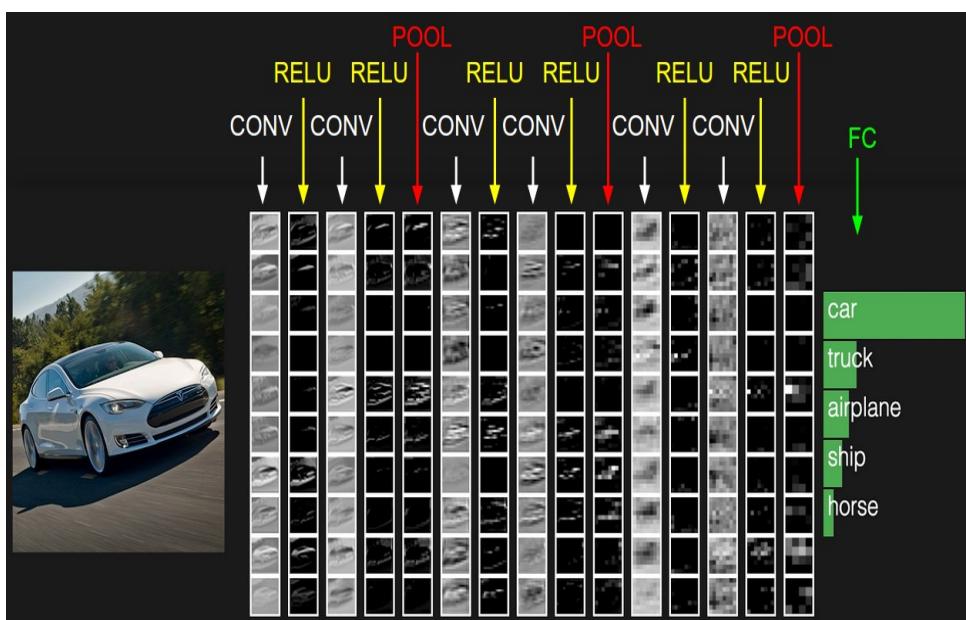
---

## 卷积神经网络(CNN)前向传播算法

在卷积神经网络(CNN)模型结构中，我们对CNN的模型结构做了总结，这里我们就在CNN的模型基础上，看看CNN的前向传播算法是什么样子的。重点会和传统的DNN比较讨论。

## 1. 回顾CNN的结构

在上一篇里，我们已经讲到了CNN的结构，包括输出层，若干的卷积层+ReLU激活函数，若干的池化层，DNN全连接层，以及最后的用Softmax激活函数的输出层。这里我们用一个彩色的汽车样本的图像识别再从感官上回顾下CNN的结构。图中的CONV即为卷积层，POOL即为池化层，而FC即为DNN全连接层，包括了我们上面最后的用Softmax激活函数的输出层。



从上图可以看出，要理顺CNN的前向传播算法，重点是输入层的前向传播，卷积层的前向传播以及池化层的前向传播。而DNN全连接层和用Softmax激活函数的输出层的前向传播算法我们在讲DNN时已经讲到了。

## 2. CNN输入层前向传播到卷积层

输入层的前向传播是CNN前向传播算法的第一步。一般输入层对应的都是卷积层，因此我们标题是输入层前向传播到卷积层。

我们这里还是以图像识别为例。

先考虑最简单的，样本都是二维的黑白图片。这样输入层 $X$ 就是一个矩阵，矩阵的值等于图片的各个像素位置的值。这时和卷积层相连的卷积核 $W$ 就也是矩阵。

如果样本都是有RGB的彩色图片，这样输入 $X$ 就是3个矩阵，即分别对应R，G和B的矩阵，或者说是一个张量。这时和卷积层相连的卷积核 $W$ 就也是张量，对应的最后一维的维度为3.即每个卷积核都是3个子矩阵组成。

同样的方法，对于3D的彩色图片之类的样本，我们的输入 $X$ 可以是4维，5维的张量，那么对应的卷积核 $W$ 也是个高维的张量。

不管维度多高，对于我们的输入，前向传播的过程可以表示为：

$$a^2 = \sigma(z^2) = \sigma(a^1 * W^2 + b^2)$$

其中，上标代表层数，星号代表卷积，而 $b$ 代表我们的偏倚， $\sigma$ 为激活函数，这里一般都是ReLU。

和DNN的前向传播比较一下，其实形式非常的像，只是我们这儿是张量的卷积，而不是矩阵的乘法。同时由于 $W$ 是张量，那么同样的位置， $W$ 参数的个数就比DNN多很多了。

为了简化我们的描述，本文后面如果没有特殊说明，我们都默认输入是3维的张量，即用RGB可以表示的彩色图片。

这里需要我们自己定义的CNN模型参数是：

- 1 ) 一般我们的卷积核不止一个，比如有K个，那么我们输入层的输出，或者说第二层卷积层的对应的输入就K个。
- 2 ) 卷积核中每个子矩阵的的大小，一般我们都用子矩阵为方阵的卷积核，比如FxF的子矩阵。
- 3 ) 填充padding（以下简称P），我们卷积的时候，为了可以更好的识别边缘，一般都会在输入矩阵在周围加上若干圈的0再进行卷积，加多少圈则P为多少。
- 4 ) 步幅stride（以下简称S），即在卷积过程中每次移动的像素距离大小。

这些参数我们在上一篇都有讲述。

## 3. 隐藏层前向传播到卷积层

现在我们再来看普通隐藏层前向传播到卷积层时的前向传播算法。

假设隐藏层的输出是M个矩阵对应的三维张量，则输出到卷积层的卷积核也是M个子矩阵对应的三维张量。这时表达式和输入层的很像，也是

$$a^l = \sigma(z^l) = \sigma(a^{l-1} * W^l + b^l)$$

其中，上标代表层数，星号代表卷积，而b代表我们的偏倚， $\sigma$ 为激活函数，这里一般都是ReLU。

也可以写成M个子矩阵子矩阵卷积后对应位置相加的形式，即：

$$a^l = \sigma(z^l) = \sigma\left(\sum_{k=1}^M z_k^l\right) = \sigma\left(\left(\sum_{k=1}^M (a_k^{l-1} * W_k^l) + b^l\right)\right)$$

和上一节唯一的区别仅仅在于，这里的输入是隐藏层来的，而不是我们输入的原始图片样本形成的矩阵。

需要我们定义的CNN模型参数也和上一节一样，这里我们需要定义卷积核的个数K，卷积核子矩阵的维度F，填充大小P以及步幅S。

## 4. 隐藏层前向传播到池化层

池化层的处理逻辑是比较简单的，我们的目的就是对输入的矩阵进行缩小概括。比如输入的若干矩阵是NxN维的，而我们的池化大小是kxk的区域，则输出的矩阵都是 $\frac{N}{k} \times \frac{N}{k}$ 维的。

这里需要我们定义的CNN模型参数是：

- 1 ) 池化区域的大小k
- 2 ) 池化的标准，一般是MAX或者Average。

## 5. 隐藏层前向传播到全连接层

由于全连接层就是普通的DNN模型结构，因此我们可以直接使用DNN的前向传播算法逻辑，即：

$$a^l = \sigma(z^l) = \sigma(W^l a^{l-1} + b^l)$$

这里的激活函数一般是sigmoid或者tanh。

经过了若干全连接层之后，最后的一层为Softmax输出层。此时输出层和普通的全连接层唯一的区别是，激活函数是softmax函数。

这里需要我们定义的CNN模型参数是：

- 1 ) 全连接层的激活函数
- 2 ) 全连接层各层神经元的个数

## 6. CNN前向传播算法小结

有了上面的基础，我们现在总结下CNN的前向传播算法。

输入：1个图片样本，CNN模型的层数L和所有隐藏层的类型，对于卷积层，要定义卷积核的大小K，卷积核子矩阵的维度F，填充大小P，步幅S。对于池化层，要定义池化区域大小k和池化标准（MAX或Average），对于全连接层，要定义全连接层的激活函数（输出层除外）和各层的神经元个数。

