# 环境搭建篇

# Windows+anaconda

## 安装anaconda

下载windows版本下的anaconda，下载地址：

<https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/Anaconda3-5.2.0-Windows-x86_64.exe>，然后安装，注意安装过程中选择“加入到系统PATH环境变量”。

## 安装cuda和cudnn

CUDA(Compute Unified Device Architecture)，是英伟达公司推出的一种基于新的并行编程模型和指令集架构的通用计算架构，它能利用英伟达GPU的并行计算引擎，比CPU更高效的解决许多复杂计算任务。

NVIDIA cuDNN是用于深度神经网络的GPU加速库。它强调性能、易用性和低内存开销。NVIDIA cuDNN可以集成到更高级别的机器学习框架中，如加州大学伯克利分校的流行CAFFE软件。简单的，插入式设计可以让开发人员专注于设计和实现神经网络模型，而不是调整性能，同时还可以在GPU上实现高性能现代并行计算。

**检查GPU是否支持CUDN​和tensorflow：**

你基本只要确认你的GPU是不是NVIDIA的就可以了，当然如果你还是不确定的话，也可以去这个网站(https://developer.nvidia.com/cuda-gpus)中的“CUDA-Enabled GeForce Products”查一下，只要你的显卡型号在里面，另外：tensorflow-gpu requires GPUs of compute capability 3.0 or higher for GPU acceleration and this has been true since the very first release of tensorflow.

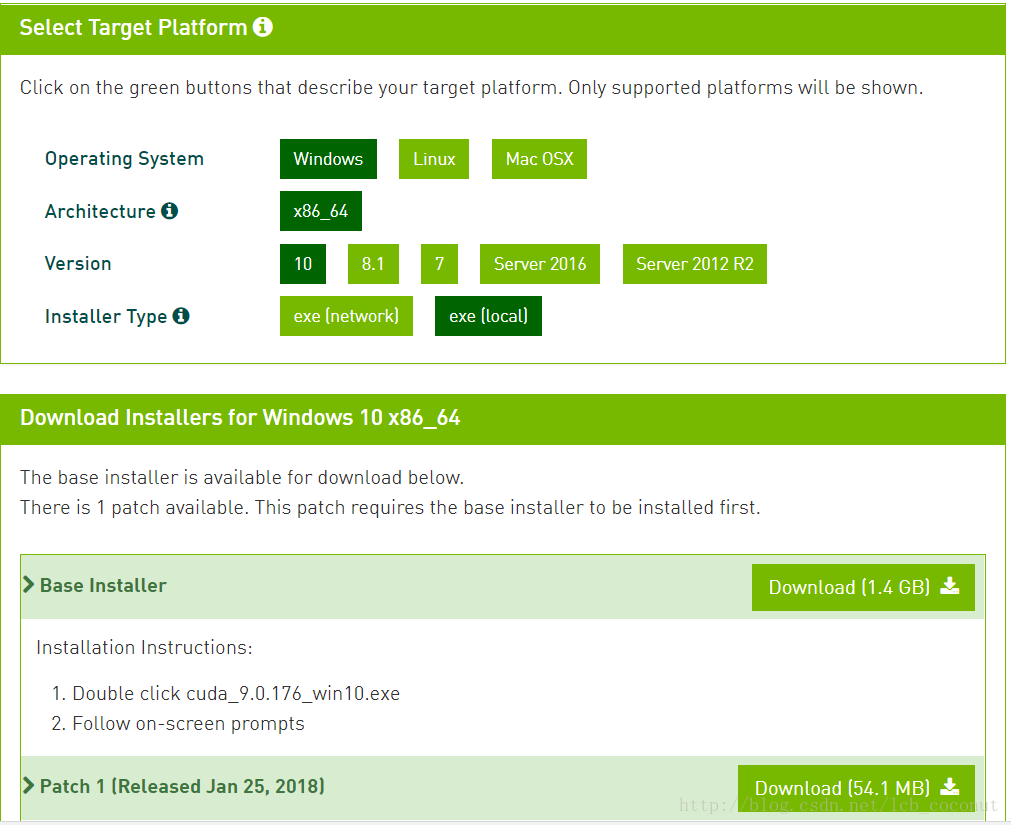
**安装**VS2015：



**安装cuda9.0：**

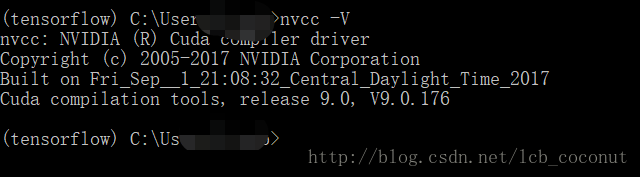
目前tensorflow还不支持cuda9.0以上的版本，下载地址：

https://developer.nvidia.com/cuda-90-download-archive?target\_os=Windows&target\_arch=x86\_64&target\_version=10&target\_type=exelocal

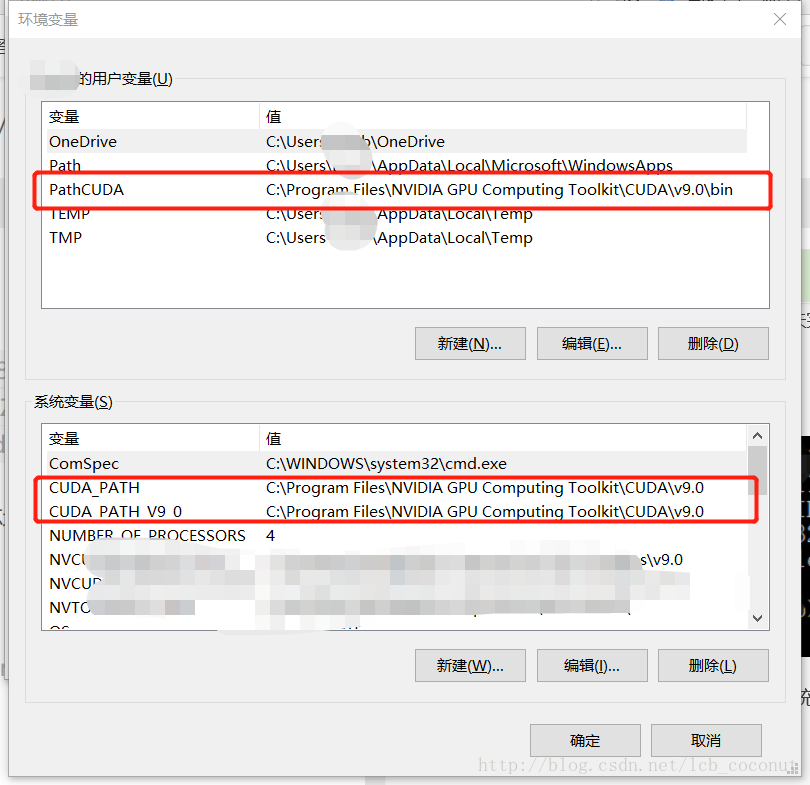


使用管理员权限安装，一路默认。

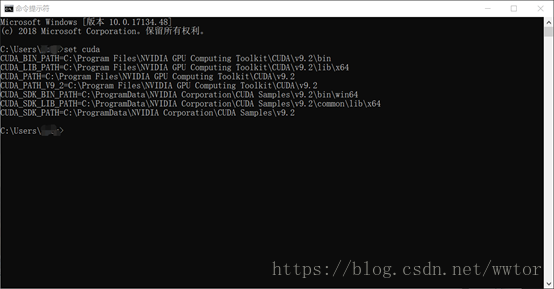
安装完成之后打开命令行，输入：nvcc -V查看版本，如果出现以下类似信息表示安装成功。

****

安装成功后，我的电脑上点右键，打开属性->高级系统设置->环境变量



只需要做两件事情：   
（1）确认系统变量中CUDA\_PATH和CUDA\_PATH\_V9.0已经存在   
（2）在用户变量，新建PathCUDA：C:\ProgramData\NVIDIA GPU Computing Toolkit\v9.0\bin

****

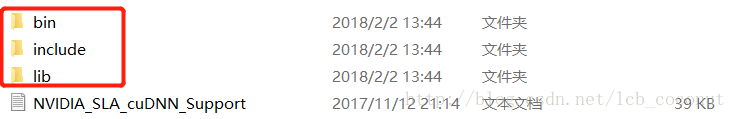
**安装cudnn：**

https://developer.nvidia.com/rdp/cudnn-download选择对应的版本下载，





解压cudnn-9.0-windows10-x64-v7，将文件夹里如下图所示的三个文件夹分别拷贝至CUDA的安装目录的对应的文件夹即可。默认文件夹在： C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0

****

## 安装tensorflow

### 安装CPU版本

在anaconda promt 下执行：Conda install tensorflow 会安装tensorflow及tensorboard

# 验证TensorFlow是否安装成功

>>> import tensorflow as tf

>>> hello = tf.constant('Hello, TensorFlow!')

>>> sess = tf.Session()

>>> print sess.run(hello)

Hello, TensorFlow! # 恭喜！安装成功！

### 安装GPU版本

**安装tensorflow-gpu**

注：windows下tensorflow只支持python3，不支持python2，如果要跑python2下的程序，只能在linux下。

新建一个环境，安装GPU版本的tensorflow。

1. 创建环境conda create -n gpu python=3.6
2. 激活环境source activate gpu
3. 安装一些基本的lib conda install numpy
4. 安装tensorflow conda install tensorflow-gpu

在python环境import tensorflow测试一下

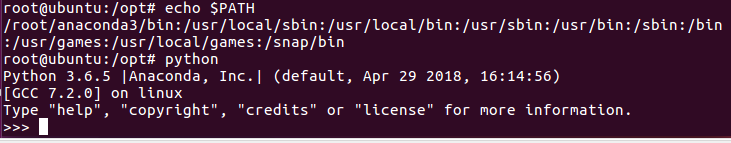
# ubuntu+anaconda

## 前期安装

安装VMware，然后安装ubuntu16.04。

## 安装anaconda

下载windows版本下的anaconda，下载地址：[https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/Anaconda3-5.2.0-Linux-x86\_64.sh](https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/下载Anaconda3-5.2.0-Linux-x86_64.sh)  复制到ubuntu。在安装包所在目录执行命令，按Enter继续。过程中有指定安装目录，不指定的话会安装到默认路径下。提示是否要将Anaconda的安装路径添加到PATH环境变量中，输入yes就好了。检查一下：



## 安装tensorflow

### 安装CPU版本

Conda install tensorflow 会安装tensorflow及tensorboard

# 验证TensorFlow是否安装成功

>>> import tensorflow as tf

>>> hello = tf.constant('Hello, TensorFlow!')

>>> sess = tf.Session()

>>> print sess.run(hello)

Hello, TensorFlow! # 恭喜！安装成功！

### 安装GPU版本

虚拟机不支持GPU。

## 安装python 2.7环境

由于有些例程是运行在python 2.7里面，如果直接运行在python3.6环境会报兼容错误，因此准备搭建一个python2.7的环境。

1. 创建环境conda create -n py27 python=2.7
2. 激活环境source activate py27
3. 安装一些基本的lib conda install numpy
4. 安装tensorflow conda install tensorflow

# Conda 命令

1. **管理conda和anaconda**  
   conda info 查询conda信息  
   conda update conda 升级conda  
   conda update anaconda 升级anaconda
2. **管理环境**  
   conda info -e 环境信息  
   conda create -n test python=2.7 创建环境test，并指定python版本，此例为2.7  
   source activate test 激活环境  
   source deactivate test 关闭环境  
   conda remove --name test --all 删除环境
3. **包管理**  
   conda list 列出所有安装的包的信息  
   conda search beautiful-soup 查询包  
   conda install -n test beautiful-soup=x.x.x 安装包，并指定安装环境，如果没有-n test，则安装到当前环境  
   conda update beautiful-soup 升级包  
   conda remove -n test beautiful-soup 移除包
4. Conda和pip区别

Conda是一个包管理器；Anaconda才是一个python发行版。

* 软件发行版是在系统上提前编译和配置好的软件包集合， 装好了后就可以直接用。
* 包管理器是自动化软件安装，更新，卸载的一种工具。Conda，有命令”conda install”, “conda update”, “conda remove”, 所以很明显， conda是包管理器。
* conda和Anaconda名字相似，但没有必然关系， 你可以不安装Anaconda的同时， 使用conda安装和管理软件。
* Conda是一个通用的包管理器，当初设计来管理任何语言的包。所以用来管理python包当然也是绰绰有余。
* Conda 和 pip 目标并不相同， 只有小部分子集有交集有竞争关系：比如python包的安装和环境隔离。
* pip可以允许你在任何环境中安装python包，而conda允许你在conda环境中安装任何语言包（包括c语言或者python）。

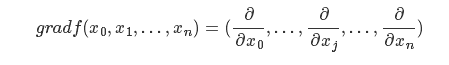
注意：conda 和 pip 之类包管理或者 virtualenv 之类虚拟环境管理软件不要一同使用。用conda就只用conda，否则容易出错，切记。

# 机器学习篇

# 梯度下降算法

## 梯度

梯度的定义如下：



梯度的提出只为回答一个问题： 函数在变量空间的某一点处，沿着哪一个方向有最大的变化率？梯度定义如下： 函数在某一点的梯度是这样一个向量，它的方向与取得最大方向导数的方向一致，而它的模为方向导数的最大值。 这里注意三点：

1）梯度是一个向量，即有方向有大小；

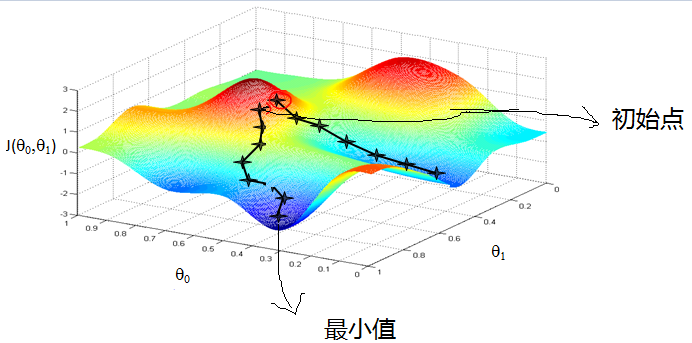
2）梯度的方向是最大方向导数的方向；

3）梯度的值（这里指的是模，值有点歧义）是某方向上最大方向导数（的值，导数本身就是值）

## 梯度下降数学解释

首先来看看梯度下降的一个直观的解释。比如我们在一座大山上的某处位置，由于我们不知道怎么下山，于是决定走一步算一步，也就是在每走到一个位置的时候，求解当前位置的梯度，沿着梯度的负方向，也就是当前最陡峭的位置向下走一步，然后继续求解当前位置梯度，向这一步所在位置沿着最陡峭最易下山的位置走一步。这样一步步的走下去，一直走到觉得我们已经到了山脚。当然这样走下去，有可能我们不能走到山脚，而是到了某一个局部的山峰低处。

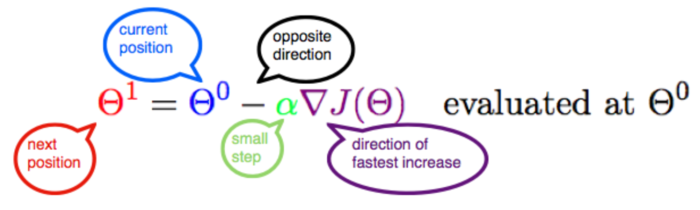
从上面的解释可以看出，梯度下降不一定能够找到全局的最优解，有可能是一个局部最优解。当然，如果损失函数是凸函数，梯度下降法得到的解就一定是全局最优解。



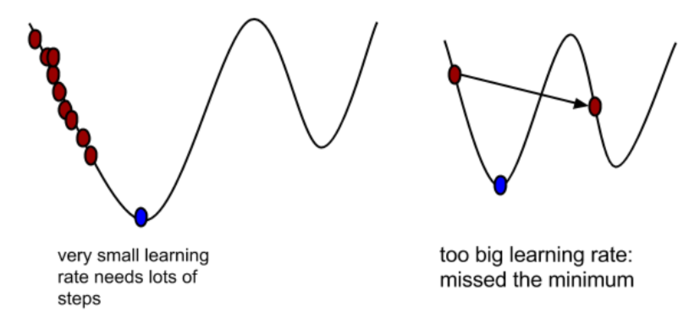
下面我们就开始从数学上解释梯度下降算法的计算过程和思想！



此公式的意义是：J是关于Θ的一个函数，我们当前所处的位置为Θ0点，要从这个点走到J的最小值点，也就是山底。首先我们先确定前进的方向，也就是梯度的反向，然后走一段距离的步长，也就是α，走完这个段步长，就到达了Θ1这个点！



下面就这个公式的几个常见的疑问：α是什么含义？α在梯度下降算法中被称作为**学习率**或者**步长**，意味着我们可以通过α来控制每一步走的距离，以保证不要步子跨的太大扯着蛋，哈哈，其实就是不要走太快，错过了最低点。同时也要保证不要走的太慢，导致太阳下山了，还没有走到山下。所以α的选择在梯度下降法中往往是很重要的！α不能太大也不能太小，太小的话，可能导致迟迟走不到最低点，太大的话，会导致错过最低点！



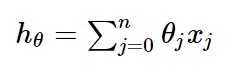
为什么要梯度要乘以一个负号？梯度前加一个负号，就意味着朝着梯度相反的方向前进！我们在前文提到，梯度的方向实际就是函数在此点上升最快的方向，而我们需要朝着下降最快的方向走，自然就是负的梯度的方向，所以此处需要加上负号。

## 梯度下降算法

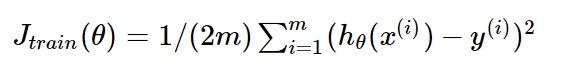
在应用机器学习算法时，我们通常采用梯度下降法来对采用的算法进行训练。其实，常用的梯度下降法还具体包含有三种不同的形式，它们也各自有着不同的优缺点。

下面我们以线性回归算法来对三种梯度下降法进行比较。

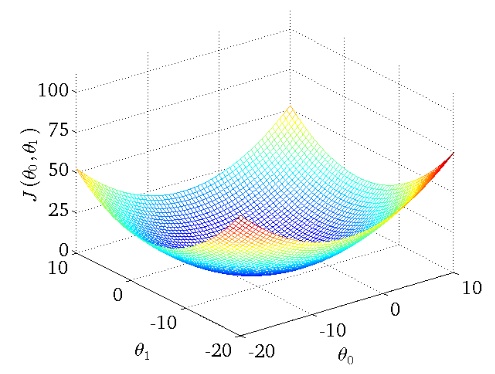
一般线性回归函数的假设函数为：



对应的损失函数为：



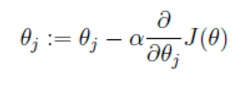
（这里的1/2是为了后面求导计算方便）,下图作为一个二维参数组对应能量函数的可视化图：



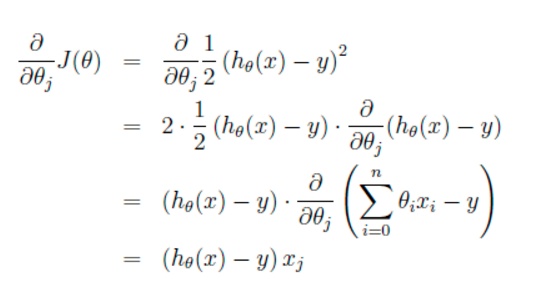
下面我们来分别讲解三种梯度下降法

批量梯度下降法BGD

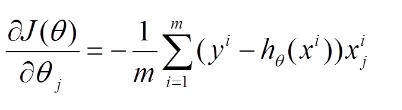
我们的目的是要误差函数尽可能的小，即求解weights使误差函数尽可能小。首先，我们随机初始化weigths，然后不断反复的更新weights使得误差函数减小，直到满足要求时停止。这里更新算法我们选择梯度下降算法，利用初始化的weights并且反复更新weights：



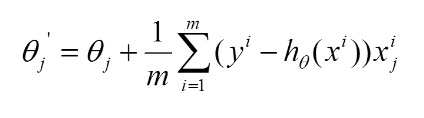
这里代表学习率，表示每次向着J最陡峭的方向迈步的大小。为了更新weights，我们需要求出函数J的偏导数。首先当我们只有一个数据点（x,y）的时候，J的偏导数是：



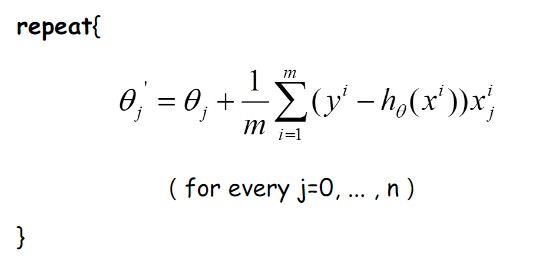
则对**所有数据点，**上述损失函数的偏导（**累和**）为：



再最小化损失函数的过程中，需要不断反复的更新weights使得误差函数减小，更新过程如下：

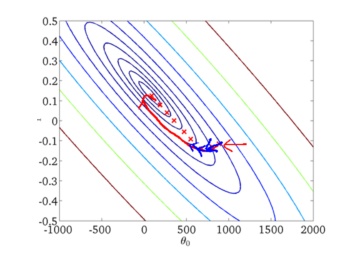


那么好了，每次参数更新的伪代码如下：



由上图更新公式我们就可以看到，我们每一次的参数更新都用到了所有的训练数据（比如有m个，就用到了m个），如果训练数据非常多的话，是非常耗时的。

下面给出批梯度下降的收敛图：



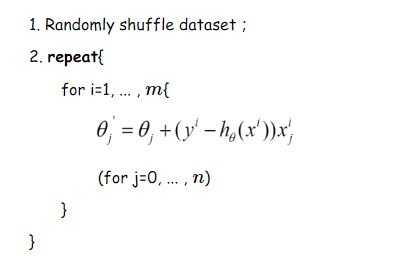
从图中，我们可以得到BGD迭代的次数相对较少。

随机梯度下降法SGD

由于批梯度下降每跟新一个参数的时候，要用到所有的样本数，所以训练速度会随着样本数量的增加而变得非常缓慢。随机梯度下降正是为了解决这个办法而提出的。它是利用每个样本的损失函数对θ求偏导得到对应的梯度，来更新θ：

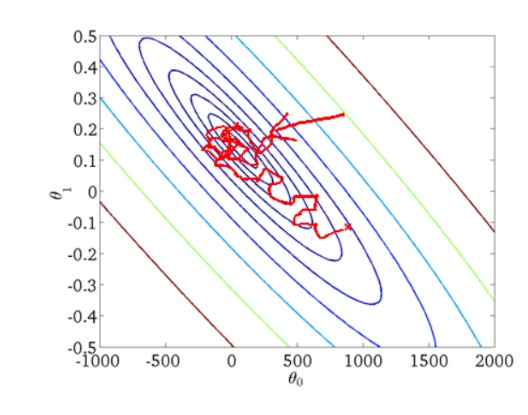
https://pic2.zhimg.com/v2-83b82dbf1a56566da84fe3eded3c9385_b.jpg

更新过程如下：



随机梯度下降是通过每个样本来迭代更新一次，对比上面的批量梯度下降，迭代一次需要用到所有训练样本（往往如今真实问题训练数据都是非常巨大），一次迭代不可能最优，如果迭代10次的话就需要遍历训练样本10次。但是，SGD伴随的一个问题是噪音较BGD要多，使得SGD并不是每次迭代都向着整体最优化方向。

随机梯度下降收敛图如下：



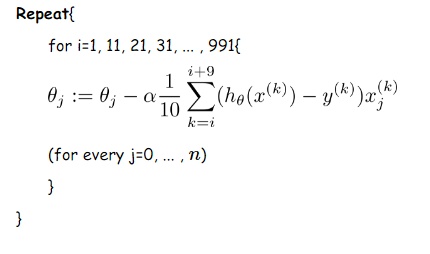
我们可以从图中看出SGD迭代的次数较多，在解空间的搜索过程看起来很盲目。但是大体上是往着最优值方向移动。

min-batch 小批量梯度下降法MBGD

我们从上面两种梯度下降法可以看出，其各自均有优缺点，那么能不能在两种方法的性能之间取得一个折衷呢？即，算法的训练过程比较快，而且也要保证最终参数训练的准确率，而这正是小批量梯度下降法（Mini-batch Gradient Descent，简称MBGD）的初衷。

我们假设每次更新参数的时候用到的样本数为10个（不同的任务完全不同，这里举一个例子而已）

更新伪代码如下：



# 线性回归（回归算法）

## 假设函数

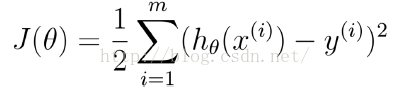
数学形式：



矩阵形式：



## 损失函数

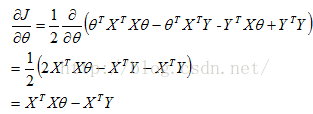


当矩阵θ可逆（满秩）时，通过normal equation可以直接求解

目标函数转化为矩阵形式：



对其求导并求驻点。



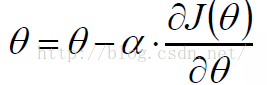
另上式为0，可求得：



此算法的缺点是：当矩阵很大是，计算非常耗时且占用资源

3.2 当矩阵θ不可逆（非满秩）时，通过梯度下降求解

初始化θ，沿着负梯度方向进行迭代，直到θ变化很小或者不变化



## sklearn实现

使用sklearn.linear\_model.LinearRegression进行线性回归

sklearn对Data Mining的各类算法已经有了较好的封装，基本可以使用fit、predict、score来训练、评价模型，并使用模型进行预测，一个简单的例子如下：

from sklearn import linear\_model

clf = linear\_model.LinearRegression()

X = [[0,0],[1,1],[2,2]]

y = [0,1,2]

clf.fit(X,y)

print(clf.coef\_)

[ 0.5 0.5]

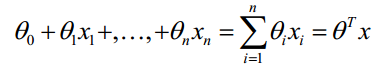
print(clf.intercept\_)

1.11022302463e-16

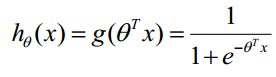
# 逻辑回归（分类算法）

## 假设函数

对于线性边界的情况，边界形式可以归纳为如下公式



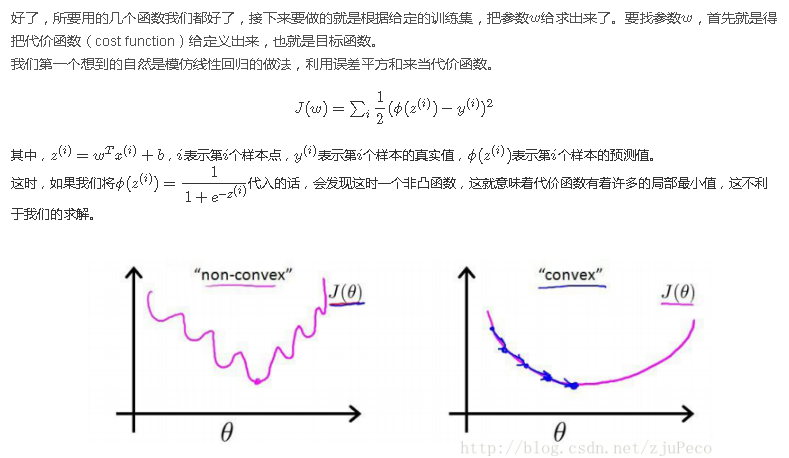
因此我们可以构造预测函数为如下公式



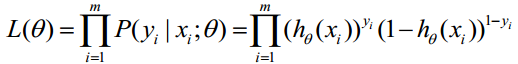
该预测函数表示分类结果为1时的概率。因此对于输入点`x`，分类结果为类别1和类别0的概率分别为如下公式

https://github.com/endymecy/spark-ml-source-analysis/raw/master/%E5%88%86%E7%B1%BB%E5%92%8C%E5%9B%9E%E5%BD%92/%E7%BA%BF%E6%80%A7%E6%A8%A1%E5%9E%8B/%E9%80%BB%E8%BE%91%E5%9B%9E%E5%BD%92/imgs/1.5.png

## 损失函数



对于训练数据集，特征数据`x={x1, x2, … , xm}`和对应的分类数据`y={y1, y2, … , ym}`。构建逻辑回归模型`f`，最典型的构建方法便是应用极大似然估计。对公式取极大似然函数，可以得到如下的公式



再对公式取对数，可得到

https://github.com/endymecy/spark-ml-source-analysis/raw/master/%E5%88%86%E7%B1%BB%E5%92%8C%E5%9B%9E%E5%BD%92/%E7%BA%BF%E6%80%A7%E6%A8%A1%E5%9E%8B/%E9%80%BB%E8%BE%91%E5%9B%9E%E5%BD%92/imgs/1.7.png

再用梯度下降法求解。

## sklearn实现

LogisticRegression回归模型在Sklearn.linear\_model子类下，调用sklearn逻辑回归算法步骤比较简单，即：

(1) 导入模型。调用逻辑回归LogisticRegression()函数。

(2) fit()训练。调用fit(x,y)的方法来训练模型，其中x为数据的属性，y为所属类型。

(3) predict()预测。利用训练得到的模型对数据集进行预测，返回预测结果。

代码如下：

from sklearn.linear\_model import LogisticRegression *#导入逻辑回归模型*

clf = LogisticRegression()

print clf

clf.fit(train\_feature,label)

predict['label'] = clf.predict(predict\_feature)

输出结果如下：

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, max\_iter=100, multi\_class='ovr', n\_jobs=1,

penalty='l2', random\_state=None, solver='liblinear', tol=0.0001,

verbose=0, warm\_start=False)

其中，参数penalty表示惩罚项（L1、L2值可选。L1向量中各元素绝对值的和，作用是产生少量的特征，而其他特征都是0，常用于特征选择；L2向量中各个元素平方之和再开根号，作用是选择较多的特征，使他们都趋近于0。）； C值的目标函数约束条件：s.t.||w||1<C，默认值是0，C值越小，则正则化强度越大。

### 正则化选择参数：penalty

LogisticRegression和LogisticRegressionCV默认就带了正则化项。penalty参数可选择的值为"l1"和"l2".分别对应L1的正则化和L2的正则化，默认是L2的正则化。

在调参时如果我们主要的目的只是为了解决过拟合，一般penalty选择L2正则化就够了。但是如果选择L2正则化发现还是过拟合，即预测效果差的时候，就可以考虑L1正则化。另外，如果模型的特征非常多，我们希望一些不重要的特征系数归零，从而让模型系数稀疏化的话，也可以使用L1正则化。

penalty参数的选择会影响我们损失函数优化算法的选择。即参数solver的选择，如果是L2正则化，那么4种可选的算法{‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’}都可以选择。但是如果penalty是L1正则化的话，就只能选择‘liblinear’了。这是因为L1正则化的损失函数不是连续可导的，而{‘newton-cg’, ‘lbfgs’,‘sag’}这三种优化算法时都需要损失函数的一阶或者二阶连续导数。而‘liblinear’并没有这个依赖。

### 优化算法选择参数：solver

solver参数决定了我们对逻辑回归损失函数的优化方法，有4种算法可以选择，分别是：

a) liblinear：使用了开源的liblinear库实现，内部使用了坐标轴下降法来迭代优化损失函数。

b) lbfgs：拟牛顿法的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。

c) newton-cg：也是牛顿法家族的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。

d) sag：即随机平均梯度下降，是梯度下降法的变种，和普通梯度下降法的区别是每次迭代仅仅用一部分的样本来计算梯度，适合于样本数据多的时候，SAG是一种线性收敛算法，这个速度远比SGD快。关于SAG的理解，参考博文[线性收敛的随机优化算法之 SAG、SVRG（随机梯度下降）](http://blog.csdn.net/sun_shengyun/article/details/53811882)

从上面的描述可以看出，newton-cg, lbfgs和sag这三种优化算法时都需要损失函数的一阶或者二阶连续导数，因此不能用于没有连续导数的L1正则化，只能用于L2正则化。而liblinear通吃L1正则化和L2正则化。

同时，sag每次仅仅使用了部分样本进行梯度迭代，所以当样本量少的时候不要选择它，而如果样本量非常大，比如大于10万，sag是第一选择。但是sag不能用于L1正则化，所以当你有大量的样本，又需要L1正则化的话就要自己做取舍了。要么通过对样本采样来降低样本量，要么回到L2正则化。

在sklearn的官方文档中，对于solver的使用说明如下：

In a nutshell, one may choose the solver with the following rules:

| Case | Solver |
| --- | --- |
| Small dataset or L1 penalty | “liblinear” |
| Multinomial loss or large dataset | “lbfgs”, “sag” or “newton-cg” |
| Very Large dataset | “sag” |

从上面的描述，大家可能觉得，既然newton-cg, lbfgs和sag这么多限制，如果不是大样本，我们选择liblinear不就行了嘛！错，因为liblinear也有自己的弱点！我们知道，逻辑回归有二元逻辑回归和多元逻辑回归。对于多元逻辑回归常见的有one-vs-rest(OvR)和many-vs-many(MvM)两种。而MvM一般比OvR分类相对准确一些。郁闷的是liblinear只支持OvR，不支持MvM，这样如果我们需要相对精确的多元逻辑回归时，就不能选择liblinear了。也意味着如果我们需要相对精确的多元逻辑回归不能使用L1正则化了。

总结而言，liblinear支持L1和L2，只支持OvR做多分类，“lbfgs”, “sag” “newton-cg”只支持L2，支持OvR和MvM做多分类。

具体OvR和MvM有什么不同我们下一节讲。

### 分类方式选择参数：multi\_class

multi\_class参数决定了我们分类方式的选择，有 ovr和multinomial两个值可以选择，默认是 ovr。

ovr即前面提到的one-vs-rest(OvR)，而multinomial即前面提到的many-vs-many(MvM)。如果是二元逻辑回归，ovr和multinomial并没有任何区别，区别主要在多元逻辑回归上。

OvR的思想很简单，无论你是多少元逻辑回归，我们都可以看做二元逻辑回归。具体做法是，对于第K类的分类决策，我们把所有第K类的样本作为正例，除了第K类样本以外的所有样本都作为负例，然后在上面做二元逻辑回归，得到第K类的分类模型。其他类的分类模型获得以此类推。

而MvM则相对复杂，这里举MvM的特例one-vs-one(OvO)作讲解。如果模型有T类，我们每次在所有的T类样本里面选择两类样本出来，不妨记为T1类和T2类，把所有的输出为T1和T2的样本放在一起，把T1作为正例，T2作为负例，进行二元逻辑回归，得到模型参数。我们一共需要T(T-1)/2次分类。

从上面的描述可以看出OvR相对简单，但分类效果相对略差（这里指大多数样本分布情况，某些样本分布下OvR可能更好）。而MvM分类相对精确，但是分类速度没有OvR快。

如果选择了ovr，则4种损失函数的优化方法liblinear，newton-cg, lbfgs和sag都可以选择。但是如果选择了multinomial,则只能选择newton-cg, lbfgs和sag了。

### 类型权重参数： class\_weight

class\_weight参数用于标示分类模型中各种类型的权重，可以不输入，即不考虑权重，或者说所有类型的权重一样。如果选择输入的话，可以选择balanced让类库自己计算类型权重，或者我们自己输入各个类型的权重，比如对于0,1的二元模型，我们可以定义class\_weight={0:0.9, 1:0.1}，这样类型0的权重为90%，而类型1的权重为10%。

如果class\_weight选择balanced，那么类库会根据训练样本量来计算权重。某种类型样本量越多，则权重越低，样本量越少，则权重越高。

sklearn的官方文档中，当class\_weight为balanced时，类权重计算方法如下：

n\_samples / (n\_classes \* np.bincount(y))，n\_samples为样本数，n\_classes为类别数量，np.bincount(y)会输出每个类的样本数，例如y=[1,0,0,1,1],则np.bincount(y)=[2,3]

那么class\_weight有什么作用呢？在分类模型中，我们经常会遇到两类问题：

第一种是误分类的代价很高。比如对合法用户和非法用户进行分类，将非法用户分类为合法用户的代价很高，我们宁愿将合法用户分类为非法用户，这时可以人工再甄别，但是却不愿将非法用户分类为合法用户。这时，我们可以适当提高非法用户的权重。

第二种是样本是高度失衡的，比如我们有合法用户和非法用户的二元样本数据10000条，里面合法用户有9995条，非法用户只有5条，如果我们不考虑权重，则我们可以将所有的测试集都预测为合法用户，这样预测准确率理论上有99.95%，但是却没有任何意义。这时，我们可以选择balanced，让类库自动提高非法用户样本的权重。

提高了某种分类的权重，相比不考虑权重，会有更多的样本分类划分到高权重的类别，从而可以解决上面两类问题。

当然，对于第二种样本失衡的情况，我们还可以考虑用下一节讲到的样本权重参数： sample\_weight，而不使用class\_weight。sample\_weight在下一节讲。

### 样本权重参数： sample\_weight

上一节我们提到了样本不失衡的问题，由于样本不平衡，导致样本不是总体样本的无偏估计，从而可能导致我们的模型预测能力下降。遇到这种情况，我们可以通过调节样本权重来尝试解决这个问题。调节样本权重的方法有两种，第一种是在class\_weight使用balanced。第二种是在调用fit函数时，通过sample\_weight来自己调节每个样本权重。

在scikit-learn做逻辑回归时，如果上面两种方法都用到了，那么样本的真正权重是class\_weight\*sample\_weight.

