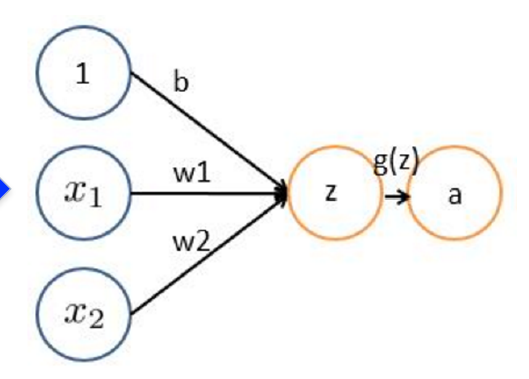
# 神经网络

## 感知器

感知器（Perceptron），1950s由Rosenblatt第一次引入,所以也叫rosenblatt感知器。

感知器是神经网络中最基础的单元，一个最简单的感知器的结构如下图：

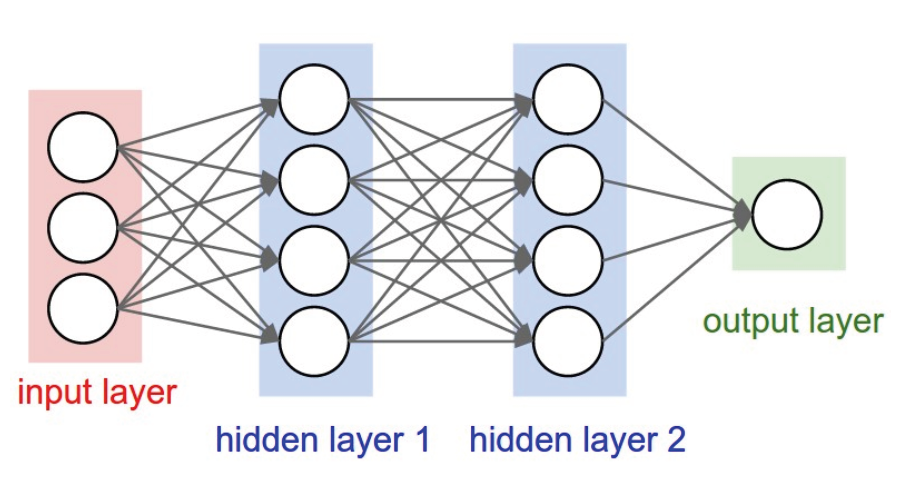




x是输入特征，w是权重，g是激活函数。如果g是sigmoid函数，这就是一个逻辑回归。

## 多隐层神经网络

多隐层神经网络的结构大概如下：



神经网络由输入层、隐藏层、输出层构成。隐藏层上的一个节点是一个神经元。一个神经元就是一个感知器。

根据隐藏层的层数又分为：**无隐层神经网络、单隐层神经网络、多隐层神经网络**。增加隐藏层层数就是**深度神经网络**(DNN)。无隐层的神经网络也就是一个感知器。

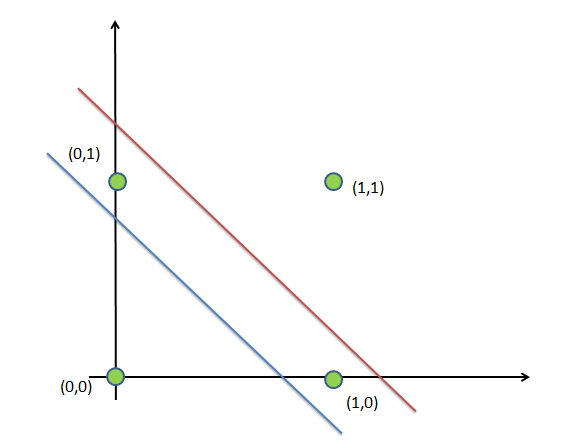
## 神经网络的能力

**神经网络能做什么？**

首先，我们看下一个神经元能做什么。如果神经元的激活函数采用sigmoid函数，那么，一个神经元就是一个逻辑回归。逻辑回归是广义线性模型，本质还是线性组合，它能做的事情只是线性分类。（线性分类在二维平面上是画一条线，在三维是一个平面，在三维以上是一个超平面）。**所以一个神经元就能做线性分类**。多层神经网络一个隐藏层有多个神经元，又有多隐藏层，它其实就是一个复杂的集成算法。理论上来说，能做非常复杂的事情。

一个神经元就能实现“逻辑与”和“逻辑或”。

4种情况: (0,0),(0,1),(1,0,),(1,1) , 分别对应二维平面上4个点。如下图。红色的线实现了“逻辑与”，蓝色的线实现了“逻辑或”



* 理论上说单隐层神经网络可以逼近任何连续函数（只要隐层的神经元个数足够多）。
* 虽然从数学上看表达能力一致，但是多隐藏层的神经网络比单隐藏层的神经网络工程效果好很多。
* 对于一些分类问题，比如CTR预估，3层神经网络效果优于2层神经网络，但是如果把层数再不断增加(4,5,6层)，对最后结果的帮助就没有那么大的跳变了。
* 图像数据比较特殊，是一种深层(多层次)的结构化数据，深层次的卷积神经网络，能够更充分和准确地把这些层级信息表达出来。

## SGD权重更新

对于神经网络这个结构，他的参数就是权重w，神经网络的目标就是要得到最优w.

