tensorfly
最专注的机器学习中文社区

第二章：”逆传播”算法的工作机制

Posted on 2015年12月23日 by hulk · 1 Comment

《神经网络和深度学习》第二章：”逆传播”算法的工作机制

在上一章我们看到了神经网络如何使用梯度下降算法来进行自我权重和偏差的学习。然而在我们的解释中却存在一个问题：我们并没有讨论如何计算代价函数的梯度。这的确是一个问题！本章我将阐述一个快速计算梯度的算法，即*后向传播*。

后向传播算法产生于1970年，但它的重要性一直到David Rumelhart，Geoffrey Hinton和Ronald Williams于1986年合著的论文才被重视。该论文介绍了几种神经网络，它们使用后向传播比早期的学习算法速度更快，而且还能处理之前不能解决的神经网络问题。当今，后向传播算法已经是学习神经网络的基础。

本章将比书中其他部分采用更多的数学推导。如果你对数学很抓狂，建议你跳过该章节，只需要将后向传播当作一个黑盒子，而忽略它的细节。为什么需要花时间来学习这些细节呢？

原因当然是为了更好的理解。后向传播的核心是网络中代价函数C相对于任意权重w（或者偏差b）的偏导表达式$\partial C/\partial w$。这个表达式告诉我们当改变权重和偏差时代价函数值改变速度有多快。尽管这个表达式有些复杂，它仍然有很美的一面，其中每个元素都有自然的，直觉的解释。因此后向传播不只是一个快速学习算法，它实际上能告诉我们更多细节来进行权重和偏差的改变，以影响整个网络的行为。学习它的细节是值得的。

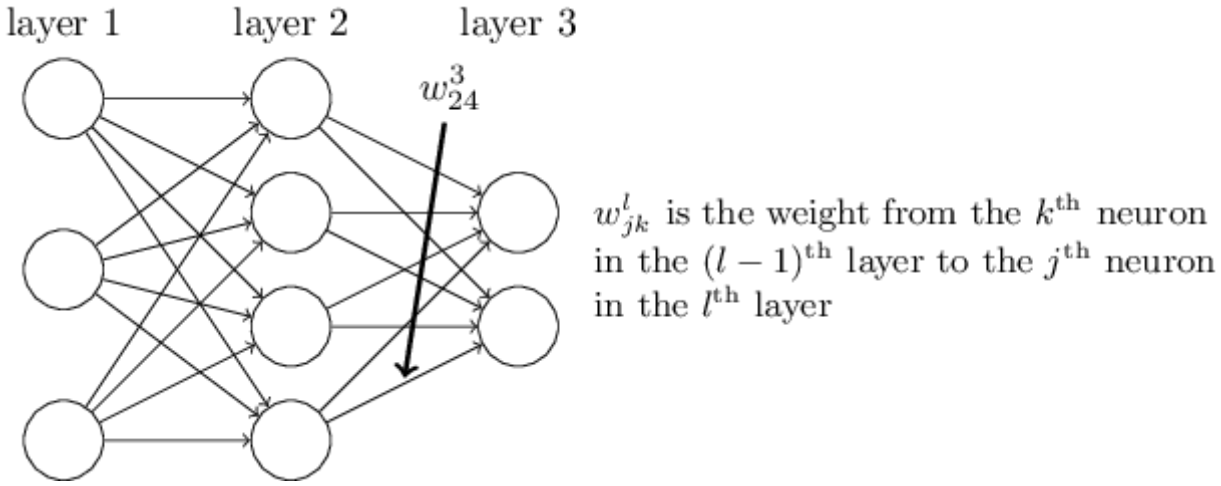
就像刚才说的，如果你想跳过本章节，或者直接到下一章，也行。我将使得本书的其他部分仍然可以读懂，即便你将后向传播当作一个黑盒子。因此书里后面部分将直接采用本章的结论。但是你仍然能够明白主要的结论，即便你没有这些理由。

预热：一种快速基于矩阵计算神经网络输出的途径

在讨论后向传播之前，让我们预热一下计算神经网络输出的快速矩阵

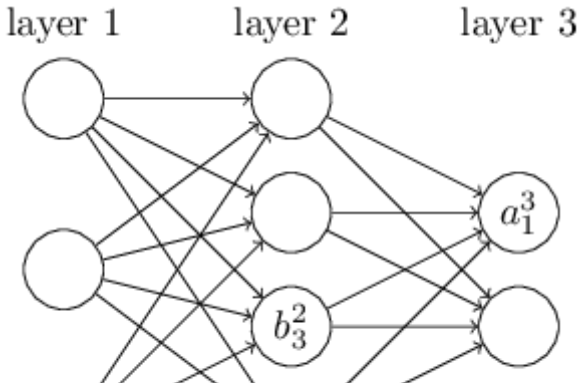
算法。我们实际上已经简要的看到过这个算法 上一章的最后部分，但是我只是快速的描述了一下，因此现在值得再看看它的细节。这也是一个很好的方式，在熟悉的上下文中舒适的接受后向传播的术语。

让我们以一个符号开始，它代表网络中任意方式的权重信息。我们将使用w^l_{jk}来表示从网络第$(l-1)^{\text{th}}$层中第k^{th}个神经元指向l^{th}层中第j^{th}个神经元的连接权重。因此举个例子，下图中的权重就表示从第二层中第四个神经元指向第三层中第二个神经元的权重：



这个符号起始比较麻烦，的确需要一些努力才能掌握。但是通过努力你会发现它将变得简单和自然。符号中的一个不容易接受的地方就是j和k的位置关系。你可能认为用j来表示输入神经元，k表示输出神经元，而不是实际定义中反过来的方式。我将在下面解释这样做法的原因。

我将使用相似的符号来表示网络中的偏差和激活。明确地，我们使用b^l_j来表示第l^{th}层中第j^{th}神经元的偏差，用a^l_j来表示第l^{th}层中第j^{th}神经元的激活。下面的图将展示这些符号：



有了这些符号，第l^{th}层中第j^{th}神经元的激活a^l_j就与第$(l-1)^{\text{th}}$层中所有激活相关（对比上一章中等式4和其环绕的注释）。

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \tag{23}$$

这里的和遍历了第$(l-1)^{\text{th}}$层所有神经元。为了用矩阵重写这个表达式，我们定义每一层l的*权重矩阵*w^l。权重矩阵w^l中每一个元素表示连接到l^{th}层神经元的每一个权重，也就是，第j^{th}行和第k^{th}列的元素是w^l_{jk}。相似的，我们为每一层定义*偏差向量*b^l。你可能会猜这是怎么定义的——偏差向量的元素就是b^l_j，每个元素就表示第l^{th}层每个神经元的偏差。最后，我们定义激活向量a^l，它的每个元素表示a^l_j。

