**卷积神经网络(CNN)详解与代码实现**

**1.应用场景**

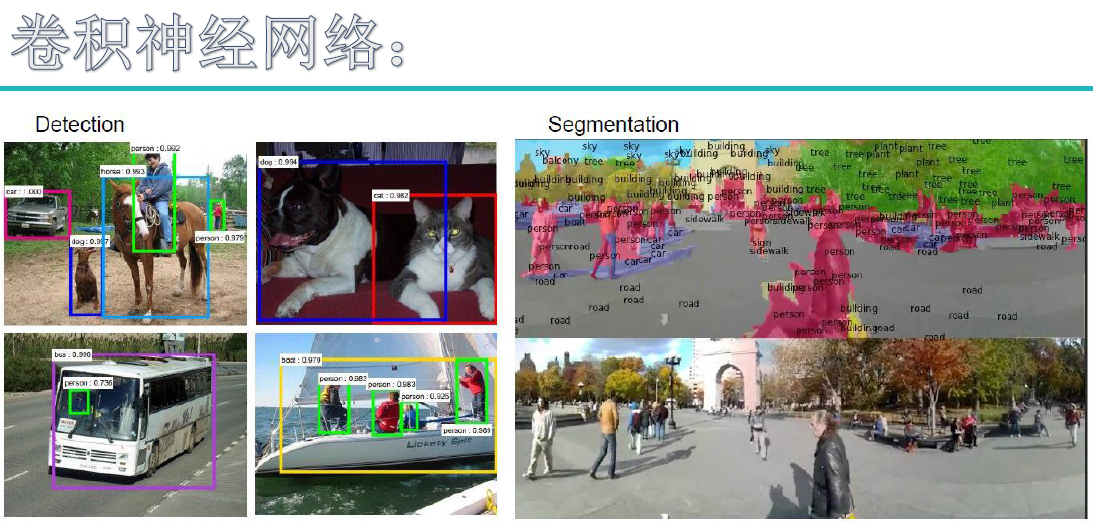
卷积神经网络的应用不可谓不广泛，主要有两大类，数据预测和图片处理。数据预测自然不需要多说，图片处理主要包含有图像分类，检测，识别，以及分割方面的应用。

图像分类：场景分类、目标分类

图像检测：显著性检测、物体检测、语义检测等等

图像识别：人脸识别、字符识别、车牌识别、行为识别、步态识别等等

图像分割：前景分割、语义分割



**2.卷积神经网络结构**

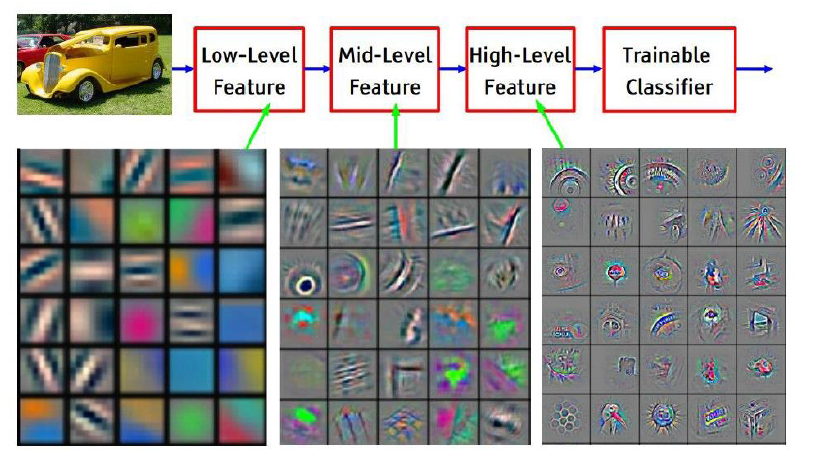
卷积神经网络主要是由输入层、卷积层、激活函数、池化层、全连接层、损失函数组成，表面看比较复杂，其实质就是特征提取以及决策推断。

要使特征提取尽量准确，就需要将这些网络层结构进行组合，比如经典的卷积神经网络模型AlexNet:5个卷积层+3个池化层+3个连接层结构。

**2.1 卷积（convolution）**

卷积的作用就是提取特征，因为一次卷积可能提取的特征比较粗糙，所以多次卷积，以及层层纵深卷积，层层提取特征（千万要区别于多次卷积，因为每一层里含有多次卷积）。

这里可能就有小伙伴问：为什么要进行层层纵深卷积，而且还要每层多次？你可以理解为物质A有自己的多个特征（高、矮、胖、瘦、、、），所以在物质A上需要多次提取，得到不同的特征，然后这些特征组合后发生化学反应生成物质B，而物质B又有一些新的专属于自己的特征，所以需要进一步卷积。这是我个人的理解，不对的话或者有更形象的比喻还请不吝赐教啊。



 在卷积层中，每一层的卷积核是不一样的。比如AlexNet

第一层：96\*11\*11（96表示卷积核个数，11表示卷积核矩阵宽\*高） stride（步长） = 4  pad（边界补零） = 0

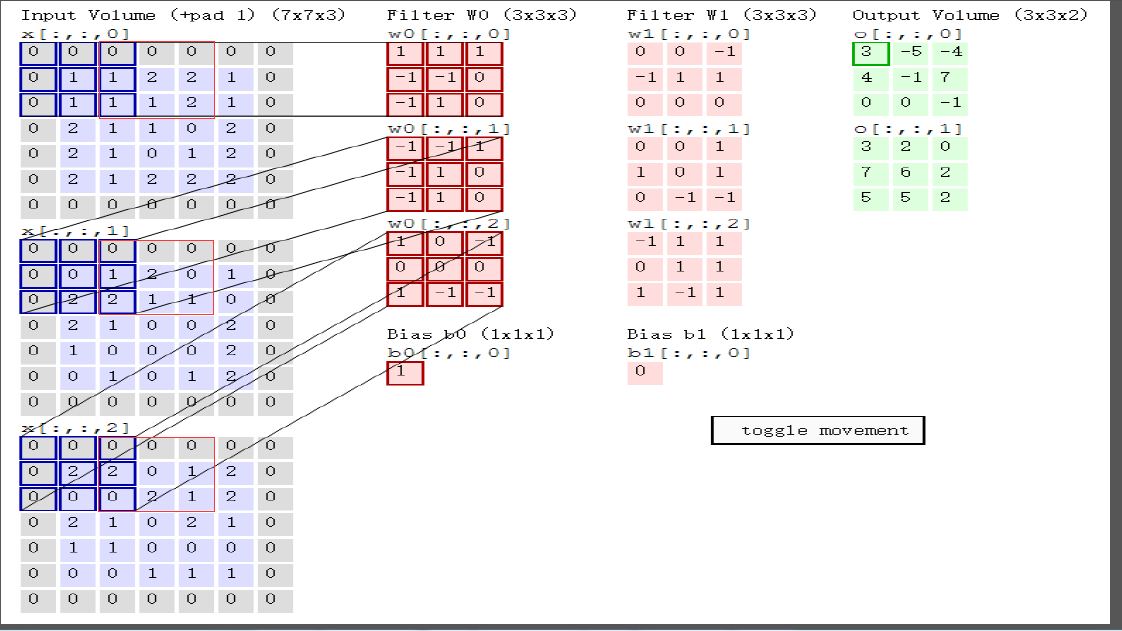
第二层：256\*5\*5 stride（步长） = 1  pad（边界补零） = 2

第三、四层：384\*3\*3 stride（步长） = 1  pad（边界补零） = 1

第五层：256\*3\*3 stride（步长） = 1  pad（边界补零） = 2

卷积的篇幅说了这么多，那么到底是如何进行运算的呢，虽说网络上关于卷积运算原理铺天盖地，但是个人总感觉讲得不够透彻，或者说本人智商有待提高，

希望通过如下这幅图（某位大神的杰作）来使各位看官们能够真正理解。



 这里举的例子是一个输入图片（5\*5\*3），卷积核（3\*3\*3），有两个（Filter W0，W1），偏置b也有两个（Bios b0，b1），卷积结果Output Volumn（3\*3\*2），步长stride = 2。

输入：7\*7\*3 是因为 pad = 1 （在图片边界行和列都补零，补零的行和的数目是1），

（对于彩色图片，一般都是RGB3种颜色，号称3通道，7\*7指图片高h \* 宽w），补零的作用是能够提取图片边界的特征。

卷积核深度为什么要设置成3呢？这是因为输入是3通道，所以卷积核深度必须与输入的深度相同。至于卷积核宽w，高h则是可以变化的，但是宽高必须相等。

卷积核输出o[0,0,0] = 3 （Output Volumn下浅绿色框结果），这个结果是如何得到的呢？ 其实关键就是矩阵对应位置相乘再相加（千万不要跟矩阵乘法搞混淆啦）

=> w0[:,:,0] \* x[:,:,0]蓝色区域矩阵(R通道) +  w0[:,:,1] \* x[:,:,1]蓝色区域矩阵（G通道）+  w0[:,:,2] \* x[:,:,2]蓝色区域矩阵（B通道） + b0（千万不能丢，因为 y = w \* x + b）

第一项  => 0 \* 1 + 0 \* 1 + 0 \* 1 + 0 \* (-1) + 1 \* (-1) + 1 \* 0 + 0 \* (-1) + 1 \* 1 + 1 \* 0  =  0

第二项 => 0 \* (-1) + 0 \* (-1) + 0 \* 1 + 0 \* (-1) + 0 \* 1 + 1 \* 0 + 0 \* (-1) + 2 \* 1 + 2 \* 0 = 2

第三项 => 0 \* 1 + 0 \* 0 + 0 \* (-1) + 0 \* 0 + 2 \* 0 + 2 \* 0 + 0 \* 1 + 0 \* (-1) + 0 \* (-1) = 0

卷积核输出o[0,0,0] = > 第一项 + 第二项 + 第三项 + b0 = 0 + 2 + 0 + 1 = 3

o[0,0,1] = -5 又是如何得到的呢？

因为这里的stride = 2 ，所以 输入的窗口就要滑动两个步长，也就是红色框的区域，而运算跟之前是一样的

第一项  => 0 \* 1 + 0 \* 1 + 0 \* 1 + 1 \* (-1) + 2 \* (-1) + 2 \* 0 + 1 \* (-1) + 1 \* 1 + 2 \* 0 = -3

第二项 => 0 \* (-1) + 0 \* (-1) + 0 \* 1 + 1 \* (-1) + 2 \* 1 + 0 \* 0 + 2 \* (-1) + 1 \* 1 + 1 \* 0 = 0

第三项 => 0 \* 1 + 0 \* 0 + 0 \* (-1) + 2 \* 0 + 0 \* 0 + 1 \* 0 + 0 \* 1 + 2 \* (-1) + 1 \* (-1)  = - 3

卷积核输出o[0,0,1] = > 第一项 + 第二项 + 第三项 + b0 = (-3) + 0 + (-3) + 1 = -5

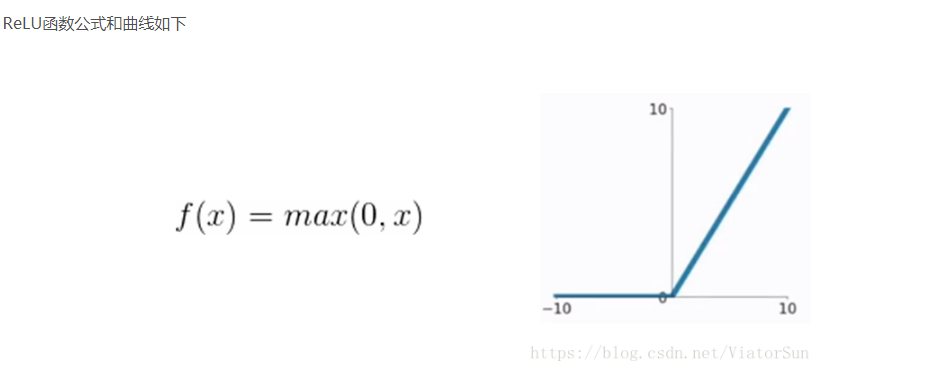
之后以此卷积核窗口大小在输入图片上滑动，卷积求出结果，因为有两个卷积核，所有就有两个输出结果。

这里小伙伴可能有个疑问，输出窗口是如何得到的呢？

这里有一个公式：**输出窗口宽 w = (输入窗口宽 w - 卷积核宽 w + 2 \* pad)/stride  + 1 ，输出高 h  = 输出窗口宽 w**

以上面例子， 输出窗口宽 w = ( 5 - 3 + 2 \* 1)/2 + 1 = 3 ，则输出窗口大小为 3 \* 3，因为有2个输出，所以是 3\*3\*2。

**2.2 激活函数（Relu）**



相信看过卷积神经网络结构（CNN）的伙伴们都知道，激活函数无处不在，特别是CNN中，在卷积层后，全连接（FC）后都有激活函数Relu的身影，那么这就自然不得不让我们产生疑问：

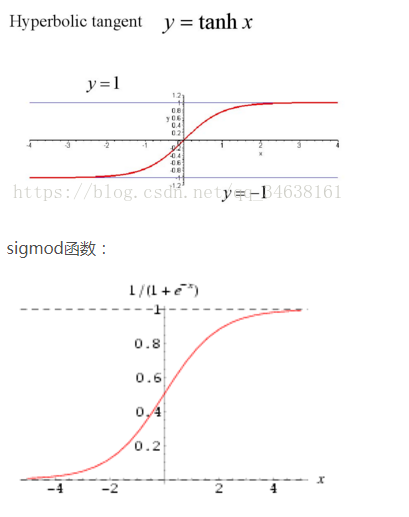
问题1、为什么要用激活函数？它的作用是什么？

问题2、在CNN中为什么要用Relu，相比于sigmoid，tanh，它的优势在什么地方？

对于第1个问题：由 y = w \* x + b 可知，如果不用激活函数，每个网络层的输出都是一种线性输出，而我们所处的现实场景，其实更多的是各种非线性的分布。

这也说明了激活函数的作用是将线性分布转化为非线性分布，能更逼近我们的真实场景。

对于第2个问题： 先看sigmoid，tanh分布



他们在 x ->https://gss1.bdstatic.com/-vo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D15/sign=d11bc50915d8bc3ec20802cf838b1dd3/f9198618367adab4f7776b7087d4b31c8701e45a.jpg 时，输出就变成了恒定值，因为求梯度时需要对函数求一阶偏导数，而不论是sigmoid，还是tanhx，他们的偏导都为0，

也就是存在所谓的梯度消失问题，最终也就会导致权重参数w ， b 无法更新。相比之下，Relu就不存在这样的问题，另外在 x > 0 时，Relu求导 = 1，这对于反向传播计算dw，db，是能够大大的简化运算的。

使用sigmoid还会存在梯度爆炸的问题，比如在进行前向传播和反向传播迭代次数非常多的情况下，sigmoid因为是指数函数，其结果中

某些值会在迭代中累积，并成指数级增长，最终会出现NaN而导致溢出。

**2.3 池化（pool）**

池化层一般在卷积层+ Relu之后，它的作用是：

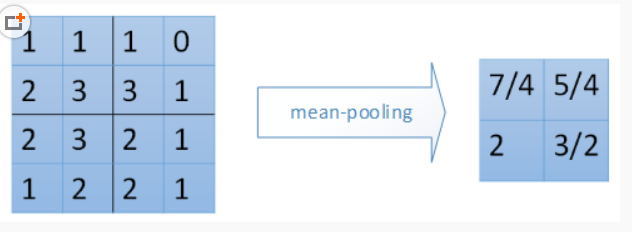
1、减小输入矩阵的大小（只是宽和高，而不是深度），提取主要特征。（不可否认的是，在池化后，特征会有一定的损失，所以，有些经典模型就去掉了池化这一层）。

它的目的是显而易见的，就是在后续操作时能降低运算。

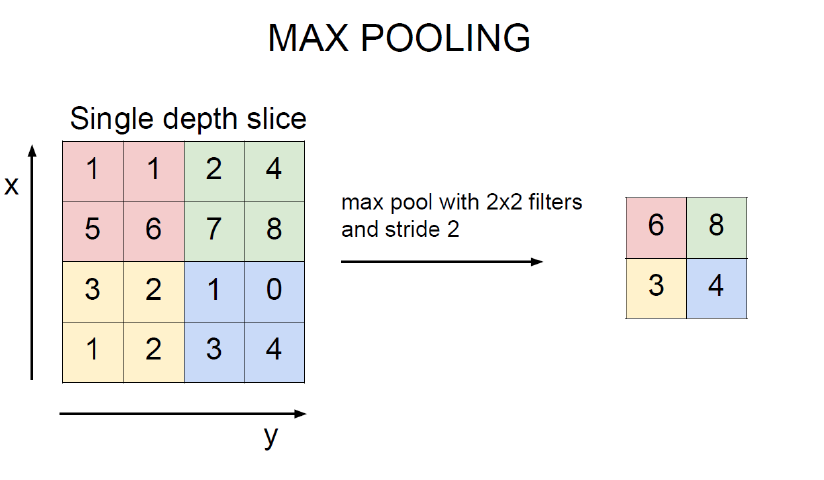
2、一般采用mean\_pooling（均值池化）和max\_pooling（最大值池化），对于输入矩阵有translation（平移），rotation（旋转），能够保证特征的不变性。

mean\_pooling 就是输入矩阵池化区域求均值，这里要注意的是池化窗口在输入矩阵滑动的步长跟stride有关，一般stride = 2.（图片是直接盗过来，这里感谢原创）

