

Table of Contents

序言	1.1
第一章 工具	1.2
简历	1.2.1
Ceres-Solver	1.2.2
参考书籍	1.2.3
第二章 C++	1.3
OOP	1.3.1
C++内存问题	1.3.2
线程与进程	1.3.3
进程间通讯与线程同步	1.3.4
模板类与模板函数	1.3.5
设计模式六大基本原则	1.3.6
设计模式	1.3.7
常用的python数据处理函数	1.3.8
第三章 算法	1.4
矩阵计算	1.4.1
线性方程组求解方法	1.4.1.1
矩阵变换技巧	1.4.1.2
矩阵分解方法	1.4.1.3
矩阵本征值求解方法	1.4.1.4
奇异矩阵分析	1.4.1.5
大规模稀疏矩阵问题	1.4.1.6
矩阵微分	1.4.1.7
算法与数据结构	1.4.2
二叉树	1.4.2.1
STL的底层机制	1.4.2.2
霍夫曼编码	1.4.2.3
最小生成树算法	1.4.2.4
最短路径Dijkstra算法	1.4.2.5
图像与信号处理	1.4.3
图像几何变换	1.4.3.1

傅里叶变换	1.4.3.2
采样定理	1.4.3.3
对比度增强	1.4.3.4
图像滤波	1.4.3.5
边缘检测	1.4.3.6
图像复原	1.4.3.7
图像压缩	1.4.3.8
图像分割	1.4.3.9
形态学处理	1.4.3.10
特征提取	1.4.3.11
EM算法及其应用	1.4.3.12
图模型	1.4.3.13
视觉词模型	1.4.3.14
Kriging插值	1.4.3.15
第四章 最优化	1.5
无约束最优化问题	1.5.1
线性搜索方法	1.5.1.1
梯度下降法	1.5.1.1.1
牛顿法	1.5.1.1.2
拟牛顿法	1.5.1.1.3
信赖域方法	1.5.1.2
高斯-牛顿法	1.5.1.2.1
Levenberg-Marquardt算法	1.5.1.2.2
Dogleg算法	1.5.1.2.3
共轭梯度法	1.5.1.3
次梯度法	1.5.1.4
随机坐标下降法	1.5.1.5
有约束最优化问题	1.5.2
拉格朗日乘子法	1.5.2.1
Karush-Kuhn-Tucker条件	1.5.2.2
罚函数法	1.5.2.3
增广拉格朗日乘子法	1.5.2.4
ADMM	1.5.2.5
最优化的优缺点及其改进方案	1.5.3
参数调节	1.5.4

线性与非线性方程组解的稳定性分析	1.5.5
第五章 机器学习	1.6
机器学习原理	1.6.1
分类	1.6.2
SVM, KKT条件与核函数方法	1.6.2.1
SVM	1.6.2.1.1
KKT条件与对偶	1.6.2.1.2
核函数方法	1.6.2.1.3
逻辑回归与最大熵模型	1.6.2.2
K近邻	1.6.2.3
回归	1.6.3
多元回归分析	1.6.3.1
正则线性模型	1.6.3.2
L1,L2正则化	1.6.3.2.1
聚类	1.6.4
K均值	1.6.4.1
EM算法与高斯混合模型	1.6.4.2
降维	1.6.5
PCA	1.6.5.1
流形学习	1.6.5.2
度量学习	1.6.5.3
集成学习	1.6.6
决策树	1.6.6.1
集成学习	1.6.6.2
Bagging	1.6.6.2.1
Boosting	1.6.6.2.2
RF,GBDT,XGBOOST,LightGBM比较	1.6.6.2.3
生成算法	1.6.7
贝叶斯方法	1.6.7.1
隐马尔科夫模型	1.6.7.2
DNN-HMM混合系统	1.6.7.3
图模型	1.6.8
马尔科夫链蒙特卡洛方法	1.6.8.1
特征工程	1.6.9
特征工程	1.6.9.1

Factorization Machines(FM)	1.6.10
模型评估	1.6.11
正则化方法原理与实践	1.6.12
机器学习学习任务	1.6.13
机器学习面试问题集	1.6.14
第六章 深度学习	1.7
BP神经网络	1.7.1
CNN	1.7.2
VGG	1.7.2.1
GoogleNet	1.7.2.2
ResiNN	1.7.2.3
RNN	1.7.3
传统RNN	1.7.3.1
LSTM	1.7.3.2
GRU	1.7.3.3
目标检测	1.7.4
R-CNN	1.7.4.1
Fast-RCNN	1.7.4.2
Faster-RCNN	1.7.4.3
YOLO	1.7.4.4
SSD	1.7.4.5
无人驾驶	1.7.5
Lane Finding	1.7.5.1
Tensorflow	1.7.5.2
深度学习成功的关键	1.7.6
先验与算法设计	1.7.7
模型压缩	1.7.8
第七章 推荐系统	1.8
计算广告学概论	1.8.1
协同过滤算法	1.8.2
FM(Factorization Machines),FFM	1.8.2.1
SVD在推荐系统的应用	1.8.2.2
基于内容的推荐算法	1.8.3
基于知识的推荐	1.8.4
在线学习	1.8.5

稀疏低秩分解	1.8.6
Learning To Rank LTR	1.8.7
Problem Solver Systems	1.8.8
第八章 项目工作	1.9
Residual NN	1.9.1
Fast项目	1.9.2
Miura项目	1.9.3
Titanic	1.9.4
Customer Segmentation	1.9.5
任务	1.9.6
深度学习的发展脉络	1.9.7
第九章 问题求解系统	1.10
Sklearn介绍	1.10.1
Sklearn的代码架构	1.10.2
Tensorflow介绍	1.10.3
Tensorflow的代码架构	1.10.4
常用数据处理技巧	1.10.5
NLP常用处理技巧	1.10.6
第十章 问题	1.11
基本ML问题	1.11.1
前向神经网络问题	1.11.2
基本C++问题	1.11.3
C++标准模板库	1.11.4
常规算法问题	1.11.5
结束	1.12

序

日知录是为效法顾炎武的《日知录》而作，里面会记录自己在学习与工作中学到的知识，该书的目的是为了打造一本属于自己的百科全书，形成自己的思想体系。最终里面的每一章可能代表一个方向与学科，比如C++，C#最终会成为一章，经济，中国史会成为一章。如果经济学里面记录的内容不多，就编成一章，如果内容多了，则会编成多章，比如宏观经济学，微观经济学，计量经济学。这样，最终一门大的学科，比如，经济学，就编成了一卷。

自然科学的基础是数学，数学是让一门学科严格化，并能称之为科学的基础。因此，本书的主线是以数学为基础，以编程为技术工具来高效的解决自然科学(如物理，半导体，光学，声学，分子化学)，商业与经济活动(推荐系统，计算广告，图像处理，信号处理，金融)中的问题。要践行毕达哥拉斯学派所谓的“世界的本质是数的理念”，换句话说，数学是所有学科唯一的通用语言。不同学科会分享相同的数学方法，这也就本书要总结的一个方面。不同学科的区别只是研究，处理问题的不同，而没有数学方法上的不同。就像随机偏微分方程可以用于量子力学，也可以用于金融；时间序列分析可以用于金融，也可以用于语音与数字信号处理。

本书的是自己学习的总结，更是自己思想体系不断进化的体现。因此，这本书会一直去增补，完善，没有停止的说法。在可以预见的未来，本书会经历几个阶段。第一阶段是知识的学习与记录；第二阶段是知识的运用与不同知识原理的提炼；第三阶段就是知识的创造过程，也就是为解决一类问题而创造出一个新的方法。三阶段简单来说，就是把书读厚，把书读薄，再把书读厚的阶段。就时间尺度来说，第一阶段是30周岁以前，第二阶段是30岁以后的三年，第三阶段就是33岁以后的岁月。正如“日知其所亡，月无忘其所能”，自己要做到每年一个总结，同时安排下一年的任务。

李品品 2018年

第一章：工具

1. 主要介绍一些Gitbook相关的工具
2. 一些库函数，比如Eigen,OpenCV
3. 一些推荐的书籍
4. 简历

简历:李品品

电话:(86)158-0083-4895

E-mail:1730443424@qq.com

求职目标:算法工程师, 机器学习, 深度学习

教育背景

2013.9-2016.6 兰州大学理论物理 硕士
2009.9-2013.6 兰州大学理论物理 学士

工作经历

2017.04-现在 KLA-Tencor半导体软件工程师

2016.07-2017.03 KLA-Tencor半导体算法工程师

技能

C++, 最优化, 线性代数, 矩阵分析, 机器学习与深度学习理论, 图像处理

研究经历

2014.3-2016.06 任意多边形的共形变换算法, 用于求解薛定谔方程, 狄拉克方程, 亥姆霍兹方程(理论推导)

2013.9-2016.06 相对论与非相对论的能谱与波函数的量子混沌研究(数值计算)

2012.9-2014.6 Sinai弹球Lyapunov exponent与K-S熵的半解析求解(理论推导+数值模拟)

获奖

2010 全国大学生数学竞赛非数学类三等奖

2007 湖南省高中生物理竞赛省二等奖

工作项目1:深度学习在晶圆缺陷检测中的应用研究(2016.07-2017.04)

- 个人搭建起来基于GPU计算的深度学习研究平台, 训练的硬件是Quardo4000, 工具是Linux下的Tensorflow。随着芯片制造中节点尺寸变小, 光学成像模型越来越复杂与困难, 因此我们研究通过训练深度神经网络来对提升现有的光学成像模型的性能的方案, 相当于做用神经网络来做一个图像增强的过程。我们基于端对端的训练, 研究了不同深度的CNN, pre-trained的VGG, 以及Residual NN。发现通过Residual NN可以很大的提升现有光学成像模型的性能。
- 光学成像建模研究: 基于傅里叶光学及其TCC matrix对光掩模成像进行理论研究

工作项目2:C++软件开发(2017.04-现在)

1. 半导体缺陷检测仪器的控制软件开发，基于MFC的C++以及半导体工厂自动化协议(SECS)的软件开发，来实现芯片的工厂自动化生产。主要用的是面向对象的C++编程，串口以及TCP/IP通讯，多线程与进程同步，数据库(SQL server)操作，COM技术，模式设计，硬件驱动以及UI设计。主要工作就是软件的维护以及新功能的开发。

工作项目3:算法优化(2018.03-现在)

1. 四探针测量薄膜特性，其核心是求解一个基于麦克斯韦方程推导出来的电流，电压方程。通过实验数据，求解出薄膜的参数。主要涉及到非线性方程的迭代求解与算法优化。求解主要基于牛顿法与梯度下降法的综合版--Levenberg-Marquardt算法，此外还利用了基于Trust-Region的Dog-Leg算法。在实际的优化过程中，我们会通过L2正则化来对一些变量进行约束，其目的是为了让我们计算出的参数更稳定。

个人项目:编写最优化与机器学习技术书籍一部(2018.03-现在)

1. 矩阵分析主要包括线性方程组求解，矩阵分解，矩阵变换，矩阵本征值问题以及奇异矩阵分析。
2. 最优化算法主要包含无约束优化问题(线性搜索与Trust-Region算法)，以及带约束优化问题。
3. 机器学习算法主要包含分类，回归，聚类，降维，集成学习，生成算法与图模型

自我评价

理论物理硕士毕业，在数值算法，最优化，线性代数，机器学习，矩阵论，概率统计，高等代数，冗长公式推导(推导过2-D Ising model理论解)方面有较大优势。两年的算法工程师与C++软件工程师经历，让我具备了面向对象程序设计经验，通过对系统软件的开发与维护，对软件架构设计具备一定经验。对图像的滤波，去噪，增强，特征提取有一定的基础。乐于接受新事物，深厚的数量基础也使我更容易理解新技术。

Ceres-Solver

Ceres-Solver是Google 2010年开源的一款非线性优化库。它主要能解决两类问题：

1. 带约束的非线性最小二乘问题
2. 一般的无约束优化问题

底层实现引用了Eigen与Lapack, Ceres是求解非线性优化问题的首选, 可以应用到SLAM中去。有现成的参考文档<http://ceres-solver.org/features.html>。

如果我们需要基于Ceres-Solver来开发求解一个非线性优化问题, 我们怎么把它封装成dll再发布?

其它的非线性优化库还有g2o。

参考书籍

1. 《凸优化》Stephen Boyd
2. 《最优化导论》Edwin K.P.Chong
3. 《线性与非线性规划》David G. Luenberger, Yinyu Ye
4. 《非线性规划》Dimitri P.Bertsekas
5. 《非线性优化计算方法》袁亚湘
6. 《Numerical Optimization》Jorge Nodedal, Stephen J. Wright
7. 《应用数值线性代数》James W. Demmel
8. 《Matrix Computations》Gene H. Golub, Charles F. Van Loan
9. 《分布式机器学习》刘铁岩
10. 《深度学习》Ian Goodfellow
11. 《语音识别实践》俞栋 邓力
12. 《The Elements of Statistical Learning》Hastie
13. 《稀疏统计学习及其应用》Trevor Hastie
14. 《热力学.统计物理》汪志诚
15. 《Introduction to Staistical Machine Learning》Masashi Sugiyama
16. 《Machine Learning A Bayesian and Optimization Perspection》Sergios Theodoridis
17. 《Machine Learning A Probabilistic Perspective》Kevin P. Murphy
18. 《统计机器学习》Vladimir N. Vapnik
19. 《统计机器学习》李航
20. 《机器学习》周志华
21. 《Hands-On Machine learning with Scikit-Learn&Tensorflow》
22. 《Pattern Recognition and Machine Learning》M.Bishop
23. 《深度学习核心技术与实践》猿辅导研究团队
24. 《剑指Offer》
25. 《计算广告》刘鹏 王超

第二章 C++

这一章主要涉及到C++编程,

1. 面向对象三大特征
2. 简单的模式设计(六大原则)
3. 进程间通讯
4. 线程同步异步
5. 泛型编程
6. STL
7. C++ 与Java的异同

主要是确保自己能够熟练的掌握与运用C++这道工具来实现自己的想法与算法。语言是一致的, 代码写多了才能体会到这一点, 语言只是外功, 数学, 建模思想才是内功; 外功可以在一两年内速成, 但是内功非十年八年难以达到登峰造极的境界, 而且有一点就是, 数学有点靠天赋, 数学好这是上天赐给你的礼物。

OOP

面向对象式语言的三大特征：封装，继承，多态。

面向对象编程(OOP)的八个基本概念：类与对象，封装，抽象，数据隐藏，多态，继承，动态绑定，消息传递。

C++最重要的特征就是代码重用。这通过继承与组合来实现。

封装

封装是一种捆绑机制，能把函数和数据捆绑成类这种紧凑的形式。封装的目的是实现数据的隐藏。数据和函数可以是私有的，也可以是公开的。私有的数据或函数只能在类里面访问，而公开的数据或代码则可以在类外访问。运用封装，可以把实现代码中的某些内容给隐藏起来。将函数代码与数据链接在一起，就可以生成对象，而这个对象则可以表示成某个以类为类型的变量。

继承

继承的目的是实现代码的复用。在原理上是通过虚函数表。

三种继承方式

继承方式\成员属性	public	protected	private
public继承	public	protected	子类无权访问
protect继承	protectd	protected	子类无权访问
private继承	private	private	子类无权访问

重载，覆盖和隐藏

成员函数被重载(**overloading**)，编译时

(1)相同的范围(同一个类中) (2)函数名字相同 (3)参数不同 (4)virtual关键字可有可无

覆盖/重写(**overriding**)是指派生类函数覆盖基类函数，运行时

(1)不同的范围(分别位于基类与派生类) (2)函数名字相同 (3)参数相同 (4)基类函数必须有virtual关键字

构造函数 析构函数 虚函数

构造函数不能使虚函数：虚函数是通过基类指针或引用来调用派生类的成员的，因此在调用之前，对象必须存在，而构造函数是为了创建对象的。从实现上看，virtual在构造函数调用后才建立，因而构造函数不可能成为虚函数。

基类的析构函数设置为虚函数，那么所有的派生类也默认为虚析构函数，即使没有带关键字Virtual。基类的析构函数设置为虚函数，目的是为了防止内存泄漏(需要指向基类的指针对派生类进行操作时，才有可能会导致内存泄漏，如果不需要这样操作，完全不需要设置成虚函数)。

构造函数，析构函数，重载

构造函数允许重载，所以在实例化对象的时候，可以根据传入参数的不同选择不同的构造函数，但是只会执行其中的一个，具体执行哪一个，按照传入的参数。

析构函数不允许重载，并且析构函数无参

二义性

C++可以多继承，但Java,C#只能是单继承。正由于C++可以多继承，因此会导致二义性问题，也就是当一个类D继承自类B，类C，而类B，类C又共同的继承于类A。因为类B，类C分别拥有类A的一份拷贝，因此类D拥有类A的两份拷贝，因此类D在调用类A的接口时，编译器不知道需要调用哪一份拷贝，从而产生错误。通常有两个解决方法：

1. 加上全局符确定调用哪一份拷贝，比如d.B::interface1()，调用属于类B的拷贝。
2. 使用虚拟继承，使得多重继承类D只拥有类A的一份拷贝，也就是：

```
class A {}
class B: virtual public A {};
class C: virtual public A {};
class D: public B, public C {};
int main()
{
    D d;
    A *pd = &d;//虚拟继承能转换成功，非虚拟继承不能
    return 0;
}
```

多重继承的优点是对象可以调用多个基类中的接口，缺点是容易出现继承上的二义性。

虚拟继承

这里要讨论为啥虚拟继承能消除二义性。

```
#include "StdAfx.h"
#include <iostream>
using namespace std;

class Parent
{
public:
```

```

Parent():num(0){cout<<"Parent"<<endl;}
Parent(int n):num(n){cout<<"Parent(int)"<<endl;}
private:
    int num;
};

class Child1 : virtual public Parent
{
public:
    Child1(){cout<<"Child1"<<endl;}
    Child1(int num):Parent(num){cout<<"Child1(int)"<<endl;}
};

class Child2 : virtual public Parent
{
public:
    Child2(){cout<<"Child2"<<endl;}
    Child2(int num):Parent(num){cout<<"Child2(int)"<<endl;}
};

class Dervied : public Child2,public Child1
{
public:
    Dervied():Child1(0),Child2(1){}
    Dervied(int num):Child1(num),Child2(num+1){}
};

void main()
{
    Dervied d(4);
}

```

多重继承类对象的构造顺序与其继承列表中基类的排列顺序一致(在上面的例子中是class Dervied : public Child2,public Child1; 因此先调用Child2的构造函数再调用Child1的构造函数);而不是与构造函数初始化列表中的顺序一致(Dervied(int num):Child1(num),Child2(num+1){}),因此结果输出如下:

```

C:\Windows\system32\cmd.exe
Parent
Child2(int)
Child1(int)
Press any key to continue . . .

```

如果不是虚拟继承，则输出结果如下，它有两份Parent的拷贝：

```
C:\Windows\system32\cmd.exe
Parent(int)
Child2(int)
Parent(int)
Child1(int)
Press any key to continue . . .
```

组合

集成与组合都是为了实现代码的复用。组合是通过把别的类作为自己的成员变量来使用其它类的功能。

集成是is-a关系, 组合是has-a关系。

多态

定义:允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。(发送消息就是函数调用)

实现多态的技术称为:动态绑定, 是指在执行期间判断所引用对象的实际类型, 根据其实际的类型调用其相应的方法。

多态的作用:消除类型之间的耦合关系。

子类覆盖父类的函数实现, 也就是子类重新实现父类的函数。不同的子类对相同的父类函数可以有不同的实现, 因此多态就是"一个接口, 多种实现"。

多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编译时并不确定, 而是在程序运行期间才确定, 即一个引用变量到底会指向哪个类的实例对象, 该引用变量发出的方法调用到底是哪个类中实现的方法, 必须在由程序运行期间才能确定。

多态原理与虚函数表

在实例化含有虚函数的类时, 会在对象中产生一个指向虚函数表的指针(大小为4字节), 虚函数表中存放着虚函数实现的地址, 每个虚函数均指向函数实现。

1. 对于基类:虚函数指向基类的虚函数实现。
2. 对于派生类:虚函数指向派生类的虚函数实现, 若继承来的虚函数, 派生类中没有实现, 则指向基类的虚函数实现。
3. 当派生类对象被视为基类对象时(派生类型转成基类型), 虽然类型被转变, 但虚函数表指针(地址)并没有发生改变, 仍然是派生类的虚函数指针, 所有在调用基类的函数时, 会调用派生类的虚函数实现。



虚函数与纯虚函数

- (1) 类里面如果有虚函数，这个函数是实现的，哪怕是空实现，它的作用是为了哪个让这个函数在它的子类里面可以被覆盖，这样编译器就可以使用后期绑定来达到多态了。纯虚函数只是以恶接口，是函数的声明而已，它要留到子类里里面去实现。
- (2) 虚函数在子类里面也可以不重载；但纯虚函数必须在子类里面实现，这和Java里面的接口一样。通常把很多函数加上virtual，是一个好的习惯，虽然牺牲一些性能，但是增加了面向对象的多态性，因为很难预料到父类里面的这个函数不在子类里面重新修改它的实现。
- (3) 虚函数的类可以用于“实作继承”，也就是说继承接口的时候同时也继承了父类的实现。当然，大家也可以有自己的实现。纯虚函数的类用于“界面继承”，即纯虚函数关注的是接口的统一性，实现由子类完成。
- (4) 带纯虚函数的类叫虚基类，这种基类不能直接生成对象，而只有被继承，并重写其纯虚函数之后，才能使用，这样的类也叫抽象类。

抽象基类与纯虚函数

纯虚函数在基类中没有定义，必须在子类中实现，就像Java或者C#中的接口。如果基类中含有一个或多个纯虚函数，那么它就属于抽象基类或者虚基类，抽象类，不能被实例化。

引入抽象基类的原因：

- (1) 为了方便的使用多态特性
- (2) 很多情况下，基类本身生成对象不合理(如基类是动物，派生类是阿猫，阿狗)。因此抽象基类不能够被实例化，它定义的纯虚函数相当于接口，能把派生类的共同行为提取出来。

多态的好处

继承和多态本质的目的是为了代码重用和解耦。而重用和解耦是为了减少开发时间和减少错误。最终达到降低成本的目的。

多态体现的就是“对扩展开放，对修改封闭”。封闭就是子类不能直接修改父类的其它函数，扩展开放也就是子类可以重写父类的函数。

1. 可替换性(substitutability)。多态对已经存在代码具有可替换性。
2. 可扩充性(extensibility)。多态对代码具有可扩充性，增加新的子类不影响已经存在类的多态性，继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现圆锥，半圆锥以及半球体的多态基础上，很容易增加球体类的多态性。
3. 接口性(interface-ability)。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。
4. 灵活性(flexibility)。它在应用中体现了灵活多样的操作，提高了使用效率。
5. 简化性(simplicity)。多态简化对应用软件的编写和修改过程，尤其在处理大量对象和操作时，这个特点尤为突出和重要。
5. 提高了代码的维护性(继承保证)
6. 提高了代码的扩展性(由多态保证)

表 B.1 面向对象的关键术语

术 语	定 义
属性	包含于对象内的数据变量
包含	两个对象实例之间的关系，其中包含者含有一个指向被包含者的指针
封装	对象实例的属性和服务与外部环境的隔离。服务只能通过名字调用，属性只能通过服务访问
继承	两个对象类之间的关系，子类可以获得父类的属性和服务
接口	一个和对象类紧密相关的描述。接口包含方法声明（没有实现）和常量值。接口不能实例化为一个对象
消息	对象交互的方式
方法	过程，是对象的组成部分，可在对象外部激活其执行某一功能
对象	现实世界实体的抽象
对象类	共享相同名字、属性集、服务的对象的有名集合
对象实例	一个赋予属性值的对象类的具体成员
多态性	指使用相同的服名，对外呈现相同的接口但却代表不同类型实体的多个对象存在
服务	在某一对象上执行某一操作的函数

C++内存问题

不像Java,.Net能自动管理内存, C++要手动的管理内存问题。因为要注意的问题如下：内存泄漏，野指针，访问越界。

C++类的大小计算

大小计算参考

1. 首先, 类大小的计算遵循结构体的对齐原则
2. 类的大小与成员变量有关, 与成员函数和静态成员(static)无关。
3. 虚函数对类的大小有影响, 是因为虚函数表指针带来的影响
4. 虚继承对类的大小有影响, 是因为虚基表指针带来的影响
5. 空类的大小是一个特殊情况, 空类的大小为1

对其的目的是加快访问速度, 节省空间在vc中默认是4字节对齐的, GNU gcc 也是默认4字节对齐。强制以多少字节对齐, #pragma pack (1)

计算就是, 前面n个成员变量的空间大小一定是最大成员变量长度K(比如double 为8)的整数倍。n个成员变量之后的成员, 能组合成K长度就组合, 不然长度就是K。

