# 分类算法总结

## k近邻算法

### 原理

近朱者赤近墨者黑，如果一个待分类样本在特征空间的k个最相似（即特征空间中K近邻）的样本中的大多数属于某一个类别，则这样本也属于这个类别

### 输入

**数据集**：样本中数据集中每个样本是有标签的

**样本间距离的计算方法**：欧式距离、余弦距离、汉明距离、曼哈顿距离

**K值得选取：**K值太大，分类偏差大（极端方法，K值是所有样本数）

K值太小，方差大容易过拟合

K值一般在3~10之间，或者是k等于训练数据的平方根

### 输出

预测样本的分类结果

### 优点

1、简单，易于理解，易于实现，无需参数估计，无需训练

2、对异常值不敏感（个别噪音数据对结果的影响不是很大）

3、适合对稀有事件进行分类

4、适合于多分类问题（对象具有多个类别标签），KNN要比SVM表现要好

### 缺点

1、对测试样本分类时的计算量大，内存开销大（需要保存全部的数据集），因为对每一个待分类的文本都要计算它到全体已知样本的距离，才能求得它的K个最近邻点（耗时）。

2、可解释性差，无法告诉你哪个变量更加重要，无法给出决策树那样的规则/

3、K值得选择：样本不均衡时，容易偏向样本中类别多的那个类别/

4、KNN是一种消极学习方法、懒惰算法

**积极学习法**：先根据训练集构造出分类模型，根据分类模型对测试集分类。

**消极学习法**（基于实例的学习法）：推迟建模，当给定训练元组时，简单地存储训练数据，一直等到给定一个测试样本。

### 性能问题

使用KNN，可以很容易的构造模型，但在对待分类样本进行分类时，为了获得K近邻，必须采用暴力搜素的方式，扫描全部训练样本并计算其与待分类样本之间的距离，系统开销很大

### 应用KNN常见的问题

K值设定

**经验规则：k一般低于训练样本数的平方根**

类别如何判定最合适？

投票法没有考虑近邻的距离的远近，距离更近的近邻也许更应该决定最终的分类，所以加权投票法更恰当一些。

如何选择合适的距离衡量？

高维度对距离衡量的影响：众所周知当变量数越多，欧式距离的区分能力就越差

训练样本是否要一视同仁？

在训练集中，有些样本可能是更值得依赖的。 可以给不同的样本施加不同的权重，加强依赖样本的权重，降低不可信赖样本的影响

### KNN算法的应用领域

推荐系统、字符识别、文本分类、图像识别、改进约会网站、手写识别系统、电影分类

### 代码

dataSetSize = dataSet.shape[0]

a = array([4,3])

diffMat = tile(intX,(dataSetSize,1))-dataSet

sqDiffMat = diffMat \*\*2

sqDistances = sqDiffMat.sum(axis=1)

distances = sqDistances\*\*0.5

sortedDistIndicies = distances.argsort()

classCount={}

for i in range(k):

voteIlabel = labels[sortedDistIndicies[i]]

classCount[voteIlabel]=classCount.get(voteIlabel,0) + 1

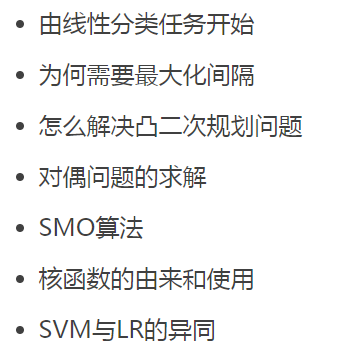
print classCount

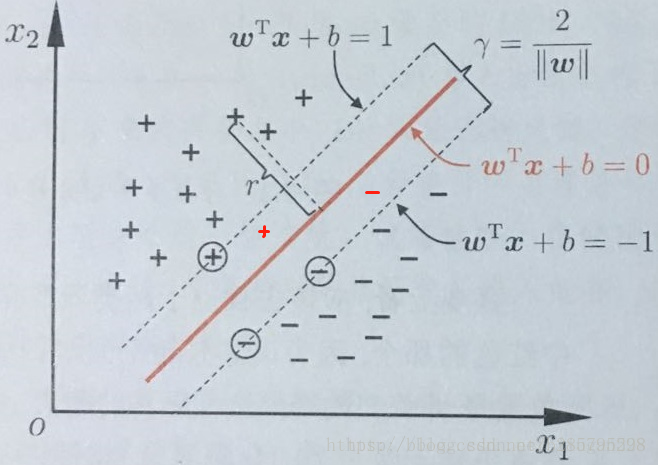
sortedClassCount = sorted(classCount.iteritems(),key = operator.itemgetter(1),reverse =True)

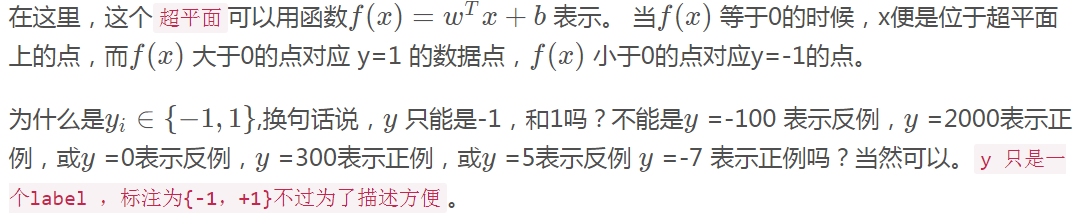
return sortedClassCount[0][0]

