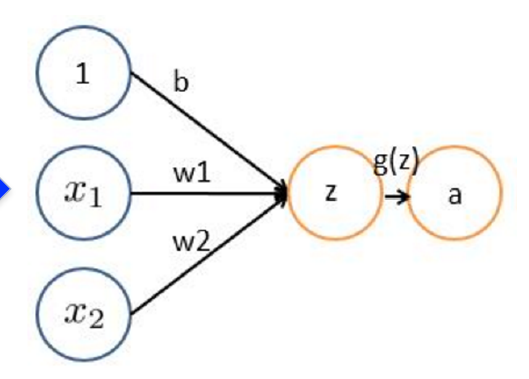
# 神经网络

## 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：

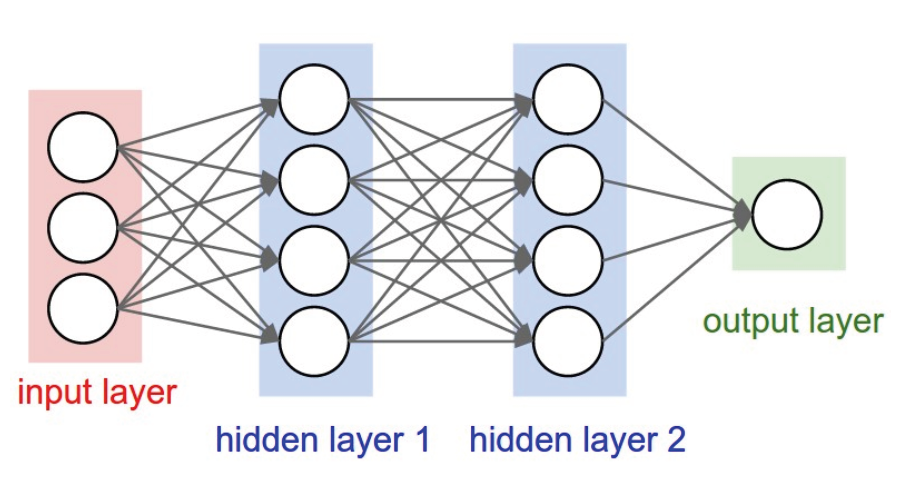




x是输入特征，w是权重，g是激活函数。如果g是sigmoid函数，这就是一个逻辑回归。

## 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。

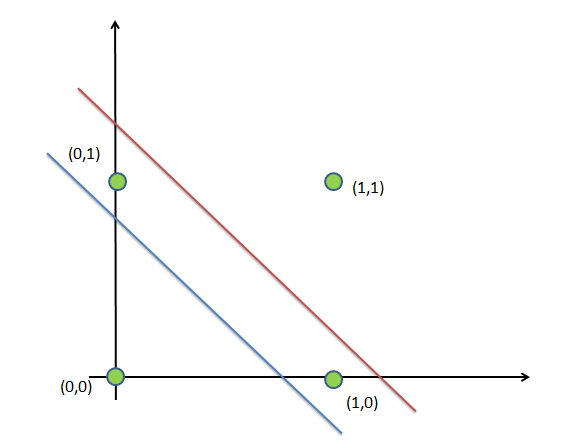
## 神经网络的能力

**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：



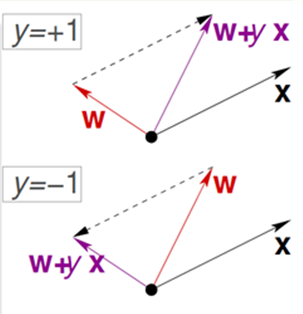
即



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

**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。

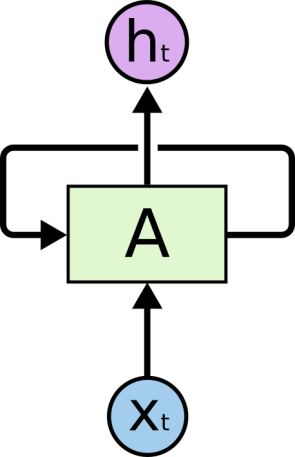
# RNN

## RNN

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

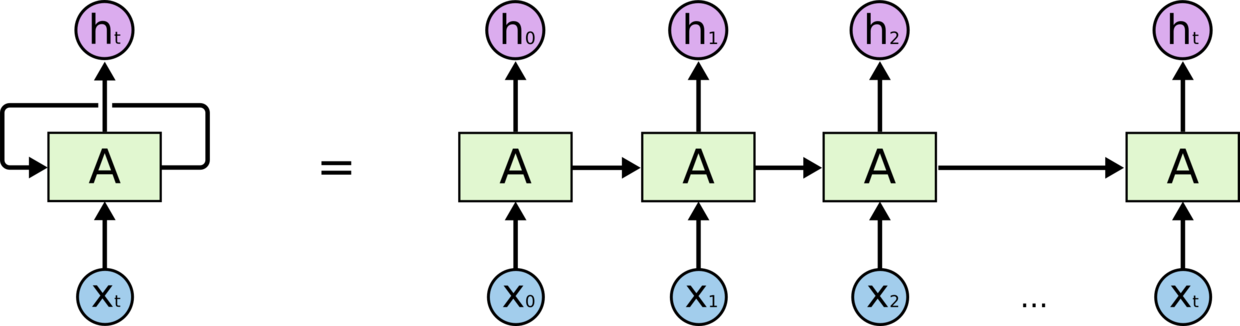
考虑这样一个问题，如果预测句子的下一个单词是什么，一般要用到当前单词和前面的单词。而DNN是解决不了这个问题的，于是，就出现了另一种神经网络结构——循环神经网络RNN。