以上就是scikit-learn中逻辑回归类库调参的一个小结，还有些参数比如正则化参数C（交叉验证就是 Cs），迭代次数max\_iter等，由于和其它的算法类库并没有特别不同，这里不多累述了。

# softmax回归（分类算法）

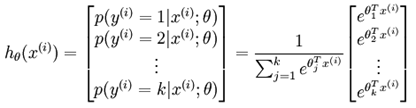
## 假设函数

给定一些数据，{（x1，y1），（x2，y2）…（xn，yn） }，x的值来预测y的值，通常地，y的值是连续的就是回归问题，y的值是离散的就叫分类问题。

logistic回归只能输出0和1，一般来说只能进行二分类问题，但是用one-vs-all的技巧也能进行多分类，softmax算是logistic回归的推广，它能直接处理多分类问题   
  softmax函数，或称归一化函数，是逻辑函数的一种推广，它能将一个含任意实数的K维向量压缩到另一个K维实向量中，使得每一个元素范围都在（0，1）之间，并且所有元素的和为1，这样就可以联系到概率上进行多分类问题的求解   
  softmax函数：



softmax回归假设函数

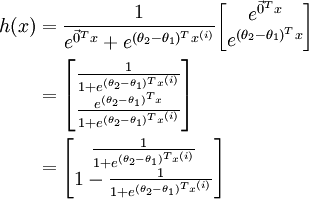


## Softmax回归与Logistic 回归的关系

当类别数 \textstyle k = 2 时，softmax 回归退化为 logistic 回归。这表明 softmax 回归是 logistic 回归的一般形式。具体地说，当 \textstyle k = 2 时，softmax 回归的假设函数为：

\begin{align}h_\theta(x) &=\frac{1}{ e^{\theta_1^Tx}  + e^{ \theta_2^T x^{(i)} } }\begin{bmatrix}e^{ \theta_1^T x } \\e^{ \theta_2^T x }\end{bmatrix}\end{align}

利用softmax回归参数冗余的特点，我们令 \textstyle \psi = \theta_1，并且从两个参数向量中都减去向量 \textstyle \theta_1，得到:



因此，用 \textstyle \theta'来表示\textstyle \theta_2-\theta_1，我们就会发现 softmax 回归器预测其中一个类别的概率为 \textstyle \frac{1}{ 1  + e^{ (\theta')^T x^{(i)} } }，另一个类别概率的为 \textstyle 1 - \frac{1}{ 1 + e^{ (\theta')^T x^{(i)} } }，这与 logistic回归是一致的。

## 损失函数

用极大似然法构造损失函数，用梯度下降法求解。

http://deeplearning.stanford.edu/wiki/images/math/7/6/3/7634eb3b08dc003aa4591a95824d4fbd.png

## 例子



# 决策树和随机森林（分类算法）

## 熵

在信息学和概率统计中，熵（entropy）是表示随机变量不确定性的度量。设X是一个取有限个值得离散随机变量，其概率分布为：

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\633828474.bmp

则随机变量X的熵定义为：

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\188496632.bmp

通常上式中对数以2，或者e为底。由定义知，熵依赖于X的分布，而于X的取值无关，所以X的熵记作H(p)，即：

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\757631954.bmp

熵越大，随机变量的不确定性就越大

## 条件熵

设有随机变量(X,Y)，其联合概率分布为：

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\3509089244.bmp

条件熵表示H(Y|X)在已知随机变量X的条件下随机变量Y的不确定性，定义为：

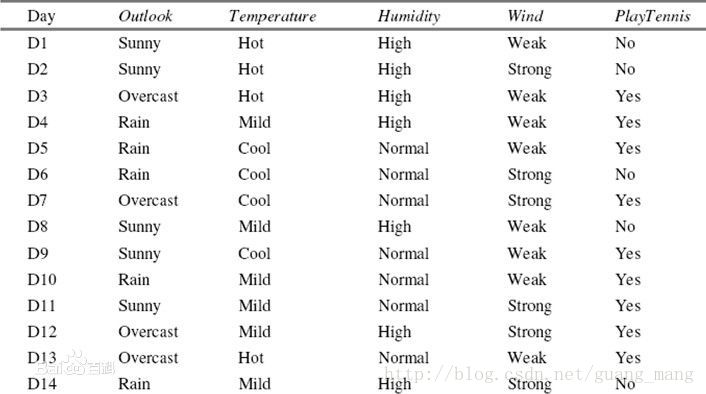
C:\Users\Administrator\AppData\Roaming\feiq\RichOle\2885684507.bmp

## 信息增益

信息增益表示得知特征A的信息而使得类Y的信息的不确定性减少的程度。特征A对训练数据集D的信息增益g(D,A)，定义为集合D的经验熵与特征A给定条件下D的经验条件熵之差，即：

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\470897191.bmp

计算过程通过下面例子来展现出来



现在还没有划分数据集，计算信息熵按照公式为

Entropy(S)=-9/14\*log2（9\14）-9/14\*log2（9\14）

当Wind固定为Weak时：记录有8条，其中yes为6个，NO为2个；同样，取值为Strong的记录6个，正例负例个3个。我们可以计算相应的熵为：

Entropy(Weak)=-(6/8)\*log(6/8)-(2/8)\*log(2/8)=0.811

Entropy(Strong)=-(3/6)\*log(3/6)-(3/6)\*log(3/6)=1.0

现在就可以计算出相应的信息增益了：所以，对于一个Wind属性固定的分类系统的信息量为

(8/14)\*Entropy(Weak)+(6/14)\*Entropy(Strong)

然后信息增益为

Gain(Wind)=Entropy(S)-(8/14)\*Entropy(Weak)-(6/14)\*Entropy(Strong)=0.940-(8/14)\*0.811-(6/14)\*1.0=0.048

其他特征也是这样计算，选择信息增益最大的特征作为根节点，依此类推。

## 剪枝

决策树对训练属于有很好的分类能力，但是对于未知的测试集未必有好的分类能力，泛化能力弱，即可能发生过拟合现象。为防止过拟合，我们需要进行剪枝。三种决策树的剪枝过程算法相同，区别是对于当前树的评价标准不同。

剪枝分为预剪枝和后剪枝：

预剪枝：

（1）每一个结点所包含的最小样本数目，例如10，则该结点总样本数小于10时，则不再分；

（2）指定树的高度或者深度，例如树的最大深度为4；

（3）指定结点的熵小于某个值，不再划分。

（4）叶子节点个数

后剪枝：

## 决策树的优缺点

决策树的优势

1. 简单易用，而且输出的结果易于解释，树能够被图形化，加深了直观的理解。
2. 几乎不需要对数据进行预处理。
3. 算法的开销不大，而且决策树一旦建立，对于未知样本的分类十分快，最坏情况下的时间复杂度是O(w),w是树的最大深度。
4. 能够用于多类的分类。
5. 能够容忍噪点。

决策树的劣势

1. 容易过拟合。
2. 容易被类别中占多数的类影响而产生bias，所以推荐在送入算法之间先平衡下数据中各个类别所占的比例。
3. 决策树采用的是自顶向下的递归划分法，因此自定而下到了末端枝叶包含的数据量会很少，我们会依据很少的数据量取做决策，这样的决策是不具有统计意义的，这就是数据碎片的问题。

## Sklean实现及参数

scikit-learn决策树算法类库内部实现是使用了调优过的CART树算法，既可以做分类，又可以做回归。分类决策树的类对应的是DecisionTreeClassifier，而回归决策树的类对应的是DecisionTreeRegressor。两者的参数定义几乎完全相同，但是意义不全相同。下面就对DecisionTreeClassifier和DecisionTreeRegressor的重要参数做一个总结，重点比较两者参数使用的不同点和调参的注意点。

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | DecisionTreeClassifier | DecisionTreeRegressor | |
| 特征选择标准criterion | 可以使用"gini"或者"entropy"，前者代表基尼系数，后者代表信息增益。一般说使用默认的基尼系数"gini"就可以了，即CART算法。除非你更喜欢类似ID3, C4.5的最优特征选择方法。 | 可以使用"mse"或者"mae"，前者是均方差，后者是和均值之差的绝对值之和。推荐使用默认的"mse"。一般来说"mse"比"mae"更加精确。除非你想比较二个参数的效果的不同之处。 | |
| 特征划分点选择标准splitter | 可以使用"best"或者"random"。前者在特征的所有划分点中找出最优的划分点。后者是随机的在部分划分点中找局部最优的划分点。  默认的"best"适合样本量不大的时候，而如果样本数据量非常大，此时决策树构建推荐"random" | | |
| 划分时考虑的最大特征数max\_features | 可以使用很多种类型的值，默认是"None",意味着划分时考虑所有的特征数；如果是"log2"意味着划分时最多考虑log2Nlog2N个特征；如果是"sqrt"或者"auto"意味着划分时最多考虑N−−√N个特征。如果是整数，代表考虑的特征绝对数。如果是浮点数，代表考虑特征百分比，即考虑（百分比xN）取整后的特征数。其中N为样本总特征数。  一般来说，如果样本特征数不多，比如小于50，我们用默认的"None"就可以了，如果特征数非常多，我们可以灵活使用刚才描述的其他取值来控制划分时考虑的最大特征数，以控制决策树的生成时间。 | | |
| 决策树最大深max\_depth | 决策树的最大深度，默认可以不输入，如果不输入的话，决策树在建立子树的时候不会限制子树的深度。一般来说，数据少或者特征少的时候可以不管这个值。如果模型样本量多，特征也多的情况下，推荐限制这个最大深度，具体的取值取决于数据的分布。常用的可以取值10-100之间。 | | |
| 内部节点再划分所需最小样本数min\_samples\_split | 这个值限制了子树继续划分的条件，如果某节点的样本数少于min\_samples\_split，则不会继续再尝试选择最优特征来进行划分。 默认是2.如果样本量不大，不需要管这个值。如果样本量数量级非常大，则推荐增大这个值。我之前的一个项目例子，有大概10万样本，建立决策树时，我选择了min\_samples\_split=10。可以作为参考。 | | |
| 叶子节点最少样本数min\_samples\_leaf | 这个值限制了叶子节点最少的样本数，如果某叶子节点数目小于样本数，则会和兄弟节点一起被剪枝。 默认是1,可以输入最少的样本数的整数，或者最少样本数占样本总数的百分比。如果样本量不大，不需要管这个值。如果样本量数量级非常大，则推荐增大这个值。之前的10万样本项目使用min\_samples\_leaf的值为5，仅供参考。 | | |
| 叶子节点最小的样本权重和min\_weight\_fraction\_leaf | 这个值限制了叶子节点所有样本权重和的最小值，如果小于这个值，则会和兄弟节点一起被剪枝。 默认是0，就是不考虑权重问题。一般来说，如果我们有较多样本有缺失值，或者分类树样本的分布类别偏差很大，就会引入样本权重，这时我们就要注意这个值了。 | | |
| 最大叶子节点数max\_leaf\_nodes | 通过限制最大叶子节点数，可以防止过拟合，默认是"None”，即不限制最大的叶子节点数。如果加了限制，算法会建立在最大叶子节点数内最优的决策树。如果特征不多，可以不考虑这个值，但是如果特征分成多的话，可以加以限制，具体的值可以通过交叉验证得到。 | | |
| 类别权重class\_weight | 指定样本各类别的的权重，主要是为了防止训练集某些类别的样本过多，导致训练的决策树过于偏向这些类别。这里可以自己指定各个样本的权重，或者用“balanced”，如果使用“balanced”，则算法会自己计算权重，样本量少的类别所对应的样本权重会高。当然，如果你的样本类别分布没有明显的偏倚，则可以不管这个参数，选择默认的"None" | | 不适用于回归树 |
| 节点划分最小不纯度min\_impurity\_split | 这个值限制了决策树的增长，如果某节点的不纯度(基尼系数，信息增益，均方差，绝对差)小于这个阈值，则该节点不再生成子节点。即为叶子节点 。 | | |
| 数据是否预排序presort | 这个值是布尔值，默认是False不排序。一般来说，如果样本量少或者限制了一个深度很小的决策树，设置为true可以让划分点选择更加快，决策树建立的更加快。如果样本量太大的话，反而没有什么好处。问题是样本量少的时候，我速度本来就不慢。所以这个值一般懒得理它就可以了。 | | |

除了这些参数要注意以外，其他在调参时的注意点有：

1）当样本少数量但是样本特征非常多的时候，决策树很容易过拟合，一般来说，样本数比特征数多一些会比较容易建立健壮的模型

2）如果样本数量少但是样本特征非常多，在拟合决策树模型前，推荐先做维度规约，比如主成分分析（PCA），特征选择（Losso）或者独立成分分析（ICA）。这样特征的维度会大大减小。再来拟合决策树模型效果会好。

3）推荐多用决策树的可视化（下节会讲），同时先限制决策树的深度（比如最多3层），这样可以先观察下生成的决策树里数据的初步拟合情况，然后再决定是否要增加深度。

4）在训练模型先，注意观察样本的类别情况（主要指分类树），如果类别分布非常不均匀，就要考虑用class\_weight来限制模型过于偏向样本多的类别。

5）决策树的数组使用的是numpy的float32类型，如果训练数据不是这样的格式，算法会先做copy再运行。

6）如果输入的样本矩阵是稀疏的，推荐在拟合前调用csc\_matrix稀疏化，在预测前调用csr\_matrix稀疏化。

## 随机森林

 尽管有剪枝等等方法，一棵树的生成肯定还是不如多棵树，因此就有了随机森林，解决决策树泛化能力弱的缺点。（可以理解成三个臭皮匠顶过诸葛亮）

而同一批数据，用同样的算法只能产生一棵树，这时Bagging策略可以帮助我们产生不同的数据集。Bagging策略来源于bootstrap aggregation：从样本集（假设样本集N个数据点）中重采样选出Nb个样本（有放回的采样，样本数据点个数仍然不变为N），在所有样本上，对这n个样本建立分类器（ID3\C4.5\CART\SVM\LOGISTIC），重复以上两步m次，获得m个分类器，最后根据这m个分类器的投票结果，决定数据属于哪一类。随机森林在bagging的基础上更进一步：

1.  样本的随机：从样本集中用Bootstrap随机选取n个样本

2.  特征的随机：从所有属性中随机选取K个属性，选择最佳分割属性作为节点建立CART决策树（泛化的理解，这里面也可以是其他类型的分类器，比如SVM、Logistics）

3.  重复以上两步m次，即建立了m棵CART决策树

4.  这m个CART形成随机森林，通过投票表决结果，决定数据属于哪一类（投票机制有一票否决制、少数服从多数、加权多数）

关于调参：1.如何选取K，可以考虑有N个属性，取K=根号N

               2.最大深度（不超过8层）

               3.棵数

               4.最小分裂样本树

               5.类别比例

# 贝叶斯（分类算法）

## 贝叶斯学派与频率主义学派

抽象一点来讲，频率学派和贝叶斯学派对世界的认知有本质不同：频率学派认为世界是确定的，有一个本体，这个本体的真值是不变的，我们的目标就是要找到这个真值或真值所在的范围；而贝叶斯学派认为世界是不确定的，人们对世界先有一个预判，而后通过观测数据对这个预判做调整，我们的目标是要找到最优的描述这个世界的概率分布。

在对事物建模时，用θ表示模型的参数，**请注意，解决问题的本质就是求θ。**那么：

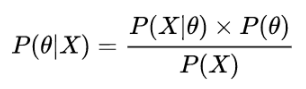
(1) 频率学派：存在唯一真值θ**。举一个简单直观的**例子—抛硬币，我们用P(head)来表示硬币的bias。抛一枚硬币100次，有20次正面朝上，要估计抛硬币正面朝上的bias P(head)=θ。在频率学派来看，θ= 20 / 100 = 0.2，很直观。

当数据量趋于无穷时，这种方法能给出精准的估计；然而缺乏数据时则可能产生严重的偏差。例如，对于一枚均匀硬币，即θ= 0.5，抛掷5次，出现5次正面 (这种情况出现的概率是1/2^5=3.125%)，频率学派会直接估计这枚硬币θ= 1，出现严重错误。

(2) 贝叶斯学派： θ是一个随机变量，符合一定的概率分布。在贝叶斯学派里有两大输入和一大输出，输入是先验 (prior)和似然 (likelihood)，输出是后验 (posterior)。

先验，即P(θ)，指的是在没有观测到任何数据时对θ的预先判断，例如给我一个硬币，一种可行的先验是认为这个硬币有很大的概率是均匀的，有较小的概率是是不均匀的；似然，即P(X|θ)，是假设θ已知后我们观察到的数据应该是什么样子的；后验，即P(θ|X)，是最终的参数分布。

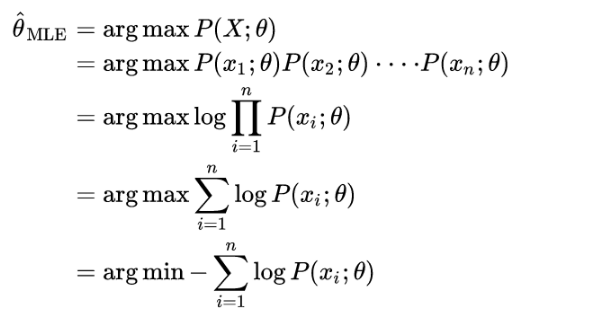
贝叶斯估计的基础是贝叶斯公式，如下：



同样是抛硬币的例子，对一枚均匀硬币抛5次得到5次正面，如果先验认为大概率下这个硬币是均匀的 (例如最大值取在0.5处的Beta分布)，那么P(head)，即P(θ|X)，是一个distribution，最大值会介于0.5~1之间，而不是武断的θ= 1。

**MLE - 最大似然估计**

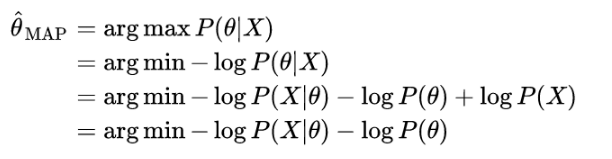
Maximum Likelihood Estimation, MLE是频率学派常用的估计方法！假设数据X1，X2，…，Xn是一组抽样，X=(X1，X2，…，Xn)。独立同分布。那么MLE对θ的估计方法可以如下推导：



**MAP - 最大后验估计**

Maximum A Posteriori, MAP是贝叶斯学派常用的估计方法！

同样的，假设数据X1，X2，…，Xn是独立同分布的一组抽样，X=(X1，X2，…，Xn) 。那么MAP对θ的估计方法可以如下推导：



## 贝叶斯公式在机器学习中的应用

在贝叶斯机器学习中，我们同样采用贝叶斯公式从 data（D）中推导模型参数（θ）。

P(θ|D) = P(D|θ) \* P(θ) / P(data)

值得说明的是，P（data）在通常情况下无法被计算，但这并不会带来什么问题。因为我们在推导及求解的过程中，最重要的还是那些含有θ的表达式，而 P（data）到底是多少，其实并不需要真的求出来。

P(θ) 为先验概率，也就是我们对样本空间中各类样本所占比例的可能性推测。通常我们认为，当训练集包含重组的独立同分步样本时，P(θ) 可通过各类样本出现的频率进行判断。关于先验的其它知识，可以参考 Where priors come from 的介绍。

P(D|θ) 是样本 D 相对于类标记θ的类条件概率，也就是我们理解的「似然」。人们经常会使用可能性来评估模型，如果对实际数据能做出更高可能性的预测，那么这个模型自然更为有效。

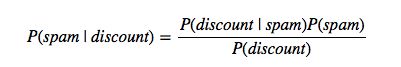
那么，等式左边的 P(θ|D) 就是我们最终想得到的东西，也就是基于先验概率与数据所获得的模型参数所呈现的概率分布。

如果能通过数据采样来估计概率分布参数，最经典的方法就是最大似然估计（maximum-likelihood estimation，MLE），也就是我们所说的极大似然法。而如果将先验考虑在内，那么就是最大后验概率（MAP）。如果在先验均匀分布的情况下，这两者应该相同。

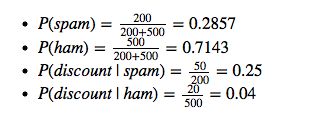
## 垃圾邮件识别

为了说明问题，让我们简化一下，假设我的邮箱里有500封正常邮件(ham)，200封垃圾邮件(spam)，现在来了一封新邮件，我发现里面出现了单词“discount”（折扣），我想知道这封邮件是垃圾邮件的概率有多大呢？

很明显，这是一个条件概率问题，让我们试试贝叶斯公式：



我检查了一下所有邮件（请放心，我不是傻傻地靠肉眼检查，一个脚本就可以搞定它），发现垃圾邮件中包含单词discount的有50封，非垃圾邮件中包含单词discount的有20封。那么：



计算P(discount)Pdiscount需要用到全概公式，这也是贝叶斯公式的推论：

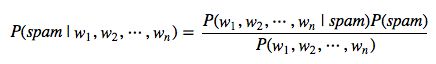


好了，现在让我们看看最终的结果：



如果只考虑单词discount，我们有71%的把握认为这是一封垃圾邮件（spam）。

然而，实际情况是，一封邮件通常包含了大量的单词，假设为w1, w2, w3,⋯,wn，那么我们需要计算的是给定单词向量，这封邮件是垃圾邮件的概率是多少。贝叶斯公式稍作改动：



问题似乎没什么变化，我们只要将单词discount换成单词向量{w1, w2, w3,⋯}，重新计算一遍就行了。

然而，读者很容易发现，计算P(w1,w2,⋯ | spam)是困难的，因为你几乎很难找到同时包含这些单词的邮件，这时候就需要借助朴素贝叶斯的“朴素”之力了。

数学以简洁为美，简洁不仅使得知识易于理解，而且让计算变得容易。

言归正传，“朴素”之意实际上是独立性假设，对于垃圾邮件过滤即认为每个单词的出现与否是条件独立的，这是一个很强的假设，然而带来的好处是上述的概率计算异常简单：



你看问题又回到了分别计算每个单词相对应的条件概率值了！

朴素贝叶斯因其独立性假设而得名，它不论模型训练或者预测分类，计算都是线性的，因此计算的开销很小，然而效果却很不错！

因此它也是公认的数据挖掘十大算法之一。

很多人好奇，朴素贝叶斯方法的条件独立假设看上去很傻很天真，为什么结果却很好很强大呢？

就拿一个句子来说，我们怎么能鲁莽地声称其中任意一个单词出现的概率只受到它前面的 3 个或 4 个单词的影响呢？

别说 3 个，有时候一个单词的概率受到上一句话的影响都是绝对可能的。那么为什么这个假设在实际中的表现却不比决策树差呢？

有人对此提出了一个理论解释，并且建立了什么时候朴素贝叶斯的效果能够等价于非朴素贝叶斯的充要条件，这个解释的核心就是：

有些独立假设在各个分类之间的分布都是均匀的所以对于似然（即P(w1,w2,⋯, wn | spam)）的相对大小不产生影响；即便不是如此，也有很大的可能性各个独立假设所产生的消极影响或积极影响互相抵消，最终导致结果受到的影响不大。

细心的读者可能会发现，假如某个单词(例如w1w1)在垃圾邮件中不出现，那么P(w1 | spam)=0，进而P(w1,w2,⋯ | spam)=0，显然这是不合理的。

毕竟由于我的邮件数量有限，邮件中未出现的单词不能认为其概率为0。

合理的做法是给那些未出现的单词赋一个很小的概率值，这叫做平滑。

另一方面，实际应用中，单词向量是很稀疏的，每个单词的概率也是很小的，为了防止计算中出现浮点溢出的情况，可以对P(w1,w2,⋯, wn | spam)求对数：

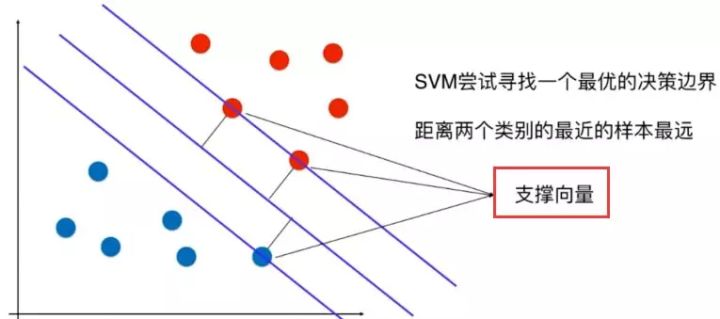


垃圾邮件过滤是文本二分类问题，对于文本多分类问题，思路是一样的。

# SVM（分类算法）

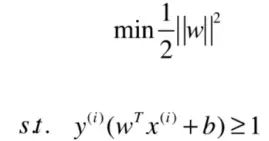
## SVM原理

SVM就是寻找一个最优的决策边界，距离两个类别的最近的样本最远，其中最近的样本点称为支持（支撑）向量：



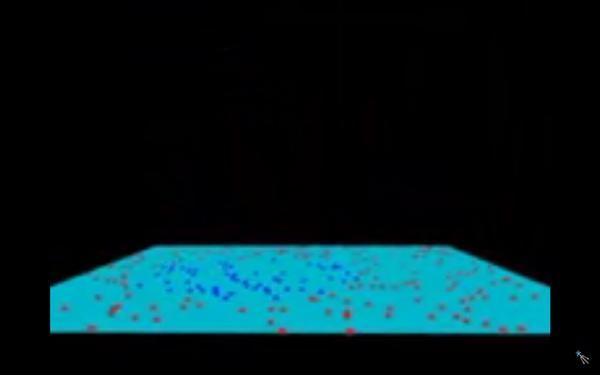
转化成数学问题，就是使得中间的直线距离两边的直线的间隔（margin）最大（这两边的直线的斜率是一样的），也就是SVM算法就是最大化margin。

最终问题转化为数学公式：

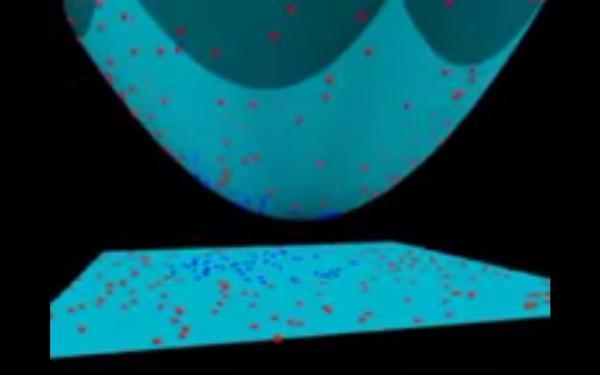


核函数：它可以将样本从原始空间映射到一个更高维的特质空间中，使得样本在这个新的高维空间中可以被线性划分为两类，即在空间内**线性划分**。这个过程可以观看[视频](https://link.jianshu.com/?t=https://www.youtube.com/watch?v=3liCbRZPrZA)感受感受，由于是youtube所以我截一下图：

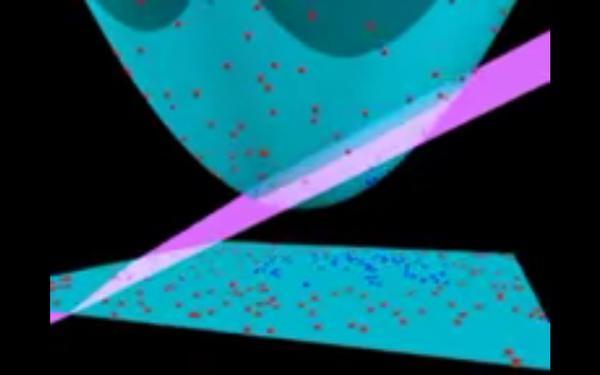
这是原始数据和原始空间，明显有红蓝两类：



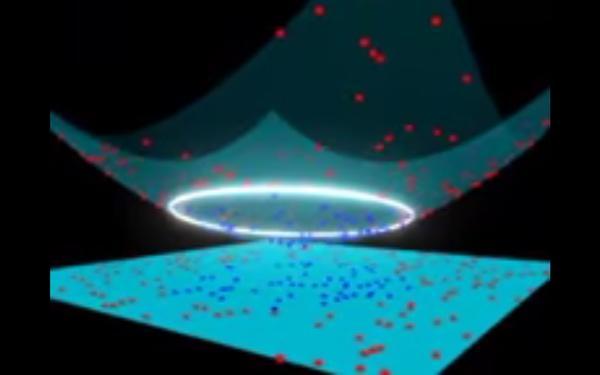
通过核函数，将样本数据映射到更高维的空间（在这里，是二维映射到三维）：

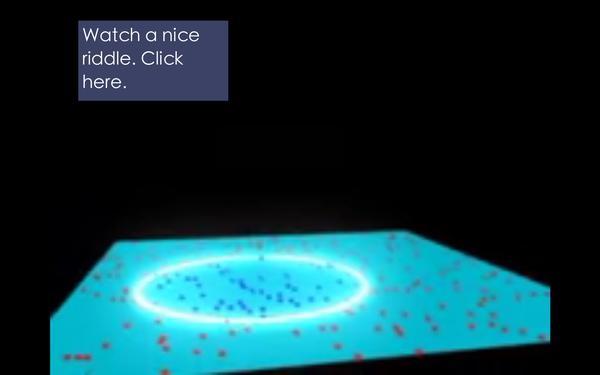


而后进行切割：



再将分割的超平面映射回去：





再进一步，核函数的选择变成了支持向量机的最大变数（如果必须得用上核函数，即核化），因此选用什么样的核函数会影响最后的结果。而最常用的核函数有：线性核、多项式核、高斯核、拉普拉斯核、sigmoid核、通过核函数之间的线性组合或直积等运算得出的新核函数。

## sklearn实现

SVM在sklearn库中主要三个参数

- kernel（核函数linear、RBF）

- C（ C是惩罚系数，即对误差的宽容度。c越高，说明越不能容忍出现误差,容易过拟合。C越小，容易欠拟合。C过大或过小，泛化能力变差）

- gamma（ gamma是选择RBF函数作为kernel后，该函数自带的一个参数。隐含地决定了数据映射到新的特征空间后的分布，gamma越大，支持向量越少，gamma值越小，支持向量越多。支持向量的个数影响训练与预测的速度。）

我们先说个例子，看看简单的使用sklean中的SVC(support vectors classification)怎么使用。

**from** sklearn **import** svm

X = [[0, 0], [1, 1], [1, 0]]  # training samples

y = [0, 1, 1]  # training target

clf = svm.SVC()  # class

clf.fit(X, y)  # training the svc model

result = clf.predict([2, 2]) # predict the target of testing samples

**print** result  # target

**print** clf.support\_vectors\_  #support vectors

**print** clf.support\_  # indeices of support vectors

**print** clf.n\_support\_  # number of support vectors for each class

# Kmeas(聚类算法)

## k-means算法原理分析

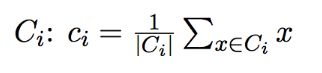
k-means算法是聚类分析中使用最广泛的算法之一。它把n个对象根据它们的属性分为k个聚类以便使得所获得的聚类满足：同一聚类中的对象相似度较高；而不同聚类中的对象相似度较小。

k-means算法的基本过程如下所示：

（1）任意选择k个初始中心。

（2）计算X中的每个对象与这些中心对象的距离；并根据最小距离重新对相应对象进行划分；

（3）重新计算每个中心对象  的值



（4）计算标准测度函数，当满足一定条件，如函数收敛时，则算法终止；如果条件不满足则重复步骤（2），（3）。

k-means算法虽然简单快速，但是存在下面的缺点：

1. 聚类中心的个数K需要事先给定，但在实际中K值的选定是非常困难的，很多时候我们并不知道给定的数据集应该分成多少个类别才最合适。
2. k-means算法需要随机地确定初始聚类中心，不同的初始聚类中心可能导致完全不同的聚类结果。

第一个缺陷我们很难在k-means算法以及其改进算法中解决，但是我们可以通过k-means++算法来解决第二个缺陷。

## k-means++算法原理分析

k-means++算法选择初始聚类中心的基本原则是：初始的聚类中心之间的相互距离要尽可能的远。它选择初始聚类中心的步骤是：

（1）从输入的数据点集合中随机选择一个点作为第一个聚类中心 ；

（2）对于数据集中的每一个点x，计算它与最近聚类中心(指已选择的聚类中心)的距离D(x)，并根据概率选择新的聚类中心  。

（3）重复过程（2）直到找到k个聚类中心。

第(2)步中，依次计算每个数据点与最近的种子点（聚类中心）的距离，依次得到D(1)、D(2)、...、D(n)构成的集合D，其中n表示数据集的大小。在D中，为了避免噪声，不能直接选取值最大的元素，应该选择值较大的元素，然后将其对应的数据点作为种子点。 如何选择值较大的元素呢，下面是spark中实现的思路。

求所有的距离和Sum(D(x))

取一个随机值，用权重的方式来取计算下一个“种子点”。这个算法的实现是，先用Sum(D(x))乘以随机值Random得到值r，然后用currSum += D(x)，直到其currSum > r，此时的点就是下一个“种子点”。

为什么用这样的方式呢？我们换一种比较好理解的方式来说明。把集合D中的每个元素D(x)想象为一根线L(x)，线的长度就是元素的值。将这些线依次按照L(1)、L(2)、...、L(n)的顺序连接起来，组成长线L。L(1)、L(2)、…、L(n)称为L的子线。 根据概率的相关知识，如果我们在L上随机选择一个点，那么这个点所在的子线很有可能是比较长的子线，而这个子线对应的数据点就可以作为种子点。

# DBSCAN(聚类算法)

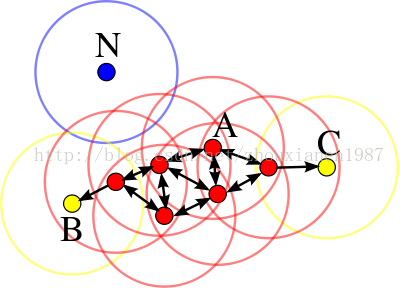
## 基本概念

（1）Eps邻域：给定对象半径Eps内的邻域称为该对象的Eps邻域;

（2）核心点（core point）：如果对象的Eps邻域至少包含最小数目MinPts的对象，则称该对象为核心对象;

（3）边界点（edge point）：边界点不是核心点，但落在某个核心点的邻域内;

（4）噪音点（outlier point）：既不是核心点，也不是边界点的任何点;

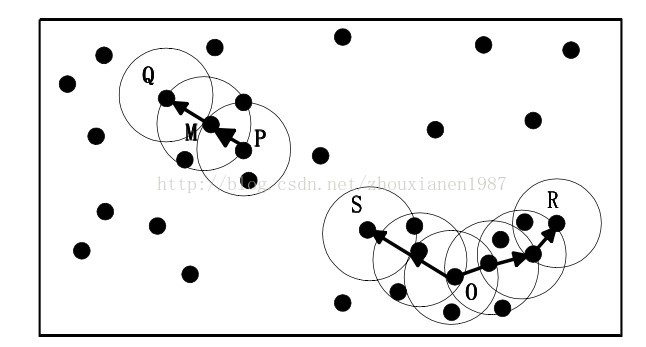


红色为核心点，黄色为边界点，蓝色为噪音点，minPts = 4

（5）直接密度可达(directly density-reachable)：给定一个对象集合D，如果p在q的Eps邻域内，而q是一个核心对象，则称对象p从对象q出发时是直接密度可达的;

（6）密度可达(density-reachable)：如果存在一个对象链  p1, …,pi,.., pn，满足p1 = p 和pn = q，pi是从pi+1关于Eps和MinPts直接密度可达的，则对象p是从对象q关于Eps和MinPts密度可达的;