最右边7/4 => (1 + 1 + 2 + 3)/4



max\_pooling 最大值池化，就是每个池化区域的最大值放在输出对应位置上。



**2.4 全连接（full connection）**

作用：分类器角色，将特征映射到样本标记空间，本质是矩阵变换（affine）。

至于变换的实现见后面的代码流程图，或者最好是跟一下代码，这样理解更透彻。

**2.5 损失函数（softmax\_loss）**

作用：计算损失loss，从而求出梯度grad。

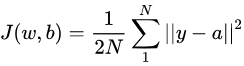
常用损失函数有：MMSE均方误差损失函数、SVM合页损失函数、Cross Entropy交叉熵损失函数、目标检测中常用的Smooth L1损失函数。其中还会涉及到梯度消失、梯度爆炸等问题：ESM均方误差+Sigmoid激活函数会导致学习缓慢；Smooth L1损失是为了解决梯度爆炸问题。仅供参考。

这几种损失函数目前还看不出谁优谁劣，估计只有在具体的应用场景中去验证了。至于这几种损失函数的介绍，

**a．均方误差损失**

**a.1均方差损失函数的定义：**

均方差损失函数常用在最小二乘法中。它的思想是使得各个训练点到最优拟合线的距离最小（平方和最小）。均方差损失函数也是我们最常见的损失函数了，相信大很熟悉了，我们以神经网络中激活函数的形式表达一下，定义如下：

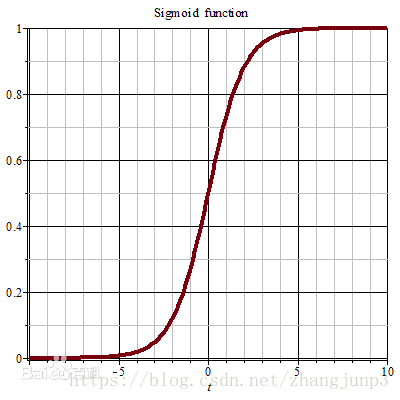




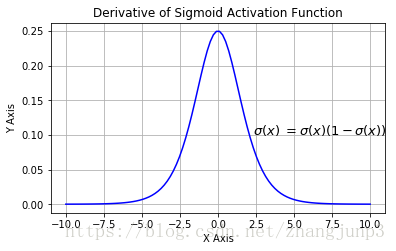
a.2 ESM均方误差+Sigmoid激活函数：输出层神经元学习率缓慢

a.2.1 Sigmoid激活函数：

这个激活函数再熟悉不过了，该函数能将负无穷到正无穷的数映射到0和1之间。先来看一下表达式以及函数图像：

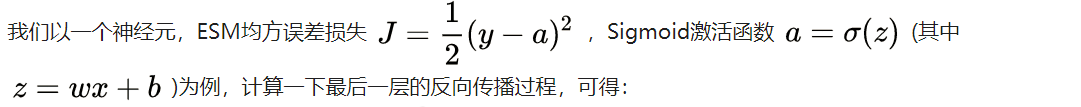


Sigmoid的导数推导以及图像：  
https://img-blog.csdn.net/20180527101606684?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW5nanVucDM=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

  
从sigmiod的导数图像中可以看到，除了中间比较小的区域，其他区域的十分值接近于0。

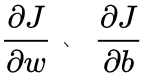
神经网络的反向传播是逐层对函数偏导相乘，因此当神经网络层数非常深的时候，最后一层产生的偏差（网络输出和标签之间的偏差）因为乘了很多的小于1的数而越来越小，最终就会变为0，从而导致层数比较浅的权重w没有更新，即梯度消失。可以看出，sigmoid函数作为激活函数本身就存在梯度消失的问题。

a.2.2 ESM均方误差+Sigmoid激活函数：输出层神经元学习率缓慢



https://img-blog.csdn.net/20180527101951700?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW5nanVucDM=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

https://img-blog.csdn.net/20180527102008181?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW5nanVucDM=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

可以看到最后一层反向传播时，所求的梯度中都含有。经过上面的分析，当神经元输出接近1时候，Sigmoid的导数变很小，这样 很小，这就导致了ESM均方误差+Sigmoid激活函数使得神经网络反向传播的起始位置——输出层神经元学习率缓慢。

想要解决这个问题，需要引入接下来介绍的交叉熵损失函数。这里先给出结论：交叉熵损失+Sigmoid激活函数可以解决输出层神经元学习率缓慢的问题，但是不能解决隐藏层神经元学习率缓慢的问题。具体的推导和总结在下面部分中介绍。

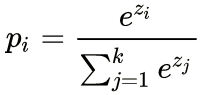
**b.叉熵损失**

**b.1交叉损失的定义**

交叉熵损失的计算分为两个部分。

**b.1.1 soft max分类器**

交叉熵损失是基于softmax计算来的，softmax将网络最后输出z通过指数转变成概率形式。首先看一下softmax计算公式：



其中， 分子是要计算的类别 的网络输出的指数；分母是所有类别网络输出的指数和，共k个类别。这样就得到了类别i的输出概率。

→这里说点题外话，实际上，softmax是由逻辑斯的回归模型（用于二分类）推广得到的多项逻辑斯蒂回归模型（用于多分类）。具体可以参考李航大神的《统计学方法》第六章，这里给一个大致的过程。

逻辑回归的P(Y=y|x)表达式如下（为了将类别标签y统一为1和0，下面将表达式分开表示）：



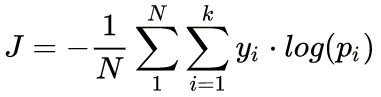
将它带入到上式，通过推导可以得到logistic的损失函数表达式，如下：

https://img-blog.csdn.net/20180527102753124?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW5nanVucDM=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

逻辑回归最后得到的目标式子如下：



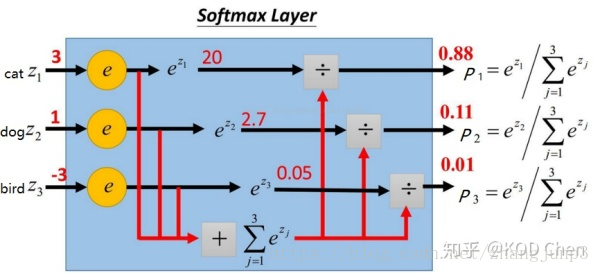
b.1.2 交叉熵损失

公式定义如下：

其中，是类别的真实标签；是上面softmax计算出的类别的概率值；k是类别数，N是样本总数。

这里看一个计算交叉熵损失的小例子：

假设共有三个类别cat、dog、bird，那么一张cat的图片标签应该为。并且训练过程中，这张cat的图片经过网络后得到三个类别网络的输出分别为3、1、-3。那么经过softmax可以得到对应的概率值，如下图：

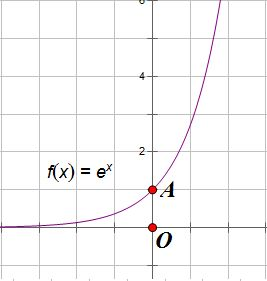


https://img-blog.csdn.net/20180527103703174?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW5nanVucDM=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

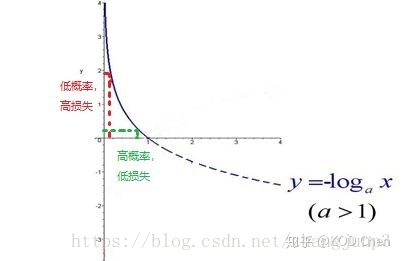
b.2 交叉熵损失的两个图像

b.2.1 指数图像

softmax分类器将各个类别的“得分”（网络输出）转变成概率值。并取e指数使得“得分”高的类别对应的概率更大，使得损失函数对网络输出“更敏感”，更有利于分类。



b.2.2 对数图像



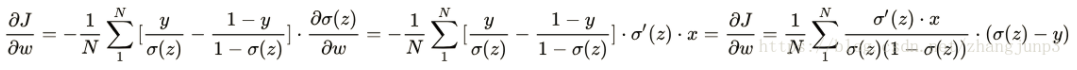
b.2.3 交叉熵损失+Sigmoid激活函数：

接着上一部分留下的问题，我们仍然以Sigmoid激活函数。这次我们引入交叉熵损失，并以二分类为例，那么s损失函数公式为：

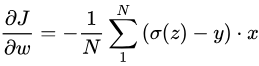


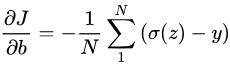
https://img-blog.csdn.net/20180527104327305?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW5nanVucDM=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

那么可以计算一下最后一层的反向传播过程，可得：



根据之前的推导已知，那么上式可以化简为：





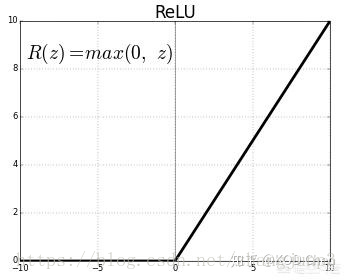
可以看到sigmoid的导数被约掉，这样最后一层的梯度中就没有。然而这只是输出层的推导，如果变成隐藏层的梯度sigmoid的导数不会被约掉，仍然存在。所以交叉熵损失+Sigmoid激活函数可以解决输出层神经元学习率缓慢的问题，但是不能解决隐藏层神经元学习率缓慢的问题。

其实损失函数包含两个部分：①计算方法（均方差、交叉熵等）②激活函数。

而之前我们遇到的是均方差损失+sigmoid激活函数造成了输出层神经元学习率缓慢，其实我们破坏任意一个条件都有可能解决这个问题：

①均方误差损失→交叉熵损失；

②sigmoid函数→不会造成梯度消失的函数，例如ReLU函数，不仅能解决输出层学习率缓慢，还能解决隐藏层学习率缓慢问题。



这里也小结一下ReLU函数相对于tanh和sigmoid函数好在哪里：

第一，采用sigmoid等函数，算激活函数是（指数运算），计算量大；反向传播求误差梯度时，求导涉及除法，计算量相对大。而采用Relu激活函数，整个过程的计算量节省很多。

第二，对于深层网络，sigmoid函数反向传播时，很容易就会出现梯度消失的情况（在sigmoid接近饱和区时，变换太缓慢，导数趋于0），这种情况会造成信息丢失，梯度消失在网络层数多的时候尤其明显，从而无法完成深层网络的训练。

第三，ReLU会使一部分神经元的输出为0，这样就造成了网络的稀疏性，并且减少了参数的相互依存关系，缓解了过拟合问题的发生。

**d. svm合页损失**

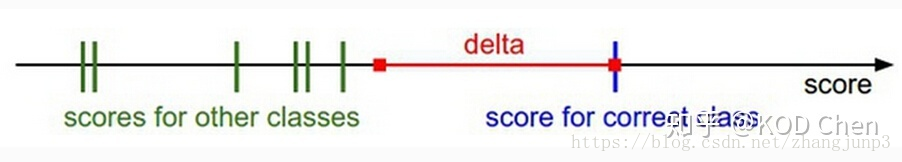
d.1 定义

合页损失函数想让正确分类的“得分”比其他错误分类的“得分”高出至少一个边界值。

如果正确分类的得分与错误分类的得分差值比边界值还要高，就会认为损失值是0；如果没有就要计算损失了。看一下计算公式和示意图：



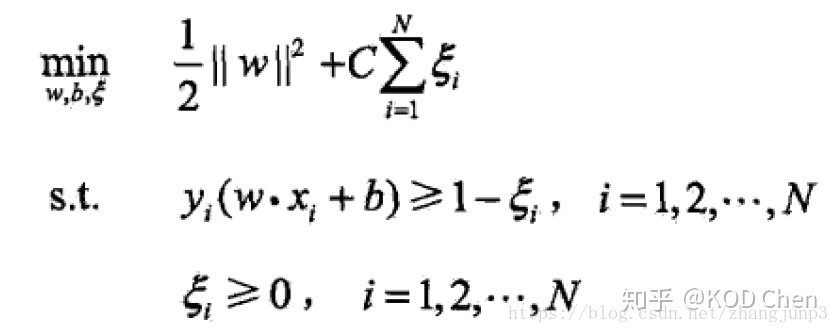
其中，是正确分类的得分、是其他错误分类的得分；是指想要正确类别的分类得分比其他错误分类类别的得分要高且至少高出 的边界值；k是类别数，N是样本总数。示意图如下：



这里看一个计算合页损失的小例子：仍然假设共有三个类别cat、dog、bird，那么一张cat的图片标签应该为 。并且训练过程中，这张cat的图片经过网络后得到三个类别网络的输出分别为3、1、-3。我们取 。此时：。其实直观上也很好理解，分类正确的得分是3，其他错误类别得分是1和-3，而我们希望分类正确的得分比其他分类错误的得分高 的边界值。显然错误分类得分为1的没有符合条件，则计算损失。

**d.2 特点**

合页损失函数其实就是线性支持向量机中，对于一些线性不可分的数据，引入了松弛变量。这样，目标函数和约束条件就变成了：





其中前面的就是合页损失函数。后面的是正则项。

线性支持向量机也是希望不仅仅可以求出分类超平面，同时也希望正确分类比其他错误分类多出一个边界值，即分类间隔，SVM目的也就是最大化分类间隔。而引入的松弛因子其实就是计算的合页损失项。

尽管合页损失函数希望正确分类的得分比其他错误分类的得分高出至少一个边界值 ，但是对于得分数字的细节是不关心的，看一个小例子：

如果两个分类器最后得分是[3,-10, -10]和[3,-2, -2]，且，那么对于合页损失来讲没什么不同，只要满足超过边界值5，那么损失值就都等于0。然而，显然第一个分类器比第二个分类器效果更好，因为高出边界更大，但是合页损失都是0，这就是合页损失对于得分数字的细节是不关心的造成的缺点。

想要解决这一问题，其实上面的交叉熵损失很好的解决这一问题，因为交叉熵将得分转变成概率，就不会造成上面说的情况；并且交叉熵损失也扩大了正确分类和错误分类得分的差距，对分数敏感，同样能得到较好的分类效果

**e.Smooth L1损失**

Smooth L1损失是为了解决梯度爆炸问题的。在看Smooth L1损失之前，先看一下梯度爆炸。

e.1梯度爆炸：

在深层神经网络或循环神经网络中，误差的梯度可在更新中累积相乘。如果网络层之间的梯度值大于 1.0，那么重复相乘会导致梯度呈指数级增长，梯度变的非常大，然后导致网络权重的大幅更新，并因此使网络变得不稳定。

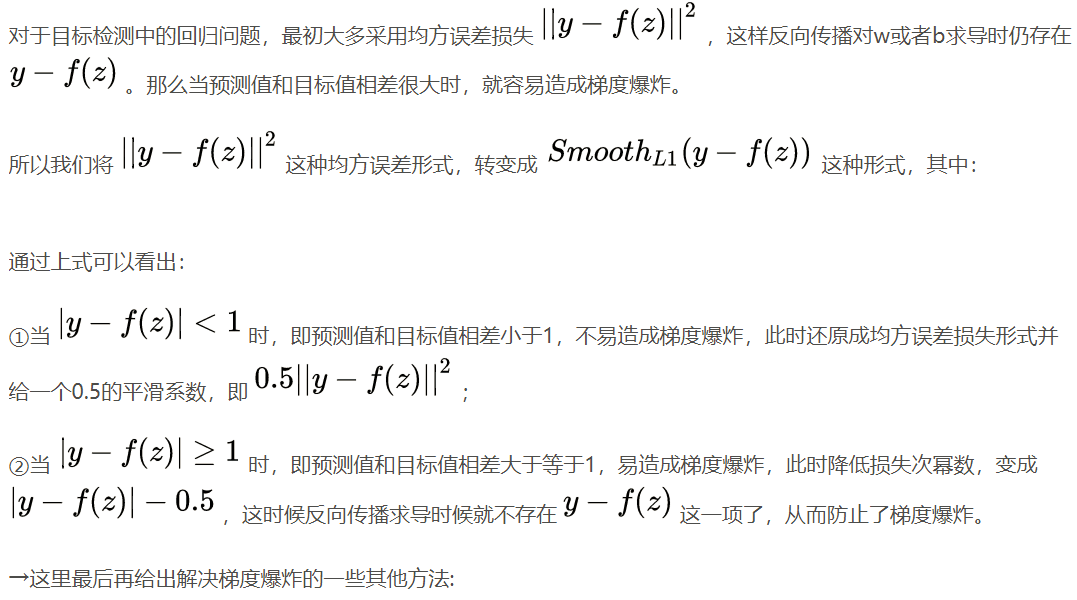
梯度爆炸会伴随一些细微的信号，如：

①模型不稳定，导致更新过程中的损失出现显著变化；

②训练过程中，在极端情况下，权重的值变得非常大，以至于溢出，导致模型损失变成 NaN等等。

**e.2 Smooth L1损失：**

Smooth L1损失函数是在Fast R-CNN中被提出，主要目的是为了防止梯度爆炸。



（1）减少学习率（个人理解梯度爆炸是模型训练发散的一种情况）；

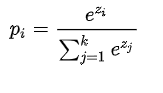
（2）使用ReLU函数，使得梯度稳定；

（3）使用正则化，即检查网络中权重的大小，对较大的权重进行惩罚，限制了梯度爆炸造成的权重变得很大的情况。

===================================

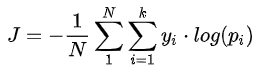
在后面的代码实例中，用到的是softmax\_loss，它属于Cross Entropy交叉熵损失函数。

softmax计算公式：



其中，https://latex.codecogs.com/gif.latex?%5CLARGE%20_%7Bz_%7Bi%7D%7D 是要计算的类别 https://latex.codecogs.com/gif.latex?%5Clarge%20i 的网络输出，分母是网络输出所有类别之和（共有 https://latex.codecogs.com/gif.latex?%5Clarge%20k 个类别），https://latex.codecogs.com/gif.latex?%5CLARGE%20_%7Bp_%7Bi%7D%7D 表示第 https://latex.codecogs.com/gif.latex?%5Clarge%20i 类的概率。