我们用矩阵形式重写(23) 是为了向量化这个函数σ。我们在上一章简要的遇到过向量化，回顾一下，思想是我们想将函数σ应用到向量v中的每个元素。我们使用显示的符号$\sigma(v)$来定义这种元素级范围的函数应用。也就是，函数$\sigma(v)$的组件为$\sigma(v)_j = \sigma(v_j)$。举例，如果我们有一个函数$f(x) = x^2$，那么向量化这个函数$f$结果为

$$f\left(\begin{bmatrix}2\\3\end{bmatrix}\right) = \begin{bmatrix}f(2)\\f(3)\end{bmatrix} = \begin{bmatrix}4\\9\end{bmatrix}, \tag{24}$$

也就是，函数f向量化就是对向量中每个元素进行平方。

记住这些术语，表达式(23) 能够用漂亮而简洁的向量格式进行重写

$$a^l = \sigma(w^l a^{l-1} + b^l). \tag{25}$$

这个表达式能给我们更多的启发，某一层的激活与上一层的激活是有什么关系：我们只是将权重矩阵应用到激活上，然后再加上一个偏差向量，最后应用σ函数 *。这个全局视图非常简单和简洁（使用了很少的下标），相对于一个神经元到一个神经元的方式。也可以将其想象成一种避免下标混乱，而且还能保持精确的方法。这个表达式在实

*顺便说
w^l_{jk}符号
指示输
中的权
很小的

际中非常有用，因为许多矩阵库都能提供快速的矩阵乘法，向量加法和向量化。实际上，上一章的 代码已经隐式的使用这个表达式来计算网络的行为。

当使用表达式(25) 来计算a^l，我们间接的计算$z^l \equiv w^l a^{l-1} + b^l$。这个值非常有用，我们将$z^l$命名为：网络$l$层的*加权输入*。我们将在本章大量的使用加权输入$z^l$。表达式(25) 有时候也写成加权输入的形式：$a^l = \sigma(z^l)$。也需要注意$z^l$具有各元素组件$z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$，也就是，$z^l_j$是对网络第$l$层第$j$个神经元激活的加权输入。

关于代价函数的两个假设

后向传播的目标是计算代价函数C对于网络中任意权重w或偏差b的偏导数$\partial C/\partial w$ and $\partial C/\partial b$。为了让后向传播有效，我们需要对代价函数进行两个主要设定。在开始这些设定前，最好头脑里面能有一个代价函数的例子。我们将使用上一章的二次代价函数（参见表达式(6) ）。在最后部分的符号中，二次代价函数有如下形式

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2, \tag{26}$$

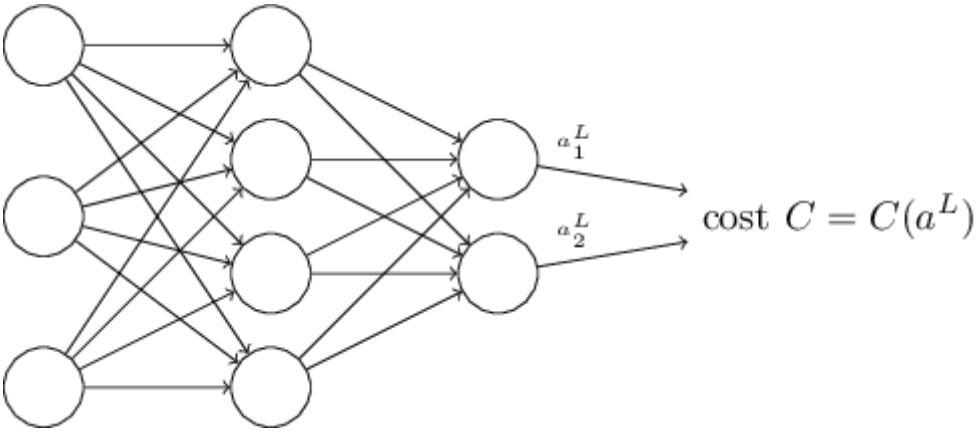
其中n表示训练样本的总数；求和是覆盖每一个独立的训练样本，x；$y = y(x)$是对应的期望输出；L表示网络层数；$a^L = a^L(x)$是当输入为x时网络的激活输出向量。

好了，我们需要关于代价函数C什么样的假设来使得后向传播能够应用起来？第一个假设是代价函数能够被写成基于每一个独立的训练样本x求代价函数C_x的平均值$C = \frac{1}{n} \sum_x C_x$。对于二次代价函数，每一个样本的代价为$C_x = \frac{1}{2}\|y - a^L\|^2$。这个假设对于本书中遇到的所有代价函数管用。

需要这个假设的原因是因为后向传播实际上让我们能够基于每一个样本计算偏导数$\partial C_x/\partial w$和$\partial C_x/\partial b$。我们然后在整个选练样本基础上经过

平均而求出$\partial C/\partial w$和$\partial C/\partial b$。实际上，记住这个假设，我们能够想象训练样本x是固定的，可以去掉x的下标，将代价函数C_x写成C。最后再将x恢复回来，但是现在最好隐式的去掉这个下标，以便于符号说明。

第二个关于代价函数的假设是可以把它当作神经网路激活输出的一个函数：



比如，二次型代价函数就满足这个假定，因为对于每一个样本x，代价函数可以写成

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2, \tag{27}$$

，因此代价函数的确是关于激活输出的函数。当然，代价函数也依赖于期望输出y，那么你可能会问为什么不把代价函数当作期望输出y的函数。记住，因为输入样本x是固定的，因此期望输出y也是固定的参数了。特别的，我们不能通过修改权重和偏差来改变y的值，也就是它不是网络需要学习的东西。因此C只是唯一与激活输出a^L相关，而y只是帮助定义该函数而已。

Hadamard乘积$s \odot t$

后向传播算法是基于通用的线性代数运算——就像向量加法，矩阵乘向量等等。但是有一个操作平常很少用到。特别的，假设s和t是相同维数的两个向量，那么我们使用$s \odot t$来表示两个向量*元素级*的乘法。因此$s \odot t$的元素$(s \odot t)_j = s_j t_j$。举个例子，