以一个感知器的权重更新举例：

设输入是x向量，权重是w向量，线性组合函数，激活函数f，损失函数h.则



损失函数最小化，也就是对w求导。复合函数求导，根据链式法则。



如果f是sigmoid函数，h是最小均方，。d是实际值，y是预测值。

则：







权重更新规则，采用梯度下降：





\*eta是学习率，是步长。

实际运用中采用的是随机梯度下降SGD。

## BP算法

BP算法又叫delta算法。

如果是一个简单的感知器，误差是很容易求得的。而多层神经网络后，如何估计隐藏层的误差成为最大的难题。1985年，人们提出了BP算法解决了这一难题。

**核心原理**：“前向传播”求损失，“反向传播”回传误差。

1. **前向传播求损失**

顾名思义，它完成逐层从输入层传播至输出层的任务。

设神经元i是神经元j的上层节点，vi是神经元i的输出，vj是神经元j的输入。

神经元j的输入是上层所有神经元输出的线性组合，相当于z。定义如下：



神经元j的输入再用激活函数处理，输出yj。



前向传播求损失很简单。

1. **反向传播”回传误差，进行权重更新**

输出层的误差很容易求得。输出层的权重采用梯度下降也很好更新。

隐藏层的误差相对复杂些。是一个嵌套的复合函数。

BP采用delta学习规则



 （ 读delta）

是神经元i连接到神经元j的权值的修正值，eta是学习率，delta是局部梯度,为神经元j的损失函数对它的输入的偏导，vi为神经元i的输出。

**怎么求？BP采用delta学习规则**

1）神经元j在输出层，则



若，则：



y表示神经元的输出，d表示输出层的输出

2）神经元j在隐藏层，则



其中表示第n层的神经元j的delta

**推导：**

**误差E计算**

举例：一个双隐藏层的神经网络。w共3层，2个隐藏层+1个输出层。

\*w的层数比神经网络的层数少一层，比隐藏层的层数多一层。

**1）输出层的误差**可以是：



用展开是：



对w求导：



对神经元k的输入求导则是：



**2）误差展开到上一层隐藏层**，也就是用展开：



对w求导,链式法则：



对神经元j的输入求导：



**3）以此类推**。

**delta法则的优势**：

如果每层的w，都通过e求导，则是十分冗余的，因为很多路径被重复访问了。对于权值动则数万的深度模型中的神经网络，这样的冗余所导致的计算量是相当大的。

同样是利用链式法则，BP算法则机智地避开了这种冗余，它对于每一个路径只访问一次就能求顶点对所有下层节点的偏导值。

**BP的使用**：

1. 建立一个固定的网络结构，主要是隐藏层的层数，每层的节点数
2. 初始化所有网络的权值为一个随机数（-1,1）
3. 主循环对训练样本进行反复迭代，而每一个训练样例，会计算这个样例网络输出的误差，更新网络的权值，然后对梯度下降步骤进行迭代。

## 激活函数

神经元的激活函数必须是可导的 。因为神经网络是采用梯度下降求解权重w时，这个过程中要求f(z)是可导的。

**一） sigmoid函数**



**它称为sigmoid曲线，s型曲线**。

它把数据压缩到 (0,1)

它的导数是：



**二） tanh函数**

双曲正切函数的数学定义为：







它把数据压缩到 (-1,1)

它的导数是：

## Hebb学习规则

Hebb学习规则是一个无监督学习规则

它的思想来源于条件反射实验，Hebb的理论认为在同一时间被激发的神经元间的联系会被强化。比如，铃声响时一个神经元被激发，在同一时间食物的出现会激发附近的另一个神经元，那么这两个神经元间的联系就会强化，从而记住这两个事物之间存在着联系。相反，如果两个神经元总是不能同步激发，那么它们间的联系将会越来越弱。

设神经元i是神经元j的上层节点，用表示神经元i的输出，表示神经元j的输入，表示神经元i和神经元j的连接权重，则hebb学习规则可以表示为：



\* eta是学习率，对于单层感知器，可以把看作，把看作

那么第n+1次迭代更新权重公式是：



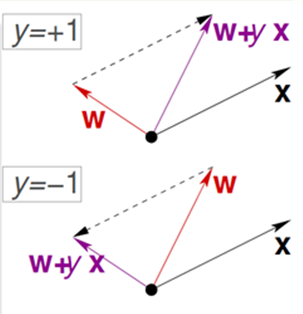
即



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

**如何理解这个公式**：

用单层感知器来简单理解，如果j是输出层，则y=1 or y=-1



当y=1时， w(n+1)=w+yx > w , 值得到强化。

当y=-1是， w+yx < w ,值得到弱化。

## Delta学习规则

**学习规则 ，误差校正学习，**适应面更广些，它可适用于非线性神经元的学习过程。

该算法**根据神经元的实际输出与期望输出差别来调整连接权**。其数学表达式为：



其中为误差函数e对神经元j输入的偏导数。也就说，是局部梯度。



但一般局部梯度更习惯用来表示。

其中， , 是线性加权和，用来表示，便于理解，则



\*f为激活函数 ，y 是经过激活函数后的值，z是输入，即vj

则delta为



\*梯度下降，优化参数w,则对w求偏导。



例如：

若激活函数是sigmoid函数，则





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

在多层感知器中，要求激活函数f(z)可导。则

1. 对于输出的神经元j , 误差是：



其中d代表实际值，y代表预测值。

**则delta计算公式是：**



1. 对于隐藏层（第n层）的神经元j

没有实际值，如何估计隐藏层的误差称为最大的难题。1985年，人们提出了BP算法解决了这一难题。

**Bp算法**：利用输出后的误差来估计输出层的前一层的误差，再用这个误差估计来估计更前一层的误差，所以**delta计算公式是：**



其中表示第n层的神经元j的delta

1. 对于输出层到隐藏层：



## DNN

“具有深度”的神经网络.