输出：CNN模型的输出 $a^L$

1) 根据输入层的填充大小P，填充原始图片的边缘，得到输入张量 $a^1$ 。

2 ) 初始化所有隐藏层的参数 $W, b$

3 ) for  $l=2$  to  $L-1$ :

a) 如果第 $l$ 层是卷积层，则输出为

$$a^l = \text{ReLU}(z^l) = \text{ReLU}(a^{l-1} * W^l + b^l)$$

b) 如果第 $l$ 层是池化层,则输出为 $a^l = pool(a^{l-1})$ , 这里的pool指按照池化区域大小k和池化标准将输入张量缩小的过程。

c) 如果第 $l$ 层是全连接层,则输出为

$$a^l = \sigma(z^l) = \sigma(W^l a^{l-1} + b^l)$$

4)对于输出层第L层:

$$a^L = softmax(z^L) = softmax(W^L a^{L-1} + b^L)$$

以上就是CNN前向传播算法的过程总结。有了CNN前向传播算法的基础, 我们后面再来理解CNN的反向传播算法就简单多了。下一篇我们来讨论CNN的反向传播算法。

---

## 卷积神经网络(CNN)反向传播算法

在卷积神经网络(CNN)前向传播算法中，我们对CNN的前向传播算法做了总结，基于CNN前向传播算法的基础，我们下面就对CNN的反向传播算法做一个总结。在阅读本文前，建议先研究DNN的反向传播算法：[深度神经网络 \( DNN \) 反向传播算法\(BP\)](#)

### 1. 回顾DNN的反向传播算法

我们首先回顾DNN的反向传播算法。在DNN中，我们是首先计算出输出层的 $\delta^L$ ：

$$\delta^L = \frac{\partial J(W, b)}{\partial z^L} = \frac{\partial J(W, b)}{\partial a^L} \odot \sigma'(z^L)$$

利用数学归纳法，用 $\delta^{l+1}$ 的值一步步的向前求出第 $l$ 层的 $\delta^l$ ，表达式为：

$$\delta^l = \delta^{l+1} \frac{\partial z^{l+1}}{\partial z^l} = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

有了 $\delta^l$ 的表达式，从而求出 $W, b$ 的梯度表达式：

$$\begin{aligned} \frac{\partial J(W, b)}{\partial W^l} &= \frac{\partial J(W, b, x, y)}{\partial z^l} \frac{\partial z^l}{\partial W^l} = \delta^l (a^{l-1})^T \\ \frac{\partial J(W, b, x, y)}{\partial b^l} &= \frac{\partial J(W, b)}{\partial z^l} \frac{\partial z^l}{\partial b^l} = \delta^l \end{aligned}$$

有了 $W, b$ 梯度表达式，就可以用梯度下降法来优化 $W, b$ ，求出最终的所有 $W, b$ 的值。

现在我们想把同样的思想用到CNN中，很明显，CNN有些不同的地方，不能直接去套用DNN的反向传播算法的公式。

### 2. CNN的反向传播算法思想

要套用DNN的反向传播算法到CNN，有几个问题需要解决：

1 ) 池化层没有激活函数，这个问题倒比较好解决，我们可以令池化层的激活函数为 $\sigma(z) = z$ ，即激活后就是自己本身。这样池化层激活函数的导数为1。

2 ) 池化层在前向传播的时候，对输入进行了压缩，那么我们现在需要向前反向推导 $\delta^{l-1}$ ，这个推导方法和DNN完全不同。

3) 卷积层是通过张量卷积，或者说若干个矩阵卷积求和而得的当前层的输出，这和DNN很不相同，DNN的全连接层是直接进行矩阵乘法得到当前层的输出。这样在卷积层反向传播的时候，上一层的 $\delta^{l-1}$ 递推计算方法肯定有所不同。

4 ) 对于卷积层，由于 $W$ 使用的运算是卷积，那么从 $\delta^l$ 推导出该层的所有卷积核的 $W, b$ 的方式也不同。

从上面可以看出，问题1比较好解决，但是问题2,3,4就需要好好的动一番脑筋了，而问题2,3,4也是解决CNN反向传播算法的关键所在。另外大家要注意到的是，DNN中的 $a_l, z_l$ 都只是一个向量，而我们CNN中的 $a_l, z_l$ 都是一个张量，这个张量是三维的，即由若干个输入的子矩阵组成。

下面我们就针对问题2,3,4来一步步研究CNN的反向传播算法。

在研究过程中，需要注意的是，由于卷积层可以有多个卷积核，各个卷积核的处理方法是完全相同且独立的，为了简化算法公式的复杂度，我们下面提到卷积核都是卷积层中若干卷积核中的一个。

### 3. 已知池化层的 $\delta^l$ ，推导上一隐藏层的 $\delta^{l-1}$

我们首先解决上面的问题2，如果已知池化层的 $\delta^l$ ，推导出上一隐藏层的 $\delta^{l-1}$ 。

在前向传播算法时，池化层一般我们会用MAX或者Average对输入进行池化，池化的区域大小已知。现在我们反过来，要从缩小后的误差 $\delta^l$ ，还原前一次较大区域对应的误差。

在反向传播时，我们首先会把 $\delta^l$ 的所有子矩阵矩阵大小还原成池化之前的大小，然后如果是MAX，则把 $\delta^l$ 的所有子矩阵的各个池化局部的值放在之前做前向传播算法得到最大值的位置。如果是Average，则把 $\delta^l$ 的所有子矩阵的各个池化局部的值取平均后放在还原后的子矩阵位置。这个过程一般叫做upsample。

用一个例子可以很方便的表示：假设我们的池化区域大小是 $2 \times 2$ 。 $\delta^l$ 的第k个子矩阵为：

$$\delta_k^l = \begin{pmatrix} 2 & 8 \\ 4 & 6 \end{pmatrix}$$

由于池化区域为2x2，我们先讲 $\delta_k^l$ 做还原，即变成：

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 0 & 4 & 6 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

如果是MAX，假设我们之前在前向传播时记录的最大值位置分别是左上，右下，右上，左下，则转换后的矩阵为：

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 \end{pmatrix}$$

如果是Average，则进行平均：转换后的矩阵为：

$$\begin{pmatrix} 0.5 & 0.5 & 2 & 2 \\ 0.5 & 0.5 & 2 & 2 \\ 1 & 1 & 1.5 & 1.5 \\ 1 & 1 & 1.5 & 1.5 \end{pmatrix}$$

这样我们就得到了上一层  $\frac{\partial J(W, b)}{\partial a_k^{l-1}}$  的值，要得到 $\delta_k^{l-1}$ ：

$$\delta_k^{l-1} = \frac{\partial J(W, b)}{\partial a_k^{l-1}} \frac{\partial a_k^l}{\partial z_k^{l-1}} = \text{upsample}(\delta_k^l) \odot \sigma'(z_k^{l-1})$$

其中，upsample函数完成了池化误差矩阵放大与误差重新分配的逻辑。

我们概括下，对于张量 $\delta^{l-1}$ ，我们有：

$$\delta^{l-1} = \text{upsample}(\delta^l) \odot \sigma'(z^{l-1})$$

## 4. 已知卷积层的 $\delta^l$ ，推导上一隐藏层的 $\delta^{l-1}$

对于卷积层的反向传播，我们首先回忆下卷积层的前向传播公式：

$$a^l = \sigma(z^l) = \sigma(a^{l-1} * W^l + b^l)$$

其中 $n\_in$ 为上一隐藏层的输入子矩阵个数。

在DNN中，我们知道 $\delta^{l-1}$ 和 $\delta^l$ 的递推关系为：

$$\delta^l = \frac{\partial J(W, b)}{\partial z^l} = \frac{\partial J(W, b)}{\partial z^{l+1}} \frac{\partial z^{l+1}}{\partial z^l} = \delta^{l+1} \frac{\partial z^{l+1}}{\partial z^l}$$

因此要推导出 $\delta^{l-1}$ 和 $\delta^l$ 的递推关系，必须计算  $\frac{\partial z^l}{\partial z^{l-1}}$  的梯度表达式。

注意到 $z^l$ 和 $z^{l-1}$ 的关系为：

$$z^l = a^{l-1} * W^l + b^l = \sigma(z^{l-1}) * W^l + b^l$$

因此我们有：

$$\delta^{l-1} = \delta^l \frac{\partial z^l}{\partial z^{l-1}} = \delta^l * \text{rot180}(W^l) \odot \sigma'(z^{l-1})$$

这里的式子其实和DNN的类似，区别在于对于含有卷积的式子求导时，卷积核被旋转了180度。即式子中的 $\text{rot180}()$ ，翻转180度的意思是上下翻转一次，接着左右翻转一次。在DNN中这里只是矩阵的转置。那么为什么呢？由于这里都是张量，直接推演参数太多了。我们以一个简单的例子说明为啥这里求导后卷积核要翻转。