因此$s \odot t$的元素是$(s \odot t)_j = s_j t_j$。举个例子，

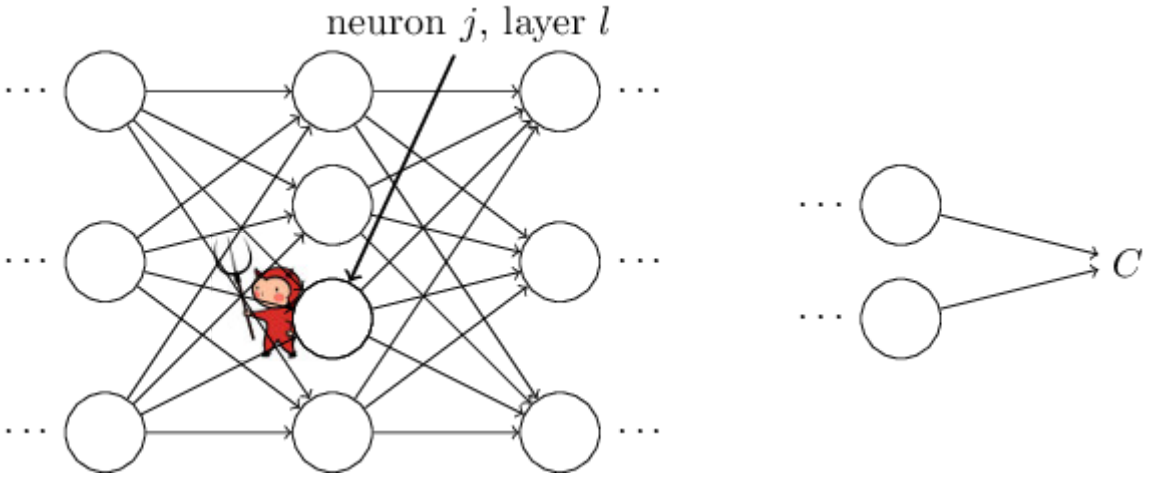
$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}. \tag{28}$$

这种元素级的乘法有时叫做 *Hadamard乘积*或者*Schur乘积*。我们将把它叫做Hadamard乘积。好的矩阵库一般都能提供Hadamard乘积的快速实施，因此在实施后向传播时候就非常方便。

后向传播背后的基础等式

后向传播能够知道如何更改网络中的权重和偏差来改变代价函数值。最终这意味着它能够计算偏导数$\partial C/\partial w^l_{jk}$和$\partial C/\partial b^l_j$。为了计算这些偏导数，我们首先引入一个中间变量$\delta^l_j$，我们把它叫做网络中第$l^{\text{th}}$层第$j^{\text{th}}$个神经元的 *误差*。后向传播能够计算出误差$\delta^l_j$，然后再将其对应回$\partial C/\partial w^l_{jk}$和$\partial C/\partial b^l_j$。

为了理解如何定义误差，想象一下在神经网络中有一个恶魔：



这个恶魔坐在网络层l的第j^{th}神经元处。当有输入到达神经元时，这个恶魔扰乱神经元的操作。它对神经元的加权输入Δz^l_j做一些小的改变，以致于不再输出$\sigma(z^l_j)$，而是输出$\sigma(z^l_j + \Delta z^l_j)$。这个改变将传播到网络中后面各个层中，最后引起整个代价函数值改变$\frac{\partial C}{\partial z^l_j}\Delta z^l_j$。

现在，这个恶魔是一个好的恶魔，它能够帮助你改进代价函数值，例如，他们能够找到一个Δz^l_j，使得代价值变得更小。假设$\frac{\partial C}{\partial z^l_j}$是一个很

大的数（要么是正，要么是负），那么恶魔能够通过选择Δz^l_j与$\frac{\partial C}{\partial z^l_j}$符号相反来减少代价值一点点。相反，如果$\frac{\partial C}{\partial z^l_j}$接近零，那么恶魔就不能通过扰动加权输入$z^l_j$来改进代价函数值。因此恶魔便会说，这个网络已经接近最优*因此它的启发意义就是$\frac{\partial C}{\partial z^l_j}$为网络中的误差测量。

*当然这魔被约

受这个故事所启发，我们定义网络层l的第j个神经元的误差为δ^l_j

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}. \tag{29}$$

按照我们通常的惯例，我们使用δ^l来表示网络层l的误差向量。后向传播能够让我们对每一层计算δ^l，然后再将这些误差对应到所感兴趣的参数$\partial C/\partial w^l_{jk}$和$\partial C/\partial b^l_j$上。

你可能在想为什么恶魔要改变加权输入z^l_j。的确更自然的方法是假设恶魔改变的激活输出a^l_j，这个结果将是使用$\frac{\partial C}{\partial a^l_j}$来作为误差的测量值。实际上，如果你这么做，同样也会得到下面的讨论结果。但是这样会使得后向传播表述上更加复杂。因此我们使用$\delta^l_j = \frac{\partial C}{\partial z^l_j}$来做为误差的测量*.

*在分类示分类值96.0区别。出误差

战斗计划：后向传播基于四个基础等式。这些等式使我们能够计算误差δ^l和代价函数的梯度。我将在下面阐述这四个等式。警告，你不能期望同时接受这四个等式。如果想同时理解清楚，可能会让你失望而归。实际上，后向传播等式如此之难，以致于需要你不断地深入这些等式，耗费大量的思考时间和耐心。好消息是通过不断的反复理解，你的时间和耐心将会需要得越来越少。因此本节讨论只是一个开始，帮助你不断理解这些等式。

这些是章节中不断深入这些等式的预览：我将 给出这些等式简要的证明，浙江帮助解释为什么他们是正确的；用算法伪代码重申这些不等式，并看到这些伪代码用Python代码怎样被实施的；在最后一节，我们将给出后向传播等式的学术性描述，并且解释其是如何被某些人发现了的。按照这种方式，我们将不断的回顾这四个基础等式，这样

就能更舒服的不断深入理解它们，也会感觉更加美丽和自然。

输出层的误差等式δ^L：δ^L的每一个组件即

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j). \tag{BP1}$$

这是一个非常自然的等式。右边的第一项$\partial C/\partial a^L_j$测量代价函数对第$j^{\text{th}}$输出神经元的激活变化有多快。例如，$C$不太依赖于输出神经元$j$，那么$\delta^L_j$就会非常小，这正是我们所期望的。右边第二部分$\sigma'(z^L_j)$测量激活函数$\sigma$对$z^L_j$变化有多快。

注意在(BP1) 中的每一个部分都是容易计算的。特别的，我们在计算网络行为时会得到z^L_j的值，而只需要增加一点点计算就能得到$\sigma'(z^L_j)$的值。而$\partial C/\partial a^L_j$的精确格式显然与具体的代价函数有关。但是，只要给出了代价函数，计算$\partial C/\partial a^L_j$就小菜一碟了。例如，如果使用二次型代价函数$C = \frac{1}{2}\sum_j (y_j - a_j)^2$，那么$\partial C/\partial a^L_j = (a_j - y_j)$，这显然也是非常容易计算的。

等式(BP1) 是δ^L的组件形式。它是一个很好的表达式，但是还不是后向传播所需要的矩阵形式。然而很容易将其重写为矩阵形式

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

这里，$\nabla_a C$被定义为向量，它的每一个组件都是偏导数$\partial C/\partial a^L_j$。你可以将$\nabla_a C$做为是$C$对激活输出的变化率。很容易看出等式(BP1a) 和(BP1) 是相等的，正因为如此，我们使用(BP1) 来表示两种等式。举个例子，当使用二次型代价函数，我们有$\nabla_a C = (a^L - y)$，因此(BP1) 的矩阵形式就变为