随着神经网络层数的加深，优化函数越来越容易陷入局部最优解，并且这个“陷阱”越来越偏离真正的全局最优。利用有限数据训练的深层网络，性能还不如较浅层网络。同时，另一个不可忽略的问题是随着网络层数增加，“梯度消失”现象更加严重。具体来说，我们常常使用 sigmoid 作为神经元的输入输出函数。对于幅度为1的信号，在BP反向传播梯度时，每传递一层，梯度衰减为原来的0.25。层数一多，梯度指数衰减后低层基本上接受不到有效的训练信号。

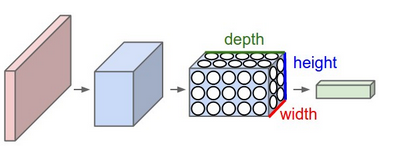
2006年，Hinton利用预训练方法缓解了局部最优解问题，将隐含层推动到了7层(参考论文：Hinton G E, Salakhutdinov R R. Reducing the Dimensionality of Data with Neural Networks[J]. Science, 2006, 313(5786):504-507.)，神经网络真正意义上有了“深度”，由此揭开了深度学习的热潮。这里的“深度”并没有固定的定义——在语音识别中4层网络就能够被认为是“较深的”，而在图像识别中20层以上的网络屡见不鲜。为了克服梯度消失，ReLU、maxout等传输函数代替了 sigmoid，形成了如今 DNN 的基本形式。单从结构上来说，全连接的DNN和上图的多层感知机是没有任何区别的。值得一提的是，今年出现的高速公路网络（highway network）和深度残差学习（deep residual learning）进一步避免了梯度弥散问题，网络层数达到了前所未有的一百多层（深度残差学习：152层，具体去看何恺明大神的paper）！

# CNN

CNN主要用于图像处理。Convolutional Netural Network

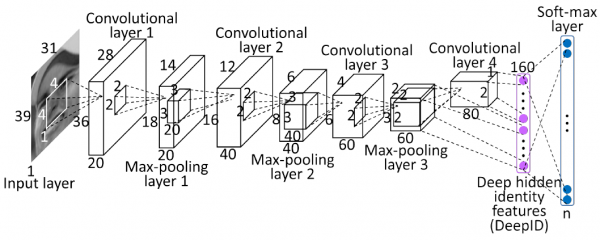
CNN一个很好的教程：<http://cs231n.github.io/convolutional-networks/>

假设输入的是一幅像素为1K\*1K的图像，隐含层有1M个节点，光这一层就有10^12个权重需要训练，这不仅容易过拟合，而且极容易陷入局部最优 对于CNN来说，并不是所有上下层神经元都能直接相连，而是通过“卷积核”作为中介。同一个卷积核在所有图像内是共享的，图像通过卷积操作后仍然间接保留原先的位置关系。卷积层之间的卷积传输的示意图如下：



depth代表卷积层多少个神经元。height x width代表窗口的大小。

另CNN与DNN的层级结构也是不一样的，它有着不同类别的层，并且每层的工作方式与普通层也有所差异。 如：



对于图像，如果没有卷积操作，学习的参数量是灾难级的。CNN之所以用于图像识别，正是由于CNN模型限制了参数的个数并挖掘了局部结构的这个特点。顺着同样的思路，利用语音语谱结构中的局部信息，CNN照样能应用在语音识别中。在普通的全连接网络或CNN中，每层神经元的信号只能向上一层传播，样本的处理在各个时刻独立，因此又被成为前向神经网络(Feed-forward Neural Networks)。

## 数学上的卷积

作者：果程C

链接：https://www.zhihu.com/question/22298352/answer/50940942

来源：知乎

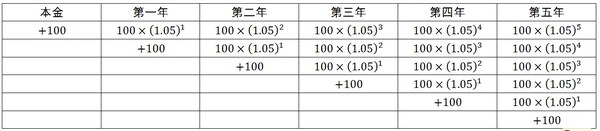
小明存入100元钱，年利率是5%，按复利计算（即将每一年所获利息加入本金，以计算下一年的利息），那么在五年之后他能拿到的钱数是100(1+5\%)^5。

https://pic2.zhimg.com/5fa86c80c31dd277d038527555aa4d75_b.jpg

一年后，小明又往银行中存入了100元钱，年利率仍为5%，那么这笔钱按复利计算，到了第五年，将收回的钱数是，我们将这一结果作为新的一行加入上面的表格中

https://pic4.zhimg.com/39f37df8c96d7219cba5d081919a3a2f_b.jpg

以此类推，如果小明每年都往银行中存入新的100元钱，那么这个收益表格将是这样的：



可见，最终小明拿到的钱将等于他各年存入的钱分别计算复利之后得到的钱数的总和，用求和符号来简化这个公式，可以得到：

http://www.zhihu.com/equation?tex=%5Csum_%7Bi%3D0%7D%5E%7B5%7D%7Bf%28i%29g%285-i%29%7D%2C+%5Cmathrm%7Bwhere%7D+%5C+f%28i%29%3D100%2C+g%285-i%29+%3D+%281.05%29%5E%7B5-i%7D