虚函数对类的大小有影响, 无论多少个虚函数, 只带来默认对其字节大小的空间, 比如vc中默认是4字节对齐的, GNU gcc 也是默认4字节对齐。因为虚函数在虚函数表中是连在一起的, 因此只需要记录开始位置。

对于指针而言, 其大小都是4, 类似于字节对齐。

在C++中, 内存分成5个区, 他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

使用未初始化的局部指针变量是件很危险的事, 所以, 在使用局部指针变量时, 一定要及时将其初始化。

```
int b = 3;
int *p = &b;
*p = 5;
```

C++浮点数表示方法

	31	30	23 22	0
32位	S	E	M	
	63	62	52 51	0
64位	S	E	M	http://blog.csdn.net/shuzfan

于是，

一个规格化的32位浮点数x的真值为：

$$x = (-1)^s \times (1.M) \times 2^{E-127}$$

一个规格化的64位浮点数x的真值为：

$$x = (-1)^s \times (1.M) \times 2^{E-1023}$$

析构函数是虚函数

```
Base* pTest = new Derived();
delete pTest;
```

如果基类Base的析构函数不是virtual函数，则上面代码delete时，衍生类Derived的析构函数不被调用，因此衍生类的资源不会释放。只有当Base类的析构函数是virtual的时候，上面的delete才能调用衍生类与基类的析构函数。

构造函数与析构函数的重载

构造函数可以重载：构造函数可以用多个，且可以带参数

析构函数不可以重载：析构函数只可能有一个，且不能带参数，即没有参数，没有返回值，没有函数类型。

Explicit

加在类的构造函数之前，防止类对象之间被隐式转换

List&Vector

list将元素按顺序储存在链表中。与向量(vectors)相比，它允许快速的插入和删除，但是随机访问却比较慢。

Vector:地址连续

List:地址不连续

线程与进程

总结

(1) 进程是系统分配资源的最小单位，线程是处理器调度的最小单位。(2) 进程是下面可以有多个线程，是线程的容器。进程下的线程共享一些资源，比如所有exe与dll模块的代码与数据。

(3) 进程间可以通过事件，信号量，互斥量，临界区来实现同步，跨机器可以通过socket来进行同步。socket主要通过IP来绑定机器，通过端口来绑定该机器中的进程，另外一个就是通信协议。通过进程间通信可以是实现跨进程，跨机器的单例。

(4) 同步与异步，阻塞与非阻塞：同步和异步关注的是消息通信机制：所谓同步就是在发出一个调用时，在没有得到结果之前，该调用就不返回。但是一旦调用者返回，就得到了返回值。就是调用者主动等待调用这个结果。

异步则相反：调用在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立即得到结果。而是在调用发出后，被调用者通过状态，通知来通知调用者，或者通过回调函数处理这个调用。

举个通俗的例子：你打电话问书店老板有没有《分布式系统》这本书，如果是同步通信机制，书店老板会说，你稍等，“我查一下”，然后开始查啊查，等查好了（可能是5秒，也可能是一天）告诉你结果（返回结果）。而异步通信机制，书店老板直接告诉你我查一下啊，查好了打电话给你，然后直接挂电话了（不返回结果）。然后查好了，他会主动打电话给你。在这里老板通过“回电”这种方式来回调。

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态：

阻塞调用是在调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才返回。非阻塞调用指在不能立即得到结果之前，该调用不会阻塞当前线程。

还是上面的例子，你打电话问书店老板有没有《分布式系统》这本书，你如果是阻塞式调用，你会一直把自己“挂起”，直到得到这本书有没有的结果，如果是非阻塞式调用，你不管老板有没有告诉你，你自己先一边去玩了，当然你也要偶尔过几分钟check一下老板有没有返回结果。在这里阻塞与非阻塞是否同步异步无关。跟老板通过什么方式回答你结果无关。

概念

本节要讨论的概念

1. 进程，线程
2. 内核对象
3. 单线程多线程
4. 同步与异步的概念
5. 线程同步(要明白这方实现方法的内核原理)
 - i. 关键段
 - ii. 读/写锁
 - iii. 条件变量

- iv. 信号量
 - v. 互斥量
 - vi. 事件
6. 线程池
 7. 分布式系统中的线程, 进程同步
 8. 内存管理

实际上按照《操作系统精髓与设计原理》来划分更为合理,但是难度大很多,还是弄懂一些基本概念就可以了。

1. 内核对象
2. 进程
3. 线程
4. 并发性:互斥与同步
5. 并发:饥饿与死锁
6. 内存管理与虚拟内存
7. 调度
 - i. 单处理器
 - ii. 多处理器

内核对象

什么是内核对象

windows中,诸如:时间,管道,互斥量,完成端口,进程,线程都是内核对象。这些内核对象虽然通过不同的系统API来创建,但是这些API有一个共同的特点就是,需要传入

SECURITY_ATTRIBUTES安全描述符结构体指针,并且返回句柄(HANDLE)。根据这个特点,可以依据创建它的函数是否允许传入SECURITY_ATTRIBUTES安全描述符。穿件内核对象之后,会返回一个值,也就是一个句柄Handle,可以通过这个句柄来操作这个内核对象。

内核对象为一个数据结构且只能为内核访问,因此应用程序无法在内存中找到这些数据结构并直接改变他们的内容,Micsoft规定了这个限制是为了确保内核对象结构保持状态的一致。所以Microsoft能自由的添加,删除和修改这些结构中的成员,同时不干扰任何程序正常运行。

进程与线程

- 1、进程是系统分配资源的最小单位。
- 2、线程是处理器调度的最小单位。
- 3、一个进程可以包含很多线程,且这些线程共享进程内的所有资源。

资源是什么?

然后又有大致三种线程模型:进程模型、用户级线程、内核级线程,三种模型如图所示



	进程	线程
组成	1. 一个内核对象，操作系统用它来管理进程。内核对象也是系统保存进程统计信息的地方。 2. 一个地址空间，其中包含所有可执行文件exe或dll模块的代码和数据。此外，它还包含动态内存分配，比如线程堆栈和堆的分配。	1. 线程的内核对象，操作系统用它来管理线程。系统还用内核对象来存放线程统计信息的地方。 2. 一个线程栈，用于维护线程执行时所需的所有函数参数和局部变量。
空间大小	32位系统最大4GB	Linux线程栈默认8Mb，在进行的堆栈分配线程栈

进程

进程是一个正在运行的程序的一个实例，它由以下两个方面组成：

1. 一个内核对象，操作系统用它来管理进程。内核对象也是系统保存进程统计信息的地方。
2. 一个地址空间，其中包含所有可执行文件exe或dll模块的代码和数据。此外，它还包含动态内存分配，比如线程堆栈和堆的分配。

进程需要记录哪些系统发生的信息？

进程要做任何事情，都必须让一个线程在它的上下文中运行，该线程负责执行进程地址空间包含的代码。

对32位系统，每个进程最大 $2^{32} = 4G$ 内存空间，但是是虚拟内存空间。系统分前半部分(2G)作为进程私有空间；后半部分(2G)作为公用空间，用来存放内核代码、驱动代码、IO缓存区。

一个进程可以有多个线程，所有线程都在进程的地址空间中“同时”执行代码。为此，每个线程都有自己的一组CPU寄存器和它自己的堆栈。每个进程至少有一个线程来执行进程地址空间中包含的代码。当系统创建一个进程时，会自动创建第一个线程，就是主线程。然后这个线程可以创建更多的线程，后者再创建更多的线程。如果没有线程执行进程地址空间包含的代码，进程就失去了继续存在的理由，这时，系统会自动销毁进程及其地址空间。

加载到进程地址空间的每一个可执行文件或者DLL文件都被赋予一个独一无二的实例。

进程间通讯

进程间通讯的四种方式:剪贴板、匿名管道、命名管道和邮槽

线程

线程是进程的不同执行序列，也就是说线程是独立运行的基本单元，也是CPU调度的基本单位。线程有时候称为轻量级进程，是CPU使用的基本单元；它由线程ID，程序计数器，寄存器集合和堆栈组成。它与属于同一进程的其它线程共享代码段，数据段和其它操作系统资源(如打开文件和信号)

在单个CPU计算机中，操作系统以轮询的方式为每个单独的线程分配时间量。

线程的两个组成部分：

1. 线程的内核对象，操作系统用它来管理线程。系统还用内核对象来存放线程统计信息的地方。
2. 一个线程栈，用于维护线程执行时所需要的所有函数参数和局部变量。

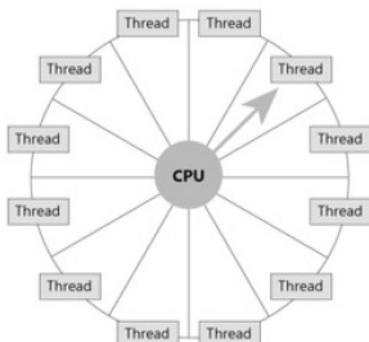


Figure 4-1: The operating system offers quanta to individual threads in a round-robin fashion on a single-CPU machine

线程之间通讯：

同一进程直接的线程：通过他们的公共地址空间交换信息，并访问进程中的共享资源。

不同进程之间的线程：通过在两个不同进程间建立共享内存交换信息。

线程的状态

线程有四种状态：新生状态，可运行状态，被阻塞状态，死亡状态。状态之间的转换如下图所示：

线程的生命周期图



堆, 栈大小

栈:Linux软限制为8Mb, 如果超过这个大小就出现segmentation fault。

堆:默认没有软限制, 只依赖于堆限制。

那么, 进程分配的空间是一个堆还是?

堆:大家共有的空间。

栈:是每个线程独有的。

线程同步

主要原理就是对进程中的公共资源进行保护, 不让其它线程访问它。

win32中四种主要的同步机制:

- (1)事件(Event)
- (2)信号量(semaphore)
- (3)互斥量(mutex)
- (4)临界区(Critical section)

内核对象被触发

进程内核对象在创建的时候总是处于未触发, 当进程终止的时候, 操作系统会自动使进程内核对象变成出发状态。

触发的意思就是, 外面的线程现在可以使用这些内核对象了。也就是等待该内核对象触发的线程变成可调度了。

触发有什么比较形象的比喻吗?

进程间通信(IPC)

这节主要讨论进程之间的通讯，实际上也是为了处理多线程之间的通讯服务的。不讨论单个进程之间的线程间的通讯是因为，单个进程之间可以通过共享该进程的全局变量来实现线程之间的通讯。因此，麻烦的是不同进程间的两个线程之间的通讯，也就是进程之间的通讯¹。

参考[多线程与多进程的区别\(小结\)](#)

Linux下进程通信的几种方式：

不同进程之间通信只能使用内核对象吗？

1. 管道(Pipe)及有名管道(Named pipe):管道具有单向性，可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。管道所传送的是无格式字节流。
2. 信号(Signal):信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction(实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了sigal函数)。
3. 报文(Message)队列(消息队列):消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承受无格式字节流以及缓冲区大小受限等缺点。
4. 共享内存¹:使得多个进程可以访问同一内存空间，是最快的可用IPC形式。是正对其它通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步与互斥。为了在多进程间交换信息，内核专门留出一块内存块，可以由需要访问的进程将其映射到自己的私有地址空间。进程就可以直接读写这一块内存而不需要进行数据的拷贝，从而大大提高效率。由于多个进程共享一段内存，因此需要依靠某种同步机制(如信号量)来达到进程间的同步及互斥。
5. 信号量(semaphore):主要作为进程间以及同意进程不同线程之间的同步手段。
6. 套接口(Socket)²:更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是用Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

套接字是支持TCP/IP的网络通信的基本操作单元，可以看作是不同主机之间的进程进行双向通信的端点，简单的说就是通信双方的一种约定，用套接字中相关函数来完成通信过程。

Socket的特性由3个属性确定，他们分别是：域，端口号，协议类型。

每一个基于TCP/IP网络通讯的程序(进程)都被赋予了唯一的端口和端口号，因此可以通过端口号来找到应用程序。端口是一个信息缓冲区，用于保留Socket中的输入/输出信息，端口号是一个16位无符号整数，范围是0-65535，以区别主机上的每一个程序(端口号就像房屋中的房间号)，低于256的端口号保留给标准应用程序，比如pop3的端口号就是110，每一个套接字都组合进了IP地址、端口，这样形成的整体就可以区别每一个套接字。

计算机如何识别单例

同一计算机中同一进程, 跨进程; 以及跨计算机怎么识别单例。

同一进程: 同一进程中的线程因为共享资源(exe或dll模块的代码和数据), 因此可以通过共享的资源老实现单例。

同一计算机, 不同进程: 上面的单例模式失效, 在两个应用程序中可以分别创建一个单例。因此必须通过IPC来实现跨进程的单例。比如创建一个单例的exe(taskmanager, windows media player), 就必须用到IPC, 比如: 信号量, 共享内存, 命名管道。

不同计算机之间: 此时必须通过Socket来实现不同计算机不同进程间的通讯来实现单例了。参数服务器。

总结来说, 实现跨进程, 跨电脑的单例涉及到进程间的通信。

比如window进程单例实现:

可以通过命名事件或者通过命名互斥量(Mutex)来实现。必须是命名的, 不是命名的不可以实现跨进程的单例。

一个问题是, 系统怎么管理这些事件, 互斥量等内核对象? 他们怎么知道这些内核对象已经存在?

答案分析: 当我们创建一个内核对象的时候, 比如当我们创建一个命名的Mutex时, 系统首先会在同一内核对象命名空间查想看是否已经存在一个同名的内核对象。如果存在, 则内核接着检查对象的类型, 如果类型相同, 系统会接着执行一次安全检查, 验证调用者是否拥有对该对象的完全访问限制。如果答案是肯定的, 系统就会在进程的句柄表中查找一个空白记录项, 并将其初始化为指向现有的内核对象。如果对象的类型不匹配, 或者调用者被拒绝访问, CreateMutex就会失败(返回NULL)。可以参照《Windows核心编程》P47。

```
auto h = ::CreateEvent(NULL, FALSE, TRUE, _T("already running"));
auto err = GetLastError();
if (err == ERROR_ALREADY_EXISTS)
{
    MessageBox(NULL, _T("already exist"), _T("ERROR"), SW_NORMAL);
    return FALSE;
}

auto handle = ::CreateMutex(NULL, TRUE, _T("already_running"));
auto err = GetLastError();
if (err == ERROR_ALREADY_EXISTS)
{
    MessageBox(NULL,_T("already exist"),_T("ERROR"),SW_NORMAL);
    return FALSE;
}
if (handle)
{
    ::ReleaseMutex(handle);
}
```

内核对象命名空间

所有这些对象都共享单个名空间。因此下面的用法是错的。

所以为了防止名字的冲突，建议创建一个GUID，并将GUID的字符串表达式用作对象名。

```
HANDLE hMutex = CreateMutex(NULL, FALSE, "JeffObj");
HANDLE hSem = CreateSemaphore(NULL, 1, 1, "JeffObj");
DWORD dwErrorCode = GetLastError();
```

多线程和多进程的对比

对比维度	多进程	多线程	总结
数据共享，同步	数据共享复杂，需要IPC;数据是分开的，同步简单	因为共享进程数据，数据共享简单，但也由此导致同步复杂	各有优势
内存，CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁，切换	创建销毁，切换复杂，速度慢	创建销毁，切换简单，速度很快	线程占优
编程，调试	编程简单，调试简单	编程简单，调试复杂	进程占优
可靠性	进程间不会相互影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适用于多核，多机分布式；如果一台机器不够，扩展到多台机器比较简单	适用于多核分布式	进程占优

线程同步与互斥

上面讨论了不同进程之间通讯的问题，现在讨论同一进程之间的多线程通信问题。由于同一进程之间多个线程共享相同的资源，因此必须处理资源共享中的同步问题。

1. 线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖于另外一个线程的消息，当他没有得到另外一个线程的时候应该等待，直到消息到达时才被唤醒。
2. 线程互斥是指对于共享的操作系统资源(比如全局变量)，在各线程访问时的排他性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。

线程互斥是一种特殊的线程同步。实际上，互斥与同步对应线程间通信发生的两种情况。

3. 当有多个线程访问共享资源而不使资源被破坏时。
4. 当一个线程需要将某个任务已经完成的情况通知另外一个或多个线程时。

用户模式和内核模式

线程同步分为用户模式的线程同步和内核对象的线程同步两大类。

用户模式中线程的同步方法主要有原子访问和临界区等方法。其特点是同步速度特别快，适合于对线程运行速度要求较高的场合。

内核对象的线程同步主要由事件、等待定时器、信号量以及信号灯等内核对象构成。由于这种同步机制使用了内核对象，使用时必须将线程从用户模式切换到内核模式，而这种切换一般需要消耗近千个CPU周期，因此同步速度较慢，但在适用性上却要远优于用户模式的线程同步方式。

事件

事件是一个内核对象，不同进程间可以通过命名事件来通信。

信号量

和其他核心对象一样，信号量也可以通过名字跨进程访问。

互斥量

互斥量是为协调共同对一个共享资源的单独访问而设计的。互斥量是内核对象，所以它比临界区更加耗费资源，但是它可以命名，因此可以被其它进程访问。

原子访问

当必须以原子操作方式来修改单个值时，互锁访问函数是相当有用的。所谓原子访问，是指线程在访问资源时能够确保所有其他线程都不在同一时间内访问相同的资源。

InterlockedExchangeAdd保证对变量的访问具有“原子性”。互锁访问的控制速度非常快，调用一个互锁函数的CPU周期通常小于50，不需要进行用户方式与内核方式的切换（该切换通常需要运行1000个CPU周期）。

互锁访问函数的缺点在于其只能对单一变量进行原子访问，如果要访问的资源比较复杂，仍要使用临界区或互斥。

临界区

临界区是通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。

同步方式	方式	怎么跨进程	模式
临界区		不可以	用户
原子访问			用户

事件	匿名事件, 命名事件	命名事件	内核
信号量	用于同步	命名	内核
互斥量	用于互斥	命名	内核
全局变量	同一进程下线程共享资源	不可以	用户

模板类与模板函数

Why can templates only be implemented in the header file?

Clarification: header files are not the _only _portable solution. But they are the most convenient portable solution Error LNK2019 unresolved external symbol "public: int __thiscall Algo<int>::LongestIncreaseSubsequence(int * const,int)" (?LongestIncreaseSubsequence@? \$Algo@H@@@QAEHQAH@Z) referenced in function _main Algo C:\Data\ShareFolder\Group\Tim\Code\Algo\Algo\Main.obj

当模板声明在头文件，实现在cpp中，一般会出现上面的问题，原因是
Algo<int> Alg实例化的时候，编译器会去创建一个int型的Algo的新类，以及
LongestIncreaseSubsequence(T arr[], int arrSize)方法，因此编译器会去找这个方法的实现，如果没有找到，自然会报错。因此有如下两个方法

1. 在头文件中实现
2. 在头文件中声明，再在cpp中显式实例化(explicit instantiations) template class Algo<int>;并实现模板函数

方式1:实现在头文件里

```
#pragma once
#include <algorithm>
using namespace std;

template<class T>
class Algo
{
public:
    Algo();
    ~Algo();
public:
    int LongestIncreaseSubsequence(T arr[], int arrSize)
    {
        int *maxLen = new int[arrSize]();
        for (int i = 0; i < arrSize; i++)
        {
            maxLen[i] = 1;
        }
        for (int j = 1; j < arrSize; j++)
        {
            int maxLenJ = 1;
            for (int i = 0; i < j; i++)
            {
                if (arr[i] < arr[j])
                {
                    maxLenJ = maxLen[i] + 1;
                    maxLen[j] = std::max({ maxLen[j] ,maxLenJ });
                }
            }
        }
    }
}
```

```

        }
    }
    return maxLen[arrSize - 1];
}
};

```

```

//调用函数
#include "stdafx.h"
#include "Algo.h"
#include <iostream>
using namespace std;

int main()
{
    int param[6] = {5,3,4,8,6,7};
    int arrSize = sizeof(param) / sizeof(int);
    Algo<int> Alg;
    int maxLength = Alg.LongestIncreaseSubsequence(param, arrSize);
    cout << "Max length: " << maxLength << endl;
    return 0;
}

```

方式二：在头文件中声明，CPP中定义并对每种实例进行声明

```

Algo.h
#pragma once
#include <algorithm>
using namespace std;

template<class T>
class Algo
{
public:
    Algo() {};
    ~Algo() {};
public:
    int LongestIncreaseSubsequence(T arr[], int arrSize);
}

```

```

// Algo.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include "Algo.h"
#include <algorithm>
using namespace std;

template class Algo<int>; //必须先声明，才可以用
template class Algo<double>;
template class Algo<char>;
template<class T>
int Algo<T>::LongestIncreaseSubsequence(T arr[], int arrSize)

```

```

{
    int *maxLen = new int[arrSize]();
    for (int i = 0; i < arrSize; i++)
    {
        maxLen[i] = 1;
    }
    for (int j = 1; j < arrSize; j++)
    {
        int maxLenJ = 1;
        for (int i = 0; i < j; i++)
        {
            if (arr[i] < arr[j])
            {
                maxLenJ = maxLen[i] + 1;
                maxLen[j] = std::max({ maxLen[j] ,maxLenJ });
            }
        }
    }
    return maxLen[arrSize - 1];
}

```

If all you want to know is how to fix this situation, read the next two FAQs.

But in order to understand why things are the way they are, first accept these facts:

A template is not a class or a function. A template is a “pattern” that the compiler uses to generate a family of classes or functions.

In order for the compiler to generate the code, it must see both the template definition (not just declaration) and the specific types/whatever used to “fill in” the template. For example, if you’re trying to use a `Foo<int>`, the compiler must see both the `Foo` template and the fact that you’re trying to make a specific `Foo<int>`.

Your compiler probably doesn’t remember the details of one .cpp file while it is compiling another .cpp file. It could, but most do not and if you are reading this FAQ, it almost definitely does not. BTW this is called the “separate compilation model.”