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{30}$$

能看出表达式中每一个部分都是向量形式，因此很容易使用像Numpy这类矩阵库来进行计算。

误差δ^l用下一层的误差表示，δ^{l+1}：特别的

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{BP2}$$

其中$(w^{l+1})^T$是$(l+1)^{\text{th}}$层权重矩阵w^{l+1}的转置。这个表达式有一些复杂，但是其中每一个部分都有很好的内部展现。假如我们知道第$l+1^{\text{th}}$层的误差δ^{l+1}。那么我们乘以权重矩阵的转置$(w^{l+1})^T$，我们就能递归的*向后*获得网络中所有误差，这就给我们了一种测量l^{th}层误差的方式，然后我们运用Hadamard乘积$\odot\sigma'(z^l)$。这就将通过l层的激活函数获取到后面的所有误差，即l层加权输入的误差δ^l。

通过合并(BP2) 和(BP1) ，我们能够计算出网络中各层所有误差δ^l。我们以(BP1) 做为开始来计算δ^L，然后使用等式(BP2) 来计算δ^{L-1}，然后再使用等式(BP2) 来计算δ^{L-2}等等，一直朝网络中后面的各层进行计算。

代价函数相对网络中任意偏差变化率的等式：

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j. \tag{BP3}$$

也就是，误差δ^l_j和变化率$\partial C/\partial b^l_j$*精确相同*。这是一个好消息，因为(BP1) 和(BP2) 已经告诉我们如何计算δ^l_j。我们能够简短的重写它为(BP3)

$$\frac{\partial C}{\partial b} = \delta, \tag{31}$$

这表明误差δ和神经元的偏差b是一个意思。

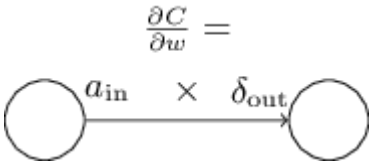
代价函数相对网络中任意权重变化率的等式：

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j. \tag{BP4}$$

这告诉我们如何根据变量δ^l和a^{l-1}来计算偏导数$\partial C/\partial w^l_{jk}$，而这两个量已经知道如何计算了。这个等式还能重写为更少下标的符号形式

$$\frac{\partial C}{\partial w} = a_{\text{in}}\delta_{\text{out}}, \tag{32}$$

其中a_{in}是对应于权重w的输入神经元激活值，δ_{out}是对应于权重w的神经元输出误差。放大网络只看到权重w和被其连接的两个神经元，我们能描绘它们为：



等式(32) 的一个很好的结果就是当a_{in}非常小的时候，$a_{\text{in}} \approx 0$，梯度$\partial C/\partial w$也将趋于很小。这种情况下，权重*缓慢学习*表示在梯度下降过程中权重更改很少。换句话说，(BP4) 的结果就是小的激活神经元的权重会比较缓慢的学习。

还有一些其它的领悟能够从(BP1) -(BP4) 中获取。让我们开始看这个输出层。考虑在(BP1) 中的$\sigma'(z^L_j)$。回顾一下上一章提到的sigmod函数图像，当$\sigma(z^L_j)$接近于0或1的时候，σ函数变得扁平。这时候我们将得到$\sigma'(z^L_j) \approx 0$。因此最后一层的权重在低激活（≈ 0）或者高激活（≈ 1）时将变得学习非常缓慢。这时候通常说输出神经元已经*饱和*，权重已经停止学习（或者学习缓慢）。同样的结论也适合于输出层的偏差。

我们也能对其它前面的网络层得到相似的领悟。特别地，注意(BP2) 中的$\sigma'(z^l)$。这表示如果神经元接近饱和，δ^l_j也将变得更小。因此也意味着对于饱和神经元的任何权重输入都将学习缓慢*。

*如果w
量，这
的趋势。

总结一下，我们知道权重将在低激活或者神经元饱和时学习缓慢起来，即要么高激活，要么地激活。

这些观察都没令人太惊讶。还有，它们有助于做为神经网络学习来提高我们的心智模型。更多的是，我们能够推广将这类推理。这四个基础等式适合于任意的激活函数，不光是标准的sigmoid函数（这是因为，就像待会将看到的，这四个等式的证明没有使用σ的任何特别属

性）。并且我们能够使用这些等式来*设计*激活函数，使得它们具有特殊的学习特性。举个例来帮助你思考，假设我们选择一个（非sigmoid）激活函数σ，使得σ'永远都是正值，也不会接近于零。这将防止像常规sigmoid神经元饱和时学习能力下降的局面。本书后面我们将看到一些对激活函数进行修改的案例。记住这四个等式(BP1) - (BP4) 能有助于解释为什么要进行这些修改，并且这些改变将影响到什么。

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

问题

- **后向传播等式另一种表述：**我们已经阐述了使用Hadamard乘积的后向传播的基础等式（(BP1) 和(BP2) ）。如果你不习惯Hadamard乘积，这种描述会让人不安。有另一种变换途径，基于传统的矩阵乘积，这会使得一些读者更有启发作用。
（1）(BP1) 可以重写为

$$\delta^L = \Sigma'(z^L) \nabla_a C, \tag{33}$$

其中$\Sigma'(z^L)$是一个对角线元素为$\sigma'(z^L_j)$的矩阵，并且它的非对角线元素都是零。注意，这个矩阵将与$\nabla_a C$进行传统的矩阵乘法。
（2）(BP2) 可以重写成

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \delta^{l+1}. \tag{34}$$

（3）可以将（1）和（2）进行合并得到

$$\delta^l = \Sigma'(z^l)(w^{l+1})^T \ldots \Sigma'(z^{L-1})(w^L)^T \Sigma'(z^L)\nabla_a C \tag{35}$$

对于那些熟悉矩阵乘法的读者，这些等式将比(BP1) 和(BP2) 更容易理解。我们关注在(BP1) 和(BP2) 的原因是这种方式将更加快速的被数字化实施。

四个基础等式的证明（可选）

我们现在来证明四个基础等式(BP1) -(BP4) 。这四个等式都是基于多元微积分的链式规则。如果你熟悉这些链式规则，我强烈建议你在阅读前自行进行推导。

让我们开始等式(BP1) ，它给出了输出层的误差表达式δ^L。为了证明这个等式，再次拿出定义

$$\delta^L_j = \frac{\partial C}{\partial z^L_j}. \tag{36}$$

应用链式规则，我们能够用对激活输出的偏导数来重新表达这个误差偏导数，

$$\delta^L_j = \sum_k \frac{\partial C}{\partial a^L_k}\frac{\partial a^L_k}{\partial z^L_j}, \tag{37}$$

其中求和是对输出层所有的神经元k进行的。当然，第k^{th}神经元的输出激活a^L_k只依赖于第j^{th}个神经元的加权输入z^L_j，这里$k = j$。因此当$k \neq j$时，$\partial a^L_k/\partial z^L_j$就为零。结果我们可以简化之前的等式为