在上式中，f(i)为小明的存钱函数，g(i)为存入银行的每一笔钱的复利计算函数。在这里，小明最终得到的钱就是他的存钱函数和复利计算函数的卷积。

如果是连续,则：

http://www.zhihu.com/equation?tex=%5Cint_%7B0%7D%5E%7Bt%7D+f%28%5Ctau%29g%28t-%5Ctau%29d%5Ctau%3D%5Cint_%7B0%7D%5E%7Bt%7D+f%28%5Ctau%29%281%2B5%5C%25%29%5E%7Bt-%5Ctau%7Dd%5Ctau

这也就是卷积的表达式了，上式可以记为。

下面我们再展开说两句：

如果我们将小明的存款函数视为一个信号发生的过程（也就是激励），而将复利函数视为一个系统对信号的响应函数（也就是响应），那么二者的卷积就可以看做是在t时刻对系统进行观察得到的结果（也就是输出）。 这个结果将是过去产生的所有信号经过系统的「处理／响应」后得到的结果的叠加，这也就是卷积的物理意义了。

## 图像数据预处理

去中心和归一化。

CNN常用去均值、CNN不做归一化。因为图像天生就是在 0-255之间

pca和白化

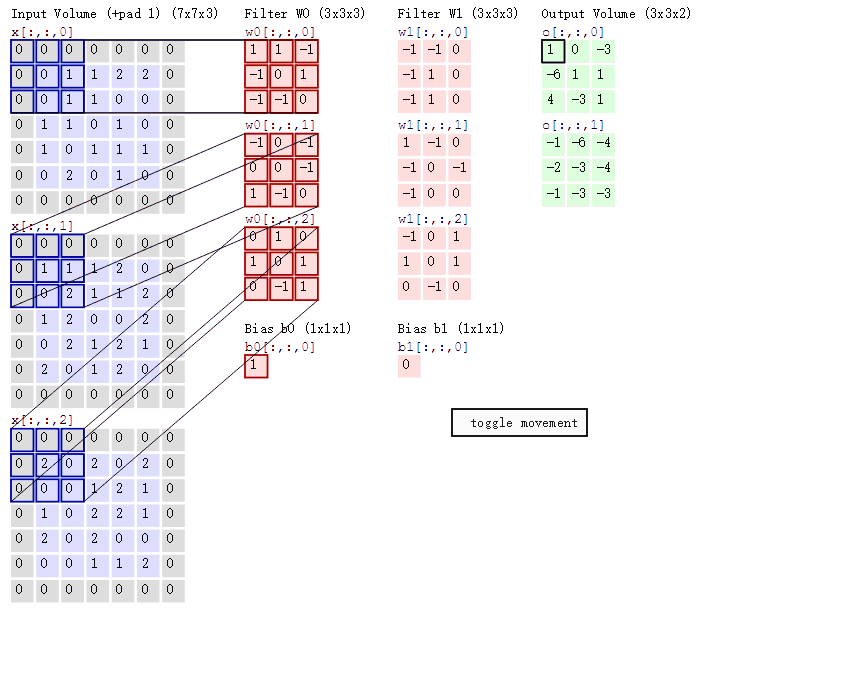
图像一般不用做 PCA、白化。

## conv layer

在神经网络中，全连接层的参数量过大。举个简单例子。图像像素是128\*128\*3，全连接的参数至少是 128\*128\*3 x 128\*128\*3 这个根本无法接受。所以通过卷积解决这个问题。

下图就是一个卷积计算。

一组固定的权重和不同的数据窗口数据做内积的过程，这在数学上刚好对应『卷积』操作，这也就是卷积神经网的名字来源。（与数学上的卷积还是不一样）



**几个重要概念**：

深度 depth ： conv层神经元的个数。如上图中有w0,w1 2个神经元，即深度=2。

步长 stride ： 窗口移动的步长。

填充值 zero-padding ：补0这个操作产生的根本原因是，为了保证窗口的滑动能从头刚好到尾。

**优势**

参数共享机制带来的参数个数的减小。即神经元的权重参数是固定的。（在前向计算中）。这样极大的减少了权重参数的个数。如上图中，一个神经元的窗口的大小是 3\*3 ，要处理RGB3个通道，所以一个神经元的参数量是3\*3\*3。 上图一共2个神经元，所以参数总个数是 2\*3\*3\*3=54. 而全连接要：2\*(7\*7\*3)^2=43218

so , 卷积层计算对高维数据处理无压力。

**关于conv的输出**：

停留次数=(data\_len-窗口\_len + 2\*zero-padding)/stride+1

输出结果=[停留次数x停留次数x RGB] x depth

\*上例中，

input层图片大小是[5\*5\*3]

停留次数=(5-3+2)/2+1=3 , 输出结果=[3\*3\*3]\*2

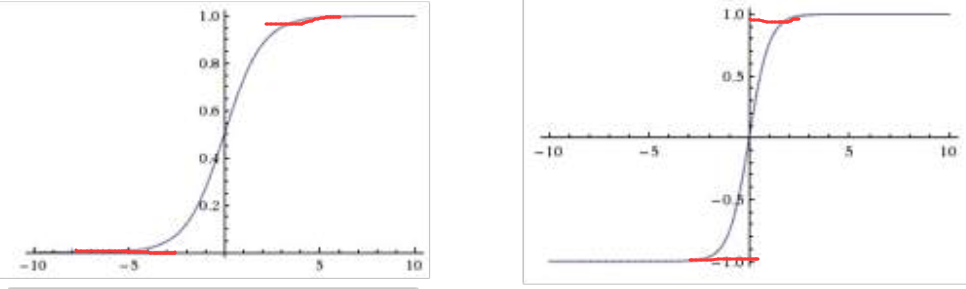
若stride=1，则 停留次数=(5-3)/1+1=3, 输出结果=[3\*3\*3]\*2 。