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：

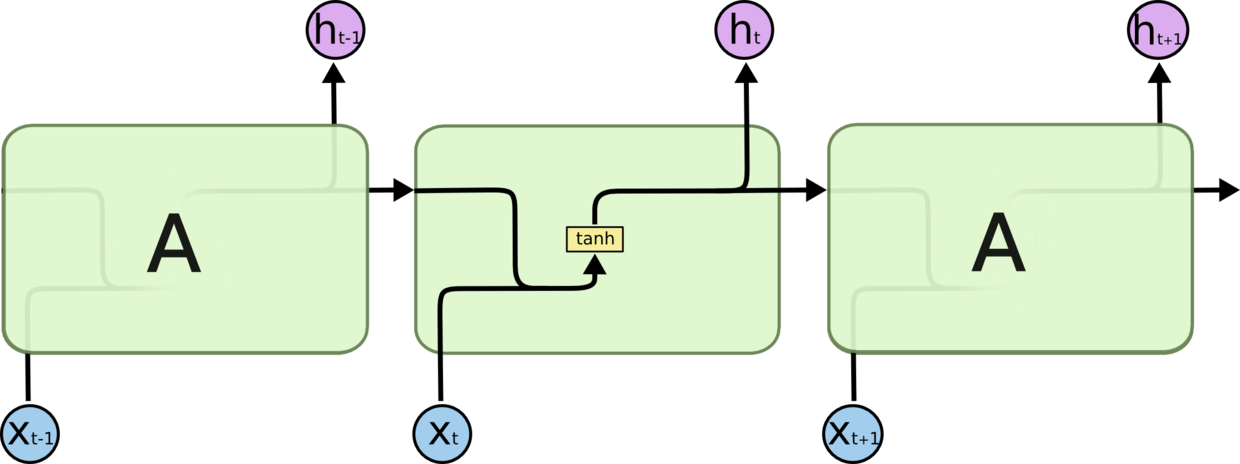


xt为t时刻的输入，ht为t时刻的输出，A为状态，用来传递给下一个时刻的模型。(ht不一定等于A,可以是ht=f(A) )

展开就是：



标准的RNN结构：



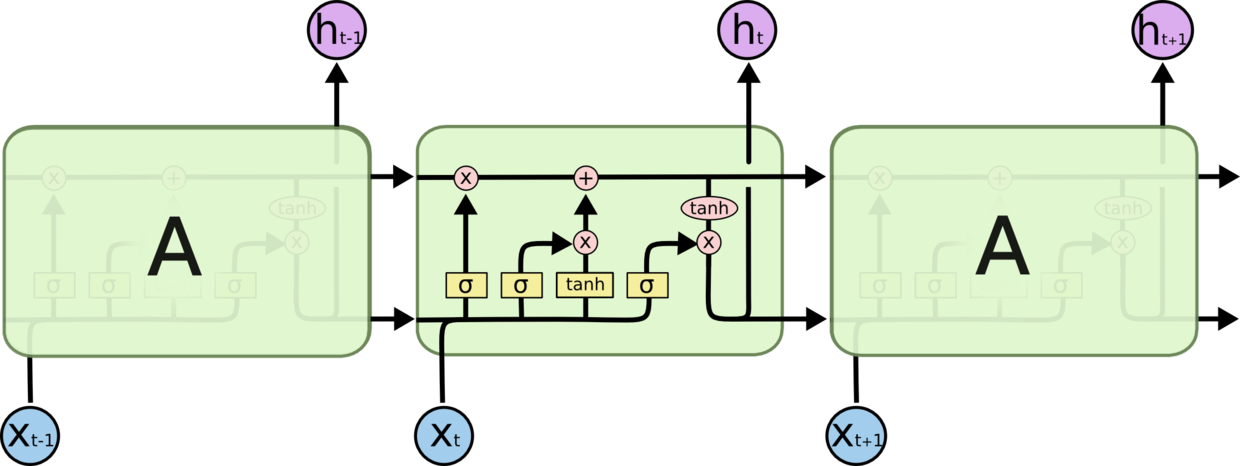
## LTSM

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

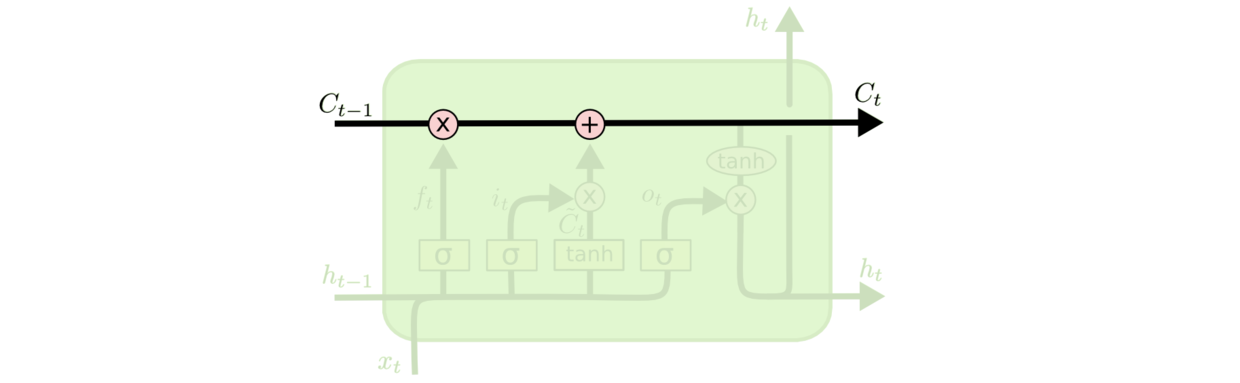
RNN理论上可以学习任意长度的序列，但实际中，序列过长会导致梯度消失的问题。所以一般会规定一个最大长度，进行截断。

“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