Now based on those facts, here’s an example that shows why things are the way they are. Suppose you have a template `Foo` defined like this:

```

template<typename T>
class Foo {
public:
    Foo();
    void someMethod(T x);
private:
    T x;
};

```

Along with similar definitions for the member functions:

```

template<typename T>
Foo<T>::Foo()
{
    // ...
}
template<typename T>
void Foo<T>::someMethod(T x)
{

```

```

// ...
}

Now suppose you have some code in file Bar.cpp that uses Foo<int>:

// Bar.cpp
void blah_blah_blah()
{
    // ...
    Foo<int> f;
    f.someMethod(5);
    // ...
}

Clearly somebody somewhere is going to have to use the “pattern” for the constructor definition and for the someMethod() definition and instantiate those when T is actually int. But if you had put the definition of the constructor and someMethod() into file Foo.cpp, the compiler would see the template code when it compiled Foo.cpp and it would see Foo<int> when it compiled Bar.cpp, but there would never be a time when it saw both the template code and Foo<int>. So by rule #2 above, it co

```

uld never generate the code for Foo<int>::someMethod(). A note to the experts: I have obviously made several simplifications above. This was intentional so please don't complain too loudly. If you know the difference between a .cpp file and a compilation unit, the difference between a class template and a template class, and the fact that templates really aren't just glorified macros, then don't complain: this particular question/answer wasn't aimed at you to begin with. I simplified things so newbies would “get it,” even if doing so offends some experts.

const对象只能访问**const**成员函数。因为**const**对象表示其不可改变，而非**const**成员函数可能在内部改变了对象，所以不能调用。

而非**const**对象既能访问**const**成员函数，也能访问非**const**成员函数，因为非**const**对象表示其可以改变。

```

#ifndef _MATRIX_H
#define _MATRIX_H
#include <iostream>
#include <algorithm>
using namespace std;

template<class T>
class CMatrix
{
public:
    CMatrix() {};
    CMatrix(int rows, int columns);
    CMatrix(const CMatrix&);
    ~CMatrix() {};
public:
    void Set(int row, int column, T val);
    T Get(int row, int column) const;
    CMatrix operator +(const CMatrix &mat2);
}

```

```

CMATRIX operator -(const CMATRIX &mat2);
CMATRIX operator *(const CMATRIX &mat2);
CMATRIX operator *(const T div);
CMATRIX operator /(const T div);
CMATRIX I(int n);
int Rows() const{ return mRows; }
int Columns() const{ return mColumns; }
int Length() { return mRows*mColumns; }
friend ostream &operator<<(ostream &os, const CMATRIX &mat)
{
    int row = mat.Rows();
    int col = mat.Columns();
    //mat是const, 因此只能访问const成员函数

    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            os << fixed << mat.Get(i, j) << '\t';
        }
        if (i < row)
        {
            os << "\n";
        }
    }
    os << "\n";
    return os;
};

private:
    int mRows;
    int mColumns;
    T* startElement;

};

#endif

```

设计模式六大基本原则¹

这里主要陈述模式设计的六大基本原则。重要的是明白这些原则的背后还是为什么服务的，也就是这六个原则背后的更少的基本原则是什么？比如代码易于管理，代码鲁棒性高等等，要自己提炼出更高一些的原则。

1. 单一职责原则
2. 里氏替换原则
3. 依赖倒置原则
4. 接口隔离原则
5. 迪米特法则
6. 开闭原则

单一职责原则

定义：不要存在多于一个导致类变更的原因。通俗的说，即一个类只负责一项职责。

问题由来：类T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。

解决方案：遵循单一职责原则。分别建立两个类T1、T2，使T1完成职责P1功能，T2完成职责P2功能。这样，当修改类T1时，不会使职责P2发生故障风险；同理，当修改T2时，也不会使职责P1发生故障风险。

遵循单一职责原的优点有：

1. 可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；
2. 提高类的可读性，提高系统的可维护性；
3. 变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都适用单一职责原则。

里氏替换原则

定义1：如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

定义2：所有引用基类的地方必须能透明地使用其子类的对象。

问题由来：有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。

解决方案：当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2

外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障。

依赖倒置原则

定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

问题由来：类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

解决方案：将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

接口隔离原则

定义：客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。问题由来：类A通过接口I依赖类B，类C通过接口I依赖类D，如果接口I对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。

解决方案：将臃肿的接口I拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。

迪米特法则

定义：一个对象应该对其他对象保持最少的了解。

问题由来：类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

解决方案：尽量降低类与类之间的耦合。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在1987年由美国Northeastern University的Ian Holland提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

开闭原则

定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

问题由来：在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。

解决方案：当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。

开闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模糊的一个了，它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。以前，如果有人告诉我“你进行设计的时候一定要遵守开闭原则”，我会觉得他什么都没说，但貌似又什么都说了。因为开闭原则真的太虚了。

在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。其实，我们遵循设计模式前面5大原则，以及使用23种设计模式的目的就是遵循开闭原则。也就是说，只要我们对前面5项原则遵守的好了，设计出的软件自然是符合开闭原则的，这个开闭原则更像是前面五项原则遵守程度的“平均得分”，前面5项原则遵守的好，平均分自然就高，说明软件设计开闭原则遵守的好；如果前面5项原则遵守的不好，则说明开闭原则遵守的不好。

其实笔者认为，开闭原则无非就是想表达这样一层意思：用抽象构建框架，用实现扩展细节。因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

说到这里，再回想一下前面说的5项原则，恰恰是告诉我们用抽象构建框架，用实现扩展细节的注意事项而已：单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。最后说明一下如何去遵守这六个原则。对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。

¹. 设计模式六大原则 <http://www.uml.org.cn/sjms/201211023.asp> ↩

设计模式

这里主要先介绍简单工厂，工厂方法与抽象工厂，对每种模式要讨论他们的优缺点。

简单工厂

简单工厂类：工厂类Factory负责创建所有的产品，通过switch结构来创建不同的产品（ProductA, ProductB, ProductC...）。但是每次新加产品必须修改工厂类Factory，在switch中添加新的case。因此，缺点是添加新产品后，整个工厂类必须重新编译。



ProductA, ProductB和ProductC继承自虚拟类Product, Show方法是不同产品的自描述。Factory依赖于ProductA, ProductB, ProductC, Factory根据不同的条件创建不同的Product对象。

没增加一个产品类，比如ProductD，则需要在enum PRODUCTTYPE中添加TypeD，并且在class Factory中创建产品函数的switch结构中新加一个case。

需要修改的地方主要是Factory，这里的工厂类的作用是生产不同种类的产品Product。

```
#include <iostream>
```

```

#include <vector>
using namespace std;

typedef enum ProductTypeTag{
    TypeA,
    TypeB,
    TypeC}PRODUCTTYPE;

// Here is the product class
class Product{public:
    virtual void Show() = 0;};

class ProductA : public Product{
public:
    void Show()
    {
        cout<<"I'm ProductA"<<endl;
    }};
}

class ProductB : public Product{
public:
    void Show()
    {
        cout<<"I'm ProductB"<<endl;
    }};
}

class ProductC : public Product{
public:
    void Show()
    {
        cout<<"I'm ProductC"<<endl;
    }};
}

// Here is the Factory class
class Factory{
public:
    Product* CreateProduct(PRODUCTTYPE type)
    {
        switch (type)
        {
        case TypeA:
            return new ProductA();

        case TypeB:
            return new ProductB();

        case TypeC:
            return new ProductC();

        default:
            return NULL;
        }
    }};
}

int main(int argc, char *argv[])
{

```

```

// First, create a factory object
Factory *ProductFactory = new Factory();
Product *productObjA = ProductFactory->CreateProduct(TypeA);
if (productObjA != NULL)
    productObjA->Show();

Product *productObjB = ProductFactory->CreateProduct(TypeB);
if (productObjB != NULL)
    productObjB->Show();

Product *productObjC = ProductFactory->CreateProduct(TypeC);
if (productObjC != NULL)
    productObjC->Show();

delete ProductFactory;
ProductFactory = NULL;

delete productObjA;
productObjA = NULL;

delete productObjB;
productObjB = NULL;

delete productObjC;
productObjC = NULL;

return 0;
}

```

工厂方法模式

由于简单工厂模式的局限性，比如：工厂现在能生产ProductA、ProductB和ProductC三种产品了，此时，需要增加生产ProductD产品；那么，首先是不是需要在产品枚举类型中添加新的产品类型标识，然后，修改Factory类中的switch结构代码。是的，这种对代码的修改，对原有代码的改动量较大，易产生编码上的错误（虽然很简单，如果工程大了，出错也是在所难免的！！！）。这种对代码的修改是最原始，最野蛮的修改，本质上不能称之为对代码的扩展。同时，由于对已经存在的函数进行了修改，那么以前进行过的测试，都将是无效的，所有的测试，都将需要重新进行，所有的代码都需要进行重新覆盖。这种，增加成本，不能提高效率的事情，在公司是绝对不允许的（除非昏庸的PM）。出于种种原因，简单工厂模式，在实际项目中使用的较少。那么该怎么办？怎么办呢？需要对原有代码影响降到最小，同时能对原有功能进行扩展。

工厂方法模式是在简单工厂模式的基础上，对“工厂”添加了一个抽象层。将工厂共同的动作抽象出来，作为抽象类，而具体的行为由子类本身去实现，让子类去决定生产什么样的产品。



如图, FactoryA专心负责生产ProductA, FactoryB专心负责生产ProductB, FactoryA和FactoryB之间没有关系;如果到了后期, 如果需要生产ProductC时, 我们则可以创建一个FactoryC工厂类, 该类专心负责生产ProductC类产品。由于FactoryA、FactoryB和FactoryC之间没有关系, 当加入FactoryC加入时, 对FactoryA和FactoryB的工作没有产生任何影响, 那么对代码进行测试时, 只需要单独对FactoryC和ProductC进行单元测试, 而FactoryA和FactoryB则不用进行测试, 则可省去大量无趣无味的测试工作。

其特点是:每个工厂只生产单一的产品, 不同工厂生产不同的产品, 工厂间相互独立, 新加的产品(或工厂)不会对原有的产品产生影响。

工厂方法模式的意义是定义一个创建工作对象的工厂接口, 将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建, 这样核心类成为一个抽象工厂角色, 仅负责具体工厂子类必须实现的接口, 这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。

1. 在设计的初期, 就考虑到产品在后期会进行扩展的情况下, 可以使用工厂方法模式;
2. 产品结构较复杂的情况下, 可以使用工厂方法模式;由于使用设计模式是在详细设计时, 就需要进行定夺的, 所以, 需要权衡多方面的因素, 而不能为了使用设计模式而使用设计模式。

```

#include <iostream>
using namespace std;

class Product{public:
    virtual void Show() = 0;};

class ProductA : public Product{
public:
    void Show()
    {
        cout<< "I'm ProductA" << endl;
    }};
}

class ProductB : public Product{
public:
    void Show()
    {
        cout<< "I'm ProductB" << endl;
    }};
}

```

```

    }};

class Factory{
public:
    virtual Product *CreateProduct() = 0;

class FactoryA : public Factory{
public:
    Product *CreateProduct()
    {
        return new ProductA ();
    };

class FactoryB : public Factory{
public:
    Product *CreateProduct()
    {
        return new ProductB ();
    };

int main(int argc , char *argv [])
{
    Factory *factoryA = new FactoryA ();
    Product *productA = factoryA->CreateProduct();
    productA->Show();

    Factory *factoryB = new FactoryB ();
    Product *productB = factoryB->CreateProduct();
    productB->Show();

    if (factoryA != NULL)
    {
        delete factoryA;
        factoryA = NULL;
    }

    if (productA != NULL)
    {
        delete productA;
        productA = NULL;
    }

    if (factoryB != NULL)
    {
        delete factoryB;
        factoryB = NULL;
    }

    if (productB != NULL)
    {
        delete productB;
        productB = NULL;
    }
    return 0;
}

```

}

抽象工厂模式

之前讲到了C++设计模式——工厂方法模式，我们可能会想到，后期产品会越来越多了，建立的工厂也会越来越多，工厂进行了增长，工厂变的凌乱而难于管理；由于工厂方法模式创建的对象都是继承于Product的，所以工厂方法模式中，每个工厂只能创建单一类型的产品，当需要生产一种全新的产品（不继承自Product）时，发现工厂方法是心有余而力不足。

举个例子来说：一个显示器电路板厂商，旗下的显示器电路板种类有非液晶的和液晶的；这个时候，厂商建造两个工厂，工厂A负责生产非液晶显示器电路板，工厂B负责生产液晶显示器电路板；工厂一直就这样运行着。有一天，总经理发现，直接生产显示器的其余部分也挺挣钱，所以，总经理决定，再建立两个工厂C和D；C负责生产非液晶显示器的其余部件，D负责生产液晶显示器的其余部件。此时，旁边参谋的人就说了，经理，这样做不好，我们可以直接在工厂A中添加一条负责生产非液晶显示器的其余部件的生产线，在工厂B中添加一条生产液晶显示器的其余部件的生产线，这样就可以不用增加厂房，只用将现有厂房进行扩大一下，同时也方便工厂的管理，而且生产非液晶显示器电路板的技术人员对非液晶显示的其余部件的生产具有指导的作用，生产液晶显示器电路板也是同理。总经理发现这是一个不错的主意。

再回到软件开发的过程中来，工厂A和B就是之前所说的C++设计模式——工厂方法模式；总经理再次建立工厂C和D，就是重复C++设计模式——工厂方法模式，只是生产的产品不同罢了。这样做的弊端就如参谋所说的那样，增加了管理成本和人力成本。在面向对象开发的过程中，是很注重对象管理和维护的，对象越多，就越难进行管理和维护；如果工厂数量过多，那么管理和维护的成本将大大增加；虽然生产的是不同的产品，但是可以二者之间是有微妙的关系的，如参谋所说，技术人员的一些技术经验是可以借鉴的，这就相当于同一个类中的不同对象，之间是可以公用某些资源的。那么，增加一条流水线，扩大厂房，当然是最好的主意了。

实际问题已经得到了解决，那么如何使用设计模式模拟这个实际的问题呢？那就是接下来所说的抽象工厂模式。

现在要讲的抽象工厂模式，就是工厂方法模式的扩展和延伸，但是抽象工厂模式，更有一般性和代表性；它具有工厂方法具有的优点，也增加了解决实际问题的能力。

如图所示，抽象工厂模式，就好比是两个工厂方法模式的叠加。抽象工厂创建的是一系列相关的对象，其中创建的实现其实就是采用的工厂方法模式。在工厂Factory中的每一个方法，就好比是一条生产线，而生产线实际需要生产什么样的产品，这是由Factory1和Factory2去决定的，这样便延迟了具体子类的实例化；同时集中化了生产线的管理，节省了资源的浪费。



适用场合

工厂方法模式适用于产品种类结构单一的场合，为一类产品提供创建的接口；而抽象工厂方法适用于产品种类结构多的场合，主要用于创建一组（有多个种类）相关的产品，为它们提供创建的接口；就是当具有多个抽象角色时，抽象工厂便可以派上用场。

```
#include <stdio.h>
using namespace std;

//*****ProductA*****
class CProductA {
public:
    virtual void Show() = 0;
};

class CProductA1 :public CProductA {
    void Show()
    {
        printf("I am product A1!\n");
    }
};

class CProductA2 :public CProductA {
    void Show()
    {
        printf("I am product A2!\n");
    }
};
```

```

// ****ProductB*****
class CProductB {
public:
    virtual void Show() = 0;
};

class CProductB1 :public CProductB {
    void Show()
    {
        printf("I am product B1!\n");
    }
};

class CProductB2 :public CProductB {
    void Show()
    {
        printf("I am product B2!\n");
    }
};

class CFactory {
public:
    virtual CProductA* CreateProductA() = 0;
    virtual CProductB* CreateProductB() = 0;
};

// ****CFactory1*****
class CFactory1 :public CFactory {
public:
    CProductA* CreateProductA()
    {
        return new CProductA1;
    }

    CProductB* CreateProductB()
    {
        return new CProductB1;
    }
};

// ****CFactory2*****
class CFactory2 :public CFactory {
public:
    CProductA* CreateProductA()
    {
        return new CProductA2;
    }

    CProductB* CreateProductB()
    {
        return new CProductB2;
    }
};

int main()

```

```

{
    CFactory* factory1 = new CFactory1();
    CProductA* productA1 = factory1->CreateProductA();
    CProductB* productB1 = factory1->CreateProductB();
    productA1->Show();
    productB1->Show();

    CFactory* factory2 = new CFactory2();
    CProductA* productA2 = factory2->CreateProductA();
    CProductB* productB2 = factory2->CreateProductB();
    productA2->Show();
    productB2->Show();

    ////////////////////////////////////////////////////delete factory 1/////////////////////////////////////////////////
    if (factory1 != NULL)
    {
        delete factory1;
        factory1 = NULL;
    }

    if (productA1 != NULL)
    {
        delete productA1;
        productA1 = NULL;
    }

    if (productB1 != NULL)
    {
        delete productB1;
        productB1 = NULL;
    }

    ////////////////////////////////////////////////////delete factory 2/////////////////////////////////////////////////
    if (factory2 != NULL)
    {
        delete factory2;
        factory2 = NULL;
    }

    if (productA2 != NULL)
    {
        delete productA2;
        productA2 = NULL;
    }

    if (productB2 != NULL)
    {
        delete productB2;
        productB2 = NULL;
    }
    return 0;
}

```

总结

(1)简单工厂类:

工厂类Factory负责创建所有的产品, 通过switch结构来创建不同的产品(ProductA,ProductB,ProductC...)。但是每次新加产品必须修改工厂类Factory, 在switch中添加新的case。因此, 缺点是添加新产品后, 整个工厂类必须重新编译。

(2)工厂方法模式:

为了克服简单工厂类在添加新产品必须重新编译所有产品的问题, 工厂方法模式为每个产品新建一个工厂, 即:工厂的职能是单一的, 之生产特定的产品。所有的子工厂类继承于一个父工厂类, 父工厂类的唯一接口是创建产品, 在客户端来选择创建不同的工厂。

3:抽象工厂类:

为克服工厂方法模式只能创建单一产品的问题而引进了抽象工厂类。在这里, 父工厂类有不同的接口, 子工厂类实现创建不同的产品。每种产品都有一个父类A, 在子类里面, 该产品有不同的实现, 并在不同的子工厂类都可以创建这个产品(的子类)。

常用的python数据处理函数

pandas

处理结构化数据(Data Retrieval) DataFrame

List of Dictionaries to DataFrame

```
In[27]: import pandas as pd
In[28]: d = [{'city':'Delhi','data':1000},
...: {'city':'Bangalore','data':2000},
...: {'city':'Mumbai','data':1000}]
In[29]: pd.DataFrame(d)
Out[29]:
city data
0 Delhi 1000
1 Bangalore 2000
2 Mumbai 1000
In[30]: df = pd.DataFrame(d)
In[31]: df
Out[31]:
city data
0 Delhi 1000
1 Bangalore 2000
2 Mumbai 1000
```

CSV Files to DataFrame

```
import pandas as pd
In [2]: city_data = pd.read_csv(filepath_or_buffer='simplemaps-worldcities-basic.csv')
In [3]: city_data.head(n=10)
Out[3]:
city city_ascii lat lng pop country \
0 Qal eh-ye Now Qal eh-ye 34.983000 63.133300 2997 Afghanistan
1 Chaghcharan Chaghcharan 34.516701 65.250001 15000 Afghanistan
2 Lashkar Gah Lashkar Gah 31.582998 64.360000 201546 Afghanistan
3 Zarjanj Zarjanj 31.112001 61.886998 49851 Afghanistan
4 Tarin Kowt Tarin Kowt 32.633298 65.866699 10000 Afghanistan
5 Zareh Sharan Zareh Sharan 32.850000 68.416705 13737 Afghanistan
6 Asadabad Asadabad 34.866000 71.150005 48400 Afghanistan
7 Taloqan Taloqan 36.729999 69.540004 64256 Afghanistan
8 Mahmud-E Eraqi Mahmud-E Eraqi 35.016696 69.333301 7407 Afghanistan
9 Mehtar Lam Mehtar Lam 34.650000 70.166701 17345 Afghanistan
```

Database to DataFrame

可以从数据库中读取数据

```
server = 'xxxxxxxx' # Address of the database server
user = 'xxxxxx' # the username for the database server
password = 'xxxxx' # Password for the above user
database = 'xxxxx' # Database in which the table is present
conn = pymssql.connect(server=server, user=user, password=password, database=database)
query = "select * from some_table"
df = pd.read_sql(query, conn)
```

Data Access

Head and Tail

head提供dataframe前面几行数据, tail提供dataframe倒数几行数据。

```
In [11]: city_data.tail()
Out[11]:
city city_ascii lat lng pop country \
7317 Mutare Mutare -18.970019 32.650038 216785.0 Zimbabwe
7318 Kadoma Kadoma -18.330006 29.909947 56400.0 Zimbabwe
7319 Chitungwiza Chitungwiza -18.000001 31.100003 331071.0 Zimbabwe
7320 Harare Harare -17.817790 31.044709 1557406.5 Zimbabwe
7321 Bulawayo Bulawayo -20.169998 28.580002 697096.0 Zimbabwe
```

Slicing and Dicing

数据切片, 提取特定标签的数据。

```
In [12]: series_es = city_data.lat//提取lat标签的数据
In [13]: type(series_es)
Out[13]: pandas.core.series.Series
In [14]: series_es[1:10:2]
Out[14]:
1 34.516701
3 31.112001
5 32.850000
7 36.729999
9 34.650000
Name: lat, dtype: float64
In [15]: series_es[:7]
Out[15]:
0 34.983000
1 34.516701
2 31.582998
3 31.112001
4 32.633298
5 32.850000
6 34.860000
```

```
Name: lat, dtype: float64
In [23]: series_es[:-7315]
Out[23]:
0 34.983000
1 34.516701
2 31.582998
3 31.112001
4 32.633298
5 32.850000
6 34.866000
Name: lat, dtype: float64
```

iloc

提取多少行(数据样本数目), 多少列(标签数目)。比如提取前4个标签的前5行数据。

```
In [28]: city_data.iloc[:5,:4]
Out[28]:
city city_ascii lat lng
0 Qal eh-ye Now Qal eh-ye 34.983000 63.133300
1 Chaghcharan Chaghcharan 34.516701 65.250001
2 Lashkar Gah Lashkar Gah 31.582998 64.360000
3 Zaranj Zaranj 31.112001 61.886998
4 Tarin Kowt Tarin Kowt 32.633298 65.866699
```

带判断的搜索

select cities that have population of more than 10 million and select columns that start with the letter l:

```
In [56]: city_data[city_data['pop'] >
10000000][city_data.columns[pd.Series(city_data.columns).str.
startswith('l')]]
Out[53]:
lat lng
360 -34.602502 -58.397531
1171 -23.558680 -46.625020
2068 31.216452 121.436505
3098 28.669993 77.230004
3110 19.016990 72.856989
3492 35.685017 139.751407
4074 19.442442 -99.130988
4513 24.869992 66.990009
5394 55.752164 37.615523
6124 41.104996 29.010002
7071 40.749979 -73.980017
```

Data Operations

Using the values attribute of the output dataframe, we can treat it in the same way as a numpy array. 利用值属性，我们可以为dataframe填充数据，做缺失值处理。

```
In [55]: df = pd.DataFrame(np.random.randn(8, 3),
...: columns=['A', 'B', 'C'])
In [56]: df
Out[56]:
A B C
0 -0.271131 0.084627 -1.707637
1 1.895796 0.590270 -0.505681
2 -0.628760 -1.623905 1.143701
3 0.005082 1.316706 -0.792742
4 0.135748 -0.274006 1.989651
5 1.068555 0.669145 0.128079
6 -0.783522 0.167165 -0.426007
7 0.498378 -0.950698 2.342104
In [58]: nparray = df.values
In [59]: type(nparray)
Out[59]: numpy.ndarray
```

Missing Data and the fillna Function

填补缺失值。

```
In [65]: df.iloc[4,2] = NA
In [66]: df
Out[66]:
A B C
0 -0.271131 0.084627 -1.707637
1 1.895796 0.590270 -0.505681
2 -0.628760 -1.623905 1.143701
3 0.005082 1.316706 -0.792742
4 0.135748 -0.274006 NaN
5 1.068555 0.669145 0.128079
6 -0.783522 0.167165 -0.426007
7 0.498378 -0.950698 2.342104
In [70]: df.fillna (0)
Out[70]:
A B C
0 -0.271131 0.084627 -1.707637
1 1.895796 0.590270 -0.505681
2 -0.628760 -1.623905 1.143701
3 0.005082 1.316706 -0.792742
4 0.135748 -0.274006 0.000000
5 1.068555 0.669145 0.128079
6 -0.783522 0.167165 -0.426007
7 0.498378 -0.950698 2.342104
```

Descriptive Statistics Functions 统计属性

```
In [76]: columns_numeric = ['lat','lng','pop']
```

```
In [78]: city_data[columns_numeric].mean()
Out[78]:
lat 20.662876
lng 10.711914
pop 265463.071633
dtype: float64

In [79]: city_data[columns_numeric].sum()
Out[79]:
lat 1.512936e+05
lng 7.843263e+04
pop 1.943721e+09
dtype: float64

In [80]: city_data[columns_numeric].count()
Out[80]:
lat 7322
lng 7322
pop 7322
dtype: int64

In [81]: city_data[columns_numeric].median()
Out[81]:
lat 26.792730
lng 18.617509
pop 61322.750000
dtype: float64

In [83]: city_data[columns_numeric].quantile(0.8)
Out[83]:
lat 46.852480
lng 89.900018
pop 269210.000000
dtype: float64

In [85]: city_data[columns_numeric].sum(axis = 1)
Out[85]:
0 3.095116e+03
1 1.509977e+04
2 2.016419e+05
3 4.994400e+04
4 1.009850e+04
```

Describe用来输出数据的所有统计属性。

Pandas also provides us with another very handy function called describe. This function will calculate the most important statistics for numerical data in one go so that we don't have to use individual functions.

```
In [86]: city_data[columns_numeric].describe()
Out[86]:
lat  lng  pop
count 7322.000000 7322.000000 7.322000e+03
mean 20.662876 10.711914 2.654631e+05
```

```
std 29.134818 79.044615 8.287622e+05
min -89.982894 -179.589979 -9.900000e+01
25% -0.324710 -64.788472 1.734425e+04
50% 26.792730 18.617509 6.132275e+04
75% 43.575448 73.103628 2.001726e+05
max 82.483323 179.383304 2.200630e+07
```

Concatenating Dataframes

由于我们需要处理的数据通常有多个，因此我们需要合并这些数据。pandas提供了很多函数来支持数据的合并。

Concatenating Using the concat method

合并两组，每组随机选择3个样本。

```
In [25]: city_data1 = city_data.sample(3)
In [26]: city_data2 = city_data.sample(3)
In [29]: city_data_combine = pd.concat([city_data1,city_data2])
In [30]: city_data_combine
Out[30]:
   city city_ascii lat lng pop \
4255 Groningen Groningen 53.220407 6.580001 198941.0
5171 Tambov Tambov 52.730023 41.430019 296207.5
4204 Karibib Karibib -21.939003 15.852996 6898.0
4800 Focsani Focsani 45.696551 27.186547 92636.5
1183 Pleven Pleven 43.423769 24.613371 110445.5
7005 Indianapolis Indianapolis 39.749988 -86.170048 1104641.5

   country iso2 iso3 province
4255 Netherlands NL NLD Groningen
5171 Russia RU RUS Tambov
4204 Namibia NaN NAM Erongo
4800 Romania RO ROU Vrancea
1183 Bulgaria BG BGR Pleven
7005 United States of America US USA Indiana
Another
```

另外一个场景是，我们对同一个数据集的不同columns进行合并，也就是对标签进行合并。index就是样本标号。

```
In [32]: df1 = pd.DataFrame({'col1': ['col10', 'col11', 'col12', 'col13'],
...: 'col2': ['col20', 'col21', 'col22', 'col23'],
...: 'col3': ['col30', 'col31', 'col32', 'col33'],
...: 'col4': ['col40', 'col41', 'col42', 'col43']},
...: index=[0, 1, 2, 3])

In [33]: df1
Out[33]:
   col1  col2  col3  col4
```

```

0 col10 col20 col30 col40
1 col11 col21 col31 col41
2 col12 col22 col32 col42
3 col13 col23 col33 col43
In [34]: df4 = pd.DataFrame({'col2': ['col22', 'col23', 'col26', 'col27'],
...: 'Col4': ['Col42', 'Col43', 'Col46', 'Col47'],
...: 'col6': ['col62', 'col63', 'col66', 'col67']},
...: index=[2, 3, 6, 7])
In [37]: pd.concat([df1, df4], axis=1)
Out[37]:
col1 col2 col3 col4 Col4 col2 col1
0 col10 col20 col30 col40 NaN NaN NaN
1 col11 col21 col31 col41 NaN NaN NaN
2 col12 col22 col32 col42 Col42 col22 col62
3 col13 col23 col33 col43 Col43 col23 col63
6 NaN NaN NaN Col46 col26 col66
7 NaN NaN NaN NaN Col47 col27 col67

```

Database Style Concatenations Using the merge Command

The most familiar way to concatenate data (for those acquainted with relational databases) is using the join operation provided by the databases. Pandas provides a database friendly set of join operations for dataframes. These operations are optimized for high performance and are often the preferred method for joining disparate dataframes.