经常设置stride=1。 stride=1时，输出的信息= 图片大小 \* depth . 信息量反而放大了。这样更好。

## 激励层

sigmoid函数和tanh函数的缺点

它们的图像是：



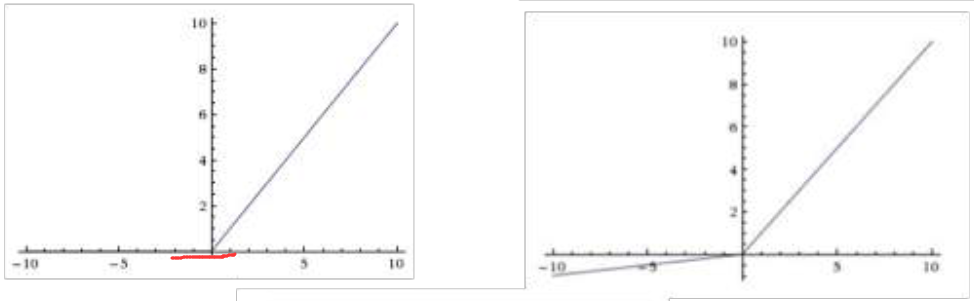
可以看到，在两端，他的斜率趋近0.（如红色标注）。也就是说偏导趋近于0。也就意味 deltaw=0， 那权重就不能更新。

所以，设计了relu函数 .



它的导数是：





relu同样会有个问题，如左图中，当x<0时，偏导也是0 。

所以设计了leaky relu



实际中，用relu就好。如果relu不行，再尝试leaky relu.

## 池化层 pooling layer

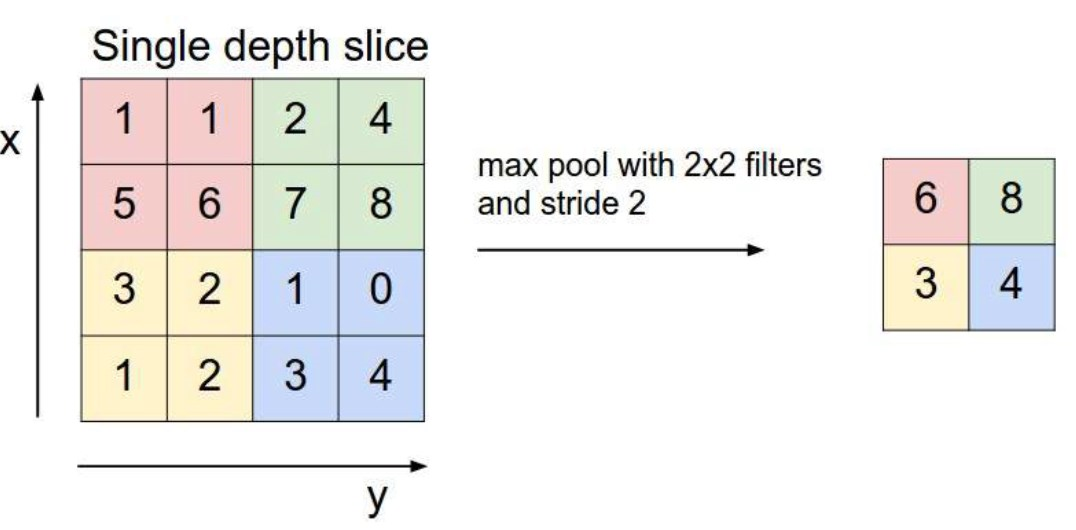
经历了卷积后，数据量依然很大。我们需要压缩。

池化层采用的是“downsampling” ,翻译为“下采样”，“亚采样”。如原 224\*224\*64 ，变成 112 \* 112 \* 32 。

位置，夹在卷积层的中间，主要目标是压缩数据的量，减少过拟合

2种池化： max 和 average 。

MaxPool用的最多。如下图所示：



## FC(全连接层 )

这是我们在介绍神经网络的时候，最标准的形式，任何神经元和上一层的任何神经元之间都有关联，然后矩阵运算也非常简单和直接。现在的很多卷积神经网络结构，末层会采用全连接去学习更多的信息。

## Batch Normalization layer

早期的研究，经常能见到它们的身影，不过近些年来的研究表明，似乎这个层级对最后结果的帮助非常小，所以后来大多数时候就干脆拿掉了。

## CNN的训练

和BP一样，也用的是SGD。

链式求导法则，每层都求导，这样每层就可以用链式求导法则串起来。

关键点：

1. 求每层的误差
2. 求局部梯度

relu层的导数是：



Pool层的导数

Pool层的操作是downsampling ,则Pool层的导数是UpSampling。

MaxPool做的是max( x11,x12,x21,x22)=x12，则它的导数是：(0,1,0,0)

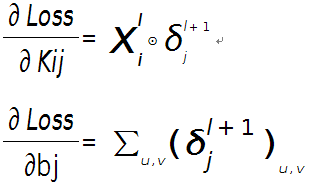
具体实现：就是记录前向传播过程中pooling区域中最大值的位置，该位置导数是1.

