|  |
| --- |
| ***Hsingmin.lee@yahoo.com*** |
| **机器学习速查** |
| ***Machine Learning Lite*** |

|  |
| --- |
| **黎兴民**  ***2018/3/5 Monday*** |

目录

[1. 机器学习 4](#_Toc511397906)

[1.1 模型评估方法 4](#_Toc511397907)

[1.1.1 数据集划分 4](#_Toc511397908)

[1.1.2 模型的性能度量 4](#_Toc511397909)

[1.1.3 偏差-误差分解 4](#_Toc511397910)

[1.2 LR与极大似然估计 6](#_Toc511397911)

[1.3 LDA 7](#_Toc511397912)

[1.4 类别不平衡问题 8](#_Toc511397913)

[1.5 判别/生成模型与先验/后验概率 9](#_Toc511397914)

[1.6 朴素贝叶斯分类器 10](#_Toc511397915)

[1.7 常用的损失函数 11](#_Toc511397916)

[1.7.1 损失函数的数学解释 11](#_Toc511397917)

[1.7.2 Tensorflow中的损失函数 11](#_Toc511397918)

[1.8 决策树 13](#_Toc511397919)

[1.9 连续值与缺失值处理 15](#_Toc511397920)

[1.10 BP算法 16](#_Toc511397921)

[1.11 无约束最优化问题求解 18](#_Toc511397922)

[1.11.1 拉格朗日乘子法与KKT条件 18](#_Toc511397923)

[1.11.2 梯度下降法 18](#_Toc511397924)

[1.11.3 牛顿法/拟牛顿法 19](#_Toc511397925)

[1.12 最大熵模型与IIS 21](#_Toc511397926)

[1.12.1 最大熵模型 21](#_Toc511397927)

[1.12.2 IIS 22](#_Toc511397928)

[1.12.3 熵/条件熵/互信息 23](#_Toc511397929)

[1.13 局部最小与全局最小 24](#_Toc511397930)

[1.14 支持向量机与拉格朗日乘子法 26](#_Toc511397931)

[1.15 正则化 29](#_Toc511397932)

[1.15.1 LASSO与Ridge 29](#_Toc511397933)

[1.15.2 L1正则化求解 30](#_Toc511397934)

[1.16 EM算法 31](#_Toc511397935)

[1.17 Boosting与Bagging 32](#_Toc511397936)

[1.17.1 Boosting方法 32](#_Toc511397937)

[1.17.2 Bagging方法 35](#_Toc511397938)

[1.18 监督学习与无监督学习 37](#_Toc511397939)

[1.19 聚类 38](#_Toc511397940)

[1.20 特征选择与稀疏学习 40](#_Toc511397941)

[1.20.1 维度灾难 40](#_Toc511397942)

[1.20.2 特征选择 40](#_Toc511397943)

[1.21 PCA 41](#_Toc511397944)

[1.21.1 特征值分解 41](#_Toc511397945)

[1.21.2 奇异值分解 42](#_Toc511397946)

[1.21.3 主成分分析 43](#_Toc511397947)

[1.22 Apriori算法与FP-Growth 45](#_Toc511397948)

[1.22.1 关联规则 45](#_Toc511397949)

[1.22.2 Apriori算法 45](#_Toc511397950)

[1.22.3 FP-Growth算法 46](#_Toc511397951)

[1.23 词频-逆文本词频 47](#_Toc511397952)

[1.24 CVR预估 48](#_Toc511397953)

[1.25 推荐系统 48](#_Toc511397954)

[1.26 Page-Rank 48](#_Toc511397955)

[1.26.1 爬虫与倒排索引 48](#_Toc511397956)

[1.26.2 PR迭代更新 48](#_Toc511397957)

[1.26.3 话题倾向排序 49](#_Toc511397958)

[1.27 概率图模型 50](#_Toc511397959)

[1.27.1 隐马尔科夫模型 50](#_Toc511397960)

[1.27.2 马尔科夫随机场 53](#_Toc511397961)

[1.27.3 条件随机场 54](#_Toc511397962)

[1.27.4 隐狄利克雷分配模型 58](#_Toc511397963)

[1.28 从伯努利到狄利克雷分布 61](#_Toc511397964)

[1.28.1 伯努利分布 61](#_Toc511397965)

[1.28.2 二项分布 61](#_Toc511397966)

[1.28.3 *Beta*分布 62](#_Toc511397967)

[1.28.4 多项式分布 62](#_Toc511397968)

[1.28.5 *Dirichlet*分布 63](#_Toc511397969)

[1.29 常用的不等式 65](#_Toc511397970)

[2. 深度学习 66](#_Toc511397971)

[2.1 神经网络基本框架 66](#_Toc511397972)

[2.1.1 神经网络训练 66](#_Toc511397973)

[2.1.2 神经网络优化 67](#_Toc511397974)

[2.2 全连接神经网络 70](#_Toc511397975)

[2.2.1 激活函数 70](#_Toc511397976)

[2.2.2 前向传播 70](#_Toc511397977)

[2.3 卷积神经网络 71](#_Toc511397978)

[2.3.1 常用的图像数据集 71](#_Toc511397979)

[2.3.2 CNN 71](#_Toc511397980)

[2.3.3 LeNet-5 72](#_Toc511397981)

[2.3.4 迁移学习 74](#_Toc511397982)

[2.4 循环神经网络 77](#_Toc511397983)

[2.4.1 RNN 77](#_Toc511397984)

[2.4.2 LSTM 78](#_Toc511397985)

[2.4.3 GRU 78](#_Toc511397986)

[2.4.4 梯度消散/爆炸与序列截断 79](#_Toc511397987)

[2.4.5 BPTT 79](#_Toc511397988)

[2.4.6 CNN与RNN对比 80](#_Toc511397989)

[2.4.7 NLP 80](#_Toc511397990)

[2.4.8 Word2vec 84](#_Toc511397991)

[2.4.9 TF-Learn 85](#_Toc511397992)

[2.5 RBM与吉布斯采样 87](#_Toc511397993)

[2.5.1 受限玻尔兹曼机 87](#_Toc511397994)

[2.5.2 吉布斯采样 89](#_Toc511397995)

[2.6 GAN 90](#_Toc511397996)

[2.7 文本检测 90](#_Toc511397997)

[2.8 Tensorflow 92](#_Toc511397998)

[2.8.1 三大组件 92](#_Toc511397999)

[2.8.2 Collection 92](#_Toc511398000)

[2.8.3 Variable和变量管理 93](#_Toc511398001)

[2.8.4 多线程 94](#_Toc511398002)

[2.8.5 数据预处理 95](#_Toc511398003)

[2.8.6 图像识别通用框架 99](#_Toc511398004)

[2.8.7 模型持久化 101](#_Toc511398005)

[2.8.8 常用生成函数 102](#_Toc511398006)

[2.8.9 常用数据处理函数 103](#_Toc511398007)

[2.8.10 常用随机数生成函数 104](#_Toc511398008)

[2.8.11 常用文件处理函数 104](#_Toc511398009)

# 机器学习基础

## 模型评估方法

### 数据集划分

泛化误差最小，训练误差过小易导致过拟合，将数据集划分为训练集和测试集，并使训练集与测试集尽量互斥，训练/测试集生成方法：

* 留出法：保留类别比例，分层采样，单次留出得到的评估结果不够稳定可靠，使用时，常采用若干次随机划分重复试验评估取平均值作为最终的评估结果；
* 交叉验证法：分层采样将原数据集划分为***k***个子集，***k-1***个子集作为训练集，1个子集作为验证集，成为**k折交叉验证**，在数据集***D***上采用 ***n***次交叉验证法，成为***n***次***k***折交叉验证；
* 自助法：对***|D| = m***的数据集进行有放回采样，得到训练数据集***D’，|D’| = m，***将初始数据集作为测试数据集测试模型的泛化性能，其中有***36.8%***的样本没有在训练数据集中出现，称为**包外估计**；

### 模型的性能度量

模型的好坏不仅取决于算法和数据，还取决于任务需求：

* 回归任务：均方误差；
* 分类任务：分类错误率；

分类精度：；

**P查准率(准确率)：预测正例中客观正例所占的比率**

**R查全率(召回率)：客观正例中能够预测为正例所占的比率**

**F1指数：**；

### 偏差-误差分解

模型在数据集***D***上的输出为，**模型期望预测**为；

* 方差来源于数据集的波动，方差刻画了数据扰动造成的影响，使用**不同样本集(样本容量相同)产生的模型输出方差**为：

；

* **噪声来源于数据集自身的缺陷**，噪声代表模型在当前数据集上所能达到的期望泛化误差下界，代表问题的难度，数据集自身固有的噪声为：

**；**

* **模型的偏差来源于模型的性能缺陷**，刻画了模型算法的拟合能力，期望输出与样本真实标记之间的偏差为：

***；***

而模型最终的泛化误差可以分解为数据集波动方差、模型预测偏差与数据集自身噪声之和：

***；***

当模型拟合能力较弱时，训练数据集带来的扰动不足以引起模型发生显著的变化，此时模型泛化误差主要来源于偏差，此时，模型处于欠拟合阶段；

随着训练程度加深，模型的拟合能力增强，数据集的轻微扰动都能够引起模型发生显著变化，此时，方差主导了泛化误差的变化，若训练数据自身的非全局的特征被学习到，模型就发生了过拟合；

因此，很多算法会引入策略避免发生过拟合：决策树剪枝，集成学习(Bagging)减少基学习器数量，神经网络引入早停等。

## LR与极大似然估计

***Logistic Regression*与最大熵模型有相同的形式，最大熵模型：**

**；**

**统称为对数线性模型，模型的训练方法为极大似然估计；**

对数几率回归将线性回归模型应用于分类问题：

；

；

对于二类分类问题：

；

；

**极大似然估计法计算参数：**

对于数据集，**使每个样本属于其真实标记的概率最大，**由于样本是独立同分布的，因此使得最大，但是连续乘积运算会导致下溢，因此常采用对数求和形式作为对数似然函数，对模型进行极大似然估计，即有：

；

此处，令***，***，采用后验概率的幂指数形式作为似然项：

；

求解；

是的高阶连续可导凸函数，采用数值方法求解(梯度下降法/牛顿法)；

## LDA

线性判别分析：将训练数据集投影到一条直线上，使得同类样本点尽可能接近，异类样本点尽可能远离，当有新的样本点加入时，根据其投影的位置判断其所属类别；

模型学习算法即使得其同类样本点的投影尽可能接近，即其协方差尽可能小：；

使得异类样本点投影尽可能远离，即使得两类样本点中心距离尽可能大：

；

定义类内散度矩阵：

；

类间散度矩阵：

***；***

优化目标为最大化广义瑞利商：

；令其分母为1作为约束条件，分子取负号作为优化目标得到等式约束最优化问题，拉格朗日乘子法求解；

## 类别不平衡问题

类别不平衡问题是指训练样本中不同类别样本数量不同的情况，处理类别不平衡的策略有：

* 再缩放： ，即使用预测正例/预测负例>正例数量/负例数量作为模型的预测几率；
* 欠采样：取出训练样本集中的过多的反例，使得正负样例数量接近；
* 过采样：增加训练集中的正例样本，使得政府样例数量接近；
* 阈值迁移：使用现有训练集训练模型，将再缩放接过嵌入预测结果；

## 判别/生成模型与先验/后验概率

以概率框架来理解机器学习，机器学习所要实现的任务是基于有限的训练样本极可能准确的估计后验概率***P(c|x)***，主要有两种策略：

* 判别式模型：给定样本***x***，直接建模***P(c|x)***预测样本所属分类***c***；
* 生成式模型：先对联合概率**P(x,c)**建模，由此得到**后验概率*P(c|x)***预测样本分类；

对于生成模型，应用贝叶斯公式：；

其中**P(c)为类别的先验概率**，**是对模型假设空间的描述，与输入样本集没有关系**；

***P(x)***为**输入样本属性的先验概率，是对输入样本集的描述，与样本分类无关**；

**P(x|c)表示在给定类别的情况下，样本在数据集中出现的概率，称为似然**；

* 先验概率：根据大数定理，可以求各个类别(***category***)的样本在训练集中出现的频率作为先验概率 ***(k = 1,2,…,K)***的估计，其中***K = |C|***为类别总数；
* 似然：通常首先假设其具有某一种形式的概率分布，再**基于训练样本子集(训练样本集中属于类别k的样本子集)估算其参数**，参数估计方法即极大似然估计；

关于概率***(Probability)***和似然***(Likelihood)***的区别：当模型参数已知时，求样本***X***出现的可能性，此时**叫做概率函数，概率是已知模型和参数，求数据分布；**

当样本已知，模型参数**作为变量，**此时**叫做似然函数，描述的是对于不同的模型参数，出现样本*X*的概率，似然是已知数据分布，求模型和参数；**

**贝叶斯公式的物理意义：**

**；**

**作为出现的证据，有多大的可能性值得相信的出现与其有关；**

## 朴素贝叶斯分类器

先验概率、后验概率与似然概率见***Chapter 7.5***；

贝叶斯公式：；

贝叶斯公式的物理意义：样本属性**独立于其类别标签被观察到的概率越大，表明其对类别的支持度越小；**

**样本所属分类；即贝叶斯分类器并不将样本直接指派给某一分类，而是给出其所属分类的一个概率，在分类过程中，所有属性都会起到作用，属性可以是离散变量，也可是连续变量；**

类条件概率代表在类别上的所有属性上的联合概率，通常样本具有多个属性(feature)，将其表示成特征向量的形式，即，为样本***x***在属性***i***上的取值***(i = 1, … , d)***，则贝叶斯模型求解的是样本各个特征取值对其所属分类的决定作用；

先验概率可依照大数定理求解，；

而类条件概率(似然概率)较难求解，朴素贝叶斯分类器引入**属性条件独立性假设**：

则有：；

其中类别-属性条件概率；

由条件独立性假设的连乘运算带来的问题：训练样本集中的未出现属性值的类条件概率，会将其他属性的类条件概率抹去，因而引入**拉普拉斯平滑**策略：

；其中***K=|C|***为数据集中类别的总数；

；其中**为**第***i***个属性的取值个数***( ，l = 1, … , )***；

## 常用的损失函数

### 损失函数的数学解释

**损失函数*Loss Function*用来度量模型输出与样本标签之间的差异**，在模型的训练过程中，以最小化损失函数为目标，选择优化策略，调整模型参数，将训练过程写作：

；

其中为模型的损失函数，衡量模型在训练数据集上的经验风险；

是为了避免模型过拟合引入的正则化项，衡量模型的结构风险；

* 回归问题常用的损失函数为均方误差；
* 分类问题常用的损失函数有：

**0/1损失 ；**

**0/1损失函数非凸不连续，数学性质不好，不易于求解，通常选用其替代函数作为替代损失函数：**

**hinge损失(hinge loss)： ；**

**指数损失(exponential loss)：**

**对率损失(logistic loss)： ；**

### Tensorflow中的损失函数

神经网络模型使用**交叉熵*(Cross Entropy)***度量网络输出向量与期望向量之间的距离：**，其中和是随机变量*X*的概率分布；**

**通常神经网络的输出向量并不是一个规范化的概率分布，要使用Softmax Regression将输出向量 归一化处理：**

**；**

**在神经网络的训练过程中，得到经过*softmax*层处理后的输出：**

**是一个概率分布，将一个*Batch*的样本输出组织成矩阵 ，将神经网络输出逐个元素取自然对数并与样本标记矩阵做哈达马乘积*(Hadamard Product)*，即对应元素相乘得到矩阵，逐行对元素求和得到*N*个样本的交叉熵，再对*N*个交叉熵求均值即可得到一个*Batch*的平均交叉熵；**

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

**而当应用于分类问题的神经网络输出只有一个正确答案时，可以使用加速计算交叉熵方法：**

**cross\_entropy =**

**tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=y, labels=argmax(y\_, 1))**

**其中logits=y必须是前向传播未经过Softmax Regression的输出结果，否则会产生错误结果，而labels=argmax(y\_, 1)是从BATCH\_SIZE\*10维样本标记向量中选出每行中最大值作为其类别标签；**

回归问题使用均方误差***(Mean Squared Error)MSE***度量输出预测值与样本标签之间的偏离程度：

mse = tf.reduce\_mean(tf.square(y\_ - y))

[神经网络优化](#_神经网络优化)

## 决策树

决策树的数据结构：

* 根节点：根节点包含全部样本集；
* 内部节点：对样本集进行属性测试，根据属性测试的结果将样本集划分到其左右子树；
* 叶子节点：对应于样本集的分类结果；

决策树划分的停机条件：

* **当前节点包含的样本全部属于同一类别；**
* **当前属性集为空，或所有样本在所有属性集上评价指标相同；**
* **当前节点包含样本集为空；**

决策树划分过程要度量按照某一属性划分后其样本集合的纯度，常采用指标有：

* ***ID3(Iterative Dichotomiser)***算法使用的性能指标为信息增益：

；

其中；***其中K为样本所属类别总数***

；***其中n为当前待划分的属性取值个数***

；

* ***C4.5(Classifier)算***法使用的性能指标为信息增益比：

；

其中；

；

* ***CART(Classification and Regression Tree)***算法使用的性能指标是基尼指数：***；***

***；***

决策树的剪枝处理，在决策树的学习过程中，为了尽可能正确地对训练数据集进行分类，节点的划分过程或不断进行，可能会导致过拟合，因此需要对生成的决策树进行剪枝处理：

* **预剪枝**：在决策树生成过程中，对每个节点在划分之前进行估计，若按照当前划分策略进行划分能够带来决策树整体的泛化性能提升，则 进行划分，否则停止划分并将当前节点设置为叶子节点，是一种贪心策略，会导致决策树欠拟合；
* **后剪枝**：对生成的决策树从下向上进行考察，若将当前节点对应的子树替换为叶子节点能够带来决策树整体泛化性能的提升，则将当前子树替换为叶子节点，即不按照原划分策略划分数据集；

决策树的剪枝目标是极小化树的整体损失函数：

；

其中，***T***为决策树叶子节点，***|T|***为决策树叶子节点的个数，代表决策树模型的复杂程度；

为当前叶子节点上样本的数量，，其中K为样本所属种类数量；

为模型的经验风险，代表决策树对训练数据集的拟合程度，为模型结构风险，代表决策树模型的复杂程度；剪枝过程的目标就是最小化整体损失函数；

后剪枝决策树欠拟合风险较小，泛化性能比预剪枝决策树好得多，但是训练过程及其时间开销较大。

## 连续值与缺失值处理

属性的连续值处理：给定样本集***D***和连续属性***a***在***D***上出现了***n***个取值，选择划分点进行划分，检查此划分对样本集合的信息增益：

；

选择使得信息增益最大的**t**作为连续属性***a***的划分点；

属性的缺失值处理：将缺失数值的样本以不同的概率划分到不同的子节点中去；

## BP算法

神经网络的误差反向传播算法***(Error Back Propagation)***，基于**梯度下降法**，以目标函数的负梯度方向对参数进行调整：

参数更新式：；

目标函数： 均方误差；

对于训练样例，神经网络输出为，代表经过***softmax***层处理后得到的一个概率分布；

而代表第j个输出层神经元的输出， ***sigmoid***函数作为其激活函数，为其激活阈值；

代表第***j***个输出层神经元的输入，为连接权重；

为第***h***个隐含层神经元的输出，；

为第***h***个隐含层神经元的输入 ；

用到***Sigmoid***函数的一个重要特性；

依次对各个权值和激活阈值求偏导数，得到各个参数的更新式；

**BP算法**的工作流程：将输入样例提供给输入层神经元，逐层向前传递，直至产生输出，计算输出与样例标签的误差，再将误差向前传递至隐含层；

参数更新列表：

**；；*沿着负梯度方向以学习速率更新参数***

**；；**

**；；**

**；**；

标准BP算法只针对单个样例进行误差反向传播，参数更新较频繁，通常使用累计误差反向传播(按照样例的***Batch***)：

；

***BP***神经网络防止过拟合的措施：

* 早停：将数据集划分为训练集和验证集，训练集误差用来计算梯度、更新网络连接权值和激活函数阈值，验证集用来估算当前模型产生的验证误差，若模型在训练集上误差降低而验证集误差上升，则停止参数更新，并返回拥有最小验证集误差的参数作为最终的模型；
* 正则化：在误差目标函数中增加用于描述网络复杂程度的正则化项：

；

***Tensorflow***中的**tf.contrib.layers.l2\_regularizer(REGULARIZATION\_RATE)**即对连接权重采用了***L2***正则化项；

此外，***LSTM***的***dropout***方法在训练时随机丢弃多层之间的连接权重也可以有效避免过拟合：**lstm\_cell = tf.nn.rnn\_cell.DropoutWrapper (lstm\_cell, output\_keep\_pro = KEEP\_PROB)**

## 无约束最优化问题求解

凸优化问题按照有无约束条件可分为约束优化问题和无约束优化问题，约束优化问题按照其约束条件是等式还是不等式可分为等式约束优化问题和不等式约束优化问题，**等式约束优化**可直接使用**拉格朗日乘子法求解，**而不等式约束优化使用***KKT*条件**下的拉格朗日乘子法将原始的**极小-极大问题**转换为求解**对偶问题极大-极小**问题；

常用的无约束最优化问题求解方法有**梯度下降法**和**牛顿法/拟牛顿法：**

* 梯度下降法使用目标函数的**一阶偏导数**求解参数的梯度向量，并沿着负梯度方向进行参数更新；
* 牛顿法/拟牛顿法使用目标函数的二阶偏导数生成目标函数的***Hessian*矩阵，**通过其逆矩阵进行参数更新求解；

### 拉格朗日乘子法与KKT条件

不等式约束的拉格朗日乘子法求解及***KKT***条件详见[**Chapter 7.14 支持向量机与拉格朗日乘子法**](#_支持向量机与拉格朗日乘子法)；

### 梯度下降法

梯度下降法即沿着目标函数 的参数的负梯度方向以学习速率***η***逐步更新模型参数以达到目标函数的极小值；深度学习中使用的**三种梯度下降算法框架的区别在于每次学习使用的样本个数不同**，从而导致学习的效果不同(正确率与运行时间)：