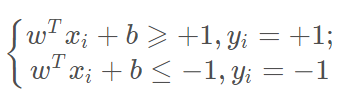
## 支持向量机

### 原理



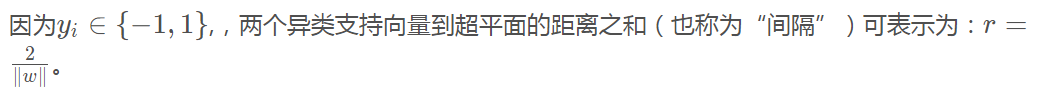


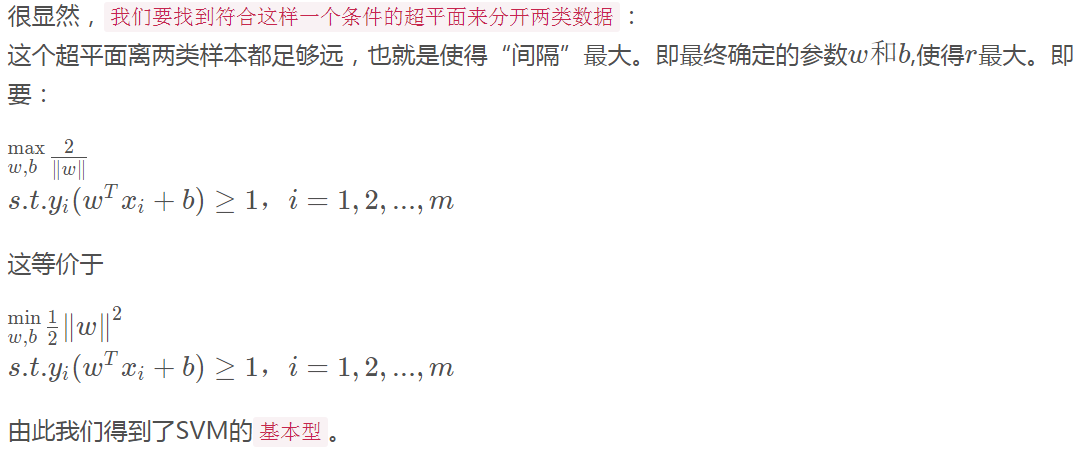


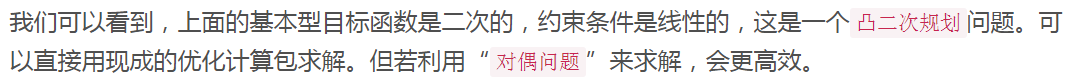


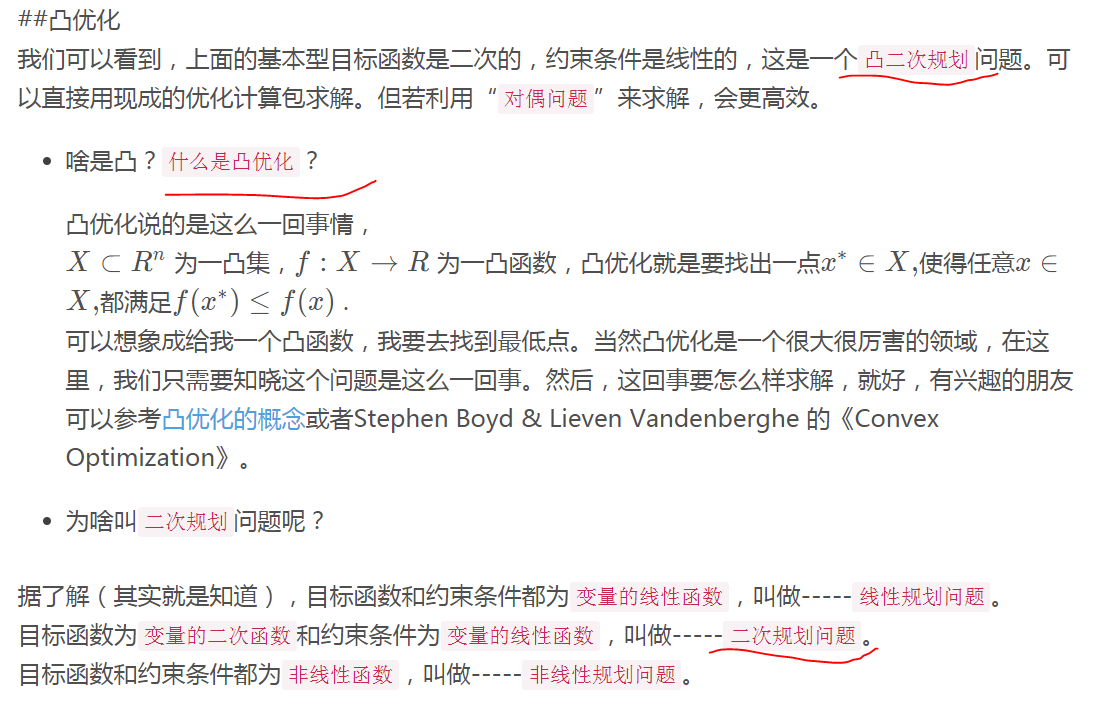
点到直线的距离为：

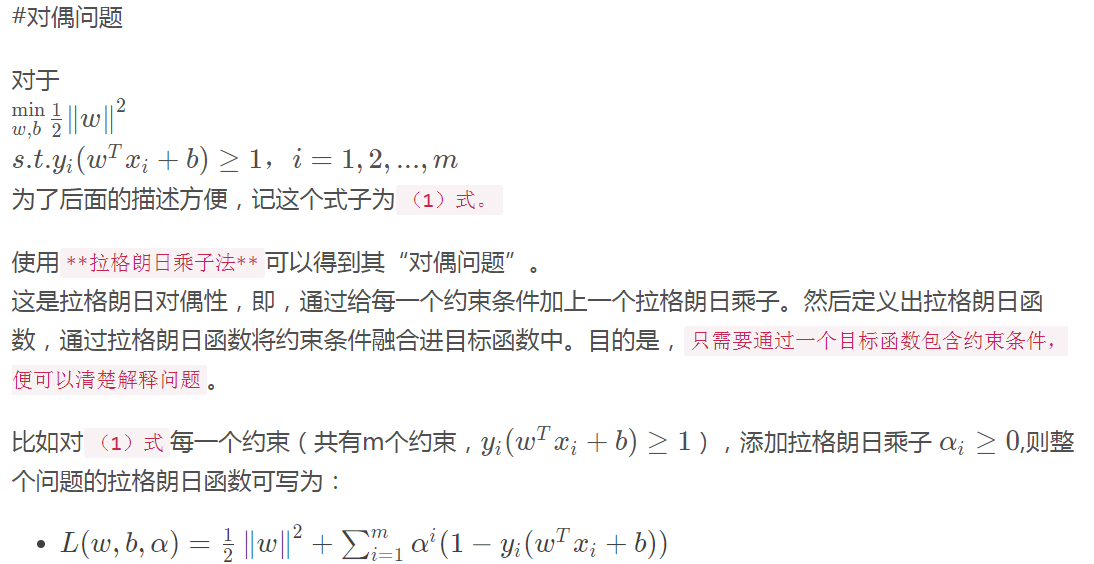


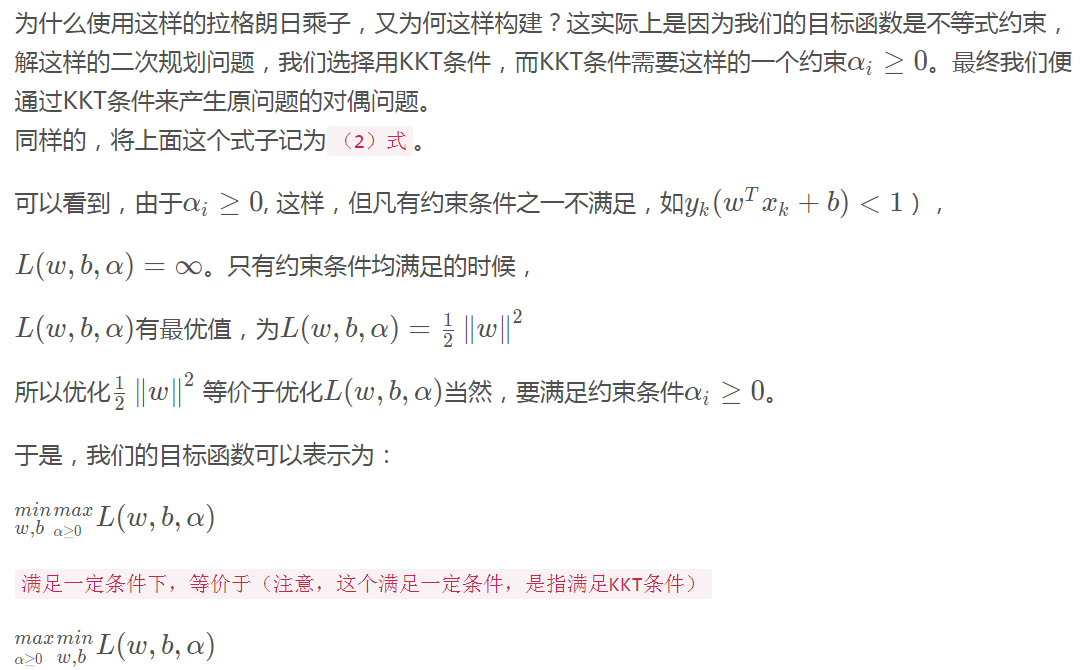


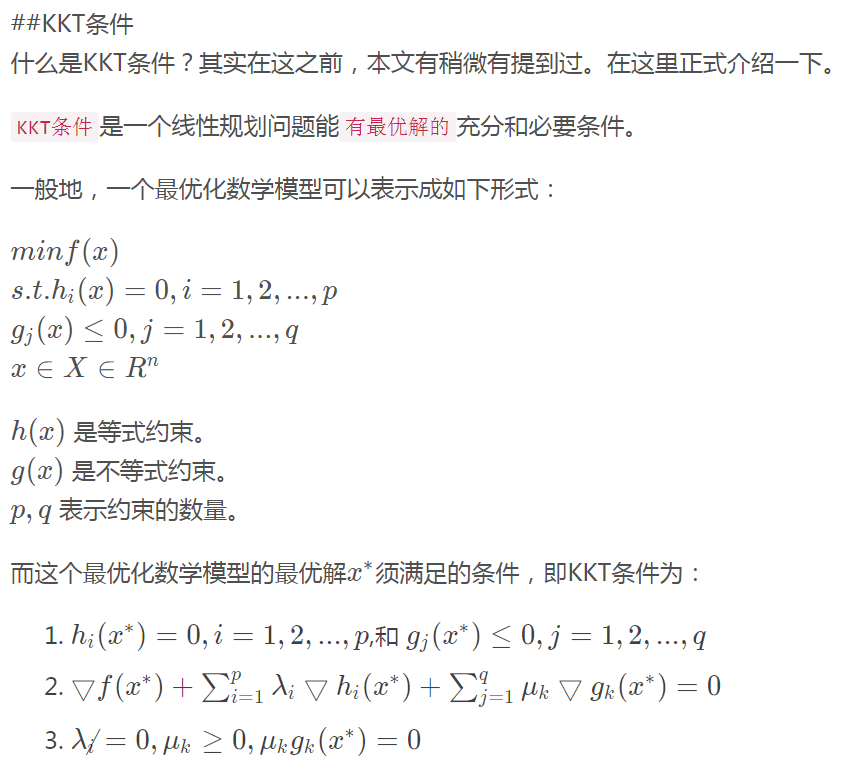


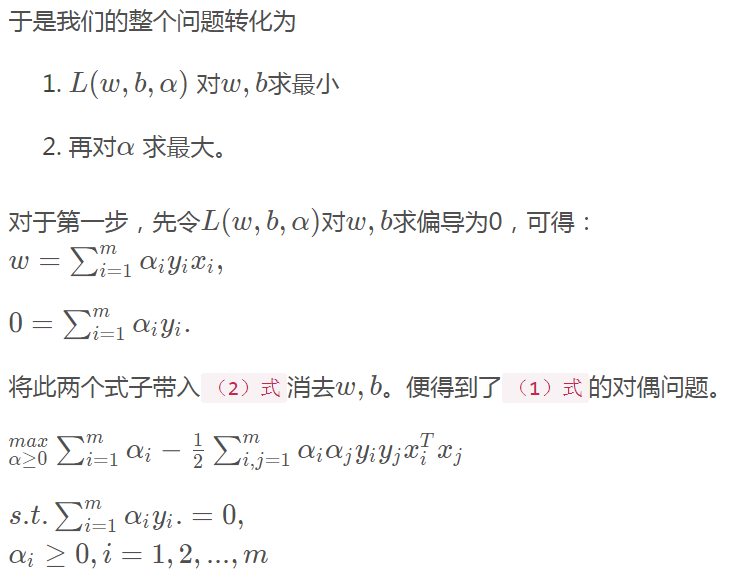


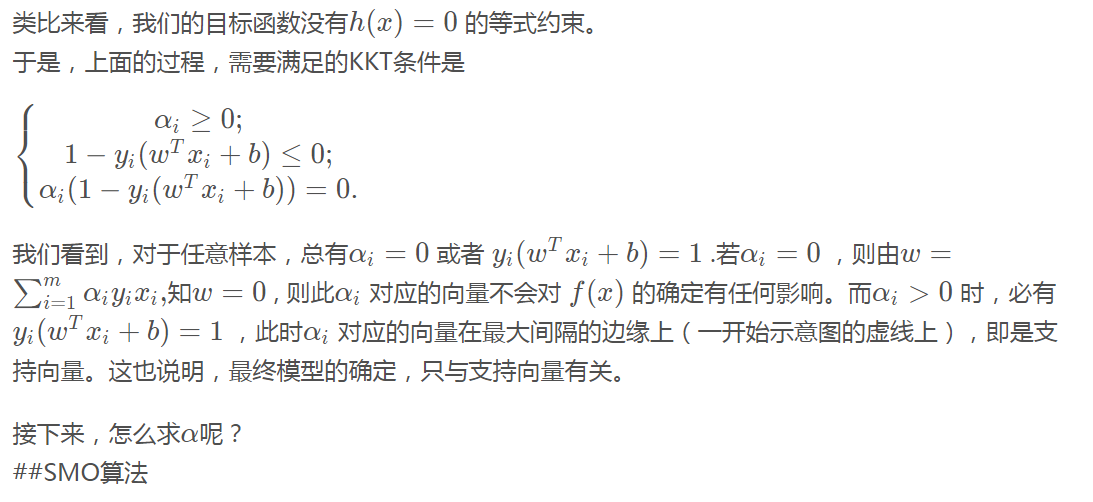








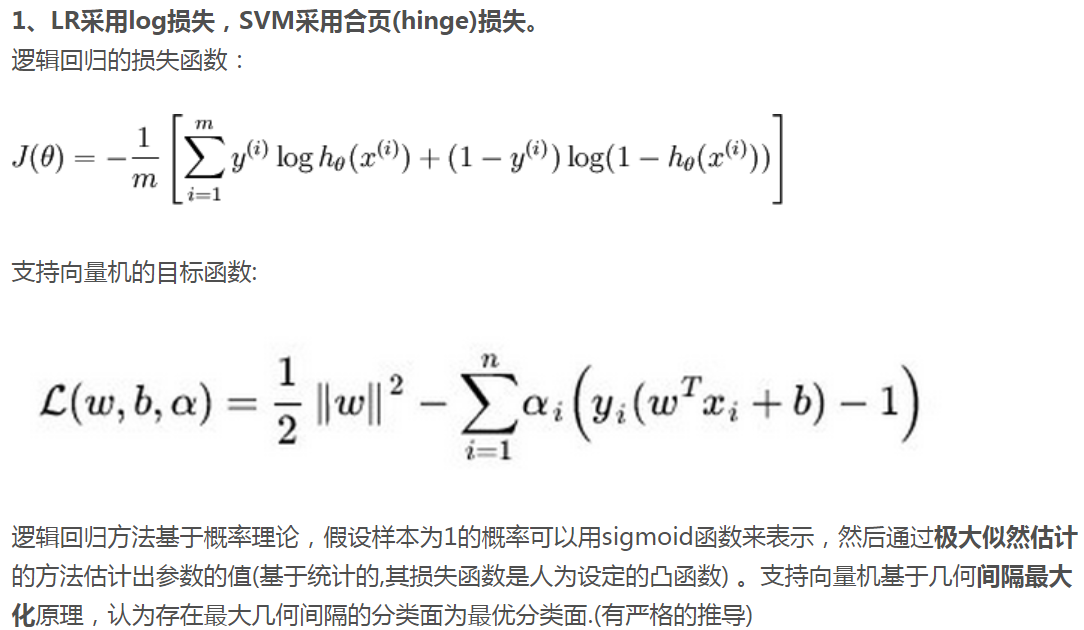




<https://blog.csdn.net/b285795298/article/details/81977271>

### 损失函数

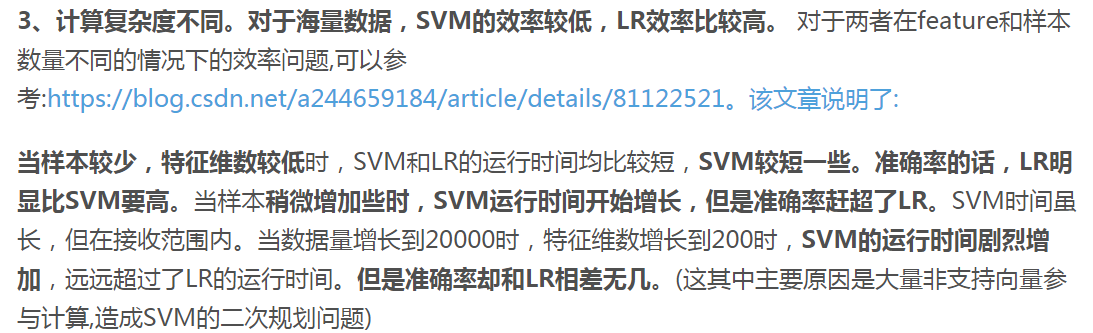
### SVM与LR的不同点



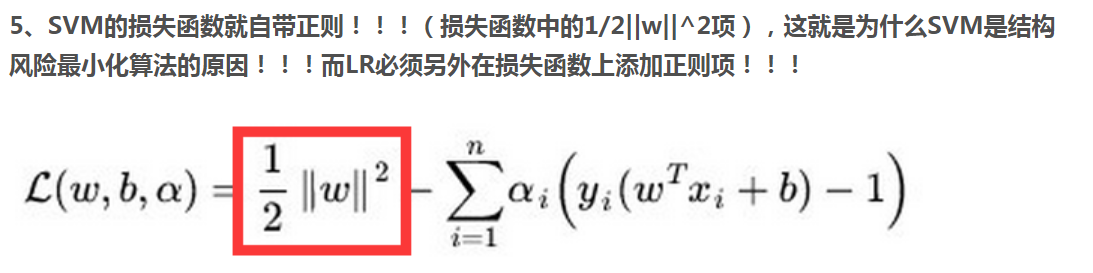


支持向量机改变非支持向量样本并不会引起决策面的变化

逻辑回归中任何样本都会引起决策面的变化







### 优点

1、对于线性不可分的情况可以通过核函数，映射到高维特征空间实现线性可分。

2、SVM学习问题可以表示为凸优化问题，因此可以利用已知的有效算法发现目标函数的全局最小值。而其他分类方法（如基于规则的分类器和人工神经网络）都采用一种基于贪心学习的策略来搜索假设空间，这种方法一般只能获得局部最优解。

3、小集群分类效果好

### 缺点

1、SVM仅仅只限于一个二类分类问题，对于多分类问题解决效果并不好。

2、仅局限于小集样本，对于观测样本太多时，效率较低；由于SVM是借助二次规划来求解支持向量，而求解二次规划将涉及m阶矩阵的计算（m为样本个数），当m数组很大时该矩阵的存储和计算将耗费大量的内存和运算时间

3、寻求合适的核函数相当难

### 应用领域

小样本分类

### 代码

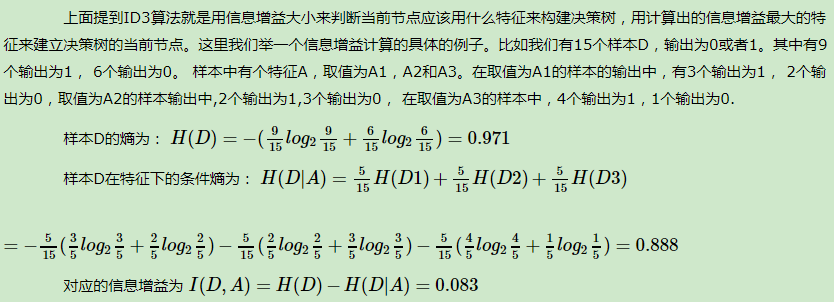
需要调试的参数，软间隔中有个惩罚错误分类因子C，还有核函数的选择，核函数里的参数也需要调整

## 决策树

### ID3原理



信息增益： ，熵越大，不确定性越大



决策树ID3正是采用这个信息增益为基础的，采用增益最大的特征作为划分特征

### 优点

简单

### 缺点

1、ID3没有考虑连续特征，比如长度，密度都是连续值，无法在ID3运用

2、ID3采用信息增益大的特征优先建立决策树节点。数据不平衡问题，会影响

3、ID3算法对缺失值得情况没有考虑

4、没有考虑过拟合问题

### 应用领域

### 代码

### C4.5原理



信息增益与特征熵之比

### 优点

1、将连续特征离散化。取相邻两样本值得平均数，分别计算以该点作为二元分类点时的信息增益。解决了ID3不能处理的连续值问题

2、信息熵增益与特征熵的比值，校正信息增益容易倾向于取值较多的特征的问题

3、解决缺失值问题。设置权重来解决

4、引入了正则化系数进行初步的剪枝

### 缺点

1. 容易过拟合
2. C4.5生成的是多叉树，即一个父节点可以有多个节点。很多时候，计算机中二叉树模型会比多叉树运算效率高。采用二叉树，可以提高效率
3. C4.5只能用于分类

4、基于信息熵，涉及到大量对数运算，比较耗时

### CART分类树原理

基尼系数：

对于给定样本D，假设有K个类别，第k个类别的数量为 ，则样本D的基尼系数表达式为：

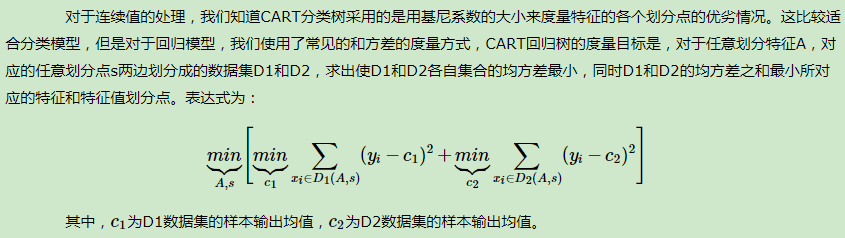
对于样本D，如果根据特征A的某个值a，把D分为D1和D2两部分，则在特征A的条件下，D的基尼系数表达式为：



基尼系数代表了模型的不纯度，基尼系数越小，则不纯度越低，特征越好。

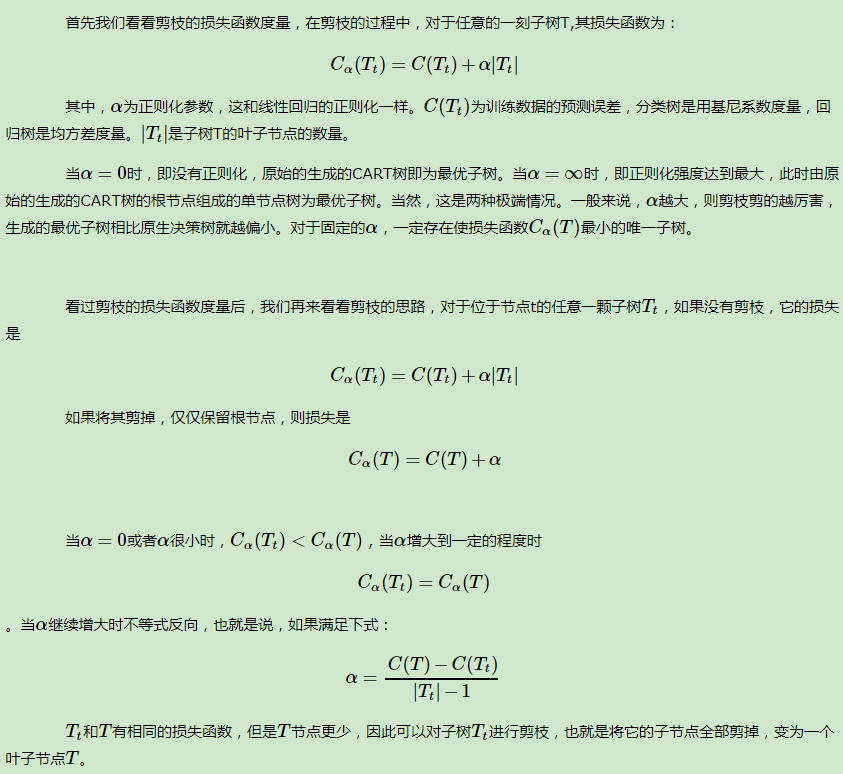
CART算法建立起来的是二叉树

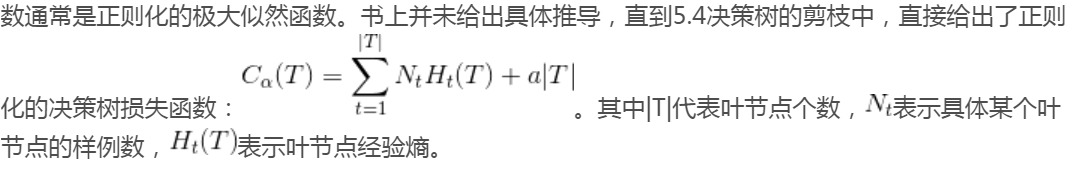
### CART回归原理

CART剪枝损失函数







剪枝过程，计算各个节点的正则化阈值，选择a最小的作为阈值，进行剪枝

### 决策树算法小结



优点

1）简单直观，生成的决策树很直观。

　　　　2）基本不需要预处理，不需要提前归一化，处理缺失值。

　　　　3）使用决策树预测的代价是O(log2m)O(log2m)。 m为样本数。

　　　　4）既可以处理离散值也可以处理连续值。很多算法只是专注于离散值或者连续值。

　　　　5）可以处理多维度输出的分类问题。

　　　　6）相比于神经网络之类的黑盒分类模型，决策树在逻辑上可以得到很好的解释

　　　　7）可以交叉验证的剪枝来选择模型，从而提高泛化能力。

　　　　8） 对于异常点的容错能力好，健壮性高。

缺点

1）决策树算法非常容易过拟合，导致泛化能力不强。可以通过设置节点最少样本数量和限制决策树深度来改进。

　　　　2）决策树会因为样本发生一点点的改动，就会导致树结构的剧烈改变。这个可以通过集成学习之类的方法解决。

　　　　3）寻找最优的决策树是一个NP难的问题，我们一般是通过启发式方法，容易陷入局部最优。可以通过集成学习之类的方法来改善。

　　　　4）有些比较复杂的关系，决策树很难学习，比如异或。这个就没有办法了，一般这种关系可以换神经网络分类方法来解决。

　　　　5）如果某些特征的样本比例过大，生成决策树容易偏向于这些特征。这个可以通过调节样本权重来改善。

## 朴素贝叶斯

### 原理



要预测当Y取哪个类别时， 概率最大，则采用上面的贝叶斯公式：目标是



由于对于所有的类别计算时，上诉的分母是一样的，因此，预测公式可以简化为：



接着利用朴素贝叶斯的独立性假设(如果特征之间非常不独立怎么办，那就尽量不要用朴素贝叶斯了)，就可以得到通常意义上的朴素贝叶斯推断公式：



### 优点

1、朴素贝叶斯模型有稳定的分类效率

2、对小规模的数据表现很好，能处理多分类任务，适合增量式训练，尤其是数据量超出内存时，可以一批批的去增量训练

3、对缺失数据不太敏感，算法也比较简单，常用于文本分类

### 缺点

1、理论上，朴素贝叶斯模型与其他分类方法相比具有最小的误差率。但是实际上并非总是如此，这是因为朴素贝叶斯模型给定输出类别的情况下,假设属性之间相互独立，这个假设在实际应用中往往是不成立的，在属性个数比较多或者属性之间相关性较大时，分类效果不好。而在属性相关性较小时，朴素贝叶斯性能最为良好。对于这一点，有半朴素贝叶斯之类的算法通过考虑部分关联性适度改进。