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j}. \tag{38}$$

重新拿出定义$a^L_j = \sigma(z^L_j)$，这个等式右边能够写作$\sigma'(z^L_j)$，并且该等式变为

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j), \tag{39}$$

这就是(BP1) 的元素级表示。

接下来，我们将证明(BP2) ，它给出误差等式δ^l基于下一层误差δ^{l+1}的表述。为了证明，我们用$\delta^{l+1}_k = \partial C/\partial z^{l+1}_k$来重写$\delta^l_j = \partial C/\partial z^l_j$。我们能够使用链式规则来做，

$$\begin{aligned}\delta^l_j &= \frac{\partial C}{\partial z^l_j} && (40)\\ &= \sum_k \frac{\partial C}{\partial z^{l+1}_k}\frac{\partial z^{l+1}_k}{\partial z^l_j} && (41)\\ &= \sum_k \frac{\partial z^{l+1}_k}{\partial z^l_j}\delta^{l+1}_k, && (42)\end{aligned}$$

其中最后一行我们交换右边两项，并且用δ^{l+1}_k来替换。为了计算最后一行的右边第一项，注意

$$z^{l+1}_k = \sum_j w^{l+1}_{kj} a^l_j + b^{l+1}_k = \sum_j w^{l+1}_{kj}\sigma(z^l_j) + b^{l+1}_k. \tag{43}$$

进行偏导，我们获得

$$\frac{\partial z^{l+1}_k}{\partial z^l_j} = w^{l+1}_{kj}\sigma'(z^l_j). \tag{44}$$

代回到(42) 我们能获得

$$\delta^l_j = \sum_k w^{l+1}_{kj}\delta^{l+1}_k\sigma'(z^l_j). \tag{45}$$

这就是(BP2) 的组件级格式。

最后两个我们想证明的等式(BP3) 和(BP4) 。它们也可以按照链式规则相似的方式来进行证明。我将把推导留给你来练习。

练习

- 证明等式(BP3) 和(BP4) 。

这已经完成了后向传播四个基础等式的证明。这些证明看起来很复杂，但它们的确就是仔细的应用链式规则得到的结果。更简练的，我们可以将后向传播当做同时应用多元微积分链式规则来计算代价函数梯度的方法。这就是所有后向传播的细节。

后向传播算法

后向传播等式给我们提供了一种计算代价函数梯度的方法。让我们用算法显示的写出它们：

1. **输入**x：为输入层设置对应的激活a^1。

2. **向前反馈**：对于每一层$l = 2, 3, \ldots, L$计算$z^l = w^l a^{l-1} + b^l$和$a^l = \sigma(z^l)$。

3. **输出层误差**δ^L：计算向量$\delta^L = \nabla_a C \odot \sigma'(z^L)$。

4. **后向传播误差**：对每一层$l = L-1, L-2, \ldots, 2$计算$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$。

5. **输出**：代价函数的梯度为$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$和$\frac{\partial C}{\partial b^l_j} = \delta^l_j$。

检查这个算法，你能看到为什么它被叫作*后向*传播。我们从最末一层开始向后计算各层误差δ^l。看起来将网络向后传播很奇怪。但是如果你再想一下后向传播的证明，向后移动就是因为代价是网络输出的函数。为了理解代价是如何跟随早期的权重和偏差进行的改变，我们需要不断的应用链式规则，向后来获取有用的表达式。

练习

- **具有一个神经元的后向传播** 假设我们更改前向网络中的一个神

经元以便这个神经元的输出是$f(\sum_j w_j x_j + b)$，其中f是某个非sigmoid的函数。那么我们应该如何更改这个后向传播算法？

- **线形神经元后向传播** 假设我们将非线性的σ函数用$\sigma(z) = z$来替换，重写该后向传播算法。

就像我上面描述那样，后向传播算法计算了某一个样本的代价函数的梯度$C = C_x$。实际上，通用的方式是将后向传播算法与随机梯度下降算法合并，我们便能计算许多训练样本的梯度。特别的，给出一小批训练样本m，下面算法将基于小批训练样本进行梯度下降学习步骤：

1. **输入训练样本集合**

2. **对于每一个训练样本x：** 设置对应的输入激活$a^{x,1}$，执行以下步骤：

   - **向前：**对于每一层$l = 2, 3, \ldots, L$计算$z^{x,l} = w^l a^{x,l-1} + b^l$和$a^{x,l} = \sigma(z^{x,l})$。

   - **输出层误差$\delta^{x,L}$：**计算向量$\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$。

   - **后向传播误差：**对于每一层$l = L-1, L-2, \ldots, 2$计算$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$。

3. **梯度下降：**对于每一层$l = L, L-1, \ldots, 2$，按照规则$w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$更新权重，按照规则$b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$更新偏差。

当然，为了实施随机梯度下降，你也许要一个外部循环来产生小批量的训练样本，还需要一个外部循环来实施更多的训练代。我们将为了简化暂且忽略掉这些。

后向传播的实施代码

抽象上理解了后向传播算法，我们能够就能够理解上一章实施后向传播的代码了。重新拿出 那一章，代码位于类Network的update_mini_batch和backprop方法中。这些方法代码就是上面描述的算法的直接翻译。特别的，这个update_mini_batch方法通过计算当前选练小批量训练样本mini_batch的梯度来更新了类Network的权重和偏移。

```
class Network(object):
...
  def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
      delta_nabla_b, delta_nabla_w = self.backprop(x, y)
      nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
      nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

许多工作是靠delta_nabla_b, delta_nabla_w = self.backprop(x, y)来处理的，它们使用backprop来计算偏导数$\partial C_x/\partial b^l_j$和$\partial C_x/\partial w^l_{jk}$。这个函数backprop使用上一节提到的算法。有一点小的变化——我们使用细微的不同方式来指示层数。这个改变能够使用Python优势特性，即使用负索引位置来从列表的最末往后计算。例如：l[-3]是列表l的倒数第三个元素。函数backprop的代码在下面，还有一些帮助函数，它们用来计算σ函数，偏导σ'以及代价函数导数。有了这些，你就能够独立的理解这些代码。如果有什么困扰到你，你可以查看原始的代码描述（完整的程序）来帮助你理解。

```
class Network(object):
...
  def backprop(self, x, y):
    """Return a tuple "(nabla_b, nabla_w)" representing the
    gradient for the cost function C_x.  "nabla_b" and
    "nabla_w" are layer-by-layer lists of numpy arrays, similar
    to "self.biases" and "self.weights"."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
```

```
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    ...