* **批量梯度下降*Batch Gradient Descent***：使用全部的训练样本累积误差更新模型参数，保证能够收敛于极小值点(凸函数收敛于全局极小值点，非凸函数收敛于局部极小值点)

**for i in range(max\_epochs):**

**Grads = evaluate\_gradient (loss\_function, dataset, params)**

**Params = params – learning\_rate \* grads**

批量梯度下降方法每次计算整个数据集的累计误差，学习时间过长，内存开销大，无法进行参数在线更新；

* **随机梯度下降*Stochastic Gradient Descent***：每次从训练集中随机选择一个样本计算目标函数值 ，沿着负梯度方向以学习速率***η***更新参数，学习速度快，并且可以在线更新参数

**for i in range(max\_epochs):**

**np.random.shuffle (dataset)**

**for example in dataset:**

**grads = evaluate\_gradient(loss\_function, example, params)**

**params = params – learning\_rate \* grads**

***SGD***算法的训练过程不会一直向着正确方向进行，会产生较大的波动，但是也利于算法跳出局部最优，对于凸函数来说，可能会收敛于全局最优，非凸函数会收敛于一个 较好的局部最优值，***SGD***算法收敛速度较慢，学习过程漫长；

* **小批量梯度下降*Mini-batch Gradient Descent***：综合***BGD***和***SGD***算法的特点，每次从训练数据集中随机选择一个***batch***进行训练***(batch\_size < dataset\_size)***，既保证算法的稳定性，又减小训练过程的内存开销，缩短训练过程

**for i in range(max\_epochs):**

**np.random.shuffle (dataset)**

**for batch in get\_batches(dataset, batch\_size = BATCH\_SIZE):**

**grads = evaluate\_gradient (loss\_function, batch, params)**

**params = params – learning\_rate \* grads**

***tensorflow***中使用***API***实现的***SGD***算法实际上是***Mini-batch Gradient Descent***，依靠训练数据集的***batch***划分实现小批量样本训练过程的随机梯度下降；

***optimizer = tf.train.GradientDescentOptimizer(LEARNING\_RATE).minimize(loss)***

### 牛顿法/拟牛顿法

对于无约束最优化问题求解：

**；**

目标函数存在极小值的必要条件是其梯度向量在极值点处为***0***，而当其***Hessian***矩阵为正定矩阵时，此极值点为极小值点；

将目标函数**在第*k*次迭代的取值处进行二阶泰勒展开：**

**；**

**其中： 为目标函数的梯度向量在处的取值；**

**为目标函数的Hessian矩阵在处的取值；**

对目标函数的二阶泰勒展开式求梯度并令其为***0***：

**，由此可以得到参数更新式：**

；

类比于梯度下降法参数更新式 **，牛顿法在负梯度之上还考虑了目标函数的*Hessian*矩阵的逆矩阵作为参数更新方向，而学习率需要一维搜索实时更新：；**

得到牛顿法求解无约束最优化问题的步骤：

1. 输入目标函数及其梯度向量和***Hessian*矩阵 ，精度**；
2. 初始化变量点，设置***k=0***；
3. **计算和 ，若梯度满足，则停机，否则更新参数：**

**；**

牛顿法中目标函数的***Hessian***矩阵求逆运算较难实现，因此引入拟牛顿方法，使用一个***n***阶矩阵**近似，也可使用矩阵逐步逼近：**

**由 ，令 可得：**

**；即有拟牛顿条件：**

**；选择一个正定矩阵近似可保证参数更新的方向是向下的，从而完成参数和矩阵的更新：**

1. 输入目标函数及其梯度向量，**精度**；
2. 初始化变量点，选择正定对称矩阵，设置***k=0***；
3. **计算，若则停机，否则进行参数更新：**

一维搜索更新学习率**；**

**更新变量；**

**计算，若则停机，并置近似解，否则更新矩阵；置*k=k+1*转下一次迭代；**

## 最大熵模型与IIS

### 最大熵模型

最大熵模型原理可表述为：在满足约束条件的模型集合中选取具有最大熵的模型作为待求模型的解；

模型集合：；

其中**是描述随机变量*x*和*y*关系的特征函数：**

**表示特征函数关于经验分布的期望；**

**表示特征函数关于模型和经验分布的期望；**

**将作为满足模型要求的约束条件，而模型的条件熵为：**

；

因此得到最大熵模型的目标就是要在模型集合**中寻找最大的模型：**

**；**

**；**

**；**

输入数据集，特征函数集 **；**

最大熵模型的求解：

**；**

**；**

**；**

构造拉格朗日函数：

**；**

**原始最优化问题是极小-极大问题 ，对偶问题为：**

**在满足*KKT*条件的情况下成立；**

先求解极小化问题称为对偶函数，对偶函数的极大化等价于模型条件概率的极大似然估计：

：

；

令**，得到：；归一化处理，分子分母同除以1，由得到：**

**；**

再求对偶函数**的极大化问题，拉格朗日乘子参数的解：**

；由求得的参数确定最大熵模型**；**

### IIS

改进的迭代尺度法***(Improved Iterative Scaling)IIS***用来解决最大熵模型(或其他对数线性模型如***LR***)的模型学习问题，最大熵模型的学习问题实质上是以对数似然函数为目标函数的最优化问题：

**；令；**

**则有：**

***；***

***IIS***寻找一组参数更新的增量**，参数向量在一次迭代后更新为 ，对数似然函数增大，不断迭代求解，直到不再增大为止；**

对数似然函数增量式寻找其下界，得到：

；

其中特征计数，令：

**，**

**得到参数更新方程：**

***；***

依次求解参数更新方程即可得到增量向量**；**

**对于参数更新方程的求解：**

**若特征计数值为常数，则 ；**

**当，而与变量*x,y*有关，则不是常数，则参数更新方程不能显式求解，应用数值方法求解：牛顿法或拟牛顿法；**

### 熵/条件熵/互信息

**熵*(Entropy)*：** ；

其中是随机变量***Y***的概率分布，表示随机变量***Y***分布的不确定性，要消除这种不确定性，唯一有效的方法就是从外部增加信息；

**条件熵*(Conditional Entropy)***： **；**

**；**

不等式代表额外引入的信息***X***使得***Y***的不确定性下降，当引入信息与***Y***毫不相关时，取等号；

**互信息*(Mutual Information)***： **；**

**互信息描述了两个信息之间的相关性；**

## 局部最小与全局最小

梯度下降法是沿着负梯度方向寻找最小值，对于非凸函数可能会陷于局部最小点，甚至有可能是鞍点，无法得到期望的全局最小值，解决方法：

* 多组参数值初始化神经网络，按照标准方法训练后，取其中验证误差最小的模型作为最终的模型；
* 模拟退火：每一步训练都以一定概率接受比当前解更差的结果，但接受次优解的概率要随着训练过程的进行逐步下降，以保证算法收敛；
* 随机梯度下降：***SGD***算法，利用单个样例训练模型引起的波动跳出局部最小；

为了**解决SGD算法求解非凸函数易陷于局部最优的情况**，可采用以下方法：

* 在目标函数的峡谷地区，某些反向上比较陡峭，***SGD***算法会产生较大的震荡，此时，可以采用**Momentum**方法对参数进行更新：

***；***

***；***

除目标函数的梯度以外，给参数更新式增加了动量因素，若当前更新方向与上次更新方向相同，则加速更新，若方向相反则减缓更新，从而有效避免在峡谷地区的震荡并加速收敛；

* 涅斯捷罗夫梯度加速***NAG***：在***Momentum***的基础之上，在损失函数中减去动量部分，能够预估下一次参数更新的位置

***；***

***；***

* 自适应学习速率梯度下降***Adagrad***：对学习速率进行约束

***；***

***；保证分母不为0***

***；***

对梯度从**1**到***t***形成一个对学习率的正则化约束项***regularizer***

；

**训练前期梯度较小时，regularizer能够放大当前梯度，加速更新；**

**训练后期梯度较大时，regularizer能够缩小当前梯度，减缓更新；**

可以辅助早停，适合处理稀疏梯度，但是依赖于预设定的全局学习速率；

* ***Adadelta***扩展了***Adagrad***方法：

***；***

***；***

继续处理如下：

**；**

；

不再依赖于全局设定的学习速率**，**训练前期收敛较快，后期在局部会产生抖动；

* ***RMSprop***算法：介于***Adadelta***和***Adagrad***之间

；令，则变成对梯度求算数均值，对再度求根式，可得到

；依赖于全局学习速率，适合处理非平稳目标，如***RNN***等；

* ***Adam***算法***Adaptive Moment Estimator***，带有动量项的***RMSprop***，经过偏置校正后，每一次迭代学习速率都有一个确定的范围，使得参数较平稳：

**；*对的一阶矩估计***

**；*对的二阶矩估计***

**；**

**；**

**；**

结合了***Adagrad***善于处理稀疏梯度和***RMSprop***善于处理非平稳目标的优点，对内存开销较小，适用于大多数非凸函数的优化；

对于稀疏数据集，尽量使用自适应优化方法，无需手动调节，最好采用默认值；

***SGD***训练时间较长，但在好的初始数值和学习率调整方案的情况下，结果更为可靠；

训练深度网络时，使用学习率自适应优化方法；

## 支持向量机与拉格朗日乘子法

支持向量机问题就是寻找一个分类超平面，使得距离其最近的异类样本点的间隔最大，其中距离分类超平面距离最近的点叫做支持向量，异类支持向量使得：

***；***

***；***

异类支持向量的间隔成为分类间隔： ；因此求分类超平面问题变为不等式约束的最优化问题(**支持向量机的基本型**)：

**；**

构造拉格朗日函数：

**；**

对模型参数求偏导，并令其为0，得到：

**；**

**；**

带入拉格朗日函数得到：

**；**

**原问题为 在满足KKT条件的前提下，原始问题与其对偶问题具有相同解，**问题转换为： **；**

**;**

**；**

**并满足KKT条件：**

**；**

**；**

；

***KKT***条件限制了非支持向量点不会出现在求和式中，出现在其中的点都是支持向量，此时不等式的等号成立，不等式约束取最大值， ，即模型训练完成后，大多数样本点不需要保留，仅保留落在分类间隔上的支持向量点；

**；**

**;**

**；**

原始问题的对偶问题是一个二次规划问题，经常使用***SMO(Sequential Minimal Optimization)*算法**求解：

每次只更新两个参数而固定其他参数，利用等式约束条件有：

**；**

**选择两个间隔最远的样本，使得目标函数有最大更新，带入原式，就会得到一个单变量的函数求极值问题，具有闭式解；**

**对于偏移项b的求解：对于所有的支持向量具有 ；**

**S是所有支持向量的下标集，更加鲁棒的结果是；**

***SVM***按照其应用是线性分类问题还是非线性分类问题可分为线性支持向量机和非线性支持向量机；线性支持向量机按照其数据集是否线性可分也分为线性可分支持向量机和线性支持向量机；

线性可分支持向量机应用于数据集严格线性可分的情况下，学习目标是分类间隔最大化；

线性支持向量机应用于数据集非严格线性可分的情况下，学习目标是使得分类间隔尽量大同时保证误分类点尽量少，引入松弛变量  **,**  ，得到优化问题：

***；***

***；***

构造拉格朗日函数，对参数求偏导并令其为0，并与原始问题联立，得到对偶问题描述为：

**；**

**;**

**;**

**；**

**；**

满足KKT条件：

**； ；**

**；**

**；**

**； ；**

称线性支持向量机算法的优化目标为**软间隔最大化**，此时，支持向量可能落在分类间隔上，对应 ，支持向量可能落在分类间隔与分离超平面之间，对应**；**支持向量可能落在分离超平面上，对应**；**支持向量还有可能落在分离超平面的误分类一侧，对应**；**

**对于非线性分类问题，常采用非线性支持向量机：**

将低维空间中的不可分数据集映射到高维空间中会得到线性可分数据集，非线性映射：

*；*将样本集从输入空间映射到高维希尔伯特空间；

假设对于输入样例 ，存在核函数为样例在高维空间中的内积，则可将核函数应用于非线性支持向量机问题中：

**；**

**;**

**；**

得到最优解 ；选择其中一个正分量 **对应的样本点**，求偏置量 **；**

**最终得到的决策函数为：**

当核函数**正定，**非线性支持向量机模型为凸二次规划问题，也使用***SMO***算法求解。

## 正则化

### LASSO与Ridge

考虑模型的优化问题

；

其中为损失函数：

* 若平方误差，模型为最小二乘；
* 若 ***hinge***损失，模型为支持向量机；
* 若 指数损失，模型为***Boosting***，基分类器的指数损失函数为：

按照分类结果与样本标签是否相同，将指数损失函数变换为：

**，**

**，**

以表示基分类器的分类错误率，则将上式表示为：

**；**

* 若 对数损失，模型为**logistic regression** ；

为正则化项，范数是常用的正则化项，参数越少，模型越简单，越不容易发生过拟合***(Occam Razor)***，模型泛化能力越强：

若***，***则**范数 为其各个分量中非0元素的个数**，希望最终得到的模型参数向量是稀疏的，此时模型的**可解释性**(**直观地表示出预测输出与哪些关键特性相关**)最好，但是范数非凸、不连续，数学性质不好，难于求解；

**范数，称为稀疏规则算子，对范数的回归称为岭回归(Ridge Regression)，也叫做权值衰减(Weight Decay)，得到的模型具有稀疏性；**

**范数 ，作为正则化项，倾向于使得权值的分量接近于0，而非直接为0，对范数的回归成为LASSO Regression，得到的模型较平滑；**

**系统权值矩阵的ill-condition number计算方法：**

；用来衡量系统的可信度，即当输入发生微小变化时，输出的变化的程度，若接近于1，则系统是***well-conditioned***，距离1越远，其病态就越严重；

对于系统权值向量的求解问题：

；为其增加一个规则项，得到问题的解：

，若没有规则项引入，解线性方程会产生不稳定解，引入规则项会改善系统的***condition number***；

规则项的引入，实际上将目标函数变成了具有**强凸性质**的函数：

**；**

而一般的凸函数定义为：

**；**

**仅依靠convex性质不能保证在梯度下降和有限的迭代次数的情况下得到全局最优解的近似，若全局最优解附近比较平坦，不停机的情况下，可能会得到一个距离较远的点**；因此，引入范数作为正则化项 ，将目标函数变为强凸，从而拥有一个二次下界，得到较稳定的解和较快的收敛速度；

观察**和**规则化的优化问题：

**；**

**；**

即***LASSO***回归使用***L1-ball*超矩形与等高线相交，大多数相交点均在超矩形的角上，会有多数权值的分量为0，因此会得到稀疏解；**

而***Ridge***回归使用***L2-ball*超球面与等高线相交，不易产生稀疏解，因而模型比较平滑；**

### L1正则化求解

***L1***正则化求解使用近端梯度下降方法***(Proximal Gradient Descent)PGD***：

**；**

**若可导，且满足*L-Lipschitz*条件：**

**；**

**将二阶泰勒展开：**

**；**

**即可得到最优化问题：**

**；**

**令，可得参数更新式：**

**；**

## EM算法

期望最大化算法(***Expectation-Maximization***)算法是对**含有未观测属性(隐变量)**的概率模型参数的**极大似然估计方法，是一种坐标下降优化算法**；

对于待解决问题**，**其中**是已观测属性**，**是模型参数，此外还有未观测属性作为隐含变量；是不完全数据的边缘概率：**

**；**

由不完全数据**边缘概率的对数似然函数极大化的迭代求解**可以导出**完全数据的对数似然函数**关于在给定**观测数据和当前参数估计下对未观测属性(隐含变量)依条件概率的数学期望：**

**；**

**称此期望为Q函数，即有：**

**；**

可证明对数似然函数是**的下界；**

EM算法分为两步进行：

* **E步**求期望： **；**
* **M步**最大化***Q***函数求参数： **；**

迭代***E***步和***M***步，直至收敛使得 ；

## Boosting与Bagging

集成学习***(Ensemble Learning)***方法可以分为两种：以***Boosting***为代表的串行化集成学习方法和以***Bagging***、***Random*** ***Forest***为代表的并行化集成学习方法；

**当样本集的个体之间存在较强的依赖关系时，使用串行化学习方法；**

**当样本集的个体之间不存在较强的依赖关系时，使用并行化学习方法；**

### Boosting方法

***Boosting***对特定的数据分布进行学习，在每一轮训练中对数据集中的样例 赋一个权重，在上一轮训练中正确分类的样本点的权重会降低，而误分类点的权重会得到相应的上升，每一轮均使用一个弱分类器作为基分类器对全部样本进行分类，同时生成当前基分类器的权重**，；**最终得到一个线性加性模型作为集成分类器；

**用于二分类问题的AdaBoost(Adaptive Boost)算法：**

1. 对原始数据集中的样本分配初始权值，并训练第一个基分类器；
2. 迭代式：对第***t (t=1,2,…,T)***轮迭代，得到基分类器，此时有组合分类器，对训练数据集中样本的分类误差为，而**基分类器 的权重为**，由**最小化指数损失函数** 求得；

更新样本集中的样例权重 **；**

1. 最终得到分类器  **；**

常采用的基分类器有**决策树桩**等；

基于梯度下降的梯度提升***(Gradient Boosting)***算法：与***AdaBoost***不同，***GB***算法是对**损失函数*Loss Function*的负梯度**的拟合，其中，

**负梯度**又称为**伪残差，是回归问题中的残差推广到分类、排序等一般性问题中的近似**；

**GBDT算法的一般过程：**

1. 初始化基学习器为常数：**；；**

**;**

1. 迭代式：对于第***t ( t=1,2,…,T )***步，计算伪残差：

拟合伪残差序列学习一个基分类器，并求其权重：

**；**

更新模型**；**

1. 最终得到线性加性分类器模型：**；**

使用决策树作为基分类器的***GBDT(Gradient Boosting Decision Tree)***算法：同***GB***算法，其主要的关注问题变为，**使用*CART*拟合伪残差序列**，经常使用深度***deepth<5 ，***叶节点数***leafnodes<10***的决策树作为基学习器，此时衡量分类效果好坏的标准不再是Gini指数，而使用均方误差，因而，此时的决策树是一棵最小二乘树，最小二乘树生成过程中切分变量及切分点的选择：

；

根据选定的切分特征及其切分点取值，将数据集划分为两个子区域：

；

；

而对于这两个子区域中样本标签的估计为：

**；**

**；**

**最小二乘回归树生成算法的停机条件：**

1. **达到最大迭代次数；**
2. **叶子节点中拥有单一的输出；**
3. **特征空间中没有更多的特征供切分；**

**最终生成的回归树模型将数据集切分为M个子区域，表示为：**

; ***t = 1,2,…,T***

**再将以上得到的最小二乘回归树应用于GB算法中，就得到了GBDT；**

除了使用***CART***还可以使用线性分类器作为基学习器的***XgBoost***算法：***XgBoost***与***GBDT***算法步骤相同，都是先初始化一个常数作为基分类器，不同点在于***GBDT***算法使用一阶偏导数生成伪残差序列，而***XgBoost***算法使用一阶偏导数和二阶偏导数生成伪残差：

**给损失函数增加一个正则化项描述决策树的复杂程度：**

**；**

在第***t***步迭代中，拟合残差得到的最优决策树就是使得目标函数最小的决策树，则损失函数可以表示成：

**；**

其中***const***表示已获得的***t-1***棵决策树的正则化损失为常数，将损失函数进行二阶泰勒展开：

对于第***t***轮迭代计算当前最有决策树的过程来说，也是常数项，最小化损失函数不需要考虑常数项，因此将目标损失函数重新写成：

将每一次迭代得到的决策树表示为：

；

***M***代表决策树的叶子节点个数，***M*个叶子节点上的输出值**组成决策树的输出空间，**代表将样本从原始特征空间到决策树输出空间的映射，是一个M维向量，是对样本的预测值，**因此***XgBoost***算法使用了正则化项：**；**

定义**，；**

则有：

**;**

决策树的作用是将原始数据集中的N个样例划分到M个叶子节点中去，因此可以将**改写成按照叶子节点求和，因此得到变形：**

**;**

得到一个简单的二次式，可求解最优解：

**；**

**；**

即第***t***次迭代得到的最优决策树仅与决策树的结构相关，而与叶子节点的取值无关，即此时选择决策树只涉及到***特征(key)***的选择，还没有涉及***切分点(value)***，以上作为决策树生成过程的特征选择过程，即划分**子区域(叶子节点) ；而特征切分点的选择需要考察增益值Gain，Gain的意义是指当前节点被切分后的损失函数的减少，此时将带切分节点作为根节点，切分后得到其左右子树，考察其函数：**

**越大，表明切分效果越好，选择使取值最大的样本标签作为切分点；**

重复上述过程，先选择切分特征，得到最好的决策树结构，再选择切分点，逐步得到最优的决策树；其中的**参数**用来控制决策树的复杂度，在生成决策树的过程中就考虑了过拟合问题，因此后续不需要剪枝处理；

此外，在每一步迭代中，生成决策树使用到的**和两个梯度都可以对每一个样本分开计算，使用并行化处理可以加速训练过程；**

**以上Boosting算法主要关注降低模型的偏差；**

### Bagging方法

***Bagging*** ***(Bootstrap Aggregating)*** 使用**自助采样法(有放回)**从数据集中抽取T个样本容量为N的子集作为训练数据集，此时原始数据集中含有36.8%的样本没有被抽到，称为包外估计，基于每个子集训练基分类器(常见于不剪枝决策树，神经网络等)最终T个分类器以投票法决定测试样本的分类结果，**以样本x的包外预测衡量算法的分类错误率**：

***；***

**；**

扩展算法为**随机森林(Random Forest)：RF算法**不仅沿用了**Bagging算法在训练数据集生成过程**中的样本扰动***(Bootstrap Sampling)***还增加了属性扰动：

从样本集***D***的属性空间中随机选择***k***个属性，得到属性子空间，用于基决策树的生成，***RF***组合基决策树得到最终的分类模型，通过增加个体学习器之间的差异提升最终模型的泛化能力；