2、需要知道先验概率，且先验概率很多时候取决于假设，假设的模型可以有很多种，因此在某些时候会由于假设的先验模型的原因导致预测效果不佳。

3、由于我们是通过先验和数据来决定后验的概率从而决定分类，所以分类决策存在一定的错误率。

4、对输入数据的表达形式很敏感。

### 应用场景

文本分类

### 代码

对文本的统计有两种模式，词袋型和词库型

## 逻辑回归

### 原理

损失函数最小化，损失函数由最大似然法来推导



### 优点

1、形式简单，模型的可解释性非常好

2、训练速度较快。分类的时候，计算量仅仅只和特征的数目相关。并且逻辑回归的分布式优化sgd发展比较成熟，训练的速度可以通过堆机器进一步提高，这样我们可以在短时间内迭代好几个版本的模型。

3、资源占用小,尤其是内存。因为只需要存储各个维度的特征值

### 缺点

1、准确率并不是很高。因为形式非常的简单(非常类似线性模型)，很难去拟合数据的真实分布

2、很难处理数据不平衡的问题。举个例子：如果我们对于一个正负样本非常不平衡的问题比如正负样本比 10000:1.我们把所有样本都预测为正也能使损失函数的值比较小。但是作为一个分类器，它对正负样本的区分能力不会很好。

3、处理非线性数据较麻烦。逻辑回归在不引入其他方法的情况下，只能处理线性可分的数据，或者进一步说，处理二分类的问题

4、逻辑回归本身无法筛选特征。有时候，我们会用gbdt来筛选特征，然后再上逻辑回归。

### 应用场景

### 代码

损失函数一般有四种，平方损失函数，对数损失函数，HingeLoss0-1损失函数，绝对值损失函数

调参包括，正则化系数，梯度下降次数，步长

库函数使用参数调整：

正则化选择参数：penalty，是L1还是L2

优化算法选择参数：solve 对逻辑所示函数的优化方法

分类方式选择参数：multi\_class 分类方式，一对多还是一对一

类型权重参数： class\_weight 标示样本中各类型的不同权重

样本权重参数： sample\_weight 样本不平衡问题中，对样本赋予的权重

# 深度学习算法总结

## 图像

### 卷积层

### 池化层

### 全连接层

## NLP

### Tf-idf

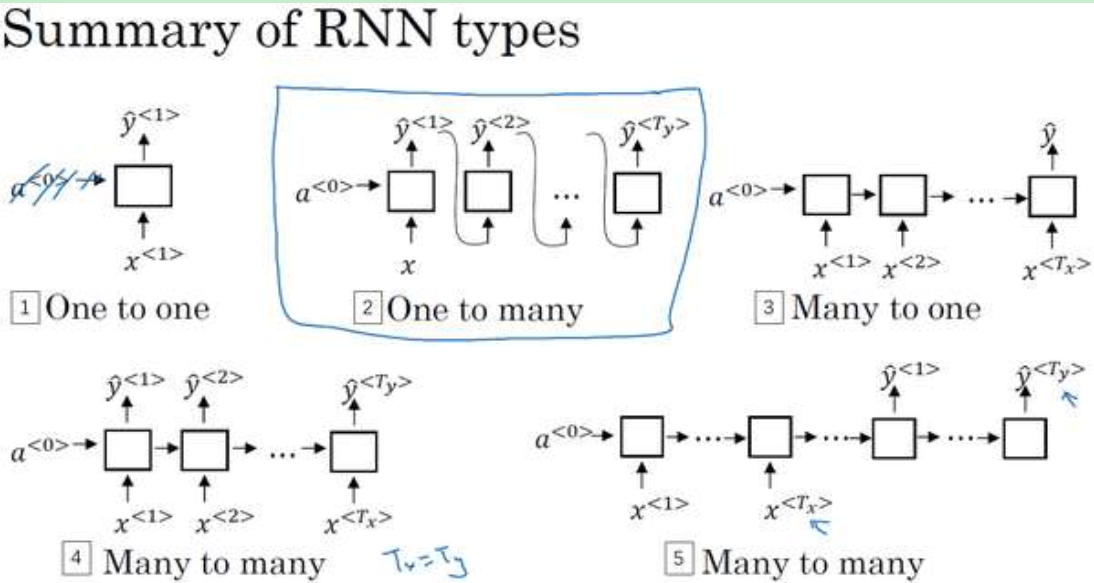
TF-IDF（Term Frequence Inverse Document Frequence）是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度

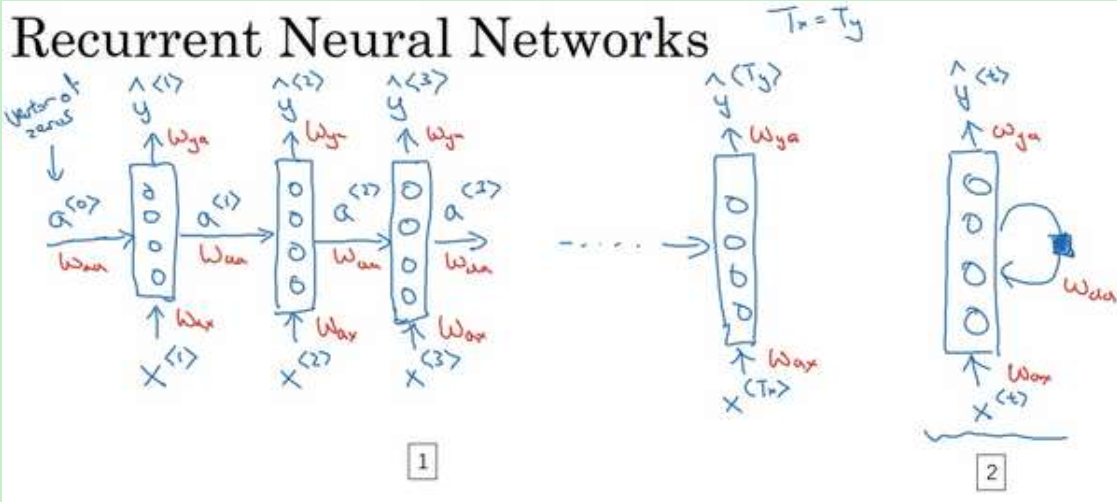
字词的重要性随着它在文件中出现的次数成正比增加。但同时会随着它在语料库中出现的频率成反比下降

如果包含词条w的文档越少，IDF越大，则说明词条具有很好的类别区分能力；某一特定词语的IDF，可以由总文件数目除以包含该词语之文件的数目，再将商取对数

该词的tfidf值为：

### RNN基本类型





RNN的每层参数是一样的，也就是相同层不同时间步之间的参数是一样的 ；

不同层之间的参数是不一样的；

# 改善网络

## 梯度下降（Gradient Descent）

### 批量梯度下降（Batch Gradient Desent）

批量梯度下降法（Batch Gradient Descent，简称BGD）是梯度下降法最原始的形式，它的具体思路是在更新每一参数时都使用所有的样本来进行更新，其数学形式如下：

从上面公式可以注意到，它得到的是一个全局最优解，但是每迭代一步，都要用到训练集**所有的数据**，如果样本数目mm很大，那么可想而知这种方法的迭代速度！所以，这就引入了另外一种方法，随机梯度下降。   
**优点**：全局最优解；易于并行实现；   
**缺点**：当样本数目很多时，训练过程会很慢

**总结：使用全部的样本进行目标参数更新，空间消耗大**

### 随机梯度下降（Stochastic Gradient Descent）

由于批量梯度下降法在更新每一个参数时，都需要所有的训练样本，所以训练过程会随着样本数量的加大而变得**异常的缓慢**(需要更多的迭代次数)。随机梯度下降法（Stochastic Gradient Descent，简称SGD）正是为了解决批量梯度下降法这一弊端而提出的。随机梯度下降是仅针对一个样本来进行梯度更新。

随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况（例如几十万），那么可能只用其中几万条或者几千条的样本，就已经将theta迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，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。

优点：训练速度快；

缺点：准确度下降，并不是全局最优；不易于并行实现。

在写代码时，一个迭代次数，指的是要把整个数据里的每个样本的梯度都要更新一遍

for j in range(numIter):

dataIndex = list(range(m))

for i in range(m):

alpha = 4/(1.0+j+i)+0.01

#降低alpha的大小，每次减小1/(j+i)。

randIndex = int(random.uniform(0,len(dataIndex)))

#随机选取样本

h=sigmoid(sum(dataMatrix[randIndex]\*weights))

#选择随机选取的一个样本，计算h

error = classLabels[randIndex] - h

#计算误差

weights = weights + alpha \* error \* dataMatrix[randIndex]

#更新回归系数

del(dataIndex[randIndex])

**总结：每次使用1个样本进行目标参数更新，需要迭代次数多**

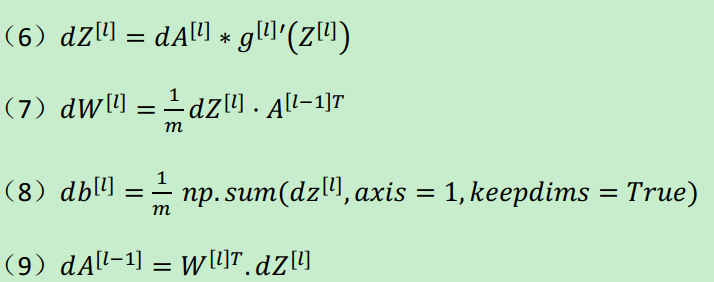
### 小批量梯度下降（Mini-batch Gradient Descent）

有上述的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

mini-batch z在写代码时同样时一个迭代次数，将所有批次的梯度都更新一遍

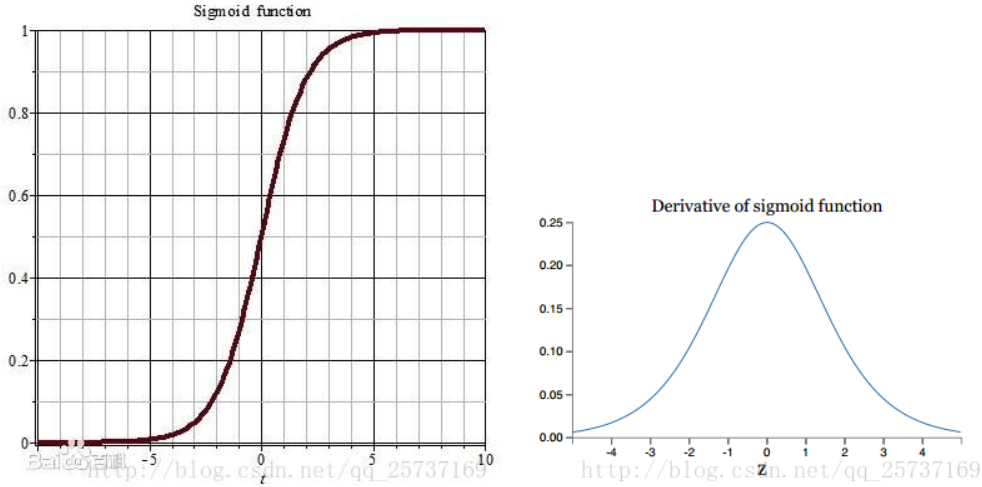
总结：**MBGD：每次使用batch个数据进行目标参数更新**

### 梯度消失、爆炸

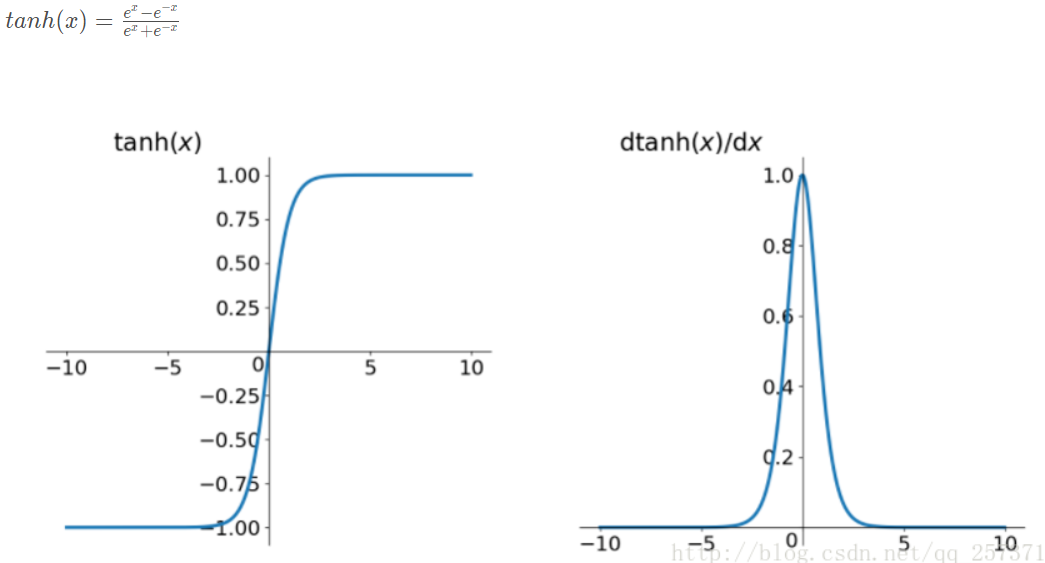


各层的梯度计算公式如上所示，梯度的值与W有关，与激活函数的导数有关，

如果此部分大于1，那么层数增多的时候，最终的求出的梯度更新将以指数形式增加，即发生**梯度爆炸**，如果此部分小于1，那么随着层数增多，求出的梯度更新信息将会以指数形式衰减，即发生了**梯度消失。**



sigmoid函数导数最大值为0.25很容易发生梯度消失。



tanh函数的倒数在接近于1的位置

### 梯度消失爆炸的解决方法

1. 方案1 预训练加微调

此方法来自Hinton在2006年发表的一篇论文，Hinton为了解决梯度的问题，提出采取无监督逐层训练方法，其基本思想是每次训练一层隐节点，训练时将上一层隐节点的输出作为输入，而本层隐节点的输出作为下一层隐节点的输入，此过程就是逐层“预训练”（pre-training）；在预训练完成后，再对整个网络进行“微调”（fine-tunning）。Hinton在训练深度信念网络（Deep Belief Networks中，使用了这个方法，在各层预训练完成后，再利用BP算法对整个网络进行训练。此思想相当于是先寻找局部最优，然后整合起来寻找全局最优，此方法有一定的好处，但是目前应用的不是很多了

1. 方案2 梯度剪切、正则

**梯度剪切**这个方案主要是针对**梯度爆炸**提出的，其思想是设置一个梯度剪切阈值，然后更新梯度的时候，如果梯度超过这个阈值，那么就将其强制限制在这个范围之内。这可以防止梯度爆炸。

另外一种解决梯度爆炸的手段是采用权重正则化（weithts regularization）比较常见的是l1l1l1正则，和l2l2l2正则，在各个深度框架中都有相应的API可以使用正则化，比如在tensorflow中，若搭建网络的时候已经设置了正则化参数，则调用以下代码可以直接计算出正则损失：

regularization\_loss= tf.add\_n(tf.losses.get\_regularization\_losses(scope='my\_resnet\_50'))

