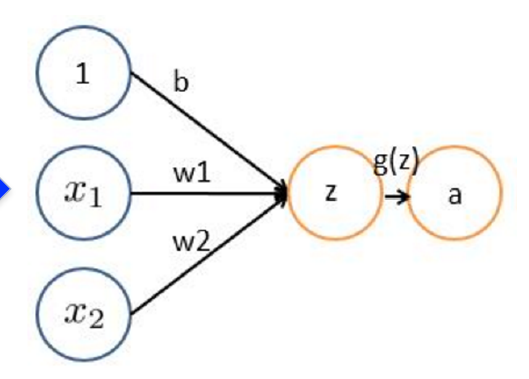
# 神经网络

## 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：

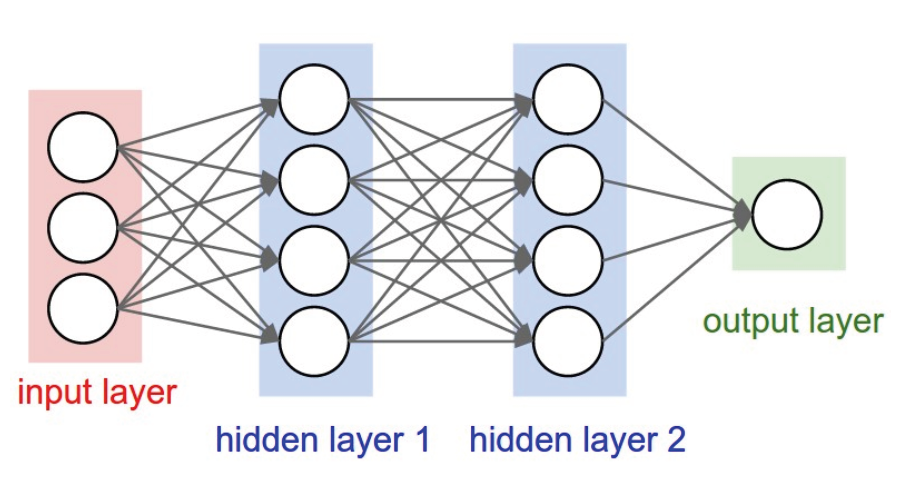




x是输入特征，w是权重，g是激活函数。如果g是sigmoid函数，这就是一个逻辑回归。

## 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。

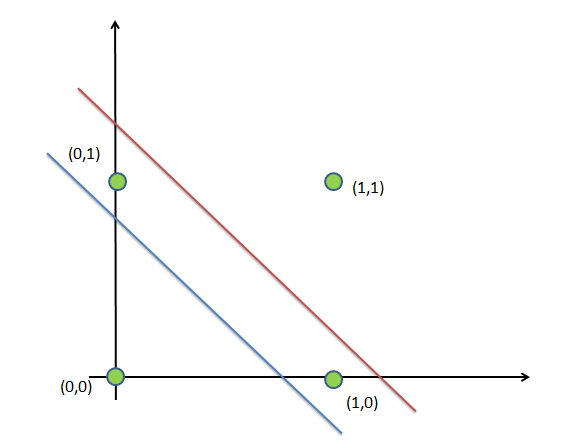
## 神经网络的能力

**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：



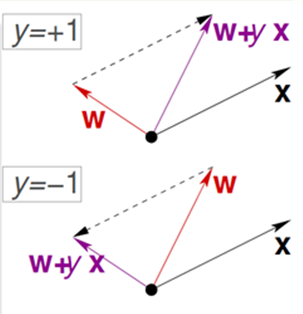
即



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

**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。或者说，神经元j的误差敏感度。



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）！

# 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。一种归一化的方法。。

**x为什么要归一化?**

因为：

1. 不同的特征的scale是不同的，通过归一化放到统一的scale.

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)

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

若是图像处理，一般就不用归一化或白化。因为图像天生就是在 0-255之间。

## batch-size

神经网络的训练都是一个batch一个batch的训练。也就是mini-batch.

batch不适宜过大，也不适宜过小。

一般设置，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0.01-0.001。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。

## 优化算法

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：







tensorflow默认值是：

learning\_rate=0.001, alpha=0.9, beta=0.999, epsilon=1e-8

## 超参数的设定

* **初始学习率**。 可以设为0.01
* **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：

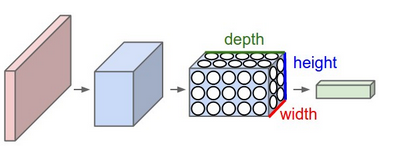
* **输入数据处理**。消除均值、去相关性、协方差均衡(白化过程)。
* **输出层处理**。期望响应值要偏离sigmoid函数的极限值(采用+或-ε扰动法)。
* **隐藏层设计** 。隐藏层不宜过多。理论上已经证明：具有偏差b和至少一个S型隐含层加上一个线性输出层的网络，能够逼近任何有理函数。增加层数主要可以进一步的降低误差，提高精度，但同时也使网络复杂化，从而增加了网络权值的训练时间。 一般情况下应优先考虑增加隐含层中神经元数。
* **激活函数。**采用双曲正切函数效果要更好些。
* **权值初始化** 。 在实践中，w(0)初始化时，要赋值0附近的小随机数。一个好的初始权值矩阵，能使网络快速收敛。
* **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。

# CNN

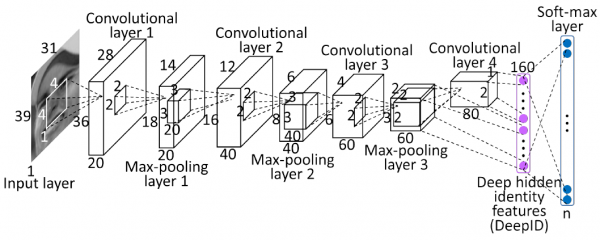
CNN主要用于图像处理。Convolutional Netural Network

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128\*128的图像，隐含层有1000个神经元，光这一层就有1638w个参数，这个对于普通的NN是灾难性的。对于CNN来说，它创造性的应用“卷积核”解决了这个难题 。

普通的NN的隐藏层是向量，而卷积层是一个三维立体结构（width,height,depth）。如： 

CNN的层级结构，不同层有不同的形式和功能：



## 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

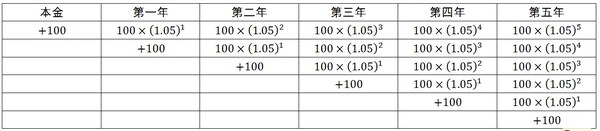
小明存入100元钱，年利率是5%，按复利计算（即将每一年所获利息加入本金，以计算下一年的利息），那么在五年之后他能拿到的钱数是100(1+5\%)^5。