卷积层的导数：

这个相对复杂些,但还是能求出来的。

具体参考：<http://www.cnblogs.com/tornadomeet/p/3468450.html>中 “问题四：求卷积层相连那层的权值、偏置值导数”。

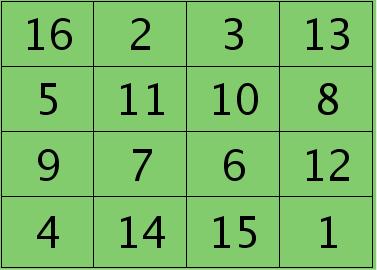
假设现在需要求第l层的第i个通道，与第l+1层的第j个通道之间的权值和偏置的导数，则计算公式如下：



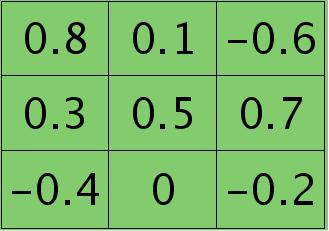
　　其中k\_ij表示卷积核

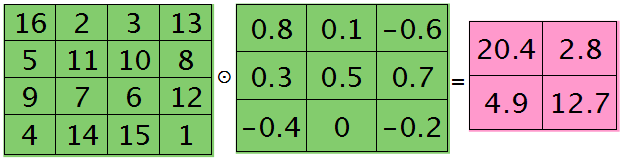
其中符号⊙表示矩阵的相关操作，可以采用conv2()函数实现。在使用该函数时，需将第l+1层第j个误差敏感值翻转。

　　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。

## dropout正则化

正则化包括：L1, L2 , L1+L2 。这种正则化一般是对损失函数做的惩罚处理。

神经网络还独有一种正则化手段，那就是dropout .

神经网络学习能力太强，容易过拟合。

dropout ,随机失活，别一次开启所有神经元。

p=0.5 #设定dropout的概率，也就是保持一个神经元激活状态的概率。

# 神经网络训练细节

## 梯度消失和梯度爆炸

计算梯度是发生在反向传播时，

**梯度消失**：前面的层比后面的层梯度变化更小，故变化更慢，从而引起了梯度消失问题。

**梯度爆炸**：前面层比后面层梯度变化更快，会引起梯度爆炸问题。

为什么会出现梯度消失呢？

以sigmoid函数举例：

它的导数是：



它的取值范围是(0,0.25] ,当 z=0，f(z)=0.5时，导数最大，等于0.25.

假设一个有2层隐藏层的神经网络。一共有3层w



根据链式求导法则





可以看到，前面的层（假设第n层）的梯度，相比比后面的层，多乘以了

因为的范围是(0, 0.25] , 矩阵的均值一般少于1，所以就会出现梯度消失。

**梯度爆炸**。

当矩阵的均值大于4时，则，这时候就会出现梯度爆炸。

神经网络的w为什么不能为0，而LR的w可以为0?

因为：神经网络是多层的，根据链式推导，非最后一层的的表达式中用到,如果w初始化为0， 则梯度为0.

而LR是单层结构，梯度公式中只涉及

## w初始化

* 每层的w都是一个矩阵。
* LR的w可以初始化为0，但NN的w不能初始化为0。
* 对于sigmoid,我们希望z的值的范围是[-4,4],如果方差等于1最好，这时候的梯度最大。
* wx+b=z ,我们希望z的分布跟x的分布保持一致（一般x的分布为mean=0,std=1）。所以w的初始化很重要。
* 一般采用xavier分布

#符合方差为0.01的高斯分布

w= np.random.randn(4,4)\*0.01

#xavier分布

w=标准高斯分布/sqrt(输入层神经元的个数)

w=标准高斯分布/sqrt(输入层神经元的个数/2) #若激励函数是relu

一个特点：

* **神经元节点输出值的方差会随着神经元节点输入样本的数量而增加。**

代码：

import numpy as np

#输入神经元个数

m=1000

x = np.random.randn(m)

b = 0

#隐藏层神经元个数

t = 10000

#输出值的方差会随着输入样本的数量而增加。

w=np.random.randn(t,m)

#xavier分布

#w=np.random.randn(t,m)/np.sqrt(m)

z=np.dot(w,x)+b

print 'z 均值：', np.mean(z)

print 'z 方差：', np.var(z)

可以看到，证明，当输入样本数量是1000时，z的方差是1000，均值0附近

当采用xavier分布时，z的方差就会保持为1.