正则化是通过对网络权重做正则限制过拟合，仔细看正则项在损失函数的形式：

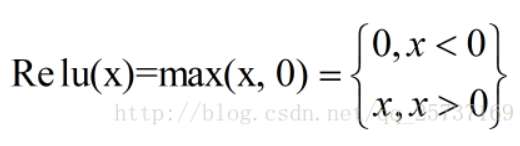


其中，α\alphaα是指正则项系数，因此，如果发生梯度爆炸，权值的范数就会变的非常大，通过正则化项，可以部分限制梯度爆炸的发生。

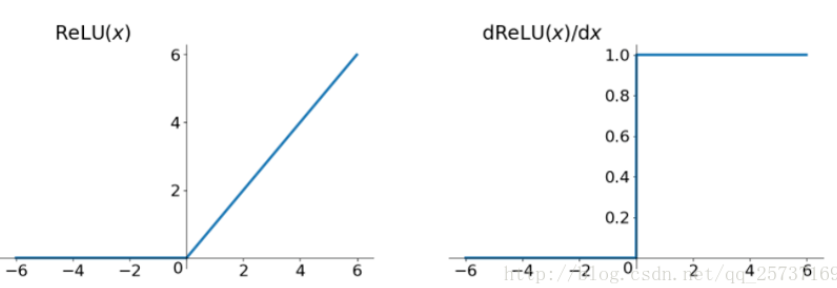
注：事实上，在深度神经网络中，往往是梯度消失出现的更多一些。

1. 方案3 relu、leakrelu、elu等激活函数

\*\*Relu:\*\*思想也很简单，如果激活函数的导数为1，那么就不存在梯度消失爆炸的问题了，每层的网络都可以得到相同的更新速度，relu就这样应运而生。先看一下relu的数学表达式：



其函数图像：



从上图中，我们可以很容易看出，relu函数的导数在正数部分是恒等于1的，因此在深层网络中使用relu激活函数就不会导致梯度消失和爆炸的问题。

relu的主要贡献在于：

（1）解决了梯度消失、爆炸的问题

（2）计算方便、计算速度快

（3）加速了网络的训练

同时也存在一些缺点：

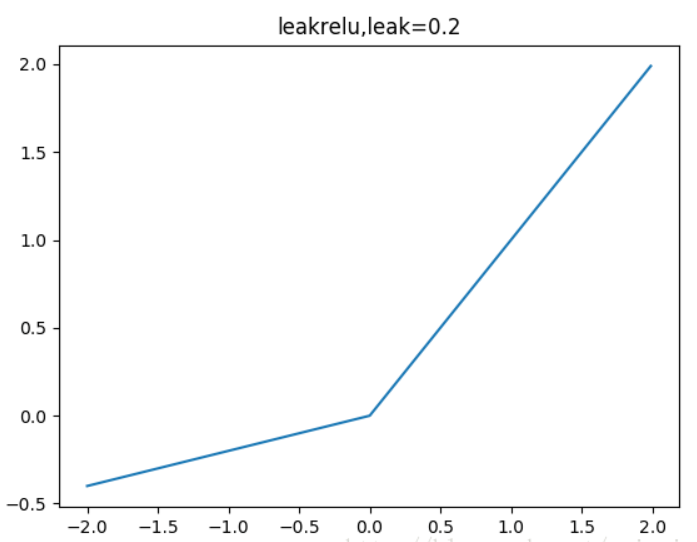
（1）由于负数部分恒为0，会导致一些神经元无法激活（可通过设置小学习率部分解决）

（2）输出不是以0为中心的

尽管relu也有缺点，但是仍然是目前使用最多的激活函数

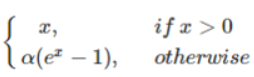
**leakrelu**

leakrelu就是为了解决relu的0区间带来的影响，其数学表达为：leakrelu=max(k∗x,x)leakrelu=max(k\*x,x)leakrelu=max(k∗x,x)其中k是leak系数，一般选择0.01或者0.02，或者通过学习而来

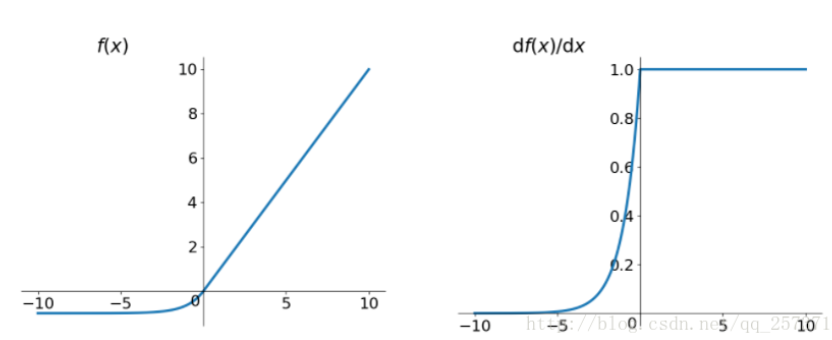


leakrelu解决了0区间带来的影响，而且包含了relu的所有优点

**elu**  
 elu激活函数也是为了解决relu的0区间带来的影响，其数学表达为：



其函数及其导数数学形式为：



但是elu相对于leakrelu来说，计算要更耗时间一些

1. 解决方案4 batchnorm

Batchnorm是深度学习发展以来提出的最重要的成果之一了，目前已经被广泛的应用到了各大网络中，具有加速网络收敛速度，提升训练稳定性的效果，Batchnorm本质上是解决反向传播过程中的梯度问题。batchnorm全名是batch normalization，简称BN，即批规范化，通过规范化操作将输出信号x规范化保证网络的稳定性。

具体的batchnorm原理非常复杂，在这里不做详细展开，此部分大概讲一下batchnorm解决梯度的问题上。具体来说就是反向传播中，经过每一层的梯度会乘以该层的权重，举个简单例子：

正向传播中 那么反向传播中，，反向传播式子中有w的存在，所以w的大小影响了梯度的消失和爆炸，batchnorm就是通过对每一层的输出规范为均值和方差一致的方法，消除了w带来的放大缩小的影响，进而解决梯度消失和爆炸的问题，或者可以理解为BN将输出从饱和区拉倒了非饱和区。

1. 解决方案5-残差结构
2. 解决方案6-LSTM

LSTM全称是长短期记忆网络（long-short term memory networks），是不那么容易发生梯度消失的，主要原因在于LSTM内部复杂的“门”(gates)，如下图，LSTM通过它内部的“门”可以接下来更新的时候“记住”前几次训练的”残留记忆“，因此，经常用于生成文本中。目前也有基于CNN的LSTM，感兴趣的可以尝试一下。

原文链接：<https://blog.csdn.net/qq_25737169/article/details/78847691>

## 激活函数

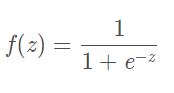
**为什么使用激活函数？**

如果不用激励函数（其实相当于激励函数是f(x) = x），在这种情况下你每一层节点的输入都是上层输出的线性函数，很容易验证，无论你神经网络有多少层，输出都是输入的线性组合，与没有隐藏层效果相当，这种情况就是最原始的感知机（Perceptron）了，那么网络的逼近能力就相当有限。正因为上面的原因，我们决定引入非线性函数作为激励函数，这样深层神经网络表达能力就更加强大（不再是输入的线性组合，而是几乎可以逼近任意函数）。

<https://blog.csdn.net/tyhj_sf/article/details/79932893>

### sigmoid函数

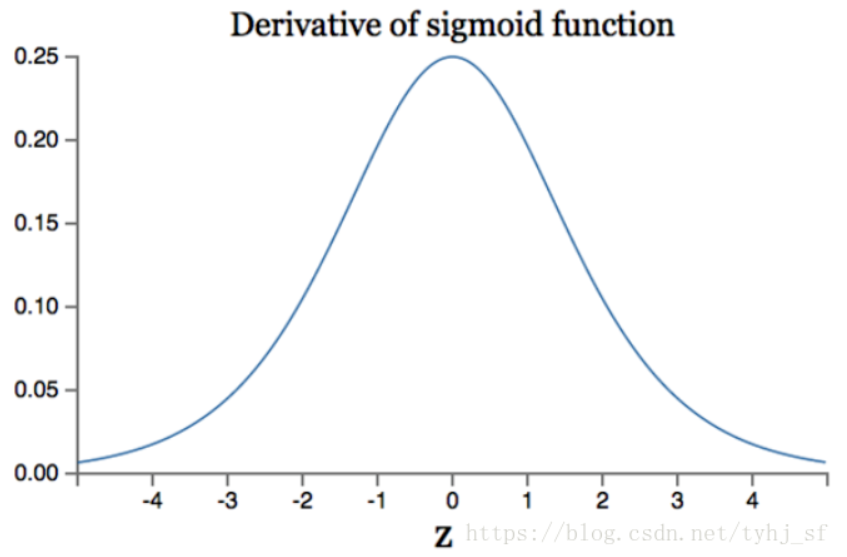
Sigmoid 是常用的非线性的激活函数，它的数学形式如下：



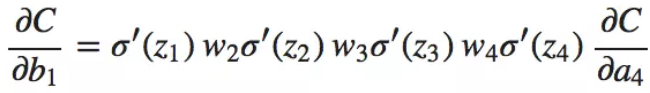


**特点：**  
它能够把输入的连续实值变换为0和1之间的输出，特别的，如果是非常大的负数，那么输出就是0；如果是非常大的正数，输出就是1

**缺点：**  
sigmoid函数曾经被使用的很多，不过近年来，用它的人越来越少了。主要是因为它固有的一些缺点。  
**缺点1：**在深度神经网络中梯度反向传递时导致梯度爆炸和梯度消失，其中梯度爆炸发生的概率非常小，而梯度消失发生的概率比较大。首先来看Sigmoid函数的导数，如下图所示：



如果我们初始化神经网络的权值为 [0,1][0,1][0,1] 之间的随机值，由反向传播算法的数学推导可知，梯度从后向前传播时，每传递一层梯度值都会减小为原来的0.25倍，如果神经网络隐层特别多，那么梯度在穿过多层后将变得非常小接近于0，即出现梯度消失现象；当网络权值初始化为 (1,+∞)(1,+∞)(1,+∞) 区间内的值，则会出现梯度爆炸情况。



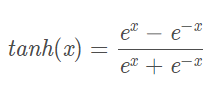
sigmoid的梯度最大是0.25所以每层的梯度值都会至少减少为原来的0.25倍，w也在0到1之间。故会出现梯度消失的问题

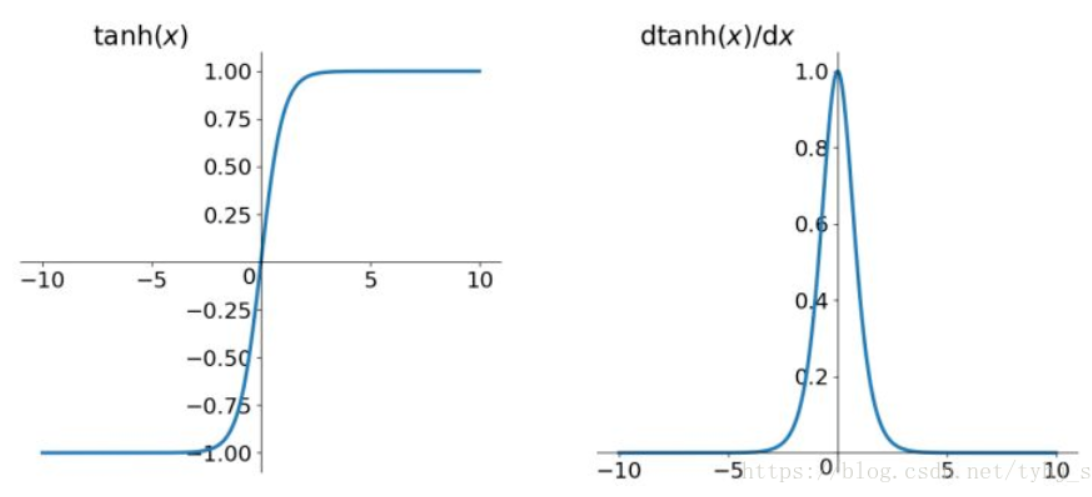
**缺点2：**Sigmoid 的 output 不是0均值（即zero-centered）。这是不可取的，因为这会导致后一层的神经元将得到上一层输出的非0均值的信号作为输入。 产生的一个结果就是：那么对w求局部梯度则都为正，这样在反向传播的过程中w要么都往正方向更新，要么都往负方向更新，导致有一种捆绑的效果，使得收敛缓慢。 当然了，如果按batch去训练，那么那个batch可能得到不同的信号，所以这个问题还是可以缓解一下的。因此，非0均值这个问题虽然会产生一些不好的影响，不过跟上面提到的梯度消失问题相比还是要好很多的。

**缺点3：**其解析式中含有幂运算，计算机求解时相对来讲比较耗时。对于规模比较大的深度网络，这会较大地增加训练时间。

### tanh函数

tanh函数解析式：



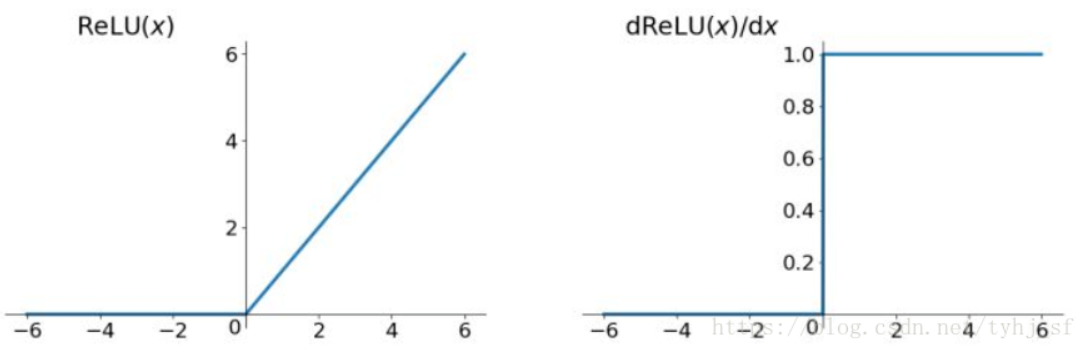


tanh读作Hyperbolic Tangent，它解决了Sigmoid函数的不是zero-centered输出问题，然而，梯度消失（gradient vanishing）的问题和幂运算的问题仍然存在。

### Relu函数

Relu函数的解析式：





ReLU函数其实就是一个取最大值函数，注意这并不是全区间可导的，但是我们可以取sub-gradient，如上图所示。ReLU虽然简单，但却是近几年的重要成果，有以下**几大优点**：

1） 解决了gradient vanishing问题 (在正区间)

2）计算速度非常快，只需要判断输入是否大于0

3）收敛速度远快于sigmoid和tanh

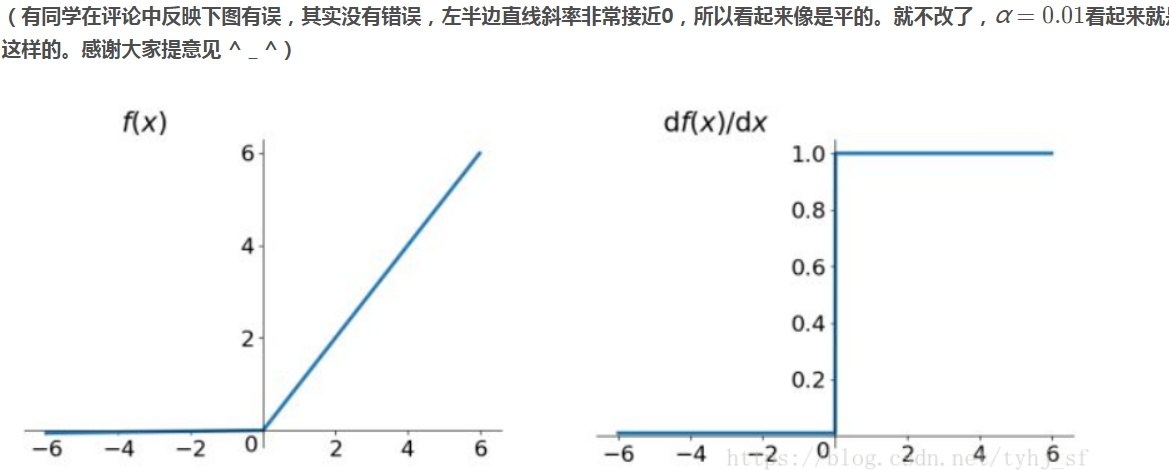
ReLU也有几个需要特别注意的问题：

1）**ReLU的输出不是zero-centered**

2）Dead ReLU Problem，指的是某些神经元可能永远不会被激活，导致相应的参数永远不能被更新。有两个主要原因可能导致这种情况产生: (1) 非常不幸的参数初始化，这种情况比较少见 (2) learning rate太高导致在训练过程中参数更新太大，不幸使网络进入这种状态。解决方法是可以采用Xavier初始化方法，以及避免将learning rate设置太大或使用adagrad等自动调节learning rate的算法。