Joining by columns: This is the most natural way of joining two dataframes. In this method, we have two dataframes sharing a common column and we can join the two dataframes using that column. The pandas library has a full range of join operations (inner, outer, left, right, etc.) and we will demonstrate the use of inner join in this sub-section. You can easily figure out how to do the rest of join operations by checking out the pandas documentation.

For this example, we will break our original cities data into two different dataframes, one having the city information and the other having the country information. Then, we can join them using one of the shared common columns.

Lasso输出系数

```

def train(self):
    x_train, y_train, x_test, y_test = self.load_data()
    lasso = Lasso(random_state=0)
    alphas = np.logspace(-4, -0.5, 30)
    estimator = GridSearchCV(lasso, dict(alpha=alphas))
    estimator.fit(x_train, y_train)
    lasso.fit(x_train, y_train)//必须先对lasso使用fit, 才可以使用coef_
    print "coef_ = ", lasso.coef_
    return estimator

```


第三章：算法

这一章主要介绍

1. 线性代数与矩阵计算中的算法，比如：LU分解，Householder变换，矩阵求本征值，矩阵求逆，PCA，SVD，Krylov子空间法，Lanczos等。主要是要对对称，非对称矩阵的线程方程求解，以及矩阵本征值，本征矢计算有一个清晰的轮廓，也就是对不同的矩阵类型知道有什么最适合的方法，以及明白各种方法的优缺点。最后，最关键的就是大规模稀疏矩阵的低秩分解，这是推荐系统的基础。
2. 传统的算法与数据结构，比如排序，动态规划，最短路径，插值；二叉树，红黑树，链表，队列，堆栈。
3. 未来可能会把图像与信号处理的算法加在这里，但是不急。

矩阵计算

这一节主要涉及到线性代数与矩阵计算方面的东西：

1. 求解线性方程

- i. 满秩的使用Gauss消元法, 矩阵求逆
- ii. 超定方程使用最小二乘法(等价于求伪逆法)
- iii. 欠定方程的正则化方法

2. 矩阵变换

- i. Gram-Schmidt 变换
- ii. Household变换
- iii. Givens变换

3. 矩阵分解技巧

- i. Schur分解
- ii. LU分解
- iii. QR分解
- iv. Krylov子空间法

4. 求解矩阵本征值的方法

- i. Power方法
- ii. QR分解(对称, $n < 25$ 最实用)
- iii. 分而治之(对称, $n > 25$ 最实用)
- iv. Jacob方法
- v. 瑞利商迭代
- vi. 对分法和分而治之
- vii. Lanczos方法求本征态

5. 奇异矩阵分析

- i. SVD
- ii. PCA
- iii. Condition Number

6. 大规模稀疏矩阵(Large Sparse Matrix)问题

- i. Lanczos
- ii. SVD
- iii. Krylov
- iv. Large Sparse Matrix在推荐系统中的应用

矩阵计算在力学, 物理学, 以及现今的推荐系统中都有着重要的应用, 后者主要处理的是大规模的稀疏矩阵问题。重要的不是理论, 而是怎么把这些理论用来巧妙地解决实际的问题, 因此, 尽可能的寻找这些方法的应用场景。

这一章主要讨论的实际问题如下:

1. 线性方程组求解问题, 基于最小二乘法的正规方程方法(或者说伪逆法)。以及变量数目远大于方程数目的问题, 主要是基于Ridge Regression, Lasso Regression, Regularized Discriminant Analysis, Regularized multinomial logistic regression。
2. 矩阵本征值求解方法, 求解矩阵本征值的方法有幂法(只能求解最大或者最小的本征值), QR分解方法, 通过一系列的QR会得到一个与原矩阵相似的上三角矩阵。常用的实现QR分解的方法有Gram-Schmidt变换, Household变换, Givens变换(Givens变换可以并行计算, 因此每次只修改i,j两行; 且最实用与大规模稀疏矩阵)。为了快速的通过QR分解求解本征值, 我们需要引入Hessenberg矩阵, 通过把矩阵变成上Hessenberg矩阵, 再通过Givens变换进行QR分解, 使得以后每次QR分解的时间复杂度是 $O(n^2)$ 。
 $(A - \sigma I)^{-1}$ 可以用来求解 σ 附近的本征值本征矢。
3. SVD, PCA这些分解有提及, 但是没有给出具体的应用场景。其实SVD可以用于推荐系统, PCA也可以用在机器学习中用于特征提取。
4. Krylov子空间法是很重要的一种求解大规模矩阵(无论稀疏与否)的有效方法, 对于一般矩阵(无论对称与否), 可以使用Arnoldi算法, 化为相似的上Hessenberg矩阵; 而对于对称的矩阵, 我们可以使用更方便的Lanczos方法, 化为三对角矩阵进行求解。此外, 共轭梯度法也是一种Krylov子空间法。

一些结论性的东西

1. 对于维数小于25的三对角矩阵, QR迭代式目前求所有本征值, 本征矢最快最实用的方法。
2. 对于维数大于25的三对角矩阵, 分而治之是目前求所有本征值, 本征矢最快最实用的方法。
3. 非对称矩阵实际使用的直接方法是隐式位移的QR迭代。(求特征值问题实际是多项式求根问题, 采用的是迭代法)
4. 矩阵的条件数或者更一般的矩阵的本征值分布是迭代法求解矩阵本征值或者求解线性方程组收敛的关键性因素。

这一章的终极目的

寻找超大规模稀疏矩阵低秩分解的快速实用方法, 对标的的是当前的SVD算法。推荐系统是其核心的应用场景。

需要做的事情

1. Lanczos方法是求解超大规模稀疏矩阵前几个特征值的有效方法, 能否把他应用到电商推荐系统中, 处理User-Item矩阵问题? 不可以, Lanczos适用于对称矩阵, 非对称矩阵需要用阿诺尔迪算法。
2. 把各种求解本征值问题的时间复杂度列成一张表。
3. 求解线性方程组的方法的时间复杂度列成一张表。
4. 在不理解一些算法的时候, 最好自己去实现她, 在实现的过程中, 就加深理解了。比如“隐式位移的QR迭代”, 分而治之的对称矩阵求解本征值, 本征矢。
5. 弄清楚矩阵本征值, 迭代法求本征值的收敛速度以及最优化收敛的关系。
6. 加一节有关随机矩阵的理论, 前人以及研究过GOE, GUE, GSE分布的随机矩阵, 是否正对当前的电商User-Item矩阵, 用0-1矩阵来模拟, 然后我们从理论上研究这种矩阵的本征值分布?
7. 用Krylov子空间理论, 给予PCA, SVD, FM, CG(共轭梯度法)这些算法一个统一的诠释。

线性方程组求解方法

1. 满秩的使用Gauss消元法, 矩阵求逆
2. 超定方程使用最小二乘法(等价于求伪逆法)
3. 欠定方程的正则化方法

满秩的线性方程组

$$Ax = b$$

对于变量数目N不多的线性方程组, 我可以通过矩阵求逆的方式来求得未知变量x;

$$x = A^{-1}b$$

可以利用矩阵的LU分解来求, 求解的时间复杂度就是 $O(N^3)$.

当N很大的时候, 还是很费时间的, 这个时候可以利用共轭梯度法来求解。具体步骤如下: 见Krylov子空间法那一部分。

超定方程使用最小二乘法

$$Ax = b$$

若此时矩阵A是MxN维的, M>N, 则是超定问题。此时我们可以求他们的最小二乘解。即:

$$L(x) = (Ax - b)^2$$

通过 $L(x)$ 对x求导并令其梯度为0, 可以得到:

$$\frac{\partial L(x)}{\partial x} = A^T Ax - A^T b = 0$$

得到:

$$x = (A^T A)^{-1} A^T b$$

这就是欠定方程的最小二乘解。

求解欠定方程的正则化方法

$$Ax = b$$

若此时矩阵A是MxN维的, M<N, 此时变量个数大于方程组个数, 因此有无穷多个解, 这就是欠定问题。在这无数多个解中, 我们要根据一些对问题系统的先验知识来对问题进行约束。使得解具有鲁棒性或者稀疏性。

使得解具有稀疏性可能是我们首先要考虑的, 因为系统并没有那么多变量, 或者系统仅仅由少数一

些变量决定的。因此我们可以通过L1正则化来求解欠定问题。

$$L(\mathbf{x}) = (\mathbf{Ax} - \mathbf{b})^2 + \lambda \|\mathbf{x}\|_1$$

其中的参数 λ 需要通过交叉验证来确定。

other

利用LU分解，我们可以求满秩的线性方程组的解。对于 $m \times n$ ($m > n$) 阶矩阵，对于超定方程(方程数目大于未知数的个数)，因为一般没有解满足 $Ax = b$ ，这就是最小二乘法发挥作用的地方。

这个问题的解就是使得: $\|Ax - b\|_2^2$ 值极小的解，通过求导(把 $x \rightarrow x + e$, 求梯度等于0的 x)，可以得到解为：

$$x = (A^T A)^{-1} A^T b$$

Gram-Schmidt正交化

Gram-Schmidt正交化的基本想法，是利用投影原理在已有基的基础上构造一个新的正交基。

$$((\beta))_1$$

<http://elsenaju.eu/Calculator/QR-decomposition.htm>

https://rosettacode.org/wiki/QR_decomposition

https://www.wikiwand.com/en/QR_decomposition#/Example_2

https://www.wikiwand.com/en/Householder_transformation

C#有开源的免费的代数库mathnet.numerics,功能也比较多，推荐通过Nuget来安装这个库

矩阵变换方法

1. Gram-Schmidt 变换
2. Household 变换
3. Givens 变换

最好对每个算法，给出实用的代码在这儿。

Household 变换，Givens 变换是进行矩阵QR分解的基础，而这些分解是对矩阵进行本征分解的基础。

对称矩阵进行Household变换或者Givens变换之后，就变成了三对角矩阵，对于三对角矩阵，有高效的求解本征值的方法，因此对于对称矩阵，计算本征值的时间复杂度就是这两个变换的时间复杂度。

对于非对称矩阵，可以通过一系列的QR分解，来把矩阵变成对角矩阵。(问题是，怎么保证通过一系列QR分解，它最终会收敛到对角矩阵)。

LU分解可以用来求解线性方程组，也可以用来对矩阵求逆。

Gram-Schmidt 变换

对矩阵A进行分解，首先内积 $[\mathbf{u}, \mathbf{a}] = \mathbf{a}\mathbf{u}$ ，再定义投影：

$$\text{proj}_{\mathbf{u}} \mathbf{a} = \frac{[\mathbf{u}, \mathbf{a}]}{[\mathbf{u}, \mathbf{u}]} \mathbf{u}$$

然后：

$$\mathbf{u}_1 = \mathbf{a}_1 \quad \mathbf{e}_1 = \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|}$$

$$\mathbf{u}_2 = \mathbf{a}_2 - \text{proj}_{\mathbf{u}_1} \mathbf{a}_2 \quad \mathbf{e}_2 = \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|}$$

$$\mathbf{u}_3 = \mathbf{a}_3 - \text{proj}_{\mathbf{u}_1} \mathbf{a}_3 - \text{proj}_{\mathbf{u}_2} \mathbf{a}_3 \quad \mathbf{e}_3 = \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|}$$

$$\mathbf{u}_k = \mathbf{a}_k - \sum_{j=1}^{k-1} \text{proj}_{\mathbf{u}_j} \mathbf{a}_k \quad \mathbf{e}_k = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}$$

$$\mathbf{u}_k = \mathbf{a}_k - \sum_{j=1}^{k-1} [\mathbf{e}_j, \mathbf{a}_k] \mathbf{e}_j$$

注意到所有的 \mathbf{e}_k 是彼此垂直的，因此：

$$\mathbf{a}_k = \sum_{j=1}^k [\mathbf{e}_j, \mathbf{a}_k] \mathbf{e}_j$$

因此矩阵A可以分解成正交矩阵与上三角矩阵：

$$A = QR$$

其中： $R_{i,j} = [\mathbf{e}_i, \mathbf{a}_j]$, $i \leq j$

$$Q = [\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n]$$

Household 变换

为了实现QR分解, 我们也可以采用Household变换。

假设 \mathbf{e}_1 是矢量 $(1, 0, 0, \dots, 0)^T$, I 是 $m \times m$ 的单位阵。设:

$$\mathbf{u} = \mathbf{x} - \alpha \mathbf{e}_1$$

$$\mathbf{v} = \frac{\mathbf{u}}{\|\mathbf{u}\|}$$

$$Q = I - 2\mathbf{v}\mathbf{v}^T$$

对于复数矩阵:

$$Q = I - 2\mathbf{v}\mathbf{v}^*$$

Q 是 $m \times m$ 的Household矩阵, 并且:

$$Q\mathbf{x} = (\alpha, 0, 0, \dots, 0)^T$$

因此:

$$Q_1 A = \begin{bmatrix} \alpha_1 & * & \cdots & * \\ 0 & & & \\ \cdots & A' & & \\ 0 & & & \end{bmatrix}$$

$$Q_k = \begin{bmatrix} I_{k-1} & 0 \\ 0 & Q'_k \end{bmatrix}$$

t次迭代以后, $t = \min(m-1, n)$:

$$R = Q_t \dots Q_2 Q_1 A$$

是一个上三角矩阵, 又因为:

$$Q = Q_1^T Q_2^T \dots Q_t^T$$

因此 $A = QR$ 是矩阵 A 的一个QR分解。一次Household变换需要 $\frac{2}{3}n^3$ 次乘法操作。

Givens 变换

Givens变换通过每次乘以一个正交矩阵使得一个非对角元变为0来实现QR分解。相对于Household变换的优点是它可以并行, 并且适合于非常稀疏的矩阵。因为每次操作只影响 i, j 两行的元素, 因此可以做到并行计算。

$$G(i, j, \theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix},$$

其中：

$$g_{kk} = 1; \quad k \neq i, j$$

$$g_{kk} = c; \quad k \neq i, j$$

$$g_{ij} = -g_{ji} = -s;$$

通过此Givens变换使得矩阵A的元素 $A_{i,j} = 0$, 其中*i < j*

一个例子：

$$\begin{aligned} G_2 A_2 &= A_3 \\ &\approx \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix} \begin{bmatrix} 7.8102 & 4.4813 & 2.5607 \\ 0 & -2.4327 & 3.0729 \\ 0 & 4 & 3 \end{bmatrix} \end{aligned}$$

在这里：

$$r \approx \sqrt{(-2.4327)^2 + 4^2} \approx 4.6817$$

$$c \approx -2.4327/r \approx -0.5196$$

$$s \approx -4/r \approx -0.8544$$

$$A_3 \approx \begin{bmatrix} 7.8102 & 4.4813 & 2.5607 \\ 0 & 4.6817 & 0.9664 \\ 0 & 0 & -4.1843 \end{bmatrix}$$

矩阵分解方法

1. Schur分解
2. LU分解
3. QR分解
4. 矩阵本征值分解
5. 矩阵奇异值分解
6. Krylov子空间法
7. Cholesky 分解

Schur分解

复 $n \times n$ 的矩阵酉相似于一个上三角阵。

$$A = Q U Q^{-1}$$

其中Q是酉矩阵(Unitary Matrix)。

LU分解

知道存在LU分解，能用来进行线性方程组求解以及矩阵求逆就可以，具体的实现细节可以查书，比如《Matrix Computations》3.2节 The LU Factorization。明白其思想更好。

任何非奇异方阵都可以分解成上三角阵与下三角阵之积

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} * \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

对比两边矩阵的，可以求得：

但是编程时，行列都是从0开始时，要注意转换。

第1行： $a_{1j} = u_{1j}, j = 1, 2, \dots, n.$ => $u_{1j} = a_{1j}$ 。

第1列： $a_{j1} = l_{j1}u_{11}, j = 1, 2, \dots, n.$ => $l_{j1} = a_{j1}/u_{11}$ 。

...

第k行： $a_{kj} = \sum_{i=1}^{i=k} l_{ki}u_{ij}, => u_{kj} = a_{kj} - \sum_{i=1}^{i=k-1} l_{ki}u_{ij}$ 。

第k列： $a_{jk} = \sum_{i=1}^{i=k} l_{ji}u_{ik}, => u_{jk} = [a_{jk} - \sum_{i=1}^{i=k-1} l_{ji}u_{ik}]/u_{kk}$ 。

因为前k-1行的 u_{ij} 都已知, 前k-1列的 l_{ij} 都已知, 因此可以求得第k行 u_{ij} , 第k列的 l_{ij} 。

问题: 得保证 a_{11} 非0, 以及矩阵非奇异。

利用LU分解求线性方程组的解

求解线性方程组 $Ax=b$ 相当于求解 $LUX=b$;

设 $Y = UX$; 因此 $LY = b$; 首先求解 $LY = b$,

$$\begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{bmatrix}$$

求解上面的方程:

第1行: $y_1 = b_1$ 。

对于第k行: $b_k = \sum_{i=1}^{i=k} l_{ki}y_i \Rightarrow y_k = b_k - \sum_{i=1}^{i=k-1} l_{ki}y_i$ 。

求得Y之后, 代入Y=UX求得X:

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix}$$

对于上三角矩阵, 我们从第n行开始求解

对第n行: $y_n = u_{nn}x_n \Rightarrow x_n = y_n/u_{nn}$ 。

...

对第k行: $y_k = \sum_{i=k}^{i=n} u_{ki}x_i \Rightarrow x_k = [y_k - \sum_{i=k+1}^{i=n} u_{ki}x_i]/u_{kk}$

这样通过LU分解矩阵就求得了线性方程组的就X.

矩阵求逆

我们可以利用LU分解来求非奇异方阵的逆矩阵。

$AB=I$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 1 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

可以分解成求:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} * \begin{bmatrix} b_{1k} \\ b_{2k} \\ \cdots \\ b_{nk} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 1 \\ \cdots \\ 0 \end{bmatrix}$$

右边的列, 就只是第k行值非0;

$A * b_k = e_k$, 对所有的 e_k 求出 b_k 就可以得到A的逆矩阵 $A^{-1} = B$

实际求解中把A换成LU来减少计算量。总的计算开销还是 n^3 。但是这样并不比直接的高斯消元法来的快。

算法实现如下。

```
//LU
template<class T>
void CMatrix<T>::LU(CMatrix &mat, int N, CMatrix* L, CMatrix* U)
{
    for (int k = 0; k < N; k++)
    {
        for (int j = k; j < N; j++)
        {
            T U_k_j = mat.Get(k, j);
            T L_j_k = mat.Get(j, k);
            for (int i = 0; i < k; i++)
            {
                U_k_j -= L->Get(k, i)*U->Get(i, j);
            }
            U->Set(k, j, U_k_j);

            for (int i = 0; i < k; i++)
            {
                L_j_k -= L->Get(j, i)*U->Get(i, k);
            }
            L_j_k = L_j_k / U->Get(k, k);
            L->Set(j, k, L_j_k);
        }
    }
}

//solve linear algebra equations
CVector Solve(CMatrix<double>& mat, CVector& vec)
{
    CVector X(vec.Size());
    CVector Y(vec.Size());
    if (mat.Columns() != mat.Rows() && mat.Rows() != vec.Size())
    {
        printf("Dimension not match!");
        return X;
    }
    CMatrix<double> L(vec.Size(), vec.Size());
    CMatrix<double> U(vec.Size(), vec.Size());
    mat.LU(mat, vec.Size(), &L, &U);
    Y = SolveLow(L, vec);
    cout << "Y\n" << Y;
    X = SolveUpper(U, Y);
    cout << "X\n" << X;
    return X;
}

CVector SolveLow(CMatrix<double>& mat, CVector& vec)
```

```

{
    CVector vecRes(vec.Size());
    if (mat.Columns() != mat.Rows() && mat.Rows() != vec.Size())
    {
        printf("Dimension not match!");
        return vecRes;
    }

    for (int k = 0; k < vec.Size(); k++)
    {
        double mRes = 0;
        mRes = vec.Get(k);
        for (int i = 0; i < k; i++)
        {
            mRes -= mat.Get(k, i)*vecRes.Get(i);
        }
        vecRes.Set(k, mRes);
    }
    return vecRes;
}

CVector SolveUpper(CMatrix<double>& mat, CVector& vec)
{
    CVector vecRes(vec.Size());
    if (mat.Columns() != mat.Rows() && mat.Rows() != vec.Size())
    {
        printf("Dimension not match!");
        return vecRes;
    }

    for (int k = vec.Size() -1; k >=0 ; k--)
    {
        double mRes = 0;
        mRes = vec.Get(k);
        for (int i = k+1; i <vec.Size(); i++)
        {
            mRes -= mat.Get(k, i)*vecRes.Get(i);
        }
        mRes = mRes / mat.Get(k, k);
        vecRes.Set(k, mRes);
    }
    return vecRes;
}

//inverse matrix
template<class T>
CMatrix<T> CMatrix<T>::Inv()
{
    CMatrix result = *this;
    CMatrix<double> L(mRows, mColumns);
    CMatrix<double> U(mRows, mColumns);
    CMatrix<double> InvMat(mRows, mColumns);
    LU(result,mRows, &L, &U);
    for (int col = 0; col < mColumns; col++)
    {
}

```

```

CVector vec(mRows, col);
CVector mSolution(mRows);
mSolution = SolveLow(L, vec);
mSolution = SolveUpper(U, mSolution);
for (int row = 0; row<mRows; row++)
{
    InvMat.Set(row, col, mSolution.Get(row));
}
return InvMat;
}

```

QR分解

定理: 设A是 $m \times n$ 阶矩阵, $m \geq n$, 假设A为满秩的, 则存在一个唯一的正交矩阵 Q ($Q^T Q = I$) 和唯一

的具有正对角元 $r_{ij} > 0$ 的 $n \times n$ 阶上三角阵 R 使得 $A = QR$.

QR分解算法:

$$A_0 = A = Q_1 R_1$$

然后:

$$A_1 = R_1 Q_1$$

一般形式:

$$A_k = R_k Q_k$$

$$A_k = Q_{k+1} R_{k+1}$$

因此有:

$$A_k = Q_{k+1} R_{k+1} = Q_{k+1} A_{k+1} Q_{k+1}^T$$

因此:

$$A = Q_1 R_1 = Q_1 (A_1 Q_1^T) = Q_1 A_1 Q_1^T$$

$$= (Q_1 Q_2 \dots Q_k) A_k (Q_1 Q_2 \dots Q_k)^T$$

当K区域无穷时, A_k 就是对称矩阵, $(Q_1 Q_2 \dots Q_k)$ 就是本征矢量。

算法的收敛性: 假设:

$$|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$$

则矩阵 A_k 的矩阵元 $a_{ij}^{(k)}$ 将以如下方式收敛到0:

$$a_{ij}^{(k)} = O(|\lambda_i / \lambda_j|^k) \text{ for all } i > j.$$

python QR分解实例

发现随着QR迭代次数的增加，与原矩阵A相似的矩阵 A^k 越来越收敛于上三角矩阵，也就是对角元以下的元素越来越趋于0，但是对角元上面的元素并不趋于0。虽然 A^k 并没有趋于一个对角阵，但是 $(Q_1Q_2\dots Q_k)$ 就是本征矢。这给你我们QR分解求解矩阵本征值一个直观的图像。

```

A:
[[2, 3], [1, 2]]
Eig of A:
[3.73205081 0.26794919]
EigVector of A:
[[ 0.8660254 -0.8660254]
 [ 0.5         0.5        ]]
EigVector : 0
[[-0.89442719 -0.4472136 ]
 [-0.4472136   0.89442719]]
A = RQ: 0
[[ 3.6 -2.2]
 [-0.2  0.4]]
EigVector : 1
[[ 0.86824314 -0.49613894]
 [ 0.49613894  0.86824314]]
A = RQ: 1
[[3.72307692 2.01538462]
 [0.01538462 0.27692308]]
EigVector : 2
[[-0.86618559 -0.49972245]
 [-0.49972245  0.86618559]]
A = RQ: 2
[[ 3.73140954e+00 -2.00110988e+00]
 [-1.10987791e-03  2.68590455e-01]]

```

缺点：

每步QR分解的时间复杂度是 $O(n^3)$ ，此外当有简并的本征值时，收敛会非常慢。

更有效的QR分解：Hessenberg矩阵

基于直接的QR分解并不是高效的方法，如果通过Household变换或者Givens变换把待分解的矩阵转换成上Hessenberg矩阵，此操作的时间复杂度是 $O(n^3)$ ，此时再利用Givens变换来进行QR分解，则QR分解的复杂度从 $O(n^3)$ 变成了 $O(n^2)$ 。

问题是为啥不直接一步用Household变换实现QR？反正时间复杂度都是 $O(n^3)$ ？

原因是,当我们把矩阵变成Hessenberg矩阵后,以后就不再需要进行 $O(n^3)$ 的操作,因为上三角乘以上Hessenberg矩阵还是上Hessenberg矩阵,也就是所有的 A^k 都是上Hessenberg矩阵。因此,以后的QR分解都在Hessenberg上进行,每次的复杂度都是 $O(n^2)$.