## input数据预处理

* 去中心。 x-均值
* 归一化。归一到一个范围。如0-1标准化，z标准化
* PCA。
* 白化。在每个特征维上归一化。

**x为什么要归一化?**

因为： z=wx+b ,y=sigmoid(z) , 当 z在[-4,4]时，它的梯度是明显的。而不在这个范围梯度几乎为0. 如果x很大，比如1000， 则z很可能就不在[-4,4]这个范围里，从而计算出的梯度为0。

所谓whitening，就是把各个特征轴上的数据除以对应特征值，从而达到在每个特征轴上都归一化幅度的结果。whitening变换的几何意义和理解是，如果输入的数据是多变量高斯，那whitening之后的 数据是一个均值为0而不同方差的高斯矩阵。

简单代码实现如下：

# 假定输入数据矩阵X是[N\*D]维的

X -= np.mean(X, axis = 0) # 去均值

cov = np.dot(X.T, X) / X.shape[0] # 计算协方差

#协方差矩阵的对角线包含了每个维度的变化幅度。另外，我们都知道协方差矩阵是对称的，我们可以在其上做矩阵奇异值分解

U,S,V = np.linalg.svd(cov)

#其中U为特征向量，是一组正交基向量，是不相关的。

#去相关性。把原始数据X投射到这组维度保持不变的正交基U上，从而也就完成了对原始数据(去均值之后)的去相关操作。

Xrot = np.dot(X, U)

这么理解一下可能更好，U。所以。

#白化数据

Xwhite = Xrot / np.sqrt(S + 1e-5)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

DNN一般都要做去均值和归一化。

CNN一般只做去均值。不做归一化。因为图像天生就是在 0-255之间。也不做PCA和白化。

## mini-batch SGD

现在的SGD一般默认指 mini-batch SGD .

若是CNN，可设置batch=128

## 学习率衰减

简单说就是学习率是不断下降的

作用：可以减少训练次数，加快训练速度。（经实践证明）

* 步伐衰减

比如，10轮完整训练周期下降10%。

* 指数衰减



tensorflow的指数衰减公式是：



alpha表示衰减率

* 1/t衰减 ， 模拟退火



t是迭代次数，t越大，则eta越小。

k是常数，控制退火的速度。如k=0.1

## 动量

梯度下降法可能会收敛到局部极小值而不是全局极小值，在这里做了改进，引入了动量。momentum是模拟物理里动量的概念，积累之前的动量来替代真正的梯度。

具体措施：它增加了冲量（动量）项，使第n次迭代时权值的更新受到n-1次迭代的影响。这样，冲量有时可以滚过误差曲面的局部最小值；而且能在梯度不变的区域内增大搜索步长，加快收敛，减少训练次数。

SGD的公式是：





公式如下：



其中



或 

## 训练前检查工作

1. 加L1/L2正则的loss要比不加正则的loss更大，在第一次计算时做这个验证。
2. **试着去拟合一个小的数据集**。这是很重要的一步，在对大数据集做训练之前，我们可以先训练一个小的数据集(比如20张图片)，然后看看你的神经网络能够做到0损失/loss(当然，是指的正则化系数为0的情况下)，因为如果神经网络实现是正确的，在无正则化项的情况下，完全能够过拟合这一小部分的数据。

## 训练过程中的监控

1. **监控loss的变化应该是震荡下降的才对**。
2. **权重更新部分的比例**。这里指的是权重更新幅度和当前权重幅度的比值。一个合适的比例大概是1e-3。如果比例比这个值小很多，说明学习率太低，反之则是设定太高。

## 二次迭代方法

最优化问题里还有一个非常有名的**牛顿法**，它采用的是二阶求导。相对于一阶求导，它的更新速度会更快。

比较尴尬的是，实际深度学习过程中，直接使用二次迭代的方法并不是很实用。原因是直接计算Hessian矩阵是一个非常耗时耗资源的过程。举个例子说，一个一百万参数的神经网络的Hessian矩阵维度为[1000000\*1000000]，算下来得占掉3725G的内存。当然，我们有L-BFGS这种近似Hessian矩阵的算法，可以解决内存问题。但是L-BFGS一般在全部数据集上计算，而不像我们用的mini-batch SGD一样在小batch上迭代。现在有很多人在努力研究这个问题，试图让L-BFGS也能以mini-batch的方式稳定迭代更新。但就目前而言，大规模数据上的深度学习很少用到L-BFGS或者类似的二次迭代方法，倒是随机梯度下降这种简单的算法被广泛地使用着。

感兴趣的同学可以参考以下文献：

* On Optimization Methods for Deep Learning：2011年的论文比较随机梯度下降和L-BFGS
* Large Scale Distributed Deep Networks：google brain组的论文，比较随机梯度下降和L-BFGS在大规模分布式优化上的差别。
* SFO算法试图结合随机梯度下降和L-BFGS的优势。

## 优化方法

前面提到的学习率更新方式，都是全局使用同样的学习率。调整学习率是一件很费时同时也容易出错的事情，因此大家一直希望有一种学习率自更新的方式，甚至可以细化到逐参数更新。

下面简单介绍几个方法：

**Adagrad**

Adagrad是Duchi等在论文[Adaptive Subgradient Methods for Online Learning and Stochastic Optimization](http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf)中提出。





简单代码实现如下：

# 假定梯度为gt，参数向量为x

cache += gt\*\*2

x += - learning\_rate \* dx / np.sqrt(cache + 1e-8)

其中变量cache有着和梯度一样的维度，然后我们用这个变量存放梯度平方的累加。初始化时， dx/np.sqrt(cache)=dx/dx =1 ; cache随着迭代次数的增加越来越大,dx/np.sqrt(cache)越来越小。这种方法的好处是，对于高梯度的权重，它们的有效学习率被降低了；而小梯度的权重迭代过程中学习率提升了。而分母开根号这一步非常重要，不开根号的效果远差于开根号的情况。平滑参数1e-8避免了除以0的情况。

**RMSprop**

RMSprop是对Adagrad的改进。主要是cache这做了改进，做了一个平滑处理。由大神Geoff Hinton的coursera课程第6节的讲义第29页提出。

大致的代码如下：

cache = decay\_rate \* cache + (1 - decay\_rate) \* gt\*\*2

x += - learning\_rate \* dx / np.sqrt(cache + 1e-8)

这里的decay\_rate是一个手动敲定的超参数，我们通常会在[0.9, 0.99, 0.999]中取值。需要特别注意的是，x+=这个累加的部分和Adagrad是完全一样的，但是cache本身是迭代变化的。

**Adam**

Adam(Adaptive Moment Estimation)本质上是带有动量项的RMSprop，它利用梯度 和梯度的平方累加，动态调整每个参数的学习率。Adam的优点主要在于经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。公式如下：











## 超参数的设定

* **初始学习率**。
* **学习率衰减**。
* **正则化系数/强度**(包括l2正则化强度，dropout比例)
* **动量参数alpha**.