***Bagging*算法关注降低由于样本扰动带来的方差**；

## 监督学习与无监督学习

按照训练数据集中的样例是否具有标签，将学习任务分成监督学习和无监督学习两类，其中样本的标签包含了其所归属的分类(***category***)信息；最常见的无监督学习就是**聚类**；

监督学习的问题分为三大类：

* 分类问题：**；**样例的标签信息即输出变量***Y***的取值为有限个离散数值时，对样例标签的预测问题即分类问题，此时样例的属性信息即输入变量***X***可以是离散的***(SVM, Decision Tree, Naïve Bayes, EM，KNN等)***，也可以是连续的***(Logistic Regression)***；
* 标注问题：**；**作为分类问题的推广，标注问题的输入**是一个观测序列，输出是一个标记序列或状态序列，**标注问题的学习目标就是找到一个模型，能够对给定的观测序列预测其对应的状态序列；
* 回归问题：**；**预测输入变量***x***与输出变量***y***之间的关系，回归问题的学习等价于函数拟合；

## 聚类

聚类算法是典型的无监督学习算法，在缺少样本标签信息的情况下，借助样本的特征信息或其他信息将其划分到若干个不相交的子集中去；

* 原型聚类：原型聚类的意义在于寻找样本空间中具有代表性的点完成类别划分；**常使用*K-Means*聚类算法，将样本空间划分成*K*个簇，学习目标为最小化均方误差：**

***；***

**其中，称为均值向量；**

1. 从数据集D中随机挑选***k***个样本作为初始均值向量**；**
2. 迭代式：置**；**

对每一个样本计算其与各个均值向量之间的距离

***；***

则样本的簇标记为 ，将样本划入对应簇：**；**

遍历结束后，更新各个簇的均值向量

1. 重复上述过程，直至均值向量不再更新或达到最大迭代次数；

* 密度聚类：***DBSCAN***算法，基于样本点的密度分布进行聚类，选择数据集中的一个核心对象作为种子，由此出发确定相应的聚类簇：

1. 初始化核心对象集合，从原始数据集中挑选出临域内样本点数量大于***MinPts***的样例作为核心对象，并加入核心对象集合；
2. 寻找所有核心对象的密度可达的样本点，通过核心对象将密度相连的样本点聚集成簇；

初始化聚类簇数***k=0***，未访问样本集合**；**

**对核心对象集合进行遍历：**

备份未访问样本集合**；**

随机选取核心对象点并初始化待聚类队列**；**

**：**

***；***取出队列的***head*元素**；

若***q***是一个核心对象，即，则将其临域中未被访问过得样本点均加入队列，并将其从中剔除：

**；**

直到队列***Q***为空停止，生成聚类簇，更新核心对象集合**；**

***；***

**核心对象o出发得到其临域内所有样本点，这些样本点中有的是核心对象，有的不是核心对象，从这些核心对象出发继续访问其临域…，直到将所有从*o*出发密度可达的点纳入一个聚类簇中，同时将簇中的元素从未访问集合中剔除，并从核心对象集合中剔除，从而得到*K*个聚类簇划分；**

* 层次聚类：在不同的层次对样例进行聚类，典型应用***AGNES***算法，自底向上层次聚类，初始时将所有样本点看做一个单点簇，两两求其**距离度量(最大距离，最小距离，平均距离)，**合并距离最近的簇并**重新为聚类簇编号，**重复进行以上过程，直至达到预设的聚类簇数量为止；常用的集合间距离度量：

最小距离：

最大距离：

平均距离：

## 特征选择与稀疏学习

### 维度灾难

给定数据集**，其中维行向量**定义了样本的属性信息，包含个特征**，，为第个特征的取值空间，**则分类器要完成所有特征的学习并找出其与样本标签的关系**，**至少需要个样本，当**数值较大时，**数值极大，即出现了**组合爆炸**问题，而实际给定的数据集不可能完全覆盖所有可能，称其为**样本稀疏**问题，这种由特征维度过大带来的组合爆炸与样本稀疏问题统称为**维度灾难**，维度灾难还会导致模型过拟合；

### 特征选择

在数据预处理过程中，保留样本的相关特征***(Relevant Feature)*，**去除无关特征***(Irrelevant Feature)***的过程叫做特征选择，即从原始特征集中选择特征子集并进行评价：

* 子集搜索与子集评价：贪心策略，选择特征作为初始特征子集，逐步选择候选特征**，**在已划分子集上计算以新增特征**进行划分**的信息增益**，若则在当前节点上以候选特征进行划分得到子集，并更新特征子集和划分子集，进入下一轮更新；**

类似于决策树的生成策略，除信息增益之外，还可以选择信息增益比和基尼指数作为子集评价标准；

* 特征选择方法：

过滤式选择方法，先对数据集进行特征选择***(Filter)***再训练学习器，二者分开进行；

包裹式选择方法：将最终要得到的学习器的性能衡量作为特征选择的评价标准；

嵌入式选择方法：目标函数求解时加入正则化，易于得到稀疏解，仅选择权值向量**的非零分量对应的特征进行学习，自动实现特征选择；**

## PCA

### 特征值分解

特征值分解***(Eigenvalue Decomposition)***只能对方阵进行：

**；**

**求解特征值，通过初等行变换和列变换求解特征值，例如：**

**；**

**；**

得到矩阵***A***的特征值为**；**

解线性方程组**，**得到二重根***λ=-2***对应的基础解系为：

**；**

***λ=4***对应的特征向量为**；**

**特征值分解可以将矩阵写成，其中是由矩阵A的特征值组成的对角阵，是对应的特征向量组成的矩阵；**

可以将矩阵看做对向量的线性变换，则矩阵的特征向量代表变换的方向，而特征值就代表变换方向的主次重要程度，要想描述一个线性变化，只需要描述主要的变换方向即可；

在***Python***中使用***numpy***进行矩阵特征值分解运算：

import numpy as np

A = np.array([[1,-3,3],[3,-5,3],[6,-6,4]])

d, v = np.linalg.eig(A) # d是特征值，v是特征向量组成的矩阵

>>d

array([4.,-2.,-2.])

>>v

array([[…,…,…],[…,…,…],[…,…,…]])

X = np.diag(d)

>>X

array([[4.,0.,0.],

[0.,-2.,0.],

[0.,0.,-2.]])

X = np.mat(X) #将array形式转换成matrix

v = np.mat(v)

v\*X\*(v.I) # v.I求其逆矩阵

>> matrix([[1.,-3.,3.],

[3.,-5.,3.],

[6.,-6.,4.]])

### 奇异值分解

不同于特征值分解只能针对方阵提取其特征，奇异值分解***(Singular Value Decomposition)SVD***适用于任意矩阵：

；

**矩阵中对角线上的元素是矩阵的奇异值，除对角线意外其他元素均为0，方阵中的向量两两正交，称为左奇异向量，中的向量也是两两正交，称为右奇异向量；**

奇异值分解通过构造方阵并进行特征值分解完成：

**得到其奇异值为 ，对应右奇异向量为，对应左奇异向量为**

**；**

在**Python**中对矩阵进行奇异值分解：

import numpy as np

A = np.array([[1,1,1,0,0],

[2,2,2,0,0],

[3,3,3,0,0],

[5,5,3,2,2],

[0,0,0,3,3],

[0,0,0,3,3],

[0,0,0,6,6]])

u, sigma, v\_t = np.linalg.svd(A)

>>u

matrix([[…,…,…,…,…,…],

[…,…,…,…,…,…],

[…,…,…,…,…,…],

[…,…,…,…,…,…],

[…,…,…,…,…,…],

[…,…,…,…,…,…]])

>>sigma

**array([10.98, 8.79, 1.04, 1.18e-15, 2.13e-32])**

>>v\_t #右奇异矩阵的转置

matrix([[…,…,…,…,…],

[…,…,…,…,…],

[…,…,…,…,…],

[…,…,…,…,…],

[…,…,…,…,…]])

在矩阵**中奇异值从大到小排列，且具有较快的衰减速度，通常前面*10%*的奇异值之和能够占到全部奇异值之和(矩阵的全部特征)的*99%*以上，即可以使用这*10%*的奇异值近似描述矩阵：**

**；**

要存储一个较大的矩阵**，**通过***SVD***运算只需要存储得到的三个较小的矩阵**，，**即可，极大地减少了存储空间；

### 主成分分析

主成分分析***(Principal Component Analysis)PCA***的作用是将给定数据集***D***

通过线性变换**转换成：，**使得数据集在新线性空间中的各个正交坐标系中拥有最大的方差，**线性变换**就是这样的一组正交坐标系，第1个坐标轴具有最大的方差，第2个坐标轴与第1个坐标轴正交，且拥有最大的方差……

样本属性数据较大的方差利于模型的训练，降低模型偏差提高泛化能力，而主导方差的方向才是数据的有效特征，要想获得数据的有效特征，选择前面***r (r<m)***个坐标轴对原始数据集***D***进行变换：，可得到主导样本方差的***r***个属性即数据集的主成分；

通过变换后新得到的在样本的属性方向上进行了压缩，忽略了方差较小的***m-r***个属性(噪声)，这一线性变换的过程可借助***SVD***实现：

***SVD*: ；**

**令： ；**

**；**

**其中 ；**

**即有 ；**

即：通过将原始数据集组成的矩阵***D***(行索引代表样本，列索引代表属性)进行奇异值分解得到其全部奇异值并排序**，**从中选择前面***r***个奇异值，使得 **，**以对应的左奇异向量构成变换矩阵，并生成对角矩阵：

**；**

进而得到对原数据集的主成分抽取：

**；**

类似于对属性的主成分提取，若原始数据集中存在过多相似的样本，要将其压缩，则可对矩阵进行行压缩：

**；**

观察***SVD*： ；**

**令 得到：**

**，其中： ；**

则有 **；**

从矩阵的奇异值中选取前***r***个后，以其对应的右奇异向量构成矩阵并转置得到变换矩阵**；**

对角矩阵**，**进而完成对原始数据集矩阵的样本提取(行压缩)；

## Apriori算法与FP-Growth

### 关联规则

有事件***A***和***B***，关联规则**的支持度：；**

关联规则**的置信度： ；**

满足最小支持度阈值和最小置信度阈值要求的规则为强规则：

，此时称**存在强关联；**

### Apriori算法

***Apriori***算法用来生成频繁项集，供规则挖掘使用，仅使用最小支持度阈值**作为参数对原始数据集进行搜索：**

* 初始扫描数据集，构建候选项集**，其中*m*表示数据集中包含的所有条目种类数，是单条目集合**，
* 扫描候选项集**，**将支持度低于阈值的项集剔除，得到**支持项集(频繁-1项集)：**

；

* 利用***Apriori***性质：**频繁项集的子集一定是频繁的，而非频繁项集的超集必是非频繁的，**在**频繁-*(k-1)*项集**中寻找**频繁*k*项集：**

1. **，连接中的子集：**

**若 则将子集与合并生成，*t*的数值由外循环决定，连接步生成候选项集**

；

1. 对候选项集**进行剪枝：若存在的*k-1*项子集在频繁-(*k-1)*项集中不存在，则直接将从候选项集中剔除：**

**；**

1. **对所有候选子集计算其支持度，若，则将其从候选项集中剔除，否则加入频繁-*k*项集中；**

***Apriori***算法的问题：产生候选项集时，循环产生的组合过多，没有直接排除不应加入候选项集的组合；计算支持度的时候，需要遍历整个数据集，增大了***I/O***开销；

优化策略：

在逐层搜索生成频繁项集的第***k-1***步，统计中的各个条目出现的次数，将出现次数少于次的条目所在项集删除，从而排除由此条目引起的一定数量的组合，使用了频繁项集性质：频繁项集中的所有条目在其**频繁项集中至少出现*k-1*次(因为**频繁项集的***k-1***阶子集一定是频繁的**)；**

同时对原始数据库***D***进行更新：在第***k***步生成候选项集后，扫描数据库，查看是否所有事务均包含至少一个候选项集中的子集，若不满足，则将此事务移至数据库末尾并加做标记，扫描完成后，从***D***中剔除做标记的事务得到新的数据集，随着***k***逐渐增大，的规模越来越小，减小在扫描事务时的***I/O***开销；

### FP-Growth算法

***FP-Growth***算法***(Frequent Pattern)***首先通过对数据集的两次扫描构造***FP-Tree***：

* 第一遍扫描：遍历数据集，统计各个条目出现的次数，只保留出现次数大于支持度阈值的条目(包含此类条目的项集肯定不是频繁的)，使用一个字典进行存储**HeaderTable={key = name : value = count,…}**，并按照条目出现的频率进行排序；
* 第二遍扫描：读入每个项集并将其添加到一条已存在的路径，若路径不存在，则创建，在将事务添加到树中时，要按照**HeaderTable**中条目出现的频率先对事务中的条目进行排序，并且，相同的事务只在**FP-Tree**中出现一次；

根节点**，**遍历筛选和排序后的事务数据集，向**FP-Tree**中不断添加事务，若树中已存在被添加事务中的条目，则对此条目所在节点进行增**1**操作**TreeNode.increment(1)，否则创建分支；**

**基于FP-Tree挖掘频繁项集：**

* 从**FP-Tree**中提取条件模式基(**Conditional Pattern Base**)：从待查找项所在节点到根节点的全部前缀路径组成的集合，遍历**HeaderTable**中的所有条目，找到其在**FP-Tree**中对应的节点回溯至根节点，记录此路径及其频率(**TreeNode.count**)；
* 创建条件树：以条件模式基为输入数据，首先对单条目项集构建条件树，删除支持度低于阈值**的项集**，生成树；然后递归对2条目项集，多条目项集递归进行构建条件树，直至树中没有节点为止，即找到了最大频繁项集；

## 词频-逆文本词频

**词频-逆文本词频*(Term Frequency – Inverse Document Frequency)TF-IDF***，是在文本向量化后的一种纠正措施：

**文本向量化*word2vec***：统计***N***篇文档**(语料库)**中各个单词在字典中出现的位置及次数，生成与字典相同维度的向量：

**；**

**其中表示在文档中对应字典中的单词出现的次数；**

文本向量化后得到的文本向量中包含的单词出现次数并不能反应单词的重要程度(包含停用词以及其他副词介词等)，因此采用***TF-IDF***指标修正文本向量；

**词频定义 ；**

**逆文本词频定义 ；其中表示出现单词的文档数，为避免出现分母为0的情况，使用拉普拉斯平滑：**

**；**

**则修正文本向量得到：**

**；**

## Page-Rank

***Page-Rank***算法基于网站接入链接的数量和质量假设：

* 在***Web***的图模型中，接入当前页面的链接数量越多，其重要性越高；
* 接入当前页面的源链接页面的重要性越高，其对当前页面重要程度的贡献越高；

### 爬虫与倒排索引

爬虫与倒排索引用于建立资料库***(key-value)***，其中***key***为页面的***url***，***value***为页面的内容，生成***Page-Rank***算法使用的数据集：

* 爬虫***(Crawler)***：从起始页面出发，通过***HTTP***协议获得页面的全部字段，将页面的***url***记录到***key***域中，将其内容记录到***value***域中，并分析内容中包含的链接，使用这些链接指向新的页面，重复上述过程可得到***Web***中的几乎所有的页面；
* 倒排索引***(Inversed Index)***：以关键字为***key***，以包含关键字的页面编号集合作为***value***建立***key-value***数据结构；

### PR迭代更新

假设整个***Web***是一个**强连通**的**图模型**，即从任一页面出发可转移到任意其他页面，转移概率矩阵：

**，其中表示页面*i*是从页面*j*转移而来的概率，且有：**

***；***

初始化每个页面的***PageRank***数值为 **，，转移概率矩阵中的第*i*行表示从各个页面转移到页面*i*的概率，而向量*v*表示各个页面的*Page-Rank*值，因此更新页面的*Page-Rank*为，即：** ；重复上述过程可以收敛于一个稳定的***Page-Rank***向量，即最终页面的排名，在每一步迭代中，对***Page-Rank***向量均有**保证了整个页面迭代更新过程是收敛的**；

而转移概率矩阵**的计算遵循：**

**表示页面*j*转移到页面*i*的概率，若页面*j*中存在*K*个外部链接，称其出度为*K*，这*K*个页面均分页面*j*的转移概率，则页面*j*到页面*i*的转移概率为 ；**

为了克服迭代过程中的***Spider Traps***和稀疏矩阵连续相乘带来的***Page-Rank***向量极度不平滑问题，对向量迭代公式进行修正：

；其中**设置为一个较小的数值，称为心灵转移概率向量；**

### 话题倾向排序

**话题倾向排序*Topic-Sensitive***：先验地得到一个话题集合**𝒯**，对每一个页面进行话题归属划分(典型的分类问题)，可以使用聚类算法，也可以使用***TF-IDF***基于关键词分类，分类后得到多个***Topic***向量，将***Page-Rank***向量迭代公式**变为：**

**，其中话题向量表明页面*k*归属于此话题，表明页面*k*不属于此话题，是向量的范数；**

**而推荐给用户的话题可由用户使用浏览器的*cookie*收集追踪得到，也可以由用户的使用习惯进行偏好分析；**

## 概率图模型

概率模型的框架下，以***X*表示观测变量(输入)**，以***Y*表示预测变量(输出)**，以***R*表示未观测的隐含变量**，判别模型直接计算条件概率，生成模型通过**得到条件概率，**都是为了解决推断问题得到观测数据下对预测变量的条件概率分布**；**

其中，判别模型直接考虑条件概率**；**

**生成模型无法直接获得条件概率，通过贝叶斯公式得到**

**；**

在概率框架中，***EM***算法可以解决含有隐变量的极大似然估计问题，但是**通过概率求和消去隐变量*R*会遭遇组合爆炸和样本稀疏问题**，算法复杂度为**，其中为样本观测特征取值空间，为样本未观测特征取值空间，此外样本的属性之间可能存在较强的耦合关系，因此基于训练样本集预测条件概率分布较困难；**

概率图模型使用有向图或无向图描述属性之间的相关关系，对概率图建模解决推断问题；若变量间存在**显式的因果关系**，使用**有向图模型(贝叶斯网络)**进行描述，若已知变量间存在相关性，但是难于使用显式因果关系描述，此时使用**概率无向图模型(马尔科夫网络)**进行描述；

***HMM*模型与*CRF*模型对比：**

* ***HMM*模型是有向图生成模型，给定模型后，生成状态序列，再由状态序列中的每一时刻状态生成对应的输出观测：**

***；*即输出观测(标记)仅与其所处时刻的状态有关，而此时刻的状态仅与其前一时刻的状态有关，*HMM*沿着时序方向进行；**

* ***CRF*模型是概率无向图判别模型，给定输入状态序列求其对应的输出标记序列的条件概率，而标记不仅与整个输入观测序列有关，还与其它标记有关，当CRF为线性链式结构时：**

**；**

### 隐马尔科夫模型

隐马尔科夫模型**(Hidden Markov Model)HMM**是一种**有向图模型(动态贝叶斯网络)**，用于解决标注问题，作用在时序上，由马尔科夫链**随机生成**不可观测的状态序列，再由这个生成的状态序列产生对应的**(每一个时刻的状态产生一个对应的观测)**观测序列；此时，模型的不可观测状态变量与输出观测变量**不再是分类问题中样本的特征与标记，而是在特定的状态空间和观测空间中的取值，其中，状态空间，观测空间，*HMM***中的马尔科夫链描述了变量之间的依赖关系，即某一时刻***t (t=1,2,…,T)***，观测变量**仅由当前时刻的状态变量决定，而状态变量仅与前一时刻的状态变量有关，整个时序中的变量联合概率分布为：**

；

***HMM***由参数**确定，**其中**为初始状态概率向量，表示*HMM*在初始时刻在状态空间中的概率分布；*A*为状态转移概率矩阵，表示状态空间中各个取值之间进行转移的概率分布；*B*为观测输出概率矩阵，表示由状态空间中取值产生观测空间中取值的概率分布；**

***；***

***；此处的t-1与t不是变量，仅代表从上一时刻到当前时刻的状态转移关系，与时序无关；***

***；此处的t不是变量，仅代表在当前时刻由某一状态产生输出的关系，与实际时序无关；***

***HMM***工作过程：

1. ***t=1***时刻，根据初始状态概率向量从状态空间中选择初始状态；
2. 迭代式：时刻，由当前的状态按输出观测概率矩阵产生对应的观测，并按状态转移概率矩阵产生下一时刻的状态；
3. 在***t=T***时刻，由此时的状态按观测概率矩阵产生输出**；**

***HMM***需要关心的三个问题：

* 概率计算问题：给定模型参数和观测序列求产生当前观测序列的概率**；前向后向算法：**

定义了处于时刻***t***时的**观测序列为且处于状态的概率作为**前向概率**；**

**在初始时刻： 表示从状态输出观测的概率，其中从给定的观测序列中得到；**

**递推式： 表示从状态输出观测的概率，其中当前状态由前一时刻状态依状态转移概率矩阵转移而来，因此需要求和；**

**得到最终时刻T的前向概率后，即可得到；**

定义在时刻***t***的状态为 **，**并由此向后出发得到观测序列的概率为后向概率**；**

**从T时刻出发：；规定在T时刻由状态输出给定的观测的概率为定值1；**

**递推式：；表示由当前状态得到下一时刻状态及其对应的输出观测的概率；由于当前状态转移可以得到多个下一时刻的状态，对应状态转移概率矩阵，因此需要求和；**

**得到最终时刻1的后向概率后，可求；**

**统一前向后向概率，即可得到：**

* 学习问题：给定观测序列估计模型参数，使得观测概率产生的概率最大，即模型参数的极大似然估计问题；***Baum-Welch*算法**：以状态序列**作为隐含变量，则学习问题变为，即含有隐变量的极大似然估计，使用EM算法，构造，其中；其中为常数，将*Q*函数变形为，其中，是*HMM*模型参数的当前估计值；则有：**