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

问题

- **对小批量样本完全基于矩阵的后向传播途径** 我们实施随机梯度下降采用小批量样本不断循环的方式。可以更改后向传播算法，使得同时对小批量样本中的每一个样本计算梯度。思路就是我们可以替换单一输入向量x，而采用矩阵$X = [x_1 x_2 \ldots x_m]$，它的列

可以替换单一输入向量x，而采用矩阵$X = [x_1 x_2 \ldots x_m]$，它的列就是小批量数量的维度。我们向前将其乘以权重矩阵，再加上偏差矩阵，最后实施sigmoid函数。我们也后向采用相似的方式。显示的写出这样的后向传播算法伪代码。更改network.py以便它使用完全基于矩阵的方式。这种方式的优势是它利用了当今线性代数库的优势。结果是这样能够比起小批量样本内不断循环方式运行更快。（比如在我的笔记本中，运行MNIST分类的速度将比上一节中的循环方式缩短二分之一。）实际上，所有后向传播库都使用完全矩阵方式或者它的某种变形。

什么意思上说明后向传播算法是快速的？

什么意思上说明后向传播算法是快速的？为了回答这个问题，让我们考虑另一种计算梯度的途径。假设这是神经网络研究早期，可能是1950年代或者1960年代，而且你是世界上第一个想到用梯度下降算法学习的人！为了使其能够有效，你需要一种计算代价函数梯度的方式。你可能想起了微积分知识，并且决定看看是否能够用链式规则来计算梯度。但是在尝试之后，这个代数看起来非常复杂，这会让人失望。因此你尝试寻找另外一种途径。你决定将代价函数只当作权重的函数$C = C(w)$（后面我们将同样针对偏差）。你将权重进行编号$w_1, w_2, \ldots$，并且想对某个权重w_j计算偏导数$\partial C/\partial w_j$。一种明显的方式是使用近似值

$$\frac{\partial C}{\partial w_j} \approx \frac{C(w + \epsilon e_j) - C(w)}{\epsilon}, \tag{46}$$

其中$\epsilon > 0$是一个很小的正数，且e_j是方向j^{th}上一个单元向量。换句话说，我们能够估算$\partial C/\partial w_j$，通过计算基于微小变化w_j的代价C，然后应用(46)。这同样的思路也可以用于计算对偏差的偏导数$\partial C/\partial b$。

这个方式看起来非常有希望。概念上很简单，并且很容易实施，甚至只需要几行代码。当然，这看起来比使用链式规则计算梯度更加有希望！

不幸的是，虽然这个途径看起来有希望，当你实施代码，它将运行得非常慢。为了理解其原因，我们假设网络中有一百万个权重。那么对每一个权重w_j，我们需要计算$C(w+\epsilon e_j)$来计算$\partial C/\partial w_j$。这意味着我们需要计算代价函数一百万次来算出梯度，需要网络向前传播一百万次（对每一个样本）。我们还需要计算$C(w)$，因此这将是在网络中百万零一次向前传播。

后向传播的聪明之处在于它允许我们同时计算*所有*的偏导数$\partial C/\partial w_j$，而且只需要网络向前计算一次，向后传播一次。简单说，向后传播和向前的计算代价差不多相同*。因此后向传播的总计算代价是两倍于网络向前传播。比较一下基于(46) 的百万零一次的向前传播！因此即使后向传播表面上看起来比基于(46) 更复杂，但是它的确非常，非常的快。

*这将是
谨慎的
要代价
置的乘
价。

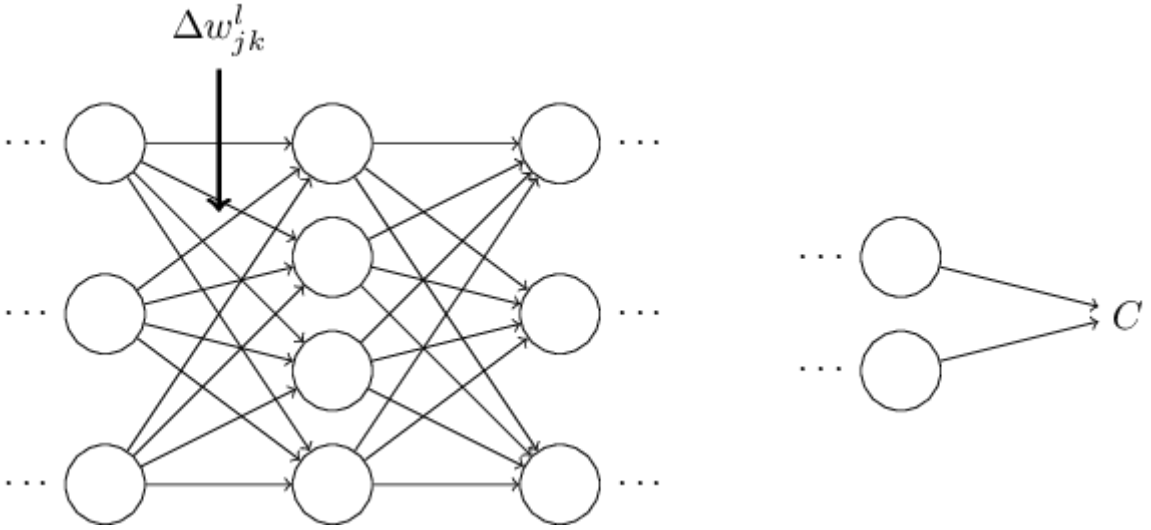
这中加速第一次在1986年被重视，它很大程度上扩展了神经网络能够解决问题的范围。也就使得大量的人开始使用神经网络。当然，后项传播也不是一个万能药。即使在1980年代后期，人们逐渐看到了它的限制，尤其是当试图训练一些深度的神经网络，例如网络中有许多隐含层。本书的后面，我们将看到现代的计算机和某些聪明的思路，如何使得利用后向传播训练如此深度的神经网络。