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

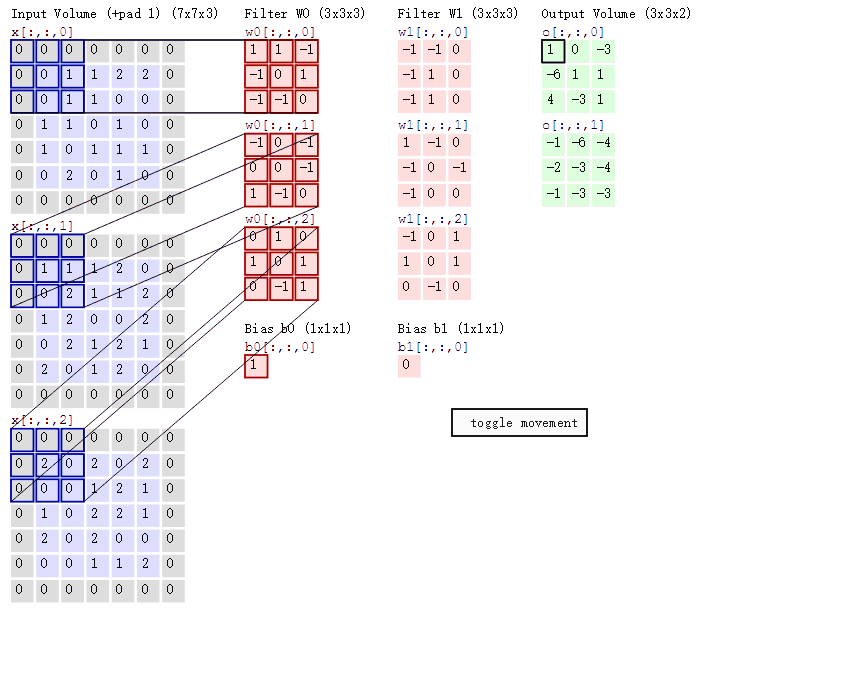
图像一般不用做 PCA、白化。

## 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。

是否填充0 zero-padding ：如果补0，则可以保证从窗头滑动到窗尾。

如果且stride=1,则，输出和输入在 height x wdith上保持不变。

**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**：

如果：补0，及padding=”SAME”,

停留次数=in\_len/stride, 向上取整数

输出结果=[停留次数x停留次数] x depth

如果：不补0， padding=”VALID”

停留次数=(in\_len-窗口\_len + 1)/stride , 向上取整数

输出结果=[停留次数x停留次数] x depth

**关于参数量**

参数量=输入通道 \* filter size \*输出通道= 3 x[3 x3] x2

\*拿上图进行举例：，

input层图片大小是[5x5x3]，filter=[3x3]x2

则：停留次数=(5-3+1)/2=3 , 输出结果=[3x3]x2。

\*注意这里，输入的时候是3通道，输出的时候则变成了2通道。所以，卷积的depth=输出矩阵的通道。

若stride=1,padding=SAME，则 停留次数=5/1=3, 输出结果=[5x5]x2 。

经常设置stride=1。 stride=1时，输出的信息= 图片大小 x depth .

## 池化层 pooling layer

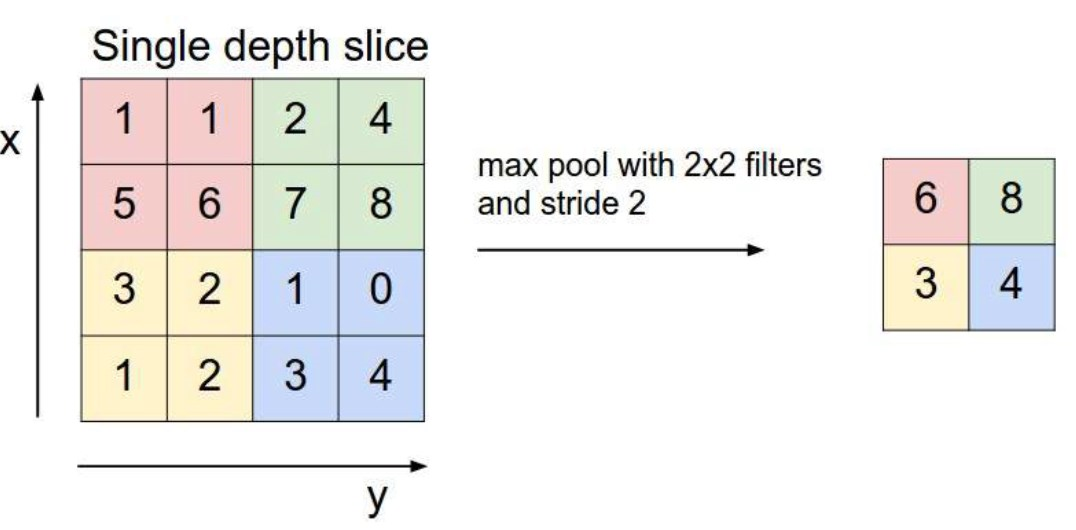
设计池化层的目的，是为了进行数据压缩。如原 224\*224\*64 ，变成 112 \* 112 \* 64 。

池化层采用的是“downsampling” ,翻译为“下采样”，“亚采样”。

2种池化方法：

* Max Pooling
* Average Pooling。

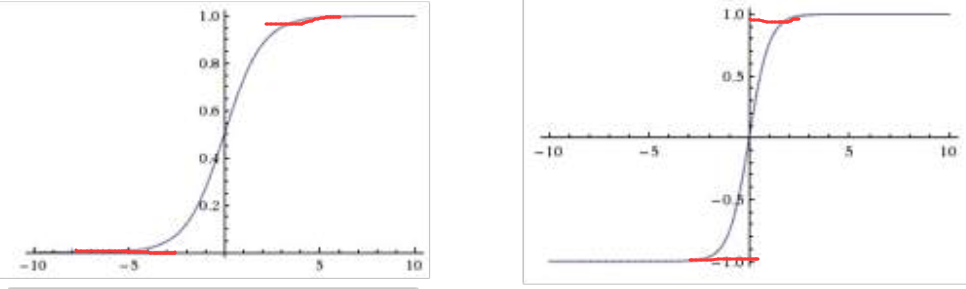
Max Pooling如下图：



## relu激励函数

sigmoid函数和tanh函数的缺点

它们的图像是：



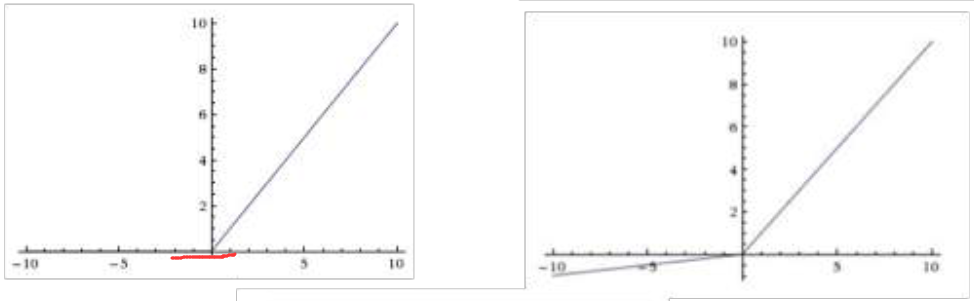
可以看到，在两端，他的斜率趋近0.（如红色标注）。也就是说偏导趋近于0。也就意味 ， 那权重就不能更新。

所以，设计了relu函数 .