**；**

**逐项最大化求解，得到：；**

**；**

**给定观测序列和初始参数值得到初始模型；**

**递推式：依据上一步得到的参数估计计算，，，；**

**更新参数**

**；**

**；**

**直至达到最大迭代轮数为止；**

* 预测(解码)问题：给定模型参数和观测序列求最有可能产生当前观测序列的状态序列，即使得条件概率最大的状态序列**；维特比算法：使用动态规划解决*HMM*的预测问题，从时刻*t=1*开始，递推计算在时刻*t*状态为的各条部分路径的最大概率：**

***；***

**由此得到最优路径的递推式：**

**；**

**而在时刻*t*状态为的所有单个路径中拥有最大概率的路径的第t-1个节点为：**

**；**

**表示当前时刻*t*由前一时刻*t-1*转移得到状态并输出观测的最大概率，而此最大概率对应的前一时刻*t-1*所处的状态就是*t*时刻的最优路径节点；**

### 马尔科夫随机场

马尔科夫网络(无向图模型)：满足成对马尔科夫性，局部马尔科夫性和全局马尔科夫性的无向图模型；

**；其中为概率归一化因子，表示对所有随机变量的求和；**

**成对马尔科夫性：，其中与两个节点随机变量没有边连接，为图中其他节点；**

**局部马尔科夫性：，其中表示节点的概率分布，表示与直接相连的节点集合变量组，节点表示其他变量组；**

**全局马尔科夫性：，其中变量组*A*和变量组*B*在无向图中被变量组C分开；**

### 条件随机场

**条件随机场*(Conditional Random Field )CRF***是一种判别式无向图模型，是指**给定随机变量*X*的条件下，随机变量*Y*的马尔科夫随机场**；

**将输入变量*X*作为观测序列，则输出变量*Y*是标记序列(对应于*HMM*的状态序列*R*)，*CRF*要解决的两个问题：**

* **利用训练数据集通过极大似然估计得到条件概率；**
* **给定输入观测序列，求使得条件概率最大的输出序列；**

考虑线性链条件随机场，**为输入观测序列，为输出标记序列，则条件概率构成条件随机场，即有**

；

即标记**仅与条件随机场中其相邻节点对应的随机变量和以及输入观测变量有关；**

参数化形式：**；**

**其中;是转移特征函数，依赖于当前位置与前一个位置节点，以及输入观测序列，是其对应的权值；**

**;是状态特征函数，依赖于当前位置和输入观测序列，为其对应的权值；**

**概率归一化因子，将指数形式转换为规范化概率；**

***CRF*中同一个位置*i (i=1,2,…,n)*的随机变量对应多个特征函数，而特征函数在每一个位置均有定义，因此先将特征函数统一定义称分段形式，可以得到随机变量在*CRF*中的：**

**；**

**；**

**将特征函数在各个位置求和：**

**；**

**则将随机变量的条件概率表示为：**

**；**

**；**

**定义向量；**

**；**

**则有*CRF*的内积形式：**

**；**

**；**

**若对特征函数每一个位置上的特征函数求和：**

**;**

**在位置*i*上随机变量的取值空间为𝒴，，则可得到在位置i上的矩阵：**

**；**

**其中，；**

**则有条件概率的矩阵形式：**

**；**

**;**

**即规范化因子是以*start*为起点，以*stop*为终点状态的所有路径的非规范化概率之和；**

同***HMM***类似，***CRF***也要解决三类问题：

* 概率计算问题：计算状态**特征条件概率和转移特征条件概率，以前向概率表示在位置*i*处状态为 ，下标为(*1,2,…,i-1*)的前部分序列的非规范化概率：**

**；**

**；**

**后向概率表示在位置i处状态为 ，下标为(*i+1,i+2,…,n+1*)的后部分序列的非规范化概率：**

**；**

**；**

**由于随机变量的取值空间为𝒴，，所以和都是m阶向量，*i=1,2,…,n*；**

**则可得到；**

**；**

**其中为概率归一化因子；**

**特征函数关于条件概率分布的期望：**

**；**

**特征函数关于联合概率分布的期望：**

**；**

**；**

* 学习问题：定义在时序上的**对数线性模型**，可使用**改进的迭代尺度法*IIS*，梯度下降法，牛顿法、拟牛顿法求解；**

**学习问题的输入是转移特征函数，状态特征函数，以及输入观测序列与输出标记序列的经验分布；**

**输出对权值向量的估计以及对应的模型；**

训练数据集**的对数似然函数为：**

**;**

**代入*CRF*条件概率的向量表示，得到：**

**；**

**其中；**

**；不断优化对数似然函数改变量的下界，极大化对数似然函数，可以得到转移特征与状态特征的更新方程为：**

**；**

**；**

**；**

其中，状态特征与转移特征函数依经验分布的期望由下式得到：

**；**

**；**

**设置权值更新增量，解特征方程，求解相应的，并更新参数：**

**；**

直至得到收敛的更新增量序列**；**

不同于迭代尺度法对各个特征函数求和，拟牛顿法对序列中所有位置求和：

**；**

**其中，；**

**构造的对数似然函数：，整理得到：**

**；**

将对数似然函数极大化转换成目标函数最优化问题即**：**

**；**

**求目标函数对的梯度：**

**；**

***CRF*模型的*BFGS***算法输入为各个位置上的特征函数**(对k求和)，**以及训练数据的经验分布**；**输出为估计参数值以及对应的模型**；**

选定初始点**，**取为正定对称矩阵**，t=0；**

**；求解向量，其中；**

**求，满足条件；**

**更新 ，计算，更新矩阵**

**；**

**，；**

***，***重复上述过程迭代更新参数值，直至到达最大迭代轮数***T***或者在计算过程中使得梯度**；**

* 预测问题，***CRF***的预测问题是指给定模型以及输入观测序列**，**求解使得条件概率最大的输出观测序列**，是典型的标注问题；**

**；**再由概率归一化因子与求最大化无关性和指数函数的单调性可得到：

**；*即求解非规范化概率最大的输出序列；***

**;**

**;**

**;**

**可得到；**

**对将原问题变形为：**

**；**

**首先求位置处对应的标记的非规范化概率：**

**；**

**递推式得到最大非规范化概率及其对应的最佳路径为：**

**；**

**；**

**直至处终止，此时得到全路径最大非规范化概率为：**

**；**

**；**

**回溯得到最优的全路径：**

**；**

**得到最优路径 ；**

### 隐狄利克雷分配模型

***LDA(Latent Dirichlet Allocation)***是一种话题模型，话题模型是生成式有向图模型；

长于处理离散型数据的词袋***(bag of words)***表示：

***Document***对应于样本**；**

***Word***对应于样本的特征**；**

***Topic***对应于样本的分类，具体表现为一个概念，以及在这个概念下出现频率较高的一系列词语**；**

**设有词典(*dictionary*)，**其长度为***N*，*T***篇文档可以表示为***T***个***N***维向量，数据集可表示为**：，**表示第***t***篇文档的第***i***个词语出现的频率(词频)**，；**

使用***K***个***N***维向量表示话题**：，**表示话题***k***中的第***j***个词语的词频**，；**

**由以*η*为参数的*Dirichlet*分布得到：**

**；**

***LDA***认为一篇文档中包含多个话题，以向量表示在文档***t***中包含的话题分布**，**其中表示话题***k***在文档***t***中所占比例，**；**

* 根据参数为的***Dirichlet***分布随机采样一个话题分布**：**

**；**

* 生成文档中的***N***个词语：

根据向量进行话题指派，得到文档***t***中各个词语的所属的话题 **；**

**再根据指派话题对应的分布向量依*Dirichlet*分布随机采样生成词语；**

文档***t***以不同的比例包含***K***个话题，话题由***Dirichlet***分布随机采样得到，文档中的***N***个词语分属于不同的话题；

文档词频**可由文档词频统计得到，是唯一的可观测变量；**

**依赖于对词语的话题指派以及话题词频向量；**

**话题指派依赖于**话题分布**；**

话题分布**依赖于*Dirichlet***分布及其参数**；**

词频统计**依赖于参数；**

**则得到*LDA*的概率分布为：**

**；**

**其中，文档*t*中的话题先验分布：**

**；*由此产生每个词语的话题指派* ；**

**其中话题*k*中的词频分布：**

**；**

由话题词频先验分布和**话题先验分布产生的话题指派共同产生文档词频 ；**

**给定数据集，*LDA*模型参数使用极大似然估计法求解：**

**；**

## 从伯努利到狄利克雷分布

### 伯努利分布

**伯努利分布(*Bernoulli Distribution*)**是指对二元随机变量进行一次伯努利试验(***Bernoulli Trial***)得到的随机变量分布律：

**；其中随机变量，；**

伯努利分布描述了随机变量的分布；

伯努利分布的均值为：

**；**

方差为：

**；**

伯努利分布的参数估计方法：

基于给定样本数据集**，估计模型参数，使用极大似然估计，；**

**构造对数似然函数；**

**对变量求导，得到；**

**令，得到 作为对参数的估计；**

### 二项分布

伯努利分布考虑的是**随机变量，在伯努利试验中的分布律，**即概率**，而二项分布(*Binomial Distribution*)考虑在数据集中，随机变量取值为1的个数*m*的概率分布，其中：**

**；**

**二项分布的均值求解转换成如下问题描述：*N*次伯努利试验(相互独立)出现的次数；**

**方差可看做*N*次伯努利试验(相互独立)的随机变量方差求和；**

***；独立事件加和的均值等于均值的加和，加和的方差等于方差的加和；***

**方差为：**

**；**

**对参数的估计是数据集中所占的比例 ；**

### *Beta*分布

极大似然估计的局限性：当数据集的样本量较少时，使用极大似然估计易发生过拟合，即对参数的估计容易受到样本波动的影响；

由贝叶斯公式**得到，即后验概率正比于似然与先验概率的乘积，先验概率在数据集给定的情况下可认为是常数：；**

由于**，选择*Beta*分布：**

**；其中*Gamma*函数：**

**；**

***Beta***分布的**期望为： ；方差为： ；**

**可以得到后验概率：**

**；**

**即有，令，再将其进行概率归一化得到拥有新参数的*Beta*分布：**

**；*先验函数关于似然函数共轭*；**

**贝叶斯估计与极大似然估计对比：在数据集规模有限的情况下，参数的后验均值比其先验分布给出的均值大，比其极大似然估计均值小，当数据集规模趋近于无穷大时，二者得到相同的结果；**

### 多项式分布

不同于伯努利分布和二项分布产生于伯努利试验，多项式分布***(Multinomial Distribution)***描述了随机变量***X***的取值空间 即**的属性拥有多种取值的情况，使用*one-hot*表示方法：，且有，**使用表示**的概率，则有 表示随机变量的概率分布；**

**；**

**；**

给定数据集**，似然函数为：**

**；**

**构造最优化问题：**

**；**

**；**

求解得到**；**

**此时，考虑变量在数据集和参数下的概率分布就得到多项式分布：**

**；**

**；且满足限制条件 ；**

### *Dirichlet*分布

***Beta***分布作为二项分布的参数的先验分布，与似然函数相乘得到其后验分布，以解决极大似然估计法在有限样本量的条件下的过拟合问题**；**

***Dirichlet***分布则是作为多项分布参数**的先验分布，**观察多项分布 **可以得到其共轭先验具有以下形式：**

**；概率归一化后得到：**

**；**

**类似于二项分布，参数的后验分布可以由其先验分布与似然做乘积得到： ；再进行概率归一化得到：**

**；仍然是*Dirichlet*分布，其中，将参数看做是的观测数；**

## 常用的数学trick

* **在确定最大熵模型参数更新对数似然函数增量的下界时用到：**

**；**

* ***Jensen*不等式：**

**；**

***Jensen*不等式是凸函数性质从二维到多维的推广：**

**；**

**若有参数序列 ，满足，则有**

**；**

**即；**

# Tensorflow与深度学习基础

## 神经网络基本框架

### 神经网络训练

无论神经网络的结构如何，训练过程总可以分为三个步骤进行：

* **定义神经网络结构和前向传播的输出结果**；

1. 定义连接权重weight = tf.Variable(tf.random\_normal(shape, stddev, seed))和偏置量bias = tf.Variable(tf.constant(const, shape))；
2. 定义数据集输入特征和样本标记x = tf.placeholder(tf.float32, shape, name = ‘x-input’)

y\_ = tf.placeholder(tf.float32, shape, name = ‘y-input’ )

1. 定义前向传导过程，设计网络结构；

**网络结构和前向传播通常在函数inference(…)中定义；**

* **定义损失函数，选择反向传播优化算法**：

1. 定义损失函数 cross\_entropy = … ;

交叉熵定义以及损失函数的数学解释见[常用的损失函数](#_常用的损失函数)；

通常由**cross\_entropy =**

**tf.nn.softmax\_cross\_entropy\_with\_logits(logits=y, labels=y\_)或者**

**cross\_entropy =**

**tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=y, labels=argmax(y\_, 1))得到的是整个batch的交叉熵，还需要对其取平均：**

**cross\_entropy\_mean = tf.reduce\_mean(cross\_entropy)并计入正则化项得到最终的损失：**

**loss = cross\_entropy\_mean + regularization**

1. 定义反向传播算法：

train\_step = tf.train.AdamOptimizer(learning\_rate).minimize(cross\_entropy)

损失函数的选择见[Tensorflow中的损失函数](#_Tensorflow中的损失函数)；

* **生成会话(tf.Session())，在训练数据上反复运行反向传播优化算法**：

with tf.Session() as sess:

init\_op = [tf.global\_variables\_initilaizer(), tf.local\_varibales\_initializer()]

sess.run(init\_op)

#…

for i in range(STEPS):

sess.run(train\_op, feed\_dict={x: x\_batch, y\_: y\_batch})

…

if i % ITERS == 0:

total\_cross\_entropy = sess.run(cross\_entropy, feed\_dict={x: X, y\_: Y})

**损失函数定义、优化算法选择以及训练步骤的执行通常放置在*train(…)*中实现；**

**训练过程中数据的输入：**

**xs, ys = mnist.train.next\_batch(BATCH\_SIZE)**

**sess.run(train\_op, feed\_dict={x:xs, y\_: ys})**

**…**

**mnist = input\_data.read\_data\_sets(path, one\_hot=True)**

**train(mnist)**

**验证数据输入：**

**validate\_feed = {x: mnist.validation.images, y\_: minist.validation.labels}**

**…**

**validate\_accuracy = sess.run(accuracy, feed\_dict=validate\_feed)**

**通常神经网络的训练都是一次输入一个Batch的样本，因此，定义输入时：**

**x = tf.placeholder(tf.float32, shape=[BATCH\_SIZE, input\_nodes], name=’x-input’)**

**y\_ = tf.placeholder(tf.float32, shape=[BATCH\_SIZE, output\_nodes], name=’y-input’)**

### 神经网络优化

神经网络的优化过程分成两个阶段：

* **训练数据集按Batch输入*x*及其样本标记*y\_*，前向传播得到*y*，比较*logits* *y和labels y\_*的差距，即损失函数(Cross Entropy，MSE等)；**
* **使用Mini-Batch Gradient Descent方法计算输入样本Batch的损失函数对每个参数的梯度，并沿着负梯度方向按学习速率*η*更新参数，**[**局部最小与全局最小**](#_局部最小与全局最小)**；**

除去选择损失函数和优化算法之外，在神经网络的训练中还采取以下措施：

* 学习速率衰减：

**decayed\_learning\_rate =**

**learning\_rate \* decay\_rate ^ (global\_step/decay\_steps)**

学习速率指数衰减策略：**，**其中为基础学习速率***(LEARNING\_RATE\_BASE)***，为学习率衰减率***(LEARNING\_RATE\_DECAY)***，**，**为当前训练的迭代次数***(global steps)***，为预设进行衰减的迭代次数***(decay steps)***，通常设定每训练一个***Batch***数据就进行一次衰减，因此使用tf.train.exponential\_decay()使得学习率以阶梯状下降：

**learning\_rate = tf.train.exponential\_decay(LEARING\_RATE\_BASE, global\_step, num\_examples/BATCH\_SIZE, LEARNING\_RATE\_DECAY, staircase=True)**

**同时，在使用优化算法调整权值时，要将全局变量global\_step传入，以使用更新的学习率：**

**train\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss\_fcuntion, global\_step = global\_step)**

***Adagrad/Adadelta/Adam*** 在解决梯度下降法易陷入局部最小问题时，均对学习速率进行了正则化约束，实现学习速率的衰减，见[局部最小与全局最小](#_局部最小与全局最小)；

* 连接权值***Dropout***：训练时随机让一些隐含节点不工作，即丢弃其连接权值，使得参数更新不再依赖于固有的节点间的连接关系，**在一定程度上避免了特征只有在其他某个特征才起作用的情况；**

而对每次输入的样本***Batch***，相当于使用具有不同结构的模型进行训练，类似于***Bagging*，利于提高模型的泛化性能；**

每个***Batch***只针对样本的一部分隐含特征进行训练，加入样本的属性扰动，提高了数据集方差，以***Naïve Bayes***的角度来看，等于加入属性间的独立性假设：

**lstm\_cell =**

**tf.nn.rnn\_cell.DropoutWrapper(lstm\_cell, output\_keep\_prob=KEEP\_PROB)**

* 正则化***Regularizer***：为避免过拟合，神经网络训练优化除了要极小化经验损失***(Cross Entropy，MSE等)***，还要考虑结构损失，既可以使用***L1***正则化(倾向于得到稀疏解)，也可以使用***L2***正则化(倾向于得到平滑解)：

loss\_function =

tf.reduce\_mean(tf.square(y-y\_)

+ **tf.contrib.layers.l2\_regularizer(lambda)(weights))**

**正则化项**定义通常与网络结构定义部分放在一起，用以**直接计算网络权值的正则化损失**，而损失函数通常在***train\_step***之前定义，二者可能不在同一个源文件中，当网络结构较复杂时，直接将正则化项加入损失函数会导致损失函数定义过长，代码可读性较差，因此，将正则化项加入集合中：

**tf.add\_to\_collection(‘losses’, tf.contrib.layers.l2\_regularizer(REGULARIZATION\_RATE)(layer\_weights))**

**...**

**loss = cross\_entropy + tf.add\_n(tf.get\_collection(‘losses’))**

计算最终的损失需要考虑选择的经验损失以及神经网络各个层的正则化损失，正则化损失的数学解释见[正则化](#_正则化)；

* 滑动平均模型是指在最后的模型选择时，倾向于选择所有待选模型的参数均值组成的模型，此模型往往比最后一次迭代得到的模型以及在验证数据集上表现最好的模型的性能更优；

；其中**为影子变量*(shadow variable)*，全程跟随变量值更新而变化，为衰减率，越大，影子变量越倾向于接近更新前的数值，越小，则越接近更新后的数值，此时参数波动较大，模型不够健壮，通常选择一个较大的数值(0.999/0.9999)，在Tensorflow中通过传入参数num\_updates可以得到一个随训练过程调整的衰减率：，其中为当前迭代的轮数；**

对所有的变量应用滑动平均模型的实现需要指定衰减速率并应用于所有变量：

**variable\_averages =**

**tf.train.ExponentialMovingAverage(MOVING\_AVERAGE\_DECAY, global\_step)**

**variable\_averages\_op = variable\_averages.apply(tf.trainable\_variables())**

**with tf.control\_dependencies([train\_step]):**

**train\_op = tf.group(variable\_averages\_op)**

在使用优化算法更新完参数后，应用滑动平均模型得到参数的影子变量，可以随模型参数和优化器参数保存到***checkpoint(ckpt)***文件中，实现持久化，也可以从中恢复使用：

**saver = tf.train.Saver() # 保存模型**

**…**

**moving\_average\_variables\_to\_restore = # 指定要恢复的参数为**

**variable\_averages.variables\_to\_restore() # ExponentialMovingAverage**

**saver = tf.train.Saver(moving\_average\_variables\_to\_restore)**

**saver.restore(sess, save\_path) # 从ckpt中恢复参数**

## 全连接神经网络

### 激活函数

为感知机***(Perceptron)***加入多层和非线性特性使得神经网络表达能力更加丰富，将权值向量与单元输入的乘积与偏置量求和并输入非线性单元完成非线性化激活，常用的非线性激活函数有：

* **限制线性单元*(Restricted Linear Unit)ReLU***：**，tf.nn.relu()；**
* **Sigmoid函数： ，tf.nn.sigmoid()；**
* **正切函数： ，tf.nn.tanh()；**

在多分类问题中，对输出节点做**tf.argmax(y, 1)**处理即可得到待分类样本的标签信息；

### 前向传播

全连接神经网络的前向传播过程即将当前层输入***tensor***与连接权重相乘再加偏置后进行非线性化作为输出给下一层的过程：

layer1 = tf.nn.relu(tf.matmul**(**input\_tensor, fc1\_weights**)**+fc1\_biases)

layer2 = tf.matmul(layer1, fc2\_weights)+fc2\_biases

预测时使用的是输出层不同的节点之间的相对大小，因此最终的输出层可以不加入非线性激活函数和***softmax***；

若使用了滑动平均模型，则需要使用***MovingAverage***后的参数值：

layer1 =

tf.nn.relu(tf.matmul**(**input\_tensor, average\_class.average(fc1\_weights)**)**+average\_class.average(fc1\_biases))

layer2 =

tf.matmul(average\_class.average(layer1), average\_class.average(fc2\_weights))+fc2\_biases

## 卷积神经网络

### 常用的图像数据集

图像分类问题常用的数据集：

* ***Cifar***
* ***MNIST***
* ***ImageNet***

**分类问题性能度量标准：TOP-N正确率，即预测给出的前N个答案中有一个是正确答案的概率；**

### CNN

***CNN***的标准结构：***Input Layer1 🡪 Convolutional1 Layer2 🡪 Pool1 Layer3 🡪 Convolutional2 Layer4 🡪 Pool2 Layer5 🡪 Full Connected1 Layer6 🡪 Full Connected2 Layer7 🡪 Softmax Layer8***

* 输入层：图片的像素矩阵 **，**其中***L***为图片的高度即纵向上像素的数量**，*W***为宽度即横向上像素的数量，***D***为深度即图片的色彩通道，黑白图片深度为***D=1*，RGB**色彩模式下，图片深度为***D=3*；**
* 卷积层：通过卷积核**(过滤器)*filter***将输入矩阵(可能是输入层的原始像素矩阵，也可能是池化层输出)的子节点矩阵转换为一个**单位节点矩阵：**