交叉熵损失：



其中，https://latex.codecogs.com/gif.latex?%5Clarge%20y_%7Bi%7D 是类别 https://latex.codecogs.com/gif.latex?%5Clarge%20i 的真实标签，https://latex.codecogs.com/gif.latex?%5CLARGE%20_%7Bp_%7Bi%7D%7D 表示第 https://latex.codecogs.com/gif.latex?%5Clarge%20i 类的概率，https://latex.codecogs.com/gif.latex?%5Clarge%20N 是样本总数，https://latex.codecogs.com/gif.latex?%5Clarge%20k 是类别数。

梯度:

https://latex.codecogs.com/gif.latex?%5Clarge%20grad_%7Bj%7D=  https://latex.codecogs.com/gif.latex?%5Clarge%20ypred_%7Bi%7D%5E%7B%7D      当  https://latex.codecogs.com/gif.latex?%5Clarge%20j != https://latex.codecogs.com/gif.latex?%5Clarge%20i

https://latex.codecogs.com/gif.latex?%5Clarge%20grad_%7Bj%7D=  https://latex.codecogs.com/gif.latex?%5Clarge%20ypred_%7Bi%7D%5E%7B%7D  - 1   当  https://latex.codecogs.com/gif.latex?%5Clarge%20j = https://latex.codecogs.com/gif.latex?%5Clarge%20i

其中 https://latex.codecogs.com/gif.latex?%5Clarge%20ypred_%7Bi%7D%5E%7B%7D 表示真实标签对应索引下预测的目标值，https://latex.codecogs.com/gif.latex?%5Clarge%20j 类别索引。

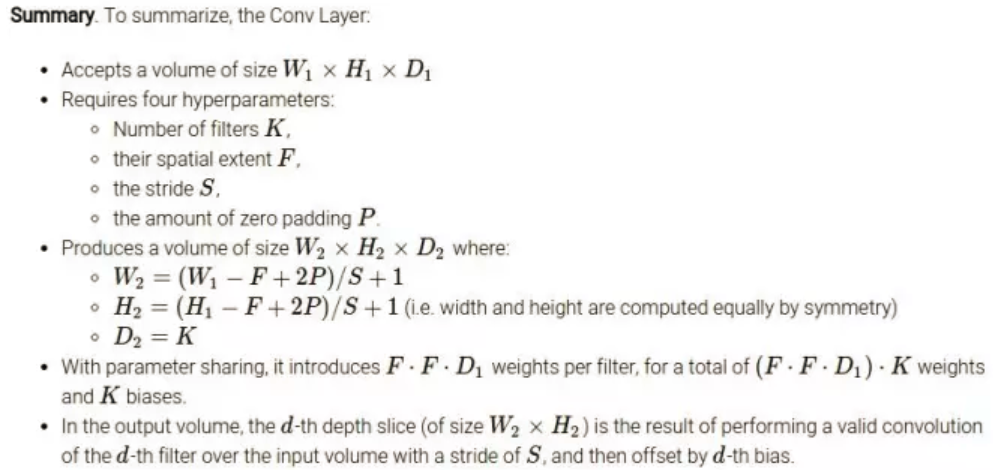
**2.6 前向传播（forward propagation）**

前向传播包含之前的卷积，Relu激活函数，池化（pool），全连接(fc)，可以说，在损失函数之前操作都属于前向传播。

主要是权重参数w , b 初始化，迭代，以及更新w, b,生成分类器模型。

**代码函数名：Conv Layer Naive Forward**

卷积层的pad，输出的dimension计算如下：

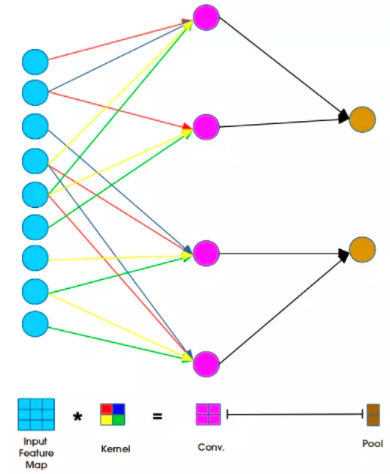


注意kernel是对输入层的所有层都进行卷积，所以如果只有一个filter那么输出只有1层，如果有n个filter，那么输出就有n层了。**一个filter中用于卷积的kernel数取决于上一层的输入channel的个数，每个channel对应一个kernel**，代码见代码

**2.7 反向传播（back propagation）**

反向传播包含损失函数，通过梯度计算dw，db，Relu激活函数逆变换，反池化，反全连接。

**代码函数名Conv Layer Backward Naive**



个人总结的感性认识：

dx一定和x的shape相同，dw一定和w的shape相同，db一定和b的shape相同，dout和out的shape一定相同。

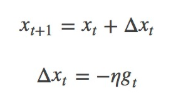
只要考虑下它们的意义就可以知道。dL/dx的含义（也就是代码中的dx）是x的改变会多大程度地改变L，而x的改变也只能是在自己的shape下。

在计算的时候先将x进行pad。我们知道在卷积的过程中，每一次卷积运算，对应于上图的2×2和2×2矩阵的作用，其实就是一个相乘累加过程。对于out层的每一个元素，都是由input layer和weight矩阵对应的相乘累加得到的。以dx为例，input矩阵x中的每个元素可能会参与１到几次不等的运算，与weight相乘将结果贡献到out layer中，比如x(0,0)只参与了一次，那么它的dx(0,0)=weight(0,0)×dout(...)，而x(0,1)参与了两次，分别贡献到out的不同位置，那么dx(0,1)=weight(0,0)×dout(...)+weight(0,1)×dout(...)，以此类推。

**2.8 随机梯度下降（sgd\_momentum）**

作用：由梯度grad计算新的权重矩阵w

sgd公式：



其中，η为学习率，gt为x在t时刻的梯度。

一般我们是将整个数据集分成n个epoch，每个epoch再分成m个batch，每次更新都利用一个batch的数据，而非整个训练集。

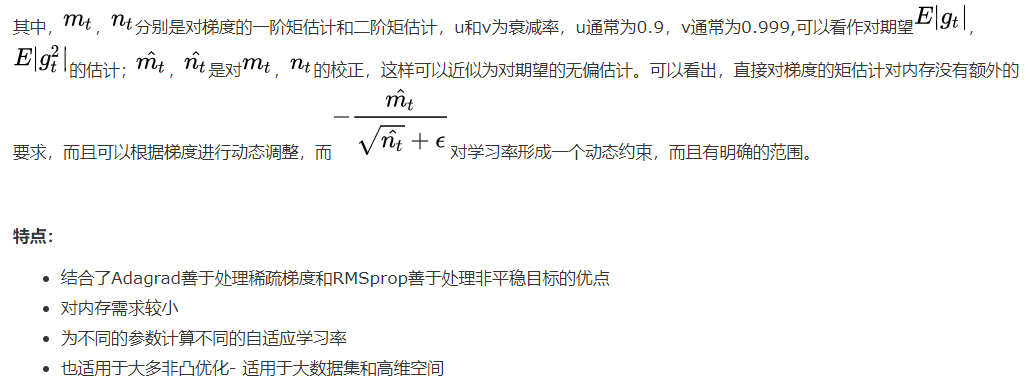
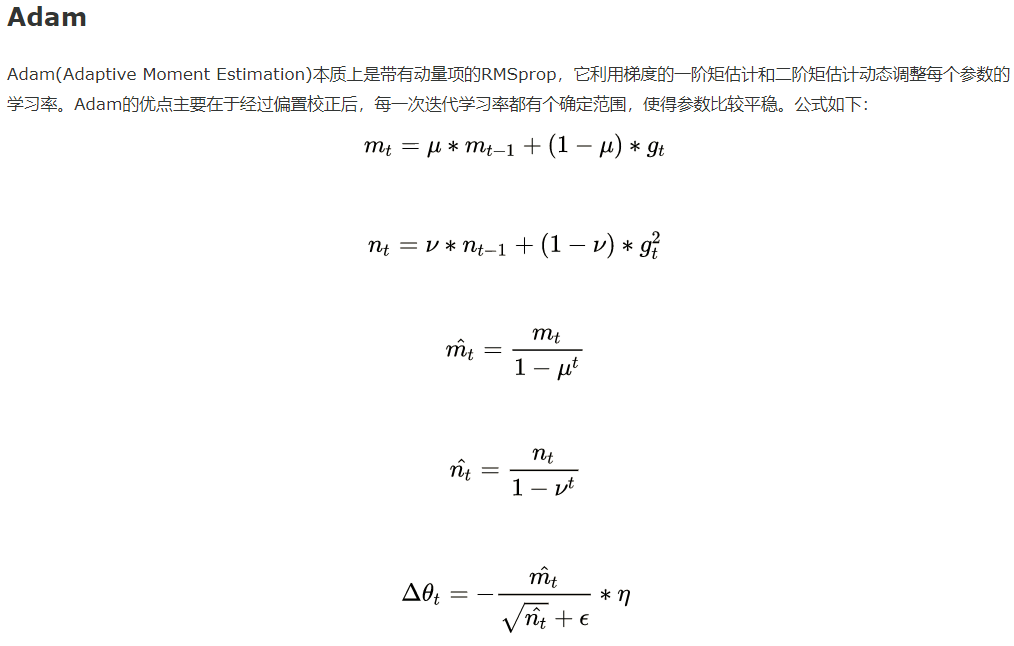
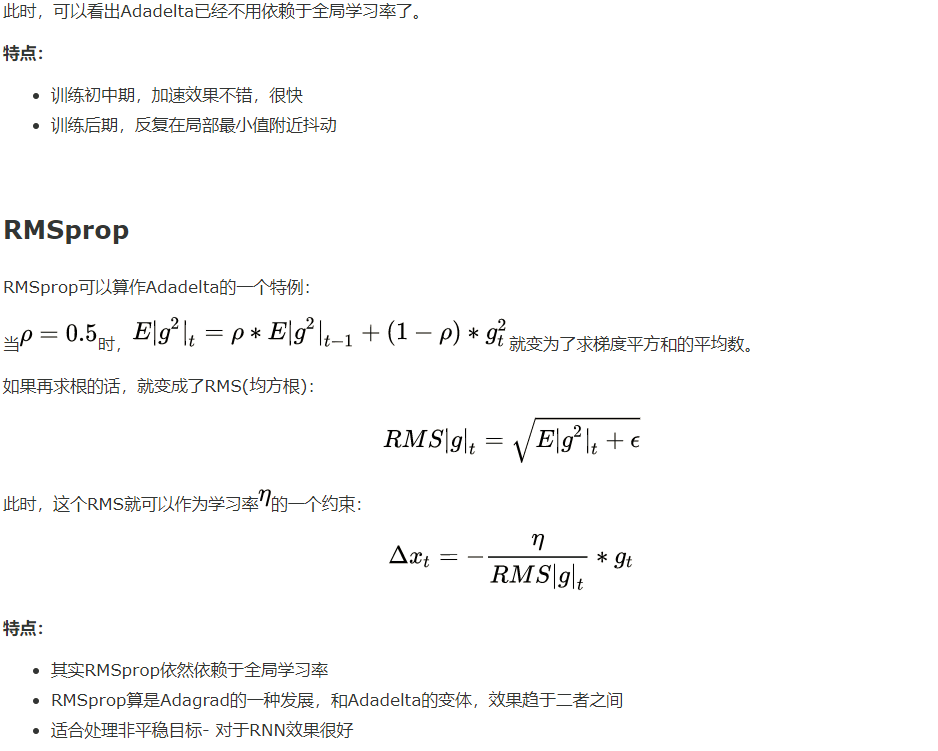
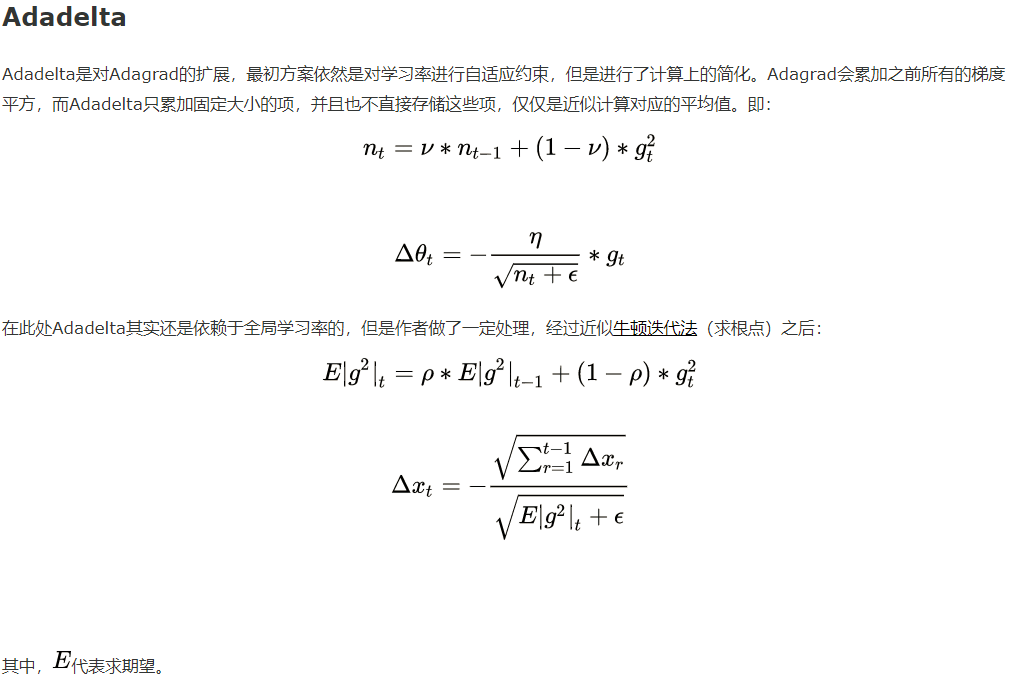
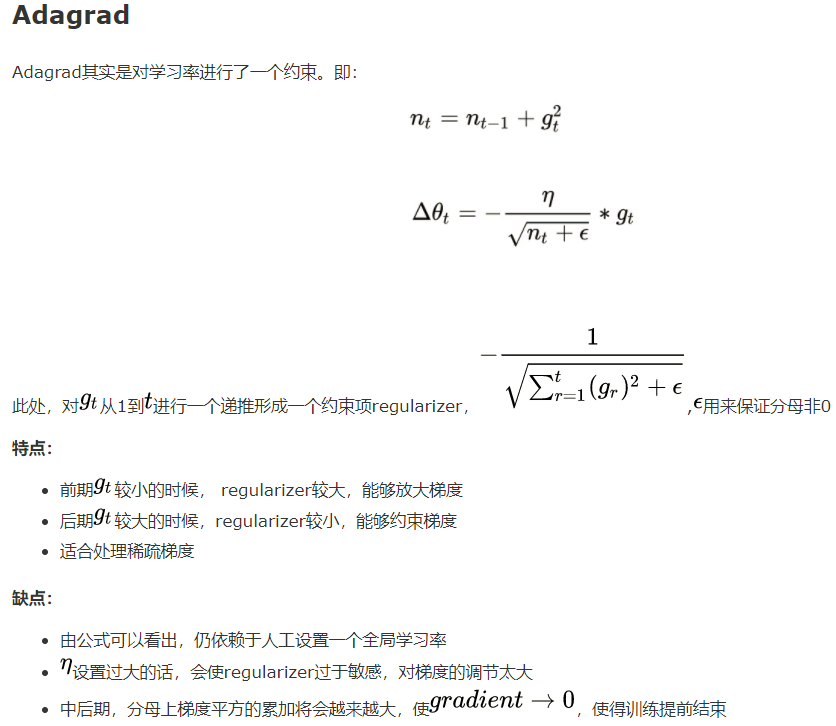
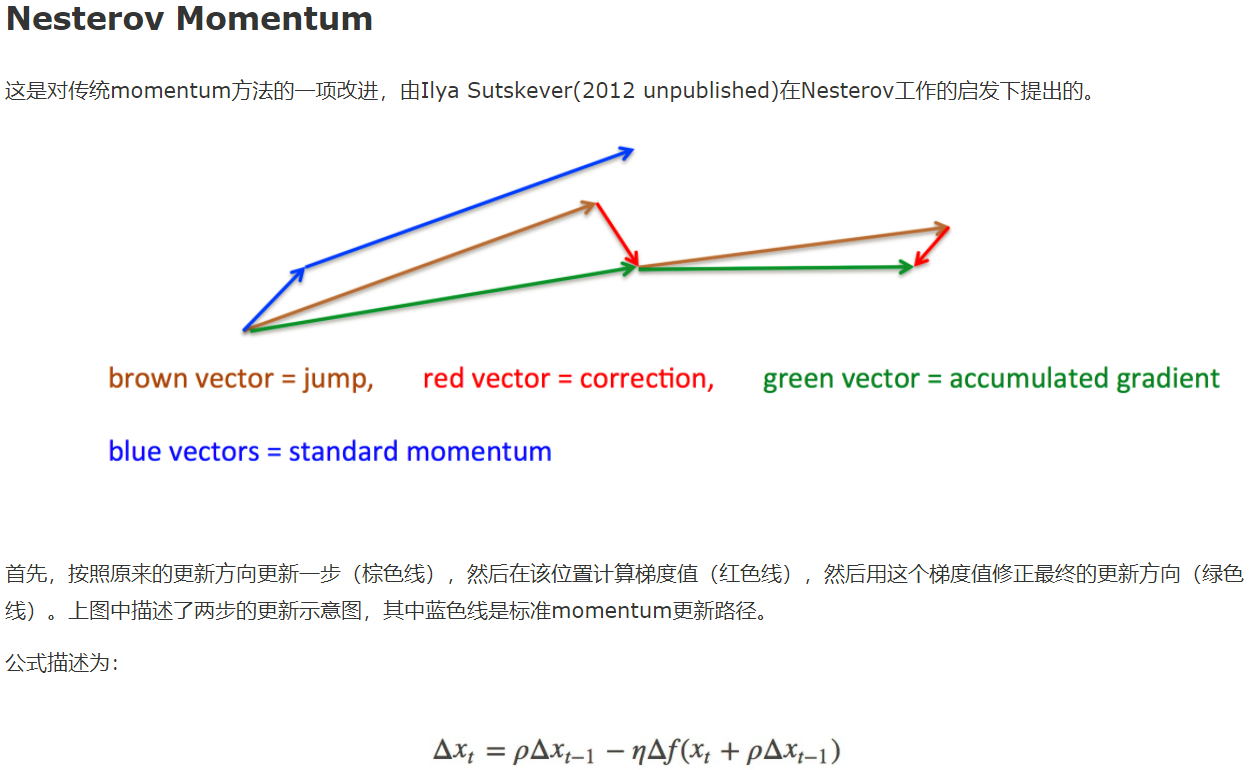
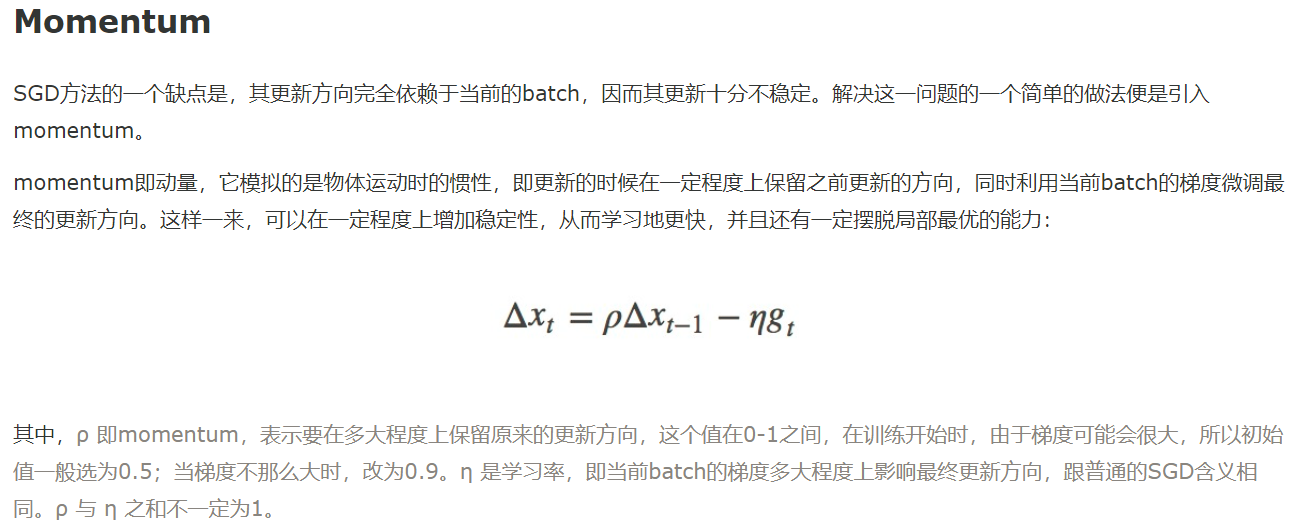
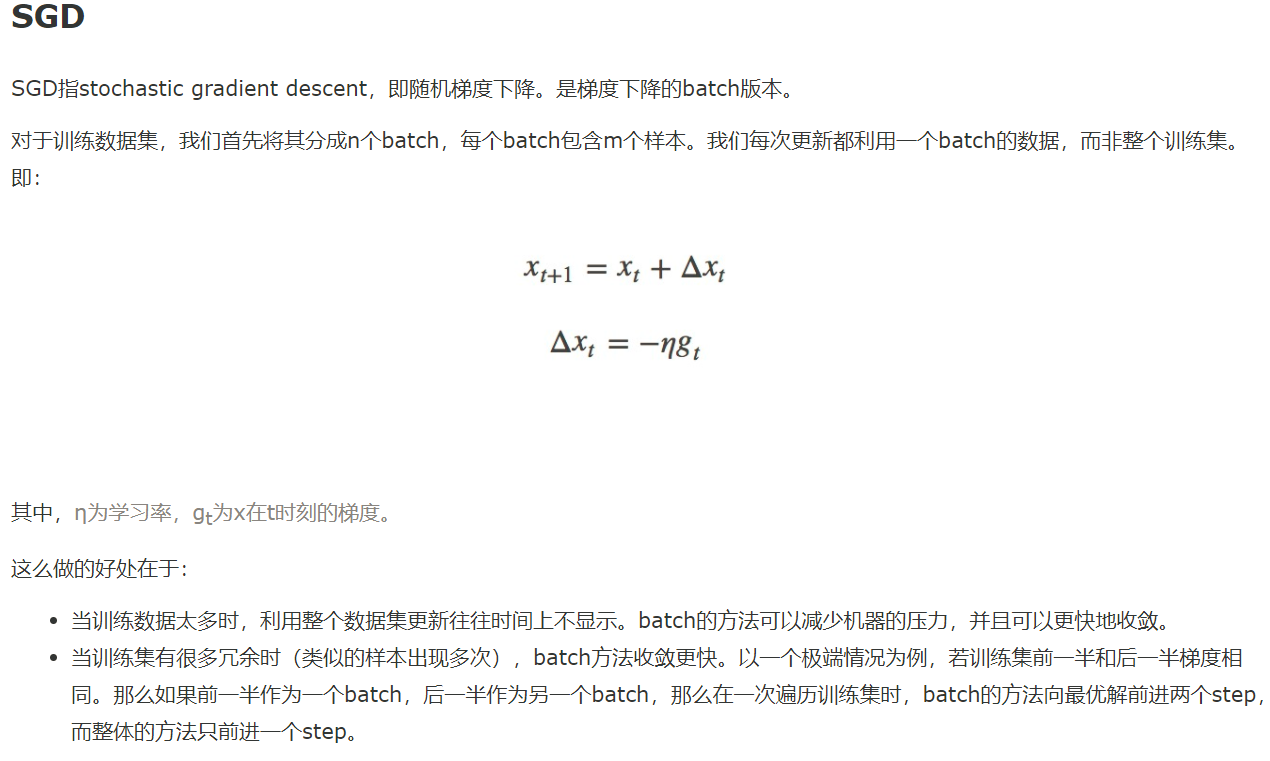
优点：batch的方法可以减少机器的压力，并且可以更快地收敛。