它的导数是：





它的**优点**：

1. x>0, 则f(x)的导数是1, 不会存在为0的情况。而sgimoid函数当x>4时，梯度几乎为0

它的**缺点**：

·缺点1：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.

·缺点2：ReLU，截断了x<0的部分，改变了数据的分布，因此ReLU后加Batch Normalization进行还原优化。

## LRN

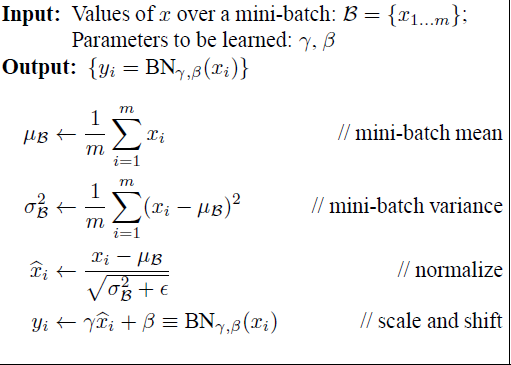
Local Response Normalization, 局部响应归一化。

sigmoid(x)的范围是(0,1),而relu(x)是[0,+8],是无上限的。 ALexNet是第一个使用relu的网络，它同时也提出了relu后接LRN ，对relu(x)的输出y进行缩放。后来的NN使用BN后，就没人使用LRN了。

## BN

Batch Normalization是由google提出的一种训练优化方法。参考论文：[Batch Normalization Accelerating Deep Network Training by Reducing Internal Covariate Shift ]( http://arxiv.org/abs/1502.03167)

BN就是对**一个mini-batch的数据**进行**正太标准化**。input的标准化，是对整个input的数据进行标准化。而最后的“scale and shift”操作是让数据变化到设定的均值和标准差（beta=均值，gamma=标准差，一般是beta=0, gamma=1）。



**why BN?**

因为，反向传播时，每层的梯度计算都会用到它的输入。



但是在NN中，中间层的输入的数据分布就发生了改变，与输入层的数据分布不一样。如果v\_i 大多小于1，那么传到这里的时候梯度会变得很小，从而梯度消失；而如果 v\_i 大多大于1，那么传到这里的时候会变得很大，从而梯度爆炸问题。BN所做的就是解决这个梯度传播的问题，因为BN作用抹去了v\_i的scale影响。

so, BN解决了梯度消失和梯度爆炸的难题。

**BN trainning**

* 设置 beta和gamma. 经过BN之后，数据分布发生了改变，而 beta和gama表示数据变换的目标，（mean和variance） .同时，也能在反向传播时还原输入层vi的数据分布。
* 计算batch的mean和variance. 在train阶段，直接计算，如：mean, variance = tf.nn.moments(x, [0, 1, 2])
* predict阶段，读取moving\_mean, moving\_variance作为mean,和variacne进行bn. 不能直接计算。

**BN use**

BN的使用位置是 conv层和relue之间： conv + BN + relu .

这个位置应该是因为：conv改变数据分布，通过bn可以进行还原。而relu又保持了很好的非线性特性。.

bn这个位置，参考的是：S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep

network training by reducing internal covariate shift. In ICML, 2015.

**BN的好处**

实践证明：

* 更快的训练速度。
* 更高的精度

## Swish激活函数

2017年10月，google提出了一个新的激活函数，swish激活函数：



其中，

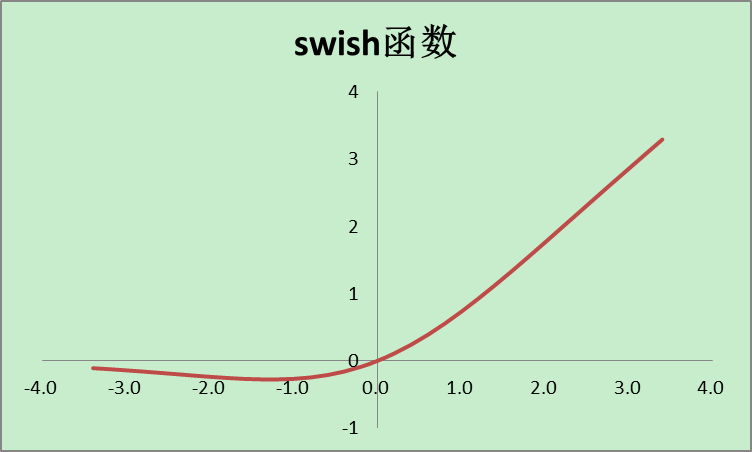
设计思路：

Swish 的设计受到 LSTM 中使用 sigmoid 函数进行门控的启发。我们使用同样的值进行门控来简化门控机制，称为自门控（self-gating）

**它的导数：**



它的图形：



**它的特性：**

* 无上界有下界。这和relu一样。
* 非单调且平滑。（relu非单调，但不平滑，生硬。而swish的平滑使的在深度神经网络的表现要好于relu）

## 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，最后一层会采用全连接。

## 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.

**卷积的导数：**

这个相对复杂些,但还是能求出来的。

对于FC层来说， , 

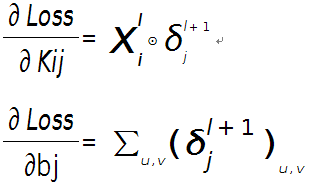
根据链式法则，误差e对w的偏导是：，

对卷积来说，矩阵乘法变成了卷积操作。，那么： 

答案：

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”。

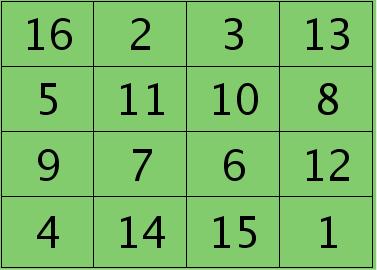
假设现在需要求第l层的第i个通道，与第l+1层的第j个通道之间的权值和偏置的导数，则计算公式如下：