（7）密度相连(density-connected)：如果存在对象O∈D，使对象p和q都是从O关于Eps和MinPts密度可达的，那么对象p到q是关于Eps和MinPts密度相连的。



其中，Eps用一个相应的半径表示，设MinPts=3，请分析Q,M,P,S,O,R这5个样本点之间的关系：

由于有标记的各点­M、P、O和R的Eps近邻均包含3个以上的点，因此它们都是核对象。

M­是从P“直接密度可达”；而Q则是从­M“直接密度可达”；基于上述结果，Q是从P“密度可达”；但P从Q无法“密度可达”(非对称)。

类似地，S和R从O是“密度可达”的；O、R和S均是“密度相连”（对称）的。

## DBSCAN算法描述:

输入: 包含n个对象的数据库，半径e，最少数目MinPts;

输出:所有生成的簇，达到密度要求。

(1)Repeat

(2)从数据库中抽出一个未处理的点；

(3)IF抽出的点是核心点 THEN 找出所有从该点密度可达的对象，形成一个簇；

(4)ELSE 抽出的点是边缘点(非核心对象)，跳出本次循环，寻找下一个点；

(5)UNTIL 所有的点都被处理。

DBSCAN对用户定义的参数很敏感，细微的不同都可能导致差别很大的结果，而参数的选择无规律可循，只能靠经验确定。

## 优缺点

**优点**

1. 与K-means方法相比，DBSCAN不需要事先知道要形成的簇类的数量。

2. 与K-means方法相比，DBSCAN可以发现任意形状的簇类。

3. 同时，DBSCAN能够识别出噪声点。

4.DBSCAN对于数据库中样本的顺序不敏感，即Pattern的输入顺序对结果的影响不大。但是，对于处于簇类之间边界样本，可能会根据哪个簇类优先被探测到而其归属有所摆动。

**缺点：**

1. DBScan不能很好反映高尺寸数据。

2. DBScan不能很好反映数据集以变化的密度。

## Sklean实现及参数

在scikit-learn中，DBSCAN算法类为sklearn.cluster.DBSCAN，DBSCAN类的重要参数也分为两类，一类是DBSCAN算法本身的参数，一类是最近邻度量的参数，下面我们对这些参数做一个总结。

1）**eps**： DBSCAN算法参数，即我们的ϵϵ-邻域的距离阈值，和样本距离超过ϵϵ的样本点不在ϵϵ-邻域内。默认值是0.5.一般需要通过在多组值里面选择一个合适的阈值。eps过大，则更多的点会落在核心对象的ϵϵ-邻域，此时我们的类别数可能会减少， 本来不应该是一类的样本也会被划为一类。反之则类别数可能会增大，本来是一类的样本却被划分开。

2）**min\_samples**： DBSCAN算法参数，即样本点要成为核心对象所需要的ϵϵ-邻域的样本数阈值。默认值是5. 一般需要通过在多组值里面选择一个合适的阈值。通常和eps一起调参。在eps一定的情况下，min\_samples过大，则核心对象会过少，此时簇内部分本来是一类的样本可能会被标为噪音点，类别数也会变多。反之min\_samples过小的话，则会产生大量的核心对象，可能会导致类别数过少。

3）**metric**：最近邻距离度量参数。可以使用的距离度量较多，一般来说DBSCAN使用默认的欧式距离（即p=2的闵可夫斯基距离）就可以满足我们的需求。

4）**algorithm**：最近邻搜索算法参数，算法一共有三种，第一种是蛮力实现，第二种是KD树实现，第三种是球树实现。这三种方法在[K近邻法(KNN)原理小结](http://www.cnblogs.com/pinard/p/6061661.html)中都有讲述，如果不熟悉可以去复习下。对于这个参数，一共有4种可选输入，‘brute’对应第一种蛮力实现，‘kd\_tree’对应第二种KD树实现，‘ball\_tree’对应第三种的球树实现， ‘auto’则会在上面三种算法中做权衡，选择一个拟合最好的最优算法。需要注意的是，如果输入样本特征是稀疏的时候，无论我们选择哪种算法，最后scikit-learn都会去用蛮力实现‘brute’。个人的经验，一般情况使用默认的 ‘auto’就够了。 如果数据量很大或者特征也很多，用"auto"建树时间可能会很长，效率不高，建议选择KD树实现‘kd\_tree’，此时如果发现‘kd\_tree’速度比较慢或者已经知道样本分布不是很均匀时，可以尝试用‘ball\_tree’。而如果输入样本是稀疏的，无论你选择哪个算法最后实际运行的都是‘brute’。

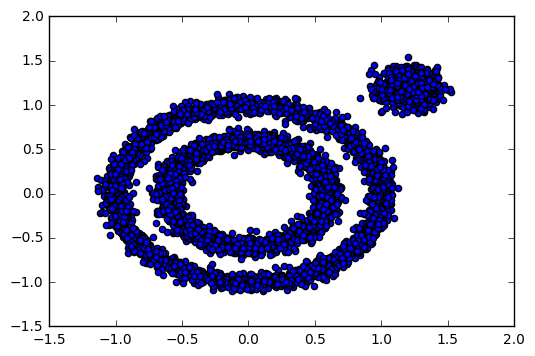
5）**leaf\_size**：最近邻搜索算法参数，为使用KD树或者球树时， 停止建子树的叶子节点数量的阈值。这个值越小，则生成的KD树或者球树就越大，层数越深，建树时间越长，反之，则生成的KD树或者球树会小，层数较浅，建树时间较短。默认是30. 因为这个值一般只影响算法的运行速度和使用内存大小，因此一般情况下可以不管它。

6） **p**: 最近邻距离度量参数。只用于闵可夫斯基距离和带权重闵可夫斯基距离中p值的选择，p=1为曼哈顿距离， p=2为欧式距离。如果使用默认的欧式距离不需要管这个参数。

以上就是DBSCAN类的主要参数介绍，其实需要调参的就是两个参数eps和min\_samples，这两个值的组合对最终的聚类效果有很大的影响。

首先，我们生成一组随机数据，为了体现DBSCAN在非凸数据的聚类优点，我们生成了三簇数据，两组是非凸的。代码如下：





那么如果使用DBSCAN效果如何呢？我们先不调参，直接用默认参数，看看聚类效果,代码如下：

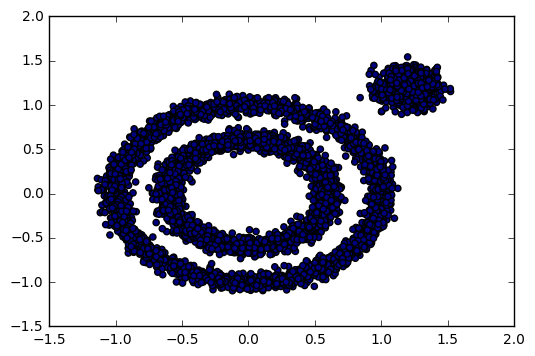
from sklearn.cluster import DBSCAN

y\_pred = DBSCAN().fit\_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y\_pred)

plt.show()

发现输出让我们很不满意，DBSCAN居然认为所有的数据都是一类！输出效果图如下：



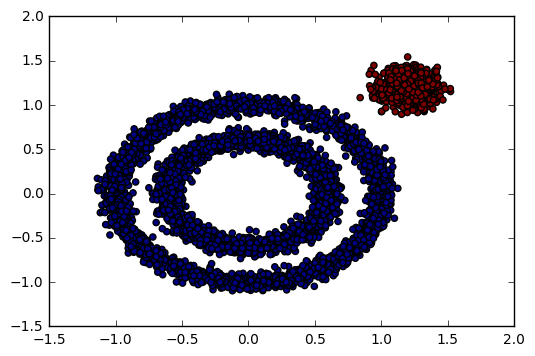
怎么办？看来我们需要对DBSCAN的两个关键的参数eps和min\_samples进行调参！从上图我们可以发现，类别数太少，我们需要增加类别数，那么我们可以减少ϵϵ-邻域的大小，默认是0.5，我们减到0.1看看效果。代码如下：

y\_pred = DBSCAN(eps = 0.1).fit\_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y\_pred)

plt.show()

对应的聚类效果图如下：



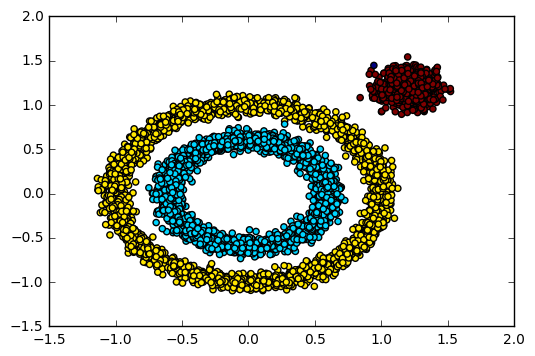
　可以看到聚类效果有了改进，至少边上的那个簇已经被发现出来了。此时我们需要继续调参增加类别，有两个方向都是可以的，一个是继续减少eps，另一个是增加min\_samples。我们现在将min\_samples从默认的5增加到10，代码如下：

y\_pred = DBSCAN(eps = 0.1, min\_samples = 10).fit\_predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y\_pred)

plt.show()

输出的效果图如下：



# 降维算法

## 矩阵相似及特征值分解

**矩阵的本质1：矩阵是一个坐标。**

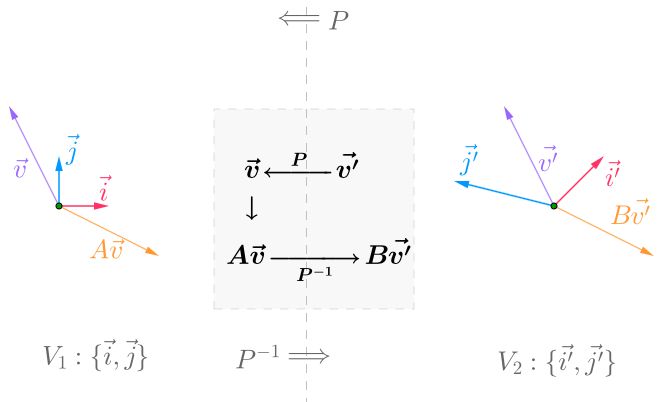
Ma=b的意思是：“有一个向量，它在坐标系M的度量下得到的度量结果向量为a，那么它在坐标系I的度量下，这个向量的度量结果是b。”这里的I是指单位矩阵，就是主对角线是1，其他为零的矩阵。而这两个方式本质上是等价的。

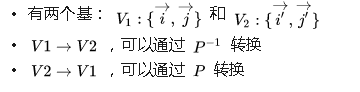
也就是说同一个向量，在直角坐标系I下表示为b，在另一个M坐标系下表示为a，其中M为在I下的描述，那么，一个向量从直角坐标系统下变换到M下，即：a=b，只需要。

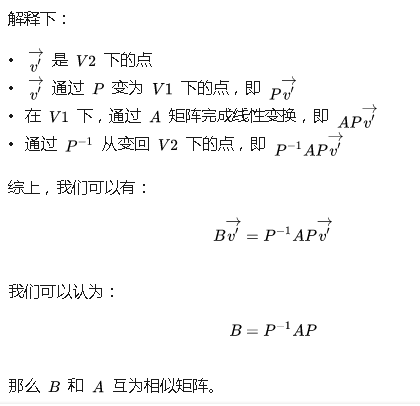
**矩阵的本质2：矩阵是线性空间中的线性变换的一个描述。在一个线性空间中，只要我们选定一组基，那么对于任何一个线性变换，都能够用一个确定的矩阵来加以描述。**

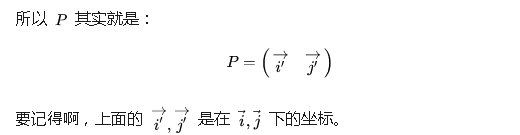
若矩阵A与B是同一个线性变换的两个不同的描述（之所以会不同，是因为选定了不同的基，也就是选定了不同的坐标系），则一定能找到一个非奇异矩阵P，使得A、B之间满足这样的关系：A=P-1BP。线性代数稍微熟一点的读者一下就看出来，这就是相似矩阵的定义。没错，所谓相似矩阵，就是同一个线性变换的不同的描述矩阵。

先上一张图，说明不同基下的矩阵的变换思路，这个图有点复杂，请参照之后的解释一起来看：









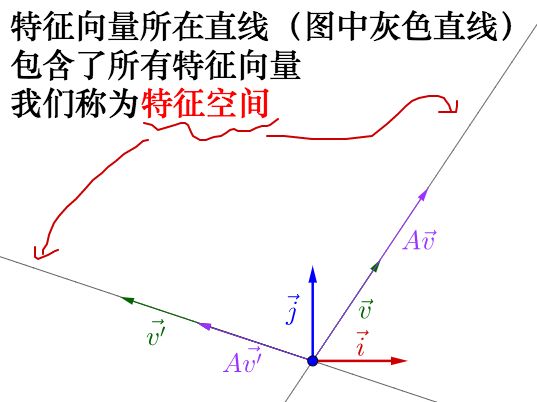
**矩阵对角化**

因为对角矩阵对应的变化只是把向量沿着坐标轴拉升而不进行旋转变化。所以把矩阵相似于对角阵的操作很有意义，而相矩阵对角化本质是求特征值和特征向量。



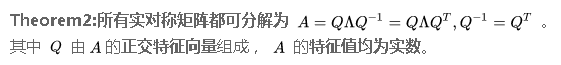
**特征值和特征向量的本质**

****

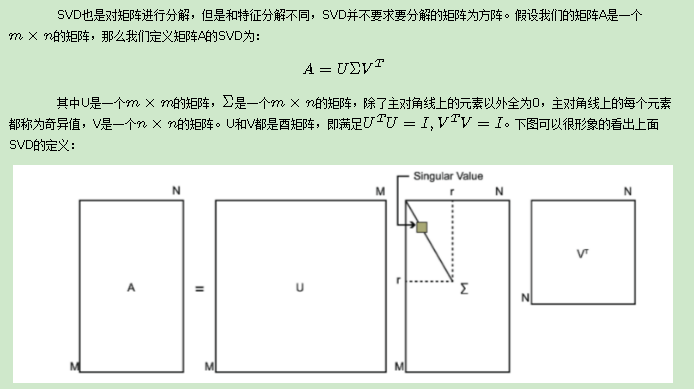
****

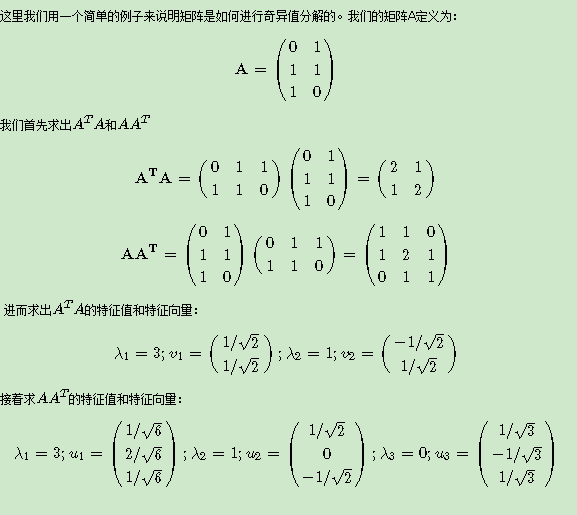
矩阵A的线性变化的作用相当于在由特征向量组成的特征空间里对被作用的向量进行比例为λ的拉升。

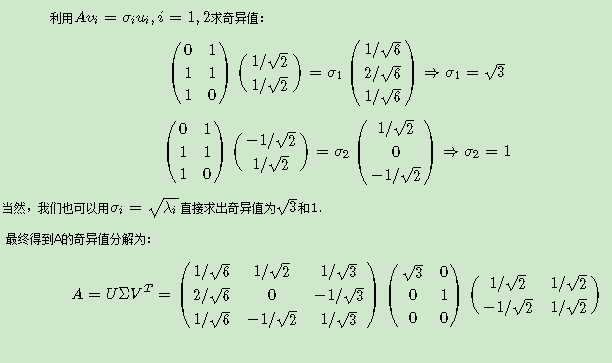
**实对称矩阵对角化**

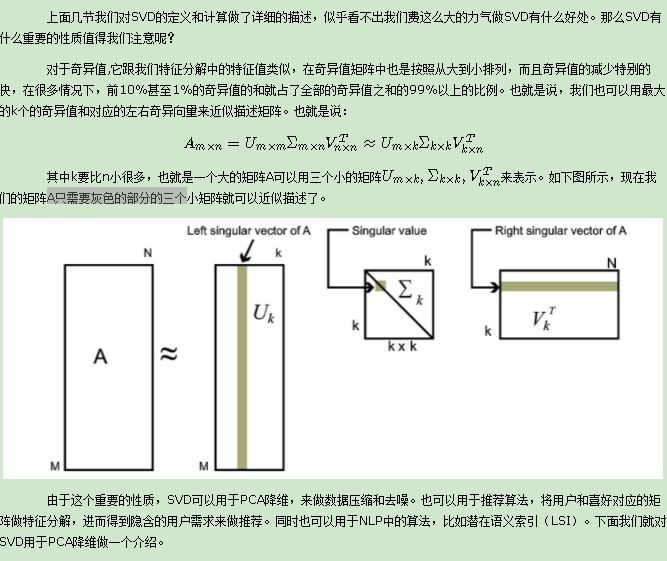


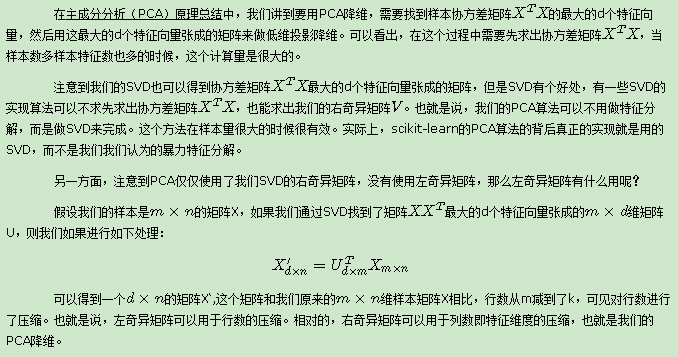
## SVD奇异值分解





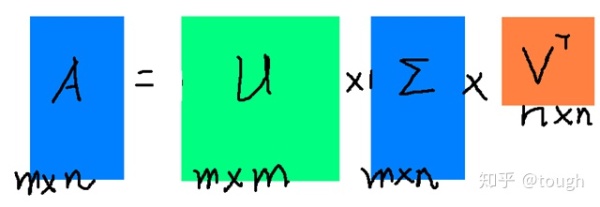




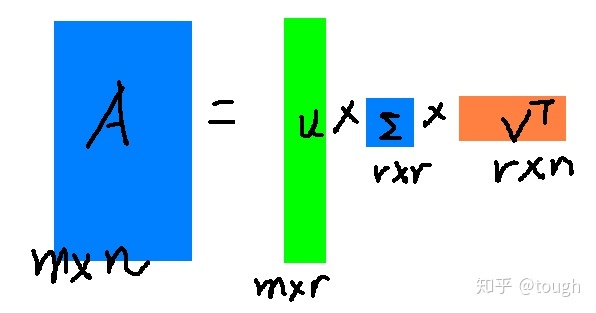


## PCA

SDV

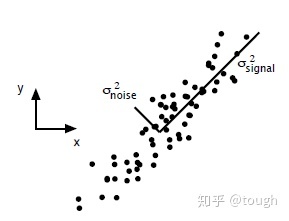


PCA





PCA的问题其实是一个基的变换，使得变换后的数据有着最大的方差。方差的大小描述的是一个变量的信息量，我们在讲一个东西的稳定性的时候，往往说要减小方差，如果一个模型的方差很大，那就说明模型不稳定了。但是对于我们用于机器学习的数据（主要是训练数据），方差大才有意义，不然输入的数据都是同一个点，那方差就为0了，这样输入的多个数据就等同于一个数据了。以下面这张图为例子：



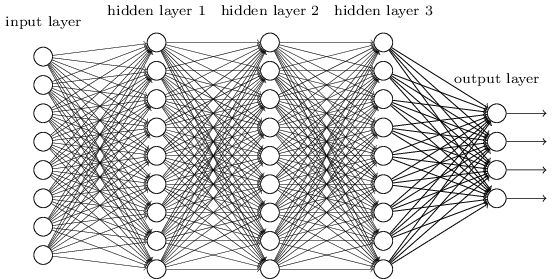
## LDA

# 深度学习篇

# DNN

## 深度神经网络

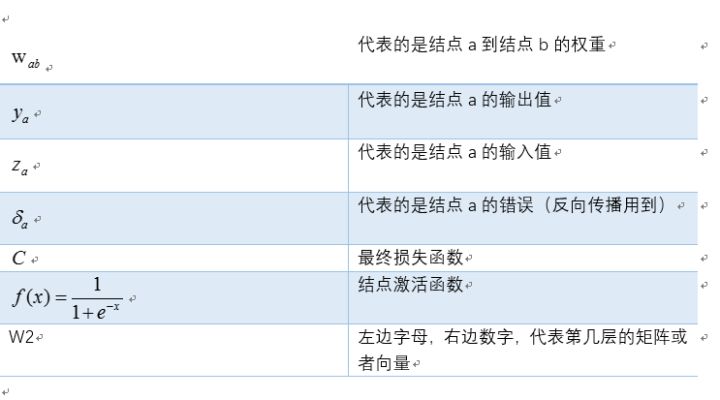
从DNN按不同层的位置划分，DNN内部的神经网络层可以分为三类，输入层，隐藏层和输出层,如下图示例，一般来说第一层是输入层，最后一层是输出层，而中间的层数都是隐藏层。



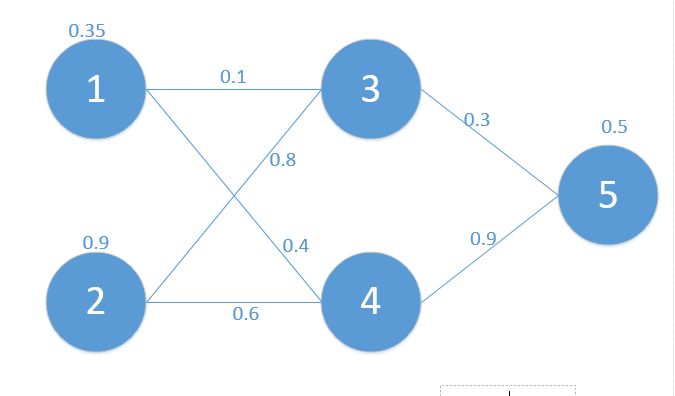
层与层之间是全连接的，也就是说第i层的任意一个神经元一定与第i+1层的任意一个神经元相连。虽然DNN看起来很复杂，但是从小的局部模型来说，还是和感知机一样，即一个线性关系z=∑wixi+b加上一个激活函数σ(z)。

## BP算法

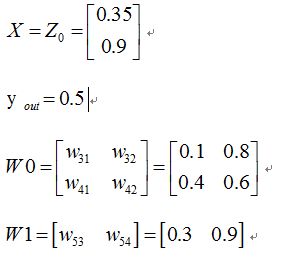
为了方便起见，这里我定义了三层网络，输入层（第0层），隐藏层（第1层），输出层（第二层）。并且每个结点没有偏置（有偏置原理完全一样），激活函数为sigmod函数（不同的激活函数，求导不同），符号说明如下：



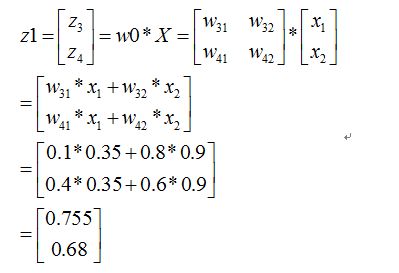
对应网络如下：



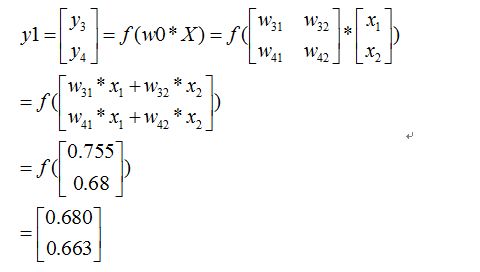
其中对应的矩阵表示如下：



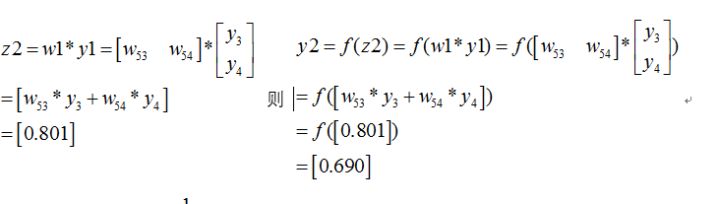
首先我们先走一遍正向传播，公式与相应的数据对应如下：



那么：



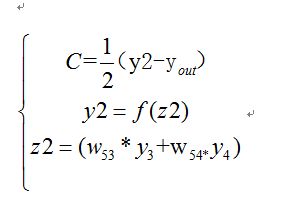
同理可以得到



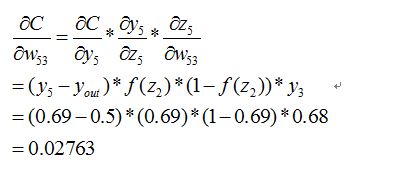
那么最终的损失为



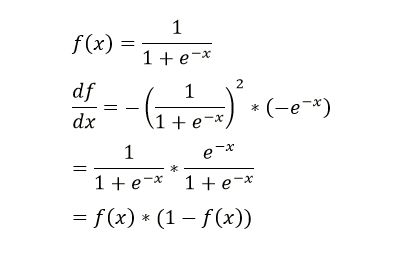
我们当然是希望这个值越小越好。这也是我们为什么要进行训练，调节参数，使得最终的损失最小。这就用到了我们的反向传播算法，实际上反向传播就是梯度下降法中链式法则的使用。下面我们看如何反向传播根据公式，我们有：



这个时候我们需要求出C对w的偏导，则根据链式法则有：

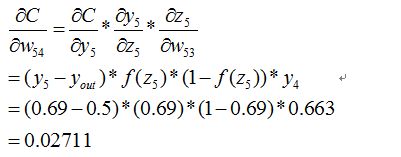


上面插入sigmod函数求导公式：



（在这里我们可以看到不同激活函数求导是不同的，所谓的梯度消失，梯度爆炸如果了解bp算法的原理，也是非常容易理解的！）

同理有：

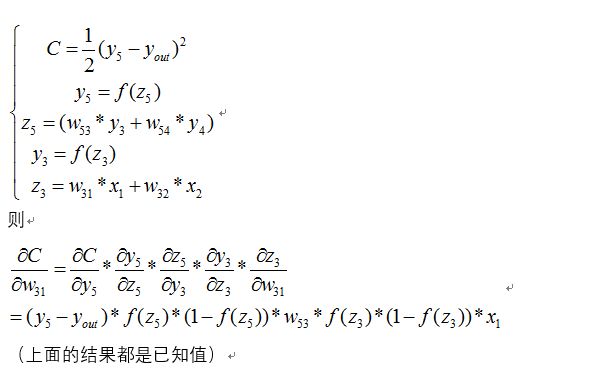


到此我们已经算出了最后一层的参数偏导了.我们继续往前面链式推导：

我们现在还需要求

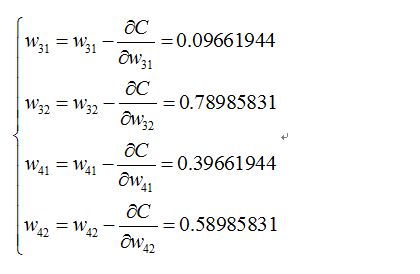


下面给出其中的一个推到，其它完全类似



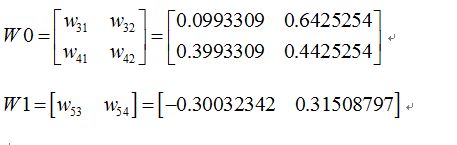
同理可得到其它几个式子：

则最终的结果为：



再按照这个权重参数进行一遍正向传播得出来的Error为0.165

而这个值比原来的0.19要小，则继续迭代，不断修正权值，使得代价函数越来越小，预测值不断逼近0.5.我迭代了100次的结果，Error为5.92944818e-07（**已经很小了，说明预测值与真实值非常接近了**），最后的权值为：



## 激活函数

每个激活函数的输入都是一个数字，然后对其进行某种固定的数学操作。激活函数给神经元引入了非线性因素，如果不用激活函数的话，无论神经网络有多少层，输出都是输入的线性组合。

激活函数的发展经历了Sigmoid -> Tanh -> ReLU -> Leaky ReLU -> Maxout这样的过程，还有一个特殊的激活函数Softmax，因为它只会被用在网络中的最后一层，用来进行最后的分类和归一化。本文简单来梳理这些激活函数是如何一步一步演变而来的。

### Sigmoid

sigmoid非线性函数的数学公式是

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\2201433306.bmp

函数图像如下图所示。它输入实数值并将其“挤压”到0到1范围内，适合输出为概率的情况，但是现在已经很少有人在构建神经网络的过程中使用sigmoid。



存在问题：

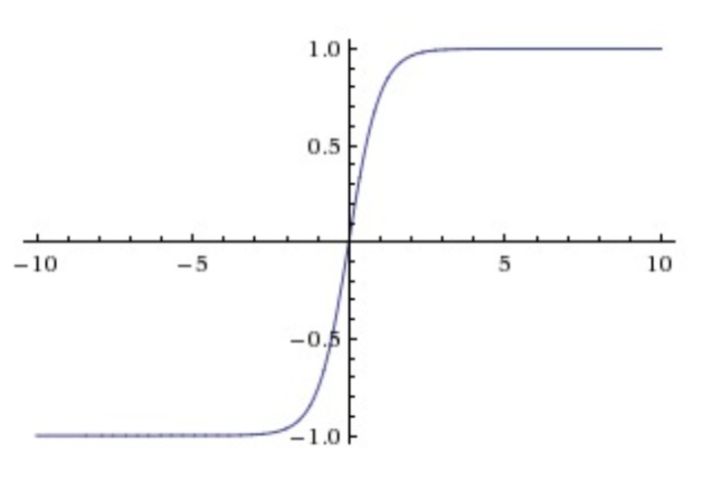
与此同时，Sigmoid函数也有两个很大的缺点：首先是Sigmoid函数会造成梯度消失问题，从图像中我们也可以得知，当输入特别大或是特别小时，神经元的梯度几乎接近于0，这就导致神经网络不收敛，模型的参数不会更新，训练过程将变得非常困难。另一方面，Sigmoid函数的输出不是以0为均值的，导致传入下一层神经网络的输入是非0的。这就导致一个后果：若Sigmoid函数的输出全部为正数，那么传入下一层神经网络的值永远大于0，这时参数无论怎么更新梯度都为正。正是基于上述的缺点，Sigmoid函数近年来的使用频率也在渐渐减弱。另外指数运算比较复杂。计算代价比较高。

### Tanh

Tanh非线性函数的数学公式是

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\3947876024.bmp

Tanh非线性函数图像如下图所示，它将实数值压缩到[-1,1]之间。



存在问题：

Tanh解决了Sigmoid的输出是不是零中心的问题，但仍然存在饱和问题。

为了防止饱和，现在主流的做法会在激活函数前多做一步*batch normalization*，尽可能保证每一层网络的输入具有均值较小的、零中心的分布。

### ReLU

函数公式是

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\2518492163.bmp

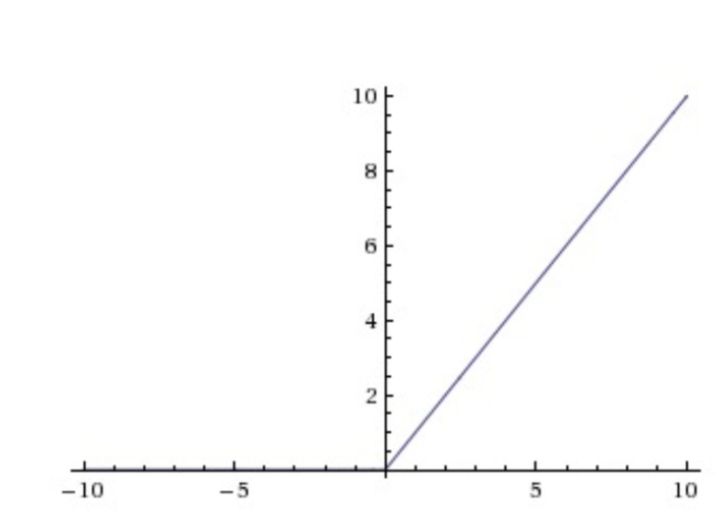
ReLU非线性函数图像如下图所示。相较于sigmoid和tanh函数，ReLU对于随机梯度下降的收敛有巨大的加速作用；sigmoid和tanh在求导时含有指数运算，而ReLU求导几乎不存在任何计算量。

对比sigmoid类函数主要变化是：

1）单侧抑制；

2）相对宽阔的兴奋边界；

3）稀疏激活性。



存在问题：

从ReLU的函数图像我们可以发现，函数原点左侧的部分，输出值为0，斜率为0；函数原点右侧是斜率为1的直线，且输出值就是输入值。相比于上述的Sigmoid和tanh两种激活函数，ReLU激活函数完美的解决了梯度消失的问题，因为它的线性的、非饱和的。此外，它的计算也更加简单，只需要设置一个特定的阈值就可以计算激活值，这样极大的提高了运算的速度。所以近年来，ReLU激活函数的应用越来越广泛。

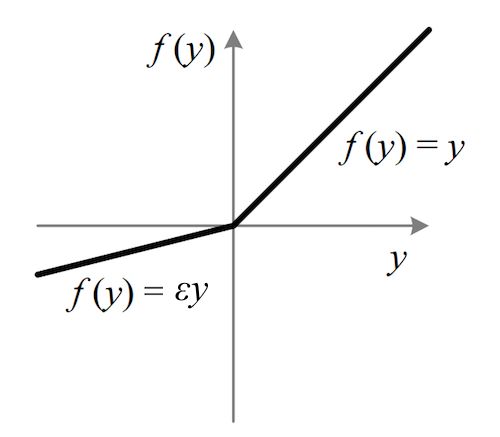
但是ReLU激活函数也有一些缺陷：训练的时候不适合大梯度的输入数据，因为在参数更新之后，ReLU的神经元不会再任何数据节点被激活，这就会导致梯度永远为0。比如：输入的数据小于0时，梯度就会为0，这就导致了负的梯度被置0，而且今后也可能不会被任何数据所激活，也就是说ReLU的神经元“坏死”了。

### Leaky ReLU

函数公式是:



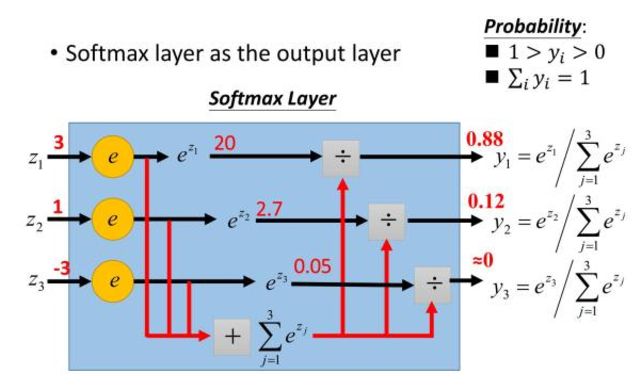
其中 ε 是很小的负数梯度值，比如0.01，Leaky ReLU非线性函数图像如下图所示。这样做目的是使负轴信息不会全部丢失，解决了ReLU神经元“死掉”的问题。更进一步的方法是PReLU，即把 ε当做每个神经元中的一个参数，是可以通过梯度下降求解的。



### Softmax

Softmax用于多分类神经网络输出，目的是让大的更大。函数公式是





Softmax是Sigmoid的扩展，当类别数k＝2时，Softmax回归退化为Logistic回归。

### 选择正确的激活函数

这么多激活函数需要在什么时候使用什么呢？这里并没有特定的规则。但是根据这些函数的特征，我们也可以总结一个比较好的使用规律或者使用经验，使得网络可以更加容易且更快的收敛。