假设我们 $l-1$ 层的输出 $a^{l-1}$ 是一个 $3 \times 3$ 矩阵，第 $l$ 层的卷积核 $W^l$ 是一个 $2 \times 2$ 矩阵，采用1像素的步幅，则输出 $z^l$ 是一个 $2 \times 2$ 的矩阵。我们简化 $b^l$ 都是0，则有

$$a^{l-1} * W^l = z^l$$

我们列出 $a, W, z$ 的矩阵表达式如下：

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} = \begin{pmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{pmatrix}$$

利用卷积的定义，很容易得出：

$$z_{11} = a_{11}w_{11} + a_{12}w_{12} + a_{21}w_{21} + a_{22}w_{22}$$

$$z_{12} = a_{12}w_{11} + a_{13}w_{12} + a_{22}w_{21} + a_{23}w_{22}$$

$$z_{21} = a_{21}w_{11} + a_{22}w_{12} + a_{31}w_{21} + a_{32}w_{22}$$

$$z_{22} = a_{22}w_{11} + a_{23}w_{12} + a_{32}w_{21} + a_{33}w_{22}$$

接着我们模拟反向求导：

$$\nabla a^{l-1} = \frac{\partial J(W, b)}{\partial a^{l-1}} = \frac{\partial J(W, b)}{\partial z^l} \frac{\partial z^l}{\partial a^{l-1}} = \delta^l \frac{\partial z^l}{\partial a^{l-1}}$$

从上式可以看出，对于 $a^{l-1}$ 的梯度误差 $\nabla a^{l-1}$ ，等于第 $l$ 层的梯度误差乘以 $\frac{\partial z^l}{\partial a^{l-1}}$ ，而 $\frac{\partial z^l}{\partial a^{l-1}}$ 对应上面的例子中相关联的 $w$ 的值。假设我们的 $z$ 矩阵对应的反向传播误差是 $\delta_{11}, \delta_{12}, \delta_{21}, \delta_{22}$ 组成的 $2 \times 2$ 矩阵，则利用上面梯度的式子和4个等式，我们可以分别写出 $\nabla a^{l-1}$ 的9个标量的梯度。

比如对于 $a_{11}$ 的梯度，由于在4个等式中 $a_{11}$ 只和 $z_{11}$ 有乘积关系，从而我们有：

$$\nabla a_{11} = \delta_{11}w_{11}$$

对于 $a_{12}$ 的梯度，由于在4个等式中 $a_{12}$ 和 $z_{12}, z_{21}$ 有乘积关系，从而我们有：

$$\nabla a_{12} = \delta_{11}w_{12} + \delta_{12}w_{11}$$

同样的道理我们得到：

$$\nabla a_{13} = \delta_{12}w_{12}$$

$$\nabla a_{21} = \delta_{11}w_{21} + \delta_{21}w_{11}$$

$$\nabla a_{22} = \delta_{11}w_{22} + \delta_{12}w_{21} + \delta_{21}w_{12} + \delta_{22}w_{11}$$

$$\nabla a_{23} = \delta_{12}w_{22} + \delta_{22}w_{12}$$

$$\nabla a_{31} = \delta_{21}w_{21}$$

$$\nabla a_{32} = \delta_{21}w_{22} + \delta_{22}w_{21}$$

$$\nabla a_{33} = \delta_{22}w_{22}$$

这上面9个式子其实可以用一个矩阵卷积的形式表示，即：

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \delta_{11} & \delta_{12} & 0 \\ 0 & \delta_{21} & \delta_{22} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} w_{22} & w_{21} \\ w_{12} & w_{11} \end{pmatrix} = \begin{pmatrix} \nabla a_{11} & \nabla a_{12} & \nabla a_{13} \\ \nabla a_{21} & \nabla a_{22} & \nabla a_{23} \\ \nabla a_{31} & \nabla a_{32} & \nabla a_{33} \end{pmatrix}$$

为了符合梯度计算，我们在误差矩阵周围填充了一圈0，此时我们将卷积核翻转后和反向传播的梯度误差进行卷积，就得到了前一次的梯度误差。这个例子直观的介绍了为什么对含有卷积的式子求导时，卷积核要翻转180度的原因。

以上就是卷积层的误差反向传播过程。

## 5. 已知卷积层的 $\delta^l$ ，推导该层的 $W, b$ 的梯度

好了，我们现在可以递推出每一层的梯度误差 $\delta^l$ 了，对于全连接层，可以按DNN的反向传播算法求该层 $W, b$ 的梯度，而池化层并没有 $W, b$ ，也不用求 $W, b$ 的梯度。只有卷积层的 $W, b$ 需要求出。

注意到卷积层 $z$ 和 $W, b$ 的关系为：

$$z^l = a^{l-1} * W^l + b$$

因此我们有：

$$\frac{\partial J(W, b)}{\partial W^l} = \frac{\partial J(W, b)}{\partial z^l} \frac{\partial z^l}{\partial W^l} = \delta^l * rot180(a^{l-1})$$

由于我们有上一节的基础，大家应该清楚为什么这里求导后 $a^{l-1}$ 要旋转180度了。

而对于 $b$ ，则稍微有些特殊，因为 $\delta^l$ 是三维张量，而 $b$ 只是一个向量，不能像DNN那样直接和 $\delta^l$ 相等。通常的做法是将 $\delta^l$ 的各个子矩阵的项分别求和，得到一个误差向量，即为 $b$ 的梯度：

$$\frac{\partial J(W, b)}{\partial b} = \sum_{u,v} (\delta^l)_{u,v}$$

## 6. CNN反向传播算法总结

现在我们总结下CNN的反向传播算法，以最基本的批量梯度下降法为例来描述反向传播算法。

输入：m个图片样本，CNN模型的层数L和所有隐藏层的类型，对于卷积层，要定义卷积核的大小K，卷积核矩阵的维度F，填充大小P，步幅S。对于池化层，要定义池化区域大小k和池化标准（MAX或Average），对于全连接层，要定义全连接层的激活函数（输出层除外）和各层的神经元个数。梯度迭代参数迭代步长 $\alpha$ ，最大迭代次数MAX与停止迭代阈值 $\epsilon$

输出：CNN模型各隐藏层与输出层的 $W, b$

1) 初始化各隐藏层与输出层的各 $W, b$ 的值为一个随机值。

2 ) for iter to 1 to MAX :

2-1) for i =1 to m :

a) 将CNN输入 $a^1$ 设置为 $x_i$ 对应的张量

b) for  $l=2$  to L-1 , 根据下面3种情况进行前向传播算法计算：

b-1) 如果当前是全连接层：则有 $a^{i,l} = \sigma(z^{i,l}) = \sigma(W^l a^{i,l-1} + b^l)$

b-2) 如果当前是卷积层：则有 $a^{i,l} = \sigma(W^l * a^{i,l-1} + b^l)$

b-3) 如果当前是池化层：则有 $a^{i,l} = pool(a^{i,l-1})$ , 这里的pool指按照池化区域大小k和池化标准将输入张量缩小的过程。

c) 对于输出层第L层:  $a^{i,L} = softmax(z^{i,L}) = softmax(W^L a^{i,L-1} + b^L)$

c) 通过损失函数计算输出层的 $\delta^{i,L}$

d) for  $l= L-1$  to 2, 根据下面3种情况进行反向传播算法计算:

d-1) 如果当前是全连接层： $\delta^{i,l} = (W^{l+1})^T \delta^{i,l+1} \odot \sigma'(z^{i,l})$

d-2) 如果当前是卷积层： $\delta^{i,l} = \delta^{i,l+1} * rot180(W^{l+1}) \odot \sigma'(z^{i,l})$

d-3) 如果当前是池化层： $\delta^{i,l} = upsample(\delta^{i,l+1}) \odot \sigma'(z^{i,l})$

2-2) for  $l = 2$  to L , 根据下面2种情况更新第 $l$ 层的 $W^l, b^l$ :

2-2-1) 如果当前是全连接层： $W^l = W^l - \alpha \sum_{i=1}^m \delta^{i,l} (a^{i,l-1})^T, b^l = b^l - \alpha \sum_{i=1}^m \delta^{i,l}$