尽管存在这两个问题，ReLU目前仍是最常用的activation function，在搭建人工神经网络的时候推荐优先尝试！

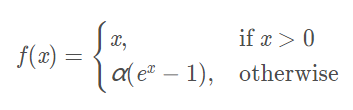
### Leaky Relu函数（PRelu）

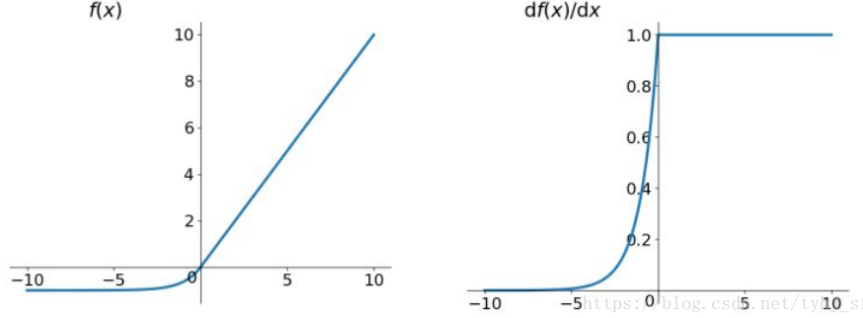


人们为了解决Dead ReLU Problem，提出了将ReLU的前半段设为αx\alpha xαx而非0，通常α=0.01\alpha=0.01α=0.01。另外一种直观的想法是基于参数的方法，即ParametricReLU:f(x)=max(αx,x)Parametric ReLU:f(x) = \max(\alpha x, x)ParametricReLU:f(x)=max(αx,x)，其中α\alphaα

可由方向传播算法学出来。理论上来讲，Leaky ReLU有ReLU的所有优点，外加不会有Dead ReLU问题，但是在实际操作当中，并没有完全证明Leaky ReLU总是好于ReLU。

### ELU（Exponential Linear Units）函数





ELU也是为解决ReLU存在的问题而提出，显然，ELU有ReLU的基本所有优点，以及：

1、不会有Dead ReLU问题

2、输出的均值接近0，zero-centered

它的一个**小问题在于计算量稍大**。类似于Leaky ReLU，理论上虽然好于ReLU，但在实际使用中目前并没有好的证据ELU总是优于ReLU。

### 应用中如何选择合适的激活函数

这个问题目前没有确定的方法，凭一些经验吧。

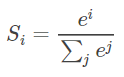
1）深度学习往往需要大量时间来处理大量数据，模型的收敛速度是尤为重要的。所以，总体上来讲，训练深度学习网络尽量使用zero-centered数据 (可以经过数据预处理实现) 和zero-centered输出。所以要尽量选择输出具有zero-centered特点的激活函数以加快模型的收敛速度。

2）如果使用 ReLU，那么一定要小心设置 learning rate，而且要注意不要让网络出现很多 “dead” 神经元，如果这个问题不好解决，那么可以试试 Leaky ReLU、PReLU 或者 Maxout.

3）最好不要用 sigmoid，你可以试试 tanh，不过可以预期它的效果会比不上 ReLU 和 Maxout.

### softmax函数

softmax的定义：



softmax求导：

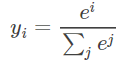
前面提到，在多分类问题中，我们经常使用交叉熵作为损失函数：



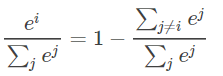
其中，ti表示真实值，yi表示求出的softmax值。当预测第i个时，可以认为ti=1。此时损失函数变成了:



接下来对Loss求导。根据定义：



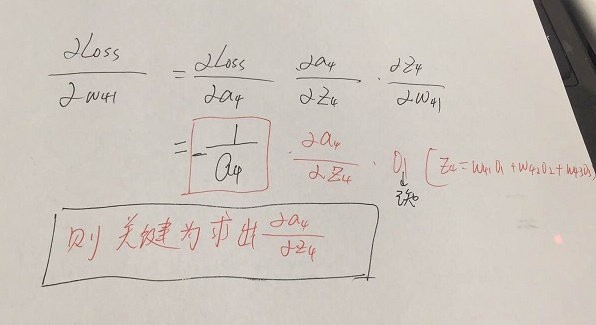
我们已经将数值映射到了0-1之间，并且和为1，则有：



接下来开始求导：

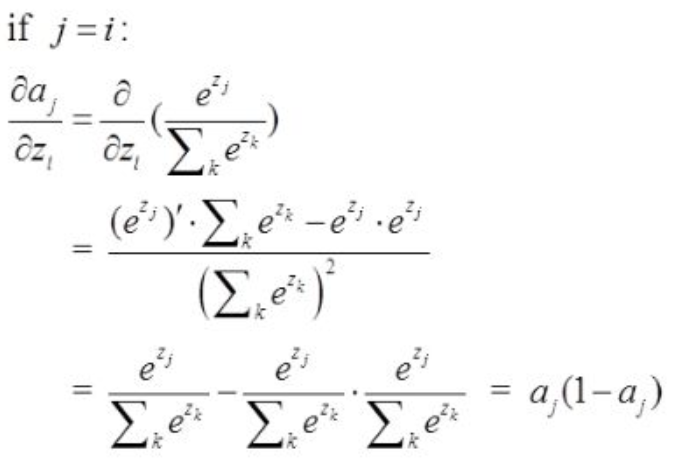


上面的结果表示，我们只需要正想求出yi，将结果减1就是反向更新的梯度，导数的计算是不是非常简单！



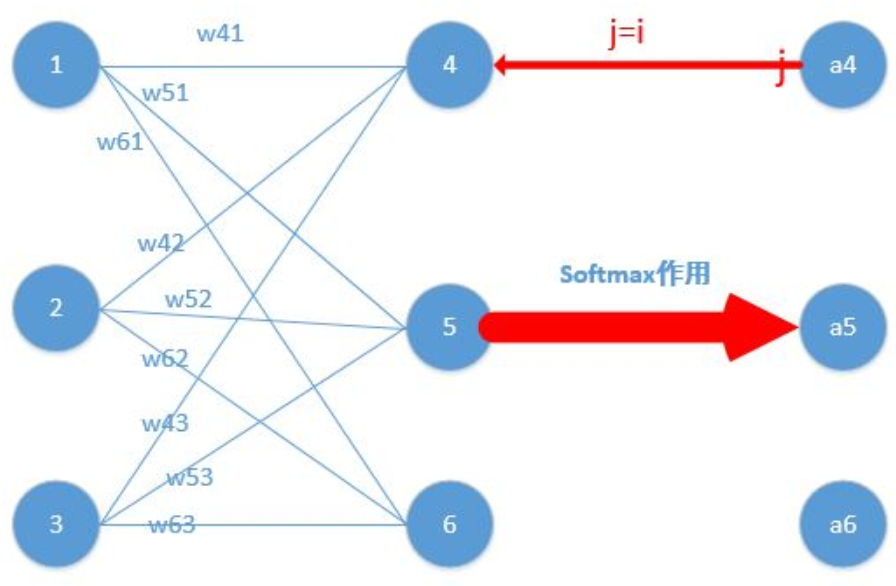
w51.....w63等参数的偏导同理可以求出，那么我们的关键就在于Loss函数对于结点4,5,6的偏导怎么求，如下：

这里分为俩种情况：



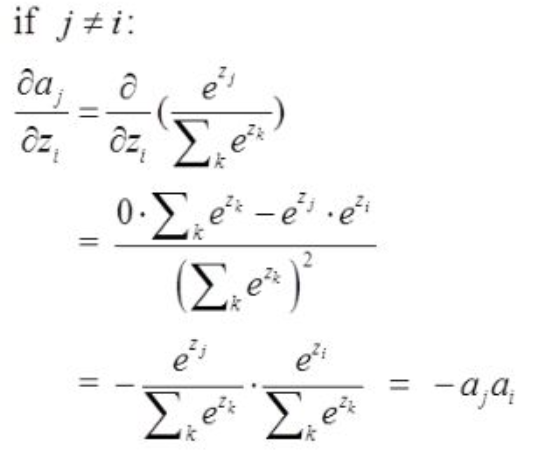
j=i对应例子里就是如下图所示：

比如我选定了j为4，那么就是说我现在求导传到4结点这！

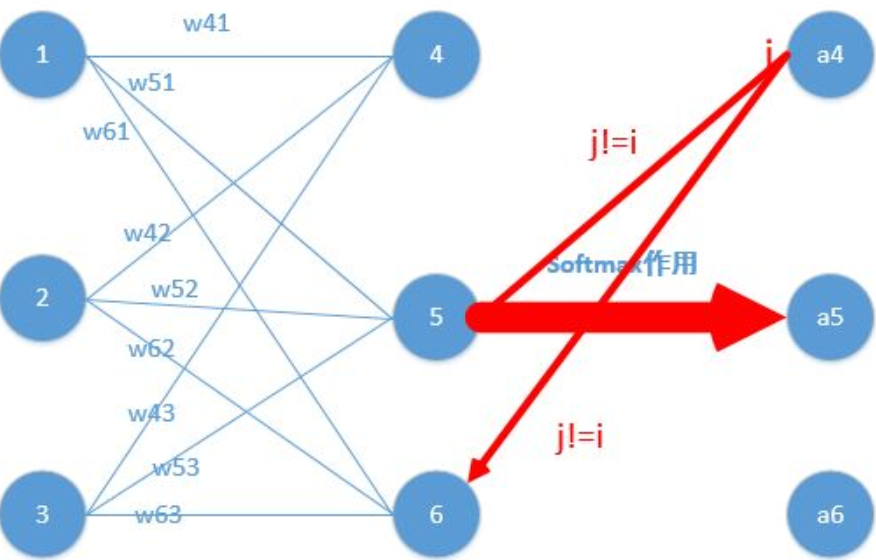


那么由上面求导结果再乘以交叉熵损失函数求导，它的导数为，与上面相乘为 （形式非常简单，这说明我只要正向求一次得出结果，然后反向传梯度的时候，只需要将它结果减1即可，后面还会举例子！）那么我们可以得到Loss对于4结点的偏导就求出了了（**这里假定4是我们的预计输出**）

第二种情况为：



这里对应我的例子图如下，我这时对的是j不等于i，往前传：



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

## 正则化

### L1和L2正则化

使用上面这些目标函数时，有时会发生过拟合。此时常常对模型的参数做一定的限制，使得模型偏好更简单的参数，这就叫“**正则化**”（regularization）。  
最常见的正则化方法，就是（软性地）限制参数的大小。  
设目标函数是要最小化的，所有参数组成向量w。  
如果往目标函数上加上 ，这就是L1正则化；

如果往目标函数上加上 ，这就是L2正则化；

L1正则化的优点是优化后的参数向量往往比较稀疏；L2正则化的优点是其正则化项处处可导。

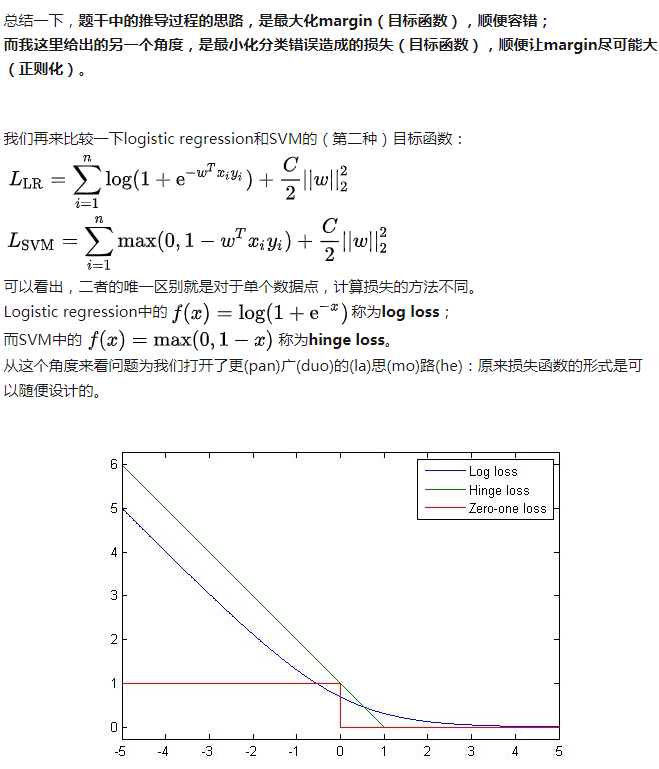
正则化为什么会起作用？

直观上理解就是如果正则化λ设置得足够大，权重矩阵W被设置为接近于 0 的值，直观理解就是把多隐藏单元的权重设为 0，于是基本上消除了这些隐藏单元的许多影响。如果是这种情况，这个被大大简化了的神经网络会变成一个很小的网络，小到如同一个逻辑回归单元，可是深度却很大，它会使这个网络从过度拟合的状态更接近左图的高偏差状态。

L1不可导的时候该怎么办？

当损失函数不可导，梯度下降不再有效，可以使用坐标轴下降法，梯度下降是沿着当前点的负梯度方向进行参数更新，而坐标轴下降法是沿着坐标轴的方向，假设有m个特征个数，坐标轴下降法进行参数更新的时候，先固定m-1个值，然后再求另外一个的局部最优解，从而避免损失函数不可导问题。

### dropout



## 优化算法

## 超参数调试、BN

### 超参数调试

### BN

1. **BN 原理本质思想**

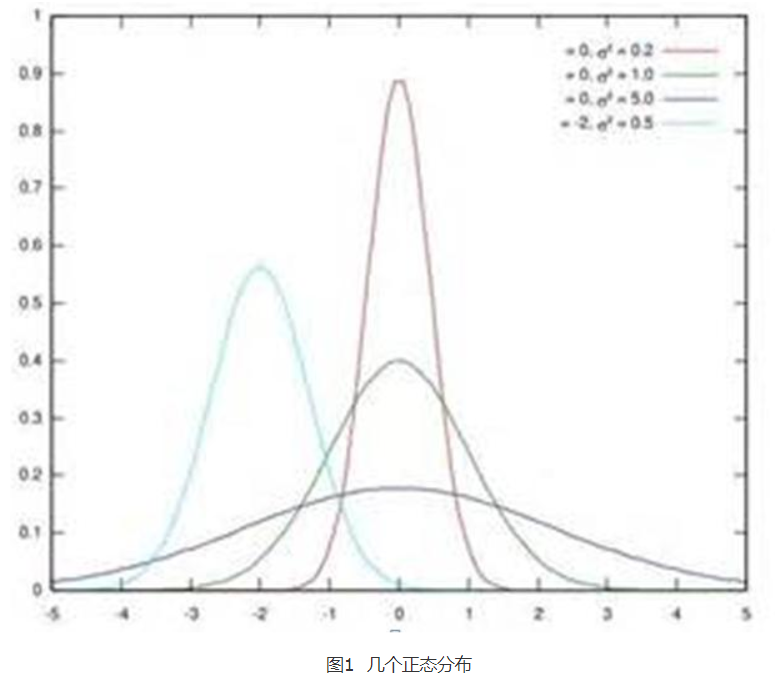
BatchNorm的基本思想：能不能**让每个隐层节点的激活输入分布固定下来呢**？

BN的基本思想其实相当直观：因为深层神经网络在做非线性变换前的激活输入值（就是那个x=WU+B，U是输入）随着网络深度加深或者在训练过程中，其分布逐渐发生偏移或者变动，之所以训练收敛慢，一般是整体分布逐渐往非线性函数的取值区间的上下限两端靠近（对于Sigmoid函数来说，意味着激活输入值WU+B是大的负值或正值），所以这导致反向传播时低层神经网络的梯度消失，这是训练深层神经网络收敛越来越慢的本质原因，而BN就是通过一定的规范化手段，把每层神经网络任意神经元这个输入值的分布强行拉回到均值为0方差为1的标准正态分布，其实就是把越来越偏的分布强制拉回比较标准的分布，这样使得激活输入值落在非线性函数对输入比较敏感的区域，这样输入的小变化就会导致损失函数较大的变化，意思是这样让梯度变大，避免梯度消失问题产生，而且梯度变大意味着学习收敛速度快，能大大加快训练速度。