**；**即对输入层的子节点矩阵中的元素按照***p***组权重和偏置求加和并进行非线性激活，即可得到卷积层单位节点矩阵的***p***个节点数值**，**其中，为卷积核的尺寸，而卷积的过程就是将卷积核从输入矩阵的左上角-右上角-左下角-右下角逐次移动，而在深度方向上不能移动，每次移动的步长决定了卷积输出矩阵的尺寸；

当使用全***0***填充***(zero-padding)***时：

**；**

**；**

不使用填充时：

**；**

**；**

代码实现如下：

#**卷积核尺寸为5\*5，当前层深度为3，卷积核深度为16**

filter\_weight = tf.get\_variable(‘weight’,

**[5,5,3,16]**, initializer=tf.truncated\_normal\_initializer(stddev=0.1))

biases = tf.get\_variable(‘biases’, [16], initializer=tf.constant\_initializer(0.1))

#输入input为4维矩阵**[index, length, width, deepth]**，**第0维代表图片在**#***Batch*中的索引;**

#卷积核作用不能跨越图片样本也不能跨越深度，因此strides的第0维与第#3维只能为1

#**padding=’SAME’** 表示全0填充，**padding=’VALID’**不填充

**conv = tf.nn.conv2d(input, filter\_weight, strides=[1,1,1,1], padding=’SAME’)**

**#为所有节点加上相同的偏置项**

**bias = tf.nn.bias\_add(conv, biases)**

activated\_conv = tf.nn.relu(bias)

* **池化层：**池化层作用与卷积层的输出之上，进一步缩小矩阵的尺寸，减少连接的个数，常使用的池化层有最大池化层***(max pooling)***和平均池化层***(average pooling)***，池化核(过滤器)filter可以在深度方向上起作用：

**#ksize=[1,3,3,1]第0维与第3维必须为1，不可跨样本和深度作用；**

**#strides=[1,2,2,1]第0维与第3为必须为1，不可减少样本个数和深度；**

pool = tf.nn.max\_pool(activated\_conv,

**ksize=[1,3,3,1], strides=[1,2,2,1]**, padding=’SAME’)

* **全连接层：**卷积神经网络最后通常要加入两层或以上全连接网络，用于完成分类任务；
* **Softmax层：Softmax层最终将分类结果输出节点上的数值归一化处理为概率分布；**

### LeNet-5

***LeNet-5*的七层结构：*Input Layer*** 🡪 ***Convolutional1 Layer*** 🡪 ***Pool1 Layer*** 🡪 ***Convolutional2 Layer*** 🡪 ***Pool2 Layer*** 🡪 ***Full Connected1 Layer*** 🡪 ***Full Connected2 Layer*** 🡪 ***Full Connected3 Layer***

数据集：***MNIST***

* ***Input Layer：MNIST***图片维度**32\*32\*1**
* ***Convolutional1 Layer：***

**filter\_size=[5, 5, 1, 6]**

**strides=[1,1,1,1]**

**PADDING=’VALID’**

**输出尺寸=28\*28\*6**

* ***Pool1 Layer：***

**filter\_size=[1, 2, 2, 1]**

**strides=[1,2,2,1]**

**PADDING=’VALID’**

**输出尺寸=14\*14\*6**

* ***Convolutional2 Layer：***

**filter\_size=[5, 5, 6, 16]**

**strides=[1,1,1,1]**

**PADDING=’VALID’**

**输出尺寸=10\*10\*16**

* ***Pool2 Layer***

**filter\_size=[1, 2, 2, 1]**

**strides=[1,2,2,1]**

**PADDING=’VALID’**

**输出尺寸=5\*5\*16**

* ***Full Connected1 Layer：***输入矩阵尺寸=**[batch\_size,** **5, 5, 16]，将第1~3维拼接称一个向量，即代表一个样本数据，即输入节点个数为5\*5\*16=400，输出节点数120；**
* ***Full Connected2 Layer：***输入节点数120，输出节点数84；
* ***Full Connected3 Layer：***输入节点数84，输出节点数10，完成分类任务；
* **分类正确率：测试数据集上可达99.4%**

调整输入矩阵以及输入训练数据：

x = tf.placeholder(tf.float32, [BATCH\_SIZE, IMAGE\_SIZE, IMAGE\_SIZE,

NUM\_CHANNELS], name=’x-input’)

reshaped\_xs = np.reshape(xs, (BATCH\_SIZE, IMAGE\_SIZE, IMAGE\_SIZE,

NUM\_CHANNELS))

网络结构设计时，使用上下文管理器隔离命名空间，以便获取变量：

with tf.variable\_scope(‘layer1-cov1’):

conv1\_weights = tf.get\_variable(“weight”, **[CONV1\_SIZE, CONV1\_SIZE,**

**CONV1\_CHANNELS, CONV1\_DEEPTH]**,

initializer=tf.truncated\_normal\_initializer(stddev=0.1))

conv1\_biases = tf.get\_variable(“bias”, [CONV1\_DEEPTH],

initializer=tf.constant\_initializer(0.0))

conv1 = tf.nn.conv2d(input\_tensor, conv1\_weight, strides=[1,1,1,1],

padding=’SAME’)

relu1 = tf.nn.relu(tf.nn.bias\_add(conv1, conv1\_biases))

with tf.name\_scope(‘layer2-pool1’):

pool1 = tf.nn.max\_pool(relu1, ksize=[1,2,2,1], strides=[1,2,2,1], padding=’SAME’)

在***Pool2 Layer***输出给全连接层时，要进行矩阵形状变换：

**pool\_shape = pool2.get\_shape().as\_list()**

**nodes = pool\_shape[1]\*pool\_shape[2]\*pool\_shape[3]**

**reshaped = tf.reshape(pool2, [pool\_shape[0], nodes])**

**卷积神经网络模型中正则化损失仅考虑全连接层的权重，且dropout仅考虑除最终输出层之外的全连接层的权重：**

with tf.variable\_scope(‘layer5-fc1’):

fc1\_weights = tf.get\_variable(“weight”, [nodes, FC1\_SIZE],

initializer=tf.truncated\_normal\_initializer(stddev=0.1))

if regularizer != None:

tf.add\_to\_collection(‘losses’, regularizer(fc1\_weights))

fc1\_biases = tf.get\_variable(“bias”, [FC1\_SIZE],

initializer=tf.constant\_initializer(0.1))

fc1 = tf.nn.relu(tf.matmul(reshaped, fc1\_weights) + fc1\_biases)

if train:

fc1 = tf.nn.dropout(fc1, DROPOUT\_KEEP\_PROB)

**…**

**#全连接层不经过Softmax处理直接给出结果，由**

**#tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits()处理**

logit = tf.matmul(fc1, fc2\_weights) + fc2\_biases

return logit

**通常选择卷积核尺寸不超过5，一般为3或1，深度通常采用逐层递增形式，卷积层步长通常设为1；**

**池化层使用最普遍的是max\_pool，尺寸通常为2或3，步长选择2或3；**

**完整源码见“./src/LeNet-5”；**

### 迁移学习

将已保留的***Inception-v3***模型中的所有卷积层看做对图片特征的提取，输出一个表达能力更强的特征向量，替换最后的全连接层，用于完成新的分类任务，将卷积层之前的网络层称为瓶颈层***(bottleneck)***；

* ***Inception-v3***模型由***Inception***模块组成，与***LeNet-5***模型不同，***Inception***模块采用并联多卷积核做卷积运算然后拼接成输出矩阵的方式，并联多个卷积核尺寸不同但都使用全***0***填充，因此卷积输出的矩阵尺寸***(length\*width)***相同，模块输出仅需要在**深度方向(维度3)**上进行拼接：

branch\_1 = tf.concat(**3**, [slim.conv2d(branch\_1, 384, **[1,3]**, scope=’Conv2d\_0b\_1x3’),

slim.conv2d (branch\_1, 384, **[3,1]**, scope=’Conv2d\_0b\_3x1’)])

实现较复杂的卷积层，可使用tensorflow-slim工具减少代码数量：

**net = slim.conv2d(input, 32, [3,3]) #必填参数input输入矩阵，卷积核深度，卷积**

**#核尺寸**

其他可选参数可以使用**slim.arg\_scope()**函数设置：

**with slim.arg\_scope([slim.conv2d, slim.max\_pool2d, slim.avg\_pool2d], stride=1,**

**padding=’SAME’)**

* 数据集：***flower\_photos***
* 模型：***classify\_image\_graph\_def.pb***
* **tensor.name()获得所需的迁移模型的图片样本输入张量**

**’DecodeJpeg/contents:0’和瓶颈层输出结果张量’pool\_3/\_reshape:0’，将从输入层到瓶颈层之间的模型视作黑盒，在新模型训练时直接使用；**

* 实现：

1. 划分数据集：留出法，遍历**’./data/flower\_photos’**下所有子路径，得到所有图片样本的不带路径的文件名称***filename.extension***，并以子路径名称作为样本的**label**，调用np.random.randint(100)生成随机数，若小于

**VALIDATION\_PERCENTAGE**，则放入验证集，若大于

**VALIDATION\_PERCENTAGE**而小于

**VALIDATION\_PERCENTAGE+TESTING\_PERCENTAGE**则放入测试集，否则放入训练集，最终使用一个字典保存当前分类的数据集：

**image\_lists[label\_name]={‘dir’: dir\_name, #子路径**

**‘training’: training\_images, #训练集**

**‘validation’: validation\_images, #验证集**

**‘testing’: testing\_images} #测试集**

1. 在训练时，从训练集中随机获取图片get\_random\_cached\_bottlenecks(**sess**, n\_classes, image\_lists, how\_many, **category**, jpeg\_data\_tensor,

bottleneck\_tensor)，得到从各个label中随机选取的图片构成的训练集，**how\_many=BATCH\_SIZE**，**category=’training’，**

jpeg\_data\_tensor, bottleneck\_tensor = tf.import\_graph\_def(graph\_def, return\_elements=[JPEG\_DATA\_TENSOR\_NAME, BOTTLENECK\_TENSOR\_NAME])

…

#***classify\_image\_graph\_def.pb***

with gfile.FastGFile(os.path.join(MODEL\_PATH, MODEL\_FILE)) as f:

graph\_def = tf.GraphDef()

graph\_def.ParseFromString(f.read())

若**./tmp/bottleneck/label\_name**目录下没有待选择图片的特征向量文件，则为其创建：

bottleneck\_values = **sess**.run(bottleneck\_tensor, {**image\_data\_tensor**:

image\_data}) #image\_data\_tensor即已获得的jpeg\_data\_tensor

**为输入后面的全连接层，应当将其拉伸成一维向量：**

**bottleneck\_values = np.squeeze(bottleneck\_values)**

**转换成字符串格式，以逗号间隔：**

bottleneck\_string = ‘,’.join(str(x) for x in bottleneck\_values)

with open(bottleneck\_path, ‘w’) as bottleneck\_file:

bottleneck\_file.write(bottleneck\_string)

同时生成一个***n\_classes*维0向量ground\_truth，置对应的label\_index为1，以ground\_truth作为其类别标签；**

**bottlenecks.append(bottleneck\_values) 🡪 train\_bottlenecks**

**ground\_truths.append(ground\_truth) 🡪 train\_ground\_truths**

得到训练数据的一个***Batch***；

1. 将图片的**bottleneck**输入全连接层实现最终分类：

bottleneck\_input = tf.placeholder(tf.float32, [None,

BOTTLENECK\_TENSOR\_SIZE], name=’BottleneckInput’)

ground\_truth\_input = tf.placeholder(tf.float32, [None, n\_classes],

name=’GroundtruthInput’)

with tf.name\_scope(‘full\_connected\_layer’):

weights = tf.get\_variable(“weight”, [BOTTLENECK\_TENSOR\_SIZE,

n\_classes], initializer=tf.truncated\_normal\_initializer(stddev=0.001))

biases = tf.get\_variable(“biases”, [n\_classes],

initializer=tf.zeros\_initializer())

logits = tf.matmul(bottleneck\_input, weights) + biases

#经过Softmax层处理

final\_tensor = tf.nn.softmax(logits)

#损失函数为交叉熵

cross\_entropy = tf.sparse\_softmax\_entropy\_with\_logits(logits=logits,

labels=ground\_truth\_input)

cross\_entropy\_mean = tf.reduce\_mean(cross\_entropy)

#**模型训练op**

train\_step =

tf.GradientDescentOptimizer(LEARNING\_RATE).minimize(cross\_entroy\_mean)

#模型验证op

with tf.name\_scope(‘evaluation’):

correct\_predication = tf.equal(tf.argmax(final\_tensor, 1),

tf.argmax(ground\_truth\_input, 1))

validation\_step = tf.reduce\_mean(correct\_predication, tf.float32)

#模型训练

sess.run(train\_op, feed\_dict={bottleneck\_input: **train\_bottlenecks**,

ground\_truth\_input: **train\_ground\_truths**})

#模型验证

for i % 100 == 0 or i + 1 == TOTAL\_STEPS:

validation\_bottlenecks, validation\_ground\_truths =

get\_random\_cached\_bottlenecks(sess, n\_classes, image\_lists, BATCH, ‘validation’, jpeg\_data\_tensor, bottleneck\_tensor)

validation\_accuracy = sess.run(validation\_step,

feed\_dict={ bottleneck\_input: **validation\_bottlenecks**,

ground\_truth\_input: **validation\_ground\_truths** })

完整源码见“./src/imgration\_learning”

## 循环神经网络

### RNN

循环神经网络***(Recurrent Neural Network)CNN***用于解决序列化数据的预测问题，处于同一层的节点之间也存在连接，网络的当前状态由当前时刻的输入和前一时刻的状态共同决定：**，基本构成单元叫做循环体；**

**输入序列： ，为*t*时刻的输入向量，输入权值矩阵为： ，对于不同时刻的输入，其权值是共享的，即权值矩阵保持不变；**

***t-1*时刻的网络状态提供给当前时刻的网络，作为输入的一部分：**

**，状态权值矩阵为 ；**

***t*时刻的网络状态既要输出给*t+1*时刻，也要计算生成当前时刻的输出：**

**；为激活函数；**

**为输出偏置；**

对于每一个输入的时序序列即输入样本，时间***t***从***0***到***T***的过程(一个样本序列)，权值矩阵**，，以及偏置向量和均保持不变，将看做序列的一个属性，因此*RNN*计算损失函数时要考量完整样本的损失，即整个时序上的输出的损失函数，在实际工程应用过程中，超长的序列会带来梯度消散(Gradient Vanishing)或梯度爆炸(Gradient Explosion)问题，因此需要将序列进行截断处理，详见**[**梯度消散/爆炸与序列截断**](#_梯度消散/爆炸与序列截断)**；**

***RNN***只能解决在序列中**预测位置**距离**相关信息**较近的问题，若二者间的间隔增大时，***RNN***会丧失学习能力；

将***RNN***循环体在同一时刻重复多次即可得到**深层循环神经网络*(deep RNN)***，每层循环体中的参数相同，而多层之间参数可以不同，同时在多层循环体之间的连接可以使用***dropout***，而在时序方向上不能使用***dropout***；

#实现基本的***LSTM Cell***

**lstm = tf.nn.rnn\_cell.BasicLSTMCell(LSTM\_SIZE)**

**lstm = tf.nn.rnn\_cell.DropoutWrapper(lstm, output\_keep\_prob=KEEP\_PROB)**

**deep\_lstm = tf.nn.rnn\_cell.MultiRNNCell([lstm]\*LAYERS)**

state = deep\_lstm.get\_state(BATCH\_SIZE, tf.float32)

#前向传播

for step in range(TRAINING\_STEPS):

if step > 0:

tf.get\_variables\_scope().reuse\_variables()

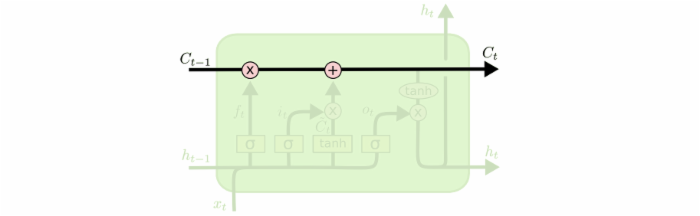
**deep\_lstm\_output, state = deep\_lstm(input\_tensor, state)**

final\_output = fully\_connected\_layer(deep\_lstm\_output)

loss += calculate\_loss(final\_output, expected\_output)

### LSTM

长短期记忆网络***(Long Short Term Memory)LSTM***，可以解决预测问题对先前状态的长期依赖问题，在***RNN***中，**，**若激活函数，则循环体中仅有一个简单的结构，而***LSTM***在循环体内部增加了三种门控单元：



* **遗忘门：，遗忘门层输出一个与状态向量相同维度的向量，其中：**

***，*对前一时刻状态向量各个元素的保留概率，0表示不保留，1表示完全保留；**

* **输入门：，需要更新的状态信息的概率；**

**，生成新的候选状态信息；**

**，输入门层通过对应向量的点乘运算并求和完成对旧的状态信息的遗忘和新的状态信息的添加，得到当前时刻的状态向量；**

* **输出门：，当前时刻的输出门控；**

**，输出门层对当前时刻的状态信息进行控制输出，得到预测值作为当前时刻的输出；**

### GRU

门控重置单元***(Gate Reset Unit)GRU***，在***LSTM***的基础上将输入门和遗忘门合并成一个更新门：**，作为上一时刻输出信号的门控；**

**，作为当前时刻输出的更新门控；**

输出候选信号**；**

**当前时刻输出信号 ；**

### 梯度消散/爆炸与序列截断

设有三个隐含层的神经网络：

**，其中：**

**表示隐含层输出，，**无论是权值***w***还是偏置***b***，其数值改变均会沿着梯度一直传递到最终的输出**，即梯度的链式反应：**

**；**

若激活函数 **，则有，在处取得最大值 ，若，则有 ，随着层数加深，梯度会呈指数下降直至消失，此时出现梯度消散(Gradient Vanishing)；**

**若，且有，则随着层数加深，会导致梯度指数上升直至发生上溢，此时出现梯度爆炸(Gradient Explosion)；**

**通常使用ReLU做为激活函数，，不会带来梯度消散问题；**

**在*RNN*的训练中，通常会在数据预处理时，将输入的样本序列截断成较短的长度来避免梯度消散和梯度爆炸问题；**

**在Tensorflow中可使用tf.clip\_by\_global\_norm(g, MAX\_GRAD\_NORM)进行梯度修剪：**

**指定了MAX\_GRAD\_NORM限定梯度向量的L2范数，对梯度向量g进行修剪，返回一个修剪后的梯度列表grads和中间计算变量global\_norm，形式化表达：**

**为待修剪梯度向量；**

**即global\_norm，是梯度向量的L2范数；**

**传入参数MAX\_GRAD\_NORM以表示；**

**则有 ；**

**即当梯度向量的L2范数大于指定的最大范数值时，就将所有梯度进行等比例缩放；**

### BPTT

时序反向传播算法***(Back Propagation Through Time)BPTT***，是***RNN***参数更新使用的算法：

状态向量**；**输出向量；

损失函数**，其中是常数，且有总损失函数；**

**设有 ；**

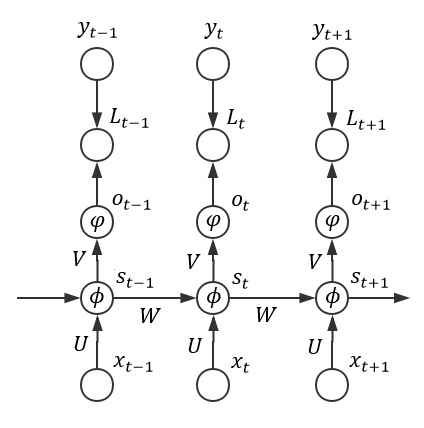
**；**

**损失函数对权值矩阵求导： ，与矩阵*V*无关，因此对权值矩阵*V*的反向传播与时序无关；**

**；**

**；**

**即梯度是沿着时间通道进行传播的：**



### CNN与RNN对比

将含有多个隐含层的神经网络统称为深度神经网络***(Deep Neural Network)DNN***，全连接神经网络***(Full Connected Nerual Network)FCNN***、卷积神经网络***CNN***和递归神经网络***RNN***的区别：

* **全连接神经网络又称前馈神经网络*(Feed-forward Neural Network)FNN***仅仅单纯地接收样本数据，认为样本间是相互独立的，其性能在一定程度上依赖于前期数据处理中的特征工程；
* **卷积神经网络*CNN***注重挖掘单个样本的结构性数据相邻区域间的关联，生成抽象的极具有概括性的特征；
* **递归神经网络*RNN*可以使用同一套网络结构处理不同的序列化数据，注重于挖掘样本与样本之间时序上的关联，*RNN*是时间循环神经网络*(Recurrent Neural Netwok)*和结构递归神经网络*(Recursive Neural Network)*的统称；**

### NLP

自然语言处理***(Natural Language Processing)NLP***是循环神经网络的一个典型应用，称为语言模型，语言模型要解决两个问题：

* 学习问题：计算一个句子出现的概率即：

***；***

传统方法常基于***n-gram***假设处理语言模型的学习问题，即假设句子中每个词语的出现仅与之前的***n***个单词有关，从而简化模型的学习：

；

* 预测问题：语言模型的预测问题即根据已学习的模型和输入序列预测下一个单词出现的概率**；**

语言模型效果评价指标：

；

**；**

**得到： ；**

防止连乘造成的数值下溢，取对数，得到模型的对数复杂度：

**；**

**使用MultiRNN实现语言模型：**

* **数据集：PTB(Penn Treebank Dataset)文本数据集，词汇量10000，句子结束标记为换行符，稀有单词标记为<unk>；**
* 需要借助**Tensorflow**官方提供的**tf.models package**，在**tensorflow1.0**及其更高版本中，已经将**models**分离出来，要使用**models**中提供的**API**以及运行**tutorials**中的**research demo**，需要从<https://github.com/tensorflow/models> **clone**仓库到本地，导入所需的包即可：

import tensorflow\_ptb\_reader as reader

train\_data, validate\_data, test\_data = reader.ptb\_raw\_data(DATA\_PATH)