2-2-2) 如果当前是卷积层，对于每一个卷积核有： $W^l = W^l - \alpha \sum_{i=1}^m \delta^{i,l} * rot180(a^{i,l-1}),$

$$b^l = b^l - \alpha \sum_{i=1}^m \sum_{u,v} (\delta^{i,l})_{u,v}$$

2-3) 如果所有 $W, b$ 的变化值都小于停止迭代阈值 $\epsilon$ ，则跳出迭代循环到步骤3。

3 ) 输出各隐藏层与输出层的线性关系系数矩阵 $W$ 和偏倚向量 $b$ 。

( 欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com )

## 参考资料：

- 1 ) [Neural Networks and Deep Learning](#) by By Michael Nielsen
- 2 ) [Deep Learning](#), book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville
- 3 ) [UFLDL Tutorial](#)
- 4 ) [CS231n Convolutional Neural Networks for Visual Recognition, Stanford](#)

## 循环神经网络(RNN)模型与前向反向传播算法

在前面我们讲到了DNN，以及DNN的特例CNN的模型和前向反向传播算法，这些算法都是前向反馈的，模型的输出和模型本身没有关联关系。今天我们就讨论另一类输出和模型间有反馈的神经网络：循环神经网络(Recurrent Neural Networks，以下简称RNN)，它广泛的用于自然语言处理中的语音识别，手写书别以及机器翻译等领域。

### 1. RNN概述

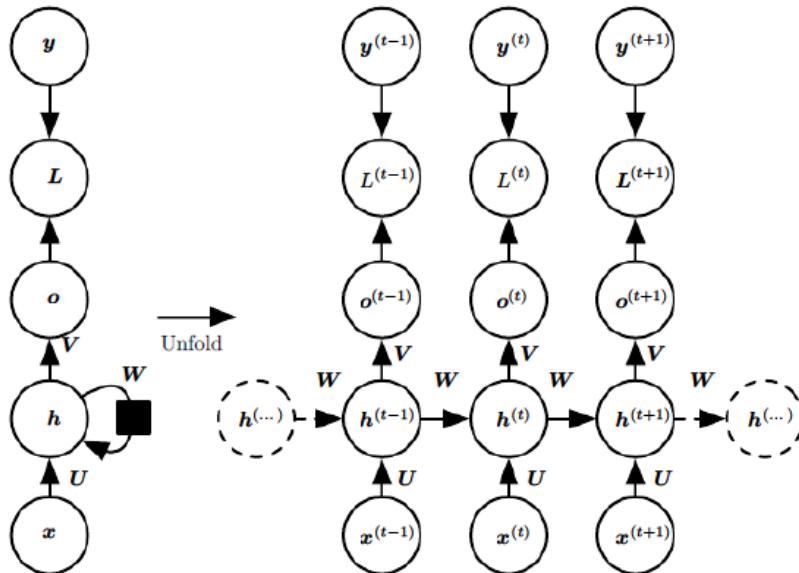
在前面讲到的DNN和CNN中，训练样本的输入和输出是比较确定的。但是有一类问题DNN和CNN不好解决，就是训练样本输入是连续的序列，且序列的长短不一，比如基于时间的序列：一段段连续的语音，一段段连续的手写文字。这些序列比较长，且长度不一，比较难直接的拆分成一个个独立的样本来通过DNN/CNN进行训练。

而对于这类问题，RNN则比较擅长。那么RNN是怎么做到的呢？RNN假设我们的样本是基于序列的。比如是从序列索引1到序列索引 $T$ 的。对于这其中的任意序列索引 $t$ ，它对应的输入是对应的样本序列中的 $x^{(t)}$ 。而模型在序列索引 $t$ 位置的隐藏状态 $h^{(t)}$ ，则由 $x^{(t)}$ 和在 $t-1$ 位置的隐藏状态 $h^{(t-1)}$ 共同决定。在任意序列索引 $t$ ，我们也有对应的模型预测输出 $o^{(t)}$ 。通过预测输出 $o^{(t)}$ 和训练序列真实输出 $y^{(t)}$ ，以及损失函数 $L^{(t)}$ ，我们就可以用DNN类似的方法来训练模型，接着用来预测测试序列中的一些位置的输出。

下面我们来看看RNN的模型。

### 2. RNN模型

RNN模型有比较多的变种，这里介绍最主流的RNN模型结构如下：



上图中左边是RNN模型没有按时间展开的图，如果按时间序列展开，则是上图中的右边部分。我们重点观察右边部分的图。

这幅图描述了在序列索引 $t$ 附近RNN的模型。其中：

- 1 )  $x^{(t)}$ 代表在序列索引 $t$ 时训练样本的输入。同样的， $x^{(t-1)}$ 和 $x^{(t+1)}$ 代表在序列索引 $t-1$ 和 $t+1$ 时训练样本的输入。
- 2 )  $h^{(t)}$ 代表在序列索引 $t$ 时模型的隐藏状态。 $h^{(t)}$ 由 $x^{(t)}$ 和 $h^{(t-1)}$ 共同决定。
- 3 )  $o^{(t)}$ 代表在序列索引 $t$ 时模型的输出。 $o^{(t)}$ 只由模型当前的隐藏状态 $h^{(t)}$ 决定。
- 4 )  $L^{(t)}$ 代表在序列索引 $t$ 时模型的损失函数。
- 5 )  $y^{(t)}$ 代表在序列索引 $t$ 时训练样本序列的真实输出。
- 6 )  $U, W, V$ 这三个矩阵是我们的模型的线性关系参数，它在整个RNN网络中是共享的，这点和DNN很不相同。也正因为是共享了，它体现了RNN的模型的“循环反馈”的思想。

### 3. RNN前向传播算法

有了上面的模型，RNN的前向传播算法就很容易得到了。

对于任意一个序列索引号 $t$ ，我们隐藏状态 $h^{(t)}$ 由 $x^{(t)}$ 和 $h^{(t-1)}$ 得到：

$$h^{(t)} = \sigma(z^{(t)}) = \sigma(Ux^{(t)} + Wh^{(t-1)} + b)$$

其中 $\sigma$ 为RNN的激活函数，一般为 $tanh$ ， $b$ 为线性关系的偏倚。

序列索引号 $t$ 时模型的输出 $o^{(t)}$ 的表达式比较简单：

$$o^{(t)} = Vh^{(t)} + c$$

在最终在序列索引号 $t$ 时我们的预测输出为：

$$\hat{y}^{(t)} = \sigma(o^{(t)})$$

通常由于RNN是识别类的分类模型，所以上面这个激活函数一般是softmax。

通过损失函数 $L^{(t)}$ ，比如对数似然损失函数，我们可以量化模型在当前位置的损失，即 $\hat{y}^{(t)}$ 和 $y^{(t)}$ 的差距。

### 4. RNN反向传播算法推导

有了RNN前向传播算法的基础，就容易推导出RNN反向传播算法的流程了。RNN反向传播算法的思路和DNN是一样的，即通过梯度下降法一轮轮的迭代，得到合适的RNN模型参数 $U, W, V, b, c$ 。由于我们是基于时间反向传播，所以RNN的反向传播有时也叫做BPTT(back-propagation through time)。当然这里的BPTT和DNN也有很大的不同点，即这里所有的 $U, W, V, b, c$ 在序列的各个位置是共享的，反向传播时我们更新的是相同的参数。

为了简化描述，这里的损失函数我们为对数损失函数，输出的激活函数为softmax函数，隐藏层的激活函数为tanh函数。

对于RNN，由于我们在序列的每个位置都有损失函数，因此最终的损失 $L$ 为：

$$L = \sum_{t=1}^{\tau} L^{(t)}$$

其中 $V, c$ 的梯度计算是比较简单的：

$$\begin{aligned}\frac{\partial L}{\partial c} &= \sum_{t=1}^{\tau} \frac{\partial L^{(t)}}{\partial c} = \sum_{t=1}^{\tau} \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial c} = \sum_{t=1}^{\tau} \hat{y}^{(t)} - y^{(t)} \\ \frac{\partial L}{\partial V} &= \sum_{t=1}^{\tau} \frac{\partial L^{(t)}}{\partial V} = \sum_{t=1}^{\tau} \frac{\partial L^{(t)}}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial V} = \sum_{t=1}^{\tau} (\hat{y}^{(t)} - y^{(t)}) (h^{(t)})^T\end{aligned}$$