其实一句话就是：对于每个隐层神经元，把逐渐向非线性函数映射后向取值区间极限

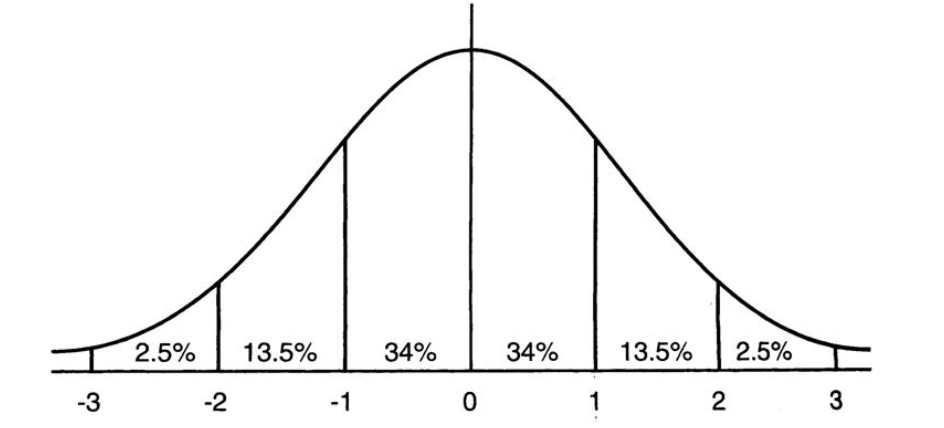
和区靠拢的输入分布强制拉回到均值为0方差为1的比较标准的正态分布，使得非线性变换函数的输入值落入对输入比较敏感的区域，以此避免梯度消失问题。因为梯度一直都能保持比较大的状态，所以很明显对神经网络的参数调整效率比较高，就是变动大，就是说向损失函数最优值迈动的步子大，也就是说收敛地快。BN说到底就是这么个机制，方法很简单，道理很深刻。

　　上面说得还是显得抽象，下面更形象地表达下这种调整到底代表什么含义。

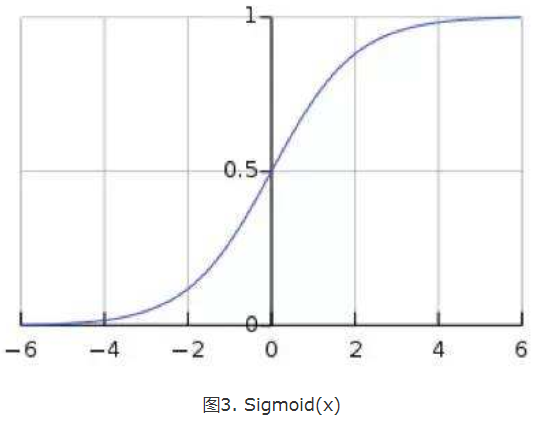


假设某个隐层神经元原先的激活输入x取值符合正态分布，正态分布均值是-2，方差是0.5，对应上图中最左端的浅蓝色曲线，通过BN后转换为均值为0，方差是1的正态分布（对应上图中的深蓝色图形），意味着什么，意味着输入x的取值正态分布整体右移2（均值的变化），图形曲线更平缓了（方差增大的变化）。这个图的意思是，BN其实就是把每个隐层神经元的激活输入分布从偏离均值为0方差为1的正态分布通过平移均值压缩或者扩大曲线尖锐程度，调整为均值为0方差为1的正态分布。

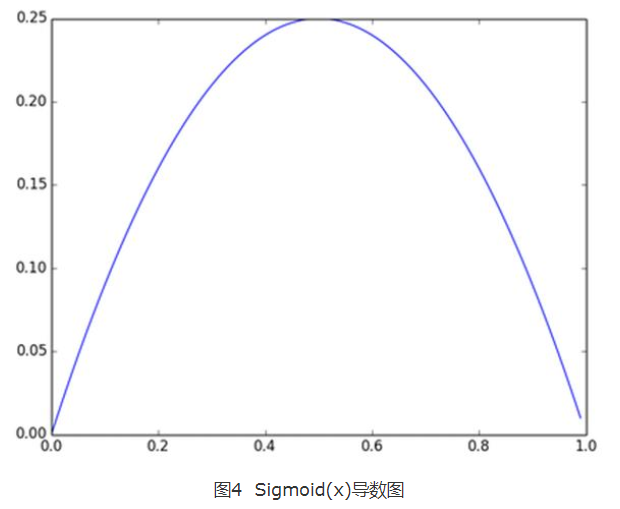
　　那么把激活输入x调整到这个正态分布有什么用？首先我们看下均值为0，方差为1的标准正态分布代表什么含义：



这意味着在一个标准差范围内，也就是说64%的概率x其值落在[-1,1]的范围内，在两个标准差范围内，也就是说95%的概率x其值落在了[-2,2]的范围内。那么这又意味着什么？我们知道，激活值x=WU+B,U是真正的输入，x是某个神经元的激活值，假设非线性函数是sigmoid，那么看下sigmoid(x)其图形：



及sigmoid(x)的导数为：G’=f(x)\*(1-f(x))，因为f(x)=sigmoid(x)在0到1之间，所以G’在0到0.25之间，其对应的图如下：



假设没有经过BN调整前x的原先正态分布均值是-6，方差是1，那么意味着95%的值落在了[-8,-4]之间，那么对应的Sigmoid（x）函数的值明显接近于0，这是典型的梯度饱和区，在这个区域里梯度变化很慢，为什么是梯度饱和区？请看下sigmoid(x)如果取值接近0或者接近于1的时候对应导数函数取值，接近于0，意味着梯度变化很小甚至消失。而假设经过BN后，均值是0，方差是1，那么意味着95%的x值落在了[-2,2]区间内，很明显这一段是sigmoid(x)函数接近于线性变换的区域，意味着x的小变化会导致非线性函数值较大的变化，也即是梯度变化较大，对应导数函数图中明显大于0的区域，就是梯度非饱和区。

　　从上面几个图应该看出来BN在干什么了吧？其实就是把隐层神经元激活输入x=WU+B从变化不拘一格的正态分布通过BN操作拉回到了均值为0，方差为1的正态分布，即原始正态分布中心左移或者右移到以0为均值，拉伸或者缩减形态形成以1为方差的图形。什么意思？就是说**经过BN后，目前大部分Activation的值落入非线性函数的线性区内，其对应的导数远离导数饱和区，这样来加速训练收敛过程。**

　　但是很明显，看到这里，稍微了解神经网络的读者一般会提出一个疑问：如果都通过BN，那么不就跟把非线性函数替换成线性函数效果相同了？这意味着什么？我们知道，如果是多层的线性函数变换其实这个深层是没有意义的，因为多层线性网络跟一层线性网络是等价的。这意味着网络的**表达能力**下降了，这也意味着深度的意义就没有了。**所以BN为了保证非线性的获得，对变换后的满足均值为0方差为1的x又进行了scale加上shift操作(y=scale\*x+shift)**，每个神经元增加了两个参数scale和shift参数，这两个参数是通过训练学习到的，意思是通过scale和shift把这个值从标准正态分布左移或者右移一点并长胖一点或者变瘦一点，每个实例挪动的程度不一样，这样等价于非线性函数的值从正中心周围的线性区往非线性区动了动。核心思想应该是想找到一个线性和非线性的较好平衡点，既能享受非线性的较强表达能力的好处，又避免太靠非线性区两头使得网络收敛速度太慢。当然，这是我的理解，论文作者并未明确这样说。但是很明显这里的scale和shift操作是会有争议的，因为按照论文作者论文里写的理想状态，就会又通过scale和shift操作把变换后的x调整回未变换的状态，那不是饶了一圈又绕回去原始的“Internal Covariate Shift”问题里去了吗，感觉论文作者并未能够清楚地解释scale和shift操作的理论原因。

**2、BN出现的背景**

“Internal Covariate Shift”问题（内部协变量变换）

从论文名字可以看出，BN是用来解决“Internal Covariate Shift”问题的，那么首先得理解什么是“Internal Covariate Shift”？

论文首先说明Mini-Batch SGD相对于One Example SGD的两个优势：梯度更新方向更准确；并行计算速度快；（为什么要说这些？因为BatchNorm是基于Mini-Batch SGD的，所以先夸下Mini-Batch SGD，当然也是大实话）；然后吐槽下SGD训练的缺点：超参数调起来很麻烦。（作者隐含意思是用BN就能解决很多SGD的缺点）

接着引入**covariate shift的概念**：**如果ML系统实例集合<X,Y>中的输入值X的分布老是变，这不符合IID假设**，网络模型很难**稳定的学规律**，这不得引入迁移学习才能搞定吗，我们的ML系统还得去学习怎么迎合这种分布变化啊。对于深度学习这种包含很多隐层的网络结构，在训练过程中，因为各层参数不停在变化，所以每个隐层都会面临covariate shift的问题，也就是**在训练过程中，隐层的输入分布老是变来变去，这就是所谓的“Internal Covariate Shift”，Internal指的是深层网络的隐层，是发生在网络内部的事情，而不是covariate shift问题只发生在输入层。**

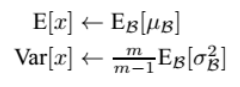
然后提出了BatchNorm的基本思想：能不能**让每个隐层节点的激活输入分布固定下来呢**？这样就避免了“Internal Covariate Shift”问题了。

BN不是凭空拍脑袋拍出来的好点子，它是有启发来源的：之前的研究表明如果在图像处理中对输入图像进行白化（Whiten）操作的话——所谓**白化**，**就是对输入数据分布变换到0均值，单位方差的正态分布**——那么神经网络会较快收敛，那么BN作者就开始推论了：图像是深度神经网络的输入层，做白化能加快收敛，那么其实对于深度网络来说，其中某个隐层的神经元是下一层的输入，意思是其实深度神经网络的每一个隐层都是输入层，不过是相对下一层来说而已，那么能不能对每个隐层都做白化呢？这就是启发BN产生的原初想法，而BN也确实就是这么做的，**可以理解为对深层神经网络每个隐层神经元的激活值做简化版本的白化操作。**

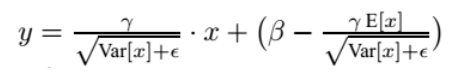
1. BN作用
2. 加快训练速度
3. 可以使用较大学习率
4. 有正则化作用（各个层之间相互独立，有类似于dropout的作用）
5. BN使得模型对网络中的参数不那么敏感，简化调参过程，使得网络学习更加稳定
6. BN的求解过程
7. 对每个minibatch进行计算均值和方差，这里要尽量保证minibatch之间的分布是相同的（可采用交叉采样方法来保证），要不各个minibatch之间的均值和方差会相差很大，网络还得学习这种分布。。。。，minibatch的参数（均值和方差）也是需要学习的，需要使用梯度下降来求解梯度
8. 然后对输入归一化

4、BN的测试过程

可能学完了上面的算法，你只是知道它的一个训练过程，一个网络一旦训练完了，就没有了min-batch这个概念了。测试阶段我们一般只输入一个测试样本，看看结果而已。因此测试样本，前向传导的时候，上面的均值u、标准差σ 要哪里来？其实网络一旦训练完毕，参数都是固定的，这个时候即使是每批训练样本进入网络，那么BN层计算的均值u、和标准差都是固定不变的。我们可以采用这些数值来作为测试样本所需要的均值、标准差，于是最后测试阶段的u和σ 计算公式如下：



上面简单理解就是：对于均值来说直接计算所有batch u值的平均值；然后对于标准偏差采用每个batch σB的无偏估计。最后测试阶段，BN的使用公式就是：



(2)根据文献说，BN可以应用于一个神经网络的任何神经元上。文献主要是把BN变换，置于网络激活函数层的前面。在没有采用BN的时候，激活函数层是这样的：

z=g(Wu+b)

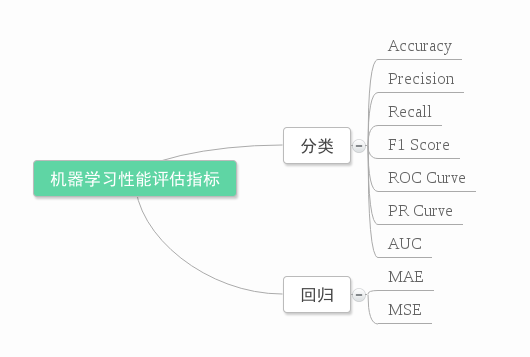
也就是我们希望一个激活函数，比如s型函数s(x)的自变量x是经过BN处理后的结果。因此前向传导的计算公式就应该是：

z=g(BN(Wu+b))

其实因为偏置参数b经过BN层后其实是没有用的，最后也会被均值归一化，当然BN层后面还有个β参数作为偏置项，所以b这个参数就可以不用了。因此最后把BN层+激活函数层就变成了：

z=g(BN(Wu))

# 评价指标

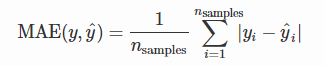


## 回归评估

回归是对连续的实数值进行预测，即输出值是连续的实数值，而分类中是离散值

### MAE

平均绝对误差（Mean Absolute Error，MAE）又被称为L1范数损失（l1-norm loss）



均方误差是指参数估计值与参数真值之差平方的期望值; MSE可以评价数据的变化程度，MSE的值越小，说明预测模型描述实验数据具有更好的精确度。

**通常用来做回归问题的代价函数**

### MSE

均方误差（Mean Squared Error，MSE）又被称为L2范数损失（l2-norm loss）

http://dingby.site/images/mse.png

平均绝对误差是绝对误差的平均值，平均绝对误差能更好地反映预测值误差的实际情况。

**通常用来作为回归算法的性能指标**。

## 二分类评估



### 准确率（accuaracy）

**计算公式：Accuracy** = (TP+TN)/(TP+TN+FP+FN)

在正负样本不平衡的情况下，准确率这个评价指标有很大的缺陷。比如在互联网广告里面，点击的数量是很少的，一般只有千分之几，如果用Accuracy，即使全部预测成负类（不点击）Accuracy也有99%以上，没有意义

### 查准率（精确率）

**定义**：正确分类的正例个数占（**预测**）分类为正例的实例个数的比例，也称**查准率**。

**计算公式**：TP/(TP+FP)

### 查全率（召回率）

**定义**：正确分类的正例个数占**实际**正例个数的比例 也称**查全率**。

**计算公式**：TP/(TP+FN)

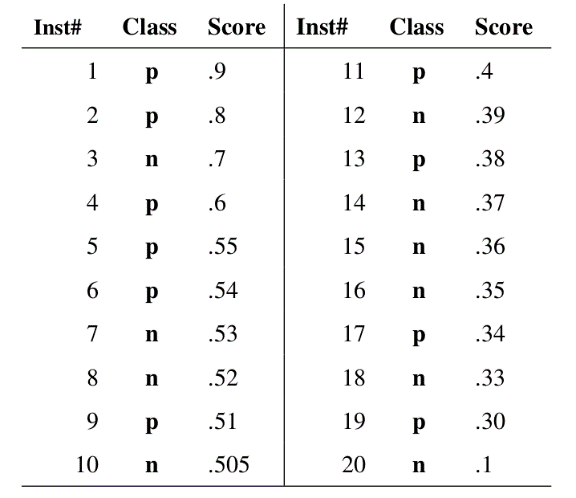
查准率与查全率是一对矛盾的度量。一般来说，查准率高时，查全率往往偏低；而查全率高时，查准率往往偏低。

### F1度量

根据学习器的预测结果对样例进行排序，排在最前面的是学习器认为“最可能”是正例的样本，排在最后的则是学习器认为“最不可能”是正例的样本。按此顺序逐个把样本作为正例进行预测，则每次可以计算出当前的查全率、查准率。以查准率为纵轴、查全率为横轴做图，就得到了查准率-查全率曲线，简称“P-R曲线”。

P-R的绘制方法和ROC基本一致，只是横轴和纵轴不一致。

在机器学习中分类器往往输出的不是类别标号，而是属于某个类别的概率值，根据分类器的预测结果从大到小对样例进行排序，逐个把样例加入正例进行预测，算出此时的P、R值。



如上图：

真实情况正例反例各有10个。

先用分数（score）：0.9作为阈值（大于等于1为正例，小于1为反例），此时TP=1，FP=0，FN=9，故P=1，R=0.1。（也就是刚开始极限点在（0,1））