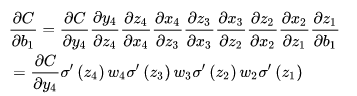
1. Sigmoid函数以及它们的联合通常在分类器的中有更好的效果
2. 由于梯度崩塌的问题，在某些时候需要避免使用Sigmoid和Tanh激活函数
3. ReLU函数是一种常见的激活函数，在目前使用是最多的
4. 如果遇到了一些死的神经元，我们可以使用Leaky ReLU函数
5. 记住，ReLU永远只在隐藏层中使用
6. 根据经验，我们一般可以从ReLU激活函数开始，但是如果ReLU不能很好的解决问题，再去尝试其他的激活函数

## 梯度消失和梯度爆炸

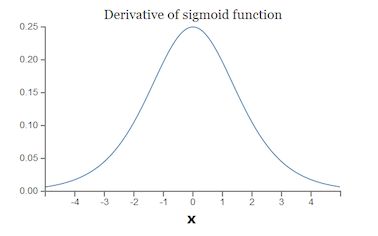
以下图的反向传播为例（假设每一层只有一个神经元且对于每一层C:\Users\Administrator\AppData\Roaming\feiq\RichOle\1694864603.bmp，其中激活函数为sigmoid函数）



其中每个节点的输入为x，输出为z，可以推导出:



而sigmoid的导数如下图:



可见，的最大值为0.25，而我们初始化的网络权值w的绝对值通常都小于1，因此，因此对于上面的链式求导，层数越多，求导结果越小，因而导致梯度消失的情况出现。

这样，梯度爆炸问题的出现原因就显而易见了，即C:\Users\Administrator\AppData\Roaming\feiq\RichOle\2421920887.bmp，也就是w比较大的情况。但对于使用sigmoid激活函数来说，这种情况比较少。因为C:\Users\Administrator\AppData\Roaming\feiq\RichOle\931205770.bmp的大小也与w有关（C:\Users\Administrator\AppData\Roaming\feiq\RichOle\3106377610.bmp），除非该层的输入值x在一直一个比较小的范围内。

**如何修复梯度爆炸问题？**

1. 重新设计网络模型

在深度神经网络中，梯度爆炸可以通过重新设计层数更少的网络来解决。

使用更小的批尺寸对网络训练也有好处。另外也许是学习率的原因，学习率过大导致的问题，减小学习率。

在循环神经网络中，训练过程中在更少的先前时间步上进行更新（沿时间的截断反向传播，truncated Backpropagation through time）可以缓解梯度爆炸问题。

2. 使用 ReLU 激活函数

在深度多层感知机神经网络中，梯度爆炸的发生可能是因为激活函数，如之前很流行的 Sigmoid 和 Tanh 函数。

使用 ReLU 激活函数可以减少梯度爆炸。采用 ReLU 激活函数是最适合隐藏层的新实践。

3. 使用长短期记忆网络

 在循环神经网络中，梯度爆炸的发生可能是因为某种网络的训练本身就存在不稳定性，如随时间的反向传播本质上将循环网络转换成深度多层感知机神经网络。

使用长短期记忆（LSTM）单元和相关的门类型神经元结构可以减少梯度爆炸问题。

采用 LSTM 单元是适合循环神经网络的序列预测的最新最好实践。

4. 使用梯度截断（Gradient Clipping）

 在非常深且批尺寸较大的多层感知机网络和输入序列较长的 LSTM 中，仍然有可能出现梯度爆炸。如果梯度爆炸仍然出现，你可以在训练过程中检查和限制梯度的大小。这就是梯度截断。

5. 使用权重正则化（Weight Regularization）

如果梯度爆炸仍然存在，可以尝试另一种方法，即检查网络权重的大小，并惩罚产生较大权重值的损失函数。该过程被称为权重正则化，通常使用的是 L1 惩罚项（权重绝对值）或 L2 惩罚项（权重平方）。

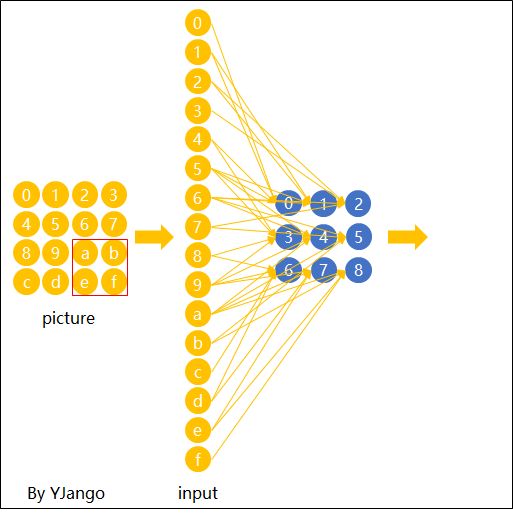
# CNN

参考https://zhuanlan.zhihu.com/p/27642620

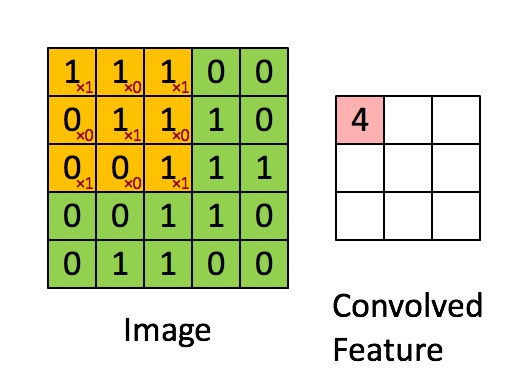
## 卷积核

### 输出表达

如先前在图像表达中提到的，图片不用向量去表示是为了保留图片平面结构的信息。 同样的，卷积后的输出若用上图的排列方式则丢失了平面结构信息。 所以我们依然用矩阵的方式排列它们，就得到了下图所展示的连接。



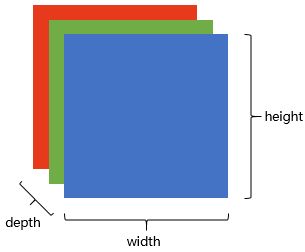
这也就是你们在网上所看到的下面这张图。在看这张图的时候请结合上图的连接一起理解，即输入（绿色）的每九个节点连接到输出（粉红色）的一个节点上的。



经过一个feature detector计算后得到的粉红色区域也叫做一个“**Convolved Feature**” 或 “**Activation Map**” 或 “**Feature Map**”。

### Depth维的处理

现在我们已经知道了depth维度只有1的灰度图是如何处理的。 但前文提过，图片的普遍表达方式是下图这样有3个channels的RGB颜色模型。 当depth为复数的时候，每个feature detector是如何卷积的？



**现象**：2x2所表达的filter size中，一个2表示width维上的局部连接数，另一个2表示height维上的局部连接数，并却没有depth维上的局部连接数，是因为depth维上并非局部，而是全部连接的。

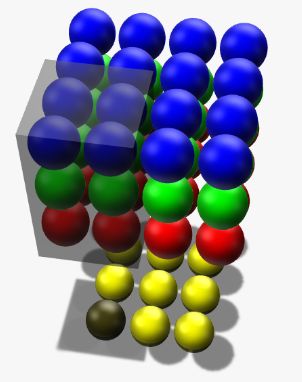
在2D卷积中，filter在张量的width维, height维上是局部连接，在depth维上是贯串全部channels的。

**类比**：想象在切蛋糕的时候，不管这个蛋糕有多少层，通常大家都会一刀切到底，但是在长和宽这两个维上是局部切割。

下面这张图展示了，在depth为复数时，filter是如何连接输入节点到输出节点的。 图中红、绿、蓝颜色的节点表示3个channels。 黄色节点表示一个feature detector卷积后得到的Feature Map。 其中被透明黑框圈中的12个节点会被连接到黄黑色的节点上。

在输入depth为1时：被filter size为2x2所圈中的4个输入节点连接到1个输出节点上。

在输入depth为3时：被filter size为2x2，但是贯串3个channels后，所圈中的12个输入节点连接到1个输出节点上。

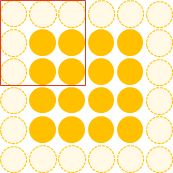


每个filter会在width维, height维上，以局部连接和空间共享，并贯串整个depth维的方式得到一个Feature Map。

### Zero padding

细心的读者应该早就注意到了，4x4的图片被2x2的filter卷积后变成了3x3的图片，每次卷积后都会小一圈的话，经过若干层后岂不是变的越来越小？ Zero padding就可以在这时帮助控制Feature Map的输出尺寸，同时避免了边缘信息被一步步舍弃的问题。

例如：下面4x4的图片在边缘Zero padding一圈后，再用3x3的filter卷积后，得到的Feature Map尺寸依然是4x4不变。



通常大家都想要在卷积时保持图片的原始尺寸。 选择3x3的filter和1的zero padding，或5x5的filter和2的zero padding可以保持图片的原始尺寸。 这也是为什么大家多选择3x3和5x5的filter的原因。 另一个原因是3x3的filter考虑到了像素与其距离为1以内的所有其他像素的关系，而5x5则是考虑像素与其距离为2以内的所有其他像素的关系。

**尺寸**：Feature Map的尺寸等于(input\_size + 2 \* padding\_size − filter\_size)/stride+1。

**注意**：上面的式子是计算width或height一维的。padding\_size也表示的是单边补零的个数。例如(4+2-3)/1+1 = 4，保持原尺寸。

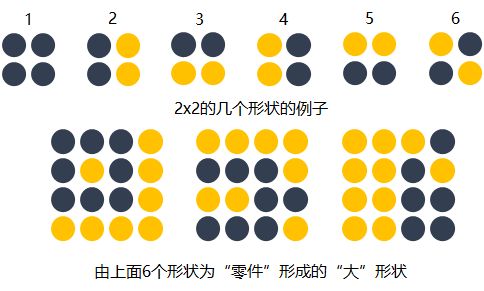
不用去背这个式子。其中(input\_size + 2 \* padding\_size)是经过Zero padding扩充后真正要卷积的尺寸。 减去 filter\_size后表示可以滑动的范围。 再除以可以一次滑动（stride）多少后得到滑动了多少次，也就意味着得到了多少个输出节点。 再加上第一个不需要滑动也存在的输出节点后就是最后的尺寸。

### 形状、概念抓取

知道了每个filter在做什么之后，我们再来思考这样的一个filter会抓取到什么样的信息。

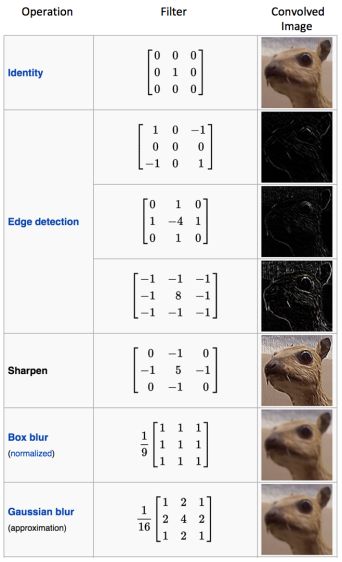
我们知道不同的形状都可由细小的“零件”组合而成的。比如下图中，用2x2的范围所形成的16种形状可以组合成格式各样的“更大”形状。

卷积的每个filter可以探测特定的形状。又由于Feature Map保持了抓取后的空间结构。若将探测到细小图形的Feature Map作为新的输入再次卷积后，则可以由此探测到“更大”的形状概念。 比如下图的第一个“大”形状可由2,3,4,5基础形状拼成。第二个可由2,4,5,6组成。第三个可由6,1组成。



除了基础形状之外，颜色、对比度等概念对画面的识别结果也有影响。卷积层也会根据需要去探测特定的概念。

可以从下面这张图中感受到不同数值的filters所卷积过后的Feature Map可以探测边缘，棱角，模糊，突出等概念。

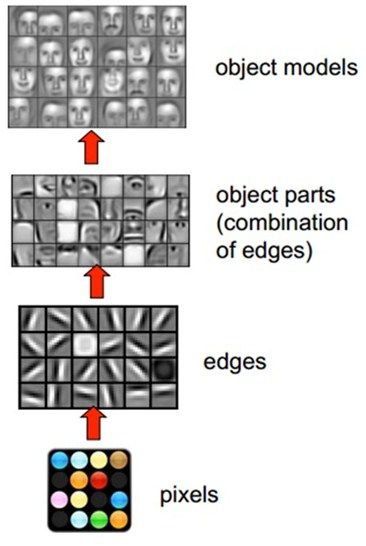


如我们先前所提，图片被识别成什么不仅仅取决于图片本身，还取决于图片是如何被观察的。

而filter内的权重矩阵W是网络根据数据学习得到的，也就是说，我们让神经网络自己学习以什么样的方式去观察图片。

拿老妇与少女的那幅图片举例，当标签是少女时，卷积网络就会学习抓取可以成少女的形状、概念。 当标签是老妇时，卷积网络就会学习抓取可以成老妇的形状、概念。

下图展现了在人脸识别中经过层层的卷积后，所能够探测的形状、概念也变得越来越抽象和复杂。



卷积神经网络会尽可能寻找最能解释训练数据的抓取方式。

### 多filters

每个filter可以抓取探测特定的形状的存在。 假如我们要探测下图的长方框形状时，可以用4个filters去探测4个基础“零件”。





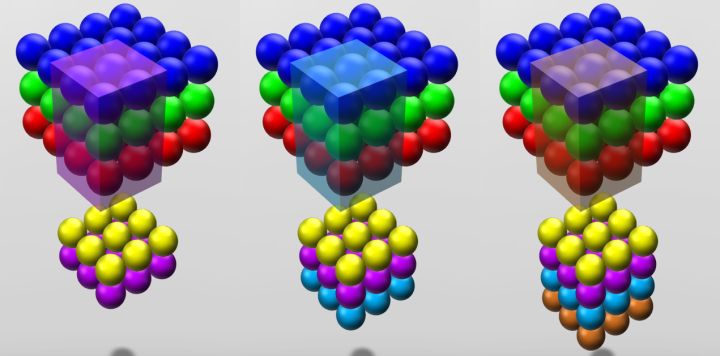
因此我们自然而然的会选择用多个不同的filters对同一个图片进行多次抓取。 如下图（动态图过大，如果显示不出，请看到该[链接](https://link.zhihu.com/?target=https%3A//ujwlkarn.files.wordpress.com/2016/08/giphy.gif)观看），同一个图片，经过两个（红色、绿色）不同的filters扫描过后可得到不同特点的Feature Maps。 每增加一个filter，就意味着你想让网络多抓取一个特征。



[[from](https://link.zhihu.com/?target=https%3A//ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/)]

这样卷积层的输出也不再是depth为1的一个平面，而是和输入一样是depth为复数的长方体。

如下图所示，当我们增加一个filter（紫色表示）后，就又可以得到一个Feature Map。 将不同filters所卷积得到的Feature Maps按顺序堆叠后，就得到了一个卷积层的最终输出。



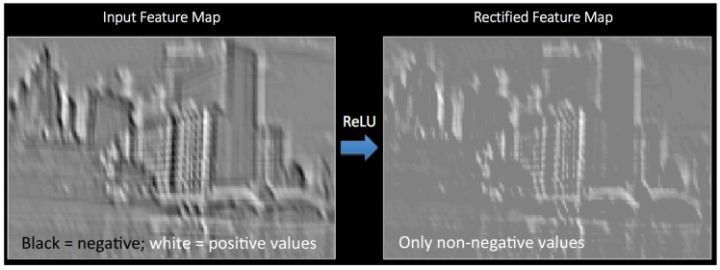
卷积层的输入是长方体，输出也是长方体。

这样卷积后输出的长方体可以作为新的输入送入另一个卷积层中处理。

### 加入非线性

和前馈神经网络一样，经过线性组合和偏移后，会加入非线性增强模型的拟合能力。

将卷积所得的Feature Map经过ReLU变换（elementwise）后所得到的output就如下图所展示。

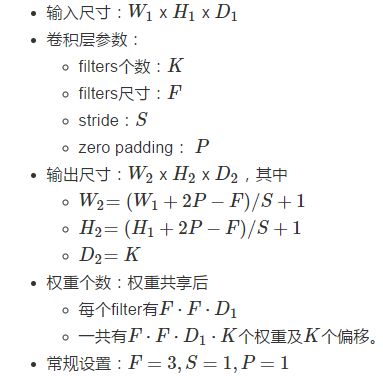


[[from](https://link.zhihu.com/?target=http%3A//mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf)]

### 输出长方体

现在我们知道了一个卷积层的输出也是一个长方体。 那么这个输出长方体的(width, height, depth)由哪些因素决定和控制。

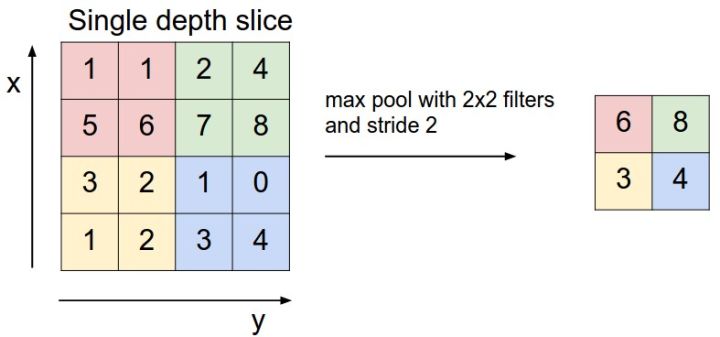
这里直接用[CS231n](https://link.zhihu.com/?target=http%3A//cs231n.github.io/convolutional-networks/)的Summary：



## Max pooling

在卷积后还会有一个pooling的操作，尽管有其他的比如average pooling等，这里只提max pooling。

max pooling的操作如下图所示：整个图片被不重叠的分割成若干个同样大小的小块（pooling size）。每个小块内只取最大的数字，再舍弃其他节点后，保持原有的平面结构得出output。

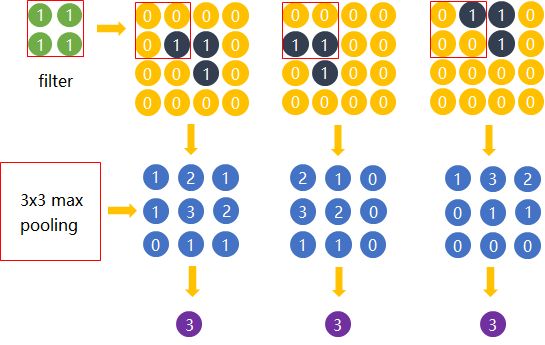


max pooling在不同的depth上是分开执行的，且不需要参数控制。 那么问题就max pooling有什么作用？部分信息被舍弃后难道没有影响吗？



Max pooling的主要功能是downsamping，却不会损坏识别结果。 这意味着卷积后的Feature Map中有对于识别物体不必要的冗余信息。 那么我们就反过来思考，这些“冗余”信息是如何产生的。

直觉上，我们为了探测到某个特定形状的存在，用一个filter对整个图片进行逐步扫描。但只有出现了该特定形状的区域所卷积获得的输出才是真正有用的，用该filter卷积其他区域得出的数值就可能对该形状是否存在的判定影响较小。 比如下图中，我们还是考虑探测“横折”这个形状。 卷积后得到3x3的Feature Map中，真正有用的就是数字为3的那个节点，其余数值对于这个任务而言都是无关的。 所以用3x3的Max pooling后，并没有对“横折”的探测产生影响。 试想在这里例子中如果不使用Max pooling，而让网络自己去学习。 网络也会去学习与Max pooling近似效果的权重。因为是近似效果，增加了更多的parameters的代价，却还不如直接进行Max pooling。



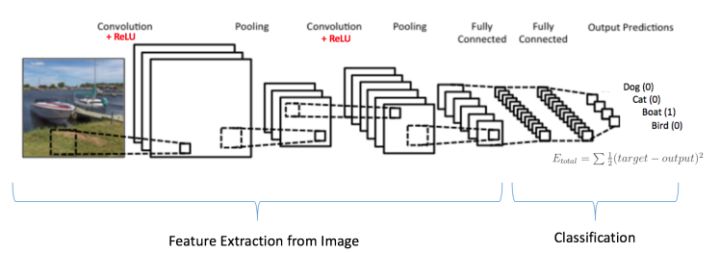
Max pooling还有类似“选择句”的功能。假如有两个节点，其中第一个节点会在某些输入情况下最大，那么网络就只在这个节点上流通信息；而另一些输入又会让第二个节点的值最大，那么网络就转而走这个节点的分支。

但是Max pooling也有不好的地方。因为并非所有的抓取都像上图的例子。有些周边信息对某个概念是否存在的判定也有影响。 并且Max pooling是对所有的Feature Maps进行等价的操作。就好比用相同网孔的渔网打鱼，一定会有漏网之鱼。

## 全连接层

当抓取到足以用来识别图片的特征后，接下来的就是如何进行分类。 全连接层（也叫前馈层）就可以用来将最后的输出映射到[线性可分的空间](https://link.zhihu.com/?target=https%3A//yjango.gitbooks.io/superorganism/content/ren_gong_shen_jing_wang_luo.html)。 通常卷积网络的最后会将末端得到的长方体平摊(flatten)成一个长长的向量，并送入全连接层配合输出层进行分类。

卷积神经网络大致就是covolutional layer, pooling layer, ReLu layer, fully-connected layer的组合，例如下图所示的结构。



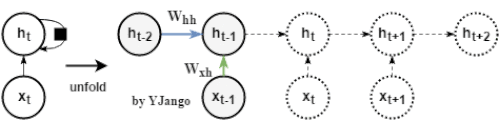
## 例子



# RNN

## 普通RNN

左侧是时间维度展开前，回路方式的表达方式，其中黑方框表示时间延迟。右侧展开后，可以看到当前时刻的并不仅仅取决于当前时刻的输入，同时与上一时刻的也相关。



数学式子：



其中激活函数一般为tanh。

## LSTM

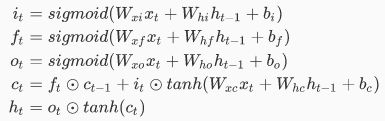
上面的现象可能并不意味着无法学习，但是即便可以，也会非常非常的慢。为了有效的利用梯度下降法学习，我们希望使不断相乘的梯度的积(the product of derivatives)保持在接近1的数值。

一种实现方式是建立线性自连接单元(linear self-connections)和在自连接部分数值接近1的权重，叫做leaky units。但Leaky units的线性自连接权重是手动设置或设为参数，而目前最有效的方式gated RNNs是通过gates的调控，**允许线性自连接的权重在每一步都可以自我变化调节。**LSTM就是gated RNNs中的一个实现。

### LSTM的初步理解

LSTM(或者其他gated RNNs)是在标准RNN的基础上装备了若干个控制数级(magnitude)的gates。可以理解成神经网络(RNN整体)中加入其他神经网络(gates)，而这些gates只是控制数级，控制信息的流动量。

**数学公式：**这里贴出基本LSTM的数学公式，看一眼就好，仅仅是为了让大家先留一个印象，不需要记住，不需要理解。



尽管式子不算复杂，却包含很多知识，接下来就是逐步分析这些式子以及背后的道理。 比如点乘的意义和使用原因，sigmoid的使用原因。

### 门(gate)的理解

理解Gated RNNs的第一步就是明白gate到底起到什么作用。

**物理意义**：gate本身可看成是十分有物理意义的一个神经网络。

输入：gate的输入是控制依据；

输出：gate的输出是值域为（0，1）数值，表示该如何调节其他数据的数级的控制方式。

**使用方式**：gate所产生的输出会用于控制其他数据的数级，相当于过滤器的作用。

类比图：可以把信息想象成水流，而gate就是控制多少水流可以流过。



例如：当用gate来控制向量[20 5 7 8]时，若gate的输出为[0.1 0.2 0.9 0.5]时，原来的向量就会被对应元素相乘(element-wise)后变成：



**控制依据**：明白了gate的输出后，剩下要确定以什么信息为控制依据，也就是什么是gate的输入。

例如：即便是LSTM也有很多个变种。一个变种方式是调控门的输入。

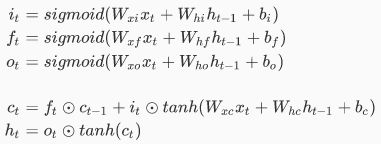


这种gate的输入有当前的输入和上一时刻的隐藏状态， 表示gate是将这两个信息流作为控制依据而产生输出的。

### LSTM的再次理解

明白了gate之后再回过头来看LSTM的数学公式

数学公式：



**gates：**先将前半部分的三个式子统一理解。在LSTM中，网络首先构建了3个gates来控制信息的流通量。

注： 虽然gates的式子构成方式一样，但是注意3个gates式子W和b的下角标并不相同。它们有各自的物理意义，在网络学习过程中会产生不同的权重。

有了这3个gates后，接下来要考虑的就是如何用它们装备在普通的RNN上来控制信息流，而根据它们所用于控制信息流通的地点不同，它们又被分为：

输入门：控制有多少信息可以流入memory cell（第四个式子）。

遗忘门：控制有多少上一时刻的memory cell中的信息可以累积到当前时刻的memory cell中。

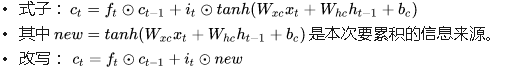
输出门：控制有多少当前时刻的memory cell中的信息可以流入当前隐藏状态中。

注：gates并不提供额外信息，gates只是起到限制信息的量的作用。因为gates起到的是过滤器作用，所以所用的激活函数是sigmoid而不是tanh。

**信息流：**信息流的来源只有三处，当前的输入，上一时刻的隐藏状态，上一时刻的cell状态，其中是额外制造出来、可线性自连接的单元（请回想起leaky units）。真正的信息流来源可以说只有当前的输入，上一时刻的隐藏状态两处。三个gates的控制依据，以及数据的更新都是来源于这两处。

分析了gates和信息流后，再分析剩下的两个等式，来看LSTM是如何累积历史信息和计算隐藏状态的。

**历史信息累积：**



所以历史信息的累积是并不是靠隐藏状态h自身，而是依靠memory cell这个自连接来累积。 在累积时，靠遗忘门来限制上一时刻的memory cell的信息，并靠输入门来限制新信息。并且真的达到了leaky units的思想，memory cell的自连接是线性的累积。

**当前隐藏状态的计算：**如此大费周章的最终任然是同普通RNN一样要计算当前隐藏状态。

式子：



当前隐藏状态是从计算得来的，因为是以线性的方式自我更新的，所以先将其加入带有非线性功能的tanh()。 随后再靠输出门的过滤来得到当前隐藏状态。

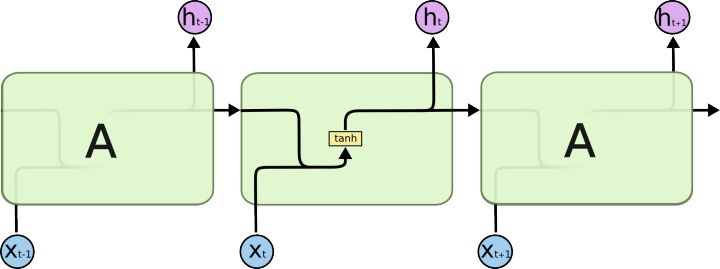
普通RNN与LSTM的比较

下面为了加深理解循环神经网络的核心，再来一起比较一下普通RNN和LSTM的区别。

**比较：**最大的区别是多了三个神经网络(gates)来控制数据的流通。

普通RNN：





LSTM：





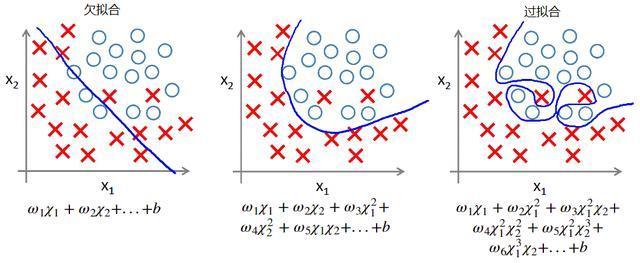
**比较：**二者的信息来源都是，不同的是LSTM靠3个gates将信息的积累建立在线性自连接的memory cell之上，并靠其作为中间物来计算当前。

# 过拟合和欠拟合

## 定义

欠拟合（under-fitting）也称为欠学习，它的直观表现是算法训练得到的模型在训练集上表现差，没有学到数据的规律。引起欠拟合的原因有：模型本身过于简单，例如数据本身是非线性的但使用了线性模型；特征数太少无法正确的建立统计关系。

过拟合（overfitting）是指在模型参数拟合过程中的问题，由于训练数据包含**抽样误差**，训练时，复杂的模型将抽样误差也考虑在内，将抽样误差也进行了很好的拟合。具体表现就是最终模型在训练集上效果好；在测试集上效果差。模型泛化能力弱。



过拟合与欠拟合的判断标准



## 模型出现过拟合现象的原因

发生过拟合的主要原因可以有以下三点：

（1）数据有噪声

（2）训练数据不足，有限的训练数据

（3）训练模型过度导致模型非常复杂

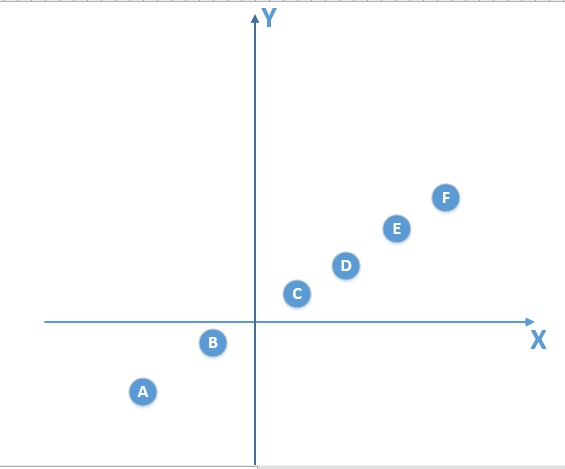
### 数据有噪声

为什么数据有噪声，就可能导致模型出现过拟合现象呢？

所有的机器学习过程都是一个search假设空间的过程！我们是在模型参数空间搜索一组参数，使得我们的损失函数最小，也就是不断的接近我们的真实假设模型，而真实模型只有知道了所有的数据分布，才能得到。

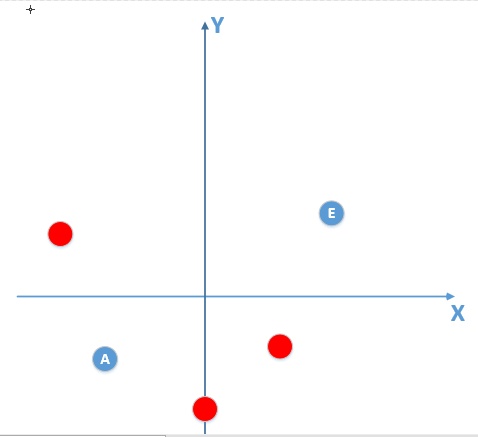
**往往我们的模型是在训练数据有限的情况下，找出使损失函数最小的最优模型，然后将该模型泛化于所有数据的其它部分。这是机器学习的本质！**

那好，假设我们的总体数据如下图所示：



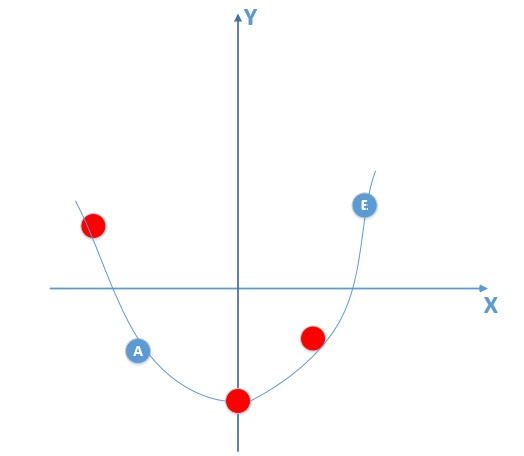
（我这里就假设总体数据分布满足一个线性模型y = kx+b,现实中肯定不会这么简单，数据量也不会这么少，至少也是多少亿级别，但是不影响解释。反正总体数据满足模型y）

此时我们得到的部分数据，还有噪声的话，如图所示：



（红色数据点为噪声）

那么由上面训练数据点训练出来的模型肯定不是线性模型（**总体数据分布下满足的标准模型**），比如训练出来的模型如下：

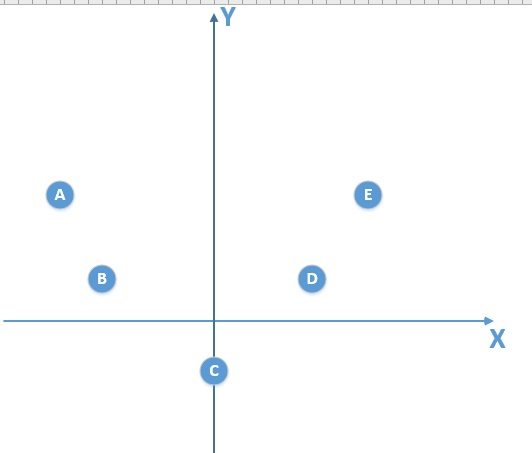


那么我拿着这个有噪声训练的模型，在训练集合上通过不断训练，可以做到损失函数值为0，但是拿着这个模型，到真实总体数据分布中（满足线性模型）去泛化，效果会非常差，因为你拿着一个非线性模型去预测线性模型的真实分布，显而易得效果是非常差的，也就产生了过拟合现象！

### 训练数据不足，有限的训练数据

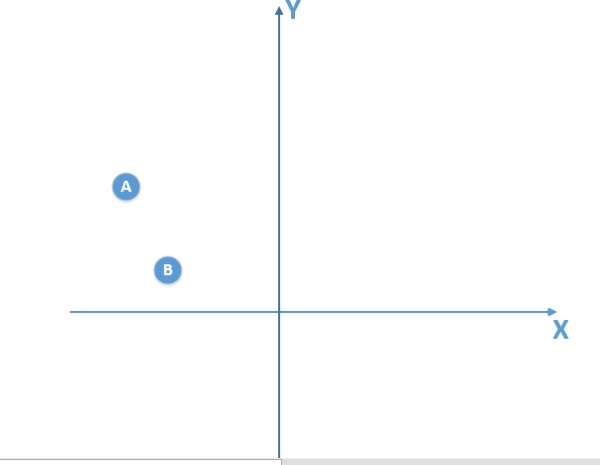
当我们训练数据不足的时候，即使得到的训练数据没有噪声，训练出来的模型也可能产生过拟合现象，解释如下：

假设我们的总体数据分布如下：



（为了容易理解，假设我们的总体数据分布满足的模型是一个二次函数模型）

我们得到的训练数据由于是有限的，比如是下面这个：



（我只得到了A,B俩个训练数据）

那么由这个训练数据，我得到的模型是一个线性模型，通过训练较多的次数，我可以得到在训练数据使得损失函数为0的线性模型，拿这个模型我去泛化真实的总体分布数据（实际上是满足二次函数模型），很显然，泛化能力是非常差的，也就出现了过拟合现象。

### 训练模型过度导致模型非常复杂

训练模型过度导致模型非常复杂，也会导致过拟合现象！这点和第一点俩点原因结合起来其实非常好理解，当我们在训练数据训练的时候，如果训练过度，导致完全拟合了训练数据的话，得到的模型不一定是可靠的。

比如说，在有噪声的训练数据中，我们要是训练过度，会让模型学习到噪声的特征，无疑是会造成在没有噪声的真实测试集上准确率下降！

模型越复杂，越容易过拟合。

## 如何防止过拟合

### 增加数据或Dada Augmentation（数据角度）

“数据和特征决定了机器学习的上限，而模型和算法只是逼近这个上限而已！“通过数据集扩增得到更多符合要求的数据，这样目标空间（target domain）和源空间（source domain）的数据分布（distribution）越趋近于一致，从而减少了过拟合的风险。

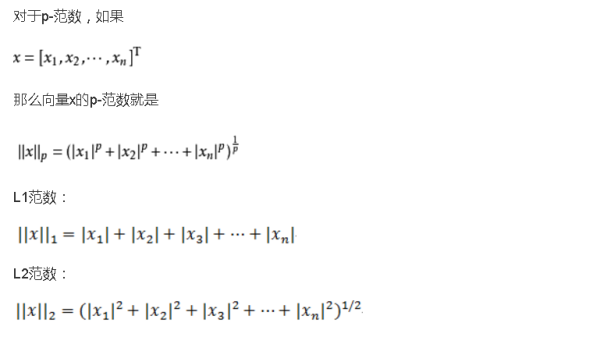


### L1 L2正则化(模型角度)

**什么是范数：**

范数是具有“长度”概念的函数。在向量空间内，为所有的向量的赋予非零的增长度或者大小。不同的范数，所求的向量的长度或者大小是不同的。

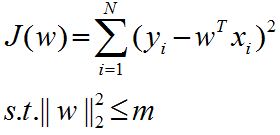
举个例子，2维空间中，向量(3,4)的长度是5，那么5就是这个向量的一个范数的值，更确切的说，是欧式范数或者L2范数的值。