但是 $W, U, b$ 的梯度计算就比较的复杂了。从RNN的模型可以看出，在反向传播时，在某一序列位置 $t$ 的梯度损失由当前位置的输出对应的梯度损失和序列索引位置 $t+1$ 时的梯度损失两部分共同决定。对于 $W$ 在某一序列位置 $t$ 的梯度损失需要反向传播一步步的计算。我们定义序列索引 $t$ 位置的隐藏状态的梯度为：

$$\delta^{(t)} = \frac{\partial L}{\partial h^{(t)}}$$

这样我们可以像DNN一样从 $\delta^{(t+1)}$ 递推 $\delta^{(t)}$ 。

$$\delta^{(t)} = \frac{\partial L}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial h^{(t)}} + \frac{\partial L}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial h^{(t)}} = V^T (\hat{y}^{(t)} - y^{(t)}) + W^T \delta^{(t+1)} \text{diag}(1 - (h^{(t+1)})^2)$$

对于 $\delta^{(\tau)}$ ，由于它的后面没有其他的序列索引了，因此有：

$$\delta^{(\tau)} = \frac{\partial L}{\partial o^{(\tau)}} \frac{\partial o^{(\tau)}}{\partial h^{(\tau)}} = V^T (\hat{y}^{(\tau)} - y^{(\tau)})$$

有了 $\delta^{(t)}$ ，计算 $W, U, b$ 就容易了，这里给出 $W, U, b$ 的梯度计算表达式：

$$\begin{aligned}\frac{\partial L}{\partial W} &= \sum_{t=1}^{\tau} \frac{\partial L}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial W} = \sum_{t=1}^{\tau} \text{diag}(1 - (h^{(t)})^2) \delta^{(t)} (h^{(t-1)})^T \\ \frac{\partial L}{\partial b} &= \sum_{t=1}^{\tau} \frac{\partial L}{\partial h^{(t)}} \frac{\partial b}{\partial b} = \sum_{t=1}^{\tau} \text{diag}(1 - (h^{(t)})^2) \delta^{(t)} \\ \frac{\partial L}{\partial U} &= \sum_{t=1}^{\tau} \frac{\partial L}{\partial h^{(t)}} \frac{\partial U}{\partial U} = \sum_{t=1}^{\tau} \text{diag}(1 - (h^{(t)})^2) \delta^{(t)} (x^{(t)})^T\end{aligned}$$

除了梯度表达式不同，RNN的反向传播算法和DNN区别不大，因此这里就不再重复总结了。

### 5. RNN小结

上面总结了通用的RNN模型和前向反向传播算法。当然，有些RNN模型会有些不同，自然前向反向传播的公式会有些不一样，但是原理基本类似。

RNN虽然理论上可以很漂亮的解决序列数据的训练，但是它也像DNN一样有梯度消失时的问题，当序列很长的时候问题尤其严重。因此，上面的RNN模型一般不能直接用于应用领域。在语音识别，手写书别以及机器翻译等NLP领域实际应用比较广泛的是基于RNN模型的一个特例LSTM，下一篇我们就来讨论LSTM模型。

( 欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com )

## 参考资料：

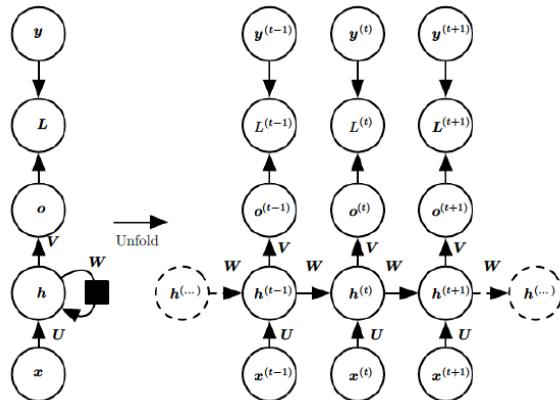
- 1 ) [Neural Networks and Deep Learning by By Michael Nielsen](#)
  - 2 ) [Deep Learning, book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville](#)
  - 3 ) [UFLDL Tutorial](#)
  - 4 ) [CS231n Convolutional Neural Networks for Visual Recognition, Stanford](#)
-

## LSTM模型与前向反向传播算法

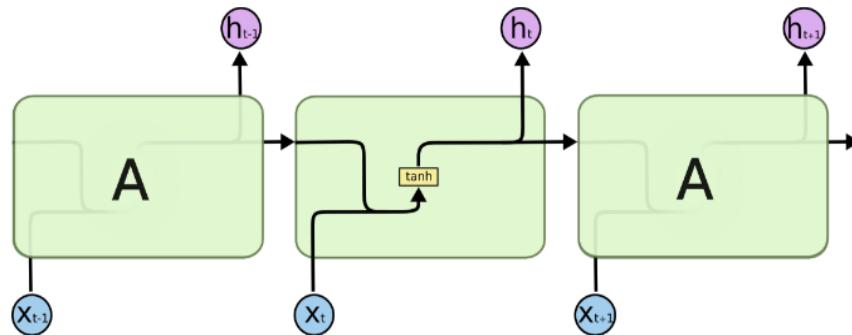
在循环神经网络(RNN)模型与前向反向传播算法中，我们总结了对RNN模型做了总结。由于RNN也有梯度消失的问题，因此很难处理长序列的数据，大牛们对RNN做了改进，得到了RNN的特例LSTM ( Long Short-Term Memory )，它可以在避免常规RNN的梯度消失，因此在工业界得到了广泛的应用。下面我们就对LSTM模型做一个总结。

### 1. 从RNN到LSTM

在RNN模型里，我们讲到了RNN具有如下的结构，每个序列索引位置 $t$ 都有一个隐藏状态 $h^{(t)}$ 。

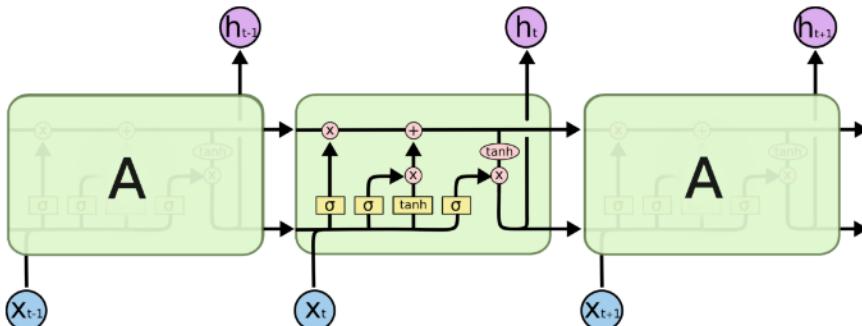


如果我们略去每层都有的 $o^{(t)}, L^{(t)}, y^{(t)}$ ，则RNN的模型可以简化成如下图的形式：



图中可以很清晰看出在隐藏状态 $h^{(t)}$ 由 $x^{(t)}$ 和 $h^{(t-1)}$ 得到。得到 $h^{(t)}$ 后一方面用于当前层的模型损失计算，另一方面用于计算下一层的 $h^{(t+1)}$ 。

由于RNN梯度消失的问题，大牛们对于序列索引位置 $t$ 的隐藏结构做了改进，可以说通过一些技巧让隐藏结构复杂了起来，来避免梯度消失的问题，这样的特殊RNN就是我们的LSTM。由于LSTM有很多的变种，这里我们以最常见的LSTM为例讲述。LSTM的结构如下图：

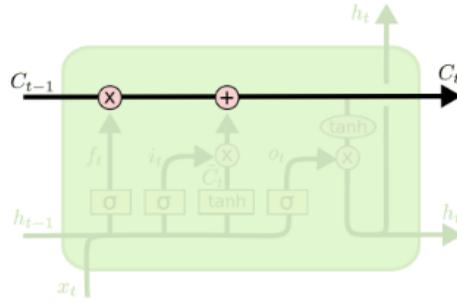


可以看到LSTM的结构要比RNN的复杂的多，真佩服牛人们怎么想出来这样的结构，然后这样居然就可以解决RNN梯度消失的问题？由于LSTM怎么可以解决梯度消失是一个比较难讲的问题，我不是很熟悉，这里就不多说，重点回到LSTM的模型本身。