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \xrightarrow{\text{Phase 1}} \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \xrightarrow{\text{Phase 2}} \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ & & & & \times \end{bmatrix}$$

Phase 1. (Section 2.2.1). In the first phase we will compute a Hessenberg matrix H (and orthogonal U) such that

$$A = UHU^*.$$

Unlike the Schur factorization ($A = UTU^*$ where T is upper triangular) such a reduction can be done with a finite number of operations.

Phase 2. (Section 2.2.2). In the second phase we will apply the basic QR-method to the matrix H . It turns out that several improvements can be done when applying the basic QR-method to a Hessenberg matrix such that the complexity of one step is $\mathcal{O}(n^2)$, instead of $\mathcal{O}(n^3)$ in the basic version.

QR分解方式的比较

QR分解的实现方式有三种,第一种通过Gram-Schmidt正交化方法,第二种是Household变换方法,第三种是Givens变换方法。

QR分解方法	时间复杂度	优缺点
Gram-Schmidt正交化方法		
Household变换		
Givens变换		可以并行。适合于稀疏矩阵
Hessenberg+Givens	一次Household进行QR分解的时间	时间复杂度最小

矩阵本征值分解

要证明的是，可以通过矩阵QR分解来实现矩阵本征值分解。

实际上，我们在求解矩阵的本征值的时候，要分矩阵是对称矩阵还是非对称矩阵，或者说是厄密矩阵还是非厄密矩阵。算法的收敛性：假设：

$$|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$$

则矩阵 A_k 的矩阵元 $a_{ij}^{(k)}$ 将以如下方式收敛到0：

$$a_{ij}^{(k)} = O(|\lambda_i/\lambda_j|^k) \text{ for all } i>j. \text{ 也就是对角元以下的矩阵元会以这样的方式收敛到0。}$$

Krylov子空间法

为了求解如下线性方程组问题：

$$\mathbf{Ax} = \mathbf{b}$$

身处大数据时代，为了更高效的处理数据，从大数据中提取我们想要的知识，我们就不得不提到Krylov子空间法，这个方法的强大之处在于我们不需要在超大的原始空间求解问题，而是在一个远小于原始空间的子空间去求解问题。适合对称矩阵的共轭梯度法，Lanczos方法，以及适合非对称矩阵的Arnoldi算法就是Krylov子空间法的实例。要明白的就是，要求一个矩阵本征值，我们还是需要使用正交变换，就是寻找一个正交矩阵Q对矩阵A进行变换，使得A变成一个上Hessenberg矩阵，再如果这个上Hessenberg矩阵有一个下次对角元 $h_{k,k-1} = 0$ ，我们就可以对矩阵进行分解，只需要求 $H_{k \times k}$ 即可得到原始矩阵A的本征值。

对一个 $n \times n$ 的矩阵，最便宜的操作莫过于让他乘以一个向量。即：我们可以得到如下序列：

$$\mathbf{y}_1 = \mathbf{b}$$

$$\mathbf{y}_2 = \mathbf{Ay}_1$$

$$\mathbf{y}_3 = \mathbf{Ay}_2 = \mathbf{A}^2\mathbf{y}_1$$

$$\mathbf{y}_n = \mathbf{Ay}_{n-1} = \mathbf{A}^{n-1}\mathbf{y}_1$$

定义： $K = (\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n)$

$$AK = (\mathbf{y}_2, \mathbf{y}_3, \dots, \mathbf{Ay}_n) = (\mathbf{y}_2, \dots, \mathbf{y}_n, \mathbf{A}^n\mathbf{y}_1)$$

假定： $\mathbf{AK} = \mathbf{KC}$

我们可以得到C的形式如下：

$$K^{-1}AK = C = \begin{bmatrix} 0 & 0 & \cdots & -c_1 \\ 1 & 0 & \cdots & -c_2 \\ 0 & 1 & \cdots & -c_3 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 1 & -c_{n-1} \\ 0 & 0 & \cdots & -c_n \end{bmatrix}$$

C是一个上三角矩阵。

一个问题是，即使A是稀疏的，K也可能是稠密的；另外一个问题，由于执行的是矩阵的幂操作，因此 y_i 最终收敛到A的最大本征值对应的本征矢量方向。因此我们需要一些正交化技巧，也就是对厄密矩阵，有Lanczos方法，对一般矩阵有Arnoldi方法。

实际上，我们用一个正交矩阵Q来近似K，对K做QR分解， $K=QR$ 。因此有：

$$K^{-1}AK = (R^{-1}Q^T)AQR = C$$

$$Q^T A Q = K^{-1} A K = R C R^{-1} = H$$

因为R是上三角矩阵，C是上Hessenberg矩阵，因此H是上Hessenberg矩阵。又因为：AQ=QH。

$$Aq_j = \sum_{i=1}^{j+1} h_{i,j} q_i$$

因为 q_i 是正交的，因此我们可以在上式同时乘以 q_m^T ，然后得到：

$$q_m^T A q_j = \sum_{i=1}^{j+1} h_{i,j} q_m^T q_i = h_{m,j} \text{ for } 1 \leq m \leq j$$

因此：

$$h_{j+1,j} q_{j+1} = Aq_j - \sum_{i=1}^j h_{i,j} q_i$$

总的来说，我们想对矩阵A做正交变换，因为正交变换不改变矩阵的本征值。因此，我们需要求这个正交变换矩阵Q。下面的算法告诉我们怎么求这个正交矩阵Q。附带我们把正交变换后得到的矩阵H也求出来了，H是一个上Hessenberg矩阵。因此如果我们发现H有一个下次对角元为 $h_{k,k-1}$ ，则我们就可以停止迭代，因为此时我们可以对矩阵H进行分解，因为我们得到的矩阵 $H_{1:k,1:k}$ 的矩阵本征值就是原始矩阵的H的本征值。

因此有如下的：

Arnoldi算法：

$$q_1 = b / \|b\|_2$$

/k is the number of columns of Q and H to computer/

for j=1 to k:

$$z = Aq_j$$

for i = 1 to j:

$$h_{i,j} = q_i^T z$$

$$z = z - h_{i,j} q_i$$

end for

$$h_{j+1,j} = \|z\|_2$$

if $h_{j+1,j} = 0$, quit

$$q_{j+1} = z / h_{j+1,j}$$

end for

q_j 成为Arnoldi矢量。时间复杂度是 $O(k^2n)$ ，k是子空间大小。在这里，每一步迭代用到 $z = Aq_j$ ，这是为了对Krylov子空间进行正交化的方式。Q是原始Krylov子空间 $K_k(A, b) = (b, Ab, A^2b, \dots, A^{k-1}b)$ 正交化之后的空间。

对称情况的Krylov

如果A是对称的，则H是上Hessenberg矩阵，也是对称矩阵，因此是三对角矩阵。假设T=H。

$$T = \begin{bmatrix} \alpha_1 & \beta_1 & \cdots & 0 \\ \beta_1 & \alpha_2 & \cdots & 0 \\ 0 & \beta_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & \alpha_n \end{bmatrix}$$

AQ=QT，计算第j列。得到：

$$Aq_j = \beta_{j-1}q_{j-1} + \alpha_j q_j + \beta_j q_{j+1}$$

因为 q_j 彼此A共轭，因此上式两边同乘 q_j 得到： $q_j A q_j = \alpha_j$ 。

对于对称情况，我们有如下的Lanczos算法。

Lanczos算法

$$q_1 = b / \|b\|_2, \beta_0 = 0, q_0 = 0$$

/k is the number of columns of Q and H to computer/

for j=1 to k:

$$z = Aq_j$$

$$\alpha_j = q_j^T z$$

$$z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$$

$$\beta_j = \|z\|_2$$

if $\beta_j = 0$, quit

$$q_{j+1} = z / \beta_j$$

end for

q_j 称为Lanczos矢量。

Krylov子空间

定义：Krylov子空间 $K_k(A, b)$ 为 $(b, Ab, A^2b, \dots, A^{k-1}b)$

Krylov方法求解线性方程组

我们讨论Krylov方法的目的是通过k步就可以求解线性方程组 $AX=b$ 。

共轭梯度法

定理：A是对称矩阵， $T_k = Q_k^T A Q_k$ ，并且 $r_k = b - Ax_k$, $x_k \in K_k$, 如果 T_k 是非奇异的，并且

$x_k = Q_k T_k^{-1} e_1 \|b\|_2$, 这里 $e_1^{k \times 1} = [1, 0, \dots, 0]^T$, 因此 $Q_k^T r_k = 0$ 。

如果A也是正定的，则 T_k 一定不是奇异的，并且 x_k 也最小化 $\|r_k\|_{A^{-1}}$ ，在所有的 $x_k \in K_k$ 。此外

$$r_k = \|r_k\|_2 q_{k+1}.$$

CG迭代只需要保留三个向量 $x_k, r_k = b - Ax_k$, 以及共轭梯度 p_k 。通过如下方式迭代

$$x_k = x_{k+1} + v p_k.$$

共轭梯度算法

对任意初始点 $x_0 \in E^n$, 定义 $d_0 = -g_0 = b - Qx_0$

$$x_{k+1} = x_k + \alpha_k d_k$$

$$\alpha_k = -\frac{g_k^T d_k}{d_k^T Q d_k}$$

$$d_{k+1} = -g_{k+1} + \beta_k d_k$$

$$\beta_k = -\frac{g_{k+1}^T Q d_k}{d_{k+1}^T Q d_k} \text{ 其中 } g_k = Qx_k - b$$

¹ A first course in linear algebra

矩阵本征值求解方法

1. Power方法
2. QR分解(对称, n<25最实用)
3. 分而治之(对称, n>25最实用)
4. Jacob方法
5. 瑞利商迭代
6. 对分法和逆迭代
7. Lanczos方法求本征态

由于我们在前面已经做好了计算本征值方法的铺垫, 也就是介绍了Household变换, Givens变换, QR分解等。因此我们直接可以利用这些工具来讨论矩阵的本征值, 本征矢量求解。

Power方法

指数方法对对称与非对称矩阵都实用。指数方法一般只能用来求解最大的本征值, 利用的是矩阵的高次幂, 作用在一个向量上面, 只有最大本征值对应的本征向量能被保留下来。因此一个问题是,

初始向量需要含有最大本征矢量。算法如下: 给定 x_0 , 进行如下迭代

```
i=0,  
repeat
```

```
     $y_{i+1} = Ax_i$   
     $x_{i+1} = y_{i+1} / \|y_{i+1}\|_2$   
     $\lambda_{i+1} = x_{i+1}^T Ax_{i+1}$   
    i=i+1  
until收敛
```

假设A的本征值为 $\lambda_1, \lambda_2, \dots, \lambda_n$, 满足

$$|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|$$

假设初始矢量可以用矩阵的本征矢 $\alpha_1, \alpha_2, \dots, \alpha_n$ 展开,

$$\mathbf{x}_0 = c_1 \alpha_1 + c_2 \alpha_2 + \dots + c_n \alpha_n$$

则:

$$A^k \mathbf{x}_0 = \sum_{i=1}^n c_i \lambda_i^k \alpha_i = \lambda_1^k \sum_{i=1}^n c_i \left(\frac{\lambda_i}{\lambda_1}\right)^k \alpha_i$$

因此当k很大的时候, $(\frac{\lambda_i}{\lambda_1})^k$ 趋于0, 因此只会保留本征值最大的本征矢量。收敛速度取决于 $\frac{\lambda_2}{\lambda_1}$ 。

如果使得 $c_1 = 0$, 则我们可以求第二大本征值与本征向量, 因此我们可以依次求解第一大, 第二大, ..., 本征值, 但是由于一些舍入误差, 我们不能保证在求第N大本征值时, 初始向量中没有混入前N-1大本征向量, 因为存在机器精度, 因此该方法一般用来求最大本征值。

逆迭代法(逆指数法)

该方法是指数方法的改进, 可以用来求矩阵A在任意点 σ 附近的本征值。

给定 x_0 , 进行如下迭代

i=0,

repeat

$$y_{i+1} = (A - \sigma I)^{-1}x_i$$

$x_{i+1} = y_{i+1}/\|y_{i+1}\|_2$, 近似本征矢量。

$$\lambda_{i+1} = x_{i+1}^T A x_{i+1}, \text{ 近似本征值。}$$

i=i+1

until收敛

QR分解

需要 $O(\frac{4}{3}n^3)$ 用来把对称矩阵化为三对角矩阵, 化为三对角矩阵之后, 可以通过代数方法求本征值, 每个本征值的时间复杂度就是 $O(n)$, 因此求所有本征值的时间复杂度是 $O(n^2)$, QR求对称矩阵的本征值的时间复杂度是 $O(\frac{4}{3}n^3)$; 求所有本征矢量的时间复杂度是 $O(6n^3)$ 。有必要讨论一下实用的QR分解方法, 也就是先转化成上Hessenberg矩阵, 再用N次Givens变换得到QR分解。以后每次QR分解的时间复杂度是 $O(n^2)$ 。

分而治之

Jacobi

Jacobi的思想是每次通过一个相似变换消除掉一个非对角矩阵元, 且能满足非对角元的平方和的减少量就是这个被消除量的平方。它可以用来求对称矩阵以及非对称矩阵。

$$\begin{aligned} A_m &= J_{m-1}^T A_{m-1} J_{m-1} \\ &= J_{m-1}^T J_{m-2}^T \dots J_0^T A_{m-1} J_0 \dots J_{m-2} J_{m-1} \\ &= J^T A J \end{aligned}$$

对于大的m, 最终A收敛到对称矩阵。

每一次迭代m, 就是调用一次Jacob旋转来消除矩阵非对角元素 A_{ij} , 每一次Jacobi旋转的时间复杂度是O(n), 它只改变i,j行与i, j列的矩阵元素。

Jacobi-Rotation

Jacobi-Rotation (A,j,k)

if $|a_{jk}|$ is not too small

$$\begin{aligned}\tau &= (a_{jj} - a_{kk}) / (2a_{jk}) \\ t &= \text{sign}(\tau)(|\tau| + \sqrt{1 + \tau^2}) \\ c &= 1 / \sqrt{1 + t^2} \\ s &= c \cdot t\end{aligned}$$

$A = R^T(j, k, \theta)AR(j, k, \theta)$...where, $c = \cos(\theta), s = \sin(\theta)$

if eigenvectors are desired

$J = JR(j, k, \theta)$

end if

end if

Jacobi算法

repeat:

choose a j,k pair

call Jacobi-Rotation (A,j,k) until A is sufficiently diagonal

收敛性保障

$$off(A) = \sqrt{\sum_{1 \leq i < k \leq n} a_{jk}^2}$$

对j,k实行Jacobi-Rotation后有: $off(A') = off(A) - a_{jk}^2$ 。

因此, 最终非对角元会趋于0。如果每次挑选最大的非对角元进行消除, 这样会加快收敛, 但是寻找最大矩阵元的时间复杂度是 $O(n^2)$, 而一次Jacobi-Rotation计算的时间复杂度是 $O(n)$, 因此所有使劲按花在寻找最大矩阵元上面了。因此一般按行便利, 一般需要对所有非对角元重复5-10次Jacobi-Rotation 操作才能使得矩阵收敛到近似对角矩阵。因此时间复杂度比QR高。

瑞利商迭代

收敛速度与QR一样是cubically(三次收敛), 该方法可以用来求任意点附近的本征值。

算法如下:

输入: x_0 , with $\|x_0\|_2 = 1$, 一个容许误差tol.

定义: $\rho_0 = \rho(x, A) = \frac{x_0^T Ax_0}{x_0^T x_0}$

i = 0.

repeat:

$$y_i = (A - \rho_i I)^{-1} x_{i-1}$$

$$x_i = y_i / \|y_i\|_2$$

$$\rho_i = \rho(x_i, A)$$

i = i+1

until $\|Ax_i - \rho_i x_i\|_2 < tol$

瑞利商迭代的三次收敛定理¹

瑞利商迭代在局部是三次收敛的。也就是说。正确的浮点数是上一步的三倍，当误差足够小的时候。

分而治之

当前求n>25矩阵的本征值与本征矢量最快的方法，最坏的开销是 $O(n^3)$ ，平均开销是 $O(n^{2.3})$ ，对某些本征值分布，时间复杂度是 $O(n^2)$ 。详细的步骤还请参考下面的文献。

对分法和逆迭代

对三对角矩阵，如果求区间[a,b]中的k个本征值，时间复杂度是 $O(kn)$ ，n是矩阵的维度。当 $k \ll n$ 时，它比QR快，因为QR时间复杂度是 $O(n^2)$ 。在最坏的情况下，也就是所有本征值聚集在一起的时候，她的时间复杂度是 $O(nk^2)$ ，而且不能保证本征矢量计算的准确性。

¹. 《实用数值线性代数》[←](#)

奇异矩阵分析

1. SVD
2. PCA
3. Condition Number

SVD

对于任何一个DxM的矩阵X可以做如下分解：

$$X = UDV^T$$

其中 $U^T U = I_D$, $VTV = I_N$, D是对角矩阵。

$$XX^T = UDV^T VDU^T = UD^2U^T$$

$$X^T X = VDU^T UDV^T = VD^2V^T$$

是矩阵的本征值分解。因此可以通过上面两式来求解U, V。主要用到的是矩阵本征值，本征矢求解技巧。

PCA

我们可以通过SVD来实现PCA,具体就是只取X的SVD中的前M个其一分量。

$$X = UDV^T \approx U_M D_M V_M^T$$

其中 U_M, D_M, V_M 对于矩阵X的前M个奇异值分量。

SVD的作用

SVD 将矩阵分解成累加求和的形式，其中每一项的系数即是原矩阵的奇异值。这些奇异值，按之前的几何解释，实际上就是空间超椭球各短轴的长度。现在想象二维平面中一个非常扁的椭圆(离心率非常高)，它的长轴远远长于短轴，以至于整个椭圆看起来和一条线段没有什么区别。这时候，如果将椭圆的短轴强行置为零，从直观上看，椭圆退化为线段的过程并不突兀。回到 SVD 分解当中，较大的奇异值反映了矩阵本身的主要特征和信息；较小的奇异值则如例中椭圆非常短的短轴，几乎没有体现矩阵的特征和携带的信息。因此，若我们将 SVD 分解中较小的奇异值强行置为零，则相当于丢弃了矩阵中不重要的一部分信息。

因此，SVD 分解至少有两方面作用：

- 分析了解原矩阵的主要特征和携带的信息(取若干最大的奇异值)，这引出了主成分分析

(PCA) ;

- 丢弃忽略原矩阵的次要特征和携带的次要信息(丢弃若干较小的奇异值), 这引出了信息有损压缩、矩阵低秩近似等话题。

Condition Number

条件数的定义

Let's linear equations:

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

Let us investigate first, how a small change in the \mathbf{b} vector changes the solution vector. \mathbf{x} is the solution of the original system and let $\mathbf{x} + \Delta\mathbf{x}$ is the solution when \mathbf{b} changes from \mathbf{b} to $\mathbf{b} + \Delta\mathbf{b}$

Then we can write:

$$\mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} + \Delta\mathbf{b}$$

or $\mathbf{A}\mathbf{x} + \mathbf{A}\Delta\mathbf{x} = \mathbf{b} + \Delta\mathbf{b}$

But because $\mathbf{A}\mathbf{x} = \mathbf{b}$, it follows that

$$\mathbf{A}\Delta\mathbf{x} = \Delta\mathbf{b}$$

So:

$$\delta\mathbf{x} = \mathbf{A}^{-1}\Delta\mathbf{b}$$

Using the matrix norm properties:

$$||\mathbf{A}\mathbf{x}|| \leq ||\mathbf{A}|| \cdot ||\mathbf{x}||$$

We can get:

$$||\mathbf{A}^{-1}\Delta\mathbf{b}|| \leq ||\mathbf{A}^{-1}|| \cdot ||\Delta\mathbf{b}||.$$

Also, we can get:

$$||\mathbf{b}|| = ||\mathbf{A}\mathbf{x}|| \leq ||\mathbf{A}|| \cdot ||\mathbf{x}||$$

Using equations 2.5 and 2.6 we can get:

$$\frac{||\Delta\mathbf{x}||}{||\mathbf{A}|| \cdot ||\mathbf{x}||} \leq \frac{||\mathbf{A}^{-1}|| \cdot ||\Delta\mathbf{b}||}{||\mathbf{b}||}$$

Let's define condition number as: $\mathbf{K}(\mathbf{A}) = ||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}||$

we can rewrite equation 2.7 as:

$$\frac{||\Delta\mathbf{x}||}{||\mathbf{x}||} \leq \mathbf{K}(\mathbf{A}) \cdot \frac{||\Delta\mathbf{b}||}{||\mathbf{b}||}$$

Now, let us investigate what happens if a small change is made in the coefficient matrix

A. Consider \mathbf{A} is changed to $\mathbf{A} + \Delta\mathbf{A}$ and the solution changes from \mathbf{x} to $\mathbf{x} + \Delta\mathbf{x}$

$$(\mathbf{A} + \Delta\mathbf{A})(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b},$$

we can obtain:

$$\frac{||\Delta\mathbf{x}||}{||\mathbf{x} + \Delta\mathbf{x}||} \leq \mathbf{K}(\mathbf{A}) \cdot \frac{||\Delta\mathbf{A}||}{||\mathbf{A}||}$$

$K(A)$ is a measure of the relative sensitivity of the solution to changes in the right-hand side vector b . When the condition number $K(A)$ becomes large, the system is regarded as being illconditioned.

Matrices with condition numbers near 1 are said to be well-conditioned.

条件数对梯度下降法收敛的影响

例子

线性方程组解的稳定性分析

系统解的稳定性定义成系统小的扰动对系统解的影响，。

Example 1:

$$\begin{bmatrix} 400 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

Solution:

$$x_1 = -100, x_2 = -200$$

If we give small change to matrix A, change A_{11} from 400 to 401 then:

$$\begin{bmatrix} 401 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

This time the solution is:

$$x_1 = 40000, x_2 = 79800$$

Ill-condition:

When the solution is highly sensitive to the values of the coefficient matrix A or the righthand side constant vector b , the equations are called to be ill-conditioned.