后向传播：大的构想

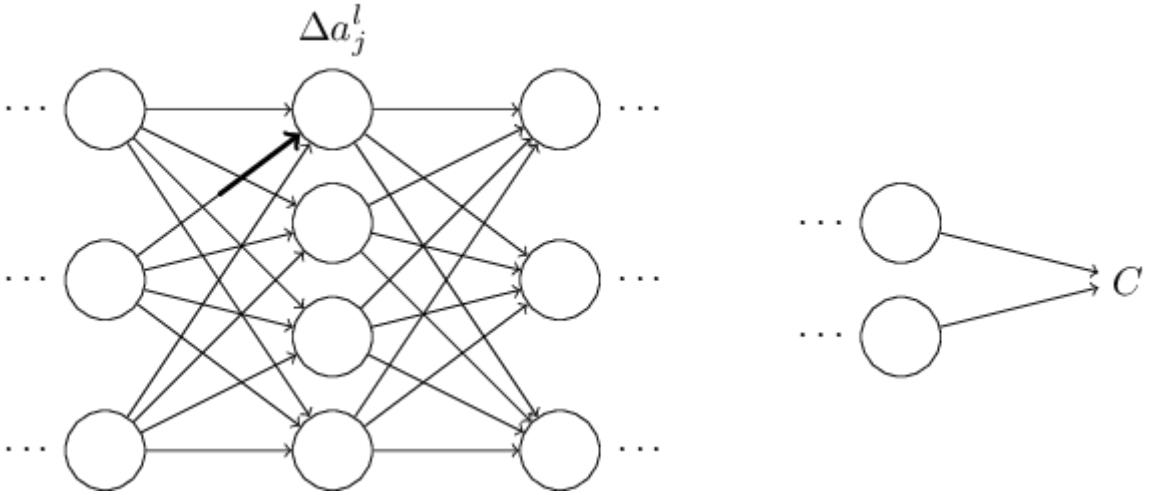
就像我之前阐述的，后向传播给我们带来了两大谜团。首先，这个算法到底在做什么？我们开发了一套从输出后向传播误差的构想。但是我们能更加深入一点，构建关于矩阵和向量乘积更加直观的解释吗？第二个谜团是某人怎样能够第一时间发现后向传播？按照算法步骤或者证明推导出这个算法是比较简单。但是这不意味着你能很好的理解这个问题，以至于能够第一时间发现这个算法。是否有某种合理的原因是你能够发现后向传播算法呢？本节我将阐述这些谜团。

为了提高我们对算法运作的直观感觉，让我们想象一下，对网络中的

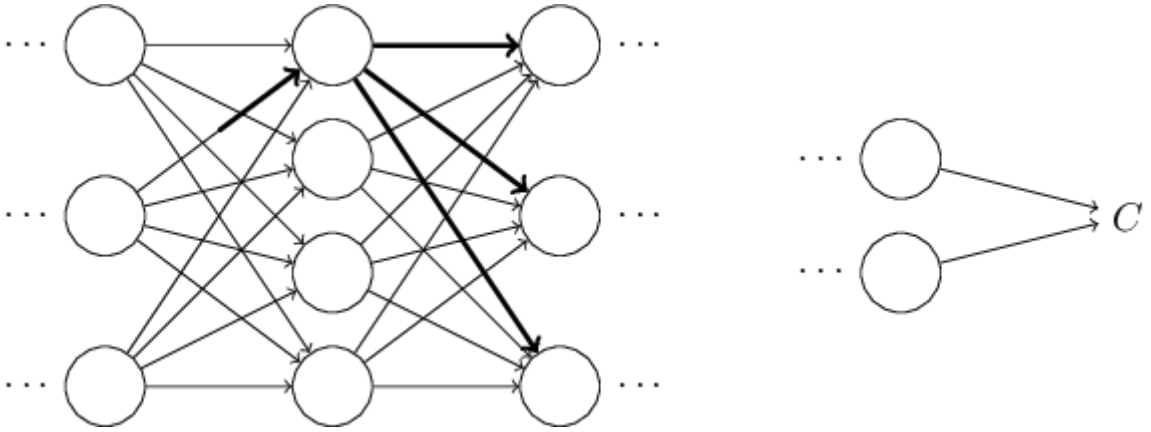
为了提高我们对算法运作的直观感觉，让我们想象一个，对网络中的某些权重w^l_{jk}进行很小的改变Δw^l_{jk}：



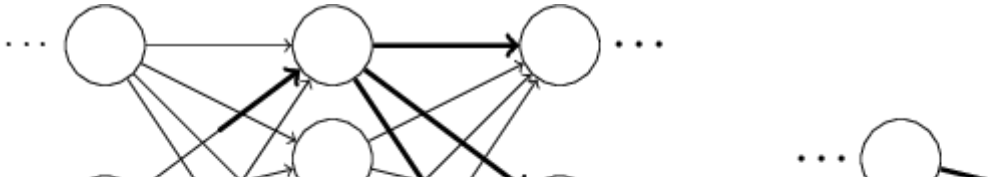
权重的改变将导致该神经元的输出产生变化：

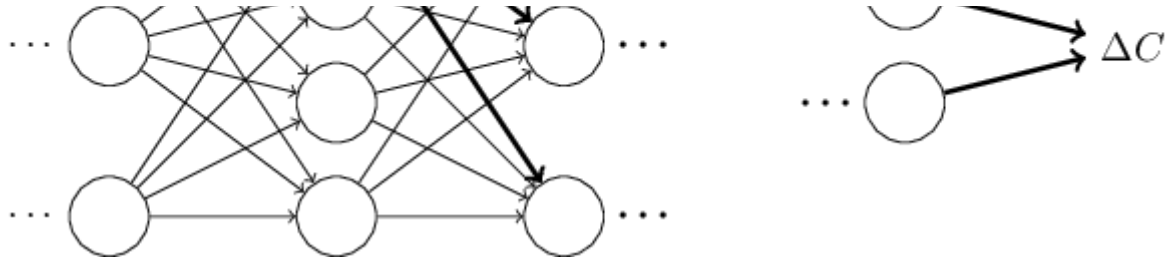


这样轮下去，会继续改变*所有*下一层的激活输出：



这些改变会继续改变更下一层，再下一层，如此继续下去直到最后一层，这是代价函数为：





代价改变ΔC与权重改变Δw^l_{jk}密切相关

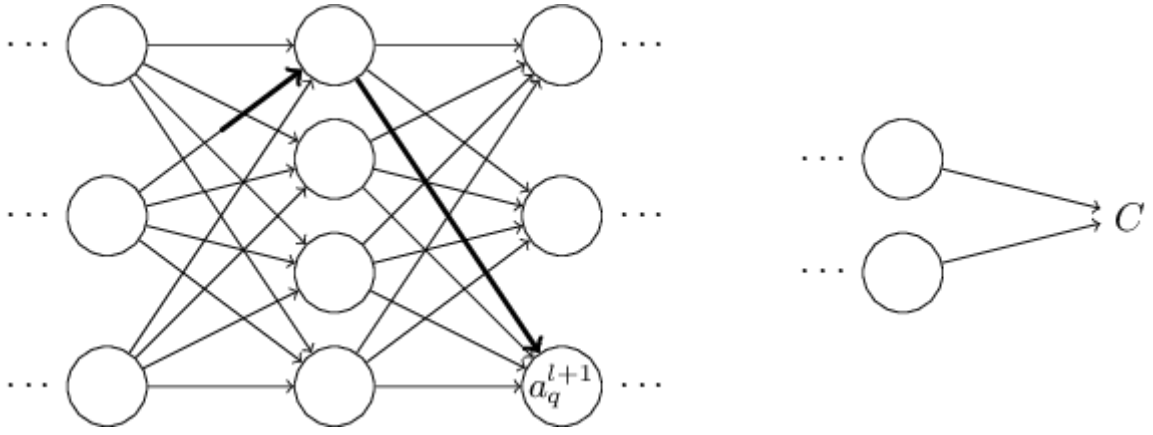
$$\Delta C \approx \frac{\partial C}{\partial w^l_{jk}} \Delta w^l_{jk}. \tag{47}$$