**使用函数reader.ptb\_iterator(train\_data, BATCH\_SIZE, NUM\_STEPS)将训练数据组织成包含N = BATCH\_SIZE个序列，每个序列长度为m = NUM\_STEPS的batch；**

**as feature**

**as label**

***…***

共***N***个**组成一个Batch作为RNN的输入；**

1. **PTBModel构建模型，定义各类操作：**

**class PTBModel(object):**

**def \_\_init\_\_(self, is\_training, batch\_size, num\_steps):**

**…**

**#以下代码都在初始化方法中**

**读入参数作为类属性并定义输入，输入是一个矩阵[batch\_size, num\_steps]：**

self.batch\_size = batch\_size

self.num\_steps = num\_steps

self.inputs = tf.placeholder(tf.int32, shape=**[batch\_size, num\_steps],**

name=’x-input’)

self.targets = tf.placeholder(tf.int32, shape=**[batch\_size, num\_steps]**,

name=’y-input’)

定义单个LSTMCell结构：

**lstm\_cell = tf.nn.rnn\_cell.BasicLSTMCell(HIDDEN\_SIZE,**

**state\_is\_tuple=True)**

**其中，LSTM隐含层规模为size=HIDDEN\_SIZE，由LSTM的更新方程**

**；**

**；**

**；**

**；**

**；**

**；**

**可以看出，LSTM内部本质上使用的也是全连接网络，将前向传导的结果经过门控处理，因此在定义LSTM结构时，要指定隐含层的规模；**

**此外，state\_is\_tuple参数指定了输出的状态值c与输出值h是否以元组形式返回，若为 False，则拼接成一个向量；**

**训练时，加入层间dropout：**

if is\_train==True:

lstm\_cell = tf.nn.rnn\_cell.DropoutWrapper(lstm\_cell,

output\_keep\_prob=KEEP\_PROB)

**构造两层LSTM网络，第一层输出作为第二层的输入**

**cell = tf.nn.rnn\_cell.MultiRNNCell([lstm\_cell]\*NUM\_LAYERS,**

**state\_is\_tuple=True)**

初始化状态为0向量：

self.initial\_state = cell.zero\_state(batch\_size, tf.float32)

***embedding layer*将字典中所有的单词转换成单词向量：**

embedding = tf.get\_variable(“embedding”, **[VOCAB\_SIZE, HIDDEN\_SIZE]**)

将一个***Batch***的训练样本转换成对应的单词向量，self.input\_data给出的是单词在字典中的id，embedding建立了一个与字典容量相同的空间，将单词id映射成HIDDEN\_SIZE维的向量，此映射表示为，通过查表**tf.embedding\_lookup(embedding\_matrix, id)**可以得到指定单词的向量表示，详见[Word2vec](#_Word2vec)，此时输入**inputs为[batch\_size, num\_steps, HIDDEN\_SIZE]矩阵**；

inputs = tf.nn.embedding\_lookup(**embedding**, self.input\_data)

outputs = []

states = self.initial\_state

模型的评价要对整个时序上的损失之和进行考量，因此，要保存在整个时序上的输出结果；

with tf.variable\_scope(“RNN”, reuse=tf.AUTO\_REUSE):

for **step** in range(num\_steps):

cell\_output, state = lstm\_cell([:, **step**, :], state)

outputs.append(cell\_output)

每一个时刻输出的cell\_output是一个**[batch\_size, 1, HIDDEN\_SIZE]**的矩阵，经过**num\_steps**步后，将整个时序上的样本输入，得到**batch**中的全部样本的预测输出，此输出与输入有相同的维度；

得到当前输入***Batch***前向传播的结果后还要经过一个全连接层进行处理，最终输出的是对下一时刻序列中可能会出现的单词的概率，即一个长度为**VOCAB\_SIZE**的向量，全连接层输出维度为**VOCAB\_SIZE**，输入应当是对应的单词向量，其维度为**HIDDEN\_SIZE**：

**outputs = tf.reshape(tf.concat(outputs, 1), [-1, HIDDEN\_SIZE])**

weight = tf.get\_variable(“weight”, shape=[HIDDEN\_SIZE, VOCAB\_SIZE])

bias = tf.get\_variable(“bias”, shape=[VOCAB\_SIZE])

logits = tf.matmul(**outputs**, weight) + bias

此处变量**weight**和**bias**的初始化值统一由主函数调用***PTBModel***时传入；

计算序列的交叉熵，设置序列中各个时刻的预测值占有相同的权重：

loss = tf.contrib.legacy\_seq2seq.sequence\_loss\_by\_example([logits],

[tf.reshape(self.targets, [-1])],

**[tf.ones([batch\_size\*num\_steps], dtype=tf.float32)]**)

**计算Mini-Batch Loss并保存最终的状态向量：**

**self.cost = tf.reduce\_sum(loss)/batch\_size**

**self.final\_state = state**

**模型训练时，还要对梯度进行修剪，防止出现梯度膨胀和梯度消散：**

**grads, \_ = tf.clip\_by\_global\_norm(tf.gradients(self.cost,**

**tf.trainable\_variables()), MAX\_GRAD\_NORM)**

**optimizer = tf.train.GradientDescentOptimizer(LEARNING\_RATE)**

**self.train\_op = optimizer.apply\_gradients(zip(grads,**

**tf.trainable\_variables()))**

**tf.gradients(self\_cost, trainable\_variables)**返回一个总损失函数对所有变量的梯度列表：**；**

**tf.clip\_by\_global\_norm(g, MAX\_GRAD\_NORM)是对梯度进行修剪，见**[**梯度消散/爆炸与序列截断**](#_梯度消散/爆炸与序列截断)**；**

**函数optimizer.apply\_gradients(zip(grads, tf.trainable\_variables()))使用已修剪过的梯度向量对所有参数进行更新，替代optimizer.minimize(self.cost)；**

1. **输入数据，运行模型run\_epoch：**

**调用ptb\_reader.iterator(data, model.batch\_size, model.num\_steps)从数据集中生成batch供模型训练使用：**

**for step, (xs, ys) in enumerate(reader.ptb\_reader.iterator(data,**

**model.batch\_size, model.num\_steps)):**

**cost, state, \_ = sess.run([model.cost, model.final\_state, train\_op],**

**feed\_dict={self.input\_data: xs, self.target: ys, self.initial\_state=state})**

total\_cost += cost

iters += model.num\_steps

count += 1

**return np.exp(total\_cost/iters)**

**total\_cost返回的数值是，要求模型的复杂度，则有，作为模型的评价指标；**

1. 主函数，运行测试模型代码：

获得训练数据集train\_data、验证数据集validate\_data和测试数据集test\_data；

定义**initializer**初始化模型中所有的参数：

**initializer** = tf.random\_uniform\_initializer(-0.05, 0.05)

生成训练模型与验证模型，两者区别在于是否需要反向传播优化参数：

with tf.variable\_scope(“language\_model”, reuse=tf.AUTO\_REUSE,

initializer=**initializer**):

train\_model = PTBModel(True, TRAIN\_BATCH\_SIZE, TRAIN\_NUM\_STEP)

with tf.variable\_scope(“language\_model”, reuse=tf.AUTO\_REUSE,

initializer=**initializer**):

eval\_model = PTBModel(False, EVAL\_BATCH\_SIZE, EVAL\_NUM\_STEP)

启动会话，训练并评测模型：

with tf.Session() as sess:

sess.run([tf.local\_variables\_initializer, tf.global\_variables\_initializer])

for epoch in range(NUM\_EPOCHS):

run\_epoch(session, train\_model, train\_data, train\_model.train\_op,

True) # outlog输出日志控制

validate\_perplexity = run\_epoch(session, eval\_model, validate\_data,

tf.no\_op(), False)

test\_perplexity = run\_epoch(session, eval\_model, eval\_data, tf.no\_op(),

False)

**运行源码，测试入口：**

**if \_\_name\_\_ == ‘main’:**

**tf.app.run()**

**完整源码见“./src/language\_model”；**

### Word2vec

在语言模型中，常用的单词表达方式有两种：

* **One-hot Representation：**

以每个单词在词汇表(字典)中出现的位置作为标识，生成此单词的向量，若字典容量为**，**则生成的单词向量维度为**，**即有：

字典 **，**

**映射单词向量为 ，*w*为待表示的单词，*v*为表示后的向量，1元素在*v*中的位置即单词*w*在字典中的位置；**

**One-hot编码得到的向量维度较大，而且假设各个维度之间彼此独立，此时网络中参数较多；**

* **Distributed Representation：**

其数学含义为：

***，***即将**文本空间 (字典)中的单词**映射成一个高维空间中的向量；

使用向量的形式表示词语，可以借助距离度量较为方便的寻找词语间的相似性，如余弦距离 **，另外google提供的word2vec tools还提供了一种analogy性质，类似于：**

；

**通常embedding vector由one-hot编码转换得到，寻找单词之间的联系：使用无监督聚类方法或者寻找一个映射，将one-hot向量映射到另外一个高维空间中，tensorflow.embedding\_lookup(embedding\_matrix, id)提供了一种直接根据单词id查找对应embedding vector的方法：**

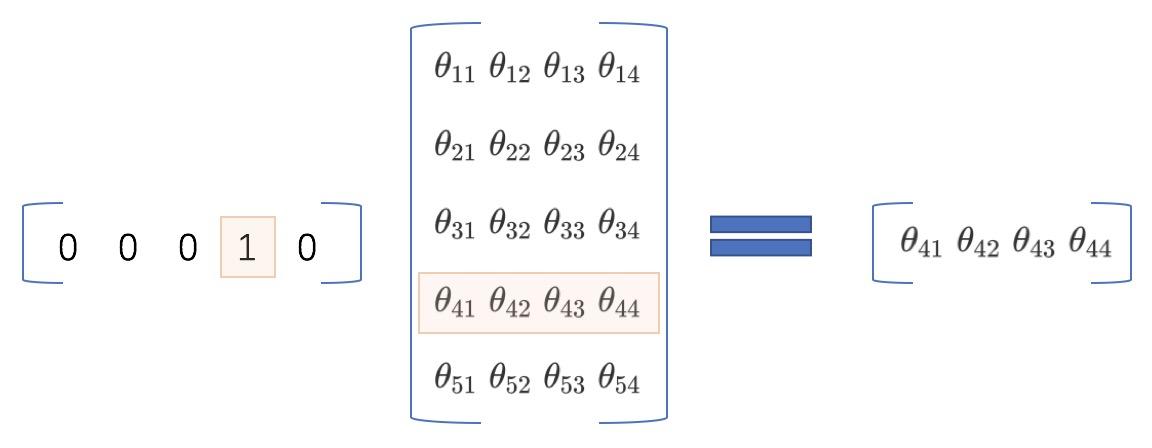
**使用embedding\_lookup()函数将字典中所有的单词转换成单词向量：**

embedding = tf.get\_variable(“embedding”, **[VOCAB\_SIZE, HIDDEN\_SIZE]**)

将一个Batch的训练样本转换称对应的单词向量，此时输入**inputs为**

**[batch\_size, num\_steps, HIDDEN\_SIZE]矩阵**；

inputs = tf.nn.embedding\_lookup(**embedding**, self.input\_data)



***Embedding***在***LSTM Language Model***中是以一个全连接层存在的，模型的输入**self.input\_data**只有经过***embedding***层处理后得到的输出**inputs**才能提供给***LSTM Cell***进行预测，模型的训练过程中包含了**Embedding Layer**参数的优化；

### TF-Learn

**Tensorflow**高层封装工具**TFLearn**可以将模型实现模板化：

**import tensorflow.contrib.learn as learn**

* 自定义模型**lstm\_model**：

1. 定义网络结构和前向传播过程：

lstm\_cell = tf.nn.rnn\_cell.BasicLSTMCell(HIDDEN\_SIZE, state\_is\_tuple=True)

lstm\_cell = tf.nn.rnn\_cell.DropoutWrapper(lstm\_cell,

output\_keep\_prob=KEEP\_PROB)

lstm = tf.nn.rnn\_cell.MultiRNNCell([lstm\_cell]\*NUM\_LAYERS)

x = tf.placeholder(dtype, [BATCH\_SIZE, NUM\_FEATURES], name=’x-input’)

y\_ = tf.placeholder(dtype, [BATCH\_SIZE, NUM\_FEATURES], name=’y-input’)

**y = tf.nn.dynamic\_rnn(lstm, x, dtype=dtype)**

1. 设置全连接层，用于完成最后的预测：

with tf.variable\_scope(“fcc”, reuse=tf.AUTO\_REUSE):

weight = tf.get\_variable(“weight”, [HIDDEN\_SIZE, LABEL\_LENGTH])

bias = tf.get\_variable(“bias”, [LABEL\_LENGTH])

logits = tf.matmul(y, weight) + bias

1. 定义损失函数及优化算法选择:

loss = tf.contrib.legacy\_seq2seq.sequence\_loss\_by\_example(logits, y\_,

example\_loss\_weights)

grads, \_ = tf.clip\_by\_global\_norm(tf.gradients(loss, tf.trainbale\_variables()))

train\_op =

tf.GradientDescentOptimizer(LEARNING\_RATE).apply\_gradients(

zip(grads, tf.trainable\_variables()))

1. 固定返回值：

**return logits, loss, train\_op**

* 对模型进行封装：regressor = **learn**.Estimator(model\_fn=lstm\_model)
* 训练模型，执行迭代：

regressor.fit(train\_data, train\_label, batch\_size=BATCH\_SIZE,

steps=TRAIN\_STEPS)

* 结果预测：**predicts** = **[predict] for predict in regressor.predict(test\_data)**
* 计算精度：**rmse = np.sqrt(((predicts – test\_label)\*\*2).mean(axis=0))**

## RBM与吉布斯采样

### 受限玻尔兹曼机

受限玻尔兹曼机***(Restricted Boltzman Machine)RBM***，是一种生成式随机网络***(Generative Stochastic Neural Network)***，由观测变量节点***(Visible Unit)***与隐藏变量***(Hidden Uint)***节点构成，观测变量与隐藏变量都是布尔型数据，整个网络是一张二部图，仅观测变量与隐藏变量异类变量之间存在连接，同类变量之间不存在连接，设观测变量节点集合为**，**隐含变量节点集合**，**连接权值矩阵为：

**；**偏置向量**，；**基于观测变量和隐含变量的联合配置***(Joint Configuration)***能量建模**：**

；

**是模型的参数组；**

**依据联合配置能量得到其联合概率分布：**

；

即有**；其中，为变量节点取值张成的向量；**

**控制变量，求观测变量的边缘分布：；**

**依据观测数据估计参数，构造似然函数：**

；

模型学习的任务是求参数使得似然函数极大化：

**；**

***RBM***的学习算法是对比散度法***(Contrastive Divergence)CD***算法：

；

由于概率归一化因子**，考虑节点各种取值组合；**

**得到：**

**；**

第一部分中的**表示在限定观测节点数据为样本时，隐含节点的经验分布；**

**第二部分表示观测节点与隐含节点的联合分布，无法直接计算，只能通过Gibbs Sampling得到其近似值；**

**不考虑求和项，可得到各个参数的梯度：**

**;**

**;**

**;**

**使用Gibbs Sampling可以得到参数梯度的近似**

***CD***算法：取**样本数据赋予观测节点**，**随机初始化**连接权值矩阵**以及**偏置向量**，；**

**外循环：for step in range(*MAX\_STEPS*):**

1. **以给定观测数据生成隐含数据的条件分布，对所有个隐含节点：**

**，其中；**

**上式表示给观测节点取值为条件下，以的概率为1，否则为0，按照节点取值的概率分布得到所有隐含节点的状态：**

**；**

1. **由(1)中得到的隐含节点的取值，对个可见节点层进行重构，**

**，其中；**

**上式表示给定隐含节点取值条件下，以的概率为1，否则为0，按照节点取值的概率分布重构所有观测节点的状态：**

**；**

1. **按照重构的观测节点取值，得到隐含节点的取值：，其中；**

**；**

1. **更新参数：**

**；**

**；**

**；**

**观测单元数量等于训练数据的特征维度，隐含节点个数需要事先指定；**

### 吉布斯采样

**吉布斯采样*(Gibbs Sampling)*：对*K*维随机向量，**无法得知其联合概率分布，但已知给定其他***K-1***个分量的条件下，第***k***个分量的条件分布：

；

从随机向量的任意状态出发，依据条件概率分布，迭代地对其各个分量进行采样，随着采样次数的增加，随机向量

**以*n*的几何级数速率逼近随机向量的联合概率分布；**

**应用于*RBM*模型中，可以使用吉布斯采样法得到联合概率分布：**

**🡪 🡪 🡪 🡪 …**

**🡪**

**在采样步数足够大的情况下，可以得到近似联合概率分布；**

## GAN

生成式对抗网络***(Generative Adversarial Network)GAN，***设计一个生成器***G(Generator)***，一个判别器***D(Discriminator)***：

神经网络的训练过程中，对训练样本加入微小的扰动可能会带来错误的输出，此时称这种样本为**对抗样本**，对抗样本的误识别是由神经网络的线性性质带来的**；**

生成式对抗网络可以从样本中学习出新的样本，生成器学习样本的分布，用服从某一分布***(Gaussian/Evenly)***的噪声***z***生成一个类似于真实训练数据的**伪样本**，评价结果是，二者越接近越好；判别器则接受混杂了伪样本的训练数据集，若接收到真实样本则输出一个较大的概率值，若接收到一个伪样本则输出一个较小的概率值；

训练时，固定生成器***G***，更新判别器***D***的权重，进而固定判别器***D***的权重，更新生成器的权重，交替进行，直至达到纳什均衡状态，此时，生成器输出的样本分布接近于训练样本中的数据分布，而判别器***D***的分类结果接近***50%***，等同于随机猜测；

；

## Tensorflow

### 三大组件

* 计算图***Graph***：计算模型，用于管理不同的集合，**节点**是其基本的运算单元，**边**是节点间的依赖关系：

tf.get\_default\_graph()获取当前默认计算图；

tf.Graph()生成新的计算图；

* 张量***Tensor***：数据模型，本身不存储数据，只是对运算结果的引用：

tf.constant(shaped\_array, name, dtype)

tensor.get\_shape()获取张量的维度信息；

tf.Session().run(result)运行张量，输出运算结果；

* 会话***Session***：运算模型，管理tensorflow的系统资源和计算操作：

with tf.Session() as sess: #上下文管理机制

sess.run(…)

在Tensorflow中，要同时完成多个操作，有两种机制可以使用：

* **使用tf.control\_dependencies()绑定多个操作，定义一个操作占位符：**

**with tf.control\_dependencies([train\_step, variables\_average\_op]):**

**train\_op = tf.no\_op(name=’train’)**

* **使用tf.group()机制：**

**train\_op = tf.group(train\_step, variables\_average\_op)，若要求train\_step先执行，则可以通过以下代码实现**

**with tf.control\_dependencies([train\_step]):**

**train\_op = tf.group(variables\_average\_op)**

**Tensorflow通过元图(MetaGraph)记录计算图中的节点信息以及运行计算图中节点所需要的数据，元图的数据格式由MetaGraphDef Protocol Buffer定义，以.meta为后缀的文件保存：**

**Message MetaGraphDef{**

**MetaInfoDef meta\_info\_def=1; #元数据：计算图版本号，运算方法信息**

**GraphDef graph\_def = 2; #计算节点间的连接信息**

**SaverDef saver\_def = 3; #持久化所需的参数**

**map<string, CollectionDef> collection\_def = 4; #集合信息**

**}**

计算图中所有变量的取值保存在**model.ckpt**文件中，是一个**key-value列表，以SSTable格式存储；**

### Collection

在计算图***Graph***中，使用集合***(collection)***管理不同类别的计算资源(张量、变量、队列等)：

Tf.add\_to\_collection() #将资源加入集合

Tf.get\_collection() #从集合中读取资源

常用的集合：

Tf.GraphKeys.VARIABLES #持久化模型时获得所有变量

Tf.GraphKeys.TRAINABLE\_VARIABLES #模型训练时获得所有可训练变量

Tf.GraphKeys.SUMMARIES #日志生成的相关变量

Tf.GraphKeys.QUEUE\_RUNNERS #输入处理QueueRunner

Tf.GraphKeys.MOVING\_AVERAGE\_VARIABLES #计算变量的滑动平均值

### Variable和变量管理

***Tensorflow***中的**Variable**是一种特殊的张量，所有变量会自动加入集合**GraphKeys.VARIABLES**中，变量的数据类型有：

* tf.int32 -- int/signed [int]/long [int]/signed long [int] ：32位有符号整型
* tf.int64 -- ：64位有符号整型，C/C++中无对应，仅有long long 64位无符号长整型
* tf.int16 -- short [int]/signed short [int]：16位有符号整型
* tf.int8 -- char/signed char：8位有符号整型
* tf.uint8 – unsigned char：8位无符号整型
* tf.float32 -- float：32位浮点型
* tf.float64 -- double：64位浮点型
* tf.bool -- bool：单字节布尔型

**Tensorflow在神经网络的训练中，对数据集输入、连接权重、前向传播结果进行不同的处理：**

* **数据输入x使用占位符：x = tf.placeholder(dtype, shape, name)，可以使得样本数据按batch输入，避免生成大量的常量占用资源；**

**sess.run(y, feed\_dict={x: …})组织样本集数据输入；**

* **连接权重使用Variable：w = tf.Variable(tf.random\_normal(shape, stddev))，在会话tf.Session()中需要初始化定义的变量**

**sess.run([tf.global\_varibales\_initializer(), tf.local\_variables\_initializer()])**

* **前向传播结果y = network\_output()，需要在会话中运行才能得到结果：**

**sess.run(y, feed\_dict={x: …})**

运行sess.run(train\_step)后，会按照已设定的优化算法对集合**tf.GraphKeys.TRAINABLE\_VARIABLES**中的变量进行优化，常用的优化算法有：