SVD在推荐系统中的应用

假设我们现在有评分矩阵 $V \in \mathbb{R}^{n \times m}$, SVD实际上就是去找到两个矩阵: $U \in \mathbb{R}^{f \times n}$, $M \in \mathbb{R}^{f \times m}$, 其中矩阵U表示 User 和 feature 之间的联系, 矩阵V表示 Item 和 feature 之间的联系。

大家这时候肯定会想, 我们的评分矩阵里面一般会有很多缺失值, 那要怎么去得到 U 和 M 呢? 实际上, 这两个矩阵是通过学习的方式得到的, 而不是直接做矩阵分解。我们定义如下的损失函数:

$$E = \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^m I_{ij} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \sum_{i=1}^n \|U_i\|^2 + \frac{k_m}{2} \sum_{j=1}^m \|M_j\|^2$$

其中 $p(U_i, M_j)$ 表示我们对用户 i 对 物品 j 的评分预测:

$$p(U_i, M_j) = U_i^T M_j$$

梯度下降：

$$-\frac{\partial E}{\partial U_i} = \sum_{j=1}^M I_{ij}((V_{ij} - p(U_i, M_j))M_j) - k_u U_i$$

$$-\frac{\partial E}{\partial M_j} = \sum_{i=1}^n I_{ij}((V_{ij} - p(U_i, M_j))U_i) - k_m M_j$$

实际上，用户的评分与用户的习惯，以及该物品的总体评分水平有关，因此可以加上用户的均值与物品的均值。

$$\hat{r}_{ui} = \mu + b_i + b_u + \mathbf{q}_i^T \mathbf{p}_u$$

μ : 训练集中所有记录的评分的全局平均数。在不同网站中，因为网站定位和销售的物品不同，网站的整体评分分布也会显示出一些差异。比如有些网站中的用户就是喜欢打高分，而另一些网站的用户就是喜欢打低分。而全局平均数可以表示网站本身对用户评分的影响。

b_u : 用户偏置(user bias)项。这一项表示了用户的评分习惯中和物品没有关系的那种因素。比如有些用户就是比较苛刻，对什么东西要求都很高，那么他的评分就会偏低，而有些用户比较宽容，对什么东西都觉得不错，那么他的评分就会偏高。

b_i : 物品偏置(item bias)项。这一项表示了物品接受的评分中和用户没有什么关系的因素。比如有些物品本身质量就很高，因此获得的评分相对都比较高，而有些物品本身质量很差，因此获得的评分相对都会比较低。

这个时候我们的损失函数变为：

$$E = \sum_{(u,i) \in k} (r_{ui} - \mu - b_i - b_u - \mathbf{q}_i^T \mathbf{p}_u) + \lambda(\|\mathbf{p}_u\|_2 + \|\mathbf{q}_i\|_2 + b_u^2 + b_i^2)$$

[1 SVD在推荐系统中的应用](#)

大规模稀疏矩阵问题

1. Lanczos
2. SVD
3. Krylov
4. The Conjugate Gradient Method
5. Large Sparse Matrix在推荐系统中的应用

关于矩阵分解(比如低秩分解, sparse Low Rank)的介绍在推荐系统那一章。

Preconditioning

思考线性方程系统: $\mathbf{Ax} = \mathbf{b}$

假设 $M = M_1 M_2$ 是非奇异的, 思考线性系统 $\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$, 其中:

$$\hat{\mathbf{A}} = M_1^{-1} A M_2^{-1}$$

$$\hat{\mathbf{b}} = M_1^{-1} \mathbf{b}$$

如果 M 与 A 很接近, 因此 $\hat{\mathbf{A}}$ 接近于单位矩阵 I 。因此可以通过 $M_2 \mathbf{x} = \hat{\mathbf{x}}$ 。矩阵 M 被称为 Preconditioner, 为了作为我们感兴趣的问题的求解框架, 它需要满足两个条件:

1. M 必须抓住 A 的本质, 即 $M \approx A$, 因此 $I \approx M_1^{-1} A M_2^{-1} = \hat{\mathbf{A}}$
2. 涉及到 M_1, M_2 矩阵的线性系统必须容易求解, 因为 Krylov 过程涉及到 $M_1^{-1} A M_2^{-1}$ 操作。一个好的 preconditioner 意味着更少的迭代。

如果 A 是对称正定的, 且 $A \approx I + \Delta A$, 并且 $\text{rank}(\Delta A) = k_*$ 远小于 n .

矩阵微分

参考闲话矩阵求导

向量 \mathbf{y} 对标量 x 求导

我们假定所有的向量都是列向量 $\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} & \frac{\partial y_2}{\partial x} & \cdots & \frac{\partial y_m}{\partial x} \end{bmatrix}$

标量 y 对向量 \mathbf{x} 求导:

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_m} \end{bmatrix}$$

向量对向量求导

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

标量对矩阵求导,

$$\frac{\partial y}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{12}} & \cdots & \frac{\partial y}{\partial x_{1q}} \\ \frac{\partial y}{\partial x_{21}} & \frac{\partial y}{\partial x_{22}} & \cdots & \frac{\partial y}{\partial x_{2q}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial x_{p1}} & \frac{\partial y}{\partial x_{p2}} & \cdots & \frac{\partial y}{\partial x_{pq}} \end{bmatrix}$$

矩阵对标量求导,

注意有个类似于转置的操作, 因为 \mathbf{Y} 是 $m \times n$ 矩阵

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \frac{\partial y_{12}}{\partial x} & \cdots & \frac{\partial y_{m1}}{\partial x} \\ \frac{\partial y_{12}}{\partial x} & \frac{\partial y_{22}}{\partial x} & \cdots & \frac{\partial y_{m2}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{1n}}{\partial x} & \frac{\partial y_{2n}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}$$

用维度分析来解决求导的形式问题

向量对向量的微分

$$\frac{\partial(\mathbf{Ax})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial(\mathbf{Ax})_1}{\partial x_1} & \frac{\partial(\mathbf{Ax})_2}{\partial x_1} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_1} \\ \frac{\partial(\mathbf{Ax})_1}{\partial x_2} & \frac{\partial(\mathbf{Ax})_2}{\partial x_2} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial(\mathbf{Ax})_1}{\partial x_n} & \frac{\partial(\mathbf{Ax})_2}{\partial x_n} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix} = \mathbf{A}^T$$

考虑 $\frac{\partial \mathbf{Au}}{\partial \mathbf{x}}$, \mathbf{A} 与 \mathbf{x} 无关, 所以 \mathbf{A} 肯定可以提出来, 只是其形式不知。假如

$\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{u} \in \mathbb{R}^{n \times 1}$, $\mathbf{x} \in \mathbb{R}^{p \times 1}$ 我们知道最终结果肯定和 $\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$ 有关, 注意到 $\frac{\partial \mathbf{u}}{\partial \mathbf{x}} \in \mathbb{R}^{p \times n}$, 于是 \mathbf{A} 只能转置以后添在后面, 因此:

$$\frac{\partial \mathbf{Au}}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \mathbf{A}^T$$

$$dL = \frac{\partial L^T}{\partial \mathbf{w}} d\mathbf{w}$$

再考虑如下问题:

$$\frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}}, \mathbf{x} \in \mathbb{R}^{m \times 1}, \mathbf{y} \in \mathbb{R}^{n \times 1}$$

其中 \mathbf{A} 与 \mathbf{x} 无关, 我们知道 $\frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times 1}$

因此

$$\frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}} = f(A, y) + g(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}, \mathbf{x}^T A) \quad \mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{y} \in \mathbb{R}^{n \times 1}, \mathbf{x}^T \mathbf{A} \in \mathbb{R}^{1 \times n}, \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n}$$

因此, 为了满足 $\frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times 1}$, 我们可以知道函数f,g的形式。最终有:

$$\frac{\partial(\mathbf{x}^T \mathbf{A})\mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \mathbf{A}^T \mathbf{x} + \mathbf{A}\mathbf{y}$$

当 $\mathbf{x} = \mathbf{y}$ 时,

$$\frac{\partial(\mathbf{x}^T \mathbf{A})\mathbf{y}}{\partial \mathbf{x}} = (\mathbf{A}^T + \mathbf{A})\mathbf{x}$$

最后一个例子(还没解出来):

$$\frac{\partial \mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}}{\partial \mathbf{x}}, \mathbf{a}, \mathbf{b}, \mathbf{x} \in \mathbb{R}^{m \times 1} \text{ 知道 } \frac{\partial \mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times 1}$$

还需要对迹形式进行求解.

算法与数据结构

主要介绍数据结构，堆，栈，树；传统的一些算法，比如排序，最短路径。

1. Dijkstra算法
2. 最小生成树Prim算法
3. 霍夫曼编码
4. 排序算法

二叉树

主要介绍二叉搜索树, AVL树和红黑树, 以及介绍怎么通过旋转来为AVL树, 红黑树删除和插入元素。

二叉搜索树

二叉搜索树(Binary Search Tree, BST)主要用于查找, 其好处则是把最差情况下的平均搜索时间降为 $O(\log n)$ 。它满足如下特征:

左子树中的元素都小于等于根节点的数据, 而右子树中的元素都大于等于根节点数据。

二叉搜索树的基本操作有搜索, 插入, 删除。

索引二叉搜索树

索引二叉搜索树源于普通二叉搜索树, 只是每个节点添加一个leftsize域, 这个域的值就是该节点左子树的元素个数。

平衡搜索树

平衡二叉树结构有两种:AVL和红黑树。AVL和红黑树适合内部储存的应用, B-树适合外部储存的应用(例如:存储在磁盘上的大型词典)。这些平衡树结构都可以在最坏情况下用时 $O(\log n)$ 实现字典操作和按名次的操作。

STL类map和multimap使用的是红黑树结构, 以保证查找, 插入和删除操作具有对数级的时间性能。¹ AVL树和红黑树都使用"旋转"来保持平衡。AVL树对每个插入操作最多需要一次旋转, 对每个删除操作最多需要 $O(\log n)$ 次旋转。而红黑树对每个插入和删除操作, 都只需要一次旋转。

AVL树(平衡二叉树)

如果树的高度总是 $O(\log n)$, 我们就能保证查找, 插入和删除的时间为 $O(\log n)$ 。

平衡树

最坏情况下的高度为 $O(\log n)$ 的树称为平衡树。

AVL树的定义:

一棵空的二叉树是AVL树;如果T是一棵非空二叉树, T_L 和 T_R 分别是其左子树和右子树, 那么当T满足以下条件时, T是一棵AVL树:

1): T_L 和 T_R 是 AVL 树。

2): $|h_L - h_R| \leq 1$, 其中 h_L 和 h_R 分别是 T_L 和 T_R 的高。

一棵 AVL 搜索树既是二叉搜索树，也是 AVL 树。

红黑树

红黑树是很多平衡查找树中的一种，它能保证在最坏的情况下，基本的动态集合操作的时间为 $O(\log n)$ 。通过对任何一条从根到叶子的路径上的各个节点着色方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因而使接近平衡的。

红黑树每个节点包含五个域:color, key, left, right, 和 p。如果某节点没有子节点或者没有父节点，则该节点相应的指针(p)域包含值 NIL。我们将把这些 NIL 视为指向二叉查找树的外节点(叶子)的指针，而把带关键字的节点视为树的内节点。

一棵二叉查找树满足下面的红黑性质就是一棵红黑树：

1) 每个节点或是红色，或是黑色。

2) 根节点是黑色的。

3) 每一个叶节点(NIL)是黑色的。

4) 如果一个节点是红色的，则它的两个儿子都是黑的。

5) 对每个节点，从该节点到其子孙节点的所有路径上包含相同数目的黑节点。

旋转²

由于在红黑树上运行插入与删除的时候，会对树做修改，而导致修改后的树不再是红黑树。为了保持这些红黑树性质，就要改变树中某些节点的颜色与指针结构。指针结构的修改是通过旋转来完成的。

二叉查找树中旋转操作。操作 LEFT-ROTATE(T, x) 通过改变常数个指针来将左边两个节点的结构变成右边的结构。左边的结构可以使用相反的操作 RIGHT-ROTATE(T, x) 来转变成左边的结构。字母 α, β 以及 γ 代表任意的子树。旋转操作保留二叉树的属性： α 的关键字在 $key[x]$ 之前， $key[x]$ 的关键字在 β 之前， β 的关键字在 $key[y]$ 之前， $key[y]$ 的关键字在 γ 之前。

一个问题就是为啥可以通过旋转来维持红黑树的性质？

1. 《数据结构，算法与应用--C++语言描述》P359 ↪

2. 《算法导论》P165 ↪

STL的底层机制

STL的底层机制都是以RB-Tree(红黑树)完成的。RB-tree也是一个独立容器，但并不给外界使用。红黑树这个名字的由来就是由于树的每个结点都被染上红色或者黑色，结点所着颜色被用来检测树的平衡性。在对结点插入和删除的操作中，肯恶搞会被旋转来保护树的平衡性。平均和最坏情况下的插入，删除，查找时间都是 $O(\log n)$ 。

一个红黑树是一棵二叉查找树，除了二叉查找树带有的一般要求外，它还具有下列的属性：

1. 结点为红色或者黑色。
2. 所有叶子结点都是空结点，并且被着为黑色。
3. 如果父节点是红色，则两个子节点都是黑色的。
4. 节点到其子孙节点的每条路径上都包含相同数目的黑色节点。
5. 根节点是黑色的。

map底层是以红黑树实现的。

树

霍夫曼编码

霍夫曼树

首先定义树的路径长度：从树根到每一个节点的路径长度之和称为树的路径长度。树有n个节点，就是n个长度之和。

$$L(T) = \sum_{i=1}^n h_i$$

其中 h_i 表示从根结点到该节点的路径或者深度；T代表这棵树。

再考虑带权重的路径长度，假设每一个节点都有一个权重 w_k ，因此带权路径长度是：

$$L(T, W) = \sum_{i=1}^n w_i h_i$$

霍夫曼树：带权路径长度WPL最小的二叉树成为霍夫曼树，或者称为最优二叉树。

霍夫曼算法

该算法是构造霍夫曼树的算法，算法如下：

1. 根据给定的n个权值 w_1, w_2, \dots, w_n 构成n棵二叉树的集合 $F = [T_1, T_2, \dots, T_n]$ ，其中每棵二叉树 T_i 中只有一个带权为 w_i 的根结点，其左右子树都为空。
2. 在F中选取两棵根节点权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左右子树上根结点的权值之和。
3. 在F中删除这两棵树，同时将新得到的二叉树加入F中。
4. 重复步骤2和3，直到F只含有一棵树为止。这棵树就是霍夫曼树。

算法的正确性见《算法导论》第16.3节。

霍夫曼编码

为了解决数据传输的最优化问题，霍夫曼发明了霍夫曼编码。

设需要编码的字符集为 d_1, d_2, \dots, d_n 。各个字符在电文中出现的次数会频率集合是 w_1, w_2, \dots, w_n ，以 d_1, d_2, \dots, d_n 作为叶子节点，以 w_1, w_2, \dots, w_n 作为相应叶子节点的权值来构造一棵霍夫曼树。规定霍夫曼树的左分支代表0，右分支代表1，则从根结点到叶子结点所经过的路径组成的0和1的序列便为给节点对应字符的编码，称为霍夫曼编码¹。

¹. 《大话数据结构》P207 ↵

最小生成树算法

最小生成树

这一节讨论图里面的最小生成树问题。也就是寻找连接图中n个点的最短路径问题。

可以把这一个问题转化成一个无向连通图 $G=(V,E)$, 其中 V 是顶点集合, V 是可能的连接边集合。对图中每一条边 $(u,v) \in E$, 都有一个权值 $w(u,v)$ 表示连接顶点u和v的代价。我们希望找到一个无回路的子集 $T \subseteq E$, 它连通了所有的顶点, 且其权值之和:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

为最小。因为 T 是无回路且连接所有的顶点, 因此它必然是一棵树, 因此称为生成树。把确定树 T 的问题称为最小生成树问题。

最短路径Dijkstra算法

需要回答一个问题，最小生成树算法与最短路径算法这些图算法之间的关联，也就是他们基于什么相同的原理？以及他们与最优化算法之间的关系是什么？图算法是否可以连续化？

首先定义从 u 到 v 之间的最短路径的权为：

$\delta(u, v) = \min(w(p) : u \rightarrow v)$, 如果存在一条从 u 到 v 的路径。

$\delta(u, v) = \infty$, 否则。

松弛技术

这些算法主要使用了松弛技术¹。

也就是说，对每个顶点 $v \in V$ ，都设置一个属性 $d[v]$ ，用来描述从源点到 v 的最短路径上权值的上界，称为最短路径估计，我们用 $\Theta(V)$ 时间的过程来对最短路径估计和前驱初始化：

第一步是对最短路径估计的值 $d[v]$ 以及 v 的前驱域 $\pi[v]$ 进行初始化。

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in V[G]$:

do $d[v] \leftarrow \infty$

$\pi[v] \leftarrow NIL$

$d[s] \leftarrow 0$

在松弛一条边 (u, v) 的过程中，要测试是否可以通过 u ，对迄今找到的 v 的最短路径进行松弛；如果可以改进的话，则更新 $d[v]$ 与 $\pi[v]$ 。一次松弛操作可以减小最短路径估计的值 $d[v]$ ，并更新 v 的前驱域 $\pi[v]$ 。下面的伪代码对边 (u, v) 进行了松弛操作。

RELAX(u, v, w)

if $d[v] > d[u] + w(u, v)$:

then do $d[v] \leftarrow d[u] + w(u, v)$

$\pi[v] \leftarrow u$

基于松弛的算法。松弛是改变最短路径和前驱的唯一方式，这节的算法之间的区别在于对每条边进行松弛操作的次数不同，以及对边进行松弛操作的次序有所不同。在Dijkstra算法以及关于有向无回路图的最短路径算法中，对每条边执行一次松弛操作。在Bellman-Ford算法中，对每条边执行多次松弛操作。

Bellman-Ford算法

Bellman-Ford算法能在一般的情况下（带有负权边的情况下），解决单源最短路径问题。

BELLMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)

for $i \leftarrow 1$ to $|V[G]| - 1$

do for each edge $(u, v) \in E[G]$

do RELAX(u, v, w)

for each edge $(u, v) \in E[G]$

```
do if  $d[v] > d[u] + w(u, v)$ 
    then return FALSE
return TRUE
```

Dijkstra算法

```
DIJKSTRA(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
 $S \leftarrow \emptyset$ 
 $Q \leftarrow V[G]$ 
while  $Q \neq \emptyset$ 
    do  $u \leftarrow EXTRACT-MIN[Q]$ 
         $S \leftarrow S \cup u$ 
        for each vertex  $v \in Adj[u]$ 
            do RELAX(u, v, w)
```

|| 1. 《算法导论》单源最短路径 ↵

图像与信号处理

这里将要讨论图像与信号处理的算法。主要包括：

1. 时域的滤波
2. 频率空间的滤波
3. 采样定理以及信号恢复
4. 图像分割
5. 特征提取(边缘提取)
6. 对象检测, 目标跟踪, 语义分割, 实例分割

还是先整理一些基础的, 毕竟很多深的东西也暂时用不着。

图像几何变换

1. 仿射变换
 - i. 平移
 - ii. 放大与缩小
 - iii. 旋转
 - iv. 插值
2. 投影变换
3. 极坐标变换

傅里叶变换

1. 二维离散傅里叶变换
2. 傅里叶幅度谱与相位谱
3. 卷积定理
4. 快速傅里叶变换

二维离散傅里叶变换

二维离散傅里叶变换只是普通二维傅里叶变换的离散形式。考虑二维情况，主要是为了处理图像。

令 $f(x,y)$ 代表一幅大小为 $M \times N$ 的数字图像 I ，其中 $x = 0, 1, 2, \dots, M - 1, y = 0, 1, 2, \dots, N - 1$ ，由

$F(u,v)$ 表示的 $f(x,y)$ 的二维离散傅里叶变换(DFT)由下式给出：

$$F(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-j2\pi(ux/M+vy/N)}$$

其中 $u = 0, 1, 2, \dots, M - 1, v = 0, 1, 2, \dots, N - 1$ 公式里面之所有除以 M 与 N ，表示我们把图当成一个周期性结构中的一个单元，因此他在 x 方向的周期是 M ，在 y 方向的周期是 N 。

离散傅里叶反变换(IDFT)的形式为：

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) e^{j2\pi(ux/M+vy/N)}$$

其中 $x = 0, 1, 2, \dots, M - 1, y = 0, 1, 2, \dots, N - 1$ 。在这个公式里面， $F(u,v)$ 被称为展开的傅里叶系数。

傅里叶幅度谱与相位谱

对图像的傅里叶变换后得到复数矩阵 F ，我们就通过幅度谱和相位谱两个度量来了解该复数矩阵。分别记 Real 为矩阵 F 的实数部分， Imag 为矩阵 F 的复数部分。

$$F = \text{Real} + i * \text{Imag}$$

幅度谱

幅度谱(Amplitude Spectrum)又称为傅里叶谱，通过如下公式计算：

$$\text{Amplitude} = (\text{Real}^2 + \text{Imag}^2)^{1/2}$$

其中：

$$\text{Amplitude}(u, v) = (\text{Real}(u, v)^2 + \text{Imag}(u, v)^2)^{1/2}$$

根据傅里叶变换公式可以知道：

$$F(0, 0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y)$$

则 $Amplitude(0, 0) = F(0, 0)$, 这是幅度谱中最大的值, 他可能比其它项大几个数量级。也称为直流成分。

相位谱

相位谱(Phase Spectrum)通过如下公式计算:

$$Phase = \arctan\left(\frac{Imag}{Real}\right)$$

其中:

$$Phase(u, v) = \arctan\left(\frac{Imag(u, v)}{Real(u, v)}\right)$$

显然复数矩阵F可以用幅度谱与相位谱来表示:

$$F = Amplitude.*\cos(Phase) + i * Amplitude.*\sin(Phase)$$

$$F(u, v) = Amplitude(u, v).*\cos(Phase(u, v)) + i * Amplitude(u, v).*\sin(Phase(u, v))$$

采样定理

1. 信号重建
2. 混叠
3. 稀疏信号重建与采样定理的冲突

采样定理

如果函数 $x(t)$ 没有频率高于 B Hz的成分，为了完美的重建这函数，采样频率 F_S 需要满足 $F_S > 2B$ 。 $2B$ Hz就是Nyquist频率。

采样过程与卷积

时域的周期性采样，相当于原始(连续)信号乘以一个周期性采样函数(δ 函数)。因为对于傅里叶变换，时域的相乘，对于的是频率的卷积。可以用下图解释。



一个周期为 T 的脉冲函数的傅里叶变换也是一个脉冲函数，变换后的周期是 $1/T$ 或者 $2\pi/T$ 。

一个函数与脉冲函数做卷积，相当于这个函数在起变量空间的周期性复制，周期就是脉冲函数的周期。

Aliasing

如果采样的频率小于Nyquist频率，就会出现频率的混叠(Aliasing)。



For band limited signal, the largest frequency is F_M , if sample rate $F_s < 2F_M$, then will produce aliasing.

What will happen if the signal has aliasing?