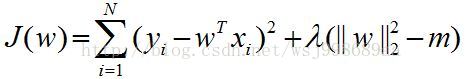
特别的，L0范数：指向量中非零元素的个数。无穷范数：指向量中所有元素的最大绝对值。

**L1,L2正则化：**

正则化的本质：约束（限制）要优化的参数**，**将模型复杂度加入损失函数，在原有的损失函数基础上加上模型的约束条件，

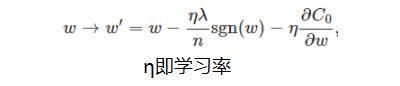


然后用拉格朗日乘子法。

****

L1正则化，即在原损失函数的基础上加上L1范数的限制。

根据损失更新ω，需要先对ω求导：那么权重ω的更新规则为：

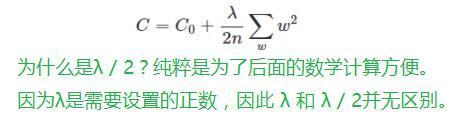


比原始的更新规则多出了η \* λ \* sgn(ω)/n。可见每次更新，



ω都是往0靠，即使网络中的权重尽可能为0。

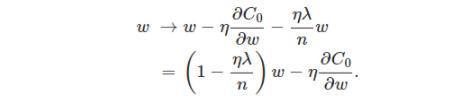
L2正则化，即在原损失函数的基础上加上L2范数的限制



同样，需要先对ω求导：

****

那么权重ω的更新规则为：

****

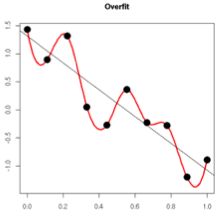
原始的更新规则ω系数为1，现在为 1-ηλ/n 。

L1 正则化可以理解为每次从权重中减去一个常数。L2 正则化可以理解为每次移除权重的 x%。本质都是为了降低模型的复杂度，防止过拟合。

**为什么减小系数能防止过拟合：**

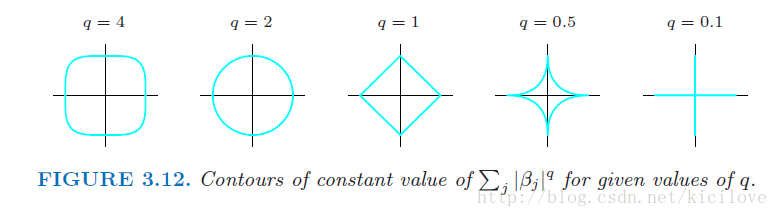
过拟合的时候，拟合函数的系数往往非常大，而正则化是通过约束参数的范数使其不要太大，所以可以在一定程度上减少过拟合情况。

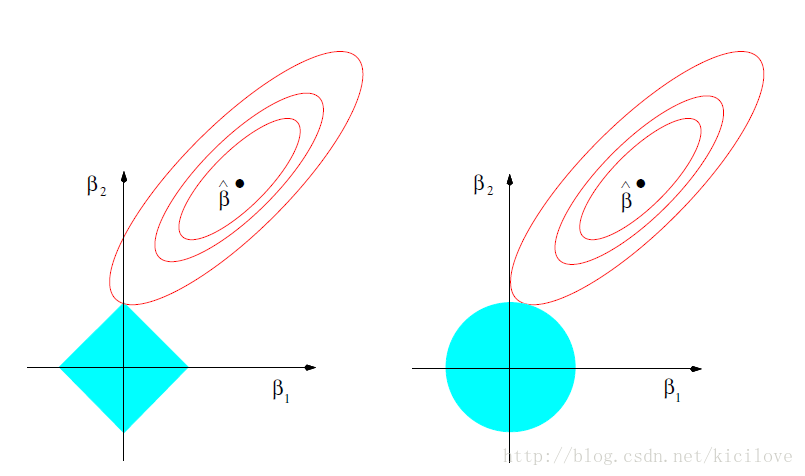
说说为什么过拟合的时候系数会很大。如下图所示，过拟合，就是拟合函数需要顾忌每一个点，最终形成的拟合函数波动很大。在某些很小的区间里，函数值的变化很剧烈。这就意味着函数在某些小区间里的导数值（绝对值）非常大，由于自变量值可大可小，所以只有系数足够大，才能保证导数值很大。



**为什么L1正则化会导致稀疏：**

其中，q=1时即为L1正则化，q=2为L2正则化。对于q取不同的值，正则化项的轮廓线如下：



****

上面的图中实心的黑点也就是 β^是真实的损失函数（不带有正则项的部分）我们暂叫做原问题的最优解，然后红色的圈圈就是系数β1、β2在原问题下可能的解的范围，接着是蓝色的实心圈是正则项约束的可能的解的范围。我们都知道如果两个函数要是有共同的解，那么在几何意义下或者说从几何图形上来看，这两个函数的图像所在范围是要有共同交点或者要有交集。L1范数意义下可能得到有的维度上的系数为零（就是切点所在的坐标处，β1坐标为0），这也就是为什么说L1可以导致稀疏解，同理L2范数意义下，相切的点就不在坐标轴上，β1和β2都不取0。

**稀疏的好处：**

上面我们反复提到了稀疏，也知道了L1范数可以实现稀疏，那么稀疏有什么好处呢？

1.特征选择，稀疏的矩阵可以起到特征选择的作用。我们知道，稀疏矩阵是只有少数值为非零，大多数值为0的矩阵。有些场景中特征数量可能会非常多，不利于直接使用机器学习算法。如果这是可以将参数矩阵转换稀疏的，那么就可以只保留一小部分特征（只有系数为非零的那一部分特征被保留）。这样转换成稀疏矩阵就可以起到特征选择的作用。

2.可解释性。另一个青睐于稀疏的理由是，模型更容易解释。例如患某种病的概率是y，然后我们收集到的数据x是1000维的，也就是我们需要寻找这1000种因素到底是怎么影响患上这种病的概率的。假设我们这个是个回归模型：y=w1\*x1+w2\*x2+…+w1000\*x1000+b（当然了，为了让y限定在[0,1]的范围，一般还得加个Logistic函数）。通过学习，如果最后学习到的w\*就只有很少的非零元素，例如只有5个非零的wi，那么我们就有理由相信，这些对应的特征在患病分析上面提供的信息是巨大的，决策性的。也就是说，患不患这种病只和这5个因素有关，那医生就好分析多了。

**λ变大，菱形和圆形怎么变化**

λ越大，菱形和圆形越小，求得的模型参数越小。

**为什么 L2 正则化比 L1 正则化应用更加广泛**

因为 L2 正则化的约束边界光滑且可导，便于采用梯度下降法，而L1正则化不可导，只能采用坐标轴下降法或最小角回归法，计算量大。而且，L1 正则化的效果并不会比 L2 正则化好。

**总结一下L1与L2正则化：**

L1范数：L1范数在正则化的过程中会趋向于产生少量的特征，而其他的特征都是0（L1会使得参数矩阵变得稀疏）。因此L1不仅可以起到正则化的作用，还可以起到特征选择的作用。

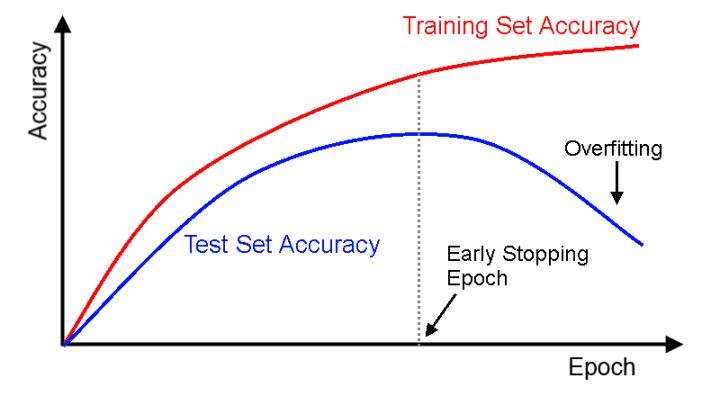
L2范数：L2范数是通过使权重衰减，进而使得特征对于总体的影响减小而起到防止过拟合的作用的。L2的优点在于求解稳定、快速。

L1正则化就是在loss function后边所加正则项为L1范数，加上L1范数容易得到稀疏解（0比较多）。L2正则化就是loss function后边所加正则项为L2范数的平方，加上L2正则相比于L1正则来说，得到的解比较平滑（不是稀疏），但是同样能够保证解中接近于0（但不是等于0，所以相对平滑）的维度比较多，降低模型的复杂度。

### Early Stopping（训练角度）

对模型进行训练的过程即是对模型的参数进行学习更新的过程，这个参数学习的过程往往会用到一些迭代方法，如梯度下降（Gradient descent）学习算法。Early stopping便是一种迭代次数截断的方法来防止过拟合的方法，即在模型对训练数据集迭代收敛之前停止迭代来防止过拟合。

Early stopping方法的具体做法是：在每一个Epoch结束时计算validation data的accuracy，当accuracy不再提高时，就停止训练。这种做法很符合直观感受，因为accurary都不再提高了，在继续训练也是无益的，只会提高训练的时间。并不是说validation accuracy一降下来便认为不再提高了，因为可能经过这个Epoch后，accuracy降低了，但是随后的Epoch又让accuracy又上去了，所以不能根据一两次的连续降低就判断不再提高。一般的做法是，在训练的过程中，记录到目前为止最好的**validation accuracy**，当连续**10**次**Epoch**（或者更多次）没达到最佳**accuracy**时，则可以认为**accuracy**不再提高了。此时便可以停止迭代了。



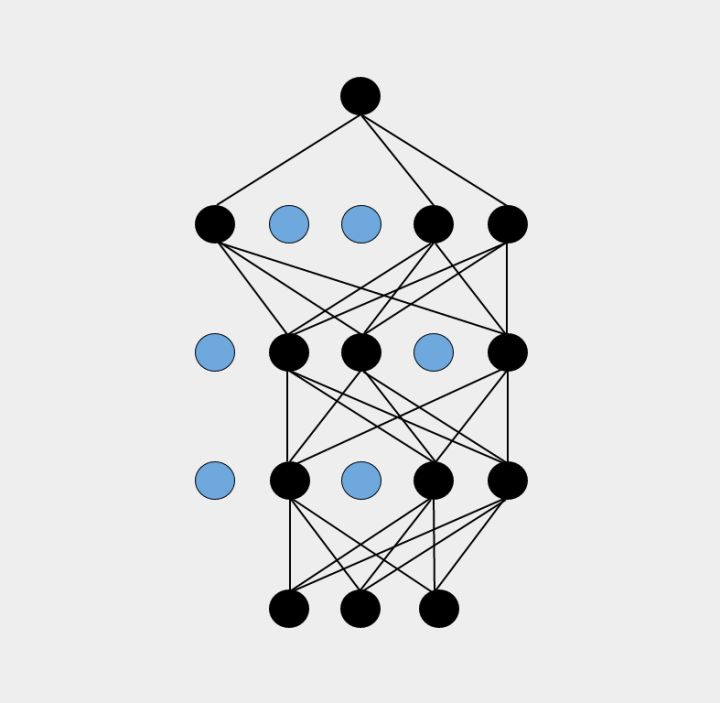
### Dropout（训练角度）

Dropout是指在深度学习网络的训练过程中，对于指定的网络层，以一定的概率使某些神经元停止工作。对于随机梯度下降来说，由于是随机丢弃，故而每一个mini-batch都在训练不同的网络。

每次从原始的网络中找到一个更瘦的网络进行训练，计算更快；

Droupout是特殊的Ensemble，即每次做完dropout后，都会产生一个新的神经网络模型，dropout rate = 0.5的时候随机生成的网络结构最多。

强迫每一个神经单元和随机挑选出来的其他神经单元共同工作，消除减弱了神经元节点间的联合适应性，增强了泛化能力。



# 模型的保存和恢复

## 什么是TensorFlow模型

训练了一个神经网络之后，我们希望保存它以便将来使用。那么什么是TensorFlow模型?Tensorflow模型主要包含我们所培训的网络参数的网络设计或图形和值。因此，Tensorflow模型有两个主要的文件:

a) Meta graph：  
     这是一个协议缓冲区，它保存了完整的Tensorflow图形;即所有变量、操作、集合等。该文件以.meta作为扩展名。

b) Checkpoint file：

    这是一个二进制文件，它包含了所有的权重、偏差、梯度和其他所有变量的值。这个文件有一个扩展名.ckpt。然而，Tensorflow从0.11版本中改变了这一点。现在，我们有两个文件，而不是单个.ckpt文件:

1. mymodel.data-00000-of-00001
2. mymodel.index

.data文件是包含我们训练变量的文件，我们待会将会使用它。

与此同时，Tensorflow也有一个名为checkpoint的文件，它只保存的最新保存的checkpoint文件的记录。

因此，为了总结，对于大于0.10的版本，Tensorflow模型如下:

在0.11之前的Tensorflow模型仅包含三个文件:

1. inception\_v1.meta
2. inception\_v1.ckpt
3. checkpoint

现在我们已经知道了Tensorflow模型的样子，接下来我们来看看TensorFlow是如何保存模型的。

## 保存TensorFlow模型

比方说，你正在训练一个卷积神经网络来进行图像分类。作为一种标准的练习，你要时刻关注损失和准确率。一旦看到网络已经收敛，我们可以暂停模型的训练。在完成培训之后，我们希望将所有的变量和网络结构保存到一个文件中，以便将来使用。因此，在Tensorflow中，我们希望保存所有参数的图和值，我们将创建一个tf.train.Saver()类的实例。

saver = tf.train.Saver()

请记住，Tensorflow变量仅在会话中存在。因此，您必须在一个会话中保存模型，调用您刚刚创建的save方法。

saver.save(sess, 'my-test-model')

这里，sess是会话对象，而'my-test-model'是保存的模型的名称。让我们来看一个完整的例子:

1. import tensorflow as tf
2. w1 = tf.Variable(tf.random\_normal(shape=[2]), name='w1')
3. w2 = tf.Variable(tf.random\_normal(shape=[5]), name='w2')
4. saver = tf.train.Saver()
5. sess = tf.Session()
6. sess.run(tf.global\_variables\_initializer())
7. saver.save(sess, 'my\_test\_model')
9. *# This will save following files in Tensorflow v >= 0.11*
10. *# my\_test\_model.data-00000-of-00001*
11. *# my\_test\_model.index*
12. *# my\_test\_model.meta*
13. *# checkpoint*

如果我们在1000次迭代之后保存模型，我们将通过通过global\_step来调用save:

saver.save(sess, 'my\_test\_model',global\_step=1000)

这将会将'-1000'追加到模型名称，并创建以下文件:

1. my\_test\_model-1000.index
2. my\_test\_model-1000.meta
3. my\_test\_model-1000.data-00000-of-00001
4. checkpoint

比方说，在训练时，我们在每次1000次迭代后都保存模型，所以.meta文件是第一次创建的(在第1000次迭代中)，我们不需要每次都重新创建.meta文件(我们在2000，3000次没有保存.meta文件)。我们仅为进一步的迭代保存模型，因为图不会改变。因此，当我们不想保存meta-graph时，我们用这个:

saver.save(sess, 'my-model', global\_step=step,write\_meta\_graph=False)

如果你希望仅保留4个最新的模型，并且希望在训练过程中每两个小时后保存一个模型，那么你可以使用max\_to\_keep和keep\_checkpoint\_every\_n\_hours这样做。

1. *#saves a model every 2 hours and maximum 4 latest models are saved.*
2. saver = tf.train.Saver(max\_to\_keep=4, keep\_checkpoint\_every\_n\_hours=2)

注意，如果我们在tf.train.Saver()中没有指定任何东西，它将保存所有的变量。如果，我们不想保存所有的变量，而只是一些变量。我们可以指定要保存的变量/集合。在创建tf.train。保护程序实例，我们将它传递给我们想要保存的变量的列表或字典。让我们来看一个例子:

1. import tensorflow as tf
2. w1 = tf.Variable(tf.random\_normal(shape=[2]), name='w1')
3. w2 = tf.Variable(tf.random\_normal(shape=[5]), name='w2')
4. saver = tf.train.Saver([w1,w2])
5. sess = tf.Session()
6. sess.run(tf.global\_variables\_initializer())
7. saver.save(sess, 'my\_test\_model',global\_step=1000)

这可以用于在需要时保存特定的Tensorflow图。

## 导入训练好的模型

如果你想用别人预先训练好的模型来进行微调，你需要做以下两件事:

**a)创建网络**

你可以通过编写python代码创建网络，以手工创建每一层，并将其作为原始模型。但是，如果你考虑一下，我们已经在.meta文件中保存了这个网络，我们可以使用tf.train.import()函数来重新创建这个网络：

saver = tf.train.import\_meta\_graph('my\_test\_model-1000.meta')

记住，import\_meta\_graph将在.meta文件中定义的网络附加到当前图。因此，这将为你创建图形/网络，但是我们仍然需要加载我们在这张图上训练过的参数的值。

b)载入参数

我们可以通过调用这个保护程序的实例来恢复网络的参数，它是tf.train.Saver()类的一个实例。

1. with tf.Session() as sess:
2. new\_saver = tf.train.import\_meta\_graph('my\_test\_model-1000.meta')
3. new\_saver.restore(sess, tf.train.latest\_checkpoint('./'))

在此之后，像w1和w2这样的张量的值已经恢复并且可以被访问:

1. with tf.Session() as sess:
2. saver = tf.train.import\_meta\_graph('my-model-1000.meta')
3. saver.restore(sess,tf.train.latest\_checkpoint('./'))
4. print(sess.run('w1:0'))
5. *##Model has been restored. Above statement will print the saved value of w1*

因此，现在你已经了解了如何为Tensorflow模型保存和导入工作。在下一节中，我描述了上面的实际使用，以加载任何预先训练过的模型。

# TensorBoard

## TensorBoard 简介及使用流程

### TensoBoard 简介

TensorBoard 和 TensorFLow 程序跑在不同的进程中，TensorBoard 会自动读取最新的 TensorFlow 日志文件，并呈现当前 TensorFLow 程序运行的最新状态。

### TensorBoard 使用流程

* 添加记录节点：tf.summary.scalar/image/histogram()等
* 汇总记录节点：merged = tf.summary.merge\_all()
* 运行汇总节点：summary = sess.run(merged)，得到汇总结果
* 日志书写器实例化：summary\_writer = tf.summary.FileWriter(logdir, graph=sess.graph)，实例化的同时传入 graph 将当前计算图写入日志
* 调用日志书写器实例对象summary\_writer的add\_summary(summary, global\_step=i)方法将所有汇总日志写入文件
* 调用日志书写器实例对象summary\_writer的close()方法写入内存，否则它每隔120s写入一次

## TensorFlow 可视化分类

### 计算图的可视化：add\_graph()

...create a graph...

# Launch the graph in a session.

sess = tf.Session()

# Create a summary writer, add the 'graph' to the event file.

writer = tf.summary.FileWriter(logdir, sess.graph)

writer.close() # 关闭时写入内存，否则它每隔120s写入一次

### 监控指标的可视化：add\_summary()

I、SCALAR

tf.summary.scalar(name, tensor, collections=None, family=None)

* 可视化训练过程中随着迭代次数准确率(val acc)、损失值(train/test loss)、学习率(learning rate)、每一层的权重和偏置的统计量(mean、std、max/min)等的变化曲线
* 输入参数：

name：此操作节点的名字，TensorBoard 中绘制的图形的纵轴也将使用此名字

tensor： 需要监控的变量 A real numeric Tensor containing a single value.

* 输出：

A scalar Tensor of type string. Which contains a Summary protobuf.

II、IMAGE

tf.summary.image(name, tensor, max\_outputs=3, collections=None, family=None)

* 可视化当前轮训练使用的训练/测试图片或者 feature maps
* 输入参数：

name：此操作节点的名字，TensorBoard 中绘制的图形的纵轴也将使用此名字

tensor： A r A 4-D uint8 or float32 Tensor of shape [batch\_size, height, width, channels] where channels is 1, 3, or 4

max\_outputs：Max number of batch elements to generate images for

* 输出：

A scalar Tensor of type string. Which contains a Summary protobuf.

III、HISTOGRAM

tf.summary.histogram(name, values, collections=None, family=None)

* 可视化张量的取值分布
* 输入参数：

name：此操作节点的名字，TensorBoard 中绘制的图形的纵轴也将使用此名字

tensor： A real numeric Tensor. Any shape. Values to use to build the histogram

* 输出：

A scalar Tensor of type string. Which contains a Summary protobuf.

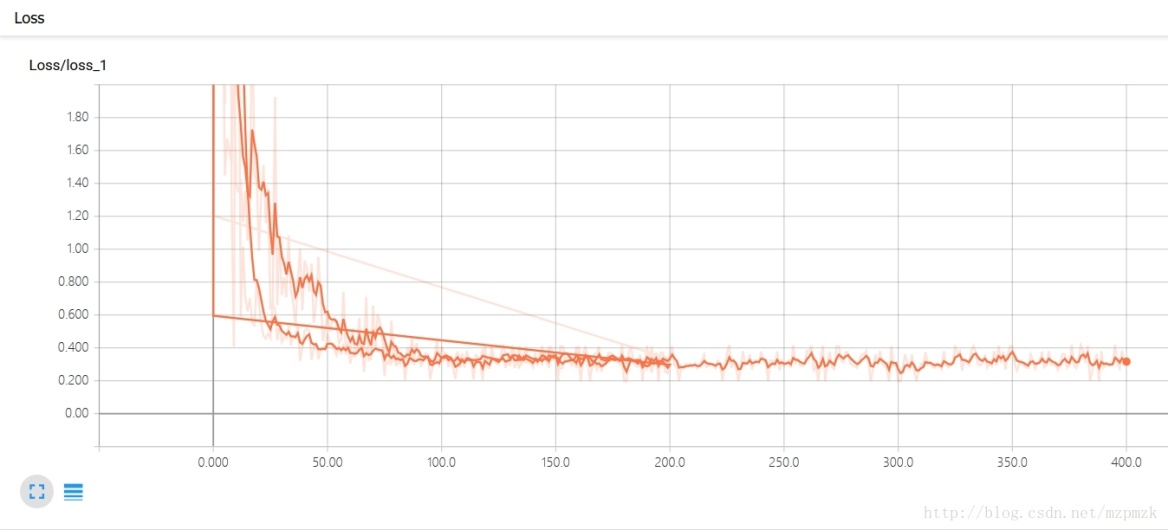
IV、MERGE\_ALL

tf.summary.merge\_all(key=tf.GraphKeys.SUMMARIES)

* Merges all summaries collected in the default graph
* 因为程序中定义的写日志操作比较多，一一调用非常麻烦，所以TensoorFlow 提供了此函数来整理所有的日志生成操作，eg：merged = tf.summary.merge\_all ()
* 此操作不会立即执行，所以，需要明确的运行这个操作(summary = sess.run(merged))来得到汇总结果
* 最后调用日志书写器实例对象的add\_summary(summary, global\_step=i)方法将所有汇总日志写入文件

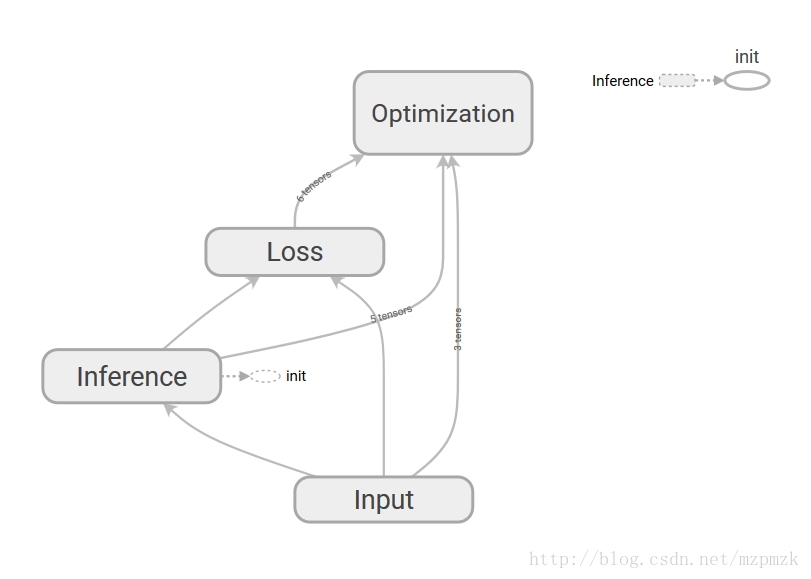
### 多个事件(event)的可视化：add\_event()

* 如果 logdir 目录的子目录中包含另一次运行时的数据(多个 event)，那么 TensorBoard 会展示所有运行的数据(主要是scalar)，这样可以用于比较不同参数下模型的效果，调节模型的参数，让其达到最好的效果！
* 上面那条线是迭代200次的loss曲线图，下面那条是迭代400次的曲线图，程序见最后。

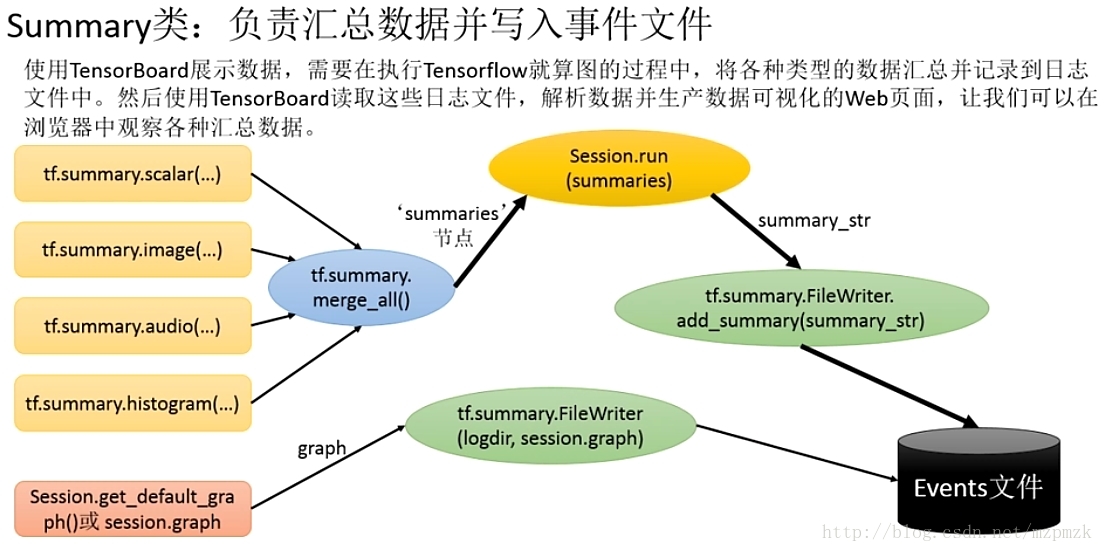


## 通过命名空间美化计算图

* 使用命名空间使可视化效果图更有层次性，使得神经网络的整体结构不会被过多的细节所淹没
* 同一个命名空间下的所有节点会被缩略成一个节点，只有顶层命名空间中的节点才会被显示在 TensorBoard 可视化效果图上
* 可通过tf.name\_scope()或者tf.variable\_scope()来实现，具体见最后的程序。



## [将所有日志写入到文件：tf.summary.FileWriter()](https://www.tensorflow.org/api_docs/python/tf/summary/FileWriter" \t "_blank)



tf.summary.FileWriter(logdir, graph=None, flush\_secs=120, max\_queue=10)

* + 负责将事件日志(graph、scalar/image/histogram、event)写入到指定的文件中
  + 初始化参数：

logdir：事件写入的目录

graph：如果在初始化的时候传入sess,graph的话，相当于调用add\_graph() 方法，用于计算图的可视化

flush\_sec：How often, in seconds, to flush the added summaries and events to disk.

max\_queue：Maximum number of summaries or events pending to be written to disk before one of the ‘add’ calls block.

* + 其它常用方法：

add\_event(event)：Adds an event to the event file

add\_graph(graph, global\_step=None)：Adds a Graph to the event file，Most users pass a graph in the constructor instead

add\_summary(summary, global\_step=None)：Adds a Summary protocol buffer to the event file，一定注意要传入 global\_step

close()：Flushes the event file to disk and close the file

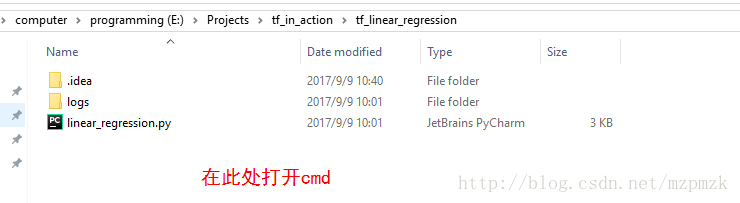
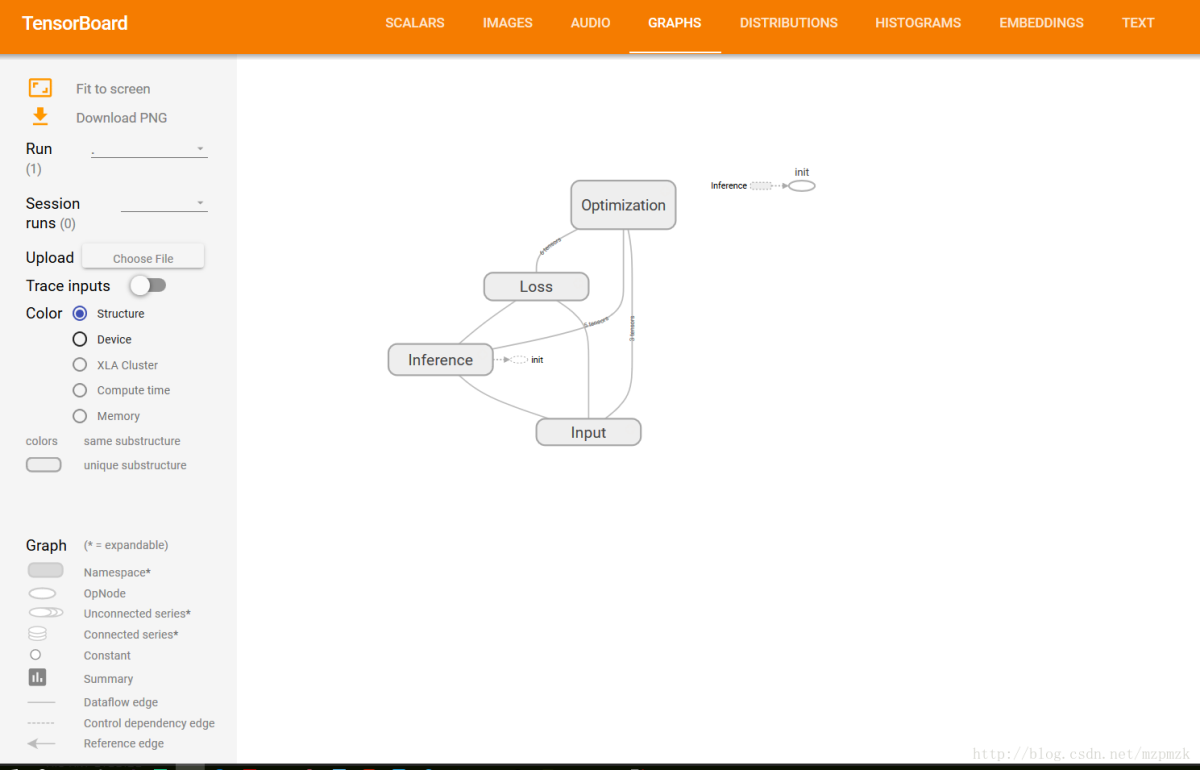
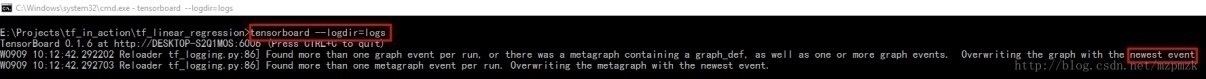
flush()：Flushes the event file to disk

add\_meta\_graph(meta\_graph\_def,global\_step=None)

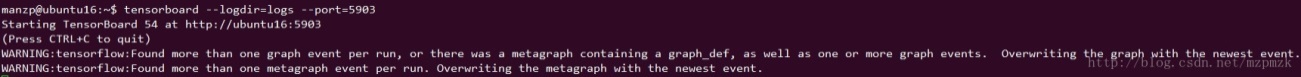
add\_run\_metadata(run\_metadata, tag, global\_step=None)

## 启动 TensorBoard 展示所有日志图表

### 通过 Windows 下的 cmd 启动

* 运行你的程序，在指定目录下(logs)生成 event 文件
* 在 logs 所在目录，按住 shift 键，点击右键选择在此处打开 cmd
* 在 cmd 中，输入以下命令启动 tensorboard --logdir=logs，注意：logs的目录并不需要加引号, logs 中有多个event 时，会生成scalar 的对比图，但 graph 只会展示最新的结果
* 把下面生成的网址(http://DESKTOP-S2Q1MOS:6006 # 每个人的可能不一样) copy 到浏览器中打开即可 ,注意：360浏览器不能用。
*  

## 通过 Ubuntu下的 bash 启动

* 运行你的程序(python my\_program.py)，在指定目录下(logs)生成 event 文件
* 在 bash 中，输入以下命令启动 tensorboard --logdir=logs --port=8888，注意：logs的目录并不需要加引号，端口号必须是事先在路由器中配置好的
* 把下面生成的网址(http://ubuntu16:8888 # 把ubuntu16 换成服务器的外部ip地址即可) copy 到本地浏览器中打开即可   
  

## 例子



# Keras, TensorLayer, TFLearn

首先对于初学者来说，这几个库的关系搞清楚，基本就成功一大半了。其中tensorflow和theano是一级的，他们都是最基础的框架，其完备程度更像是一种编程语言了，所以这两个框架不只是可以用来做DL，还可以写别的东西。keras、tflearn和tensorlayer都是在此基础上搭建的二次封装库，其中keras可以使用theano和tf做后端，剩余两个只针对tf。

## keras

难易程度上而言，keras是最简单的，封装非常完善，底层不透明，随意切换后端、任意切换CPU\GPU而基本不用改动代码，非常适合初学者。我觉得keras最好的一点是，对于theano和tf这样奇怪的图模型计算模式做了很好的屏蔽，使得整个编程的流程更加像是一般的代码程序。一般初学者都会对先架构计算图然后喂进数据执行运算很困惑，然而keras尽了最大可能屏蔽掉了这种模式机制，让整个程序看起来更像是一般的程序，甚至不懂theano和tf的人可以直接拿来上手干活。当然，keras的缺点也很明显，速度较原生的tf和theano要慢一些，而且由于封装过于完善，灵活性就降低了。

优点：Theano时代就推出了，使用者较多，有个人维护的中文文档，虽然更新很慢。纯scikit-learn式编程，隐藏了数据流的细节，数据流没有叙述的很清楚，到目前为止，并没有理解keras的时间数据输入方式。

缺点：运行TensorFlow时很慢，拓展性差（国外测评说是因为 Keras 最开始只是为了 Theano 而开发的，TensorFlow发布后才写支持TensorFlow的代码，所以为了兼容牺牲了效率）。不适合科研单位和企业，换句话说毕业以后要重新学另外一个框架..... 复杂的情况会出现error，这时候调试只能靠运气了。

## tensorlayer

tensorlayer是新出现的一个顶层库，他的好处是对tf做了一个恰到好处的补充。tf有一个很大的问题就是太过底层，如果你要架构一个复杂网络，你和希望一些常规的部件可以被直接调用而不用过分的自己操心。这时tl是绝佳选择，他的代码几乎是透明的，你心里很清楚数据在这些封装好的函数背后是怎么流动的，这一点强于keras，速度也快于keras，但是其封装效果就弱了一些，因此需要懂一些tf的运作机制才能顺利完成任务。

优点：对学术界的优势是灵活性很强，甚至可以很简单地实现动态网络结构（Neural Modular Network）.... 对工业界的优势是运行速度快。教程很强大，还包含了 Google TensorFlow 官网的模块化实现。同时提供scikit-learn式的API，和专业级的API，适合新手到老手过渡。  
 缺点：刚刚推出（16年8月份）使用者少，但我发现中文使用者不少，有很多微信群、qq群可以交流，有可能和TL官方推出中文文档有关。