缺点：其更新方向完全依赖于当前的batch，因而其更新十分不稳定。

为了解决这个问题，momentum就横空出世了，对深度学习最全优化方法总结比较（SGD，Momentum，Nesterov Momentum，Adagrad，Adadelta，RMSprop，Adam）。

这里讨论的优化问题指的是，给定目标函数f(x)，我们需要找到一组参数x（权重），使得f(x)的值最小。

本文以下内容假设读者已经了解机器学习基本知识，和梯度下降的原理。



========================================

选用momentum：

https://img2018.cnblogs.com/blog/797382/201902/797382-20190225201006070-1516485048.png

momentum即动量，它模拟的是物体运动时的惯性，即更新的时候在一定程度上保留之前更新的方向，同时利用当前batch的梯度微调最终的更新方向。

这样一来，可以在一定程度上增加稳定性，从而学习地更快，并且还有一定摆脱局部最优的能力：

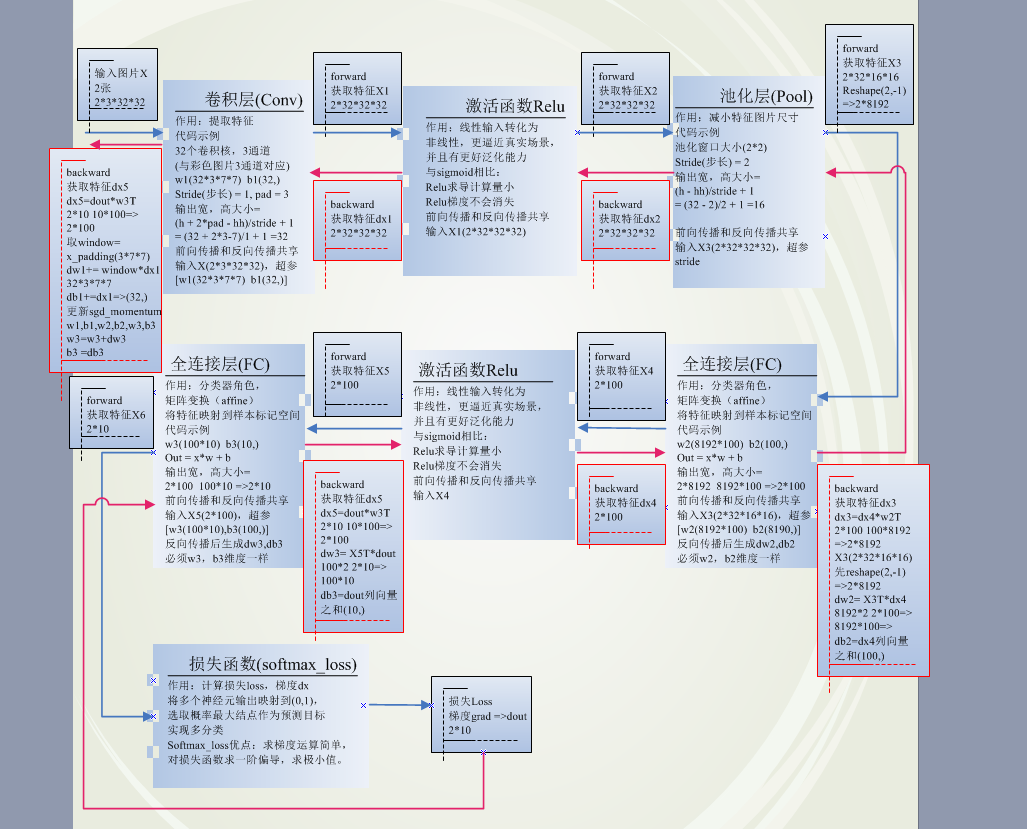
其中，ρ 即momentum，表示要在多大程度上保留原来的更新方向，这个值在0-1之间，在训练开始时，由于梯度可能会很大，所以初始值一般选为0.5；

当梯度不那么大时，改为0.9。η 是学习率，即当前batch的梯度多大程度上影响最终更新方向，跟普通的SGD含义相同。ρ 与 η 之和不一定为1。

**3.代码实现流程图以及介绍**

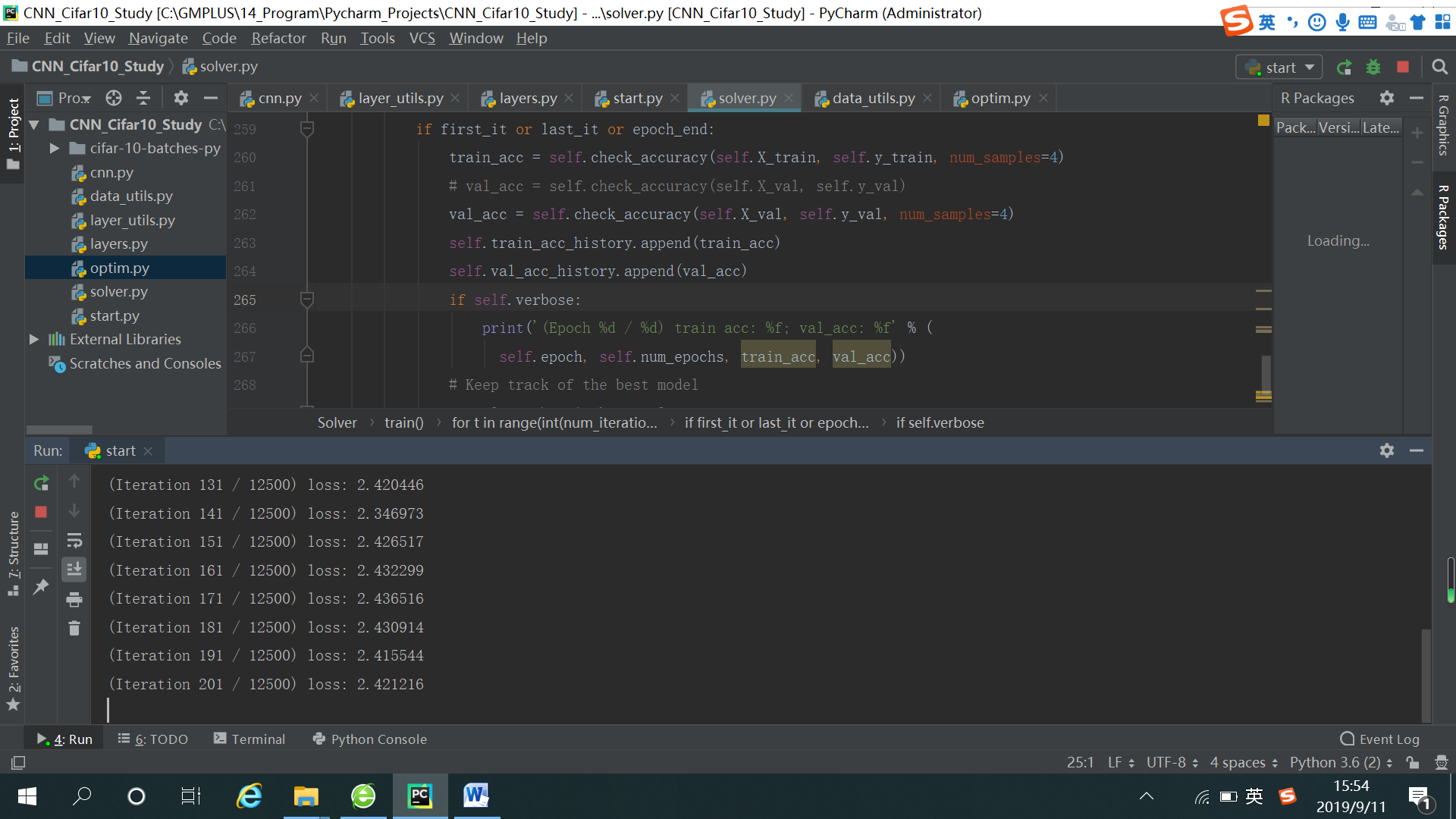
代码流程图：费了老大劲，终于弄完了，希望对各位看官们有所帮助，建议对比流程图和跟踪代码，加深对原理的理解。

特别是前向传播和反向传播维度的变换，需要重点关注。

**4.代码实现**

当然，代码的整个实现是某位大神实现的，我只是在上面做了些小改动以及重点函数做了些注释，有不妥之处也希望大家不吝指教。

因为原始图片数据集太大，不好上传，大家可以直接在[http://www.cs.toronto.edu/~kriz/cifar.html](http://www.cs.toronto.edu/~kriz/cifar.html" \t "_blank)下载CIFAR-10 python version，有163M，放在代码文件同路径下即可。



**5.注意事项**

网上下载源代码，直接运行会报错是因为它是python2，而不是python3。如果是python2版本是可以直接运行的。Python3需要做一下修改：

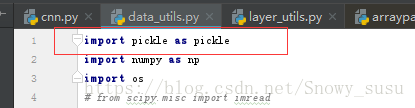
1. python2和3使用的默认编码是不同的，我们会发现pickle.load()函数报错。在代码开始部分（start.py）添加如图所示代码：