如果出现了Alias，我们就不能完美的重建原始信号。下面我们以一个余弦信号的采样重建为例。我们发现当采样频率小于Nyquist频率的时候，我们重建出的信号不再是原始信号。对于余弦信号，重建后的信号多了一些频率成分。



实际的信号重建方法

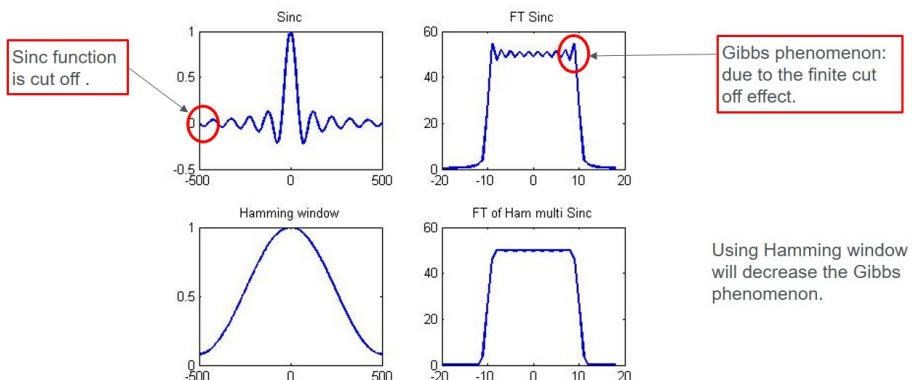
实际的信号一般没有截至频率，尤其对于一阶导数不连续的信号(比如矩形信号，我们需要很高阶的频率成分才能生成这种一阶不连续信号)。因此要完美重建信号的采样频率是无穷大，这是不可能的，因此我们需要一些辅助措施，以较低的采样频率也能较好的重建信号。一般通过加窗函数来实现。

1. 实际采样过程，我们只采集频率小于采样频率的频率成分，这就相当于在频率空间加了一个矩形窗。矩形窗傅里叶变换之后的函数就是Sinc函数，信号的恢复就是相当于时域的采样点函数与Sinc函数的卷积，也就是一些一系列不同强度的Sinc函数的叠加。



1. 如果Sinc函数也被截断了，则其傅里叶变换之后的函数不再是完美的矩形函数，会在角处出现震荡，这就是Gibbs效应。

通过对Sinc函数加Hamming窗能减缓这种效应。相应的，我们也可以通过Hamming窗来实现信号的恢复，或者说信号的插值。来获得高分辨率的图像。



$$\text{Hamming-Sinc interpolation: } x_r(t) = \sum_{n=-N}^N x_a(nT) \frac{\sin\left(\frac{(t-nT)\pi}{T}\right)}{\left(\frac{(t-nT)\pi}{T}\right)} \left(0.54 + 0.46 \cos\left(\frac{(t-nT)\pi}{T}\right) \right)$$

稀疏信号恢复与采样定理的冲突

对比度增强

1. 直方图均衡

通过一个函数变换，使得像素的概率密度在想要的区间是均匀分布的，实际上使得图像对应的熵变得最大，也就是信息最多。变换后的变量不是原来的像素值而是累计概率密度分布(CDF)：

考虑灰度范围在 $0 \sim 1$ 之间，此时图像的归一化直方图就是概率密度分布(PDF)：

$$p(x), 0 \leq x \leq 1$$

满足：

$$\int_{x=0}^1 p(x)dx = 1$$

设转换前图像的概率密度函数为 $p_r(r)$ ，转换后的概率密度函数是 $p_s(s)$ 。转换的映射关系是 $s=f(r)$ 。

由概率论的知识可以得到：

$$p_s(s) = p_r(r) \frac{dr}{ds}$$

如果我们想转换后的像素值分布均匀，也就是：

$$p_s(s) = 1, 0 \leq s \leq 1$$

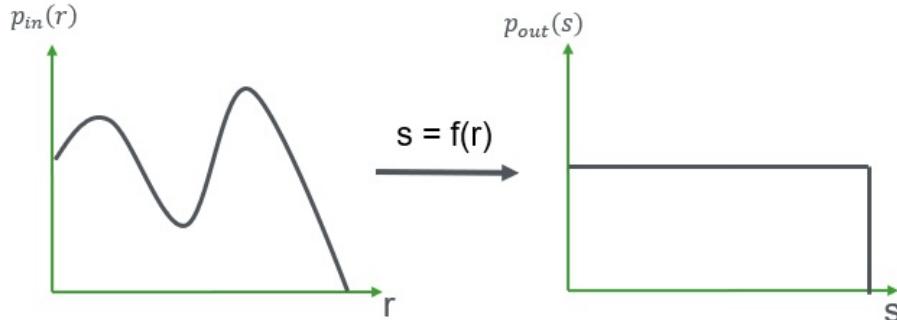
$p_s(s) = 1$ 代入 $p_r(r)$ ，可以得到：

$$p_r(r) = \frac{ds}{dr}$$

因此可以得到：

$$s = f(r) = \int_{x=0}^r p_r(u)du$$

上式被称为图像的累计分布函数(CDF)。下图就是转换前后的图像概率分布：



图像滤波

1. 时域滤波
 - i. 中值
 - ii. 最大值, 最小值
 - iii. *Alpha-trimmed* 均值
2. 频域滤波
 - i. 低通, 高通
3. 维纳滤波
4. 约束最小二乘法滤波

时域滤波

中值

对图像中任意一点, 用改点领域 $H \times W$, 大小区域内的中值代替改点的值, 满足H, W是奇数。这种方式能减少椒盐噪声。

最小值滤波

用该点领域的最小值代替该点, 能去除盐噪声(值很大)。

最大值滤波

用该点领域的最大值代替该点, 能去除胡椒噪声(值很小)。

频率滤波

低通滤波

高通滤波

Band Reject滤波

总结如下：

Lowpass filters

Ideal	Butterworth	Gaussian
$H(u, v) = \begin{cases} 1, & \text{if } D(u, v) \leq D_0 \\ 0, & \text{if } D(u, v) > D_0 \end{cases}$	$H(u, v) = \frac{1}{1 + [D(u, v)/D_0]^{2n}}$	$H(u, v) = e^{-D^2(u, v)/2D_0^2}$

Highpass filters

Ideal	Butterworth	Gaussian
$H(u, v) = \begin{cases} 0, & \text{if } D(u, v) \leq D_0 \\ 1, & \text{if } D(u, v) > D_0 \end{cases}$	$H(u, v) = \frac{1}{1 + [D_0/D(u, v)]^{2n}}$	$H(u, v) = \frac{1 - e^{-D^2(u, v)/2D_0^2}}{1 - e^{-D^2(u, v)/2D_0^2}}$

Bandreject filters

Ideal	Butterworth	Gaussian
$H(u, v)$ $= \begin{cases} 0, & \text{if } D_0 - \frac{W}{2} \leq D \leq D_0 + \frac{W}{2} \\ 1; & \text{otherwise} \end{cases}$	$H(u, v) = \frac{1}{1 + [\frac{DW}{D^2 - D_0^2}]^{2n}}$	$H(u, v) = 1 - e^{-[\frac{D^2 - D_0^2}{DW}]^2}$

一般，对于周期性噪声，比如周期性条纹，通过对图像做FT，在频率中找到这些造成周期性噪声的频率区域，再去除就可以了。这是典型的频率滤波。

频率滤波与边缘检测的关系

边缘，对应的是图像的高频部分，因此，高通滤波主要保留图像的高频部分，也就是边缘部分，相当于做边缘检测，会锐化原始图像。

低通，只是保留图像的低频部分，也就是会钝化图像，使得图像变模糊。

边缘检测

1. 点检测
2. 线检测
3. 霍夫变换
4. Sobel算子
5. Canny算子
6. Laplacian算子
7. 高斯拉普拉斯边缘监测

总的而言，梯度算子(有一阶的，也有二阶的)都可以用来作为边缘检测

点检测

点检测的模板一般如下：

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

线检测

下面的模板分别对于水平, 45度, 负45度以及垂直线的检测模板：

$$\begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \quad \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$

边缘检测算子

Sobel算子

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Prewitt算子

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Roberts算子

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

角点检测

所谓角，就是在该点，x,y方向的导数都很大，近乎不连续。因此，如果满足x,y方向导数很多，就是角点。

Harris角点检测

图像复原

1. 空间滤波
2. 频率滤波
3. 退化函数建模
4. 直接逆滤波
5. 维纳滤波
6. 约束最小二乘法滤波

图像复原，可以看成是一个二维矩阵受到一些干扰，我们需要把它恢复到未受干扰前的样子。一个类似的问题就是推荐系统中的矩阵填充，我们需要填充矩阵中那些矩阵元为0的元素。

受干扰的矩阵 H_d 是原始矩阵 H_0 加上一个微扰矩阵 H_1 。我们要尽力恢复 H_0 。

通过空间滤波和频率滤波的方法以及讨论过了，这里不再讨论，我们只讨论其它图像复原的方法。

图像退化/复原处理的模型

在这里我们要讨论一种退化与复原模型。就是用退化函数对退化过程建模，在数学上就是退化函数 H 作用在原始图像 $f(x,y)$ 上，再加上一些噪声 $\eta(x,y)$ 而产生一幅退化的图像 $g(x,y)$ ：

$$g(x,y) = H[f(x,y)] + \eta(x,y)$$

我们需要通过退化函数 H 、噪声 $\eta(x,y)$ 以及退化后的图像 $g(x,y)$ 来得到原始图像的估计 $\hat{f}(x,y)$ 。具体过程如下所示：

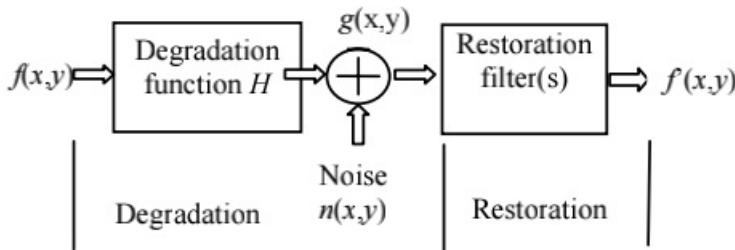


图 1 图像的退化和恢复模型
Fig.1 Optical image of degradation and restoration

通过，我们对退化函数 H 、噪声 $\eta(x,y)$ 知道的越多，则 $\hat{f}(x,y)$ 越接近 $f(x,y)$ 。

考虑 H 是线性的，空间不变(也就是与图像中特定点无关)的过程。那么退化图像在空间域由下面卷积给出：

$$g(x,y) = h(x,y) * f(x,y) + \eta(x,y)$$

在频率域可以通过卷积定理给出：

$$G(u,v) = H(u,v)F(u,v) + N(u,v)$$

大写字母表示的函数表示的是空间域的傅里叶变换。退化函数 $H(u,v)$ 有时成为光传递函数(OTF)。

大写字母表示的函数表示的是空间域的傅里叶变换。退化函数 $H(u,v)$ 有时成为光传递函数(OTF)。空间域的 $h(x,y)$ 被称为点扩散函数(PSF), 其实就是傅里叶光学那一套东西。对于任何种类的输入, 让 $h(x,y)$ 作用于点光源来得到退化的特征。OTF和PSF是傅里叶变换对。

逆滤波

当考虑退化函数是线性的, 空间不变的函数时, 我们在频率域就有:

$$G(u,v) = H(u,v)F(u,v) + N(u,v)$$

退化可以用复原滤波器消除, 该滤波器的传递函数是退化 H 的逆。我们可以从退化图像 G 根据下面的公式得到未退化之前的原始图像 F (原始图像 $f(x,y)$ 的傅里叶变换):

$$F(u,v) = G(u,v)H^{-1}(u,v) - N(u,v)H^{-1}(u,v)$$

上面方法也存在两个问题:

1. $H^{-1}(u,v)$ 接近于0的地方不好处理, 一般就是高频的区域。
2. 没有足够的信息来很好的确定噪声 $N(u,v)$ 。

维纳滤波

维纳滤波在图像复原公式中合并考虑一些先验信息从而将有关噪声的特性考虑进来。维纳滤波复原给出了对未被噪声污染的原始图像的最小均方误差 e^2 估计 \hat{f} :

$$e^2 = E([f(i,j) - \hat{f}(i,j)]^2)$$

E 代表取均值。复原图像的估计在频率域中的表达式为:

$$\hat{F}(u,v) = [\frac{1}{H(u,v)} \frac{|H(u,v)|^2}{|H(u,v)|^2 + S_\eta(u,v)/S_f(u,v)}]G(u,v)$$

$$|H(u,v)|^2 = H^*(u,v)H(u,v)$$

$H^*(u,v)$ 是 $H(u,v)$ 复共轭。

$$S_\eta(u,v) = |N(u,v)|^2 = \text{噪声的功率谱}$$

$$S_f(u,v) = |F(u,v)|^2 = \text{未退化图像的功率谱}$$

$S_\eta(u,v)/S_f(u,v)$ 被称为信噪功率比。

我们感兴趣的是两个量: 平均噪声功率和平均图像功率, 定义为:

$$\eta_A = \frac{1}{MN} \sum_u \sum_v S_\eta(u,v)$$

$$f_A = \frac{1}{MN} \sum_u \sum_v S_f(u,v)$$

约束的最小二乘法(规则化)滤波¹

空间域的复原问题如下：

$$g(x, y) = h(x, y) * f(x, y) + \eta(x, y)$$

表示成向量形式就是：

$$\mathbf{g} = \mathbf{Hf} + \boldsymbol{\eta}$$

约束的最小二乘法(规则化)滤波就是求解带约束的优化问题，优化是基于平滑度测量的复原最优化，也就是优化如下函数：

$$C = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} [\nabla^2 f(x, y)]^2$$

而约束条件就是：

$$\|\mathbf{g} - \mathbf{H}\hat{\mathbf{f}}\|^2 = \|\boldsymbol{\eta}\|^2$$

\hat{f} 就是非退化函数的估计。这种方式就是保证复原的函数平滑度最优。求解的问题就是带约束的优化问题，可以参照数值优化那一章里面的带约束的优化问题那一部分。

1. 《数字图像处理的Matlab实现》Rafael C.Gonzalez 第四章 [←](#)

图像压缩

1. 图像压缩理论¹
 - i. 图像冗余
 - ii. 香农定理
2. 霍夫曼编码
3. 算数编码
4. JPEG与JPEG2000压缩标准

压缩理论

香农第一定理

香农第一定理-无损编码理论：数据的核心在于信息量，任何压缩手段(无损压缩)都不可能使压缩结果小于原数据的信息量大小。

信息量由信息熵来刻画。

将一幅图像看作是一个具有随机输出的信息源，它从符号集0-255中产生符号序列。设信号源符号

集合 $B = [b_i]$ ，每个符号出现的概率分别是 P_i ，故单个符号的自信息为 $I(b_i) = -\log(P(b_i))$ ，信息源输出的平均信息可以由下面公式求得：

$$E(B) = - \sum_{i=0}^n P(b_i) \log P(b_i)$$

$E(B)$ 就是信源的熵。熵代表了信源输出单个字符时包含的平均信息量的大小。当各个字符出现的概率均相等的时候，熵值最大。

图像编码：

图像由多个像素点构成，相当于信源产生了多个符号，则码字总长 L_{total} 满足：

$$\lim_{n \rightarrow \infty} \frac{L_{total}}{n} = E(B)$$

即平均码字长将趋近于信源的熵。

编码效率：

$$\eta = \frac{E(B)}{L_{total}/n}$$

η 的值在0 ~ 1之间，越接近于1，编码效率越高， η 越小，编码效率越低。

霍夫曼编码

在算法与数据结构中讨论了霍夫曼编码。主要步骤如下：

1. 统计信源中各字符出现的概率
2. 构建霍夫曼树
3. 对图像进行编码

霍夫曼解码则根据霍夫曼树以及压缩数据来进行解码。

1. 《数字图像处理与机器视觉》张铮 徐超 第10章 ↵

图像分割

1. 边缘检测
2. 霍夫变换
3. 阈值分割
4. 区域分割

边缘检测前面已经讨论过了，因此我们主要讨论下面三个话题。

霍夫变换

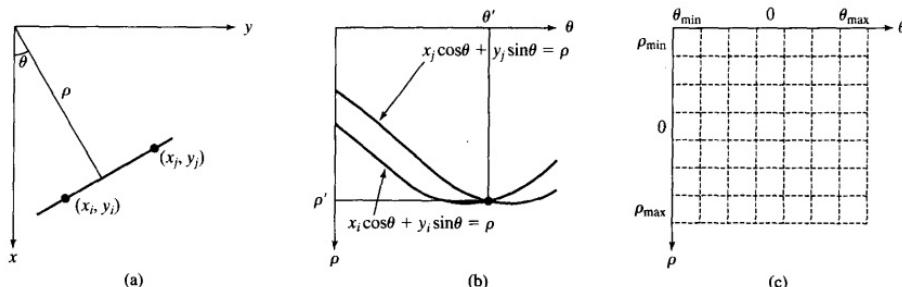
对于线条检测而言，霍夫变换不是在图像中直接检测某种类型的线条(比如直线或者圆)，它是在决定这些线条的参数空间(比如直线就是斜率与截距，二维的圆对应的就是三个参数，圆心与半径)去做符合特定线条的线条个数，把参数空间划分为方格，统计方格中符合条件的个数，也就是生成以累加器数组，也称为霍夫变换矩阵。有一点要注意的是，霍夫变换只能处理二值图像。因此一般先要把图像通过阈值方法变成二值图像。

直线检测

我们把直线表示成如下的形式：

$$\rho = x \cos \theta + y \sin \theta$$

因此参数空间就是 (ρ, θ) 。



图a是在原始空间的两点，图b是在参数空间，两条曲线分别所有过 i,j 两点的所有参数形成的所有参数形成的曲线， i,j 有一个交点，该点对应的就是过 i,j 两点的直线的参数。图c就是统计不同的参数区域块内的点数目，点的数目就是原始图像中任何两点形成的线条的参数是否落在该参数区域内，再就加1。这个参数累加器数组就是霍夫变换矩阵。

具体步骤就是，对于原始图像空间的每一个非0点，我们在参数空间画出所有过该点所形成的曲线，比如 θ 每隔一度去做一条直线，因此，每点我们需要做180条直线，在参数空间就是180个点。然后得到霍夫变换矩阵。矩阵中的亮点就是直线。

其它曲线

我们可以用霍夫变换来检测任何曲线，只要该曲线可以用参数来描述，假设参数的维数是n维，也就是需要n个参数来描述该曲线(比如二维直线n=2,二维圆n=3)，因此霍夫变换矩阵的参数维度也就是n维，每一维采样点固定为S，则对于原始图像空间的每个点，在参数空间需要取 S^n 个点，因此随着n的增大会发生维度灾难。

如下是用Hough变换来寻找半径已知的圆，也就是寻找过原始图像中一点的半径R一定的所有圆。这些所有的圆的参数，在参数空间也就是一个圆，这个圆就是以该点为圆心，做半径w为R的圆。在参数空间通过霍夫变换矩阵：



阈值分割¹

全阈值分割

全阈值分割指的是将灰度值大于阈值threshold的像素设为白色，小于或者等于阈值的像素设为黑色。

$$\begin{aligned} O(x, y) &= 255. \text{ if } I(x, y) > \text{thresh} \\ O(x, y) &= 0. \text{ if } I(x, y) \leq \text{thresh} \end{aligned}$$

局部阈值分割

不像全阈值分割中阈值在整个图像区域是个常数，在局部阈值分割中，阈值是位置的函数，也是一个与图像一样大小的矩阵。此时有：

$$\begin{aligned} O(x, y) &= 255. \text{ if } I(x, y) > \text{thresh}(x, y) \\ O(x, y) &= 0. \text{ if } I(x, y) \leq \text{thresh}(x, y) \end{aligned}$$

局部阈值分割的核心是计算阈值矩阵，比较常用的是自适应阈值算法(又称为移动平均值算法)。这是一种简单但是高效的局部阈值算法，其核心思想就是把每一个像素的领域平均值作为该位置的阈值。

Otsu阈值处理

熵算法

自适应阈值

步骤如下：

1. 对图像进行平滑处理，平滑结果记为 $f_{smooth}(I)$ ，其中 $f_{smooth}(I)$ 可以是均值平滑，中值平滑或高斯平滑。
2. 自适应阈值矩阵 $Thresh = (1 - ratio) * f_{smooth}$ ，一般令 $ratio = 0.15$ 。
3. 利用局部阈值分割进行阈值分割，操作如上面所示。

1. 《OpenCV算法精解--基于Python与C++》张平 ↵

形态学处理

1. 膨胀
2. 腐蚀
3. 形态学重建

膨胀涉及到的是集合的加法，而腐蚀涉及到的是集合的减法。膨胀和腐蚀都不是可逆运算。

特征提取

1. PCA
2. 基于PCA的人脸重建
3. K-均值
4. 高斯混合模型

特征提取涉及到了机器学习的方法，主要还是一些降维(PCA)与聚类(K-Mean, GMM)的方法。这些方法在机器学习那一章我们有比较多的讨论，因此在这里我不会过多介绍。

EM算法及其应用

1. 人脸检测
2. 目标识别
3. 分割
4. 正脸识别
5. 改变人的姿态(回归)

EM(期望最大化)算法是机器学习中很重要的求解模型参数的一种方法。在机器学习那章里面讨论了，这里就不讲了。这里主要讨论它在计算机视觉中的一些应用。

图模型

图模型在机器学习中那一章会具体的讲述。这里这是占个位置。

1. 有向图
2. 无向图
3. MM与HMM
4. MAP

链式模型与树模型

1. 马尔科夫模型
2. 链式MAP推理
3. 树的MAP推理
4. 应用
 - i. 手势跟踪
 - ii. 立体视觉
 - iii. 形象化结构
 - iv. 分割

视觉词模型

视觉词模型暂时还不了解，先占个位置。具体可以看《计算机视觉--模型，学习与推理》的第20章。

1. 词袋
2. LDA(隐狄利克雷分布)
3. 单一创作-主题模型
4. 场景模型
5. 星座模型
6. 应用
 - i. 视频搜索
 - ii. 行为识别

Kriging插值

<https://xg1990.com/blog/archives/222>

空间插值问题，就是在已知空间上若干离散点 (x_i, y_i) 的某一属性(如气温, 海拔)的观测值

$z_i = z(x_i, y_i)$ 的条件下，估计空间上任意一点 (x, y) 的属性值的问题。

直观来讲，根据地理学第一定律，

大意就是，地理属性有空间相关性，相近的事物会更相似。由此人们发明了反距离插值，对于空间上任意一点 (x, y) 的属性 $z = z(x, y)$ ，

定义反距离插值公式估量 $\hat{z} = \sum_{i=0}^n \frac{1}{d_i^\alpha} z_i$

其中 α 通常取 1 或者 2。

即，用空间上所有已知点的数据加权求和来估计未知点的值，权重取决于距离的倒数(或者倒数的平方)。

那么，距离近的点，权重就大；距离远的点，权重就小。反距离插值可以有效的基于地理学第一定律估计属性值空间分布，但仍然存在很多问题：

α 的值不确定 用倒数函数来描述空间关联程度不够准确

因此更加准确的克里金插值方法被提出来了

克里金插值公式 $\hat{z}_o = \sum_{i=0}^n \lambda_i z_i$

其中 \hat{z}_o 是点 (x_o, y_o) 处的估计值，即 $z_o = z(x_o, y_o)$

假设条件：

1. 无偏约束条件 $E(\hat{z}_o - z_o) = 0$
2. 优化目标/代价函数 $J = Var((\hat{z}_o - z_o))$ 取极小值
3. 半方差函数 r_{ij} 与空间距离 d_{ij} 存在关联，并且这个关联可以通过这两组数据拟合出来，因此可以

用距离 d_{ij} 来求得 r_{ij}

半方差函数 $r_{ij} = \sigma^2 - C_{ij}$ ，等价于 $r_{ij} = \frac{1}{2}E[(z_i - z_j)^2]$

其中： $C_{ij} = Cov(z_i, z_j) = Cov(R_i, R_j)$

求得 r_{ij} 之后，我们就可以求得 λ_i

$$\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} & 1 \\ r_{21} & r_{22} & \cdots & r_{2n} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ r_{n1} & r_{n2} & \cdots & r_{nn} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ 0 \end{bmatrix} = \begin{bmatrix} r_{1o} \\ r_{2o} \\ \cdots \\ r_{no} \\ 1 \end{bmatrix}$$

第四章 最优化方法

这章主要讨论常见的非线性最优化方法，以及优化技巧与不同方法的优缺点。最优化方法的分类很多，按照不同方式主要分为如下四种。

无约束与有约束问题

最优化问题分为有约束优化问题与无约束优化问题。

1. 无约束优化问题

$$\min f(x)$$

2. 有约束优化问题

$$\min f_0(x)$$

$$\text{s.t. } f_i(x) \leq 0, i = 1, 2, \dots, m$$

$$\text{s.t. } h_i(x) = 0, i = 1, 2, \dots, p$$

一般我们把带约束问题转化为无约束优化问题进行求解。比如拉格朗日乘子法：

$$\min L(x, \lambda, v) = f_0(x) + \sum_{i=1}^m \lambda_i f_i(x) + \sum_{i=1}^p v_i h_i(x)$$

优化搜索方式

主要有两大类方法：

1. 线性搜索：先确定方向，再确定步长，典型的就是梯度下降法与牛顿法，quasi-newton法
2. 置信域方法：先确定步长，再找步长方向。常见的有Gauss-Newton法，L-M, Dog-Leg方法

利用导数的阶数

从应用的泰勒展开的阶数来说，可分为一阶算法与二阶算法。

1. 一阶方法
 - i. SGD
 - ii. Mini-batch SGD
2. 二阶方法
 - i. 牛顿法
 - ii. 拟牛顿法
 - i. BFGS
 - ii. DFP

iii. Brody类方法

问题凸与否

就解决的问题而言，有凸优化算法与非凸优化算法。对凸优化问题而言，由于局部极小就是全局最小，因此通过梯度下降方法就可以很好的解决。大多数我们面对的是非凸优化问题，存在大量的局部极小值，我们也不需要得到全局最小值，只需要求得一个较好的局部极小值就可以了，因为局部极小值对应的解是稳定的，就可能是一个满足系统要求的解。

1. 全局最小算法

- i. 模拟退火
- ii. 遗传算法

本章写作方式

思考这一章该怎么去写？但是得先回答以下两个问题，你写这一章的目的是什么？你想把它写成什么样子？答案不能独立于问题而存在，答案与问题是一对，[Key,Value]，且一个key可以对应与多个value。

本章写作的目的是为了解决实际的问题，实际的问题一般分为无约束优化问题与带约束问题优化问题，因此本章的分类是就问题是否带约束来分的。

带约束的优化问题也可以细分，看求解的问题线性而是非线性，非为带约束线性优化问题，比如普通的SVM，带约束的非线性优化问题(非线性优化是相对于线性规划而言的，除了线性规划以外的优化问题称为非线性优化问题)，比如核函数SVM。因此优化问题可以分为如下三类：

1. 线性规划

线性规划暂时不讲，因为现在主要讨论非线性优化问题，此外可以通过平方把线性优化问题转化为非线性优化问题，一个例子就是求解线性方程组问题转化为最小二乘问题。

2. 无约束非线性优化问题

一维搜索问题，包含二分法，弦割法，牛顿法，(黄金分割法)这些也不重用，因此主要讨论如下几种算法。

- 1. 梯度下降法
- 2. 牛顿法
- 3. 拟牛顿法
- 4. L-M(Levenberg-Marquardt)算法
- 5. DogLeg算法
- 6. 共轭梯度法