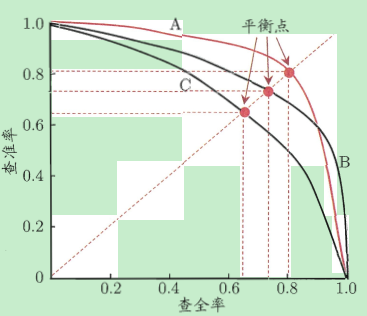
用0.8作为阈值，P=1，R=0.2。

用0.7作为阈值，P=0.67，R=0.2。

用0.6作为阈值，P=0.75，R=0.3。

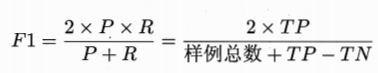
以此类推。。。

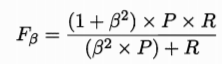
故其P-R曲线绘制如下：



**P-R曲线越靠近右上角越好**

“平衡点”（Break-Event Point，简称BEP）就是这样一个度量，它是“查准率=查全率”时的取值，如上图中的学习器C的BEP是0.64，而基于BEP的比较可认为学习器A优于B。但BEP还是过于简化了些，更常用的是F1度量：

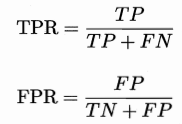




其中，β>0度量了查全率对查准率的相对重要性，β= 1时退化为标准的F1；β> 1时查全率有更大影响；β< 1时查准率有更大影响。

### ROC与AUC

很多学习器是为测试样本产生一个实值或概率预测，然后将这个预测值与一个分类阈值进行比较，若大于阈值则分为正类，否则为反例。这个实值或概率预测结果的好坏，直接决定了学习器的泛化能力。实际上，根据这个实值或概率预测结果，我们可将测试样本进行排序，“最可能”是正例的排在最前面，“最不可能”是正例的排在最后面。这样，分类过程就相当于这个排序中以某个“截断点”（cut point）将样本分为两部分，前一部分判作正例，后一部分则判为反例。不同的截断点有不同的TPR，和FPR



以TPR为纵轴，FPR为衡轴绘制ROC曲线如下：

和P-R曲线类似，ROC曲线用FPR（假正例率）作横轴，用TPR（真正例率）作纵轴，其中：

FPR=FPTN+FPFPR=FPTN+FP

TPR=TPTP+FNTPR=TPTP+FN

同样用上面的数据，

用0.9作为阈值，此时TP=1，FP=0，FN=9，TN=10，故TPR=0.1，FPR=0。

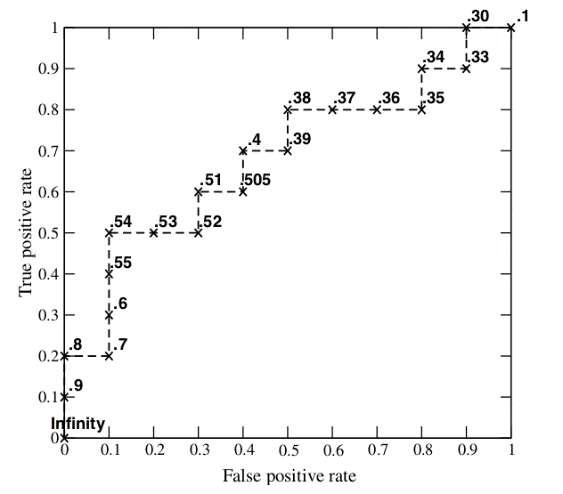
用0.8作为阈值，此时TP=2，FP=0，FN=8，TN=10，故TPR=0.2，FPR=0。

用0.7作为阈值，此时TP=2，FP=1，FN=8，TN=9，故TPR=0.2，FPR=0.1。

用0.6作为阈值，此时TP=3，FP=1，FN=7，TN=9，故TPR=0.3，FPR=0.1。

以此类推。。。

最后的ROC曲线如下图：



AUC的值就是ROC曲线下方围成区域的面积大小。ROC越靠近左上角越好。

PR曲线和ROC曲线的关系

PR曲线和ROC曲线都能评价分类器的性能。如果分类器a的PR曲线或ROC曲线包围了分类器b对应的曲线，那么分类器a的性能好于分类器b的性能。

PR曲线和ROC曲线有什么联系和不同：

相同点：

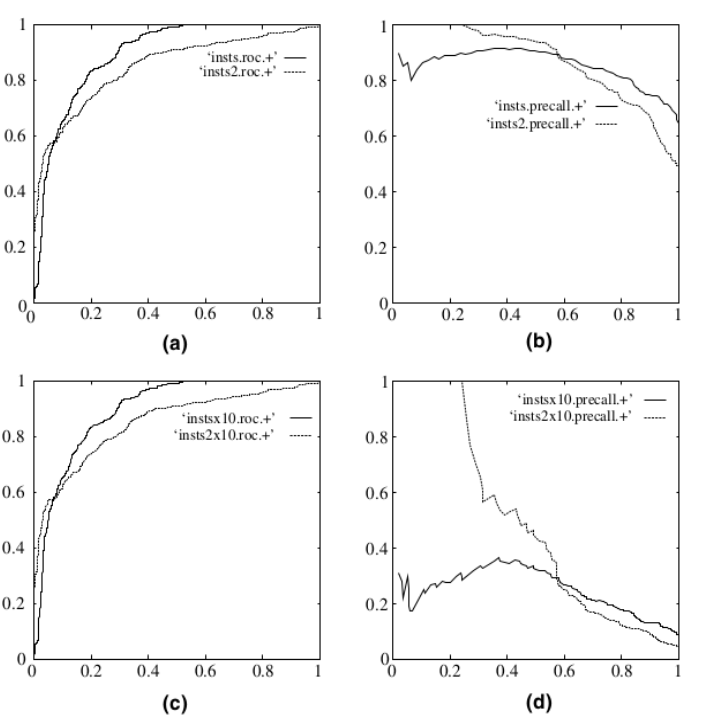
首先从定义上PR曲线的R值是等于ROC曲线中的TPR值。

都是用来评价分类器的性能的。

不同点：

ROC曲线是单调的而PR曲线不是（根据它能更方便调参），可以用AUC的值得大小来评价分类器的好坏（是否可以用PR曲线围成面积大小来评价呢？）。

正负样本的分布失衡的时候，ROC曲线保持不变，而PR曲线会产生很大的变化。

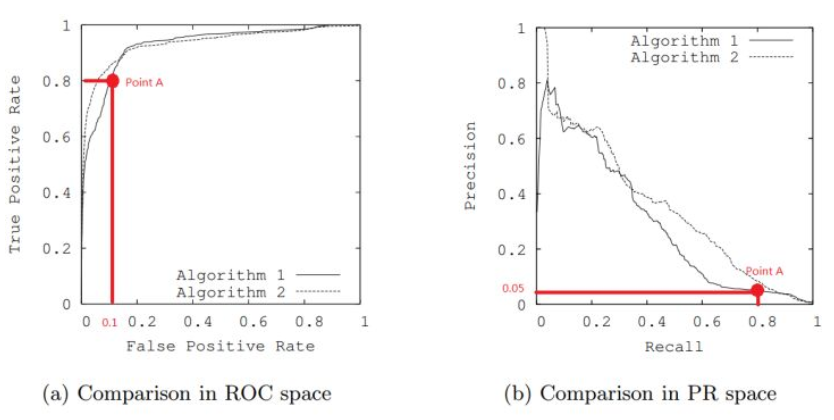


（a）（b）分别是正反例相等的时候的ROC曲线和PR曲线

（c）（d）分别是十倍反例一倍正例的ROC曲线和PR曲线

可以看出，在正负失衡的情况下，从ROC曲线看分类器的表现仍然较好（图c），然而从PR曲线来看，分类器就表现的很差。

事实情况是分类器确实表现的不好，是ROC曲线欺骗了我们。



单从图a看，这两个分类器都接近完美（非常接近左上角）。图b对应着相同分类器的PR space。而从图b可以看出，这两个分类器仍有巨大的提升空间。

原因是什么呢？通过看点A，可以得出一些结论。首先图a和b中的点A是相同的点，只是在不同的空间里。因为TPR=Recall=TP/(TP+FN)，也就是，真阳率（TPR）和召回率是同一个东西，只是不同的名字。所以图a中TPR为0.8的点对应着图b中Recall为0.8的点。

假设数据集有100个positive instances。由图a中的点A，可以的到一下结论：

TPR=TP/(TP+FN)=TP/actual positive = TP/100=0.8，所以TP=80.

由图b中的点A，可得：

Precision = TP/(TP+FP)=80/(80+FP)=0.05，所以FP=1520.

再由图a中的A点，可得：

FPR=FP/(FP+TN)=FP/actual negative=1520/actual negatives=0.1，所以actual negatives是15200.

由此，可以得出原数据集中只有100个正例，却有15200个负例！这就是极不均匀的数据集。直观的说，在点A处，分类器将1600（1520+80）个样本分为正例，而其中实际上只有80个是真正的正例.我们凭直觉来看，这个分类器并不好。但由于真正负样本的数量远远大于正样本，ROC的结果却“看上去很美”。所以在这种情况下，PRC更能体现本质。

结论：在negative instances的数量远远大于positive instances的data set里，PRC更能有效衡量分类器的好坏

ROC、AUC曲线python实现：

<http://bei.dreamcykj.com/2018/08/19/ROC%E5%8E%9F%E7%90%86%E4%BB%8B%E7%BB%8D%E5%8F%8A%E5%88%A9%E7%94%A8python%E5%AE%9E%E7%8E%B0%E4%BA%8C%E5%88%86%E7%B1%BB%E5%92%8C%E5%A4%9A%E5%88%86%E7%B1%BB%E7%9A%84ROC%E6%9B%B2%E7%BA%BF%20(1)/>

P-R曲线python实现：

<https://blog.csdn.net/u014568921/article/details/53843311>

# 降维算法

机器学习领域中所谓的降维就是指采用某种映射方法，将原高维空间中的数据点映射到低维度的空间中。降维的本质是学习一个映射函数 f : x->y，其中x是原始数据点的表达，目前最多使用向量表达形式。 y是数据点映射后的低维向量表达，通常y的维度小于x的维度（当然提高维度也是可以的）。f可能是显式的或隐式的、线性的或非线性的。

之所以使用降维后的数据表示是因为在原始的高维空间中，包含有冗余信息以及噪音信息，在实际应用例如图像识别中造成了误差，降低了准确率；而通过降维,我们希望减少 [冗余信息](http://www.hudong.com/wiki/%E5%86%97%E4%BD%99%E4%BF%A1%E6%81%AF) 所造成的误差,提高识别（或其他应用）的精度。又或者希望通过降维算法来寻找数据内部的本质结构特征。

## PCA

在许多领域的研究与应用中，往往需要对反映事物的多个变量进行大量的观测，收集大量数据以便进行分析寻找规律。多变量大样本无疑会为研究和应用提供了丰富的信息，但也在一定程度上增加了数据采集的工作量，更重要的是在多数情况下，许多变量之间可能存在相关性，从而增加了问题分析的复杂性，同时对分析带来不便。如果分别对每个指标进行分析，分析往往是孤立的，而不是综合的。盲目减少指标会损失很多信息，容易产生错误的结论。

因此需要找到一个合理的方法，在减少需要分析的指标同时，尽量减少原指标包含信息的损失，以达到对所收集数据进行全面分析的目的。由于各变量间存在一定的相关关系，因此有可能用较少的综合指标分别综合存在于各变量中的各类信息。主成分分析与因子分析就属于这类降维的方法。

## LDA

## LLE

# Tensorflow使用

## 基础知识

### 张量（Tensor）

TensorFlow 内部的计算都是基于张量的，因此我们有必要先对张量有个认识。张量是在我们熟悉的标量、向量之上定义的，详细的定义比较复杂，我们可以先简单的将它理解为一个多维数组



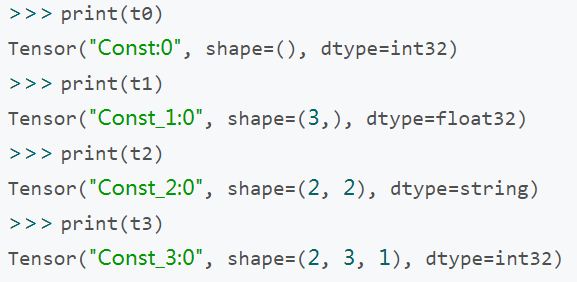
TensorFlow 内部使用tf.Tensor类的实例来表示张量，每个 tf.Tensor有两个属性：

**dtype Tensor** 存储的数据的类型，可以为tf.float32、tf.int32、tf.string…

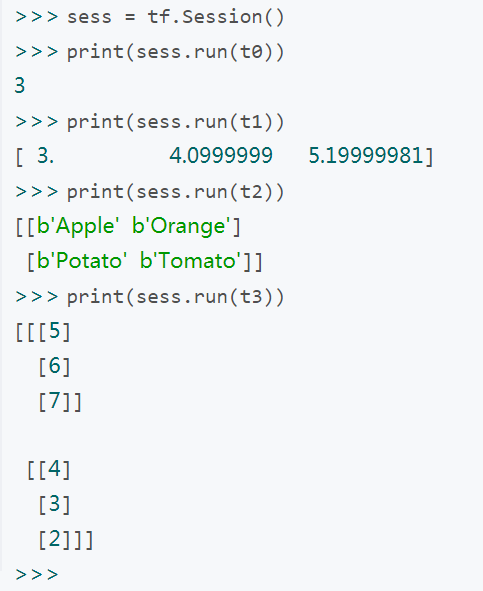
**shape Tensor** 存储的多维数组中每个维度的数组中元素的个数，如上面例子中的shape



上面代码的输出为，注意shape的类型：



print 一个 Tensor 只能打印出它的属性定义，并不能打印出它的值，要想查看一个 Tensor 中的值还需要经过Session 运行一下：



### 数据流图(Dataflow Graph)

数据流是一种常用的并行计算编程模型，数据流图是由节点(nodes)和线(edges)构成的有向图：

**节点(nodes)** 表示计算单元，也可以是输入的起点或者输出的终点

**线(edges)** 表示节点之间的输入/输出关系

在 TensorFlow 中，每个节点都是用 tf.Tensor的实例来表示的，即每个节点的输入、输出都是Tensor，如下图中 Tensor 在 Graph 中的流动，形象的展示 TensorFlow 名字的由来

TensorFlow 中的数据流图有以下几个优点：

**可并行** 计算节点之间有明确的线进行连接，系统可以很容易的判断出哪些计算操作可以并行执行

**可分发** 图中的各个节点可以分布在不同的计算单元(CPU、 GPU、 TPU等)或者不同的机器中，每个节点产生的数据可以通过明确的线发送的下一个节点中

**可优化** TensorFlow 中的 XLA 编译器可以根据数据流图进行代码优化，加快运行速度

**可移植** 数据流图的信息可以不依赖代码进行保存，如使用Python创建的图，经过保存后可以在C++或Java中使用

### Sesssion

我们在Python中需要做一些计算操作时一般会使用NumPy，NumPy在做矩阵操作等复杂的计算的时候会使用其他语言(C/C++)来实现这些计算逻辑，来保证计算的高效性。但是频繁的在多个编程语言间切换也会有一定的耗时，如果只是单机操作这些耗时可能会忽略不计，但是如果在分布式并行计算中，计算操作可能分布在不同的CPU、GPU甚至不同的机器中，这些耗时可能会比较严重。

TensorFlow 底层是使用C++实现，这样可以保证计算效率，并使用 tf.Session类来连接客户端程序与C++运行时。上层的Python、Java等代码用来设计、定义模型，构建的Graph，最后通过tf.Session.run()方法传递给底层执行。

## Bacth Normal

# 用于在指定维度计算均值与方差

tf.nn.moments(

x,

axes,

shift=None, # pylint: disable=unused-argument

name=None,

keep\_dims=False)

**参数：**

x：一个Tensor，可以理解为我们输出的数据，形如 [batchsize, height, width, kernels]。

axes：整数数组，用于指定计算均值和方差的轴。如果x是1-D向量且axes=[0] 那么该函数就是计算整个向量的均值与方差。

shift：未在当前实现中使用。

name：用于计算moment的操作范围的名称。

keep\_dims：产生与输入具有相同维度的moment，通俗点说就是是否保持维度。

**返回**：

Two Tensor objects: mean and variance.

两个Tensor对象：mean和variance.