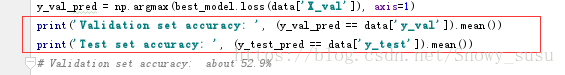
同时把这个函数(data\_utils.py)里面添加:

https://img-blog.csdn.net/20180725181716393?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1Nub3d5X3N1c3U=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

因为python3把cpickle改成pickle了，记得把import那里修改好



2 . python3的print输出得放在()里面



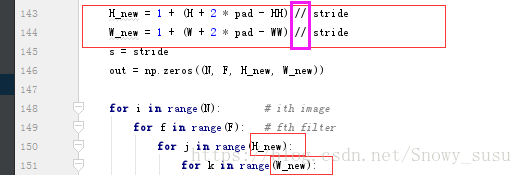
3 . Python3不支持xrange，改成range

https://img-blog.csdn.net/20180725181716384?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1Nub3d5X3N1c3U=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

range里面要求是整形变量：

（1）可以使用int强制转化

（2）可以将紫色框中原来的‘/’变化成”//”,python2的除法（/）结果是取整；python3的除法结果（/）是有小数的，可以使用双斜杠（//）来实现取整。



4 . python3不支持optim\_config.iteritems()改成.items，如图所示

https://img-blog.csdn.net/20180725181716741?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1Nub3d5X3N1c3U=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

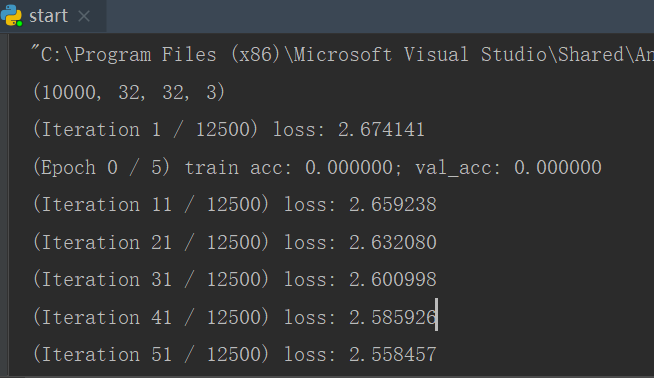
5 . 最后，记得修改你的数据集路径

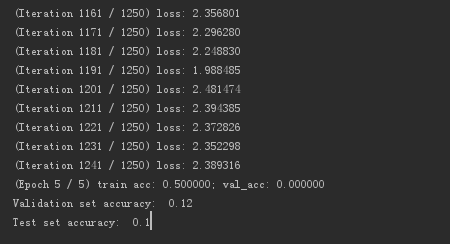
https://img-blog.csdn.net/20180725181716747?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1Nub3d5X3N1c3U=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70

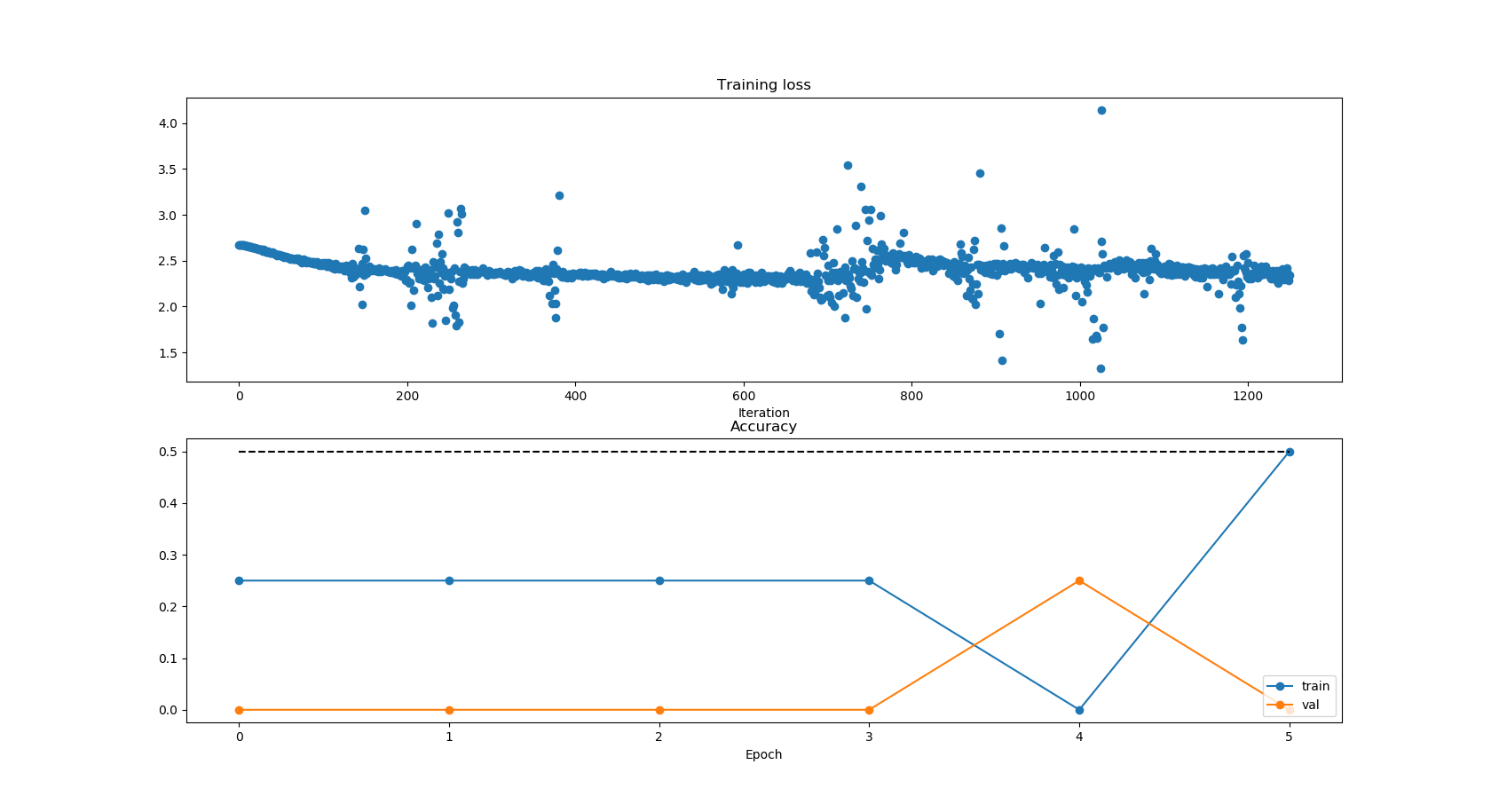
**6.运行结果以及分析**

这里选取500张图片作为训练样本，epoch = 5，batch = 2，每次随机选取2张图片，迭代 5 \* 500/2 = 1250次，测试样本选取50张。

由运行结果可以看出，损失loss是逐步下降的。







测试结果只有12%左右，原因有以下几点：

1. 模型比较简单，特征提取不能反映真实特征（一次卷积）；

2. 会出现过拟合问题；

3. 原始训练数据分类图片纹理复杂，这些图片可变性大，从而导致分类结果准确度低；

(airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck)

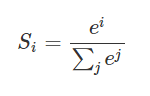
 后续会通过tensorflow来实现CNN，测试准确率可以达到71.95%。

**7. Softmax函数详解与推导**

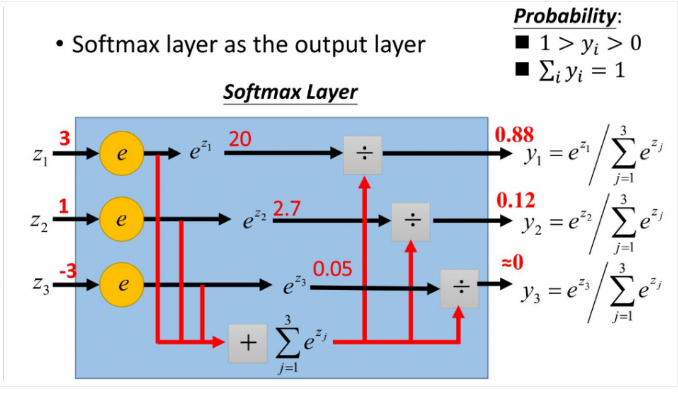
**7.1 softmax函数**

softmax用于多分类过程中，它将多个神经元的输出，映射到（0,1）区间内，可以看成概率来理解，从而来进行多分类！

假设我们有一个数组，V，Vi表示V中的第i个元素，那么这个元素的softmax值就是

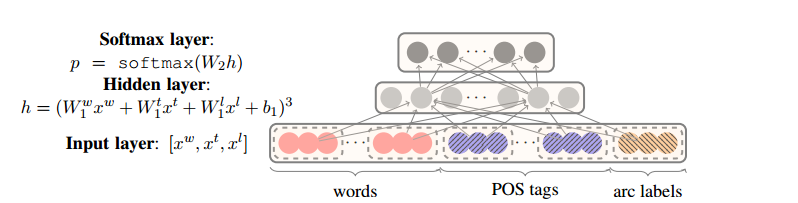


更形象的如下图表示：



softmax直白来说就是将原来输出是3,1,-3通过softmax函数一作用，就映射成为(0,1)的值，而这些值的累和为1（满足概率的性质），那么我们就可以将它理解成概率，在最后选取输出结点的时候，我们就可以选取概率最大（也就是值对应最大的）结点，作为我们的预测目标！

举一个我最近碰到利用softmax的例子：我现在要实现基于神经网络的句法分析器。用到是基于转移系统来做，那么神经网络的用途就是帮我预测我这一个状态将要进行的动作是什么？比如有10个输出神经元，那么就有10个动作，1动作，2动作，3动作...一直到10动作。（这里涉及到NLP的知识，大家不用管，只要知道我现在根据每个状态（输入），来预测动作（得到概率最大的输出），最终得到的一系列动作序列就可以完成我的任务即可），原理图如下图所示：



那么比如在一次的输出过程中输出结点的值是如下：

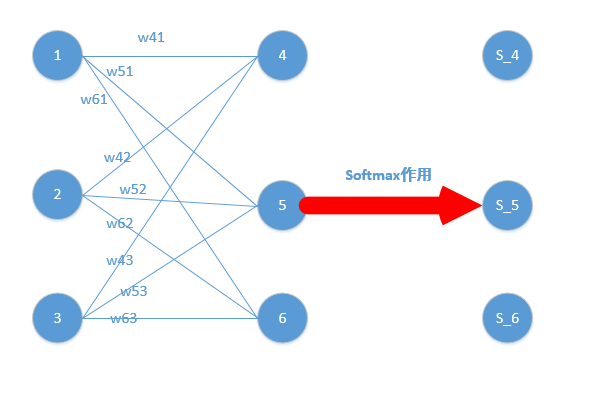
[0.2,0.1,0.05,0.1,0.2,0.02,0.08,0.01,0.01,0.23]

那么我们就知道这次我选取的动作是动作10，因为0.23是这次概率最大的，那么怎么理解多分类呢？很容易，如果你想选取俩个动作，那么就找概率最大的俩个值即可~（这里只是简单的告诉大家softmax在实际问题中一般怎么应用）

**7.2 softmax相关求导**

当我们对分类的Loss进行改进的时候，我们要通过梯度下降，每次优化一个step大小的梯度，这个时候我们就要求Loss对每个权重矩阵的偏导，然后应用链式法则。那么这个过程的第一步，就是对softmax求导传回去，不用着急，我后面会举例子非常详细的说明。在这个过程中，你会发现用了softmax函数之后，梯度求导过程非常非常方便！

下面我们举出一个简单例子，原理一样，目的是为了帮助大家容易理解！



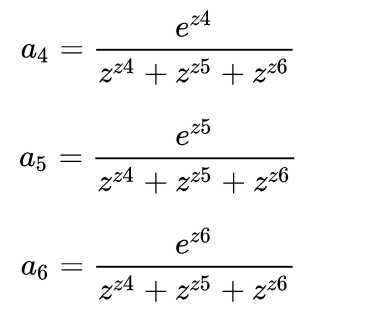
我们能得到下面公式：

z4 = w41\*o1+w42\*o2+w43\*o3

z5 = w51\*o1+w52\*o2+w53\*o3

z6 = w61\*o1+w62\*o2+w63\*o3

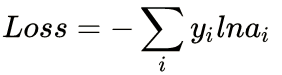
z4,z5,z6分别代表结点4,5,6的输出，01,02,03代表是结点1,2,3往后传的输入.那么我们可以经过softmax函数得到



好了，我们的重头戏来了，怎么根据求梯度，然后利用梯度下降方法更新梯度！

要使用梯度下降，肯定需要一个损失函数，这里我们使用交叉熵作为我们的损失函数，为什么使用交叉熵损失函数，不是这篇文章重点，后面有时间会单独写一下为什么要用到交叉熵函数（这里我们默认选取它作为损失函数）

交叉熵函数形式如下：



其中y代表我们的真实值，a代表我们softmax求出的值。i代表的是输出结点的标号！在上面例子，i就可以取值为4,5,6三个结点（当然我这里只是为了简单，真实应用中可能有很多结点）

现在看起来是不是感觉复杂了，居然还有累和，然后还要求导，每一个a都是softmax之后的形式！

但是实际上不是这样的，我们往往在真实中，如果只预测一个结果，那么在目标中只有一个结点的值为1，比如我认为在该状态下，我想要输出的是第四个动作（第四个结点）,那么训练数据的输出就是a4 = 1,a5=0,a6=0，哎呀，这太好了，除了一个为1，其它都是0，那么所谓的求和符合，就是一个幌子，我可以去掉啦！

为了形式化说明，我这里认为训练数据的真实输出为第j个为1，其它均为0！

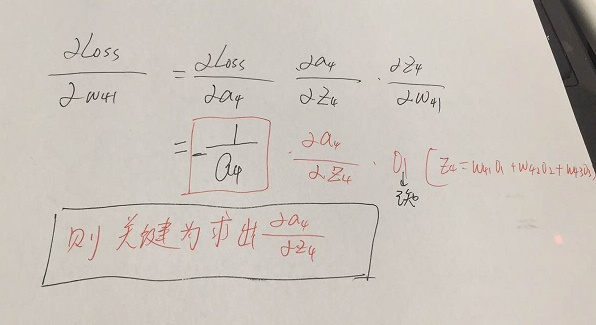
那么Loss就变成了 ,累和已经去掉了，太好了。现在我们要开始求导数了！

我们在整理一下上面公式，为了更加明白的看出相关变量的关系：

其中,那么形式变为

那么形式越来越简单了，求导分析如下：

参数的形式在该例子中，总共分为w41,w42,w43,w51,w52,w53,w61,w62,w63.这些，那么比如我要求出w41,w42,w43的偏导，就需要将Loss函数求偏导传到结点4，然后再利用链式法则继续求导即可，举个例子此时求w41的偏导为:

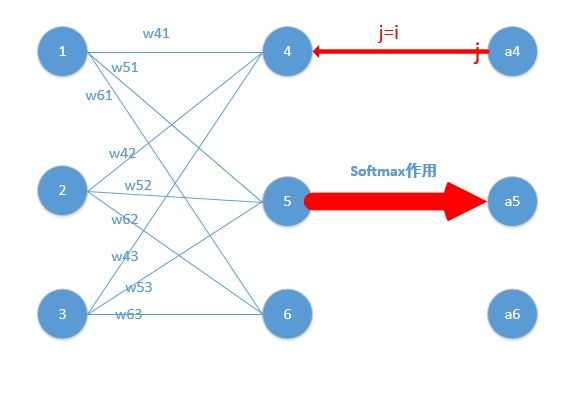


w51.....w63等参数的偏导同理可以求出，那么我们的关键就在于Loss函数对于结点4,5,6的偏导怎么求，如下：

这里分为俩种情况：

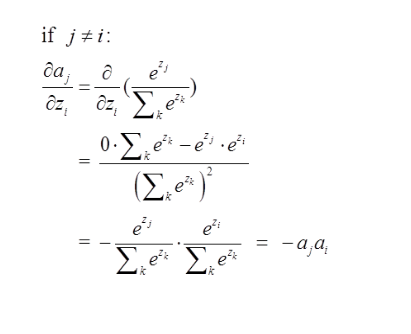


 j=i对应例子里就是如下图所示：比如我选定了j为4，那么就是说我现在求导传到4结点这！

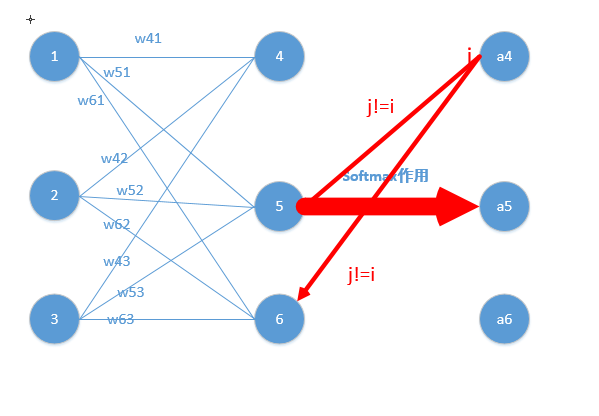


那么由上面求导结果再乘以交叉熵损失函数求导，它的导数为 ,与上面**相乘为**（形式非常简单，这说明我只要正向求一次得出结果，然后反向传梯度的时候，只需要将它结果减1即可，后面还会举例子！）那么我们可以得到Loss对于4结点的偏导就求出了了（这里假定4是我们的预计输出）