## 2. LSTM模型结构剖析

上面我们给出了LSTM的模型结构，下面我们就一点点的剖析LSTM模型在每个序列索引位置t时刻的内部结构。

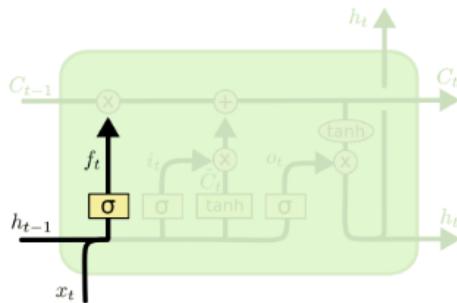
从上图中可以看出，在每个序列索引位置t时刻向前传播的除了和RNN一样的隐藏状态 $h^{(t)}$ ，还多了另一个隐藏状态，如图中上面的长横线。这个隐藏状态我们一般称为细胞状态(Cell State)，记为 $C^{(t)}$ 。如下图所示：



除了细胞状态，LSTM图中还有很多奇怪的结构，这些结构一般称之为门控结构(Gate)。LSTM在在每个序列索引位置t的门一般包括遗忘门，输入门和输出门三种。下面我们就来研究上图中LSTM的遗忘门，输入门和输出门以及细胞状态。

### 2.1 LSTM之遗忘门

遗忘门(forget gate)顾名思义，是控制是否遗忘的，在LSTM中即以一定的概率控制是否遗忘上一层的隐藏细胞状态。遗忘门子结构如下图所示：



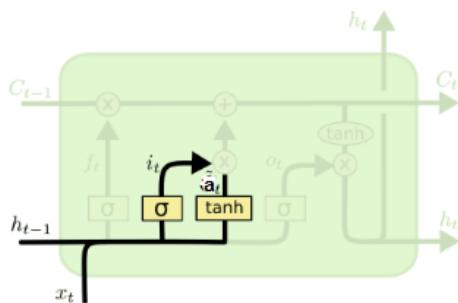
图中输入的有上一序列的隐藏状态 $h^{(t-1)}$ 和本序列数据 $x^{(t)}$ ，通过一个激活函数，一般是sigmoid，得到遗忘门的输出 $f^{(t)}$ 。由于sigmoid的输出 $f^{(t)}$ 在[0,1]之间，因此这里的输出 $f^{\wedge}(t)$ 代表了遗忘上一层隐藏细胞状态的概率。用数学表达式即为：

$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f)$$

其中 $W_f, U_f, b_f$ 为线性关系的系数和偏倚，和RNN中的类似。 $\sigma$ 为sigmoid激活函数。

### 2.2 LSTM之输入门

输入门(input gate)负责处理当前序列位置的输入，它的子结构如下图：



从图中可以看到输入门由两部分组成，第一部分使用了sigmoid激活函数，输出为 $i^{(t)}$ ，第二部分使用了tanh激活函数，输出为 $a^{(t)}$ ，两者的结果后面会相乘再去更新细胞状态。用数学表达式即为：

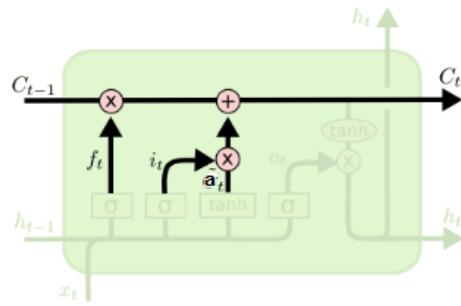
$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i)$$

$$a^{(t)} = \tanh(W_a h^{(t-1)} + U_a x^{(t)} + b_a)$$

其中 $W_i, U_i, b_i, W_a, U_a, b_a$ 为线性关系的系数和偏倚，和RNN中的类似。 $\sigma$ 为sigmoid激活函数。

## 2.3 LSTM之细胞状态更新

在研究LSTM输出门之前，我们要先看看LSTM之细胞状态。前面的遗忘门和输入门的结果都会作用于细胞状态 $C^{(t)}$ 。我们来看看从细胞状态 $C^{(t-1)}$ 如何得到 $C^{(t)}$ 。如下图所示：



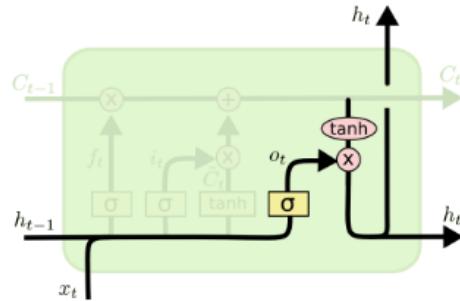
细胞状态 $C^{(t)}$ 由两部分组成，第一部分是 $C^{(t-1)}$ 和遗忘门输出 $f^{(t)}$ 的乘积，第二部分是输入门的 $i^{(t)}$ 和 $a^{(t)}$ 的乘积，即：

$$C^{(t)} = C^{(t-1)} \odot f^{(t)} + i^{(t)} \odot a^{(t)}$$

其中， $\odot$ 为Hadamard积，在DNN中也用到过。

## 2.4 LSTM之输出门

有了新的隐藏细胞状态 $C^{(t)}$ ，我们就可以来看输出门了，子结构如下：



从图中可以看出，隐藏状态 $h^{(t)}$ 的更新由两部分组成，第一部分是 $o^{(t)}$ ，它由上一序列的隐藏状态 $h^{(t-1)}$ 和本序列数据 $x^{(t)}$ ，以及激活函数sigmoid得到，第二部分由隐藏状态 $C^{(t)}$ 和tanh激活函数组成，即：

$$\begin{aligned} o^{(t)} &= \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o) \\ h^{(t)} &= o^{(t)} \odot \tanh(C^{(t)}) \end{aligned}$$

通过本节的剖析，相信大家对于LSTM的模型结构已经有了解了。当然，有些LSTM的结构和上面的LSTM图稍有不同，但是原理是完全一样的。

## 3. LSTM前向传播算法

现在我们来总结下LSTM前向传播算法。LSTM模型有两个隐藏状态 $h^{(t)}, C^{(t)}$ ，模型参数几乎是RNN的4倍，因为在多了 $W_f, U_f, b_f, W_a, U_a, b_a, W_i, U_i, b_i, W_o, U_o, b_o$ 这些参数。

前向传播过程在每个序列索引位置的过程为：

1 ) 更新遗忘门输出：

$$f^{(t)} = \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f)$$

2 ) 更新输入门两部分输出：

$$i^{(t)} = \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i)$$

$$a^{(t)} = \tanh(W_a h^{(t-1)} + U_a x^{(t)} + b_a)$$

3 ) 更新细胞状态：

$$C^{(t)} = C^{(t-1)} \odot f^{(t)} + i^{(t)} \odot a^{(t)}$$

4 ) 更新输出门输出：

$$o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o)$$

$$h^{(t)} = o^{(t)} \odot \tanh(C^{(t)})$$

5 ) 更新当前序列索引预测输出 :

$$\hat{y}^{(t)} = \sigma(Vh^{(t)} + c)$$

## 4. LSTM反向传播算法推导关键点

有了LSTM前向传播算法，推导反向传播算法就很容易了，思路和RNN的反向传播算法思路一致，也是通过梯度下降法迭代更新我们所有的参数，关键点在于计算所有参数基于损失函数的偏导数。

在RNN中，为了反向传播误差，我们通过隐藏状态 $h^{(t)}$ 的梯度 $\delta^{(t)}$ 一步步向前传播。在LSTM这里也类似。只不过我们这里有两个隐藏状态 $h^{(t)}$ 和 $C^{(t)}$ 。因此这里我们要定义两个 $\delta$ 来一步步反向传播，即：

$$\delta_h^{(t)} = \frac{\partial L}{\partial h^{(t)}}$$

$$\delta_C^{(t)} = \frac{\partial L}{\partial C^{(t)}}$$

而在最后的序列索引位置 $\tau$ 的 $\delta_h^{(\tau)}$ 和 $\delta_C^{(\tau)}$ 为：

$$\delta_h^{(\tau)} = \frac{\partial L}{\partial o^{(\tau)}} \frac{\partial o^{(\tau)}}{\partial h^{(\tau)}} = V^T (\hat{y}^{(\tau)} - y^{(\tau)})$$

$$\delta_C^{(\tau)} = \frac{\partial L}{\partial h^{(\tau)}} \frac{\partial h^{(\tau)}}{\partial C^{(\tau)}} = \delta_h^{(\tau)} \odot o^{(\tau)} \odot (1 - \tanh^2(C^{(\tau)}))$$

接着我们由 $\delta_h^{(t+1)}$ 和 $\delta_C^{(t+1)}$ 反向推导 $\delta_h^{(t)}$ 和 $\delta_C^{(t)}$