解释如下：

mean 就是均值

variance 就是方差

# 面试总结

## 平安

1、残差网络结构

2、lstm结构

3、卷积层作用

4、BN层作用

5、tf\_idf公式

6、逻辑回归的损失函数

7、大卷积核小卷积核区别

8、pooling层为什么能保存特征？会不会丢失特征？

9、BN层原理，为什么能防止过拟合

10、dropout

11、过拟合方法

12、调参调了哪些参数

编程：

1、回文检测

2、矩阵旋转90度不增加内存

## OPPO

1、python编程题：完全数、正则表达式

2、LSTM

3、你在项目中主要负责什么？

4、你的成果？

## 中信

### 技术面

1. 马尔科夫链一阶和二阶指的是什么？
2. PCA算法
3. BN作用，里面的超参数是怎么调整的？
4. BN中，每次训练是根据Mini-batch来训练的，怎么计算他的均值和方差
5. MAX POOLing 的反向传播计算梯度是怎么计算的？

### 经理面

1、你的优点是什么？

可从对这份工作的**热爱**、**兴趣**爱好入手

数学基础比较扎实，如果会有新的算法，理解掌握的会比较快

对编码敢兴趣，喜欢写代码，没事会去github上找一些源码，分析，自己也会写一下代码玩一玩

**态度方面**：对交代的任务会一定会在规定时间内完成，如果时间充裕，会完美的完成，比如代码的优化，性能优化等

2、你最近写了多少行代码？

没统计过，因为代码很多，不同模块，写了很多，具体多少行还真没统计过

3、你们组的分工

项目驱动，来了额项目，先大致分裂，就比如说是回归问题还是分类问题，然后每个人会去试试模型，分析结果然后汇总，决定模型

4、你们的部门是哪个？

人工智能开发部

5、职业规划是？

## Speakin

1. 项目里数据问题，怎么清洗的？
2. 编程题，单链表的共同点
3. PCA原理
4. softmax？为什么要用softmax？有什么好处？
5. 混合高斯分布？
6. n-gram，skip-gram？

# 面试题目总结

## 笔试

## 技术

1. 在项目中扮演的角色？运用到什么技术等；

数据清洗、模型训练（可从那一套入手）、调参、模型评估、模型部署（tfserving）

1. 在项目中遇到什么困难，是如何解决的？

从AUC说起，分析AUC，效果不好，怎么换模型，调参数，梯度爆炸消失的分析，等等最后怀疑数据等

1. 对技术 或者未来工作  有什么想法？
2. 技术方向，技术能力，专注某个领域（业务，算法）
3. 管理方向：得需要人脉、兴趣、运气、天赋等

   D.离职原因  （尽可能是企业能接受的）

## 性格

# 牛客面试题刷题

## 简答题

1、SGD,Momentum,Adagard,Adam原理

SGD为随机梯度下降（对一个样本进行梯度更新）,每一次迭代计算数据集的mini-batch的梯度,然后对参数进行跟新。

Momentum参考了物理中动量的概念,前几次的梯度也会参与到当前的计算中,但是前几轮的梯度叠加在当前计算中会有一定的衰减。

Adagard在训练的过程中可以自动变更学习的速率,设置一个全局的学习率,而实际的学习率与以往的参数模和的开方成反比。

Adam利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率,在经过偏置的校正后,每一次迭代后的学习率都有个确定的范围,使得参数较为平稳

<https://blog.csdn.net/u012759136/article/details/52302426>

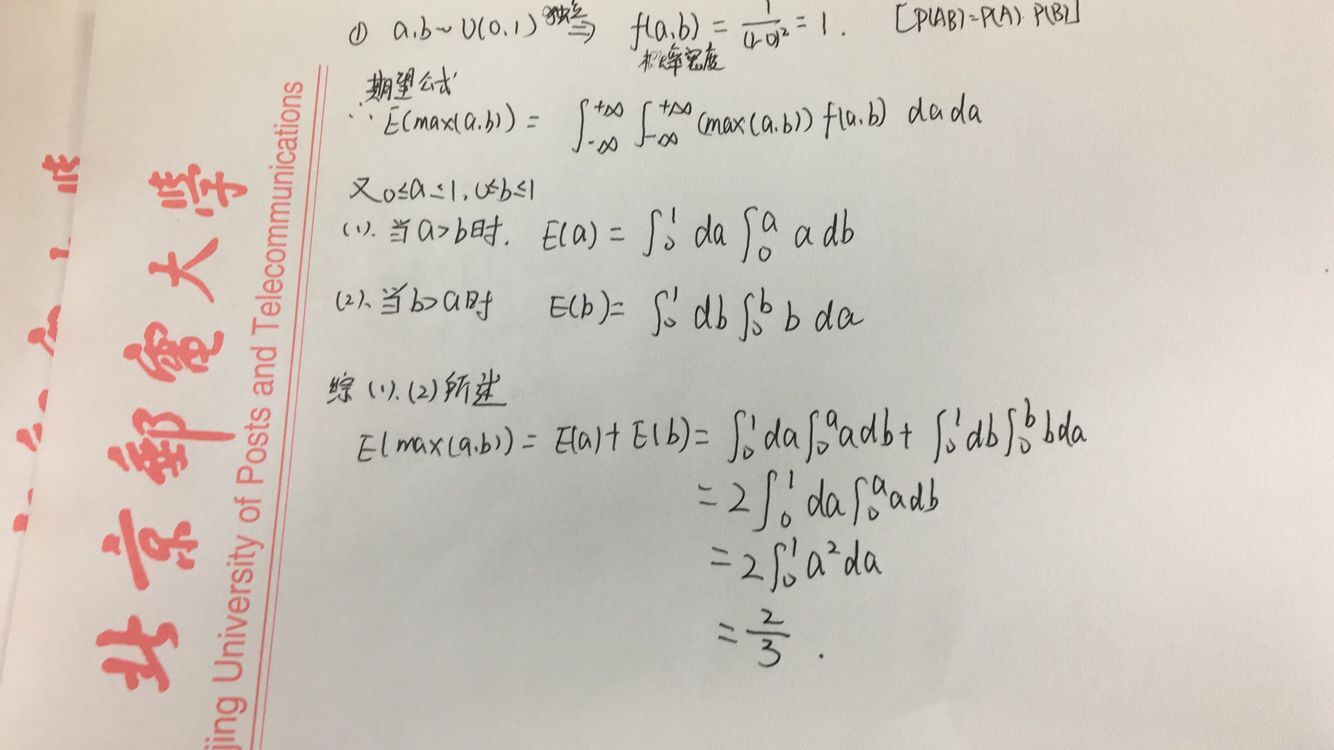
2、L1不可导的时候怎么办？

当损失函数不可导,梯度下降不再有效,可以使用坐标轴下降法,梯度下降是沿着当前点的负梯度方向进行参数更新,而坐标轴下降法是沿着坐标轴的方向,假设有m个特征个数,坐标轴下降法进参数更新的时候,先固定m-1个值,然后再求另外一个的局部最优解,从而避免损失函数不可导问题。使用Proximal Algorithm（近端梯度下降法）对L1进行求解,此方法是去优化损失函数上界结果，这里用到泰勒级数展开式

3、sigmoid函数特性

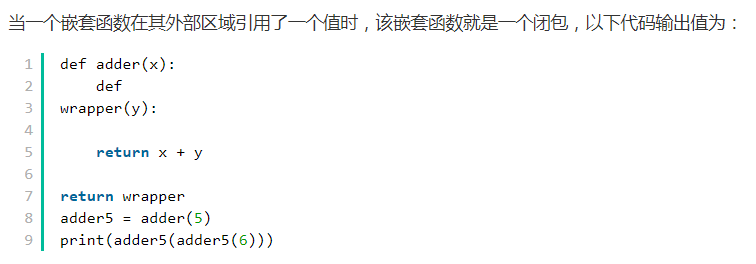
定义域为 值域为(-1,1)函数在定义域内为连续和光滑的函数处处可导,导数为

1. a,b~U[0,1]，互相独立,求Max(a,b)期望（a、b服从均匀分布）



## python题目总结

### 嵌套函数



1、如果在一个函数的内部定义了另一个函数，外部的我们叫他外函数，内部的我们叫他内

函数。

2. 在一个外函数中定义了一个内函数，内函数里运用了外函数的临时变量，并且外函数的返回值是内函数的引用。这样就构成了一个闭包。

3. 一般情况下，如果一个函数结束，函数的内部所有东西都会释放掉，还给内存，局部变量都会消失。但是闭包是一种特殊情况，如果外函数在结束的时候发现有自己的临时变量将来会在内部函数中用到，就把这个临时变量绑定给了内部函数，然后自己再结束。

具体就是5+6+5=16

### 文件读取

read:读取整个文件，内容为字符串

readline：返回下一行，内容为字符串；

readlines:返回list,保存有每行的数据。

### python中复数

1 虚数不能单独存在，它们总是和一个值为 0.0 的实数部分一起来构成一个复数。

2复数由实数部分和虚数部分构成

3表示虚数的语法： real+imagj

4 实数部分和虚数部分都是浮点数

5 虚数部分必须有后缀 j 或 J

6 Python2 与 Python3 均不支持复数比较大小

### 向下取整

print math.floor(5.5) 向下取整

math.ceil(2.3) 向上取整

Python2 支持数字与字符串之间的比较，而 Python3 则不支持，ASCII值得比较

### python scket操作

1. **sk.recv(bufsize[,flag])**:接受套接字的数据。数据以字符串形式返回，bufsize指定最多可以接收的数量。flag提供有关消息的其他信息，通常可以忽略。
2. **recvfrom(bufsize[.flag]):**与recv()类似，但返回值是（data,address）。其中data是包含接收数据的字符串，address是发送数据的套接字地址

3、**getsockname():**返回套接字自己的地址。通常是一个元组(ipaddr,port)

4、**connect(address):**连接到address处的套接字。一般，address的格式为元组（hostname,port）,如果连接出错，返回socket.error错误

5、**listen(backlog):**开始监听传入连接。backlog指定在拒绝连接之前，可以挂起的最大连接数量

### 字典的相关操作

字典的键值必须是不可变类型，如数字，字符串，元组，而列表是可变类型。

字典本身是可变数据类型，字典的键记住两点：唯一：同一个字典中的键必须唯一，如果出现多个相同的键，则最新的键会被记住；不可变：键是不可变数据类型，使用可以是int,string,float,tuple

### isinstance函数，继承

abc isinstance（object，classinfo），用于判断object是否是classinfo的一个实例，或者object是否是classinfo类的子类的一个实例，如果是返回True. issubclass（class，classinfo），用于判断class是否是classinfo类的子类，如果是返回True.

python中继承的相关概念：<https://www.cnblogs.com/bigberg/p/7182741.html>

1. 类的概述

面向对象编程 (OOP) 语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

　　通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”，继承的过程，就是从一般到特殊的过程。在某些 OOP 语言中，一个子类可以继承多个基类。但是一般情况下，一个子类只能有一个基类，要实现多重继承，可以通过多级继承来实现。

　　继承概念的实现方式主要有2类：实现继承、接口继承。

实现继承是指使用基类的属性和方法而无需额外编码的能力。

接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力(子类重构爹类方法)。

　　在考虑使用继承时，有一点需要注意，那就是两个类之间的关系应该是“属于”关系。例如，Employee 是一个人，Manager 也是一个人，因此这两个类都可以继承 Person 类。但是 Leg 类却不能继承 Person 类，因为腿并不是一个人。

　　OO开发范式大致为：划分对象→抽象类→将类组织成为层次化结构(继承和合成) →用类与实例进行设计和实现几个阶段。

1. 类的继承
2. 继承的定义



1. 构造函数的继承

如果我们要给实例 c 传参，我们就要使用到构造函数，那么构造函数该如何继承，同时子类中又如何定义自己的属性？

继承类的构造方法：

        1.经典类的写法： 父类名称.\_\_init\_\_(self,参数1，参数2，...)

        2. 新式类的写法：super(子类，self).\_\_init\_\_(参数1，参数2，....)



如果我们只是简单的在子类Chinese中定义一个构造函数，其实就是在重构。这样子类就不能继承父类的属性了。所以我们在定义子类的构造函数时，要先继承再构造，这样我们也能获取父类的属性了。

      子类构造函数基础父类构造函数过程如下：

      实例化对象c ----> c 调用子类\_\_init\_\_()  ---- > 子类\_\_init\_\_()继承父类\_\_init\_\_()  ----- > 调用父类 \_\_init\_\_()

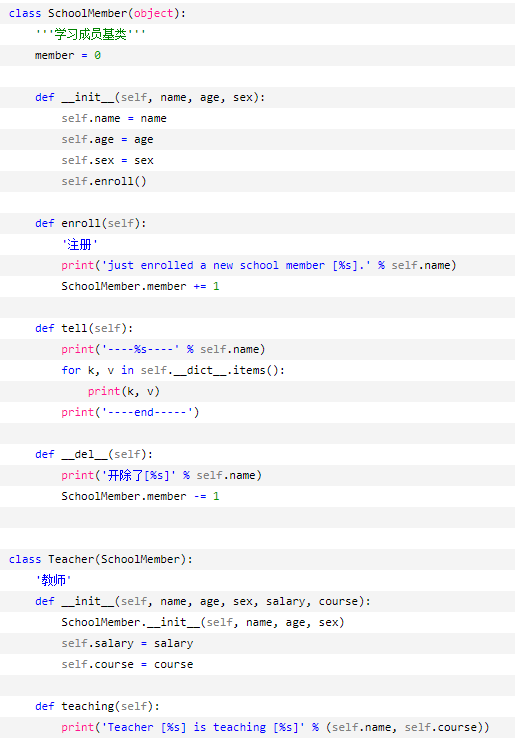


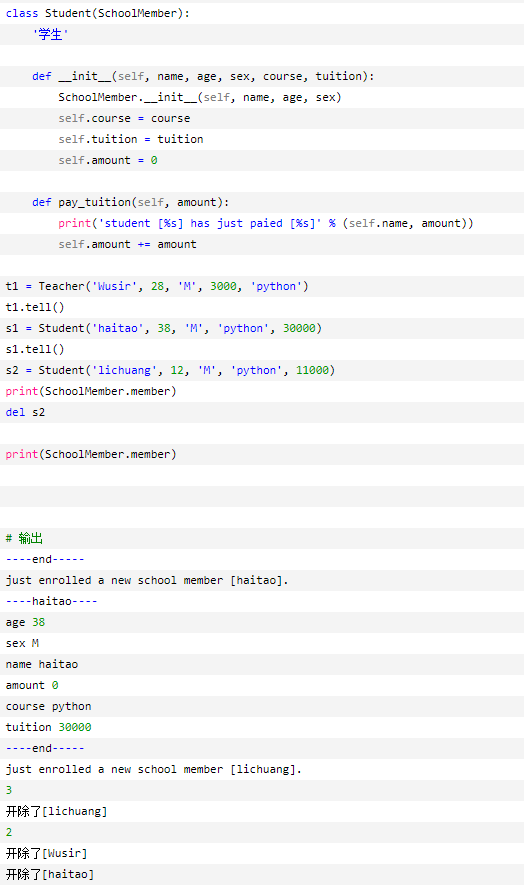
1. 子类对父类

如果我们对基类/父类的方法需要修改，可以在子类中重构该方法。如下的talk()方法



1. 类继承的事例





### python中for循环

for I in range(len(nums))

假设len（nums）为3，则i的值为从0开始到2

## 数据库题目总结

### IFNULL函数的使用

MySQL的isfull 与limit的用法

ifnull(value1,value2)

      1、如果value1不为空，结果返回value1。

      2、如果value1为空，结果返回value2。同上

Limit x 返回多少条数据 limit x,y 从第X+1条开始，返回y条。

### 数据库笛卡尔集

假设集合A={a,b}，集合B={0,1,2}，则两个集合的笛卡尔积为{(a,0),(a,1),(a,2),(b,0),(b,1),(b,2)}。

即，为A中元素个数与B中元素个数的乘积种

### 判断一个数是否为奇数

mod(r.id,2) = 0

### 更新指定列满足指定条件的值

给定一个 salary 表，如下所示，有 m = 男性 和 f = 女性 的值。交换所有的 f 和 m 值（例如，将所有 f 值更改为 m，反之亦然）。要求只使用一个更新（Update）语句，并且没有中间的临时表。

update salary set sex = char(ascii('m') + ascii('f') - ascii(sex));

UPDATE salary

SET

sex = CASE sex

WHEN 'm' THEN 'f'

ELSE 'm'

END;

# 项目相关

图像的项目,80万张照片,每张0.5M,640X360