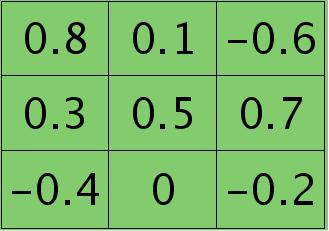
　　其中k\_ij表示卷积核

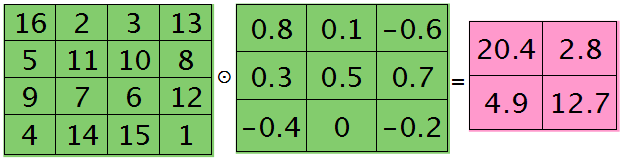
其中符号⊙表示矩阵的相关操作，可以采用conv2()函数实现。在使用该函数时，需将（第l+1层第j个误差敏感值）翻转。

　　例如，xi(第l层某个通道的矩阵i)大小为4×4,如下：



（第l+1层第j个神经元的误差敏感值矩阵）大小为3×3,如下：



　　卷积核为2×2，则,卷积核的导数是：　

　　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。

\*dropout只在FC层使用。

## CNN发展历史

* LeNet(1998)：卷积神经网络始自1990年代，最著名的是LeNet
* AlexNet(2012)：2012年，Alex Krizhevsky发布了AlexNet，是LeNet的更深、更宽版本，并且大比分赢得了当年的ImageNet大规模图像识别挑战赛(ILSVRC)。这是一次非常重要的大突破，现在普及的卷积神经网络应用都要感谢这一壮举。
* ZF Net(2013)：2013年的ILSVRC赢家是Matthew Zeiler和Rob Fergus的卷积网络，被称作ZF Net，这是调整架构超参数的AlexNet改进型。
* VGGNet(2014)：2014年 ILSVRC亚军，突出贡献是展示了网络的深度（层次数量）是良好表现的关键因素。
* GoogleNet(2014)：2014年ILSVRC冠军，来自Google的Szegedy et al.。其主要贡献是研发了Inception Module，它大幅减少了网络中的参数数量（四百万，相比AlexNet的六千万）。
* VGGNet(2014)：2014 ILSVRC亚军，突出贡献是展示了网络的深度（层次数量）是良好表现的关键因素。
* ResNet(2015)： Kaiming He研发的Residual Network是2015年的ILSVRC冠军，也代表了卷积神经网络的最高水平，同时还是实践的默认选择（2016年5月）。
* DenseNet(2016.8)：由Gao Huang发表，DenseNet的每一层都直接与其他各层前向连接。DenseNet已经在五个高难度的物体识别基础集上，显式出非凡的进步。

## LeNet

1998年提出，mnist准确率达到99.2%。LeNet标志着CNN的真正面世。

后续，2012 ALexNet(8 layers), 2014 VGG(19 layres) 都是类似LeNet结构。是LeNet的改进型。

**结构：**

input->卷积->池化->卷积->池化->FC->FC->FC

即：（卷积+->池化?)+全连接 的结构。

**具体：**

input (32x32x1)

->卷积 (w=[5,5]x6,strides=1, padding="VALID") => 32-5+1=28, 输出：28 x 28 x 6

->池化 (ksize=[2,2], strides=2, padding="SAME") => 14 x 14 x6

->卷积 (w=[5,5]x6,strides=1, padding="VALID") => 10 x10 x6

->池化 (ksize=[2,2], strides=2, padding="SAME") => 5 x 5 x6

->FC (size=120)

->FC (size=84)

->FC (size=10)

**规律总结：**

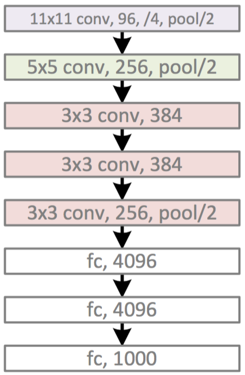
depth,一般逐层递增。

strids, 一般是1， 也有2，3.

## AlexNet

AlexNet 可以说是具有历史意义的一个网络结构，可以说在AlexNet之前，深度学习已经沉寂了很久。历史的转折在2012年到来，AlexNet 在当年的ImageNet图像分类竞赛中，top-5错误率比上一年的冠军下降了十个百分点

AlexNet网络的架构，它包含8层，5个卷积层和3个FC层



AlexNet 之所以能够成功，原因在于：

* Data augmentation(数据增强)，防止过拟合。
* 网络增大。（5个卷积层+3个全连接层）
* relu的出现。
* dropout的使用。
* 多GPU加速计算

图像的数据增强方式常用有：

* crop, 修剪。如：从原始图像（256,256）中，随机的crop出一些图像（224,224）
* flip，翻转。如：水平翻转
* color jittering，颜色抖动。如：给图像增加一些随机的光照

python代码参考：http://www.cnblogs.com/gongxijun/p/6117588.html

**VGG**

VGG很好地继承了AlexNet的衣钵，一个字：深，两个字：更深。19层

VGG的结构，都采用3x3的卷积核，深度从64，一直增加到后面的512，然后 pool/2, 最后接3个FC层分别是：FC\_4096, FC\_4096,FC\_1000.

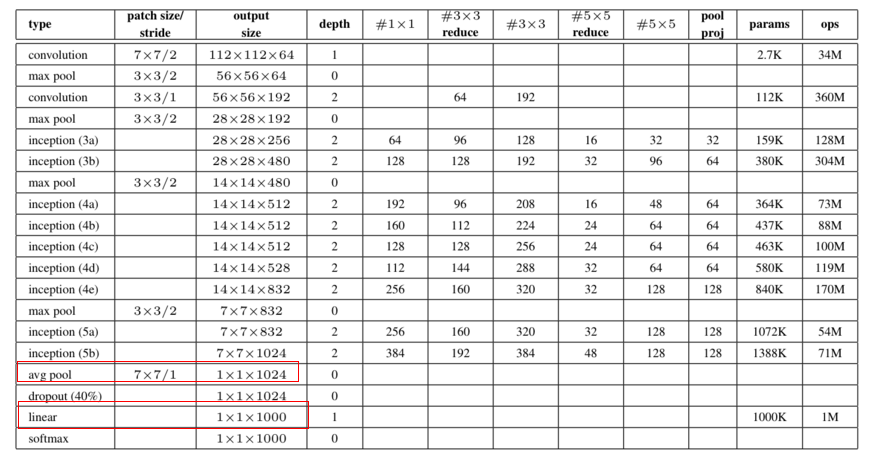
![image](https://www.52ml.net/wp-content/uploads/2016/08/vgg19.png?\_=5821591)

## google(Inception) Net

google Inception Net 首次出现在ILSVRC 2014的比赛中. 这一届比赛中的Inception Net称为Inception V1, 现在发展到 Inception V5

**GoogLeNet的模型结构及参数：**

![image]( https://raw.githubusercontent.com/stdcoutzyx/Blogs/master/blogs2016/imgs\_inception/5.png)



* 层数更多，Inception v1是22层，VGG是19层。（这里只统计conv,FC层）
* 参数更少。参数不到六百万，而AlexNet是六千万。

**参数怎么减少的？**

**AlexNet**

AlexNet只计算FC层参数量就非常huge.