* 梯度下降法：tf.train.GradientDescentOptimizer()；
* 带动量的梯度下降法：tf.MomentumOptimizer()；
* 自适应动量项梯度下降法：tf.AdamOptimizer()；

以上优化算法数学解释见[局部最小与全局最小](#_局部最小与全局最小)；

在***Tensorflow***中可以使用两种方式创建变量：

**tf.Variable(“v”, shape=[1], initializer=tf.constant\_initializer(1.0))**

**tf.get\_variable(tf.constant(1.0, shape=[1]), name=”v”)**

**tf.constant\_initializer()与tf.constant()是等效的，对于tf.Variable()参数name=”v”是可选的，但是tf.get\_variable()中则是必填项，函数会根据变量名name=”v”创建变量；**

在神经网络训练过程中，使用命名空间隔离不同层，使得tf.get\_variable()函数可以直接使用变量名称获得变量：

with tf.variable\_scope(“layer1”, reuse=**tf.AUTO\_REUSE**):

weights = tf.get\_variable(“weights”, [input\_node, output\_node],

initializer=tf.random\_normal\_initializer(stddev=0.1))

**第一次调用tf.get\_variable()时，变量layer/weights尚未创建，则调用initializer=tf.random\_normal\_initializer(stddev=0.1)创建一个维度为**

**[input\_node, output\_node]的矩阵，一旦创建成功后，initializer便不再起作用，后续调用tf.get\_variable()得到的是经过更新后的变量(反向传播优化)；**

也会使用with tf.name\_scope(“layer”)管理上下文，在使用tf.get\_variable()创建变量时，不会为其加上前缀，但都会为tf.Variable()创建的变量加前缀；

在应用Tensorflow训练神经网络时，通常将指示训练轮数的变量***global\_step***指定为不可训练变量：**global\_step = tf.Variable(0, trainable=False)，global\_step变量在优化器中进行更新：**

**train\_step =**

**tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss\_function, global\_step=global\_step)**

### 多线程

**Tensorflow**可使用队列机制实现多线程输入数据，多线程操作队列使用三个工具：**Queue**是**tensorflow**实现队列和缓存机制的数据结构；

**QueueRunner**是对同一队列多线程操作的封装，无论通过显式调用**qr.create\_threads(sess, coord=coord)创建线程**，还是隐式调用**tf.train.start\_queue\_runners(sess, coord=coord)都会返回创建的线程组queue\_op\_threads，将其加入Coordinator实现多线程协同：coord.join(queue\_op\_threads)**；

两种队列：

* **FIFOQueue**：先入先出队列
* **RandomShuffleQueue**：随机打乱元素次序队列

创建队列q = tf.FIFOQueue(capacity, “dtype”)，”dtype” like “int32” “float”…

队列操作***API***：

* q.enqueue([element]) #入队列
* x = q.dequeue() #出队列
* q.enqueue\_many([element0, element1, …]) #多元素入队列

**tf.QueueRunner()启动多个线程操作同一个队列**，线程间通信使用**tf.Coordinator()**：

**python**多线程操作导入包**threading**实现：

def Loop(coord, worker\_id):

#检查线程停止标志

while not **coord.should\_stop()**:

if np.random.rand() < 0.1:

print(“Stoping from id : %d\n” % worker\_id)

**#通知其他线程退出**

**coord.request\_stop()**

else:

print(“Working on id : %d\n” % worker\_id)

time.sleep(1)

**#创建tf.train.Coordinator()协同多线程**

**coord = tf.train.Coordinator()**

**#创建多个线程**

**threads = [threading.Thread(target=Loop, args=(coord, i, )) for i in range(5)]**

**for t in threads:**

**t.start()**

**#等待所有线程退出**

**coord.join(threads)**

* **显式创建tf.train.QueueRunner()后可通过调用create\_threads()启动多线程，启动的线程数也是显式指定的：**

queue = tf.FIFOQueue(capacity=50, dtype=[tf.float32, tf.int32])

enqueue\_op = queue.enqueue\_many([feature, label])

x, y = queue.dequeue()

**qr = tf.train.QueueRunner(queue, [enqueue\_op]\*5)**

with tf.Session() as sess:

coord = tf.Coordinator()

**#创建并启动多个线程**

**enqueue\_threads = qr.create\_threads(sess, coord, start=True)**

for step in range(100):

if **coord.should\_stop()**:

break

sample, label = sess.run([x, y])

**coord.request\_stop()**

coord.join(**enqueue\_threads**)

* **创建QueueRunner，将其加入默认集合tf.GraphKeys.QUEUE\_RUNNERS，并显式调用tf.train.start\_queue\_runners()启动集合中的线程：**

**…**

**tf.train.add\_queue\_runner(qr)**

**…**

**enqueue\_threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)**

**…**

**coord.join(enqueue\_threads)**

### 数据预处理

统一输入图片数据格式为***TFRecord***，***TFRecord***是一种文件类型，文件内容使用**tf.train.Example Protocol Buffer**存储，**Example**实例由数据结构**Features**组织，**Features**是一个由属性名称**string**到属性实体**Feature**的映射，而属性**Feature**可以是字符串**BytesList**，实数列表**FloatList**或者整数列表**Int64List**：

message Example{

Features features = 1;

};

message **Features**{

map<string, Feature> feature = 1;

};

message Feature{

one of kind{

BytesList bytes\_list = 1;

FloatList float\_list =2;

Int64List int64\_list =3;

}

};

对于图片数据来说，**一张图片可以看做一个Example**，**而图片的像素矩阵是BytesList属性实例，图片的分辨率作为Int64List属性实例，图片的类别是Int64List实例，还要创建一个从图片数据到三种属性的映射函数**，参见条目“图片数据格式”；

* 图片数据格式：一个***RGB***色彩模式的图像可以看做一个三维矩阵，图像存储时经过压缩编码处理，要还原图像矩阵，首先要对图像进行解码**(tf.image.decode\_jpeg(image\_raw\_data)/tf.image.decode\_png(image\_raw\_data))**得到一个**tensor**：

image\_raw\_data = tf.gfile.FastGFile(“path/picture.jpeg”, ‘rb’).read()

with tf.Session() as sess:

image\_data = tf.image.decode\_jpeg(image\_raw\_data)

…

#image矩阵数据格式转换

image\_data = tf.image.convert\_image\_dtype(image\_data, **dtype=tf.float32**)

#重新编码并写入文件

encoded\_image = tf.image.encode\_jpeg(image\_data)

with tf.gfile.GFile(“path/picture\_0.jpeg”, ‘wb’) as f:

f.write(encoded\_image.eval())

将训练数据集中的所有图片转换成**tf.train.Example Protocol Buffer**格式，并写入***TFRecord***文件，便于多线程读入组织Batch进行训练：

writer = tf.python\_io.TFRecordWriter(tfrecord)

# Convert into Example Protocol Buffer .

**example = tf.train.Example(features=tf.train.Feature(features={**

**'length': \_int64\_feature(length),**

**'width': \_int64\_feature(width),**

**'channel': \_int64\_feature(channel),**

**'image\_raw': \_bytes\_feature(image\_raw),**

**'label': \_int64\_feature(label)**

**}))**

writer.write(example.SerializedToString(example))

writer.close()

**多线程读取TFRecord文件见条目“多线程组织Batch”**；

对应的解析方法为：

features = tf.parse\_single\_example(

example, features={

'length': tf.FixedLenFeature([], tf.int64),

'width': tf.FixedLenFeature([], tf.int64),

'channel': tf.FixedLenFeature([], tf.int64),

'image\_raw': tf.FixedLenFeature([], tf.string),

'label': tf.FixedLenFeature([], tf.int64)})

image = tf.decode\_raw(features[‘image\_raw’], tf.float32)

label = tf.cast(features[‘label’], tf.int32)

….

若要解析多个example，使用tf.parse\_example()；

* 图片预处理：图片预处理遵循以下流程：

1. 读入图像文件：image\_raw\_data = tf.gfile.FastGFile(“path/picture**.jpeg**”, ‘rb’).**read()**；
2. 解码读入的图像文件：

image\_data = tf.image.decode\_jpeg(image\_raw\_data)；

1. 设置标注框：

**boxes** = tf.constant([[[box1\_y0, box1\_x0, box1\_y1, box1\_x1],

[box2\_y0, box2\_x0, box2\_y1, box2\_x1]]])；

1. 调整图片至预定尺寸：

image = tf.image.resize\_images(image, [height, width],

method=tf.random.randint(4))

1. 随机生成标注框用于标注样本，增大样本方差：

bbox\_begin, bbox\_size, **draw\_bbox** =

tf.image.sample\_distorted\_bounding\_box(tf.shape(image), bounding\_boxes=**boxes**, min\_object\_covered=0.1)

1. 转换图片矩阵的数据格式：

image = tf.image.convert\_image\_dtype(image, dtype=tf.float32)

1. 标注框绘制函数tf.image.draw\_bounding\_boxes()要求输入图片格式为tf.float32的4-D矩阵[batch\_index, height, width, channel]，因此要扩展image的第0维：

**image = tf.expand\_dims(image, 0)**

1. 绘制标注框：

distorted\_image = tf.image.draw\_bounding\_boxes(image, **draw\_bbox**)

1. 其他图像处理函数的输入均要求参数为3-D矩阵，因此要重新调整图像尺寸，并沿着标注框切割：

distorted\_image = tf.reshape(distorted\_image, [height, width, channel])

distorted\_iamge = tf.slice(distorted\_image, bbox\_begin, bbox\_size)

1. 翻转与调色：

distorted\_image = tf.random\_flip\_left\_right(distorted\_image)

**distorted\_image** = distort\_color(distorted\_image)

最后可以使用预处理后的**distorted\_image**生成**tf.train.Example**并写入***TFRecord***文件中，按照**’training’/’validation’/’testing’**的**category**划分可以写入不同的***TFRecord***文件中；

* 多线程组织**Batch：**

1. 获得原始数据集列表：

**files = tf.train.match\_filenames\_once(‘path/data.tfrecords-\*’)，使用正则表达式模式匹配所有tfrecord文件，得到一个文件列表；**

1. 创建文件列表队列：**file\_queue =** **tf.train.string\_input\_producer(files, shuffle=True, num\_epochs=REPEAT\_ITERS)，依据获得的文件列表创建一个文件列表队列，系统会一直拥塞至显式调用tf.train.start\_queue\_runners(sess=sess, coord=coord)，且定义了重复加载文件列表的轮数为REPEAT\_ITERS；**
2. 从文件队列中读取数据：创建**tf.TFRecordReader()**，读取文件，解析**Example**；

**reader = tf.TFRecordReader()**

**\_, serialized\_example = reader.read(filename\_queue**)

features = tf.parse\_single\_example(serialized\_example, features={

‘image\_raw’: tf.FixedLenFeature([], tf.float32),

‘label’: tf.FixedLenFeature([], tf.int32),…})

with tf.Session() as sess:

sess.run([tf.global\_variables\_initializer(),

tf.local\_variables\_initializer()])

coord = tf.train.Coordinator()

#显式调用，启动集合tf.GraphKeys.QUEUE\_RUNNERS中所有的QueueRunner，创建线程；

**threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)**

**…**

**coord.request\_stop()**

**coord.join(threads)**

1. 数据预处理：将(3)中得到的样本数据和标签转换成所需的数据类型：

image = tf.decode\_raw(features[‘image\_raw’], tf.float32)

label = tf.cast(features[‘label’], tf.int32)

依据条目“图像预处理”中流程对图片进行标注框、裁剪、随机翻转、调色等操作，得到调整后的图像**distorted\_image**及其标签**label**将单个样例组织成**Batch**；

1. 整理成**Batch**输入神经网络：

**example\_batch, label\_batch = tf.train.batch([image, label],**

**batch\_size=batch\_size, capacity=capacity, num\_threads=NUM\_THREADS)**

调用函数tf.train.batch()/tf.train.shuffle\_batch()执行样例及其标签的入队操作，将样例组合成batch，其中capacity = 1000 + 3 \* batch\_size，是队列的最大长度：

实际工程应用时，通常要求样本被随机打乱：

example\_batch, label\_batch = tf.train.shuffle\_batch([image, label],

batch\_size=batch\_size, capacity=capacity,

min\_after\_dequeue=LIMITED\_EXAMPLE\_IN\_QUEUE,

**num\_threads=NUM\_THREADS**)

其中参数**min\_after\_dequeue**限制了出队操作时队列中的最少元素数量；

**Batch**组织方法定义完成后，即可组织输出给网络训练使用，还可定义参数**num\_threads指定执行入队操作的线程数**，此时多线程对同一个文件中的样例执行**入队操作(数据读取、数据预处理)，**使用**tf.train.shuffle\_batch\_join()**可以实现对多个文件的入队，而文件队列是由tf.train.string\_input\_producer(filenames)得到的：

curr\_example\_batch, curr\_label\_batch = sess.run([example\_batch,

label\_batch])

**logits = inference(curr\_example\_batch)**

**loss = calc\_loss(logits, label\_batch)**

**train\_step =**

**tf.train.GradientDescentOptimizer(LEARNING\_RATE).minimize(loss)**

with tf.Session() as sess:

sess.run([tf.global\_variables\_initializer(),

tf.local\_variables\_initializer()])

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)

for step in range(TRAINING\_STEPS):

**sess.run(train\_step)**

coord.request\_stop()

coord.join(threads)

**Tensorflow API性能对比：**

**tf.shuffle\_batch()**：多个线程从同一个**TFRecord**文件中读取样例，并将读到的样例随机打乱入队列，然而当**TFRecord**文件中的样例相似度较高时，例如将相同***label***的样本写入同一个文件，极大程度上**会造成过拟合**；

**tf.shuffle\_batch\_join()**：多个线程从不同的**TFRecord**文件中读取样例，并随机打乱入队列组织**Batch**，此时应当控制线程数量，若线程数远远多于文件数量，则效果等同于**tf.shuffle\_batch()**，而且多个线程读取多个文件会带来过多的硬盘寻址，**增大IO开销**；

### 图像识别通用框架

卷积神经网络***CNN***处理图像问题的通用框架：

* 依据给定路径，读入图片文件并转换成**tf.train.Example Protocol Buffer**格式写入**TFRecord**文件，依据图片的**label**，可以将其写入不同的***TFRecord***文件：

**#遍历path/data下所有子目录，将图片逐一转换成TFRecord文件**

**def create\_image\_lists(testing\_percentage, validation\_percentage):**

**#创建原始图片文件列表**

**…**

**def get\_image\_path(image\_lists, image\_dir, label\_name, index, catagory):**

**#获得图片路径**

**…**

**def get\_tfrecord\_path(image\_lists, label\_name, index, category):**

**#获得图片对应的tfrecord文件路径**

**…**

**def convert\_to\_tfrecord(sess, image, n\_classes, label\_index, tfrecord):**

**#将指定图片转换成tfrecord文件**

**…**

* 建立文件列表队列，得到单个样例：

filenames = tf.train.match\_filenames\_once(“./path/pattern-\*”)

**file\_queue** = tf.train.string\_input\_producer(filenames)

reader = tf.TFRecordReader()

serialized\_example\_key, serialized\_example = reader.read(**file\_queue**)

* 解析图片**Example**，得到对应的**Features**：

features = tf.parse\_single\_example(serialized\_example, features={

‘image\_raw’: tf.FixedLenFeature([], tf.string),

‘length’: tf.FixedLenFeature([], tf.int32),

‘width’: tf.FixedLenFeature([], tf.int32),

‘channel’: tf.FixedLenFeature([], tf.int32),

‘label’: tf.FixedLenFeature([], tf.int32)})

image\_raw, label = features[‘image\_raw’], features[‘label’]

length, width = features[‘length’], features[‘width’]

channel = features[‘channel’]

* 对得到的图片**Features**进行预处理(标注框、翻转、调色、裁剪等)：

image = tf.decode\_raw(image\_raw)

image.set\_shape([height, width, channels])

image\_size = IMAGE\_SIZE

**boxes** = tf.constant([[[box1\_y0, box1\_x0, box1\_y1, box1\_x1],

[box2\_y0, box2\_x0, box2\_y1, box2\_x1]]])

distorted\_image = preprocess\_for\_train(image, image\_size, image\_size, boxes)

* 组织样本***Batch***，为神经网络训练提供输入，此时没有指定启动线程，只是隐式启动集合***(collection)* tf.GraphKeys.QUEUERUNNERS**中所有的**QueueRunner，集合中的QueueRunner由系统自动维护**：

capacity = MIN\_AFTER\_DEQUEUE + BATCH\_SIZE \* 3

image\_batch, label\_batch = tf.train.shuffle\_batch.join([distorted\_image, label], batch\_size=BATCH\_SIZE, capacity=capacity,

min\_after\_queue=MIN\_AFTER\_QUEUE)

* 定义前向传播：

logits = inference(image\_batch)

loss = calculate\_loss(logits=logits, labels=label\_batch)

train\_step = tf.train.GradientDescentOptimizer(LEARNING\_RATE).minimize(loss)

* 启动会话，完成训练：

with tf.Session() as sess:

sess.run([tf.global\_variables\_initializer(),

tf.local\_variables.initializer()])

coord = tf.train.Coordinator()

threads = tf.train.start\_queue\_runners(sess=sess, coord=coord)

for step in range(TRAINING\_STEPS):

sess.run(train\_step)

coord.should\_stop()

coord.join(threads)

**使用tf.train.batch()/tf.train.batch\_join()/tf.train.shuffle\_batch()/**

**tf.train.shuffle\_batch\_join()函数直接为网络训练提供输入，无需feed\_dict，可极大地节省内存开销；**

### 模型持久化

模型持久化使用**API tf.train.Saver()**实现，将计算图Graph的结构和参数取值保存到指定路径下的***checkpoint***文件中：

saver = tf.train.Saver()

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

saver.save(sess, “path/model.ckpt”)

…

从持久化模型中恢复，变量声明要与持久化模型中保持一致：

v1 = tf.Variable(tf.constant(1.0, shape=[1]), name=”v1”)

v2 = tf.Variable(tf.constant(2.0, shape=[1]), name=”v2”)

result = v1 + v2

saver = tf.train.Saver()

with tf.Session() as sess:

saver.restore(sess, “path/model.ckpt”)

sess.run(result)

若在新模型中声明变量别名，则可使用一个字典参数列表将模型变量名称与新定义变量联系起来：

v1 = tf.Variable(tf.constant(1.0, shape=[1]), **name=”other\_v1”**)

v2 = tf.Variable(tf.constant(2.0, shape=[1]), **name=”other\_v2”**)

saver = tf.train.Saver(**{“v1”: v1, “v2”: v2}**)

with tf.Session() as sess:

saver.restore(sess, “path/model.ckpt”)

sess.run(result)

**迁移学习中仅需要从持久化模型中获得前向传播的参数取值即可，*Tensorflow***提供了graph\_util.convert\_variables\_to\_constants()将变量及其取值以常量形式保存，将整个计算图保存在单个文件中：

from tensorflow.python.framework import graph\_util

v1 = tf.Variable(tf.constant(1.0, shape=[1]), name=”v1”)

v2 = tf.Variable(tf.constant(2.0, shape=[1]), name=”v2”)

result = v1 + v2

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

**#获得当前计算图的GraphDef部分并保存**

**graph\_def = tf.get\_default\_graph().as\_graph\_def()**

**#仅保存指定的节点**

**output\_graph\_def =**

**graph\_util.convert\_variables\_to\_constants(sess, graph\_def, [‘add’])**

#保存模型计算图

**with tf.gfile.GFile(“path/combined\_model.pb”, “wb”) as f:**

**f.write(output\_graph\_def.SerializeToString())**

**…**

**加载训练好的模型：**

**import tensorflow.python.platform import gfile**

**with tf.Session() as sess:**

**model\_filename = “path/combined\_model.pb”**

**with gfile.FastGFile(model\_filename, ‘rb’) as f:**

**graph\_def = tf.GraphDef()**

**graph\_def.parseFromString(f.read())**

**result = tf.import\_graph\_def(graph\_def, return\_elements=[“add:0”])**

**sess.run(result)**

测试程序中，从持久化模型中读取参数的滑动平均值作为测试模型，通过参数重命名方式加载模型：

variable\_averages = tf.train.ExponentialMovingAverage(MOVING\_AVERAGE\_DECAY)

variables\_to\_restore = variable\_averages.variables\_to\_restore()

saver = tf.train.Saver(variables\_to\_restore)

…

**ckpt = tf.train.get\_checkpoint\_state(MODEL\_SAVE\_PATH)**

**saver.restore(sess, ckpt.model\_checkpoint\_path)**

### 常用生成函数

**tf.random\_normal(shape, mean, stddev, dtype) #正态分布**

**tf.random\_normal\_initializer(mean, stddev, dtype)**

**tf.truncated\_normal(shape, mean, stddev, dtype) #正态分布，偏离校正**

**tf.truncated\_normal\_initializer(mean, stddev, dtype)**

**tf.random\_uniform(shape, min, max, dtype) #平均分布**

**tf.random\_uniform\_initializer(min, max, dtype)**

**tf.random\_gamma() #Gamma分布**

**tf.zeros(shape, dtype) #全0数组**

**tf.zeros\_initializer(dtype)**

**tf.ones(shape, dtype) #全1数组**

**tf.ones\_initializer(dtype)**

**tf.fill(shape, number) #填充数组**

**tf.constant([…]) #指定数值数组**

**tf.identify(input) #返回一个与输入相同的tensor**

**np.identity(dim, dtype) #生成对角1矩阵**

**enumerate(list) #返回元组(index, element)**

### 常用数据处理函数

**tf.clip\_by\_value(value, min, max) #限制数值范围**

**tensor.eval() #启动运算，以array形式得到结果**

**v1 = tf.constant([[1., 2.], [3., 4.]])**

**v2 = tf.constant([[5., 6.], [7., 8.]])**

**(v1\*v2).eval() #元素直接相乘**

**tf.matmul(v1, v2) #矩阵乘法**

**tf.where(tf.greater(v1, v2), (v1-v2)\*a, (v1-v2)\*b) # tf.greater()逐项比较选择较大的元素，tf.where()根据tf.greater()判断结果选择后面的元素，若为True选择第二个参数，否则选择第三个参数**