设定一个搜索范围，用交叉验证等去搜索和找到最合适的超参数。例如初始学习率设为0.1,0.05,0.01等。

## 一个实验

(2014年9月30)

目标：用多层感知器区分y=sin(x),flag=1和y=cos(x),flag=0.

1）使用neurolab的情况：一般情况下，Neurolab能正确识别。但如果一开始就不训练了，或者说一开始就达到局部最优值，那也不能正确识别。

2）自编多层感知器py脚本：mul\_layer.py

* 数据归一化。这里需要对输入数据和输出数据进行标准化。不做这个处理，得不到满意的结果。归一化的范围与激活函数的范围保持一致。若激活函数是sigmoid,则范围是[0,1]，若是tanh,则范围是[-1,1]。

y=(ymax-ymin)\*(x-xmin)/(xmax-xmin)+ymin

* **权值矩阵的均值非常重要**。采用xavier初始化。
* **学习率eta**。由于采用的是随机梯度下降SGD。即是对每个样本进行优化。那么步长eta选择小一点的数据比较好。目前采样模拟退火，设置eta\_0=0.1,r=max/10,还比较合适。
* **动量参数alpha**. 动量参数不宜过大，否则容易错过最优值。目前测试alpha=0.005还好。
* **误差的评估**。可以使用最小均方，在测试实践中，也可以使用了误差个数来判定更直观，即二元误差（一致=0，不一致=1）。
* **偏置b**。可以初始赋值=0.也可以赋值unifom(-1,1)
* **要使用口袋算法**。简单的说就是，口袋里面只装入最优值。因为w是在不断变化的，且在误差评估前变化的，为避免已经达到最优值，w还发生计算从而错过最优值这种情况。需要使用口袋算法。

## 实验总结

BP算法

每个样本的学习分：**前向计算+后向计算**

前向计算，根据权重矩阵由前向后一层层神经元的推进，每次推进根据权值计算每层的输出值；

反向计算，根据输出结果与目标的误差来由后向前调整神经网络的权值。

整个算法过程是反向传播算法，使用梯度下降方法搜索可能假设的空间，迭代减少误差以拟合权值矩阵。

开发的几个要点：

* **输入数据处理**。消除均值、去相关性、协方差均衡(白化过程)。
* **输出层处理**。期望响应值要偏离sigmoid函数的极限值(采用+或-ε扰动法)。
* **隐藏层设计** 。隐藏层不宜过多。理论上已经证明：具有偏差b和至少一个S型隐含层加上一个线性输出层的网络，能够逼近任何有理函数。增加层数主要可以进一步的降低误差，提高精度，但同时也使网络复杂化，从而增加了网络权值的训练时间。 一般情况下应优先考虑增加隐含层中神经元数。
* **激活函数。**采用双曲正切函数效果要更好些。
* **权值初始化** 。 在实践中，w(0)初始化时，要赋值0附近的小随机数。一个好的初始权值矩阵，能使网络快速收敛。
* **学习率和动量参数**。学习率大，训练速度快，但可能错过了误差曲线的全局最小值；而过小，则训练时间加长，还容易陷入局部最小值。动量参数也有类似的问题，这些参数需要在实践中反复调试，对训练效果反复对比，才能找到合适的参数。

# RNN

全连接的DNN还存在着另一个问题——无法对时间序列上的变化进行建模。然而，样本出现的时间顺序对于自然语言处理、语音识别、手写体识别等应用非常重要。对了适应这种需求，就出现了另一种神经网络结构——循环神经网络RNN。而在RNN中，神经元的输出可以在下一个时间戳直接作用到自身，即第i层神经元在m时刻的输入，除了（i−1）层神经元在该时刻的输出外，还包括其自身在（m−1）时刻的输出。

RNN可以看成一个在时间上传递的神经网络，它的深度是时间的长度！正如我们上面所说，“梯度消失”现象又要出现了，只不过这次发生在时间轴上。对于t时刻来说，它产生的梯度在时间轴上向历史传播几层之后就消失了，根本就无法影响太遥远的过去。因此，之前说“所有历史”共同作用只是理想的情况，在实际中，这种影响也就只能维持若干个时间戳。为了解决时间上的梯度消失，机器学习领域发展出了长短时记忆单元 LSTM，通过门的开关实现时间上记忆功能，并防止梯度消失

事实上，不论是那种网络，他们在实际应用中常常都混合着使用，比如CNN和RNN在上层输出之前往往会接上全连接层，很难说某个网络到底属于哪个类别。不难想象随着深度学习热度的延续，更灵活的组合方式、更多的网络结构将被发展出来。

# GAN

# DQN