## Tflearn

优点：不像Keras那样兼容两种后端，所以效率比 Keras 快，但根据国外测评还是比 TensorLayer慢一些。  
缺点：维护不好

## 结论

如果只是想玩玩深度学习，想快速上手 -- Keras

如果工作中需要解决内部问题，想快速见效果 -- TFLearn 或者 Tensorlayer

如果正式发布的产品和业务，自己设计网络模型，需要持续开发和维护 -- Tensorlayer

# tensorflow 相关记录

计算交叉熵：

y = tf.nn.softmax(output)

cross\_entropy = tf.reduce\_mean(-tf.reduce\_sum(y\_ \* tf.log(y)))

或者

cross\_entropy=tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=output, labels=y\_))

计算准确率：

correct\_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y\_, 1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

# PYTHONPATH

PYTHONPATH是Python搜索路径，默认我们import的模块都会从PYTHONPATH里面寻找。若要让 Python 解释器找到自己编写的模块,该模块必须在PYTHONPATH 上，否则在导入该模块时会出现找不到该模块的错误，因此必须把所需要的模块的路径添加到PYTHONPATH。

打印PYTHONPATH：

import os

print sys.path

注意：sys.path 也可以用 os.sys.path 替换，两个应该是同一个命令，推荐使用sys.path, 因为 os.sys.path 在python document 中好像没有提及．

设置PYTHONPATH：

Linux:

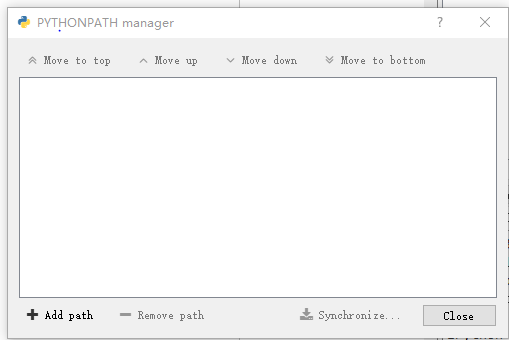
方法一：命令窗口添加路径

export PYTHONPATH=$PYTHONPATH:/home/ershisui

注意：此方法只在当前命令窗口生效，即如果打开一个新的Terminal 窗口，定位到当前目录，　打印PYTHONPATH 是没有刚才加入的路径的．

Windows:

设置环境变量，或在anaconda tools🡪pythonpath里添加。

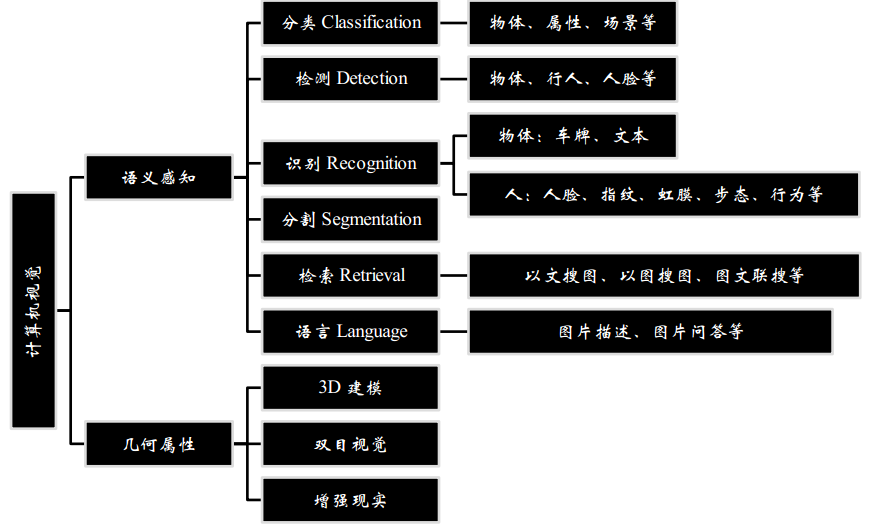


方法二：在python 中添加：

import sys

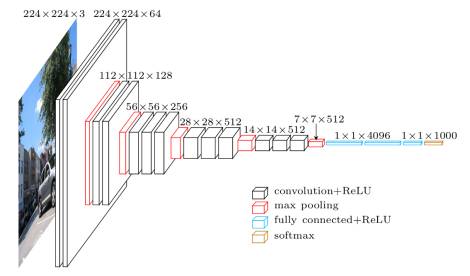
sys.path.append('/home/ershisui/')

# 计算机视觉（CV）

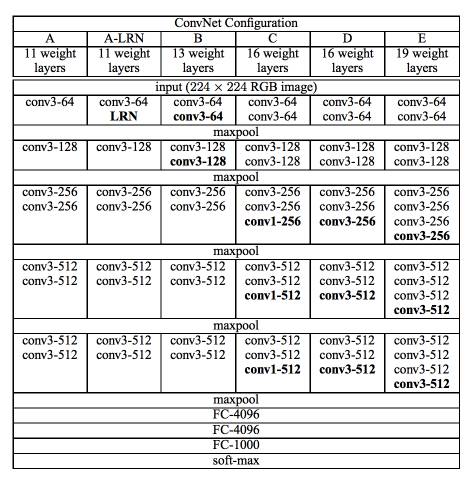


# 图像分类

## VGG

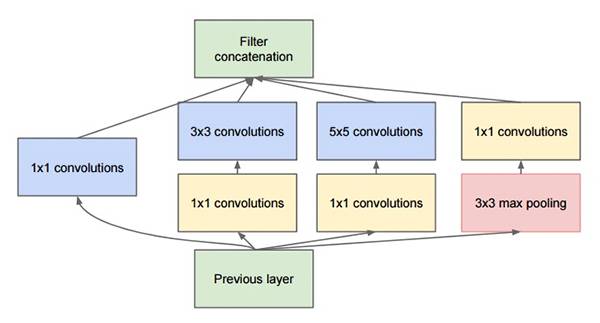


VGG16和VGG19: 该架构仅仅使用堆放在彼此顶部、深度不断增加的3×3卷积层，并通过max pooling来减小volume规格；然后是两个4096节点的全连接层，最后是一个softmax分类器。“16”和“19”代表网络中权重层的数量（表2中的D和E列）:



## Inception

GoogLeNet 由 Google 公司提出，性能与 VGGNet 相近。 GoogLeNet 的创新在于2提出了一种“ Inception”结构，它把原来的单个结点又拆成一个神经网络，形成了 “网 中网 ”（ Network in Network ）。 Inception 单元的结构。



整个 GoogLeNet 就是由许多这样的 Inception 单元组成的，色的构造比较复杂，但同样是深层的卷积神经网络

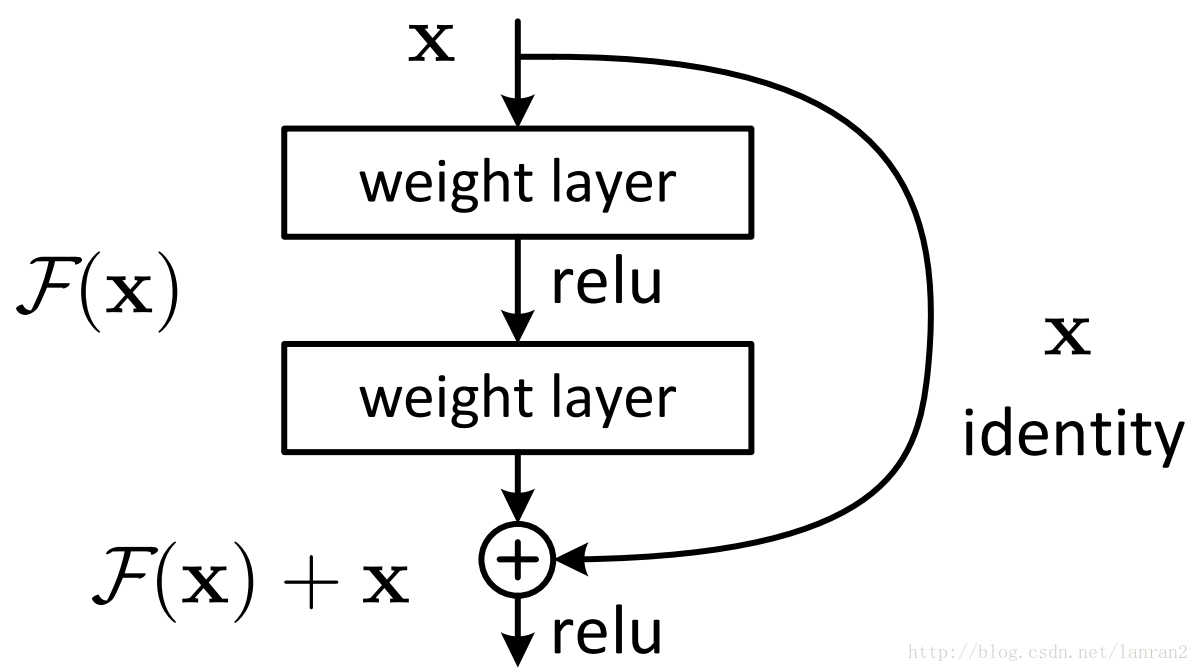


值得一握的是， GoogLeNet 又被称为 Inception Vl 模型， Google 又对该模型做了后续研究，相继提出了 Inception V2, Inception V3, Inception V 4模型，每一代模型的性能都高提升。

## resnet

2015 年，一种名为深度残差网络（ Deep Residual Network, ResNet ） 的模型赢得了 ILSVRC 图像识别竞赛的冠军。深度残差网络比以往的任何模型都要深，白可以训练 100 层，甚至 1000 层。深度残差网络把错误率从 6%( GoogLeNet ）、 7%( VGGNet ）降到了 3.57% ，这也是在 ImageNet 数据集上 ，机器的表现首次优于人类。

深度残差模型的优势在于使用了跳过连攘，让神经网络从拟合 F(x）变成拟合残差 F(x)-x 。 残差比原始函数更窑易学习，也更适合深层模型迭代。因此， 即使训练非常深的神经网络也不会发生非常严重的过拟合。残差网络使用的基本结构如图 2-25 所示，实际的残差网络就是由大量这样的基本单元组成的。



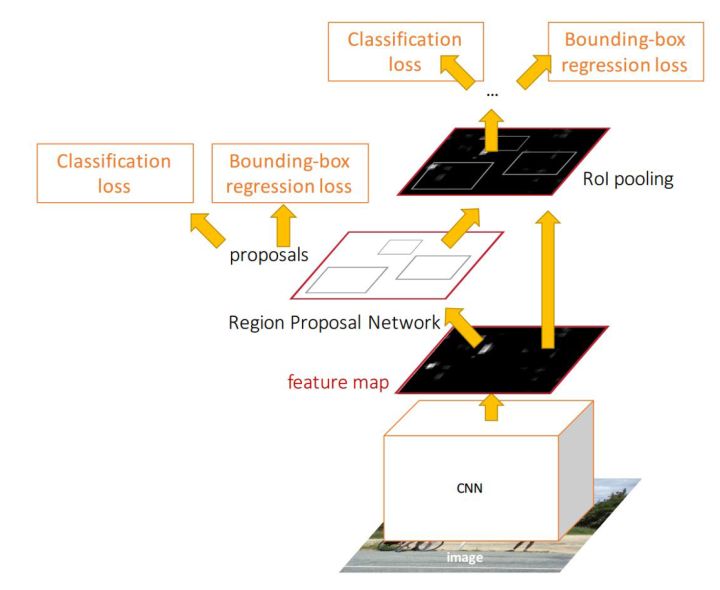
# 目标检测

<https://blog.csdn.net/hunterlew/article/details/71075925>

<https://zhuanlan.zhihu.com/p/35922980>

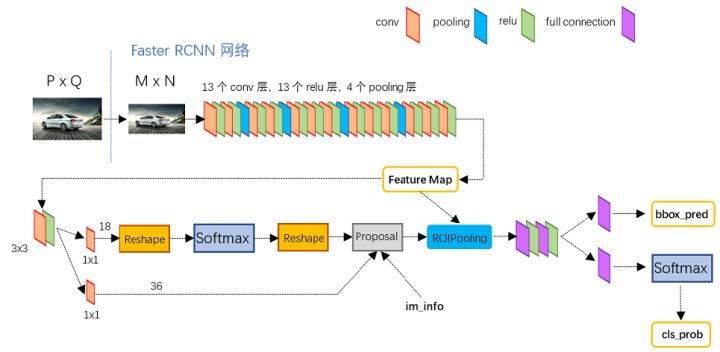
https://zhuanlan.zhihu.com/p/31426458

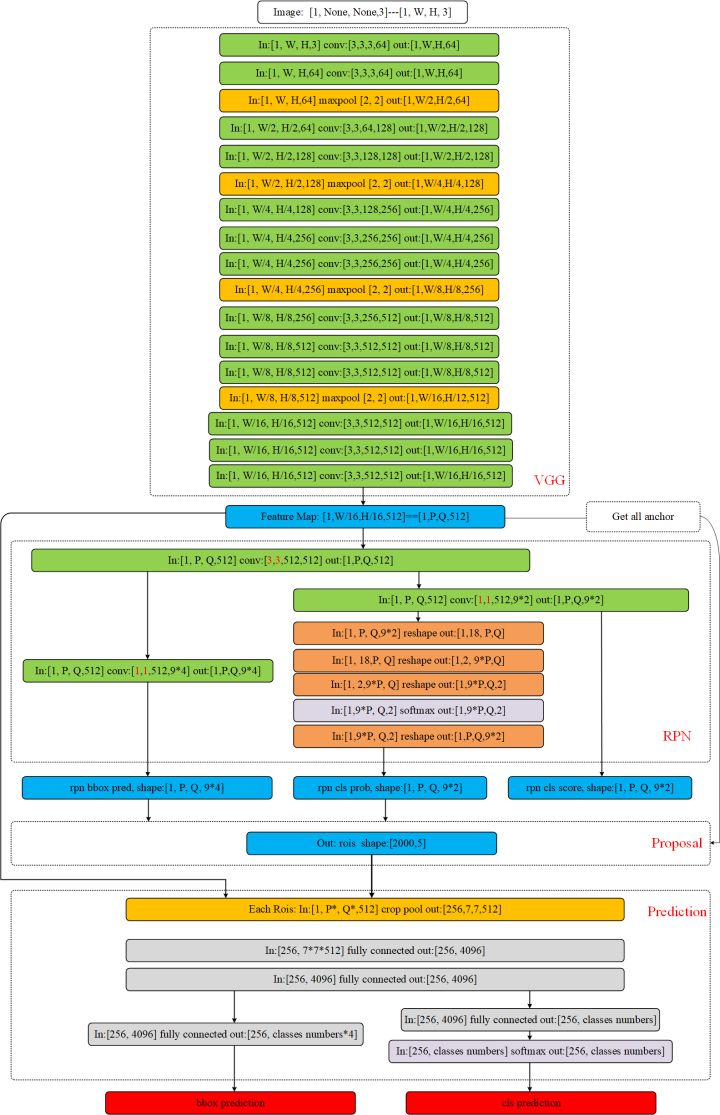
## Faster r-cnn原理



Faster RCNN其实可以分为4个主要内容：

1. Conv layers。作为一种CNN网络目标检测方法，Faster RCNN首先使用一组基础的conv+relu+pooling层提取image的feature maps。该feature maps被共享用于后续RPN层和全连接层。
2. Region Proposal Networks。RPN网络用于生成region proposals。该层通过softmax判断anchors属于foreground或者background，再利用bounding box regression修正anchors获得精确的proposals。
3. Roi Pooling。该层收集输入的feature maps和proposals，综合这些信息后提取proposal feature maps，送入后续全连接层判定目标类别。
4. Classification。利用proposal feature maps计算proposal的类别，同时再次bounding box regression获得检测框最终的精确位置。





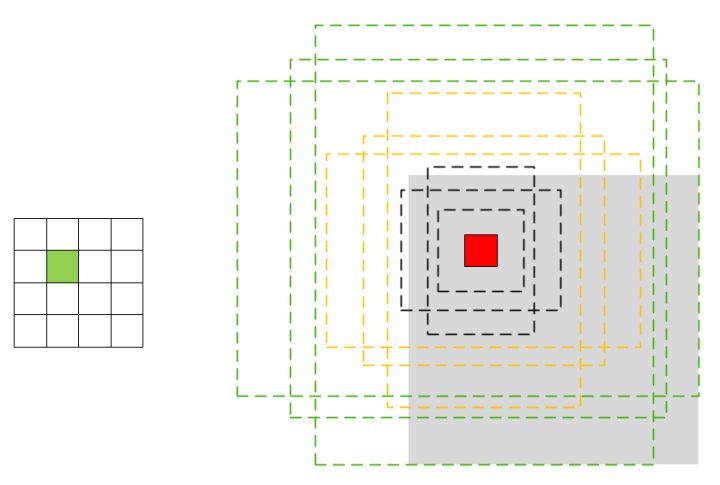
### Feature map提取网络

这层的cnn功能就是特征提取。对输入[1, W, H, 3]的图片(这个batch size我设为1了)。进行多层卷积操作。输出是一个[1, W/16, H/16, 512]维的特征形式。我们可以称为feature map。后面很多操作都是基于feature map来的。我们令P = W/16, Q=H/16。这里的倍数16是在后面找原图和feature map的anchor对应关系时要用到的比例。

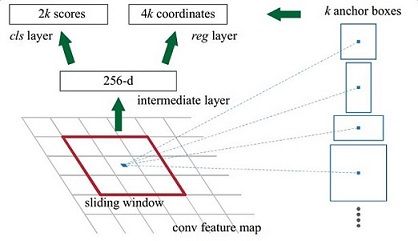
### RPN

步骤1：得到Anchor，如下图，左边的feature map以P=4，Q=4为例，绿色区域的feature map上的一个点，通过计算，以及映射回原图，可以得到原图上9个anchor。Anchor的数量为P\*Q\*9，比如原图800x600，VGG下采样16倍，feature map每个点设置9个Anchor，所以：



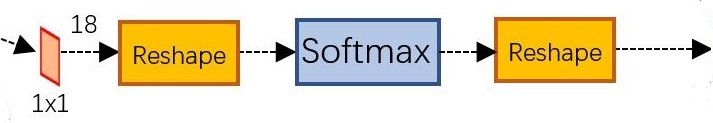
****

步骤2：计算每个anchor的是否是目标的概率值以及每个anchor的偏差值。

****

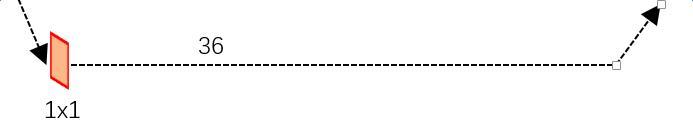
由上图可以看出模型输入就是vgg得到的feature map，大小是[P, Q, 512]（省略了batch size）。论文里说以3\*3为感知野进行卷积操作，即相当于对整个feature map做一个卷积核大小为[3, 3]，卷积核个数为512\*512的卷积操作。输出大小任然是[P, Q, 512]。然后对这个特征在并行两路分别处理。

softmax判定foreground与background：



一路进行[1, 1, 512, 18]的卷积操作。输出大小是[P, Q, 9\*2]。这里很好理解，每个点有9个anchor，相当于对每个anchor对应生成一个1\*2的向量，用来表示这个anchor是不是目标。我们可以令[1, 0]代表这个anchor是目标，[0,1]表示是框的是背景。当然，因为要归一化到0,1所以要经过一个简单的reshape把每个1\*2的向量单独分出来再做一个softmax分类。这样分类之后再reshape回去就可以了。这一路输出得到输出大小是[1, P, Q, 9\*2]。

Regression校正anchor:

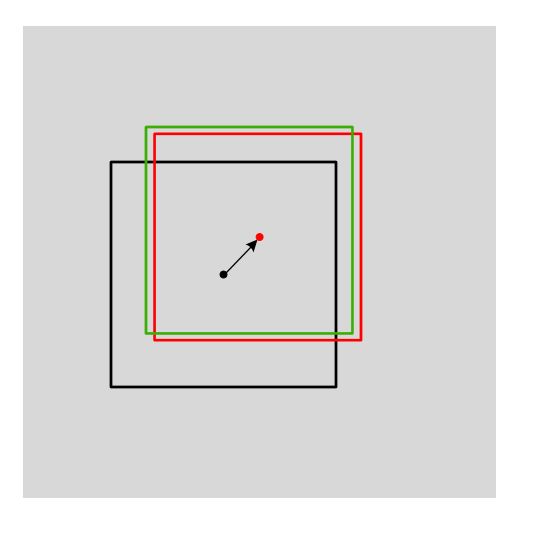


bounding box:另一路同样经过[1, 1, 512, 9\*4]的卷积操作得到[P, Q, 9\*4]的输出，对应的是每个点的每个anchor的四个偏移量 。为什么要四个偏移量呢，因为我们前面的anchor就是一个定值，而实际的物体box和前面的anchor肯定有偏差。所以我们再用四个偏差来进行一次box的修正。修正方法如下：

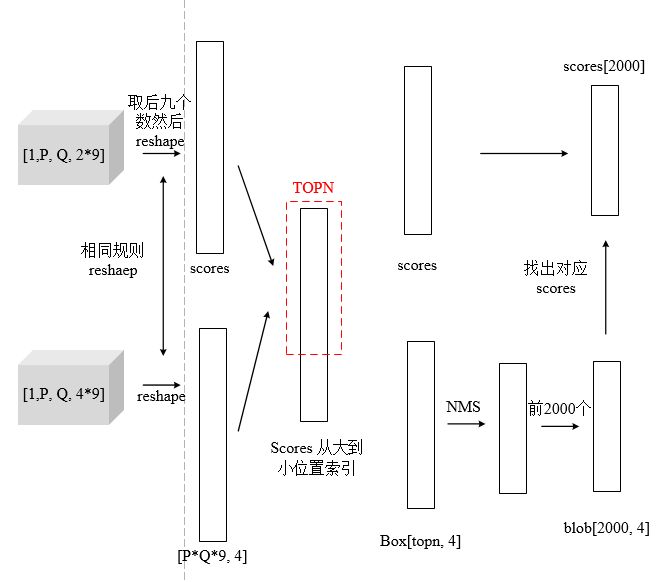
我们默认如下变量定义：rpn输出的anchor偏移值 。标准的anchor生成函数得到的anchor值对应到不同坐标点上后得到的box的值 。这都是上面介绍过的了。那么我们的修正后的box如下公式:



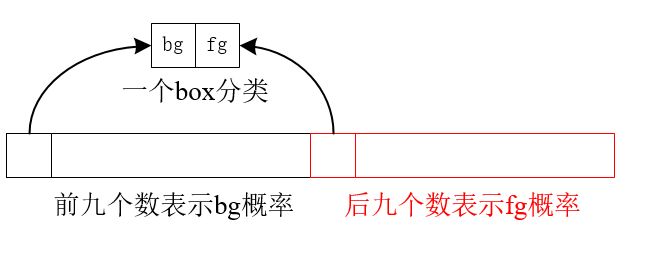
修正后的box: 分别为中心点以及宽和高，我们称为第一次修正后的box。这里相当于对标准的anchor先做中心点平移再做宽高的尺度缩放。下图黑色输出的标准的anchor，我们想将其修正为红色的box（即第一次修正后的box），绿色的是真实的box（即ground true）。



步骤3：通过非极大值抑制取部分blob。



首先用scores表示rpn cls prob这一项。[1, P, Q, 2\*9]，我们先取出第四个维度的后九个数。用后九个数的大小表示fg（真实物体）的概率。



那么reshape成向量：scores=[1, P\*Q\*9]。我们再按大小排序后取出对应的索引值（这里用了argsort()函数，并没有改变scores每个数的位置，只返回其从小到大的数的位置的索引）。我们再取scores比较大的前topN个（12000），然后因为有scores的索引值就可以将对应proposal\_clip\_box的值取出来。然后再进行NMS（非极大值抑制）。

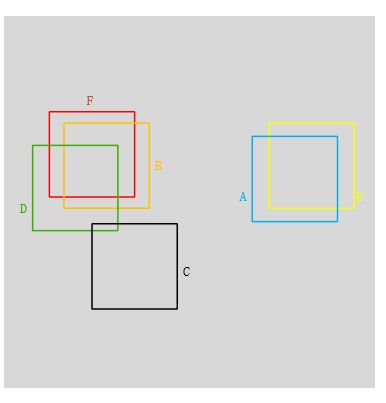
所谓非极大值抑制：先假设有6个输出的矩形框(即proposal\_clip\_box)，根据分类器类别分类概率做排序，从小到大分别属于车辆的概率(scores)分别为A、B、C、D、E、F。

(1)从最大概率矩形框F开始，分别判断A~E与F的重叠度IOU是否大于某个设定的阈值;

(2)假设B、D与F的重叠度超过阈值，那么就扔掉B、D；并标记第一个矩形框F，是我们保留下来的。

(3)从剩下的矩形框A、C、E中，选择概率最大的E，然后判断E与A、C的重叠度，重叠度大于一定的阈值，那么就扔掉；并标记E是我们保留下来的第二个矩形框。

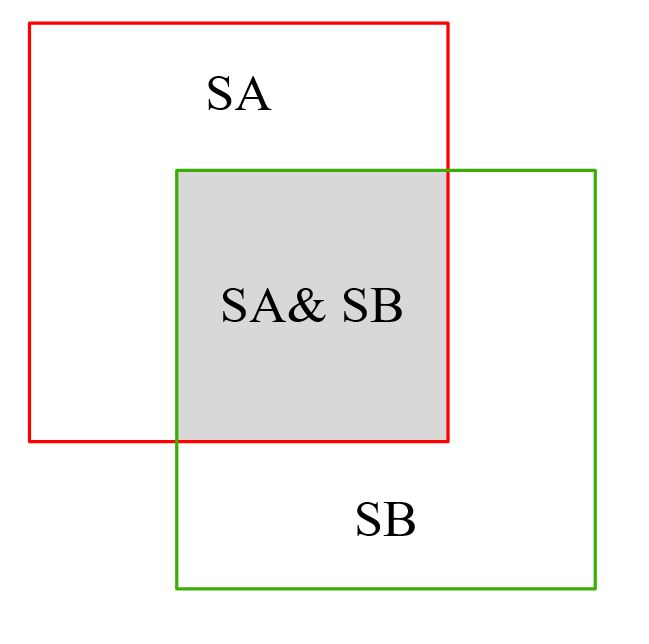
就这样一直重复，找到所有被保留下来的矩形框。



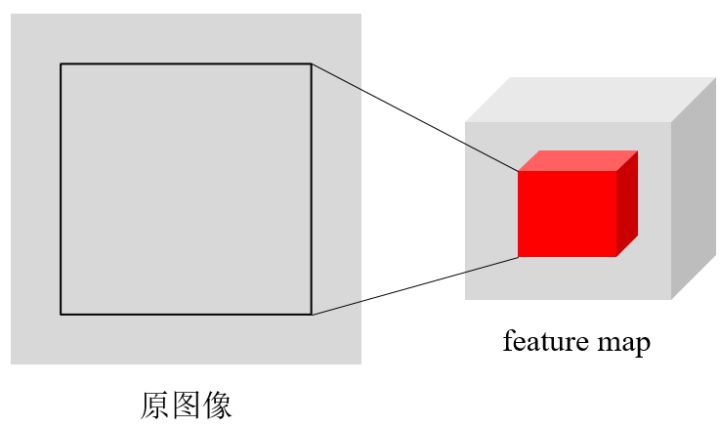
如上图F与BD重合度较大，可以去除BD。AE重合度较大，我们删除A,保留scores较大的E。C和其他重叠都小保留C。最终留下了C、E、F三个。

其中IOU比较好理解，假设proposal\_clip\_box的面积为SA,真实box面积为SB，两者的重叠面积为 SA and SB。那么IOU:



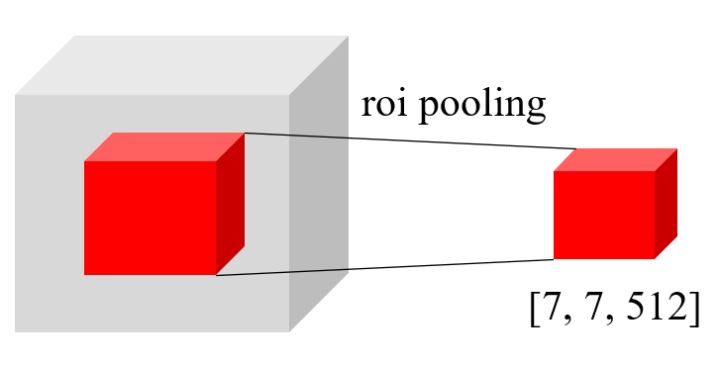


步骤4:把blob映射到feature map上得到rois，我们得到的就是保留下来的2000个box（称为blob）。然后下面就是真正的proposal过程，将这些blob反向投影到前面的feature map上得到对应的2000个rois。Blob表示的是原图上的box大小，rois表示的是投影到[1, P, Q, 512]的feature map上时对应的box大小。经过这个反向投影就得到了2000个roils了(图里维度是[2000,5]而不是[2000,4]是因为加了一维全0的项用于表示一个图片，要是多张图片就要改了)。

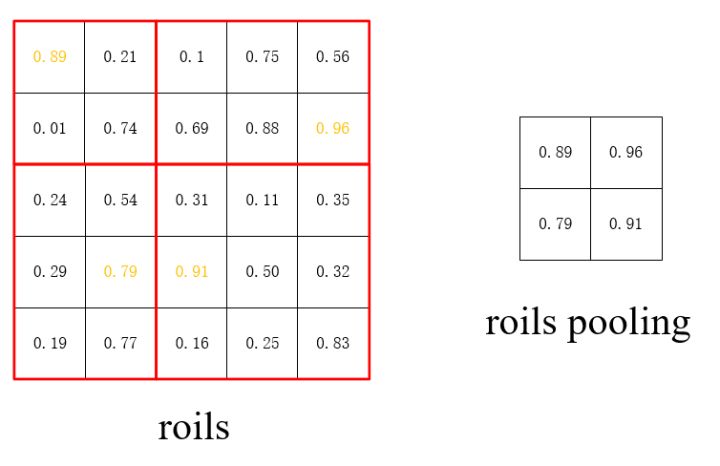


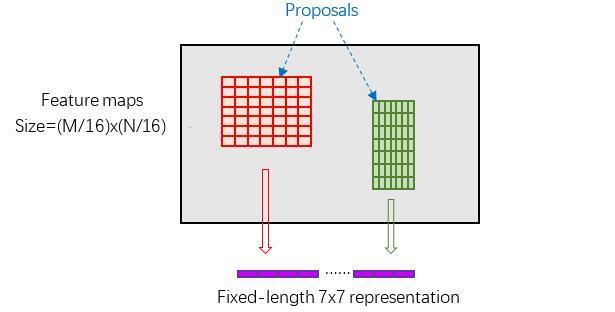
### ROI poling

前面得到了所有的roils后就可以进行最后的预测模型了。假设我们的一个roils取出来大小是[10, 10, 512]即从原图像投影到feature map上大小变成[10, 10]。我们后面分类的时候要复用同一个卷积网络，所以对不同大小的roils要池化成同一个大小的输入：这里统一池化成[7, 7, 512]。



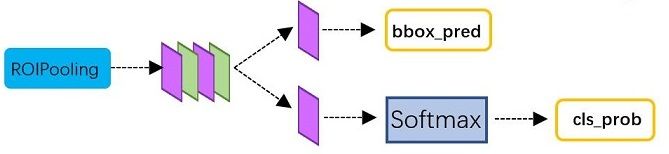
Roi pooling也比较清楚，将输入特征先进行均匀划分，然后对分的每一块进行max pooling即可。



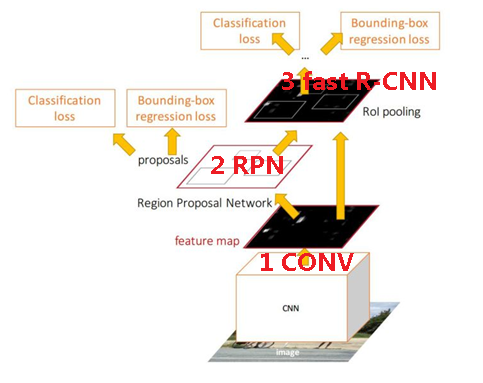


### 目标分类和框回归

经过前面roil pooling后我们对每个roils得到了固定尺寸的特征矩阵。然后可以进行分类卷积操作。分类时一样进行两路。两路输入都是[256, 7, 7, 512]，256代表有多少个roils。将这个值进行两次全连接输出[256, 4096]。然后再分开两个全连接，一个输出大小[256, classes numbers], classes numbers是我们的目标物体分类数，假设是10分类的任务，那么就是10。输出经过softmax后是One hot形式。另一路输出[256, classes numbers\*4],表示对256个box的第二次修正偏差值 。因为上面虽然修正一次了但是可能还不太正确，这里再修正一次。修正方法和上面一样。修正后就可以得到我们最后真正的输出了。同样还可以得到每个box的分类概率。通过分类概率大小我们设置一个阈值，当分类概率达到一定阈值之后才输出。这样就可以将所有可能的目标的box和分类类别画出来了。



## Faster r-cnn模型训练

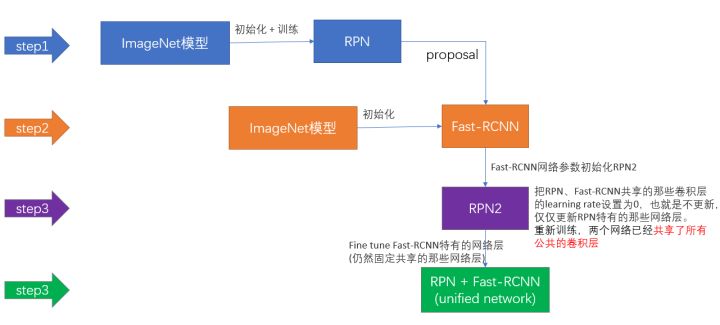


第一步：用ImageNet模型初始化，独立训练一个RPN网络。即上图中的1和2参与训练，并且1和2的网络参数都可变。

第二步：仍然用ImageNet模型初始化，但是使用上一步RPN网络产生的proposal作为输入，训练一个Fast-RCNN网络，至此，两个网络每一层的参数完全不共享。即1，2，3都参与训练，但2的参数不更新，1和3的参数更新。

第三步：再次训练RPN，1，2参与训练，其中1的参数使用第二步的Fast-RCNN网络参数初始化，且在训练过程中不变，2的参数更新，此时两个网络已经共享了所有公共的卷积层。

第四步：再次训练fast R-CNN，1，2，3都参与训练，1，2的参数不变，3的参数变。



## 实践

**下载 TensorFlow models**

地址：<https://github.com/tensorflow/models> git clone 到本地目录。

**Protobuf 编译**

从<https://github.com/google/protobuf/releases>下载win版的工具，即：protoc-3.4.0-win32.zip，解压生成：bin, include两个文件夹。将bin文件夹中的【protoc.exe】放到C:\Windows\System32文件夹下，

编译Protobuf库，在object\_detection同级目录打开终端运行：

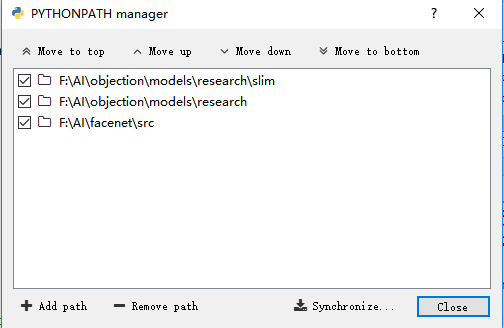
protoc object\_detection/protos/\*.proto --python\_out=.

预料之中的报错，protoc不能被识别的命令。

因为Windows需要加上 protobuf 的路径：

C:\Windows\System32\protoc object\_detection\protos\\*.proto --python\_out=.