**tf.cast(tensor, dtype) #将输入tensor转换为指定数据类型**

**tf.equal(tensor1, tensor2) #比较并返回bool型tensor**

**tf.concat([tensor1, tensor2], axis) #拼接张量axis=0扩展行，axis=1扩展列**

**tf.reduce\_sum(input\_tensor, axis)**

*x = tf.constant([[1, 1, 1], [1, 1, 1]])*

*tf.reduce\_sum(x) # 6*

*tf.reduce\_sum(x, 0) # [2, 2, 2]*

*tf.reduce\_sum(x, 1) # [3, 3]*

*tf.reduce\_sum(x, 1, keep\_dims=True) # [[3], [3]]*

*tf.reduce\_sum(x, [0, 1]) # 6*

**tf.reduce\_mean(input\_tensor, axis)**

**tf.stack(values, axis=0) #将张量按照指定维度打包**

*x = tf.constant([1, 4])*

*y = tf.constant([2, 5])*

*z = tf.constant([3, 6])*

*tf.stack([x, y, z]) # [[1, 4], [2, 5], [3, 6]] (Pack along first dim.)*

*tf.stack([x, y, z], axis=1) # [[1, 2, 3], [4, 5, 6]]*

### 常用随机数生成函数

**np.random.randint(integer) #生成integer以内的正整数**

**np.random.rand() #生成0~1之间的随机数**

**np.random.randn(1000, 4) #生成1000个4-D vector**

### 常用文件处理函数

**with open(file\_full\_path, ‘w’) as file:**

**file.write(string\_line)**

**with open(file\_full\_path, ‘r’) as file:**

**string\_line = file.read()**

**os.walk(DIR) #深度优先得到指定路径下所有子路径**

**with gfile.FastGFile(os.path.join(MODEL\_DIR, MODEL\_NAME), ‘rb’) as f:**

**graph\_def = tf.GraphDef()**

**graph\_def.ParseFromString(f.read())**

with tf.Session() as sess:

sess.run(tf.global\_variables\_initializer())

**#获得当前计算图的GraphDef部分并保存**

**graph\_def = tf.get\_default\_graph().as\_graph\_def()**

**#仅保存指定的节点**

**output\_graph\_def =**

**graph\_util.convert\_variables\_to\_constants(sess, graph\_def, [‘add’])**

#保存模型计算图

**with tf.gfile.GFile(“path/combined\_model.pb”, “wb”) as f:**

**f.write(output\_graph\_def.SerializeToString())**

**…**

**filename = “path/output.tfrecords”**

**writer = tf.python\_io.TFRecordWriter(filename)**

**image\_raw = image.tostring()**

**example = tf.train.Example(features=tf.train.Features(feature={**

**‘pixels’: \_int64\_feature(pixels),**

**‘labels’: \_int64\_feature(np.argmax(labels)),**

**‘image\_raw’: \_bytes\_feature(image\_raw)}))**

**writer.write(example.SerializeToString())**

**writer.close()**

图片数据的解码见[数据预处理](#_数据预处理)；

**image\_raw\_data = tf.gfile.FastGFile(“path/picture.jpeg”, ‘rb’).read()**

**with tf.Session() as sess:**

**image\_data = tf.image.decode\_jpeg(image\_raw\_data)**

**…**

**#image矩阵数据格式转换**

**image\_data = tf.image.convert\_image\_dtype(image\_data, dtype=tf.float32)**

**#重新编码并写入文件**

**encoded\_image = tf.image.encode\_jpeg(image\_data)**

**with tf.gfile.GFile(“path/picture\_0.jpeg”, ‘wb’) as f:**

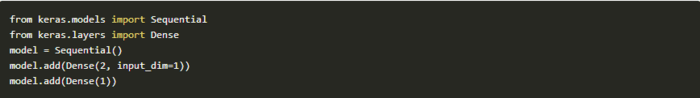
**f.write(encoded\_image.eval())**

# Keras

## 序贯模型

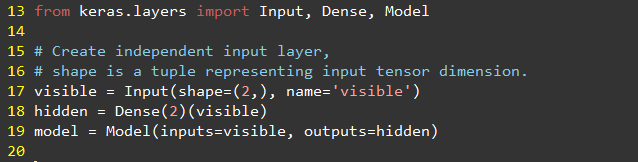


层定义传递给Sequential作为数组；



可以逐层定义网络结构，add到Sequential的尾部；

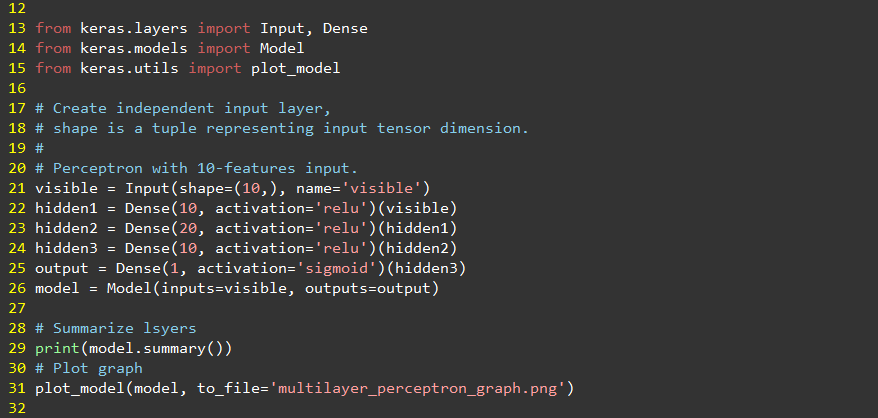
## 功能函数模型



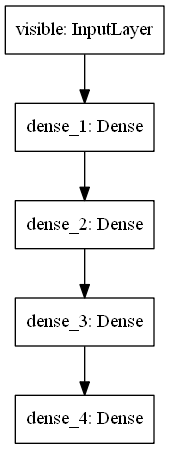
定义输入层，模型参数shape代表了输入tensor的维度，明确留出batch size，挂起最后一维的定义；

定义连接层以本层输入作为参数；

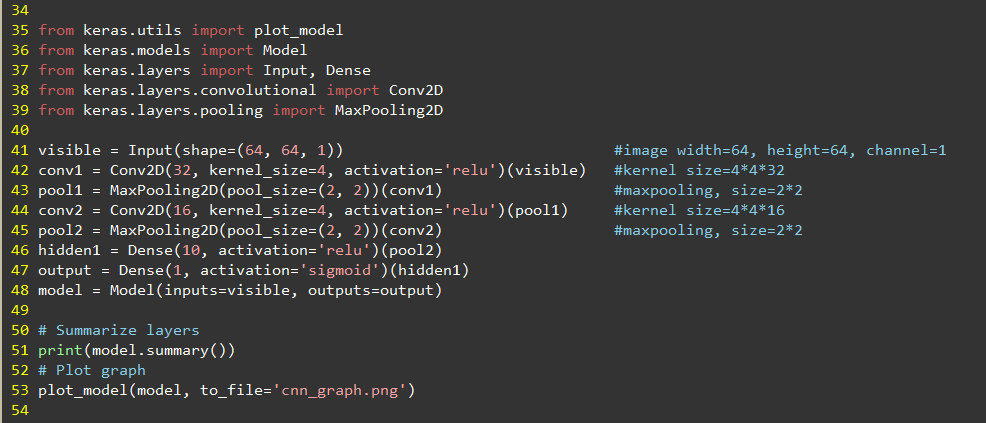
使用Model类，指定输入层与输出层创建模型；



输入样本具有10个feature，shape=(10,)，第一个参数代表样本数据的维度，第二个参数挂起，预留给batch size；三个隐含层hidden1 hidden2 hidden3分别具有10个节点，20个节点，10个节点，输出分类结果，keras模型图可视化结果保存到本地：



使用功能函数创建一个CNN模型：

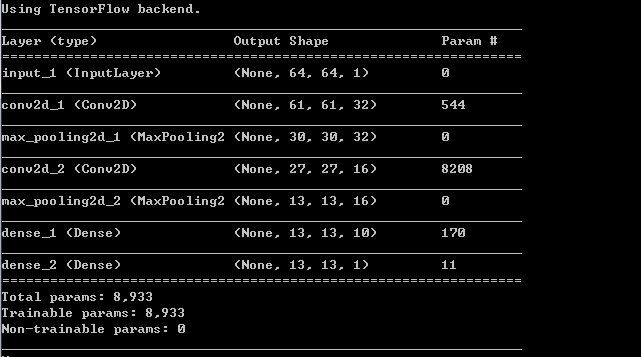


卷积核默认不使用全0填充，padding=’SAME’，且stride=1；

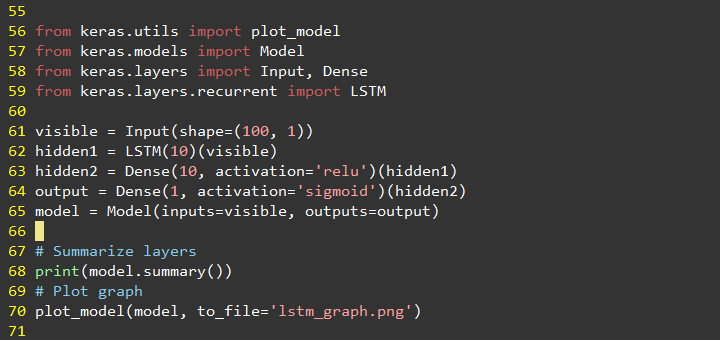
Output Shape中第1维None代表Batch index；

输出层经过逻辑回归，得到一个概率值：

***；***



构建基于LSTM的RNN模型：



hidden1 = LSTM(10)(visible)代表以Input层作为输入(样本feature=1，时序长度T为100)，LSTM单元输出为10，由LSTM状态方程：

**；**

**；**

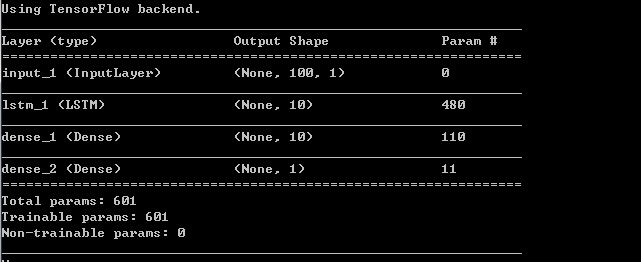
**；**

**；**

**；**

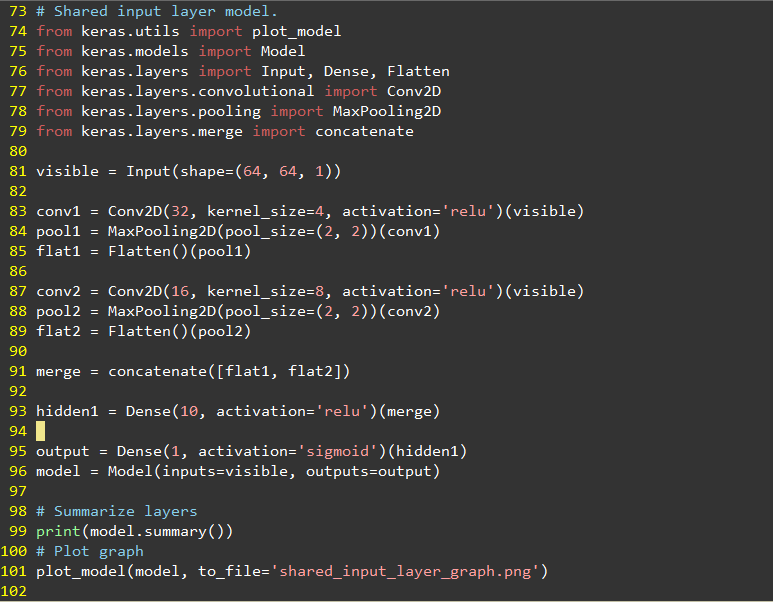
**；**

**concat**成为一个11维向量，其中**维度为10，与输出相同，维度恒为1，权重矩阵，，，均为10\*11矩阵，偏置向量维度均为10；**

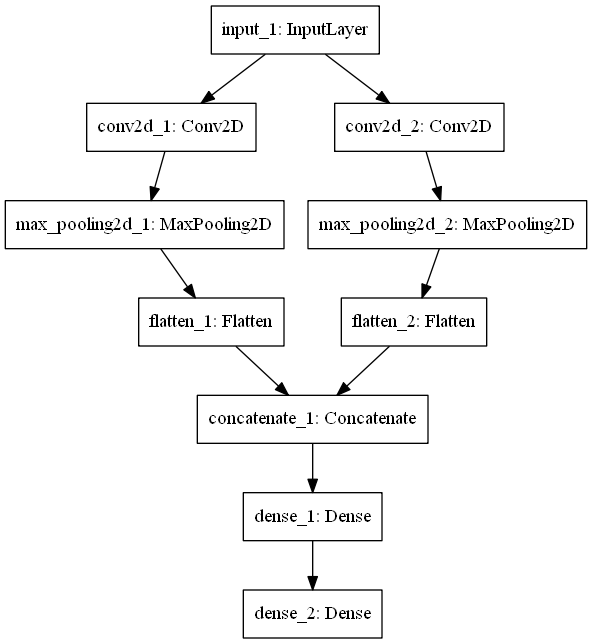


## 共享层模型

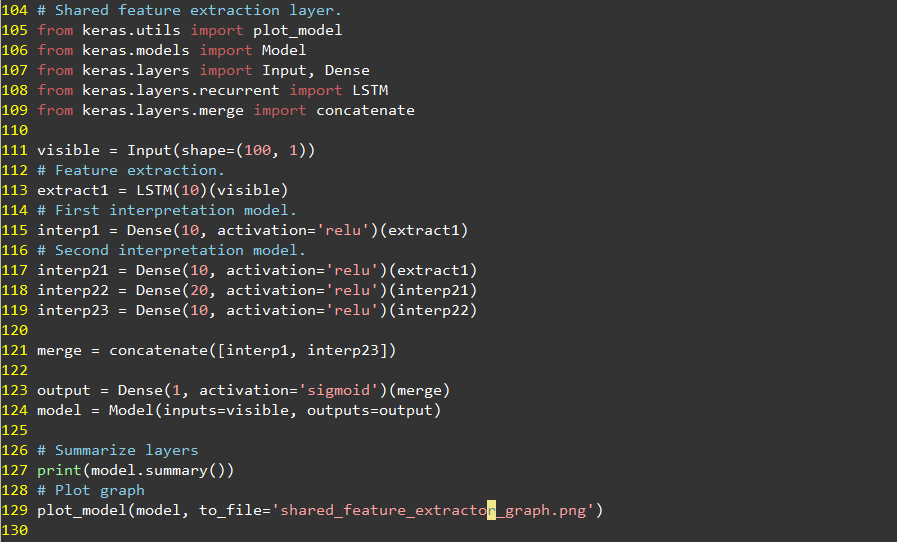
共享层模型接受相同的输入，以不同的卷积核提取特征，经过平滑处理，concat成一个向量；

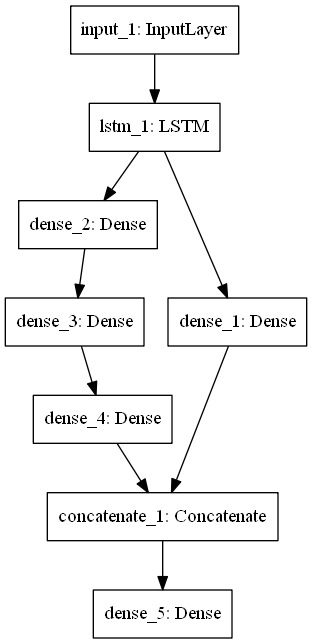


池化层默认strides为2，此处得到pool1[30, 30, 32]，pool2[28, 28, 16]，Flatten的作用是将输入tensor拉成一维向量，由concatenate拼接，作为全连接层的输入；



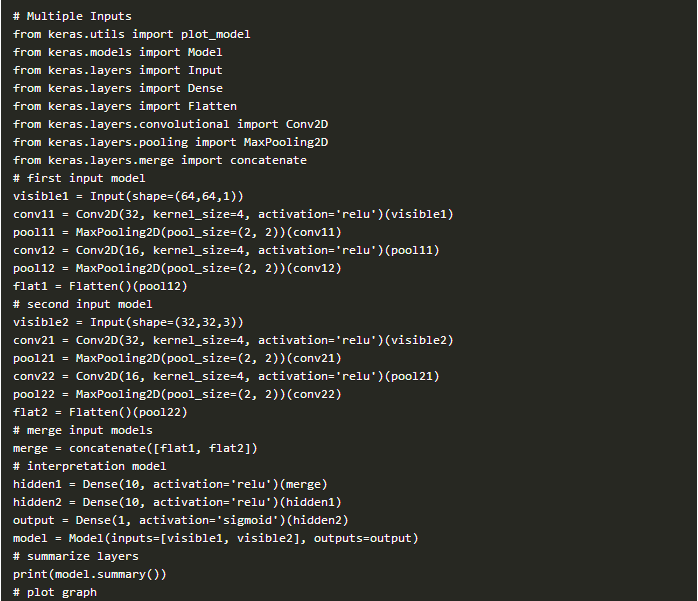
共享特征提取层：





## 多输入输出模型

多输入模型用于设计具有多个输入的复杂模型，每个输入具有单独的特征提取过程，将特征提取的结果concatenate，得到拼接向量，输入给一个全连接网络处理并输出预测结果；



## OCR模型

使用Keras构建OCR模型：CNN+GRU+CTC；

* ***CTC Loss：***
* ***Batch Normalization：***

# 深度学习实践

## CVR预估问题

**来自*Ali-IJCAI-2018***：

***CVR***预估是指在业务场景下，用户点击进入详情页后，发生购买行为的概率：

**；**

***CVR***的意义在于为商家寻找可以转化为潜在客户的关键特征，同时可以通过广告的投入产出比衡量广告投放的效果；  
***Query***：检索词；

***User***：用户信息；

***Ad***：商品；

***Context***：上下文内容；

***Shop***：店铺信息；

## 推荐系统

* 基于用户的推荐：
* 基于物品的推荐：

## 文本检测

* 光学字符检测***OCR***，模型内部使用***CNN + LSTM + CTC***包含两个步骤：

1. Objection detection：基于CNN给出文本所在的区域proposed regions，同时会对object region进行分类；
2. Classification：基于RNN/GRU/LSTM+CTC实现，可实现对文字与文字间联系的学习，CTC(connectionist temporal classification)是一种改进的RNN模型，解决标注序列长度小于输入序列的问题，引入空白标记符号blank，使用RNN进行标注，将最后输出的标注序列中的blank符号和重复标记删除，就可得到标注序列；

* 端到端end-to-end：预先不对字符进行切割，将图像直接输入神经网络得到分类结果输出；
* 意念机制attention-mechanism：

预处理：对包含文字的图像进行处理以便后续进行特征提取、学习，主要目的是减少图像中的无用信息；

1. **灰度化**（如果是彩色图像）、**降噪**、**二值化**、**字符切分以及归一化；**经过二值化后，图像只剩下两种颜色，即黑和白，其中一个是图像背景，前景颜色就是要识别的文字；
2. 降噪在这个阶段非常重要，降噪算法的好坏对特征提取的影响很大；
3. 字符切分则是将图像中的文字分割成单个文字——识别的时候是一个字一个字识别的，如果文字行有倾斜的话往往还要进行倾斜校正；
4. 归一化则是将单个的文字图像规整到同样的尺寸，在同一个规格下，才能应用统一的算法；
5. 特征提取和降维：特征是用来识别文字的关键信息，每个不同的文字都能通过特征来和其他文字进行区分，对于数字和英文字母来说，都是小字符集；对于汉字来说，特征提取比较困难，因为首先汉字是大字符集，国标中光是最常用的第一级汉字就有3755个；第二个汉字结构复杂，形近字多；
6. 在确定了使用何种特征后，视情况而定，还有可能要进行特征降维，这种情况就是如果特征的维数太高（特征一般用一个向量表示，维数即该向量的分量数），分类器的效率会受到很大的影响，为了提高识别速率，往往就要进行降维；
7. 分类器设计、训练和实际识别：分类器是用来进行识别的，就是对于第二步，对一个文字图像，提取出特征输入分类器，分类器对其进行分类；
8. 后处理：后处理对分类结果进行优化：分类器的分类有时候不一定是完全正确的，通过语言模型来进行校正——如果分类器将“在哪里”识别成“存哪里”，通过语言模型会发现“存哪里”是错误的，然后进行校正；
9. OCR的识别图像往往是有大量文字的，而且这些文字存在排版、字体大小等复杂情况，后处理中可以尝试去对识别结果进行格式化，比如按照图像中的排版排列什么的，例如一张图像，其左半部分的文字和右半部分的文字毫无关系，而在字符切分过程中，往往是按行切分的，那么识别结果中左半部分的第一行后面会跟着右半部分的第一行诸如此类；

目标序列长度固定时，可以看做一个多标签分类任务，使用CNN即可解决；

目标序列长度不固定时，可以看做语音识别问题，使用End-to-End方法解决；

* ***End-to-End***忽略中间处理过程，使用单个神经网络代替，从输入序列经过映射直接得到预测结果，End-to-End识别需要计算序列的CTC loss并进行反向传播：

***；***

**；**

***；***

**使用空字符’\_’填充输入字符序列s得到 \_h\_e\_l\_l\_o\_，则字符序列的长度由扩展为，定义标尺变量，时序标尺t每移动1步，序列标尺u移动1步或2步，满足；**

递推式：

***；*其中：；**

**；其中；**

**对于每一时刻t，其下一时刻t-1的输出只能是空白字符’\_’或者t时刻之前最后一个有效字符在字符序列中紧邻的下一有效字符；**

**；**

**检测问题：Faster-RCNN，CTPN和Yolo都属于object-detection方法，用于检测图片中的目标(文本或其他)，可借助VGG-16网络模型实现目标识别；**

**识别问题：**