这个等式表明计算$\frac{\partial C}{\partial w^l_{jk}}$的一种可能途径是仔细跟踪$w^l_{jk}$的细小变化怎样传播引起$C$的细小变化。如果我们能这么做，仔细沿着这些可计算的量进行传播，那么我们就能够计算出$\partial C/\partial w^l_{jk}$。

让我们试着这么做，变化Δw^l_{jk}会导致网络第l^{th}层中第j^{th}个神经原的激活产生一个小变化Δa^l_j。这个变化为

$$\Delta a^l_j \approx \frac{\partial a^l_j}{\partial w^l_{jk}} \Delta w^l_{jk}. \tag{48}$$

这个激活变化Δa^l_j又会对下一层*所有*激活产生一些改变，例如，对$(l+1)^{\text{th}}$层。我们先只关注在受影响的激活中的其中一个，比如a^{l+1}_q，



实际上，它将产生如下的变化：

$$\Delta a^{l+1}_q \approx \frac{\partial a^{l+1}_q}{\partial a^l_j} \Delta a^l_j. \tag{49}$$

将表达式(48) 代入进来，我们将获得：

$$\Delta a_q^{l+1} \approx \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l. \tag{50}$$

显然，变化Δa_q^{l+1}将轮到继续往下一层激活继续传播改变。实际上，我们能构想沿着网络传播的一条路径，从w_{jk}^l一直改变到C，每一个激活改变都会产生下一个激活改变，最后，使得输出的代价产生改变。如果这条激活路径是沿着$a_j^l, a_q^{l+1}, \ldots, a_n^{L-1}, a_m^L$，那么最后结果表达式为

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l, \tag{51}$$

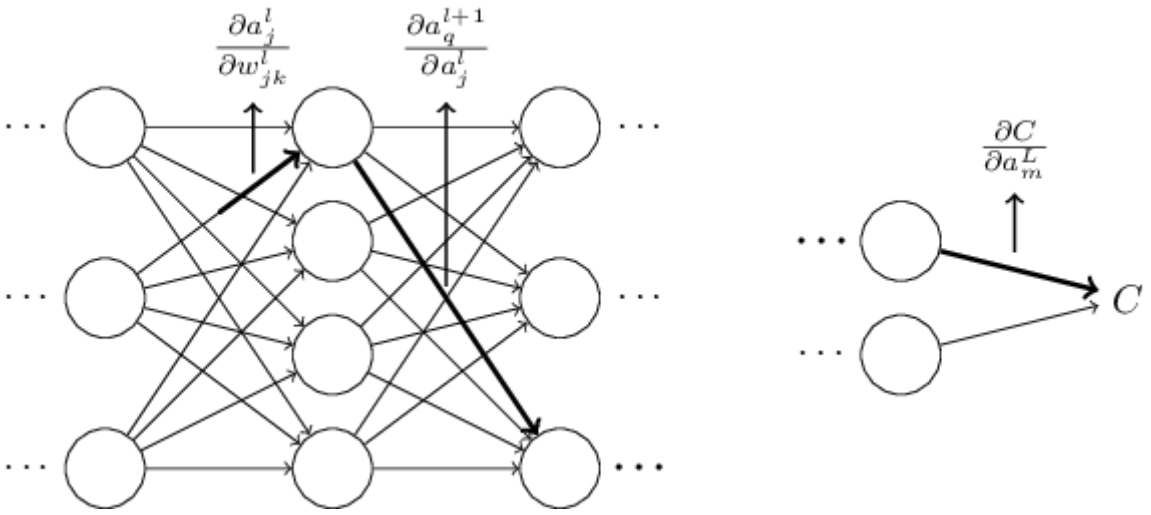
也就是，对我们经过的每一个神经元，使用$\partial a/\partial a$的项来表示，直到最后用$\partial C/\partial a_m^L$项。这就表明了沿着网络某一条特殊路径传播的激活改变对C产生的影响。当然，网络中有许多这样的路径传播w_{jk}^l的改变，使得对最后代价产生影响，而现在我们只考虑了其中一条路径。为了计算C的总的改变，我们需要将所有可能路径的权重和代价偏导进行求和。例如：

$$\Delta C \approx \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l, \tag{52}$$

其中我们对路径中所有可能的神经元影响进行求和。比较(47) ，我们看到

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l}. \tag{53}$$

现在，等式(53) 看起来非常复杂。但是，它具有更加直观的表述。我们计算网络中某个权重对C的改变率。这个等式告诉我们的是，两个神经原之间的边与它们的激活偏导数密切相关。从第一个神经元产生的第一个权重边有因子$\partial a_j^l/\partial w_{jk}^l$。路径的因子就是所有沿该条路径的因子乘积。且最后的总的改变率$\partial C/\partial w_{jk}^l$是权重到代价影响的所有传播路径的变化率的求和。对一个路径的描述如下：



直到现在我们提供了一种启发式讨论，一种考虑扰动网络权重将会引发什么的思路。让我们勾画出来这种思路来进一步发展这种讨论。首先，你需要对表达式(53) 中每一个偏导进行显示的推导。这只需要一点计算，还算简单。这么做了之后，你会尝试如何用矩阵乘法方式写出这个求和。这会是乏味的，也需要一些坚持，但是还不需要多么非凡的洞察力。这么做了之后，将会越来越简单，以至于后面你将逐步发现这个后向传播算法！你也能把后向传播算法当作一种计算所有路径上变化率因子的方法。或者稍微不同，后向传播算法是一种聪明的方法来跟踪权重（包括偏差）的扰动在网络中的传播，并到达输出，最后影响到代价函数值。

现在，我们纵观全章。这是非常混乱的，需要仔细的考虑所有细节的作用。如果你喜欢挑战，那么你将喜欢并尝试它。如果你不喜欢，我希望这也能够让你认识到后向传播完成了什么。

另外一个谜团呢——后向传播是如何被第一时间发现的？实际上，如果你沿着刚才我勾画出来的途径，你也会发现后向传播的证明。不幸的是，证明过程比起本章我前面描述的更加漫长和复杂。那么这个简洁的（更加成谜）证明是如何发现的？当你写完这个长的证明之后就会发现有好多简单的亮点出现在你面前。你就会得到这些简单思路，写出简单的证明。而后还有更多的简单思路跳出来。所以你再次重复。在进行完几次迭代之后，就会出现之前我们看到的证明*

*有一种
有a^{l+1}_q。
换，如
a^{l+1} ，

aq^{-},

——简

移出了。

上早期

简化我

已。

原文链接：http://neuralnetworksanddeeplearning.com

Posted in 神经网络教程

One response

Paul
2015年12月24日 下午2:16

I’m not that much of a online reader to be honest but your sites really nice, keep it up! I’ll go ahead and bookmark your site to come bacdk later on. Many thanks