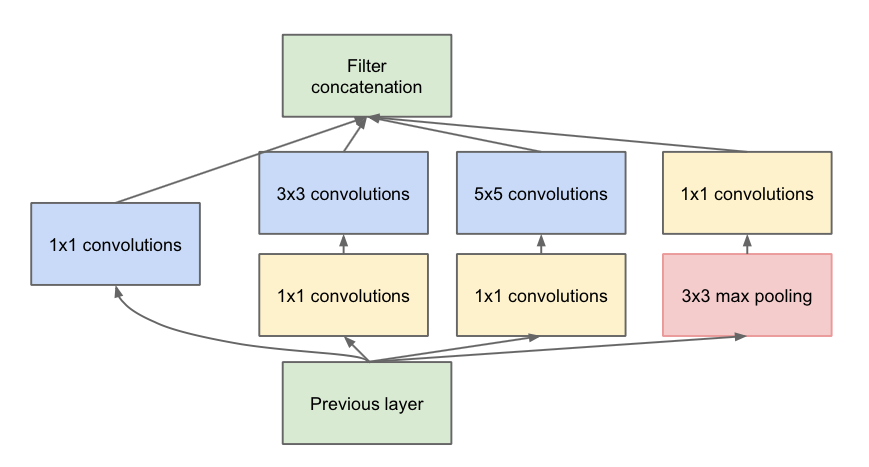
1. pool, FC\_4096之间的参数量: [2\*2]\*256\*4096=400w

2）FC\_4096, FC\_4096之间的参数量 4096 \*4096 =1600w

**GoogLeNet**

1. google net只有一层FC，而alexNet是3层FC。FC x FC的参数huge.
2. google net FC前面的avg pool的输出是1x1x1024, FC层的神经元是1024，它们之间的参数量只有100w，而alexNet这层有400w
3. google net的结构，大部分采用1x1的卷积核，1x1的卷积核相比3x3只有其1/9的参数量。

**inception module**



主要特点：

* LeNet的卷积采用的是串联，而Inception结构采用的并联。采用不同大小的卷积核意味着不同大小的感受野，最后拼接意味着不同尺度特征的融合；
* 文章说很多地方都表明pooling挺有效，所以Inception里面也嵌入了。
* 黄色的1x1卷积核的作用是降维 。

**为什么可以并联?**

因为：设定stirde=1,padding用0填充的话，则输出和输入的尺寸(height x width)一样。

so, 用1x1xdepth, 3x3 xdepth, 5x5 xdepth 的过滤器，最后的输出的尺寸(height x width)和input是一样的，这样就可以concat.（输出depth=过滤器depth之和）

tensorflow提供了 tf.contrib.slim快速实现。

**Inception V2**

使用3×3的已经很小了，那么更小的2×2呢？2×2虽然能使得参数进一步降低，但是不如另一种方式更加有效，那就是Asymmetric方式，即使用1×3和3×1两种来代替3×3。

使用2个2×2的话能节省11%的计算量，而使用这种方式则可以节省33%。于是，Inception再次进化为V2。

## 1x1卷积

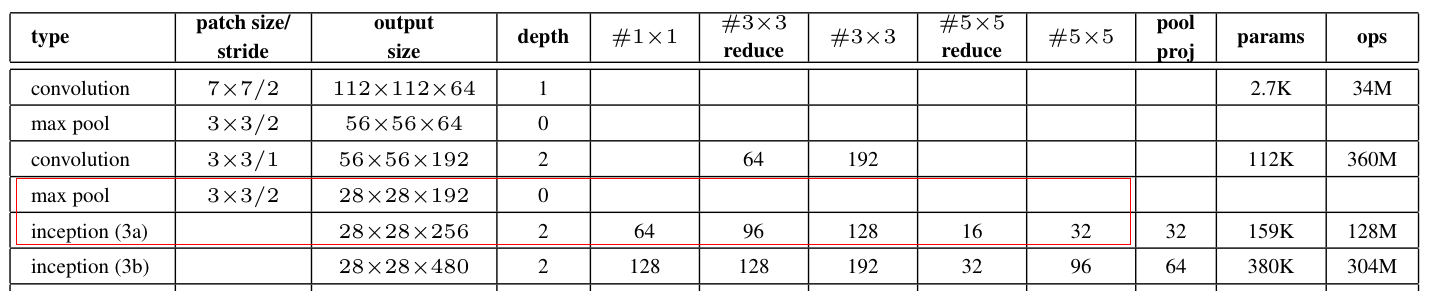
在google inception net，及 resNet中，大量使用1x1卷积，那么1x1卷积究竟起什么作用呢？

答案：

1）如果输入信号是2维的，1x1的卷积的确没什么用。但是图像的输入是三维的（有depth通道），所以，1x1的卷积能对depth维度进行处理。

2）1x1卷积可以实现通道的降维和升维，从而减少参数量。

例如，google net的inception(3a) ：

1×1卷积通道为64，3×3卷积通道为128，5×5卷积通道为32

3×3和5×5卷积层前分别加入了通道数为96和16的1×1卷积层。（即reduce层）

分别计算参数量，这里只举例计算3x3卷积和：

无reduce层：192\*3\*3\*128=221184

有reduce层：182\*1\*1\*96 + 96\*3\*3\*128 =128064

可以看到参数量减少了.

# ResNet

## ResNet

参考：<http://www.jianshu.com/p/e502e4b43e6d>

论文：<https://arxiv.org/abs/1512.03385>

理论上，神经网络越深，学习能力越强。但神经网络存在梯度消失的问题。梯度消失”问题指的是即当梯度在被反向传播到前面的层时，重复的相乘可能会使梯度变得无限小。so，层数越多，前面的层学习原始数据的能力消失掉。