第二种情况为：



这里对应我的例子图如下，我这时对的是j不等于i，往前传：



那么由上面求导结果再乘以交叉熵损失函数求导，它的导数为,与上面**相乘为**（形式非常简单，这说明我只要正向求一次得出结果，然后反向传梯度的时候，只需要将它结果保存即可，后续例子会讲到）这里就求出了除4结点之外的其它所有结点的偏导，然后利用链式法则继续传递过去即可！我们的问题也就解决了！

**7.3 下面我举个例子来说明为什么计算会比较方便，给大家一个直观的理解**

举个例子，通过若干层的计算，最后得到的某个训练样本的向量的分数是[ 2, 3, 4 ],

那么经过softmax函数作用后概率分别就是= [0.0903,0.2447,0.665],

如果这个样本正确的分类是第二个的话，那么计算出来的偏导就是 [0.0903,0.2447-1,0.665]=[0.0903,-0.7553,0.665]，是不是非常简单！！然后再根据这个进行back propagation就可以了

到这里，这篇文章的内容就讲完了，我希望根据自己的理解，通过列出大量例子，直白的给大家讲解softmax的相关内容，让大家少走弯路，真心希望对大家的理解有帮助！欢迎交流指错！画图整理不易，觉得有帮助的给个赞呗，哈哈！

**8. BP算法**

反向传播算法（Backpropagation Algorithm，简称BP算法）是深度学习的重要思想基础，对于初学者来说也是必须要掌握的基础知识！本文希望以一个清晰的脉络和详细的说明，来让读者彻底明白BP算法的原理和计算过程。

全文分为上下两篇，上篇主要介绍BP算法的原理（即公式的推导），介绍完原理之后，我们会将一些具体的数据带入一个简单的三层神经网络中，去完整的体验一遍BP算法的计算过程；下篇是一个项目实战，我们将带着读者一起亲手实现一个BP神经网络（不适用任何第三方的深度学习框架）来解决一个具体的问题。

读者在学习的过程中，有任何的疑问，欢迎加入我们的交流群（扫描文章最后的二维码即可加入），和大家一起讨论！

1.BP算法的推导

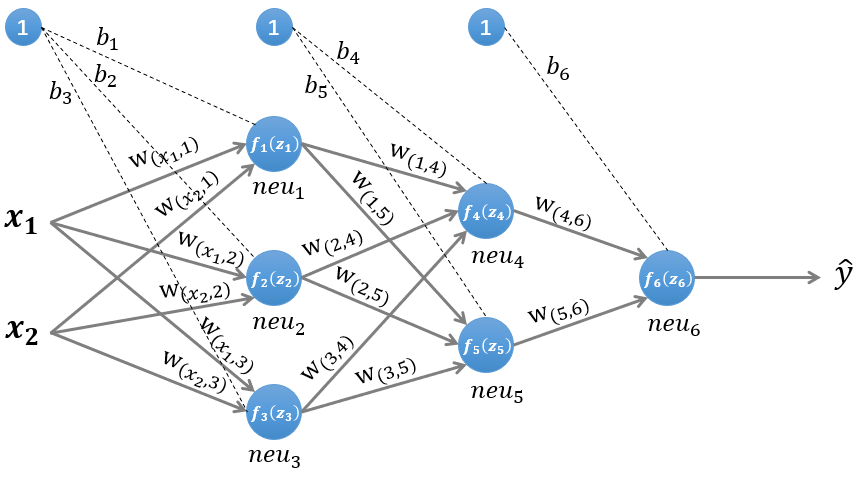


图1 一个简单的三层神经网络

图1所示是一个简单的三层（两个隐藏层，一个输出层）神经网络结构，假设我们使用这个神经网络来解决二分类问题，我们给这个网络一个输入样本 ，通过前向运算得到输出 。输出值 的值域为 ，例如 的值越接近0，代表该样本是“0”类的可能性越大，反之是“1”类的可能性大。

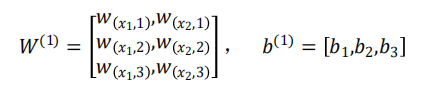
1.1前向传播的计算

为了便于理解后续的内容，我们需要先搞清楚前向传播的计算过程，以图1所示的内容为例：

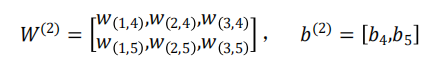
输入的样本为：

http://www.tensorflownews.com/wp-content/uploads/2018/03/%E5%9B%BE%E7%89%8728.png

第一层网络的参数为：



第二层网络的参数为：



第三层网络的参数为：

http://www.tensorflownews.com/wp-content/uploads/2018/03/%E5%9B%BE%E7%89%8731.png

1.1.1第一层隐藏层的计算

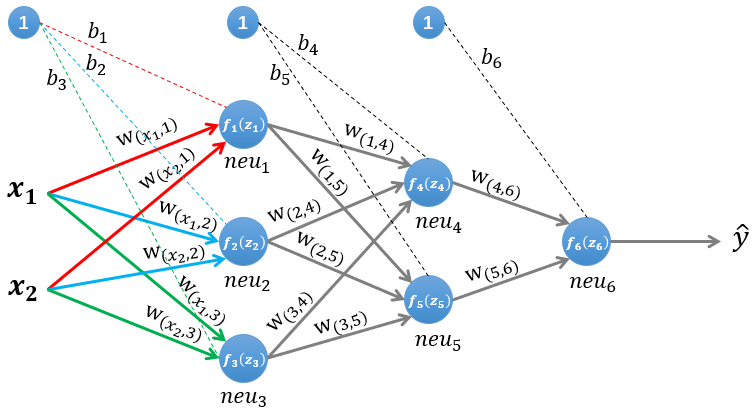
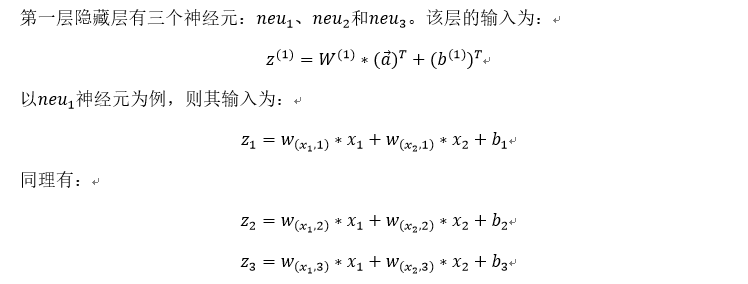
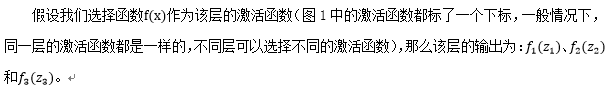


图2 计算第一层隐藏层





1.1.2第二层隐藏层的计算

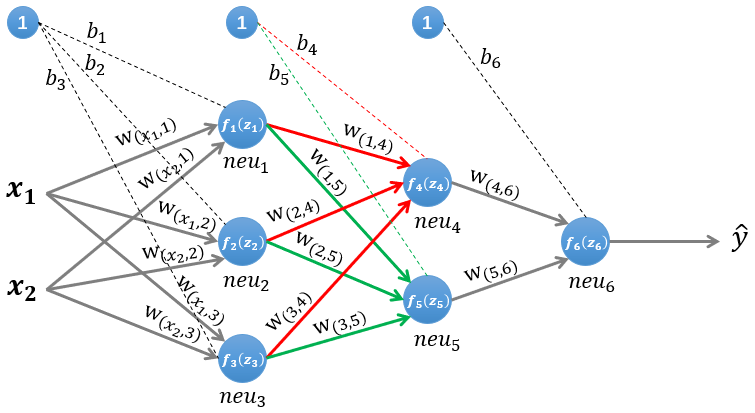
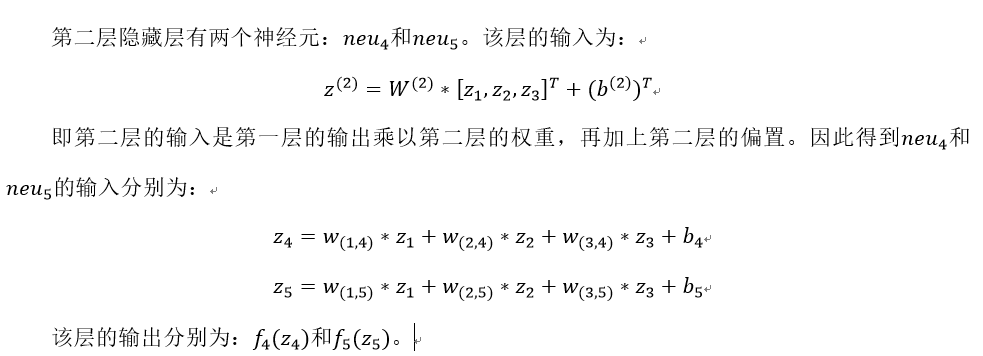


图3 计算第二层隐藏层



1.1.3输出层的计算

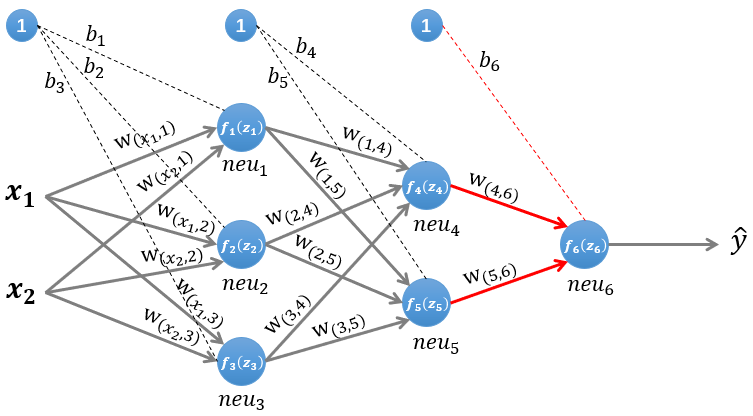


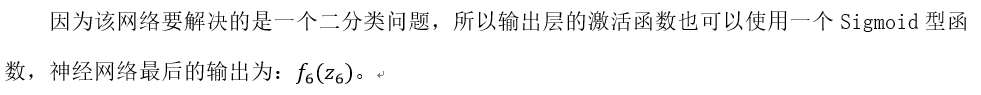
图4 计算输出层

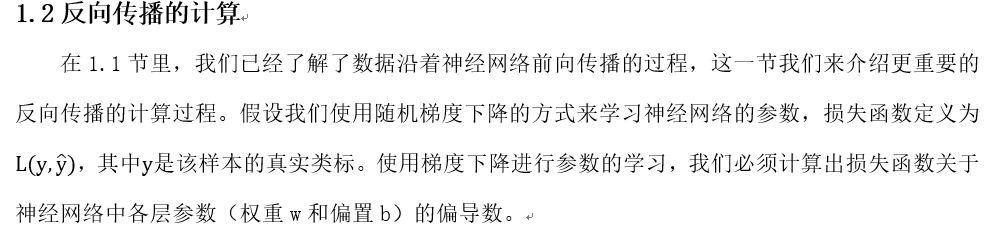
http://www.tensorflownews.com/wp-content/uploads/2018/03/5-5.png

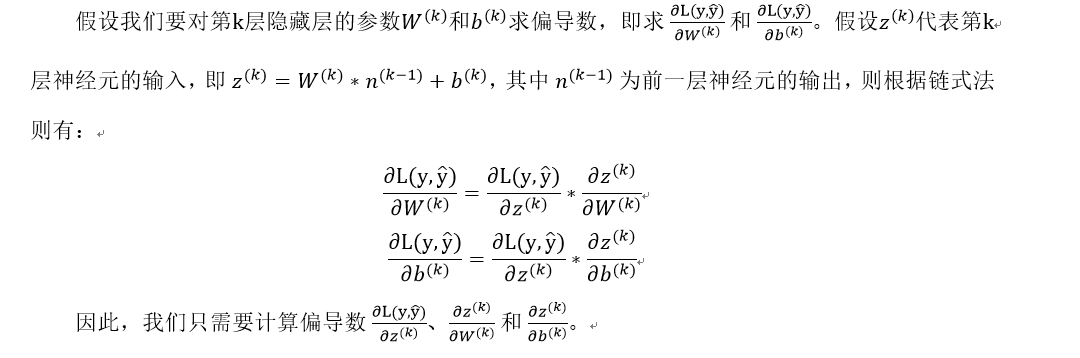
http://www.tensorflownews.com/wp-content/uploads/2018/03/%E5%9B%BE%E7%89%8738.png

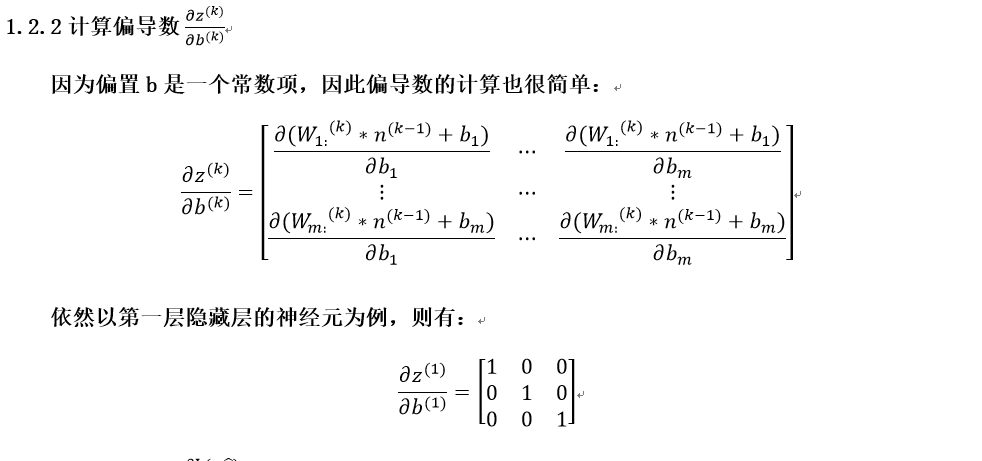
即：

http://www.tensorflownews.com/wp-content/uploads/2018/03/%E5%9B%BE%E7%89%8739.png