$\delta_h^{(t)}$ 的反向推导和RNN中的类似，因为它的梯度误差由前一层 $\delta_h^{(t+1)}$ 的梯度误差和本层的输出梯度误差两部分组成，

即：

$$\delta_h^{(t)} = \frac{\partial L}{\partial o^{(t)}} \frac{\partial o^{(t)}}{\partial h^{(t)}} + \frac{\partial L}{\partial h^{(t+1)}} \frac{\partial h^{(t+1)}}{\partial h^{(t)}} = V^T (\hat{y}^{(t)} - y^{(t)}) + W^T \delta^{(t+1)} \text{diag}(1 - (h^{(t+1)})^2)$$

而 $\delta_C^{(t)}$ 的反向梯度误差由前一层 $\delta_C^{(t+1)}$ 的梯度误差和本层的从 $h^{(t)}$ 传回来的梯度误差两部分组成，即：

$$\delta_C^{(t)} = \frac{\partial L}{\partial C^{(t+1)}} \frac{\partial C^{(t+1)}}{\partial C^{(t)}} + \frac{\partial L}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial C^{(t)}} = \delta_C^{(t+1)} \odot f^{(t+1)} + \delta_h^{(t)} \odot o^{(t)} \odot (1 - \tanh^2(C^{(t)}))$$

有了 $\delta_h^{(t)}$ 和 $\delta_C^{(t)}$ ，计算这一大堆参数的梯度就很容易了，这里只给出 $W_f$ 的梯度计算过程，其他的 $U_f, b_f, W_a, U_a, b_a, W_i, U_i, b_i, W_o, U_o, b_o, V$ 的梯度大家只要照搬就可以了。

$$\frac{\partial L}{\partial W_f} = \sum_{t=1}^T \frac{\partial L}{\partial C^{(t)}} \frac{\partial C^{(t)}}{\partial f^{(t)}} \frac{\partial f^{(t)}}{\partial W_f} = \delta_C^{(t)} \odot C^{(t-1)} \odot f^{(t)} (1 - f^{(t)}) (h^{(t-1)})^T$$

## 5. LSTM小结

LSTM虽然结构复杂，但是只要理顺了里面的各个部分和之间的关系，进而理解前向反向传播算法是不难的。当然实际应用中LSTM的难点不在前向反向传播算法，这些有算法库帮你搞定，模型结构和一大堆参数的调参才是让人头痛的问题。不过，理解LSTM模型结构仍然是高效使用的前提。

( 欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com )

## 参考资料：

- 1 ) [Neural Networks and Deep Learning](#) by By Michael Nielsen
- 2 ) [Deep Learning](#), book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville
- 3 ) [UFLDL Tutorial](#)
- 4 ) [Understanding-LSTMs](#)

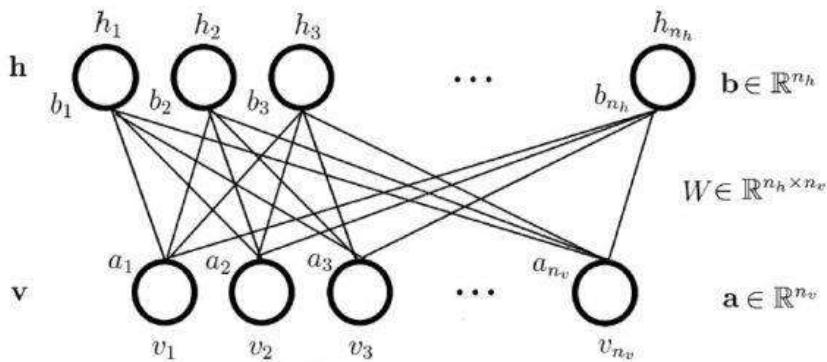
## 受限玻尔兹曼机 (RBM) 原理总结

在前面我们讲到了深度学习的两类神经网络模型的原理，第一类是前向的神经网络，即DNN和CNN。第二类是有反馈的神经网络，即RNN和LSTM。今天我们就总结下深度学习里的第三类神经网络模型：玻尔兹曼机。主要关注于这类模型中的受限玻尔兹曼机 ( Restricted Boltzmann Machine , 以下简称RBM )，RBM模型及其推广在工业界比如推荐系统中得到了广泛的应用。

### 1. RBM模型结构

玻尔兹曼机是一大类的神经网络模型，但是在实际应用中使用最多的则是RBM。RBM本身模型很简单，只是一个两层的神经网络，因此严格意义上不能算深度学习的范畴。不过深度玻尔兹曼机 ( Deep Boltzmann Machine , 以下简称DBM ) 可以看做是RBM的推广。理解了RBM再去研究DBM就不难了，因此本文主要关注于RBM。

回到RBM的结构，它是一个个两层的神经网络，如下图所示：



上面一层神经元组成隐藏层(hidden layer), 用 $h$ 向量表示隐藏层神经元的值。下面一层的神经元组成可见层(visible layer), 用 $v$ 向量表示可见层神经元的值。隐藏层和可见层之间是全连接的，这点和DNN类似，隐藏层神经元之间是独立的，可见层神经元之间也是独立的。连接权重可以用矩阵 $W$ 表示。和DNN的区别是，RBM不区分前向和反向，可见层的状态可以作用于隐藏层，而隐藏层的状态也可以作用于可见层。隐藏层的偏倚系数是向量 $b$ , 而可见层的偏倚系数是向量 $a$ 。

常用的RBM一般是二值的，即不管是隐藏层还是可见层，它们的神经元的取值只为0或者1。本文只讨论二值RBM。

总结下RBM模型结构的结构：主要是权重矩阵 $W$ ，偏倚系数向量 $a$ 和 $b$ ，隐藏层神经元状态向量 $h$ 和可见层神经元状态向量 $v$ 。

### 2. RBM概率分布

RBM是基于能量的概率分布模型。怎么理解呢？分两部分理解，第一部分是能量函数，第二部分是基于能量函数的概率分布函数。

对于给定的状态向量 $h$ 和 $v$ ，则RBM当前的能量函数可以表示为：

$$E(v, h) = -a^T v - b^T h - h^T W v$$

有了能量函数，则我们可以定义RBM的状态为给定 $v, h$ 的概率分布为：

$$P(v, h) = \frac{1}{Z} e^{-E(v, h)}$$

其中 $Z$ 为归一化因子，类似于softmax中的归一化因子，表达式为：

$$Z = \sum_{v,h} e^{-E(v, h)}$$

有了概率分布，我们现在来看条件分布 $P(h|v)$ :

$$P(h|v) = \frac{P(h, v)}{P(v)} \quad (1)$$

$$= \frac{1}{P(v)} \frac{1}{Z} \exp\{a^T v + b^T h + h^T W v\} \quad (2)$$

$$= \frac{1}{Z'} \exp\{b^T h + h^T W v\} \quad (3)$$

$$= \frac{1}{Z'} \exp\{\sum_{j=1}^{n_h} (b_j^T h_j + h_j^T W_{:,j} v_j)\} \quad (4)$$

$$= \frac{1}{Z'} \prod_{j=1}^{n_h} \exp\{b_j^T h_j + h_j^T W_{:,j} v_j\} \quad (5)$$

其中 $Z'$ 为新的归一化系数，表达式为：

$$Z' = \frac{1}{P(v)} \frac{1}{Z} \exp\{a^T v\}$$

同样的方式，我们也可以求出 $P(v|h)$ ，这里就不再列出了。

有了条件概率分布，现在我们来看看RBM的激活函数，提到神经网络，我们都绕不开激活函数，但是上面我们并没有提到。由于使用的是能量概率模型，RBM的基于条件分布的激活函数是很容易推导出来的。我们以 $P(h_j = 1|v)$ 为例推导如下。

$$P(h_j = 1|v) = \frac{P(h_j = 1|v)}{P(h_j = 1|v) + P(h_j = 0|v)} \quad (6)$$

$$= \frac{\exp\{b_j + W_{:,j} v_j\}}{\exp\{0\} + \exp\{b_j + W_{:,j} v_j\}} \quad (7)$$

$$= \frac{1}{1 + \exp\{-(b_j + W_{:,j} v_j)\}} \quad (8)$$

$$= \text{sigmoid}(b_j + W_{:,j} v_j) \quad (9)$$

从上面可以看出，RBM里从可见层到隐藏层用的其实就是sigmoid激活函数。同样的方法，我们也可以得到隐藏层到可见层用的也是sigmoid激活函数。即：

$$P(v_j = 1|h) = \text{sigmoid}(a_j + W_{:,j} h_j)$$

有了激活函数，我们就可以从可见层和参数推导出隐藏层的神经元的取值概率了。对于0,1取值的情况，则大于0.5即取值为1。从隐藏层和参数推导出可见的神经元的取值方法也是一样的。