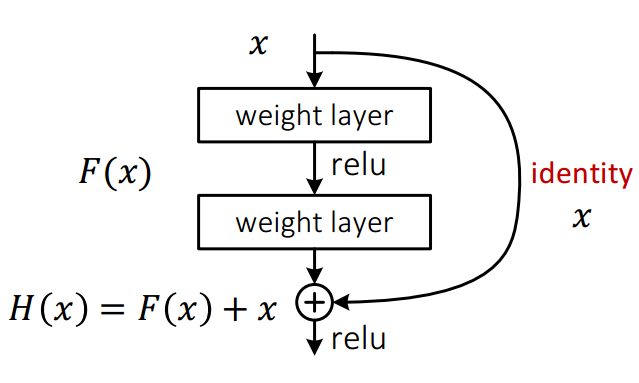
**ResNet的思路：**

ResNet的基本思想是引入了能够跳过一层或多层的“shortcut connection”。目的：增加网络的层不应该降低网络的性能，因为我们可以将“恒等变换(identity mapping)”简单地叠加在网络上，而且所得到的输出架构也会执行相同的操作。这就暗示了更深层的模型的训练错误率不应该高于与之对应的浅层模型。

可以这样理解：如果只有n层时,误差是L1, 若增加堆叠的层后的误差是L2, 由于shortcut, L2的误差应该少于L1, 所以，堆叠的层学习的效果是L1-L2 ，也就是残差。

**网络结构：**

ResNet的block结构,“Deeper Bottleneck Architectures”（以下简称DBA）：



图中的identity，是identitiy map , 翻译为恒等变换

【Q】看到这个图时，产生了一个疑问？

H(x)=F(x)+x , x的size or depth与 F(x) 不一致，怎么做“+”?

【A】 将x转换为与F(X)一致。

例如：

def subsample(inputs, stride, scope=None):

"""如 6x6 ,变成3x3"""

if stride == 1:

return inputs

else:

return tf.nn.max\_pool(inputs, ksize=[1, stride, stride, 1], strides=[1, stride, stride, 1], padding='SAME', name=scope)

#

if in\_depth != out\_depth:

shortcut = slim2.conv2d(orig\_x, 'shortcut', out\_depth, [stride,stride], stride, activation\_fn=None)

else:

shortcut = slim2.subsample(orig\_x, stride)

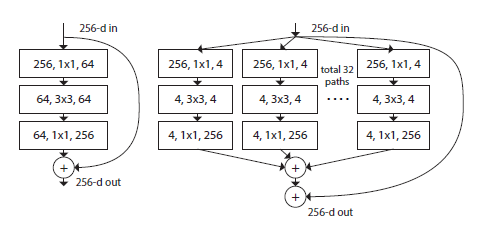
x += shortcut

## ResNeXt

论文链接：https://arxiv.org/abs/1611.05431

这是一篇发表在2017CVPR上的论文，介绍了ResNet网络的升级版：ResNeXt

**基本构件**：



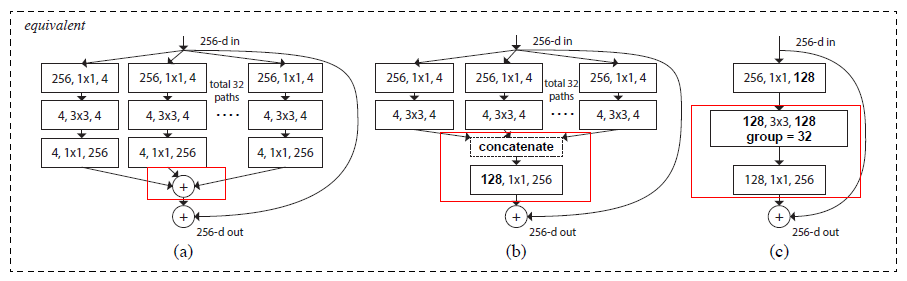
256,1x1,64表示：输入depth=256, 卷积核=1x1，输出depth=64

这看起来跟Inception模块非常相似。因为都采用并联。

不同的是：

* Inception每一个路径互不相同，有而在这个架构中，所有的路径都遵循了相同的拓扑结构。
* 不同路径输出的合并是通过相加来实现的，当然也可以设置为合并，group等多种变种

更多的合并方式：



a 就是前面所说的aggregated residual transformations。

b 则采用两层卷积后 concatenate，再卷积，

c采用的是grouped convolutions，也就是只在3x3这个conv进行并联。

作者在文中明确说明这三种结构是严格等价的，并且用这三个结构做出来的结果一模一样。

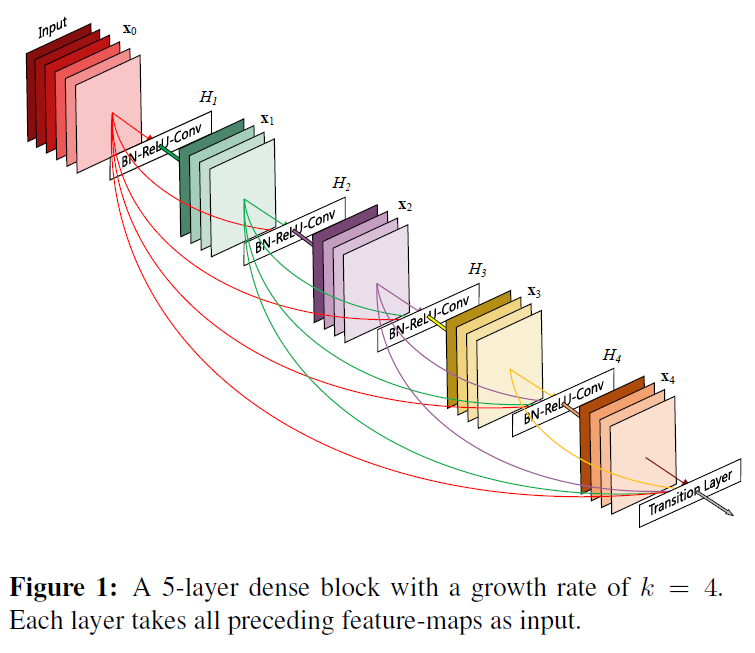
## DenseNet

2016年8月发布。论文 (<https://arxiv.org/abs/1608.06993>)

github: <https://github.com/liuzhuang13/DenseNet>

参考：<http://blog.csdn.net/u012938704/article/details/53468483>

核心组件：dense block:



dense block进一步使用了shortcut connections，将所有的层互相连接起来。在这个新架构中，每一层的输入都包含了所有较早的层的feature maps，而且它的输出被传递至每个后续层。这些feature maps在depth维度上进行 concatenation。

每一层的输入都包含了所有较早的层的feature maps，这个好处是：后面的层的输入包含第一层的feature, 则损失的梯度就能direct传导给第一次，从而减轻梯度消失。

图中的block，有4个卷积层，它一共有4\*(4+1)/2个连接。而传统的CNN，4个卷基层，则只有4个连接。

这些feature maps在depth维度上进行 concatenation，这个后果就是后面的层的输入特征的通道会越来越大，第L层的输入特征的depth= k0+k(L-1)。于是使用了一个“growth rate”(k)的超参数用来控制网络的宽度（即通道数）。（dense block里，stride=1,所以输入输出的size一样。超参数k表示dense block里所有conv层的depth都等于k,即每个conv层的depth=k）