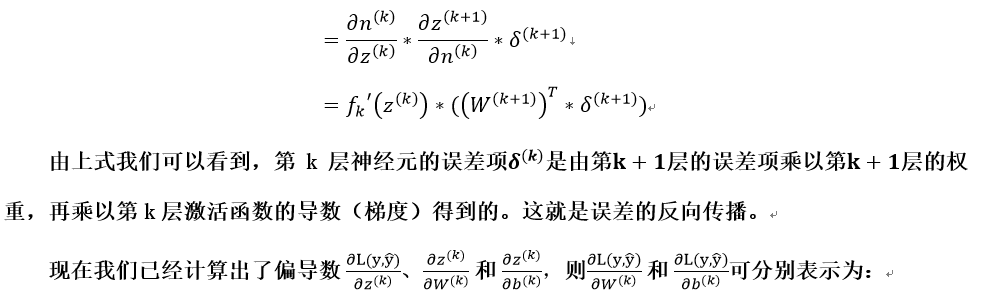


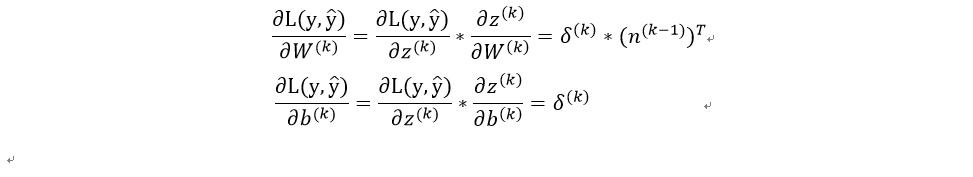










单纯的公式推导看起来有些枯燥，下面我们将实际的数据带入图1所示的神经网络中，完整的计算一遍。

2.图解BP算法

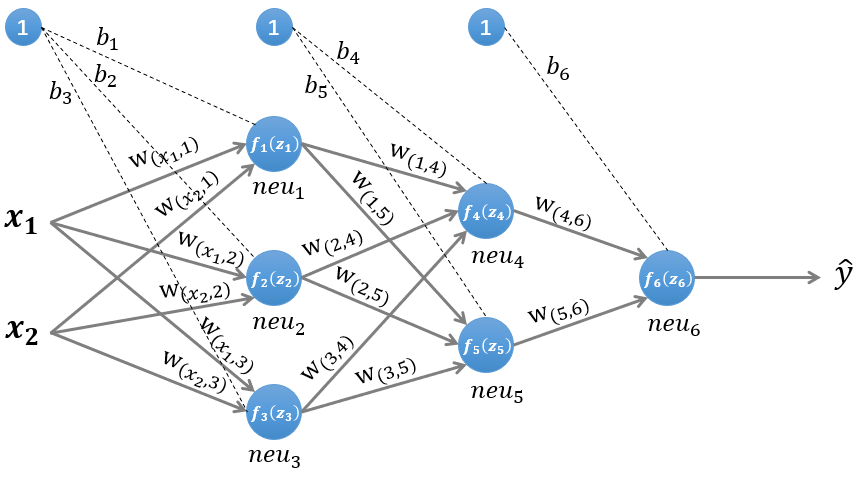


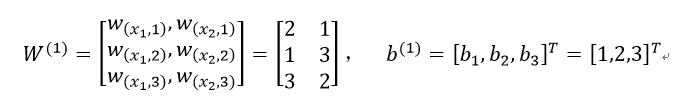
图5 图解BP算法

我们依然使用如图5所示的简单的神经网络，其中所有参数的初始值如下：

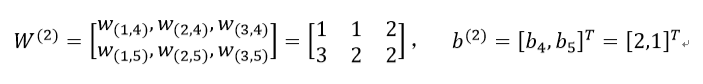
输入的样本为（假设其真实类标为“1”）：

http://www.tensorflownews.com/wp-content/uploads/2018/03/14.png

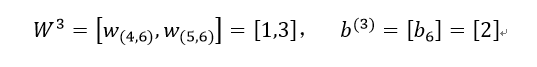
第一层网络的参数为：

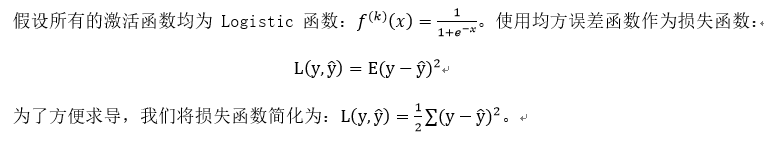


第二层网络的参数为：



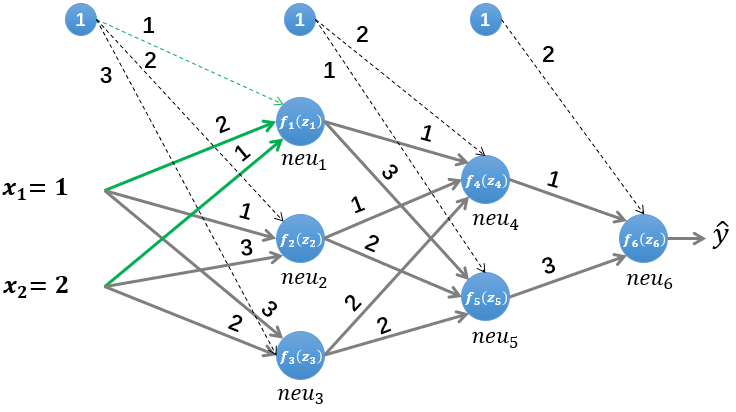
第三层网络的参数为：

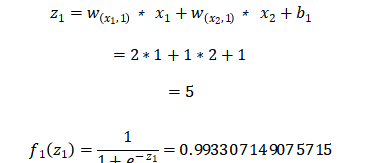


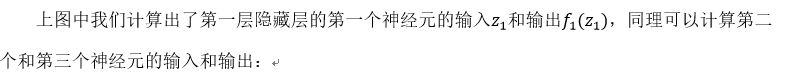


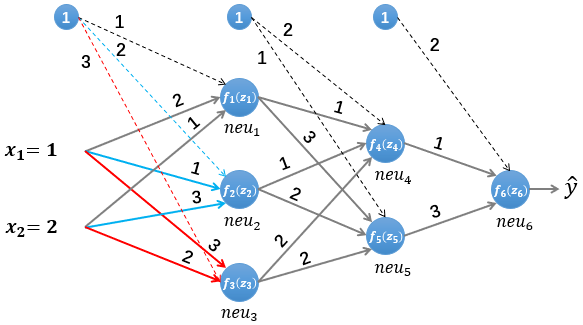
2.1前向传播

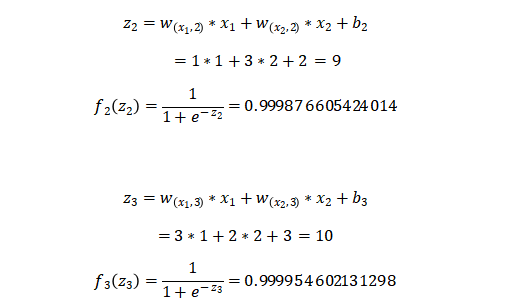
我们首先初始化神经网络的参数，计算第一层神经元：



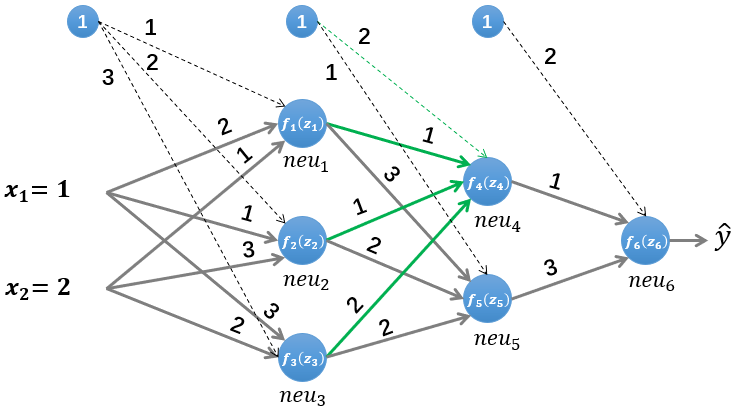


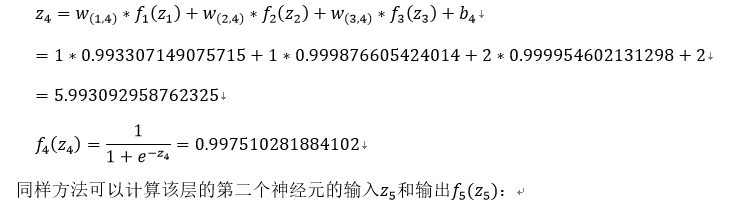


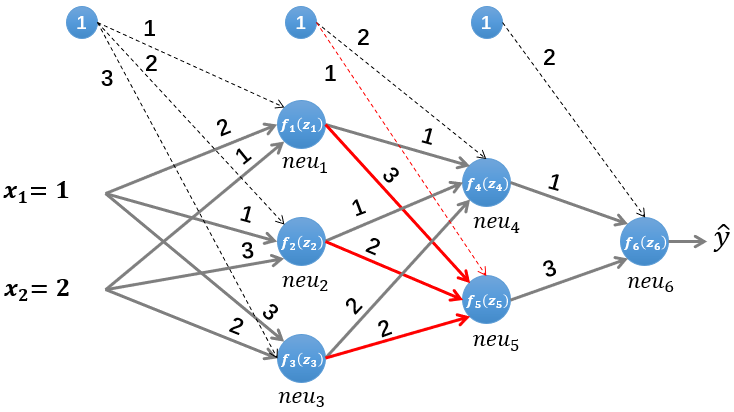


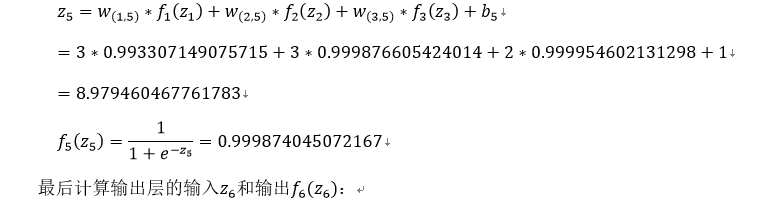


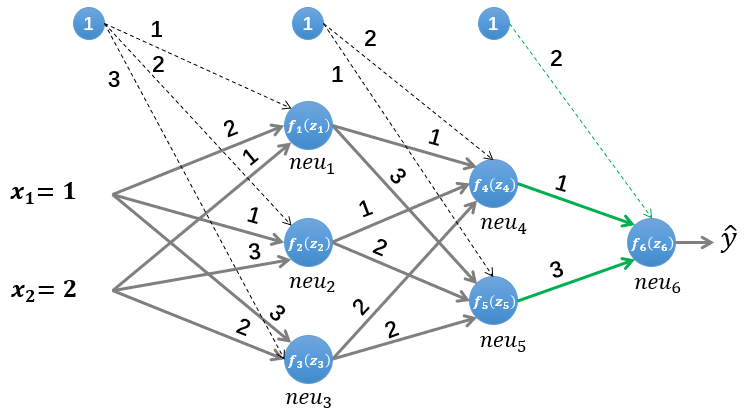
http://www.tensorflownews.com/wp-content/uploads/2018/03/21.png

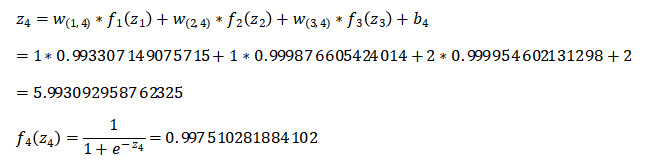




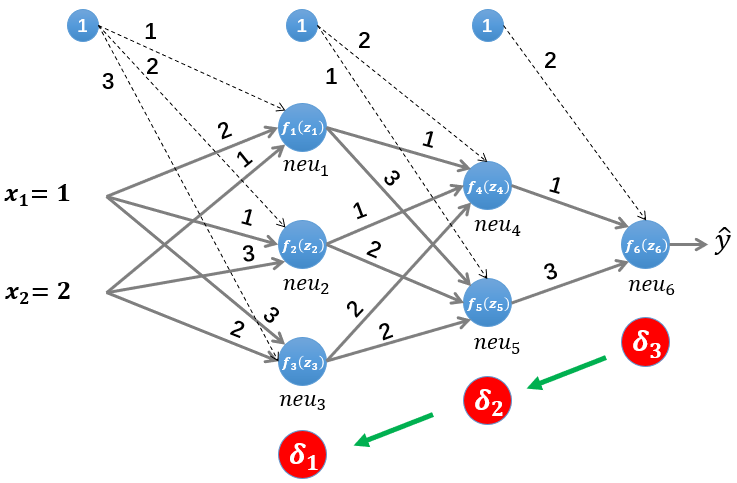


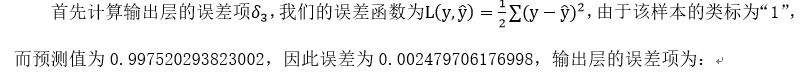


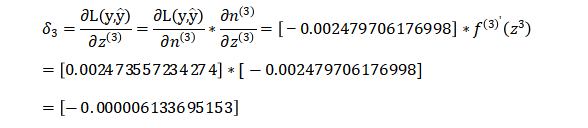




2.2误差反向传播



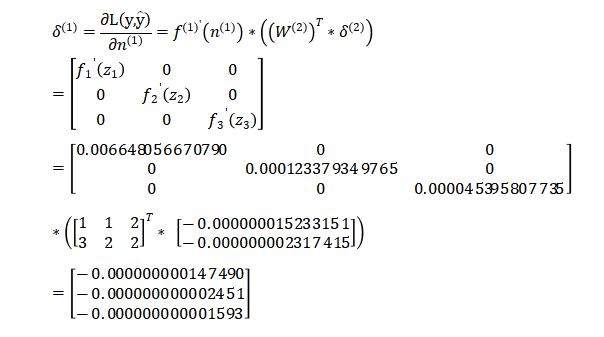




接着计算第二层隐藏层的误差项，根据误差项的计算公式有：

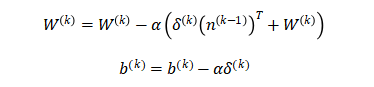


最后是计算第一层隐藏层的误差项：

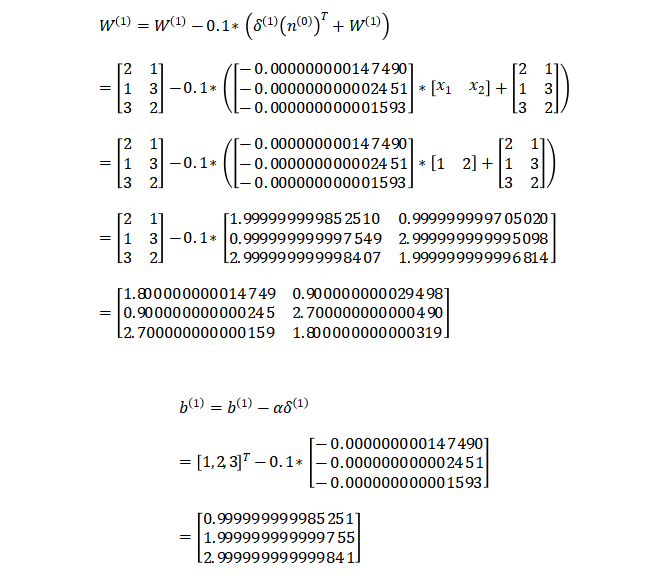


2.3更新参数

上一小节中我们已经计算出了每一层的误差项，现在我们要利用每一层的误差项和梯度来更新每一层的参数，权重W和偏置b的更新公式如下：



通常权重W的更新会加上一个正则化项来避免过拟合，这里为了简化计算，我们省去了正则化项。上式中的 是学习率，我们设其值为0.1。参数更新的计算相对简单，每一层的计算方式都相同，因此本文仅演示第一层隐藏层的参数更新：

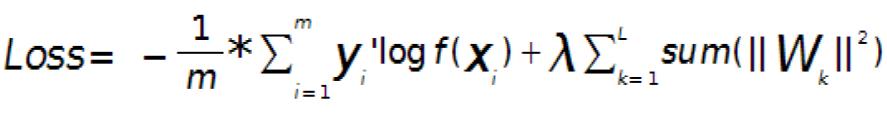


3.小结

至此，我们已经完整介绍了BP算法的原理，并使用具体的数值做了计算。在下篇中，我们将带着读者一起亲手实现一个BP神经网络（不适用任何第三方的深度学习框架），敬请期待！有任何疑问，欢迎加入我们一起交流！

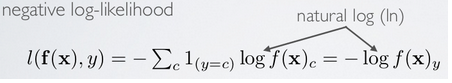
[**9.CNN的反向求导**](https://www.cnblogs.com/tornadomeet/p/3468450.html)

首先我们来看看CNN系统的目标函数，设有样本(xi, yi)共m个，CNN网络共有L层，中间包含若干个卷积层和pooling层，最后一层的输出为f(xi)，则系统的loss表达式为(对权值进行了惩罚，一般分类都采用交叉熵形式)：



**问题一：求输出层的误差敏感项。**

现在只考虑个一个输入样本(x, y)的情形，loss函数和上面的公式类似是用交叉熵来表示的，暂时不考虑权值规则项，样本标签采用one-hot编码，CNN网络的最后一层采用softmax全连接(多分类时输出层一般用softmax)，样本(x,y)经过CNN网络后的最终的输出用f(x)表示，则对应该样本的loss值为:

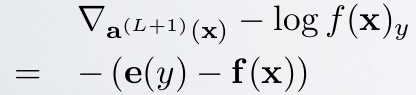


其中f(x)的下标c的含义见公式：

https://images0.cnblogs.com/blog/381513/201312/10232822-b40af6ecea5d4a7d9073a1e5bc5c018f.png

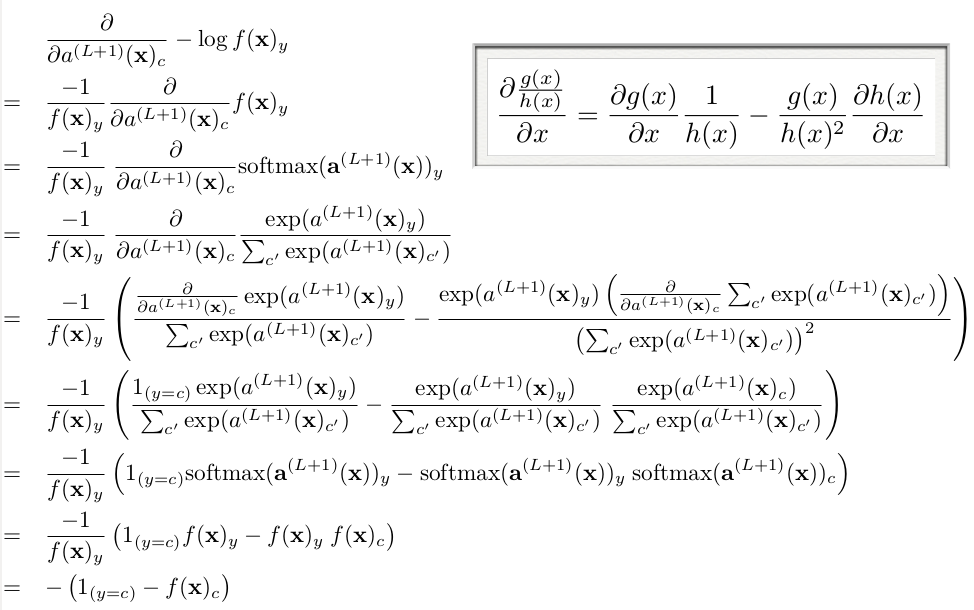
　　因为x通过CNN后得到的输出f(x)是一个向量，该向量的元素值都是概率值，分别代表着x被分到各个类中的概率，而f(x)中下标c的意思就是输出向量中取对应c那个类的概率值。

　　采用上面的符号，可以求得此时loss值对输出层的误差敏感性表达式为：



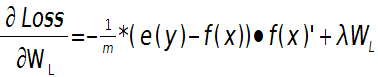
其中e(y)表示的是样本x标签值的one-hot表示，其中只有一个元素为1,其它都为0.