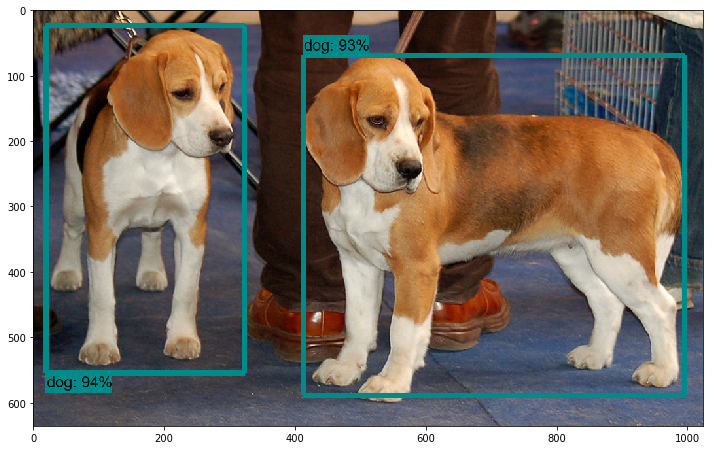
**添加PYTHONPATH变量：**

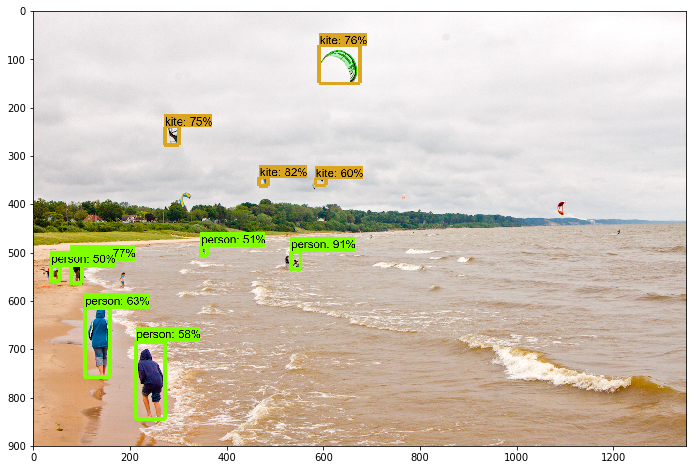


返回research目录测试环境是否准备完毕：

python object\_detection/builders/model\_builder\_test.py

运行models\research\object\_detection、object\_detection\_tutorial.ipynb





# 人脸识别

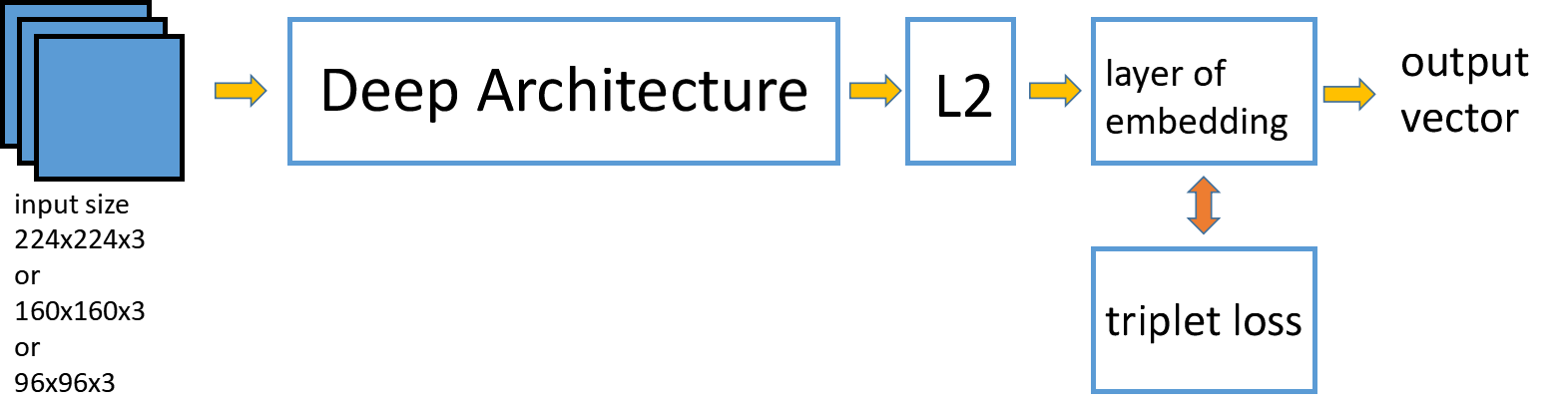
参考<https://blog.csdn.net/xingwei_09/article/details/79161931>

参考https://blog.csdn.net/MrCharles/article/details/80360461

## 原理

facenet是Google开源的人脸识别框架，它的作用是把输入的人脸图像映射为多维特征向量，相当于对不同的人脸进行了不同的编码，同一个人脸的图像生成的编码几乎一致，不同的人脸图像生成的编码差异非常大，并以此达到识别的目的。设计一个能够达到这样效果的映射的网络是一个很难的问题，我们下面就一步一步来看facenet是怎样解决这个问题的。

首先，facenet的结构是这样的：

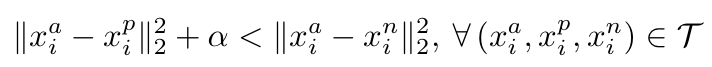


facenet网络的输入有多种不同的大小，中间部分是一个深度卷积神经网络，与其他普通CNN没有多大区别。facenet不一样的地方在于后面部分，它对深度卷积神经网络的输出做了一个L2正则化，然后再对输出进行了embedding，直接将embedding的映射结果作为特征向量输出。facenet并没有像其他一般的CNN用softmax作为损失函数，而是设计了一种新的损失，triplet loss。

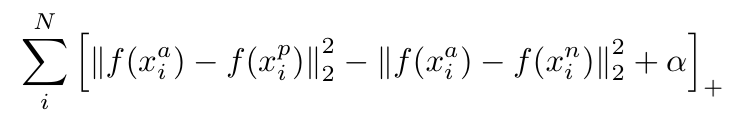
在理想的情况下，特征向量之间的距离可以直接反映人脸的相似度，即：对于同一个人的两张人脸图像，对应的向量之间的欧几里得距离比较小对于不同人的两张图像，对应的向量之间的欧几里得距离比较大，假设人脸图像为x1和x2，对应的特征为f(x1)和f(x2)。当x1和x2对应的是同一个人脸时，其距离II f(x1)-f(x2) II应该很小，而当x1和x2对应的是不同的人脸时，其距离II f(x1)-f(x2) II应该很大。

然而事实并非如此。在一般CNN网络中，最后的输出经过softmax分类器，使用的是softmax损失。这个损失是不同类别间的损失。对于人脸来说，每一个人脸就是一个人。看起来似乎很合理，但是用softmax表示损失，以此区别出不同的人是不可行的。softmax本质上没有对每一类的向量表示之间的距离做出要求。用softmax分类的结果，可能同一个类中的向量，它的类间距比不同类中的向量间距还要大。对于这种情况，就要考虑设计新的损失函数解决问题。

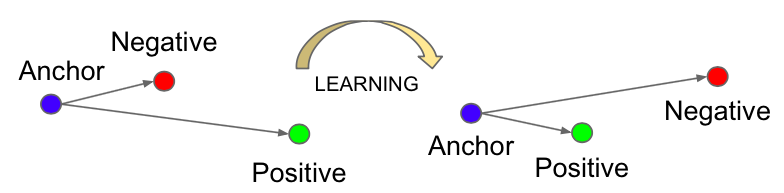
下面重点看triplet loss的定义及原理，三元组损失(triplet loss)的原理：所谓的三元组就是三个样例，如(anchor, pos, neg)，其中，x和p是同一类，x和n是不同类。那么学习的过程就是学到一种表示，对于尽可能多的三元组，使得anchor和pos的距离，小于anchor和neg的距离。即：



所以，变换一下，得到目标函数：



目标函数的含义就是对于不满足条件的三元组，进行优化；对于满足条件的三元组，就pass先不管。



另一种损失：中心损失(center loss)

中心损失的定义：中心损失不对距离进行优化，它保留了原有的分类模型，但又为每一个类指定了一个类别中心。同一个类的图像对应的特征应该尽量靠近自己的类别中心，不同的类别中心尽量远离。

输入的人脸图像为，该人脸的类别为，对每一个类别都规定一个类别中心。希望每个人脸对应的特征都尽可能的接近其中心，因此中心损失定义为：



多张图像的中心损失就是将单张图像的损失相加：



此外，不能只使用中心损失来训练分类模型，还需要加入softmax损失，也就是说，最终的损失由两部分构成，即，其中的是一个超参数。

## 操作

1. 下载facenet：git clone <https://github.com/davidsandberg/facenet.git>
2. 按照facenet/requirements.txt文件里要求的安装库

tensorflow==1.7

scipy

scikit-learn

opencv-python

h5py

matplotlib

Pillow

requests

psutil

1. 用conda list \*\*\*lib命令在anaconda prompt中查询，发现其中opencv-python

没有安装，下载地址：http://www.lfd.uci.edu/~gohlke/pythonlibs/，因为python装的是3.6.2版本，所以对应着选择，就安装[opencv\_python‑3.4.2‑cp36‑cp36m‑win\_amd64.whl](javascript:;)这个了。下载好之后把文件复制到Anaconda3\Lib\site-packages文件夹下。在anaconda prompt里进到此目录，然后执行pip install [opencv\_python‑3.4.2‑cp36‑cp36m‑win\_amd64.whl](javascript:;)。



1. 在地址http://vis-www.cs.umass.edu/lfw/lfw.tgz 下载lfw数据集，并解压到facenet/ datasets/lfw/raw中。
2. LFW 数据库上的人脸检测和对齐，对LFW进行人脸检测和对齐：

run src/align/align\_dataset\_mtcnn.py \

data/lfw/raw \

data/lfw/lfw\_mtcnnpy\_160 \

--image\_size 160 --margin 32 \

--random\_order

在CPU里面经过半小时的运行后，在输出目录data/lfw/lfw\_mtcnnpy\_160中可以找到检测、对齐后裁剪好的人脸。

1. 导入已有的模型，模型下载链接：https://pan.baidu.com/s/1aiSq7wGpdHIe6MUKPnXgrA 密码：4dcn，然后运行：

run src/validate\_on\_lfw.py \

data/lfw/lfw\_mtcnnpy\_160 \

src/models/facenet

可得到如下结果：注意一定要用20180408-102900的最新模型。

Model directory: src/models/facenet

Metagraph file: model-20180408-102900.meta

Checkpoint file: model-20180408-102900.ckpt-90

Runnning forward pass on LFW images

............

Accuracy: 0.97750+-0.00396

Validation rate: 0.82100+-0.03709 @ FAR=0.00100

Area Under Curve (AUC): 0.997

Equal Error Rate (EER): 0.022

1. Facenet可以直接对比2个人脸经过它的网络映射之后的欧式距离；

run src/compare.py src/models/facenet data/images/Anthony\_Hopkins\_0001.jpg data/images/Anthony\_Hopkins\_0002.jpg

结果为：

Distance matrix

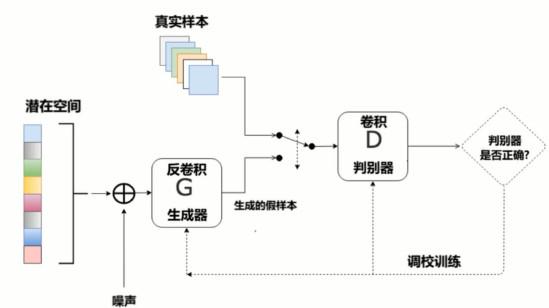
0 1

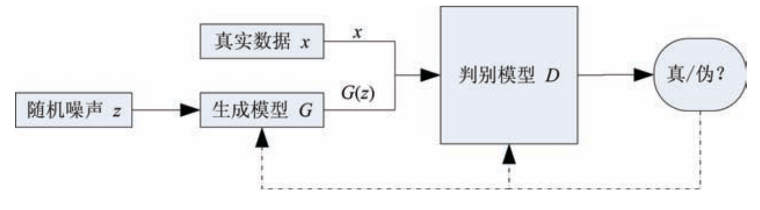
0 0.0000 0.8516

1 0.8516 0.0000

# 图像生成

## 生成对抗网络（GAN）原理





训练步骤：

1、固定生成器参数，训练判别器，训练目的：

 对于给定的真实数据x，判别器输出结果D(x)即为真的概率希望是1；用交叉熵来计算损失，[D(x),1-D(x)]对应的标签为[1,0]，交叉熵为-logD(x)。

 对于给定的真实数据G(z)，判别器输出结果D(G(z))即为真的概率希望是0；用交叉熵来计算损失，[D(G(z)),1-D(G(z))]对应的标签为[0,1]，交叉熵为-log(1-D(x))。

因此判别器总共损失:loss\_D=-logD(x) -log(1-D(x))，用梯度下降法训练。

2、固定判别器参数，训练生成器，训练目的：

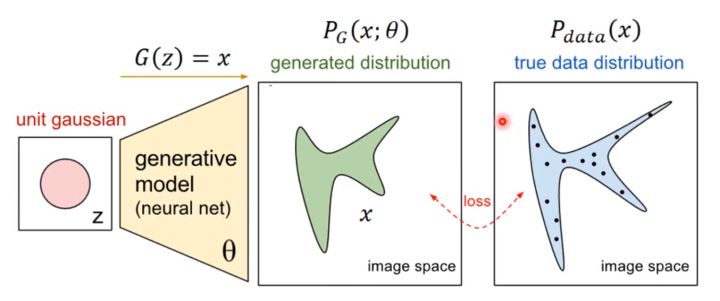
 对于生成器传给辨别器的生成图片，生成器希望辨别器打上标签1。用交叉熵来计算损失，[D(G(z)),1-D(G(z))]对应的标签为[1,0]，交叉熵为-log(D(x))。

因此生成器损失:loss\_G=-logD(x)，用梯度下降法训练。

图像的分布：假设彩色图像是64x64大小，则相当于图像的分布满足一个多变量分布函数，变量的总数是64x64x3。即每一个像素都是一个单变量分布，而整幅图像的所有像素构成了一个多变量分布。建模为多变量分布的好处之一是：帮助更好的formulate图像的生成过程。

图像的生成，可以看作是从一个多变量分布函数中进行随机采样。当我们使用照相机拍了一张照片，也就相当于从自然界中采样了一张图像。

神经网络只要有非线性激活函数，就可以去拟合任意的函数，那么分布也是一样，所以可以用一直正态分布，或者高斯分布，取样去训练一个神经网络，学习到一个很复杂的分布。

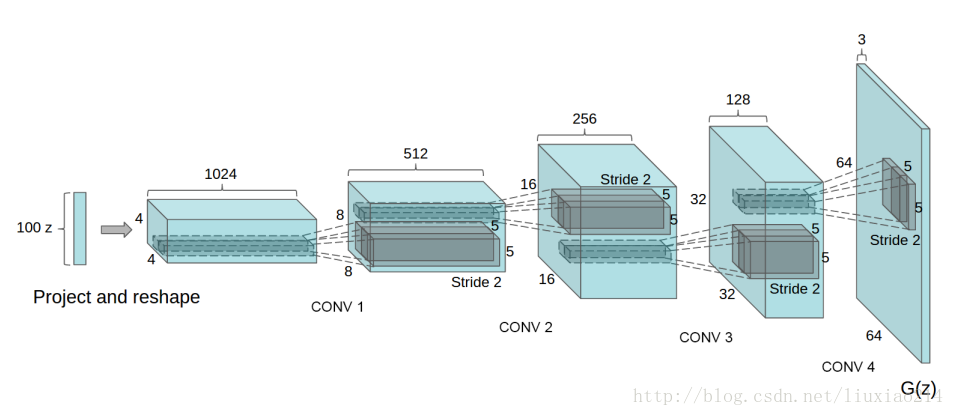


本质是训练生成器网络，把特定分布映射到和真实数据相同的分布上。

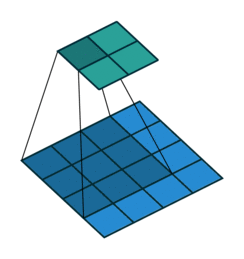
## 实践

DCGAN对卷积神经网络的结构做了一些改变，以提高样本的质量和收敛的速度，这些改变有：

1. 取消所有pooling层。G网络中使用转置卷积（transposed convolutional layer）进行上采样，D网络中用加入stride的卷积代替pooling。
2. 在D和G中均使用batch normalization
3. 去掉FC层，使网络变为全卷积网络
4. G网络中使用ReLU作为激活函数，最后一层使用tanh
5. D网络中使用LeakyReLU作为激活函数

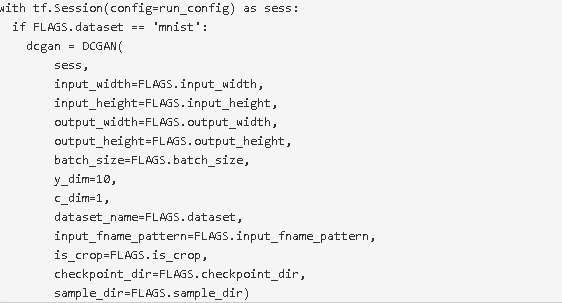


反卷积：

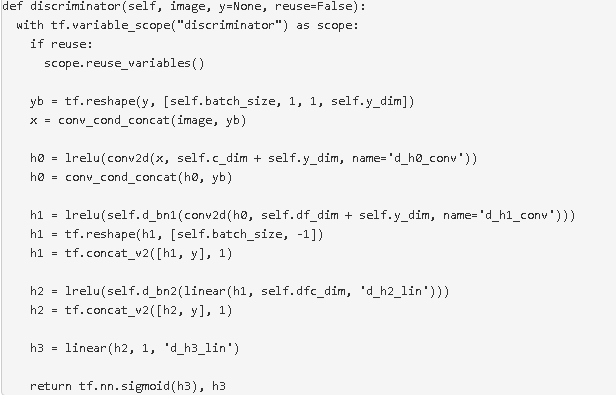


生成mnist项目开源地址：<https://github.com/carpedm20/DCGAN-tensorflow>。

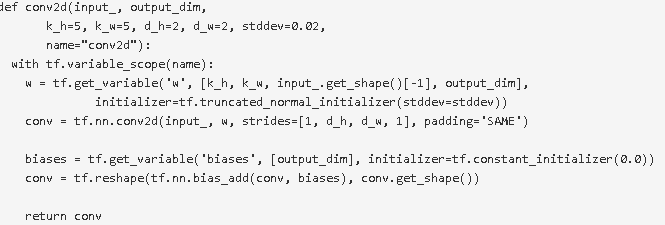
先看main.py：因为我们使用DCGAN来生成MNIST数字手写体图像，注意这里的y\_dim=10，表示0到9这10个类别，c\_dim=1，表示灰度图像。



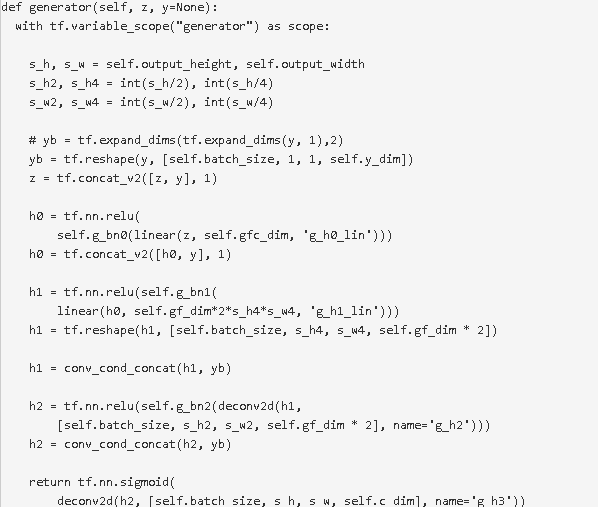
再看model.py：



这里batch\_size=64，image的维度为[64 28 28 1]，y的维度是[64 10]，yb的维度[64 1 1 10]，x将image和yb连接起来，这相当于是使用了Conditional GAN，为图像提供标签作为条件信息，于是x的维度是[64 28 28 11]，将x输入到卷积层conv2d，conv2d的代码如下：

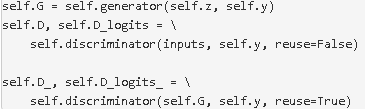


卷积核的大小为5\*5，stride为[1 2 2 1]，通过2的卷积步长可以替代pooling进行降维，padding=‘SAME’，则卷积的输出维度为[64 14 14 11]。然后使用batch normalization及leaky ReLU的激活层，输出与yb再进行concat，得到h0，维度为[64 14 14 21]。同理，h1的维度为[64  7\*7\*74+10]，h2的维度为[64 1024+10]，然后连接一个线性输出，得到h3，维度为[64 1]，由于我们希望判别器的输出代表概率，所以最终使用一个sigmoid的激活。



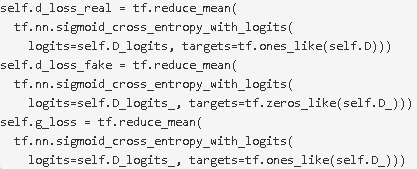
output\_height和output\_width为28，因此s\_h和s\_w为28，s\_h2和s\_w2为14，s\_h4和s\_w4为7。在这里z为平均分布的随机分布数，维度为[64 100]，y的维度为[64 10]，yb的维度是[64 1 1 10]，z与y进行一个concat得到[64 110]的tensor，输入到一个线性层，输出维度是[64 1024]，再经过batch normalization以及ReLU激活，并与y进行concat，输出h0的维度是[64 1034]，同样的再经过一个线性层输出维度为[64 128\*7\*7]，再进行reshape并与yb进行concat，得到h1，维度为[64 7 7 138]，然后输入到一个deconv2d，做一个反卷积，也就是文中说的fractional strided convolutions，再经过batch normalization以及ReLU激活，并与yb进行concat，输出h2的维度是[64 14 14 138]，最后再输入到deconv2d层以及sigmoid激活，得到生成器的输出，维度为[64 28 28 1]。

生成器以及判别器的输出：



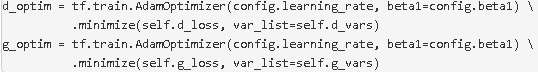
其中D表示真实数据的判别器输出，D\_表示生成数据的判别器输出。

再看损失函数：

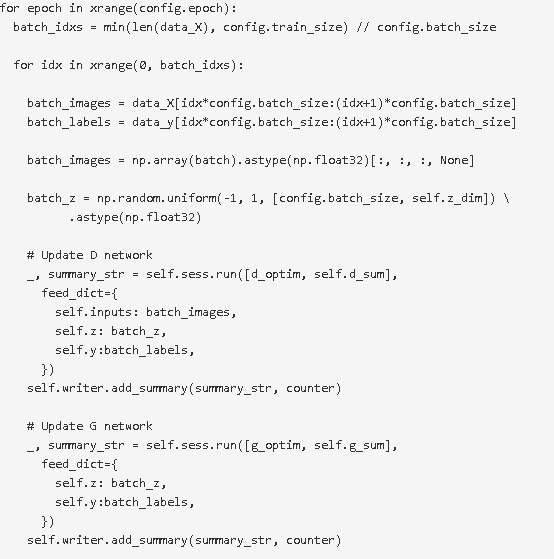


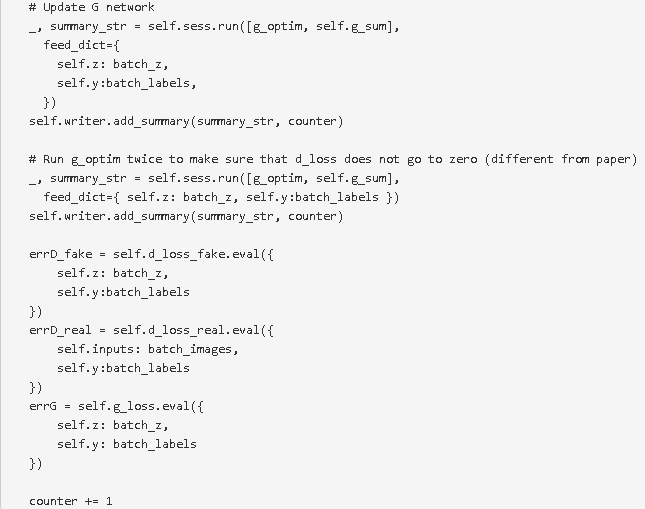
即对于真实数据，判别器的损失函数d\_loss\_real为判别器输出与1的交叉熵，而对于生成数据，判别器的损失函数d\_loss\_fake为输出与0的交叉熵，因此判别器的损失函数d\_loss=d\_loss\_real+d\_loss\_fake；生成器的损失函数是g\_loss判别器对于生成数据的输出与1的交叉熵。

优化器：



训练阶段：





实验结果：

由于自己的笔记本配置有限，仅使用CPU来运行速度较慢，因此epoch仅设置为2，对于MNIST手写数字数据集的生成情况如下



# 图像理解

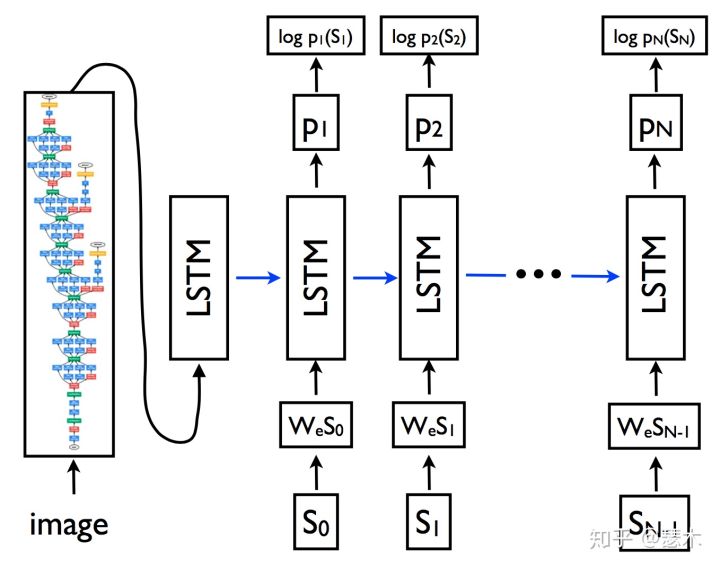
图像描述任务是输入一张图片，生成对该图像的描述文本，可以表示为单词编码序列

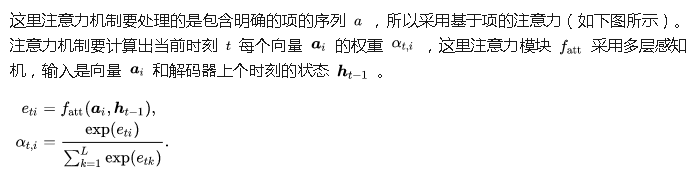
C:\Users\Administrator\AppData\Roaming\feiq\RichOle\2421776268.bmp

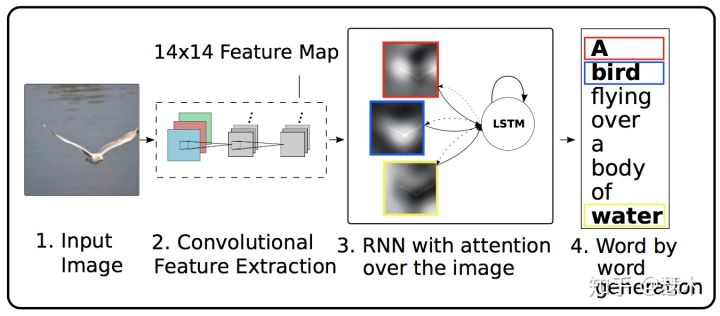
其中K是词表的大小，C是描述文本的长度。对于原图像，使用CNN来抽取它的特征，最后一个卷积层的输出可以产生 L 个向量，每一个向量是原图中一个区域的 D 维表示，可以看成一个序列。

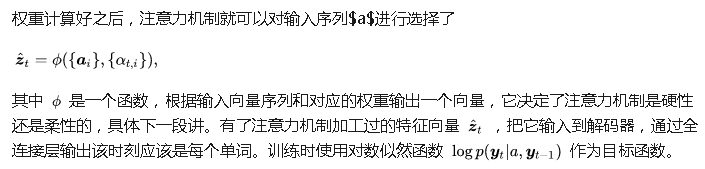
C:\Users\Administrator\AppData\Roaming\feiq\RichOle\2307268147.bmp

一种不使用注意力机制的简单做法就是将上述向量 a 取平均池化或最大池化，然后输入到RNN解码器中做文本序列的生成，如下图所示。这种方法把图片每个部分的特征等权重对待，不能更细化地针对不同部位来生成每个单词，而注意力机制正合适解决这种问题。









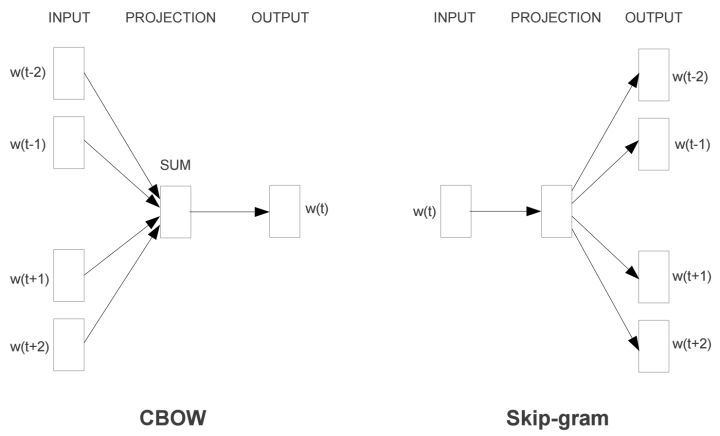
对于soft attention

C:\Users\Administrator\AppData\Roaming\feiq\RichOle\3769686712.bmp

# 自然语言处理（NLP）

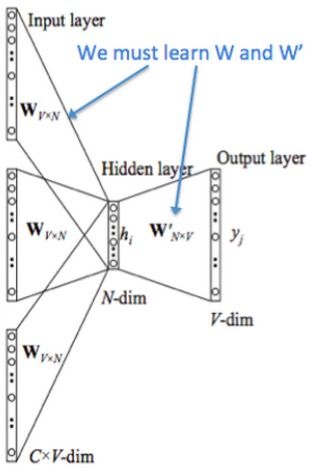
# Word2vec

Word2Vec模型中，主要有Skip-Gram和CBOW两种模型，从直观上理解，Skip-Gram是给定input word来预测上下文。而CBOW是给定上下文，来预测input word。



## CBOW模型

先来看着这个结构图，用自然语言描述一下CBOW模型的流程：

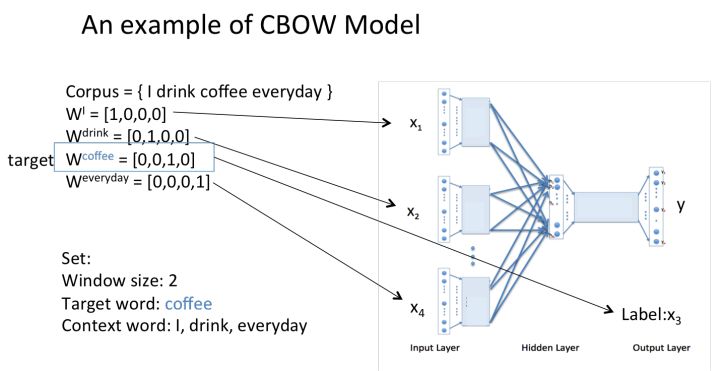


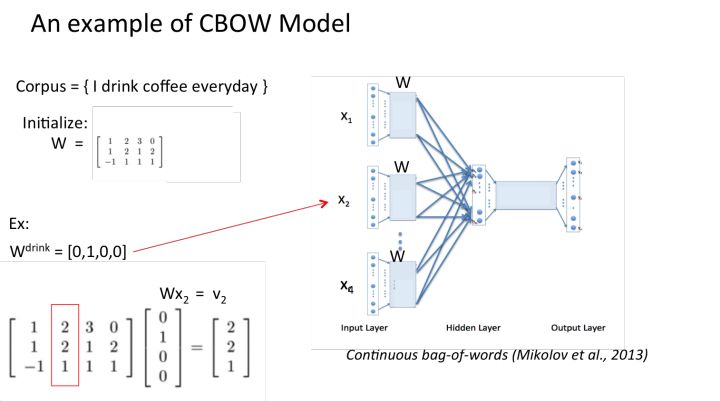
输入层：上下文单词的onehot. {假设单词向量空间dim为V，上下文单词个数为C}，所有onehot分别乘以共享的输入权重矩阵W. {V\*N矩阵，N为自己设定的数，初始化权重矩阵W}所得的向量 {因为是onehot所以为向量} 相加求平均作为隐层向量, size为1\*N.乘以输出权重矩阵W' {N\*V}得到向量 {1\*V} 激活函数处理得到V-dim概率分布 {PS: 因为是onehot嘛，其中的每一维斗代表着一个单词}，概率最大的index所指示的单词为预测出的中间词（target word）与true label的onehot做比较，误差越小越好

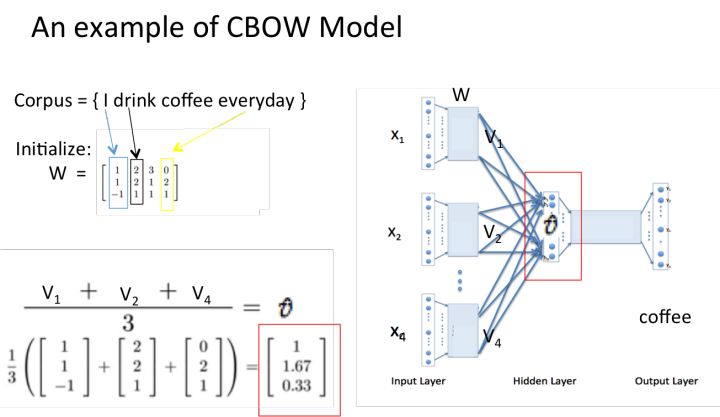
所以，需要定义loss function（一般为交叉熵代价函数），采用梯度下降算法更新W和W'。训练完毕后，输入层的每个单词与矩阵W相乘得到的向量的就是我们想要的词向量（word embedding），这个矩阵（所有单词的word embedding）也叫做look up table（其实聪明的你已经看出来了，其实这个look up table就是矩阵W自身），也就是说，任何一个单词的onehot乘以这个矩阵都将得到自己的词向量。有了look up table就可以免去训练过程直接查表得到单词的词向量了。

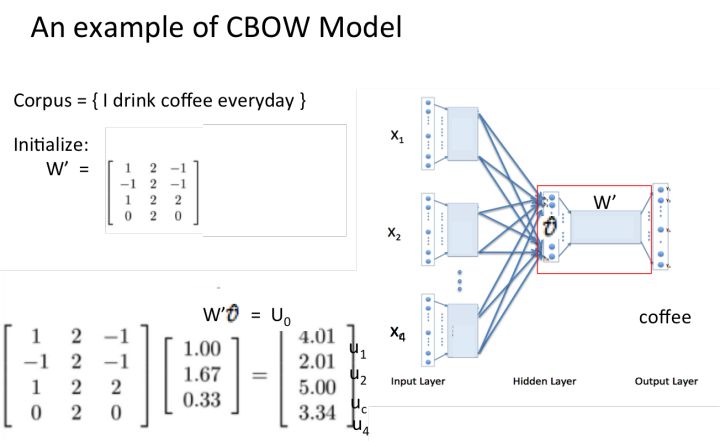
CBOW模型流程举例：

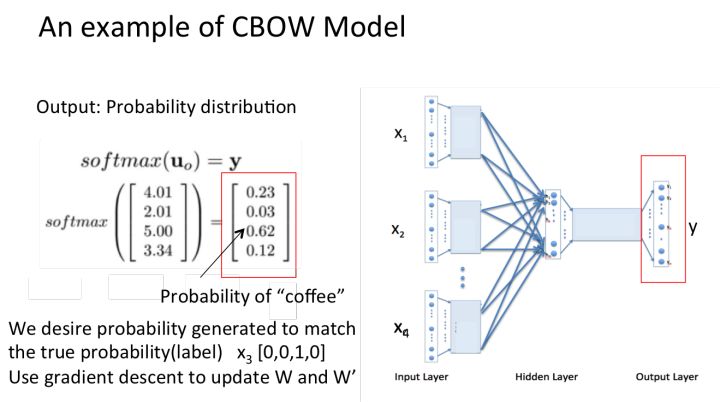
假设我们现在的Corpus是这一个简单的只有四个单词的document：{I drink coffee everyday}，我们选coffee作为中心词，window size设为2，也就是说，我们要根据单词"I","drink"和"everyday"来预测一个单词，并且我们希望这个单词是coffee。