**DenseNet-B**

虽然每个层的out\_depth=k，但是后面层的输入依然很多，第L层的in\_depth= k0+k(L-1)，若L=30, k0=k=32,则=30\*32，这个就非常大了。因此引入了Bottleneck layers 。本质上是引入[1x1] x4k的卷积层来进行降维。 于是:

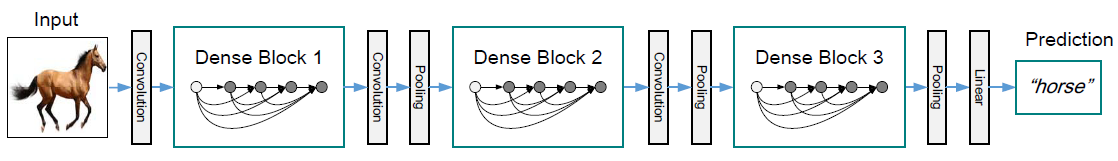
BN−>ReLU−>Conv(3×3) => BN−>ReLU−>Conv(1×1)−>BN−>ReLU−>Conv(3×3)

这样第L=30层的 3x3卷积层的输入就由 30\*32 变成了 4\*32. 减少了很大的参数量。

**DenseNet-C**

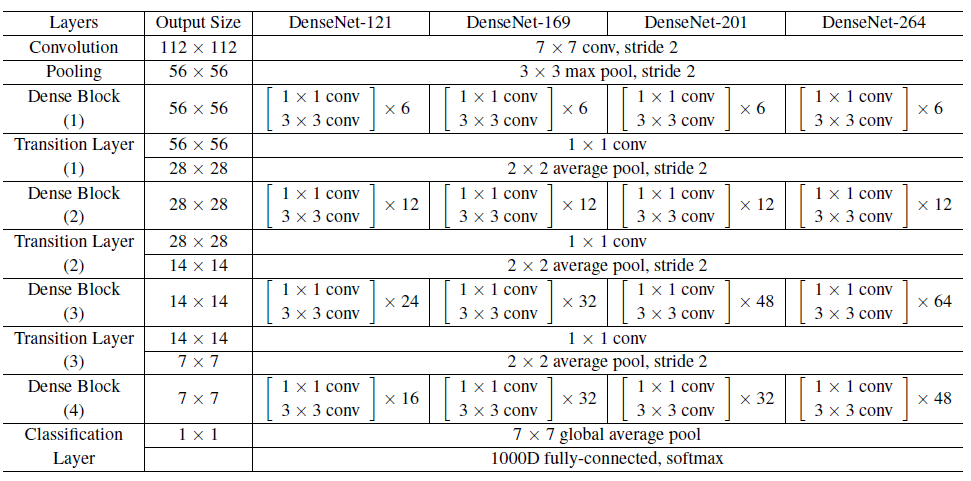
dense block和dense block 通过 conv + pool进行过渡。 conv+pool用来控制 input size,进行compression. 所以叫DenseNet-C

A deep DenseNet with three dense blocks 如下：



即采用Bottleneck,又采用compression，叫**DenseNet-BC**。但一般DenseNet-BC简称DenseNet

以下为论文中给列出的**DenseNet-BC**具体结构，包括DenseNet-121, DenseNet-169等。



图中的 conv= BN-ReLU-Conv , 前3个网络超参数k=32. 最后denseNet-264的k=48

1x1 conv是用来降维的，depth=4k, 作为Bottleneck layer

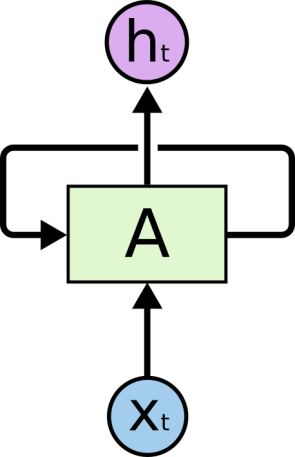
# RNN

## RNN

http://www.jianshu.com/p/9dc9f41f0b29

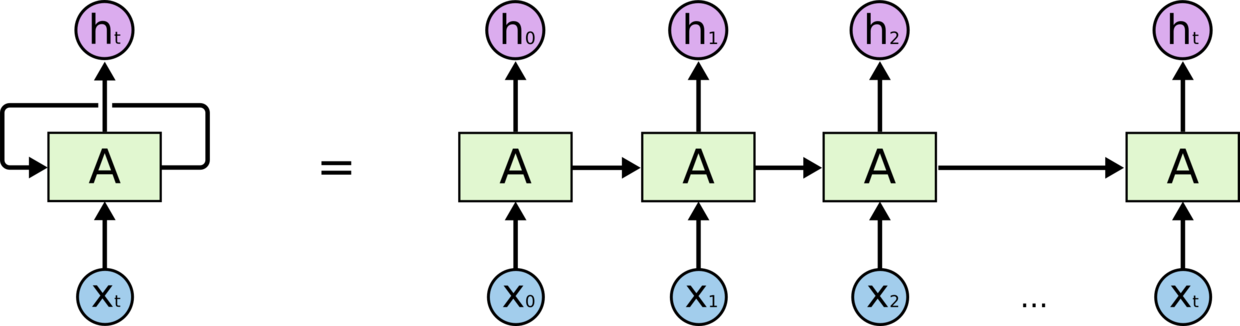
考虑这样一个问题，如果预测句子的下一个单词是什么，一般要用到当前单词和前面的单词。而DNN是解决不了这个问题的，于是，就出现了另一种神经网络结构——循环神经网络RNN。