### 3. RBM模型的损失函数与优化

RBM模型的关键就是求出我们模型中的参数 $W, a, b$ 。如果求出呢？对于训练集的 $m$ 个样本，RBM一般采用对数损失函数，即期望最小化下式：

$$L(W, a, b) = - \sum_{i=1}^m \ln(P(v^{(i)}))$$

对于优化过程，我们是首先想到的当然是梯度下降法来迭代求出 $W, a, b$ 。我们首先来看单个样本的梯度计算，单个样本的损失函数为： $-\ln(P(v))$ ，我们先看看 $-\ln(P(v))$ 具体的内容，：

$$-\ln(P(v)) = -\ln\left(\frac{1}{Z} \sum_h e^{-E(v,h)}\right) \quad (10)$$

$$= \ln Z - \ln\left(\sum_h e^{-E(v,h)}\right) \quad (11)$$

$$= \ln\left(\sum_{v,h} e^{-E(v,h)}\right) - \ln\left(\sum_h e^{-E(v,h)}\right) \quad (12)$$

我们以 $a_i$ 的梯度计算为例：

$$\frac{\partial(-\ln(P(v)))}{\partial a_i} = \frac{1}{\partial a_i} \partial \ln\left(\sum_{v,h} e^{-E(v,h)}\right) - \frac{1}{\partial a_i} \partial \ln\left(\sum_h e^{-E(v,h)}\right) \quad (13)$$

$$= -\frac{1}{\sum_{v,h} e^{-E(v,h)}} \sum_{v,h} e^{-E(v,h)} \frac{\partial E(v, h)}{\partial a_i} + \frac{1}{\sum_h e^{-E(v,h)}} \sum_h e^{-E(v,h)} \frac{\partial E(v, h)}{\partial a_i} \quad (14)$$

$$= \sum_h P(h|v) \frac{\partial E(v, h)}{\partial a_i} - \sum_{v,h} P(h, v) \frac{\partial E(v, h)}{\partial a_i} \quad (15)$$

$$= -\sum_h P(h|v) v_i + \sum_{v,h} P(h, v) v_i \quad (16)$$

$$= -\sum_h P(h|v) v_i + \sum_v P(v) \sum_h P(h|v) v_i \quad (17)$$

$$= \sum_v P(v) v_i - v_i \quad (18)$$

其中用到了：

$$\sum_h P(h|v) = 1$$

同样的方法，可以得到 $W, b$ 的梯度。这里就不推导了，直接给出结果：

$$\frac{\partial(-\ln(P(v)))}{\partial b_i} = \sum_v P(v)P(h_i = 1|v) - P(h_i = 1|v)$$

$$\frac{\partial(-\ln(P(v)))}{\partial W_{ij}} = \sum_v P(v)P(h_i = 1|v)v_j - P(h_i = 1|v)v_j$$

虽然梯度下降法可以从理论上解决RBM的优化，但是在实际应用中，由于概率分布的计算量大，因为概率分布有 $2^{n_v+n_h}$ 种情况，所以往往不直接按上面的梯度公式去求所有样本的梯度和，而是用基于MCMC的方法来模拟计算求解每个样本的梯度损失再求梯度和，常用的方法是基于Gibbs采样的对比散度方法来求解，对于对比散度方法，由于需要MCMC的知识，这里就不展开了。对对比散度方法感兴趣的可以看参考文献中2的《A Practical Guide to Training Restricted Boltzmann Machines》，对于MCMC，后面我专门开篇来讲。

## 4. RBM在实际中应用方法

大家也许会疑惑，这么一个模型在实际中如何能够应用呢？比如在推荐系统中是如何应用的呢？这里概述下推荐系统中使用的常用思路。

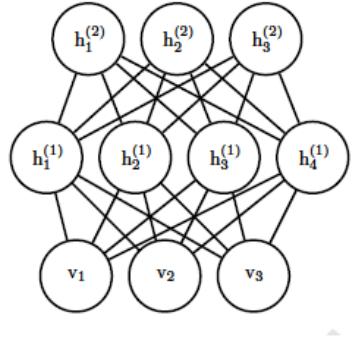
RBM可以看做是一个编码解码的过程，从可见层到隐藏层就是编码，而反过来从隐藏层到可见层就是解码。在推荐系统中，我们可以把每个用户对各个物品的评分做为可见层神经元的输入，然后有多少个用户就有了多少个训练样本。由于用户不是对所有的物品都有评分，所以任意样本有些可见层神经元没有值。但是这不影响我们的模型训练。在训练模型时，对于每个样本，我们仅仅用有用户数值的可见层神经元来训练模型。

对于可见层输入的训练样本和随机初始化的 $W, a$ ，我们可以用上面的sigmoid激活函数得到隐藏层的神经元的0,1值，这就是编码。然后反过来从隐藏层的神经元值和 $W, b$ 可以得到可见层输出，这就是解码。对于每个训练样本，我们期望编码解码后的可见层输出和我们的之前可见层输入的差距尽量的小，即上面的对数似然损失函数尽可能小。按照这个损失函数，我们通过迭代优化得到 $W, a, b$ ，然后对于某个用于那些没有评分的物品，我们用解码的过程可以得到一个预测评分，取最高的若干评分对应物品即可做用户物品推荐了。

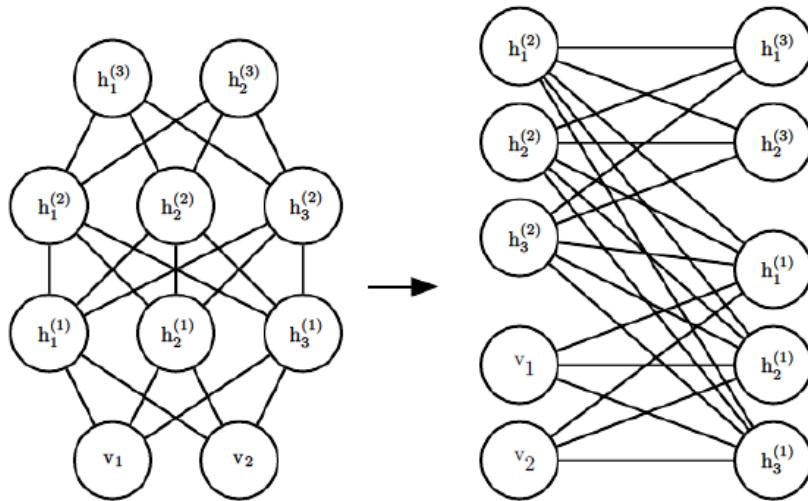
如果大家对RBM在推荐系统的应用具体内容感兴趣，可以阅读参考文献3中的《Restricted Boltzmann Machines for Collaborative Filtering》

## 5. RBM推广到DBM

RBM很容易推广到深层的RBM，即我们的DBM。推广的方法就是加入更多的隐藏层，比如一个三层的DBM如下：



当然隐藏层的层数可以是任意的，随着层数越来越复杂，那模型怎么表示呢？其实DBM也可以看做是一个RBM，比如下图的一个4层DBM，稍微加以变换就可以看做是一个DBM。



将可见层和偶数隐藏层放在一边，将奇数隐藏层放在另一边，我们就得到了RBM，和RBM的细微区别只是现在的RBM并不是全连接的，其实也可以看做部分权重为0的全连接RBM。RBM的算法思想可以在DBM上使用。只是此时我们的模型参数更加的多，而且迭代求解参数也更加复杂了。

## 6. RBM小结

RBM所在的玻尔兹曼机流派是深度学习中三大流派之一，也是目前比较热门的创新区域之一，目前在实际应用中的比较成功的是推荐系统。以后应该会有更多类型的玻尔兹曼机及应用开发出来，让我们拭目以待吧！

( 欢迎转载，转载请注明出处。欢迎沟通交流： pinard.liu@ericsson.com )

## 参考资料：

- 1) [Deep Learning](#), book by Ian Goodfellow, Yoshua Bengio, and Aaron Courville
- 2) [A Practical Guide to Training Restricted Boltzmann Machines](#), by G. Hinton
- 3) [Restricted Boltzmann Machines for Collaborative Filtering](#), by G. Hinton

---