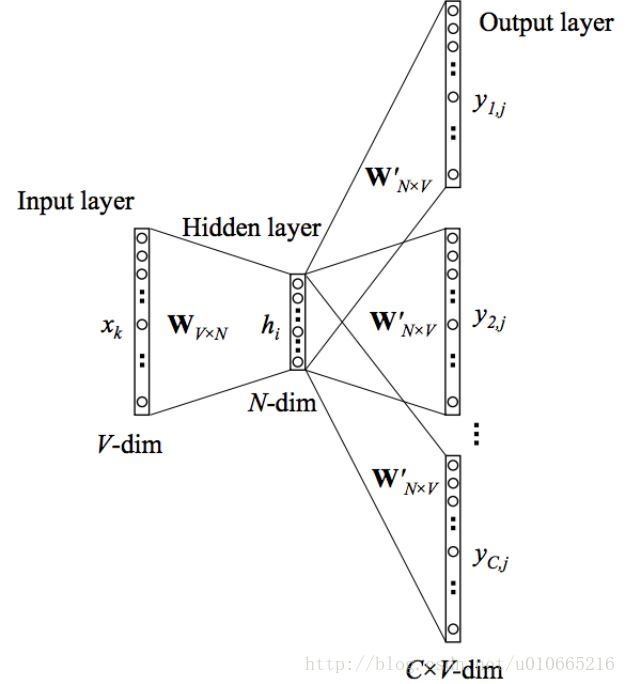




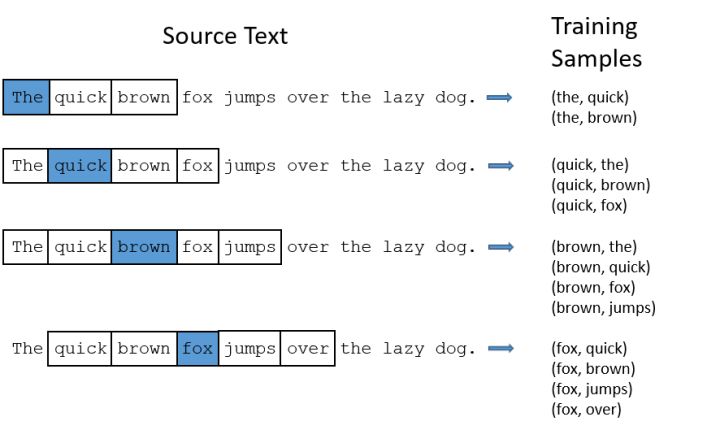
假设我们此时得到的概率分布已经达到了设定的迭代次数，那么现在我们训练出来的look up table应该为矩阵W。即，任何一个单词的one-hot表示乘以这个矩阵都将得到自己的word embedding。

## Skip-gram模型

接下来我们来看下skip-gram神经网络模型，skip-gram的神经网络模型是从前馈神经网络模型改进而来，说白了就是在前馈神经网络模型的基础上，通过一些技巧使得模型更有效。我们先上图，看一波skip-gram的神经网络模型：



我们将通过给神经网络输入文本中成对的单词来训练它完成上面所说的概率计算。下面的图中给出了一些我们的训练样本的例子。我们选定句子“The quick brown fox jumps over lazy dog”，设定我们的窗口大小为window size=2，也就是说我们仅选输入词前后各两个词和输入词进行组合。下图中，蓝色代表input word，方框内代表位于窗口内的单词。



再来看参数 num\_skips和skip\_window 。在生成单词对时，会在语料库中先取出一个长度为skip\_window\*2+1连续单词列表，其中最中间的词是输入，其余skip\_window\*2个单词是它的上下文，会在这其中随机选取num\_skips个单词，放入标签中。

如 skip\_window=2， num skips=2 ，会首先选取一个长度为 buffer, 假设是The quick brown fox jumps，此时中心单词为 brown, 再在剩下的单词中随机选取两个构成单词对，比如选择brown🡪quick ，brown🡪fox。

模型细节

我们如何来表示这些单词呢？

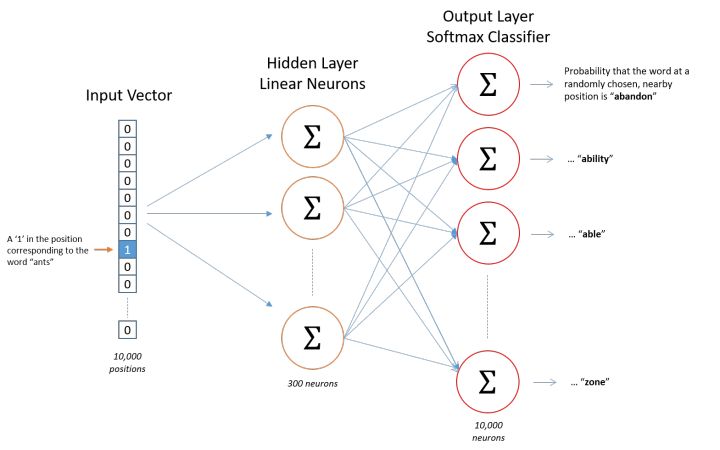
首先，我们都知道神经网络只能接受数值输入，我们不可能把一个单词字符串作为输入，因此我们得想个办法来表示这些单词。最常用的办法就是基于训练文档来构建我们自己的词汇表（vocabulary）再对单词进行one-hot编码。

假设从我们的训练文档中抽取出10000个唯一不重复的单词组成词汇表。我们对这10000个单词进行one-hot编码，得到的每个单词都是一个10000维的向量，向量每个维度的值只有0或者1，假如单词ants在词汇表中的出现位置为第3个，那么ants的向量就是一个第三维度取值为1，其他维都为0的10000维的向量。

还是上面的例子，“The dog barked at the mailman”，那么我们基于这个句子，可以构建一个大小为5的词汇表（忽略大小写和标点符号）：("the", "dog", "barked", "at", "mailman")，我们对这个词汇表的单词进行编号0-4。那么”dog“就可以被表示为一个5维向量[0, 1, 0, 0, 0]。

模型的输入如果为一个10000维的向量，那么输出也是一个10000维度（词汇表的大小）的向量，它包含了10000个概率，每一个概率代表着当前词是输入样本中output word的概率大小。

下图是我们神经网络的结构：



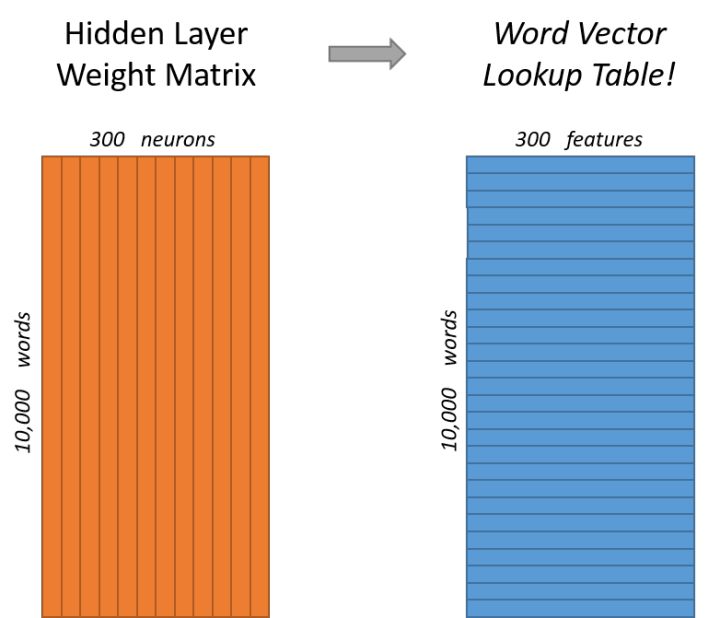
隐层没有使用任何激活函数，但是输出层使用了sotfmax。我们基于成对的单词来对神经网络进行训练，训练样本是 ( input word, output word ) 这样的单词对，input word和output word都是one-hot编码的向量。最终模型的输出是一个概率分布。

**隐层**

说完单词的编码和训练样本的选取，我们来看下我们的隐层。如果我们现在想用300个特征来表示一个单词（即每个词可以被表示为300维的向量）。那么隐层的权重矩阵应该为10000行，300列（隐层有300个结点）。

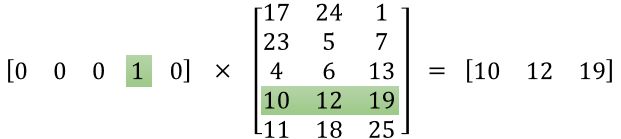
Google在最新发布的基于Google news数据集训练的模型中使用的就是300个特征的词向量。词向量的维度是一个可以调节的超参数（在Python的gensim包中封装的Word2Vec接口默认的词向量大小为100， window\_size为5）。

看下面的图片，左右两张图分别从不同角度代表了输入层-隐层的权重矩阵。左图中每一列代表一个10000维的词向量和隐层单个神经元连接的权重向量。从右边的图来看，每一行实际上代表了每个单词的词向量。



所以我们最终的目标就是学习这个隐层的权重矩阵。我们现在回来接着通过模型的定义来训练我们的这个模型。

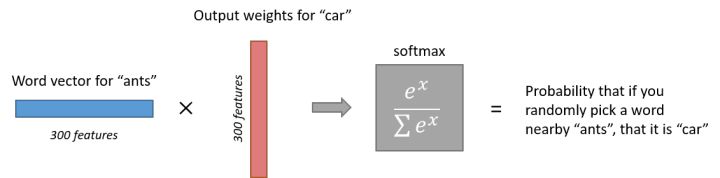
上面我们提到，input word和output word都会被我们进行one-hot编码。仔细想一下，我们的输入被one-hot编码以后大多数维度上都是0（实际上仅有一个位置为1），所以这个向量相当稀疏，那么会造成什么结果呢。如果我们将一个1 x 10000的向量和10000 x 300的矩阵相乘，它会消耗相当大的计算资源，为了高效计算，它仅仅会选择矩阵中对应的向量中维度值为1的索引行（这句话很绕），看图就明白。



这样模型中的隐层权重矩阵便成了一个”查找表“（lookup table），进行矩阵计算时，直接去查输入向量中取值为1的维度下对应的那些权重值。隐层的输出就是每个输入单词的“嵌入词向量”。

输出层

经过神经网络隐层的计算，ants这个词会从一个1 x 10000的向量变成1 x 300的向量，再被输入到输出层。输出层是一个softmax回归分类器，它的每个结点将会输出一个0-1之间的值（概率），这些所有输出层神经元结点的概率之和为1。下面是一个例子，训练样本为 (input word: “ants”， output word: “car”) 的计算示意图。



直觉上的理解

下面我们将通过直觉来进行一些思考。如果两个不同的单词有着非常相似的“上下文”（也就是窗口单词很相似，比如“Kitty climbed the tree”和“Cat climbed the tree”），那么通过我们的模型训练，这两个单词的嵌入向量将非常相似。

那么两个单词拥有相似的“上下文”到底是什么含义呢？比如对于同义词“intelligent”和“smart”，我们觉得这两个单词应该拥有相同的“上下文”。而例如”engine“和”transmission“这样相关的词语，可能也拥有着相似的上下文。

实际上，这种方法实际上也可以帮助你进行词干化（stemming），例如，神经网络对”ant“和”ants”两个单词会习得相似的词向量。

## 两种方法比较

总结来说就是CBOW模型中input是context（周围词）而output是中心词，训练过程中其实是在从output的loss学习周围词的信息也就是embedding，但是在中间层是average的，一共预测V(vocab size)次就够了。Skip-gram是用中心词预测周围词，预测的时候是一对word pair，等于对每一个中心词都有K个词作为output，对于一个词的预测有K次，所以能够更有效的从context中学习信息，但是总共预测K\*V词。

因此，skip gram的训练时间更长，但是对于一些出现频率不高的词，在CBOW中的学习效果就不及skip-gram。skip准确率比CBOW高。

## Skip-gram实例

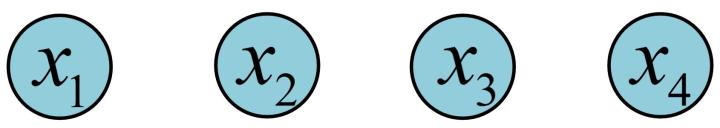
https: //github.com/tensorflow/tensorflow/blob/master/tensorflow/example／tutorials/word2vec/word2vec\_ basic.py



# Seq2seq

## N vs N

在实际应用中，我们还会遇到很多序列形的数据：



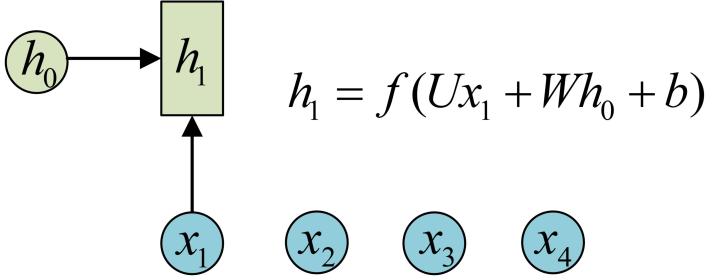
如：

自然语言处理问题。x1可以看做是第一个单词，x2可以看做是第二个单词，依次类推。

语音处理。此时，x1、x2、x3……是每帧的声音信号。

时间序列问题。例如每天的股票价格等等。

序列形的数据就不太好用原始的神经网络处理了。为了建模序列问题，RNN引入了隐状态h（hidden state）的概念，h可以对序列形的数据提取特征，接着再转换为输出。先从h1的计算开始看：



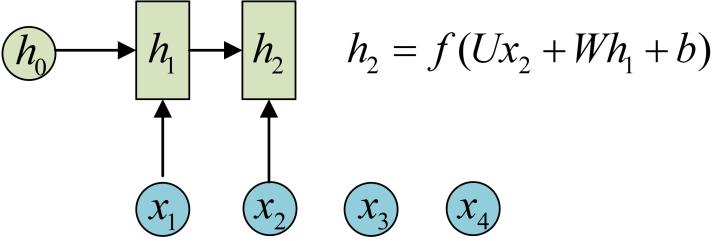
图示中记号的含义是：

圆圈或方块表示的是向量。

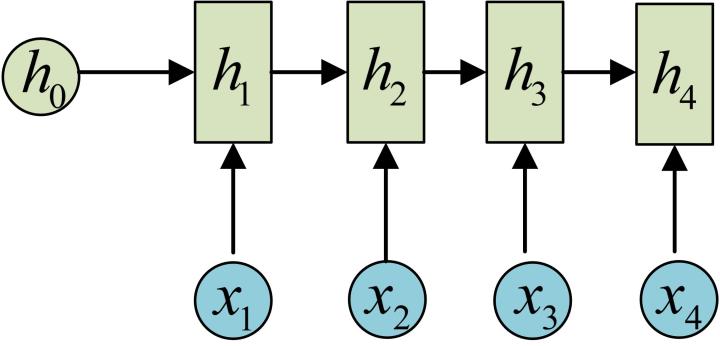
一个箭头就表示对该向量做一次变换。如上图中h0和x1分别有一个箭头连接，就表示对h0和x1各做了一次变换。

在很多论文中也会出现类似的记号，初学的时候很容易搞乱，但只要把握住以上两点，就可以比较轻松地理解图示背后的含义。

h2的计算和h1类似。要注意的是，在计算时，每一步使用的参数U、W、b都是一样的，也就是说每个步骤的参数都是共享的，这是RNN的重要特点，一定要牢记。



依次计算剩下来的（使用相同的参数U、W、b）：



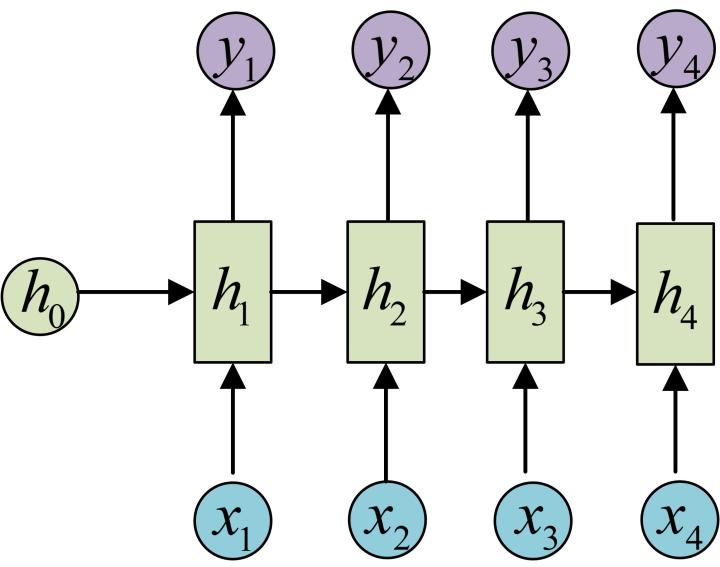
我们这里为了方便起见，只画出序列长度为4的情况，实际上，这个计算过程可以无限地持续下去。

我们目前的RNN还没有输出，得到输出值的方法就是直接通过h进行计算：



正如之前所说，一个箭头就表示对对应的向量做一次类似于f(Wx+b)的变换，这里的这个箭头就表示对h1进行一次变换，得到输出y1。

剩下的输出类似进行（使用和y1同样的参数V和c）：



OK！大功告成！这就是最经典的RNN结构，我们像搭积木一样把它搭好了。它的输入是x1, x2, .....xn，输出为y1, y2, ...yn，也就是说，**输入和输出序列必须要是等长的**。

由于这个限制的存在，经典RNN的适用范围比较小，但也有一些问题适合用经典的RNN结构建模，如：

计算视频中每一帧的分类标签。因为要对每一帧进行计算，因此输入和输出序列等长。

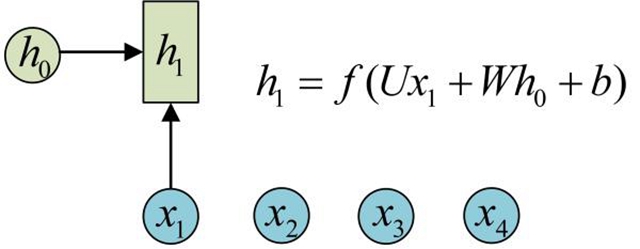
## N vs N应用(AI写诗)

一个Char RNN实现示例，可以用来写诗，生成歌词，甚至可以用来写网络小说!

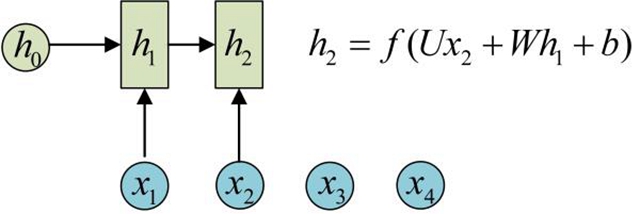
**一、学习单步的RNN：RNNCell**

如果要学习TensorFlow中的RNN，第一站应该就是去了解“RNNCell”，它是TensorFlow中实现RNN的基本单元，每个RNNCell都有一个call方法，使用方式是：(output, next\_state) = call(input, state)。

借助图片来说可能更容易理解。假设我们有一个初始状态h0，还有输入x1，调用call(x1, h0)后就可以得到(output1, h1)：

[](http://s1.51cto.com/wyfs02/M00/9D/80/wKioL1mBPYjxgNo6AAC4ehp1RCY693.jpg)

再调用一次call(x2, h1)就可以得到(output2, h2)：

[](http://s2.51cto.com/wyfs02/M01/9D/80/wKiom1mBPZfyRkypAACuJrWU3jU769.jpg)

也就是说，每调用一次RNNCell的call方法，就相当于在时间上“推进了一步”，这就是RNNCell的基本功能。

在代码实现上，RNNCell只是一个抽象类，我们用的时候都是用的它的两个子类BasicRNNCell和BasicLSTMCell。顾名思义，前者是RNN的基础类，后者是LSTM的基础类。这里推荐大家阅读其源码实现，一开始并不需要全部看一遍，只需要看下RNNCell、BasicRNNCell、BasicLSTMCell这三个类的注释部分，应该就可以理解它们的功能了。

除了call方法外，对于RNNCell，还有两个类属性比较重要：

state\_size

output\_size

前者是隐层的大小，后者是输出的大小。比如我们通常是将一个batch送入模型计算，设输入数据的形状为(batch\_size, input\_size)，那么计算时得到的隐层状态就是(batch\_size, state\_size)，输出就是(batch\_size, output\_size)。

可以用下面的代码验证一下(注意，以下代码都基于TensorFlow最新的1.2版本)：

import tensorflow as tf

import numpy as np

cell = tf.nn.rnn\_cell.BasicRNNCell(num\_units=128) # state\_size = 128

print(cell.state\_size) # 128

inputs = tf.placeholder(np.float32, shape=(32, 100)) # 32 是 batch\_size

h0 = cell.zero\_state(32, np.float32) # 通过zero\_state得到一个全0的初始状态，形状为(batch\_size, state\_size)

output, h1 = cell.call(inputs, h0) #调用call函数

print(h1.shape) # (32, 128)

对于BasicLSTMCell，情况有些许不同，因为LSTM可以看做有两个隐状态h和c，对应的隐层就是一个Tuple，每个都是(batch\_size, state\_size)的形状：

import tensorflow as tf

import numpy as np

lstm\_cell = tf.nn.rnn\_cell.BasicLSTMCell(num\_units=128)

inputs = tf.placeholder(np.float32, shape=(32, 100)) # 32 是 batch\_size

h0 = lstm\_cell.zero\_state(32, np.float32) # 通过zero\_state得到一个全0的初始状态

output, h1 = lstm\_cell.call(inputs, h0)

print(h1.h)  # shape=(32, 128)

print(h1.c)  # shape=(32, 128)

**二、学习如何一次执行多步：tf.nn.dynamic\_rnn**

基础的RNNCell有一个很明显的问题：对于单个的RNNCell，我们使用它的call函数进行运算时，只是在序列时间上前进了一步。比如使用x1、h0得到h1，通过x2、h1得到h2等。这样的h话，如果我们的序列长度为10，就要调用10次call函数，比较麻烦。对此，TensorFlow提供了一个tf.nn.dynamic\_rnn函数，使用该函数就相当于调用了n次call函数。即通过{h0,x1, x2, …., xn}直接得{h1,h2…,hn}。

具体来说，设我们输入数据的格式为(batch\_size, time\_steps, input\_size)，其中time\_steps表示序列本身的长度，如在Char RNN中，长度为10的句子对应的time\_steps就等于10。最后的input\_size就表示输入数据单个序列单个时间维度上固有的长度。另外我们已经定义好了一个RNNCell，调用该RNNCell的call函数time\_steps次，对应的代码就是：

# inputs: shape = (batch\_size, time\_steps, input\_size)

# cell: RNNCell

# initial\_state: shape = (batch\_size, cell.state\_size)。初始状态。一般可以取零矩阵

outputs, state = tf.nn.dynamic\_rnn(cell, inputs, initial\_state=initial\_state)

此时，得到的outputs就是time\_steps步里所有的输出。它的形状为(batch\_size, time\_steps, cell.output\_size)。state是最后一步的隐状态，它的形状为(batch\_size, cell.state\_size)。

此处建议大家阅读tf.nn.dynamic\_rnn的文档做进一步了解。

**三、学习如何堆叠RNNCell：MultiRNNCell**

很多时候，单层RNN的能力有限，我们需要多层的RNN。将x输入第一层RNN的后得到隐层状态h，这个隐层状态就相当于第二层RNN的输入，第二层RNN的隐层状态又相当于第三层RNN的输入，以此类推。在TensorFlow中，可以使用tf.nn.rnn\_cell.MultiRNNCell函数对RNNCell进行堆叠，相应的示例程序如下：

import tensorflow as tf

import numpy as np

# 每调用一次这个函数就返回一个BasicRNNCell

def get\_a\_cell():

return tf.nn.rnn\_cell.BasicRNNCell(num\_units=128)

# 用tf.nn.rnn\_cell MultiRNNCell创建3层RNN

cell = tf.nn.rnn\_cell.MultiRNNCell([get\_a\_cell() for \_ in range(3)]) # 3层RNN

# 得到的cell实际也是RNNCell的子类

# 它的state\_size是(128, 128, 128)

# (128, 128, 128)并不是128x128x128的意思

# 而是表示共有3个隐层状态，每个隐层状态的大小为128

print(cell.state\_size) # (128, 128, 128)

# 使用对应的call函数

inputs = tf.placeholder(np.float32, shape=(32, 100)) # 32 是 batch\_size

h0 = cell.zero\_state(32, np.float32) # 通过zero\_state得到一个全0的初始状态

output, h1 = cell.call(inputs, h0)

print(h1) # tuple中含有3个32×128的向量

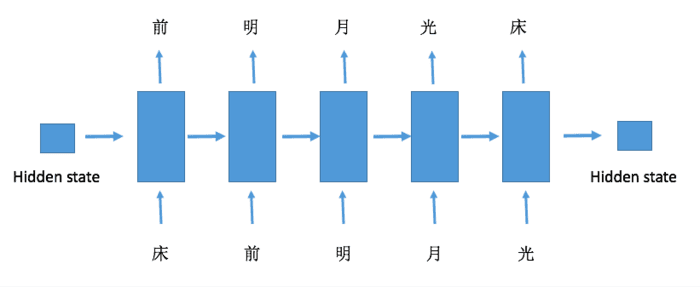
通过MultiRNNCell得到的cell并不是什么新鲜事物，它实际也是RNNCell的子类，因此也有call方法、state\_size和output\_size属性。同样可以通过tf.nn.dynamic\_rnn来一次运行多步。

此处建议阅读MutiRNNCell源码中的注释进一步了解其功能。

**四、训练过程**

一般而言，RNN的输入和输出存在着多种关系，比如1对多，多对多等等，不同的输入输出关系对应着不同的应用，网上也有很多这方面的文章可以去看看，这里我们要讲的Char RNN在训练网络的时候是相同长度的多对多的类型，也就是输入一个序列，输出一个相同的长度的序列。

具体的网络结构就是下面这个样子



完整代码见：<https://github.com/hzy46/Char-RNN-TensorFlow>

## N vs M

Seq2Seq模型是RNN最重要的一个变种：N vs M（输入与输出序列长度不同）。

这种结构又叫Encoder-Decoder模型。

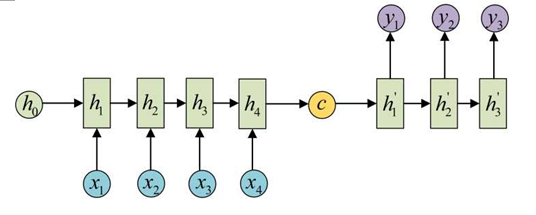
原始的N vs N RNN要求序列等长，然而我们遇到的大部分问题序列都是不等长的，如机器翻译中，源语言和目标语言的句子往往并没有相同的长度。

**为此，Encoder-Decoder结构先将输入数据编码成一个上下文向量c：**

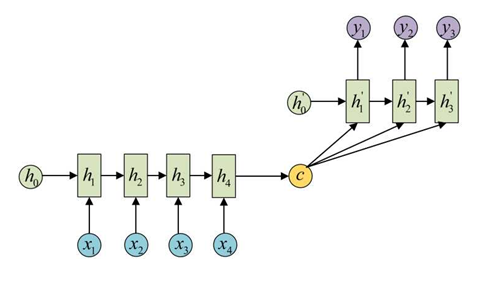


得到c有多种方式，最简单的方法就是把Encoder的最后一个隐状态赋值给c，还可以对最后的隐状态做一个变换得到c，也可以对所有的隐状态做变换。

**拿到c之后，就用另一个RNN网络对其进行解码**，这部分RNN网络被称为Decoder。具体做法就是将c当做之前的初始状态h0输入到Decoder中：



还有一种做法是将c当做每一步的输入：



由于这种Encoder-Decoder结构不限制输入和输出的序列长度，因此应用的范围非常广泛，比如：

**机器翻译。**Encoder-Decoder的最经典应用，事实上这一结构就是在机器翻译领域最先提出的

**文本摘要**。输入是一段文本序列，输出是这段文本序列的摘要序列。

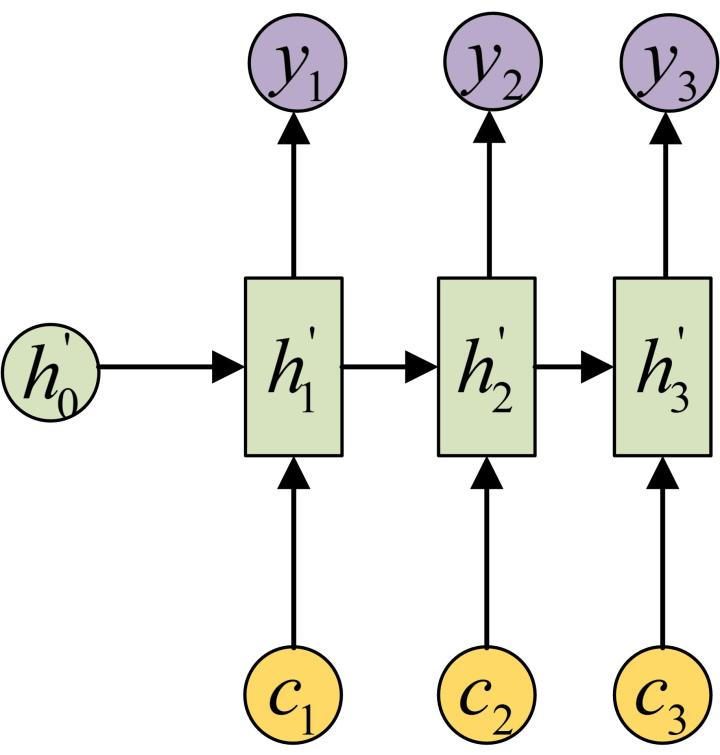
**阅读理解。**将输入的文章和问题分别编码，再对其进行解码得到问题的答案。

**语音识别。**输入是语音信号序列，输出是文字序列。

## Attention机制

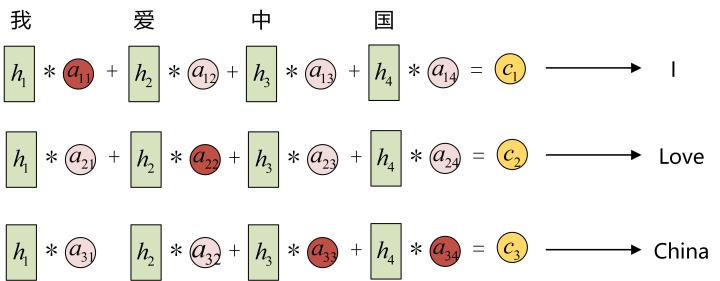
在Encoder-Decoder结构中，Encoder把所有的输入序列都编码成一个统一的语义特征c再解码，**因此， c中必须包含原始序列中的所有信息，它的长度就成了限制模型性能的瓶颈。**如机器翻译问题，当要翻译的句子较长时，一个c可能存不下那么多信息，就会造成翻译精度的下降。

Attention机制通过在每个时间输入不同的c来解决这个问题，下图是带有Attention机制的Decoder：



每一个c会自动去选取与当前所要输出的y最合适的上下文信息。具体来说，我们用 aij 衡量Encoder中第j阶段的hj和解码时第i阶段的相关性，最终Decoder中第i阶段的输入的上下文信息 ci 就来自于所有 hj 对 aij 的加权和。

以机器翻译为例（将中文翻译成英文）：

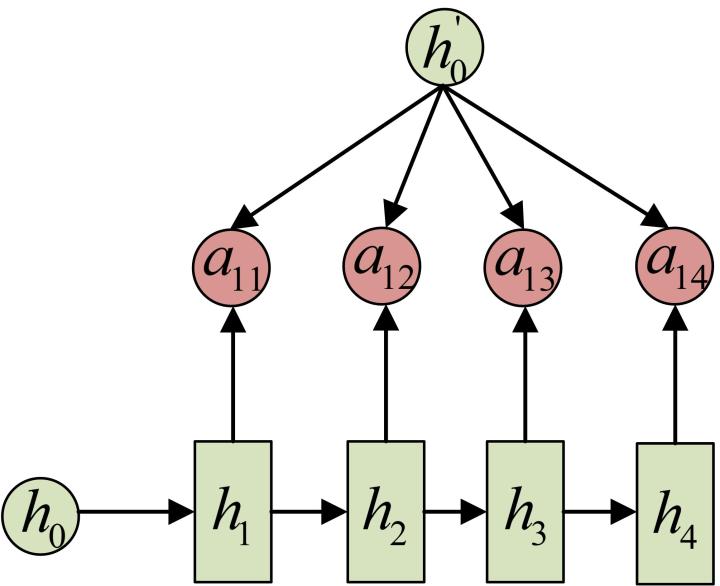


输入的序列是“我爱中国”，因此，Encoder中的h1、h2、h3、h4就可以分别看做是“我”、“爱”、“中”、“国”所代表的信息。在翻译成英语时，第一个上下文c1应该和“我”这个字最相关，因此对应的 a11 就比较大，而相应的 a12、 a13、 a14 就比较小。c2应该和“爱”最相关，因此对应的 a22 就比较大。最后的c3和h3、h4最相关，因此 a33 、 a34 的值就比较大。

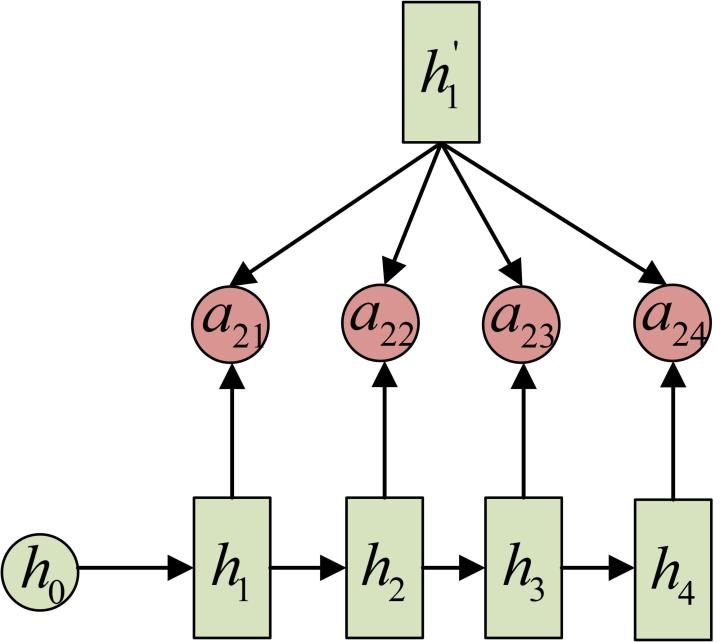
至此，关于Attention模型，我们就只剩最后一个问题了，那就是：这些权重 aij 是怎么来的？

事实上， aij 同样是从模型中学出的，它实际和Decoder的第i-1阶段的隐状态、Encoder第j个阶段的隐状态有关。

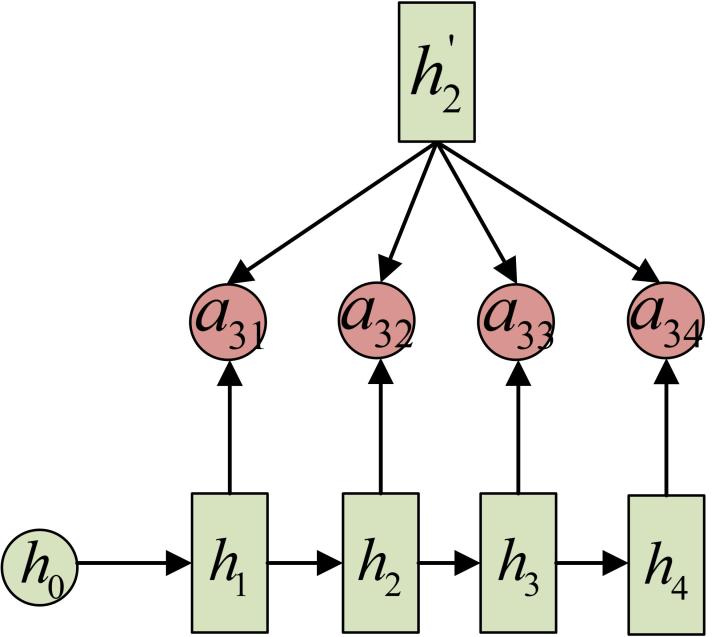
同样还是拿上面的机器翻译举例， a1j 的计算（此时箭头就表示对h'和 hj 同时做变换）：



a2j的计算：



a3j的计算：



## N vs M应用（聊天机器人）

<https://blog.csdn.net/irving_zhang/article/details/79088143>