RNN中，一个重要的概念就是时刻，RNN会对每一个时刻的输入结合当前模型的状态给出一个输出。它的一个典型细胞结构如下：

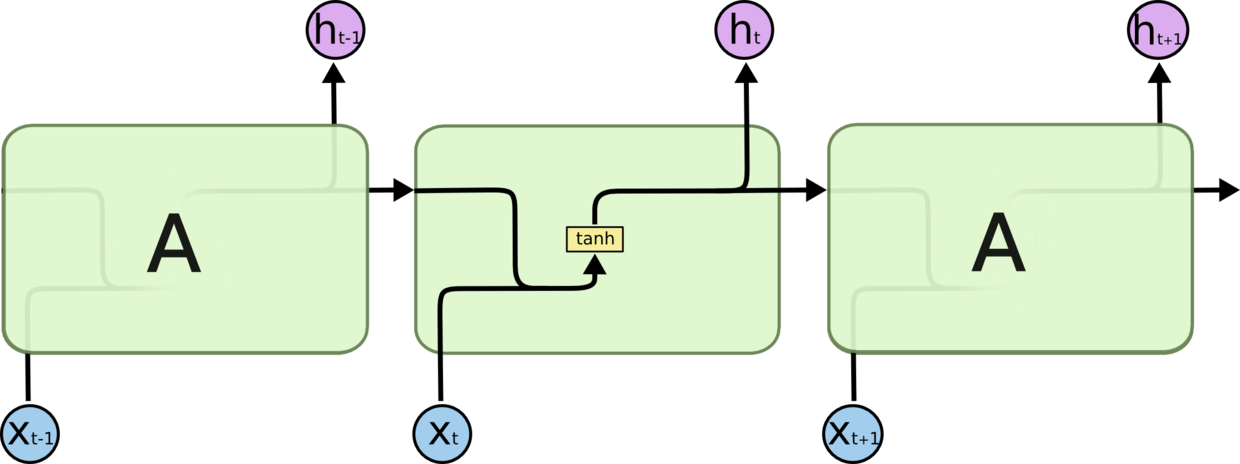


xt为t时刻的输入，ht为t时刻的输出，A为状态，用来传递给下一个时刻的模型。(ht不一定等于A,可以是ht=f(A) )

展开就是：



标准的RNN结构：



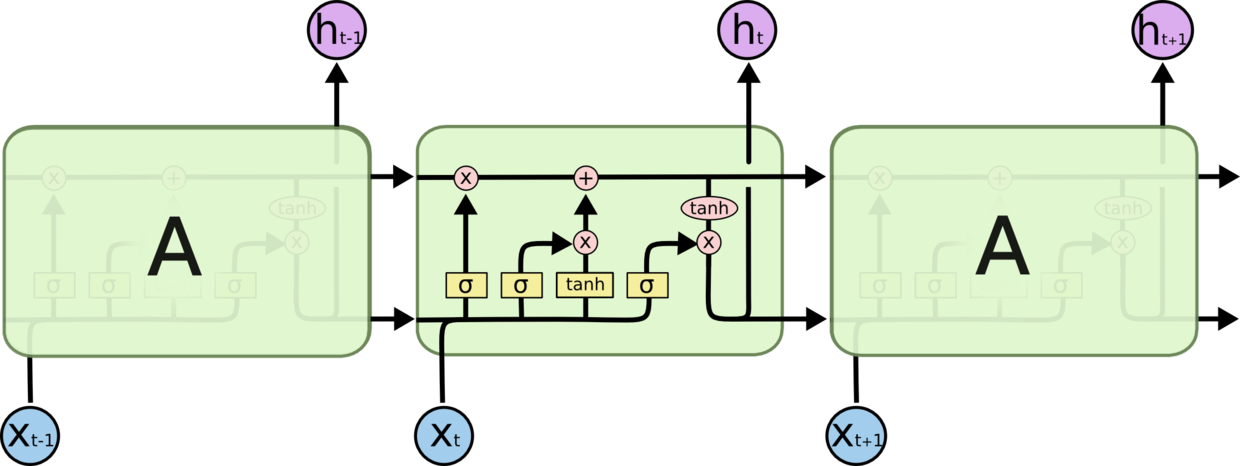
## LTSM

具体参考：理解 LSTM 网络（http://www.jianshu.com/p/9dc9f41f0b29）

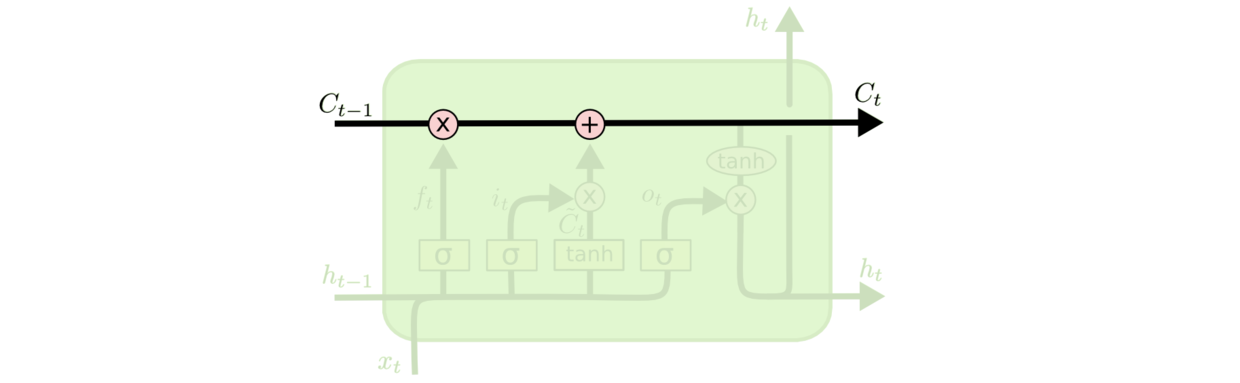
RNN理论上可以学习任意长度的序列，但实际中，序列过长会导致梯度消失的问题。所以一般会规定一个最大长度，进行截断。

“某地开始大量化工厂，空气污染十分严重，这里的天空是\_”,灰色还是蓝色？如果只是看“天空是”,则应该预测为“蓝色”，但结合前面的意思，则应该预测为“灰色”。上面单一的tanh结构容易丧失对位置较远信息的学习能力。而LSTM则在实际表现中要好于简单的RNN.

结构：



粉色的圈代表 pointwise 的操作，如乘法，加法，而黄色的矩阵代表神经网络层。



Ct表示t时刻的状态， ht表示输出， xt表示输入。

LSTM的设计思路：

* sigmoid函数，具有“门”的作用，要实现哪些信息被保留下来，哪些信息被忘记，通过 乘以 sigmoid的输出值(0,1)就能实现。LSTM有3个“门”
* 忘记门。决定忘记C[t-1]中哪些信息。实现： C[t-1] \* sigmoid(h[t-1],x[t])。（乘法操作是图中粉红色的圆圈）。 sigmoid的输入是( h[t-1]和x[t]),训练参数是w,b 。
* 输入门。决定哪些新的信息被加入到新的状态中。C[t]=C[t-1]+tanh(h[t-1],x[t]) \*sigmoid(h[t-1],x[t])
* 输出门。决定哪些信息被输出。C[t] \* sigmoid(h[t-1],x[t])

## 变种

RNN有很多变种，如多重神经网络（MultiRNN）和双向循环网络(bidirection RNN)。

略。

另一个改动较大的变体是 Gated Recurrent Unit (GRU)，这是由 Cho, et al. (2014) 提出。它将忘记门和输入门合成了一个单一的更新门，也就是由3个门变成2个门。同样还混合了细胞状态和隐藏状态，和其他一些改动。最终的模型比标准的 LSTM 模型要简单，也是非常流行的变体

# GAN

# DQN