　　其推导过程如下（先求出对输出层某个节点c的误差敏感性，参考[Larochelle关于DL的课件:](file:///\\\\https:\\dl.dropboxusercontent.com\\u\\19557502\\2_03_output_layer_gradient.pdf)[Output layer gradient](file:///\\https:\dl.dropboxusercontent.com\u\19557502\2_03_output_layer_gradient.pdf)）,求出输出层中c节点的误差敏感值：

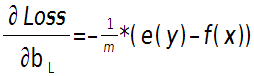


　　由上面公式可知，如果输出层采用softmax，且loss用交叉熵形式，则最后一层的误差敏感值就等于CNN网络输出值f(x)减样本标签值e(y),即f(x)-e(y),其形式非常简单，这个公式是不是很眼熟？很多情况下如果model采用MSE的loss，即loss=1/2\*(e(y)-f(x))^2，那么loss对最终的输出f(x)求导时其结果就是f(x)-e(y),虽然和上面的结果一样，但是大家不要搞混淆了，这2个含义是不同的，一个是对输出层节点输入值的导数(softmax激发函数)，一个是对输出层节点输出值的导数(任意激发函数）。而在使用MSE的loss表达式时，输出层的误差敏感项为(f(x)-e(y)).\*f(x)’，两者只相差一个因子。

　　这样就可以求出第L层的权值W的偏导数：



　　输出层偏置的偏导数：



　　上面2个公式的e(y)和f(x)是一个矩阵，已经把所有m个训练样本考虑进去了,每一列代表一个样本。

**问题二：当接在卷积层的下一层为pooling层时，求卷积层的误差敏感项。**

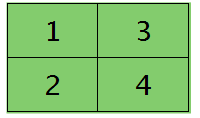
　　假设第l(小写的l，不要看成数字’1’了)层为卷积层，第l+1层为pooling层，且pooling层的误差敏感项为： https://images0.cnblogs.com/blog/381513/201312/10233031-f52820fe834e42dc96a4cc7a5b58705f.png  ,卷积层的误差敏感项为：https://images0.cnblogs.com/blog/381513/201312/10233042-6242eed2c71641c19113ca8f1046ddd5.png  , 则两者的关系表达式为：

https://images0.cnblogs.com/blog/381513/201312/11080853-06c3fdc33e66423490d7560de2cbbeb9.png

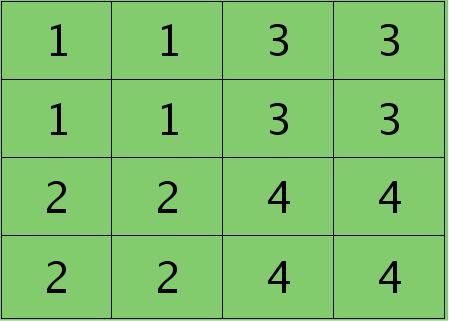
　　这里符号●表示的是矩阵的点积操作，即对应元素的乘积。卷积层和unsample()后的pooling层节点是一一对应的，所以下标都是用j表示。后面的符号https://images0.cnblogs.com/blog/381513/201312/11081018-e68028bc98334a07bc495255350769e6.png表示的是第l层第j个节点处激发函数的导数(对节点输入的导数)。

　　其中的函数unsample()为上采样过程，其具体的操作得看是采用的什么pooling方法了。但unsample的大概思想为：pooling层的每个节点是由卷积层中多个节点(一般为一个矩形区域)共同计算得到，所以pooling层每个节点的误差敏感值也是由卷积层中多个节点的误差敏感值共同产生的，只需满足两层见各自的误差敏感值相等，下面以mean-pooling和max-pooling为例来说明。

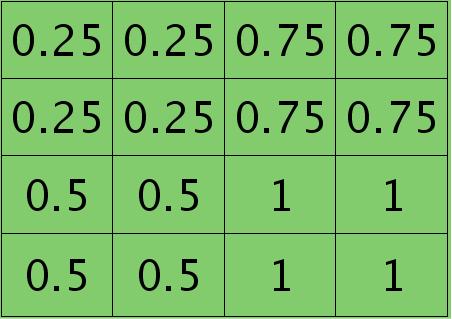
　　假设卷积层的矩形大小为4×4, pooling区域大小为2×2, 很容易知道pooling后得到的矩形大小也为2\*2（本文默认pooling过程是没有重叠的，卷积过程是每次移动一个像素，即是有重叠的，后续不再声明）,如果此时pooling后的矩形误差敏感值如下：



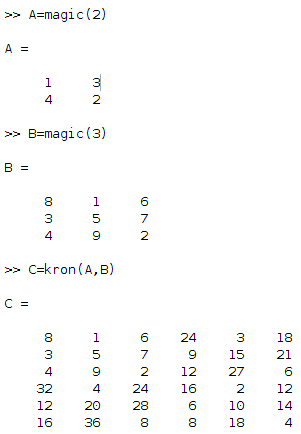
　　则按照mean-pooling，首先得到的卷积层应该是4×4大小，其值分布为(等值复制)：



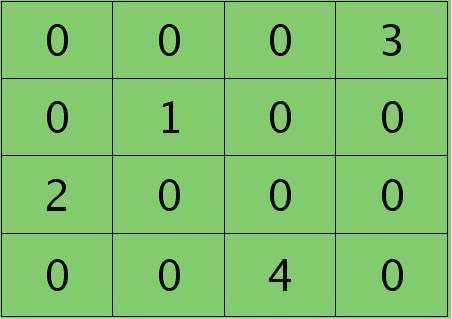
　　因为得满足反向传播时各层间误差敏感总和不变，所以卷积层对应每个值需要平摊（除以pooling区域大小即可，这里pooling层大小为2×2=4)），最后的卷积层值分布为：



　　mean-pooling时的unsample操作可以使用matlab中的函数kron()来实现，因为是采用的矩阵Kronecker乘积。C=kron(A, B)表示的是矩阵B分别与矩阵A中每个元素相乘，然后将相乘的结果放在C中对应的位置。比如：



　　如果是max-pooling，则需要记录前向传播过程中pooling区域中最大值的位置，这里假设pooling层值1,3,2,4对应的pooling区域位置分别为右下、右上、左上、左下。则此时对应卷积层误差敏感值分布为：



　　当然了，上面2种结果还需要点乘卷积层激发函数对应位置的导数值了，这里省略掉。

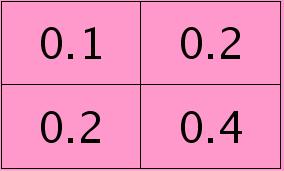
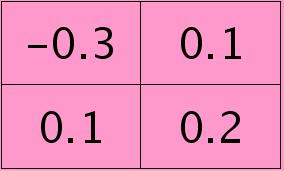
**问题三：当接在pooling层的下一层为卷积层时，求该pooling层的误差敏感项。**

　　假设第l层(pooling层)有N个通道，即有N张特征图，第l+1层(卷积层)有M个特征，l层中每个通道图都对应有自己的误差敏感值，其计算依据为第l+1层所有特征核的贡献之和。下面是第l+1层中第j个核对第l层第i个通道的误差敏感值计算方法：

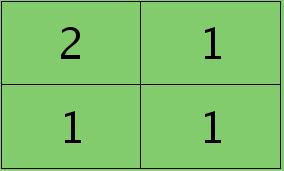
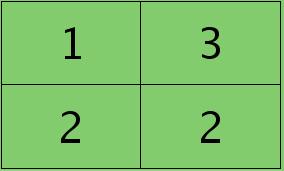


　　符号★表示的是矩阵的卷积操作，这是真正意义上的离散卷积，不同于卷积层前向传播时的相关操作，因为严格意义上来讲，卷积神经网络中的卷积操作本质是一个相关操作，并不是卷积操作，只不过它可以用卷积的方法去实现才这样叫。而求第i个通道的误差敏感项时需要将l+1层的所有核都计算一遍，然后求和。另外因为这里默认pooling层是线性激发函数，所以后面没有乘相应节点的导数。

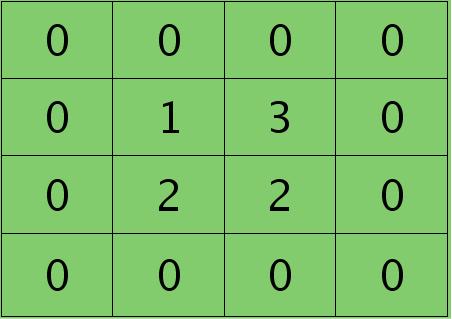
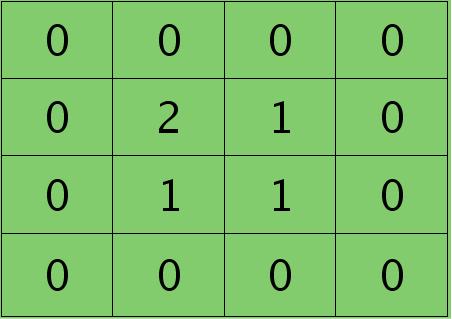
　　举个简单的例子，假设拿出第l层某个通道图，大小为3×3，第l+1层有2个特征核，核大小为2×2，则在前向传播卷积时第l+1层会有2个大小为2×2的卷积图。如果2个特征核分别为：

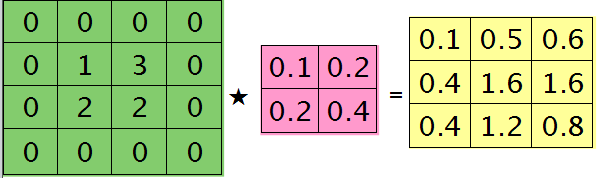
　　反向传播求误差敏感项时，假设已经知道第l+1层2个卷积图的误差敏感值：

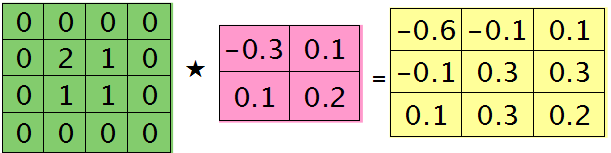


　　离散卷积函数conv2()的实现相关子操作时需先将核旋转180度(即左右翻转后上下翻转)，但这里实现的是严格意义上的卷积，所以在用conv2()时，对应的参数核不需要翻转（在有些toolbox里面，求这个问题时用了旋转，那是因为它们已经把所有的卷积核都旋转过，这样在前向传播时的相关操作就不用旋转了。并不矛盾）。且这时候该函数需要采用’full’模式，所以最终得到的矩阵大小为3×3,（其中3=2+2-1）,刚好符第l层通道图的大小。采用’full’模式需先将第l+1层2个卷积图扩充，周围填0,padding后如下：

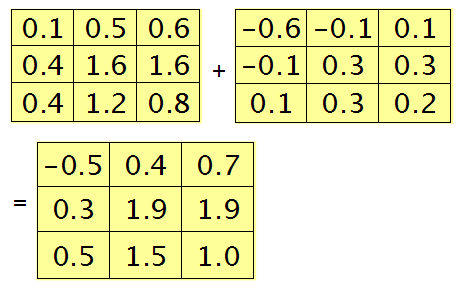
     

　　扩充后的矩阵和对应的核进行卷积的结果如下情况：





　　可以通过手动去验证上面的结果，因为是离散卷积操作，而离散卷积等价于将核旋转后再进行相关操作。而第l层那个通道的误差敏感项为上面2者的和，呼应问题三，最终答案为：

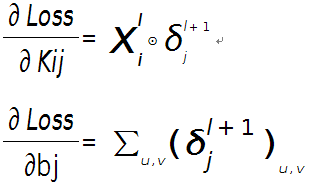


　　那么这样问题3这样解的依据是什么呢？其实很简单，本质上还是bp算法，即第l层的误差敏感值等于第l+1层的误差敏感值乘以两者之间的权值，只不过这里由于是用了卷积，且是有重叠的，l层中某个点会对l+1层中的多个点有影响。比如说最终的结果矩阵中最中间那个0.3是怎么来的呢？在用2×2的核对3×3的输入矩阵进行卷积时，一共进行了4次移动，而3×3矩阵最中间那个值在4次移动中均对输出结果有影响，且4次的影响分别在右下角、左下角、右上角、左上角。所以它的值为2×0.2+1×0.1+1×0.1-1×0.3=0.3, 建议大家用笔去算一下，那样就可以明白为什么这里可以采用带’full’类型的conv2()实现。

**问题四：求与卷积层相连那层的权值、偏置值导数。**

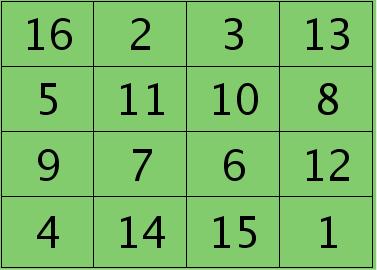
　　前面3个问题分别求得了输出层的误差敏感值、从pooling层推断出卷积层的误差敏感值、从卷积层推断出pooling层的误差敏感值。下面需要利用这些误差敏感值模型中参数的导数。这里没有考虑pooling层的非线性激发，因此pooling层前面是没有权值的，也就没有所谓的权值的导数了。现在将主要精力放在卷积层前面权值的求导上(也就是问题四)。

　　假设现在需要求第l层的第i个通道，与第l+1层的第j个通道之间的权值和偏置的导数，则计算公式如下：

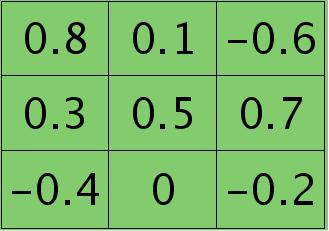


　　其中符号⊙表示矩阵的相关操作，可以采用conv2()函数实现。在使用该函数时，需将第l+1层第j个误差敏感值翻转。

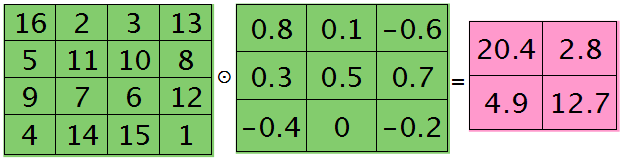
　　例如，如果第l层某个通道矩阵i大小为4×4,如下：



　　第l+1层第j个特征的误差敏感值矩阵大小为3×3,如下：



　　很明显，这时候的特征Kij导数的大小为2×2的，且其结果为：



　　而此时偏置值bj的导数为1.2 ，将j区域的误差敏感值相加即可(0.8+0.1-0.6+0.3+0.5+0.7-0.4-0.2=1.2)，因为b对j中的每个节点都有贡献，按照多项式的求导规则(和的导数等于导数的和)很容易得到。

　　为什么采用矩阵的相关操作就可以实现这个功能呢？由bp算法可知，l层i和l+1层j之间的权值等于l+1层j处误差敏感值乘以l层i处的输入，而j中某个节点因为是由i+1中一个区域与权值卷积后所得，所以j处该节点的误差敏感值对权值中所有元素都有贡献，由此可见，将j中每个元素对权值的贡献(尺寸和核大小相同)相加，就得到了权值的偏导数了(这个例子的结果是由9个2×2大小的矩阵之和)，同样，如果大家动笔去推算一下，就会明白这时候为什么可以用带’valid’的conv2()完成此功能。

**总结：**

（1）卷积层过后，可以先跟pooling层，再通过非线性传播函数。也可以是先通过非线性传播函数再经过pooling层。

（2）CNN的结构本身就是一种规则项。

（3）实际上每个权值的学习率不同时优化会更好。

（4）发现Ng以前的ufldl中教程里面softmax并没有包含偏置值参数，至少他给的start code里面没有包含，严格来说是错误的。

（5）当输入样本为多个时，bp算法中的误差敏感性也是一个矩阵。每一个样本都对应有自己每层的误差敏感性。

（6）血的教训啊，以后循环变量名不能与终止名太相似，一不小心引用下标是就弄错，比如for filterNum = 1:numFilters 时一不小心把下标用numFilters去代替了，花了大半天去查这个debug.

（7）matlab中conv2()函数在卷积过程中默认是每次移动一个像素，即重叠度最高的卷积。

**10. 参考文献**

视觉一只白的博客《常用损失函数小结》<https://blog.csdn.net/zhangjunp3/article/details/80467350>

理想万岁的博客《Softmax函数详解与推导》：[http://www.cnblogs.com/zongfa/p/8971213.html](http://www.cnblogs.com/zongfa/p/8971213.html" \t "_blank)

下路派出所的博客《深度学习（九） 深度学习最全优化方法总结比较（SGD，Momentum，Nesterov Momentum，Adagrad，Adadelta，RMSprop，Adam）》

<http://www.cnblogs.com/callyblog/p/8299074.html>