3. 带约束非线性优化问题

原则就是通过拉格朗日乘子法把等式约束，用KKT乘子法把不等式约束问题转化为无约束优化问题来求解，这样可以利用无约束优化问题来求解。

主要有如下几种方法

1. 拉格朗日乘子法
2. 罚函数法
3. 障碍函数法
4. 增广拉格朗日乘子法
5. ADMM

需要做的事情

1. 把最前沿的分布式优化算法，比如ADMM算法加进来。Done
2. 分布式机器学习中应用的优化算法要有清晰的轮廓，以及她与单机机器学习系统中的优化算法有啥区别。因此，是否有必要加一个section，关于分布式机器学习优化方法？
3. 运筹学，线性规划，非线性规划里面处理的基本东西要在你的脑海中有一个基本的印象。
4. 有必要加一节对偶。Done
5. 加一节“随机优化算法与鞅”²，并证明对数化误差是一个下鞅。随机优化方法很重要，我们需要从随机过程的角度去分析这些算法的性质，尤其是收敛行为，因此很有必要加这么一节。
6. 一些新的二阶优化方法，比如“Kronecker 系数近似曲率(K-FAC)方法”，以及非近似Hessian方法以及非Hessian方法，都要知道他们的原理与优缺点(实用场景)。
7. 思考结合Trust-Region方法与 δ -局部磨光算法来求解非凸优化问题，或者研究等级优化算法。
8. 稀疏建模理论，怎么调和稀疏建模结果与采样定理之间的矛盾。在什么情况下稀疏建模理论可以违背采样定理³？

1. 刘铁岩《分布式机器学习--算法, 理论与实战》[←](#)

2. Jean Walrand《EECS应用概率论》P194 [←](#)

3. 《稀疏建模理论, 算法及其应用》Irina Rish, Genady Ya. Grabarnik [←](#)

无约束最优化问题

无约束优化问题相对于有约束优化问题而言是简单的。在处理这些问题时，主要要考虑的是算法的收敛速度(线性收敛，二次收敛等等)，算法的时间复杂度与空间复杂度(也就是考虑内存问题)，算法的稳定性(收敛点的特性，鞍点，好的局部极小点，坏的局部极小点还是全局最小点)。Hessian矩阵的奇异性决定了算法的收敛速度。

要思考的一个问题是为啥不同的算法收敛到真正解的精度不同，这是由算法的什么特征导致的？

要思考的另外一个问题是，各算法的使用场景是什么？怎么根据不同场景选择不同的算法？有什么标准吗？怎么解决现实的非教科书的问题？需要增加次梯度法，随机坐标下降法。

按照线性搜索方法与信赖域方法分类。

Linear Search Methods

线性搜索与Armijo准则

符号约定：

$g_k : \nabla f(x_k)$, 即目标函数关于k次迭代值 x_k 的导数

$G_k : G(x_k) = \nabla^2 f(x_k)$, 即Hessian矩阵

d_k : 第k次迭代的优化方向, 在最速下降算法中, 有 $d_k = -g_k$

α_k : 第k次迭代的步长因子, 有 $x_{k+1} = x_k + \alpha_k d_k$

在精确线性搜索中, 步长因子 α_k 由下面的因子确定:

$$\alpha_k = \operatorname{argmin}_\alpha f(x_k + \alpha d_k)$$

而对于非精确线性搜索, 选取的 α_k 只要使得目标函数f得到可接受的下降量, 即:

$$\Delta f(x_k) = f(x_k) - f(x_k + \alpha_k d_k)$$

Armijo 准则用于非精确线性搜索中步长因子 α 的确定, 内容如下:

Armijo 准则:

已知当前位置 x_k 和优化方向 d_k , 参数 $\beta \in (0, 1), \delta \in (0, 0.5)$. 令步长因子 $\alpha_k = \beta^{m_k}$, 其中 m_k 为满足下列不等式的最小非负整数m:

$$f(x_k + \beta^m d_k) \leq f(x_k) + \delta \beta^m g_k^T d_k$$

由此确定下一个位置 $x_{k+1} = x_k + \alpha_k d_k$

对于梯度上升, 上面的方程变成:

$$f(x_k - \beta^m d_k) \geq f(x_k) - \delta \beta^m g_k^T d_k$$

由此确定下一个位置 $x_{k+1} = x_k - \alpha_k d_k$

最速下降法

the steepest descent algorithm proceeds as follows: at each step, starting from the point $x^{(k)}$, we conduct a line search in the direction $-\nabla f(x^{(k)})$, until a minimizer, $x^{(k+1)}$, is found.

$$\alpha_k = \operatorname{argmin}_{\alpha \geq 0} f(x^{(k)} - \alpha \nabla f(x^{(k)}))$$

Proposition 8.1: if $x^{(k)}$ is the steepest descent sequence for $f: R^n \rightarrow R$, then for each k the vector $x^{(k+1)} - x^{(k)}$ is orthogonal to the vector $x^{(k+2)} - x^{(k+1)}$

Proposition 8.2: if $x^{(k)}$ is the steepest descent sequence for $f: \mathbb{R}^n \rightarrow \mathbb{R}$ and if $\nabla f(x^{(k)}) \neq 0$, then

$$f(x^{(k+1)}) < f(x^{(k)})$$

Stopping criterion:

$$\frac{|f(x^{(k+1)}) - f(x^{(k)})|}{\|f(x^{(k)})\|} < \epsilon$$

Example: Quadratic function of the form:

$$f(x) = \frac{1}{2}x^T Qx - b^T x$$

Gradient: $g^{(k)} = \nabla f(x^{(k)}) = Qx - b$

so

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

where:

$$\begin{aligned} \alpha_k &= \operatorname{argmin}_{\alpha \geq 0} f(x^{(k)} - \alpha g^{(k)}) \\ &= \operatorname{argmin}_{\alpha \geq 0} \left(\frac{1}{2}(x^{(k)} - \alpha g^{(k)})^T Q(x^{(k)} - \alpha g^{(k)}) - (x^{(k)} - \alpha g^{(k)})^T b \right) \end{aligned}$$

Hence:

$$\alpha_k = \frac{g^{(k)T} g^{(k)}}{g^{(k)T} Q g^{(k)}}$$

Covergence properties:

Define:

$$V(x) = f(x) + \frac{1}{2}x^{*T} Qx^* = \frac{1}{2}(x - x^*)^T Q(x - x^*)$$

With: $x^* = Q^{-1}b$

Lemma 8.1 The iterative algorithm

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

with $g^{(k)} = Qx^{(k)} - b$ satisfies

$$V(x^{(k+1)}) = (1 - \gamma_k)V(x^{(k)}),$$

where, if $g^{(k)} = 0$ then $\gamma_k = 1$, and if $g^{(k)} \neq 0$ then:

$$\gamma_k = \alpha_k \frac{g^{(k)T} Q g^{(k)}}{g^{(k)T} Q^{-1} g^{(k)}} \left(2 \frac{g^{(k)T} g^{(k)}}{g^{(k)T} Q g^{(k)}} - \alpha_k \right)$$

Submit α_k into γ_k , then

$$\gamma_k = \frac{(g^{(k)T} g^{(k)})^2}{(g^{(k)T} Q g^{(k)}) (g^{(k)T} Q^{-1} g^{(k)})}$$

Theorem 8.1 Let $x^{(k)}$ be the sequence resulting from a gradient algorithm

$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$. Let γ_k be as defined in Lemma 8.1, and suppose that $\gamma_k > 0$ for all k ,

then $x^{(k)}$ converges to x^* for any initial condition $x^{(0)}$ if and only if:

$$\sum_{k=0}^{\infty} \gamma_k = \infty$$

Theorem 8.2 In the steepest descent algorithm, we have

$$x^{(k)} \rightarrow x^* \text{ for any } x^{(0)}$$

Theorem 8.3 For the fixed step size gradient algorithm, $x^{(k)} \rightarrow x^*$ for any $x^{(0)}$

$$\text{if and only if } 0 < \alpha < \frac{2}{\lambda_{\max}(Q)}$$

Convergence Rate:

Theorem 8.4 In the method of steepest descent applied to the quadratic function, at every step k, we have :

$$V(x^{(k+1)}) \leq \left(\frac{\lambda_{\max}(Q) - \lambda_{\min}(Q)}{\lambda_{\max}(Q)}\right) V(x^{(k)}).$$

Let:

$$r = \frac{\lambda_{\max}(Q)}{\lambda_{\min}(Q)} = ||Q|| ||Q^{-1}|| \text{ the so-called condition number of Q.}$$

Then, it follows from Theorem 8.4 that $V(x^{(k+1)}) \leq (1 - \frac{1}{r})V(x^{(k)})$. We refer to $1 - \frac{1}{r}$ as the

convergence ratio. If $r = 1$, then $\lambda_{\max}(Q) = \lambda_{\min}(Q)$, corresponding to circular contours of f. You can use the convergence ratio r to judge the speed of convergence.

Definition 8.1 Given a sequence $x^{(k)}$ that converges to x^* , that is, $\lim_{k \rightarrow \infty} \|x^{(k)} - x^*\| = 0$, we

say that the order of convergence is o, where $p \in R$, if $0 < \lim_{k \rightarrow \infty} \frac{x^{(k+1)} - x^*}{\|x^{(k)} - x^*\|^p} < \infty$,

if for all $p > 0$, $\lim_{k \rightarrow \infty} \frac{x^{(k+1)} - x^*}{\|x^{(k)} - x^*\|^p} = 0$,

then we say that the order of convergence is ∞ .

Lemma 8.3. In the steepest descent algorithm, if $g(k) \neq 0$ for all k, then $\gamma_k = 1$ if and only if $g(k)$ is an eigenvector of Q.

Theorem 8.6 Let $x^{(k)}$ be the sequence resulting from a gradient algorithm applied to a function f.

Then, the order of convergence of $x^{(k)}$ is 1 in the worst case, that is, there exist a function f and an initial $x^{(0)}$ such that the order of convergence of $x^{(k)}$ is equal 1.

Newton's Method

$$f(x) = f(x^{(k)}) + (x - x^{(k)})^T g^{(k)} + \frac{1}{2} (x - x^{(k)})^T F(x^{(k)}) (x - x^{(k)})$$

Theorem 9.1 Suppose that $f \in C^3$, and $x^* \in R^n$ is a point such that $\nabla f(x^*) = 0$ and $F(x^*)$ is invertible. Then, for all $x^{(0)}$ sufficiently close to x^* , Newton's method is well defined for all k, and converges to x^* with order of convergence at least 2.

As stated in the above theorem, Newton's method has superlinear convergence properties if the starting point is near the solution. However, the method is not guaranteed to converge to the solution if we start away from it (in fact, it may not even be well defined because the Hessian may be singular). In particular, the method may not be a descent method; that is, it is possible that $f(x^{(k+1)}) > f(x^{(k)})$. Fortunately, it is possible to modify the algorithm such that descent property holds. To see this, we need the following result.

Theorem 9.2 Let $x^{(k)}$ be the sequence generated by Newton's method for minimizing a given objective function $f(x)$. If the Hessian $F(x^{(k)}) > 0$ and $g^{(k)} = \nabla f(x^{(k)}) \neq 0$, then the direction

$d^{(k)} = -F(x^{(k)})^{-1}g^{(k)} = x^{(k+1)} - x^{(k)}$ from $x^{(k)}$ to $x^{(k+1)}$ is a descent direction for f in the sense that there exists an $\alpha > 0$ such that for all $a \in (a, \alpha)$,

$$f(x^{(k)} + \alpha d^{(k)}) < f(x^{(k)}).$$

The above theorem motivates the following modification of Newton's method:

$$x^{(k+1)} = x^{(k)} - \alpha_k F(x^{(k)})^{-1} g^{(k)} \text{ where}$$

$$\alpha_k = \operatorname{argmin}_{\alpha > 0} f(x^{(k)} - \alpha_k F(x^{(k)})^{-1} g^{(k)})$$

A drawback of Newton's method is that evaluation of $F(x^{(k)})$ for large n can be computationally expensive. Furthermore, we have to solve the set of n linear equations $F(x^{(k)})d^{(k)} = -g^{(k)}$. In the Chapters 10 and 11, we discuss methods that alleviate this difficulty.

Levenberg-Marquardt Modification:

If the Hessian matrix $F(x^{(k)})$ is not positive definite, then the search direction

$d^{(k)} = -F(x^{(k)})^{-1}g^{(k)}$ may not point in a descent direction. A simple technique to ensure that the search direction is a descent direction is to introduce the so-called Levenberg-Marquardt Modification to Newton's algorithm:

$$x^{(k+1)} = x^{(k)} - \alpha_k F(x^{(k)} + u_k I)^{-1} g^{(k)}$$

where $u_k \geq 0$

Newton's methods for nonlinear least-squares

Consider the following problem:

$$\text{minimize } \sum_{i=1}^m (r_i(x))^2$$

where $r_i : R^n \rightarrow R$, $i = 1, \dots, m$, are given functions. This particular problem is called a nonlinear least-squares problem

$$F_{jk} = 2 \sum_i \left(\frac{\partial r_i}{\partial x_j} \frac{\partial r_i}{\partial x_k} + r_i \frac{\partial^2 r_i}{\partial x_j \partial x_k} \right),$$

$$\text{Let } s(x) = r_i(x) \frac{\partial^2 r_i}{\partial x_j \partial x_k}(x)$$

So the Hessian matrix as

$$F(x) = 2(J(x)^T J(x) + S(x)).$$

Therefore, Newton's method applied to the nonlinear least-squares problems is given by

$$x^{(k+1)} = x^{(k)} - (J(x)^T J(x) + S(x))^{-1} J(x)^T r(x).$$

In some case, $s(x)$ can be ignored because its components are negligibly small. In this case, the above Newton's algorithm reduces to what is commonly called the Gauss-Newton method:

$$x^{(k+1)} = x^{(k)} - (J(x)^T J(x))^{-1} J(x)^T r(x).$$

A potential problem with the Gauss-Newton method is that the matrix $J(x)^T J(x)$ may not be positive definite. As described before, this problem can be overcome using a Levenberg-Marquardt modification:

$$x^{(k+1)} = x^{(k)} - (J(x)^T J(x) + u_k I)^{-1} J(x)^T r(x).$$

Conjugate Direction Methods

The conjugate direction methods typically perform better than the method of steepest descent, but not as well as Newton's method. As we saw from the method of steepest descent and Newton's method, the crucial factor in the efficiency of an iterative search method is the direction of the search at each iteration.

Definition 10.1 Let Q be a real symmetric $n \times n$ matrix. The directions $d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(m)}$ are Q -conjugate if, for all $i \neq j$, we have $d(i)^T Q d(j) = 0$.

Lemma 10.1 Let Q be a symmetric positive definite $n \times n$ matrix. If the directions

$d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(m)} \in R^n$, $k \leq n - 1$, are nonzero and Q -conjugate, then they are linearly independent.

Basic conjugate Direction Algorithm. Given a start $x^{(0)}$, and Q -conjugate direction

$$d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(m)}, \text{ for } k \geq 0,$$

$$g^{(k)} = \nabla f(x^{(k)}) = Qx^{(k)} - b,$$

$$\alpha_k = -\frac{g^{(k)T} d^{(k)}}{d^{(k)T} Q d^{(k)}}$$

$$x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$$

Theorem 10.1 For any starting point $x^{(0)}$, the basic conjugate direction algorithm converges to the unique x^* (that solves $Qx = b$) in n steps; that is, $x^{(n)} = x^*$

Lemma 10.2 In the conjugate direction algorithm,

$g^{(k+1)T}d^{(i)} = 0$. For all k , $0 \leq k \leq n - 1$, and $0 \leq i \leq k$.

The conjugate gradient algorithm is summarized below.

1. Set $k := 0$; select the initial point $x^{(0)}$
2. $g^{(0)} = \nabla f(x^{(0)})$. If $g^{(0)} = 0$, stop, else set $d^{(0)} = -g^{(0)}$.
3. $\alpha_k = -\frac{g^{(k)T}d^{(k)}}{d^{(k)T}Qd^{(k)}}$
4. $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$
5. $g^{(k+1)} = \nabla f(x^{k+1})$. If $g^{(k+1)} = 0$, stop.
6. $\beta_k = -\frac{g^{(k+1)T}Qd^{(k)}}{d^{(k)T}Qd^{(k)}}$
7. $d^{(k+1)} = -g^{(k+1)} + \beta_k d^{(k)}$
8. Set $k := k+1$; go to step 3.

Proposition 10.1 In the conjugate gradient algorithm, the directions $d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(n-1)}$ are Q-conjugate.

梯度下降法

到时加上投影梯度法与近点梯度法。

[最速下降法, 牛顿法, LBFGS](#)

梯度下降法是最早最简单，也是最为常用的最优化方法。梯度下降法实现简单，当目标函数是凸函数时，梯度下降法的解是全局解。一般情况下，其解不保证是全局最优解，梯度下降法的速度也未必是最快的。梯度下降法的优化思想是用当前位置负梯度方向作为搜索方向，因为该方向为当前位置的最快下降方向，所以也被称为是“最速下降法”。最速下降法越接近目标值，梯度趋于0，所以步长越小，前进越慢。把损失函数展开到一阶。

$$f(x + \alpha \mathbf{d}) = f(x_0) + \alpha f'(x_0) \mathbf{d} + O(\alpha^2)$$

$$x_{t+1} = x_t - \alpha f'(x_t),$$

梯度下降法的搜索迭代示意图如下图所示：



梯度下降法的缺点：

(1) 越靠近极小值的地方收敛速度越慢，如下图所示



(c) 2006 P. A. Simionescu

从上图可以看出，梯度下降法在接近最优解的区域收敛速度明显变慢，利用梯度下降法求解需要很多次的迭代。

在机器学习中，基于基本的梯度下降法发展了两种梯度下降方法，分别为随机梯度下降法和批量梯度下降法。

比如对一个线性回归(Linear Logistics)模型，假设下面的 $h(x)$ 是要拟合的函数， $J(\theta)$ 为损失函数， θ 是参数，要迭代求解的值， θ 求解出来了那最终要拟合的函数 $h(\theta)$ 就出来了。其中 m 是训练集的样本个数， n 是特征的个数。

$$h(\theta) = \sum_{j=0}^n \theta_j x_j$$

$$J(\theta) = \frac{1}{2m} \sum_{i=0}^m (y^i - h_\theta(x^i))^2$$

//n是特征的个数，m的训练样本的数目

1) 批量梯度下降法 (Batch Gradient Descent, BDG)

(1) 将 $J(\theta)$ 对 θ 求偏导，得到每个 θ 对应的梯度：

$$\frac{\partial J(\theta)}{\partial \theta_j} = -\frac{1}{m} \sum_{i=1}^m (y^i - h_\theta(x^i)) x_j^i$$

(2) 由于要最小化风险函数，所以按照每个参数 θ 的负梯度方向来更新 θ

$$\theta'_j = \theta_j - \frac{\partial J(\theta)}{\partial \theta_j} \theta_j + \frac{1}{m} \sum_{i=1}^m (y^i - h_\theta(x^i)) x_j^i$$

(3) 从上面公式可以注意到，它得到的是一个全局最优解，但是每迭代一步，都要用到训练集所有的数据，如果 m 很大，那么可想而知这种方法的迭代速度会相当的慢。所以，这就引入了另外一种方法——随机梯度下降。

一个实验结果，说明随机梯度下降法能收敛到最终结果：

对于批量梯度下降法，样本个数 m ， x 为 n 维向量，一次迭代需要把 m 个样本全部带入计算，迭代一次计算量为 $m * n^2$ 。

2) 随机梯度下降 (Stochastic Gradient Descent, SGD)

(1) 上面的风险函数可以写成如下这种形式，损失函数对应的是训练集中每个样本的粒度，而上面批量梯度下降对应的是所有的训练样本

$$\begin{aligned} J(\theta) &= \frac{1}{2m} \sum_{i=0}^m (y^i - h_\theta(x^i))^2 \\ &= \frac{1}{m} \sum_{i=1}^m \text{cost}(\theta, (x^i, y^i)) \\ L(\theta, (x^i, y^i)) &= \frac{1}{2} (y^i - h_\theta(x^i))^2 \end{aligned}$$

(2) 每个样本的损失函数，对 θ 求偏导得到对应的梯度，来更新 θ

$$\theta'_j = \theta_j + (y^i - h_\theta(x^i)) x_j^i$$

(3) 随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况（例如几十万），那么可能只用其中几万条或者几千条的样本，就已经将 θ 迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，一次迭代不可能最优，如果迭代 10 次的话就需要遍历训练样本 10 次。但是，SGD 伴随的一个问题是噪音较 BGD 要多，使得 SGD 并不是每次迭代都向着整体最优化方向。

随机梯度下降每次迭代只使用一个样本，迭代一次计算量为 n^2 ，当样本个数 m 很大的时候，随机梯度下降迭代一次的速度要远高于批量梯度下降方法。两者的关系可以这样理解：随机梯度下降方法以损失很小的一部分精确度和增加一定数量的迭代次数为代价，换取了总体的优化效率的提升。增加的迭代次数远远小于样本的数量。

比较批量梯度下降法与随机梯度下降法：

用 $y = 2 * x1 + x2 - 1 + rand(0, 1)$ 生成1000组数, 用这一组数据来反求生成函数中的系数

$(2, 1, -1)^T$.

迭代停止条件就是, 训练得到的相邻两次参数的差的范数小于0.000001(严格来说, 停止条件是损

失函数一介导数等于0, 也就是 $L'(\theta) = -\frac{1}{m}X^T(X\theta - Y) = 0$.

1) 批量梯度下降法: 1000个样本, 设置的learn rate = 0.0001, 一次迭代所有1000个样本, 最终经过38000次迭代收敛到可以接受的标准。如果设置learn rate 是较大的值, 比如为1, 则发现结果马上发散了, 因此批量梯度下降法对learn rate比较敏感, 而下面的随机梯度下降法就不会出现这个问题, 因为每次只对一个样本进行处理, 即使这个样本对参数的梯度很大, 但是下一个样本又可以马上把参数拉回来。

```
C:\Windows\system32\cmd.exe
Grad:4.34235e-05 1.78119e-05 -0.00100021
Paras: 1.99971 0.999882 -0.993389 para diff norm: 1.00131e-06 iter num: 37888
Grad: 4.3417e-05 1.78092e-05 -0.00100005
Paras: 1.99971 0.999882 -0.99339 para diff norm: 1.00115e-06 iter num: 37889
Grad: 4.34104e-05 1.78065e-05 -0.000999903
Paras: 1.99971 0.999882 -0.993391 para diff norm: 1.001e-06 iter num: 37890
Grad: 4.34038e-05 1.78038e-05 -0.000999751
Paras: 1.99971 0.999882 -0.993392 para diff norm: 1.00085e-06 iter num: 37891
Grad:4.33973e-05 1.78011e-05 -0.0009996
Paras: 1.99971 0.999882 -0.993393 para diff norm: 1.0007e-06 iter num: 37892
Grad: 4.33907e-05 1.77984e-05 -0.000999449
Paras: 1.99971 0.999882 -0.993394 para diff norm: 1.00055e-06 iter num: 37893
Grad: 4.33841e-05 1.77957e-05 -0.000999298
Paras: 1.99971 0.999882 -0.993395 para diff norm: 1.0004e-06 iter num: 37894
Grad: 4.33776e-05 1.7793e-05 -0.000999147
Paras: 1.99971 0.999882 -0.993396 para diff norm: 1.00025e-06 iter num: 37895
Grad: 4.3371e-05 1.77903e-05 -0.000998996
Paras: 1.99971 0.999882 -0.993397 para diff norm: 1.00009e-06 iter num: 37896
Grad: 4.33644e-05 1.77876e-05 -0.000998844
Paras: 1.99971 0.999882 -0.993398 para diff norm: 9.99944e-07 iter num: 37897
pass
solution:
1.99971
0.999882
-0.993398
Golden Solution:
2
1
-1
Press any key to continue . . .
```

2) 随机梯度下降法:

1000个样本, 设置的learn rate = 1, 一次迭代处理一个样本, 最终经过25000次迭代收敛到可以接受的标准。

```

C:\Windows\system32\cmd.exe
Paras: 1.99795 0.998981 -0.951616 para diff norm: 0.000446962 iter num: 24714
Paras: 1.99818 0.999232 -0.951604 para diff norm: 0.000338804 iter num: 24715
Paras: 1.99816 0.99922 -0.951605 para diff norm: 1.82845e-05 iter num: 24716
Paras: 1.99805 0.998911 -0.951617 para diff norm: 0.000328846 iter num: 24717
Paras: 1.99804 0.998721 -0.951623 para diff norm: 0.000190122 iter num: 24718
Paras: 1.9981 0.999161 -0.951611 para diff norm: 0.000444339 iter num: 24719
Paras: 1.99816 0.999501 -0.951604 para diff norm: 0.000344163 iter num: 24720
Paras: 1.99813 0.998416 -0.951627 para diff norm: 0.00108582 iter num: 24721
Paras: 1.99815 0.998517 -0.951624 para diff norm: 0.000102945 iter num: 24722
Paras: 1.99834 0.999645 -0.951597 para diff norm: 0.00114423 iter num: 24723
Paras: 1.99829 0.999464 -0.951642 para diff norm: 0.000192287 iter num: 24724
Paras: 1.99833 0.99955 -0.95164 para diff norm: 9.18398e-05 iter num: 24725
Paras: 1.99831 0.99951 -0.951641 para diff norm: 4.20315e-05 iter num: 24726
Paras: 1.99805 0.999223 -0.951665 para diff norm: 0.000389985 iter num: 24727
Paras: 1.99804 0.999007 -0.951669 para diff norm: 0.000216928 iter num: 24728
Paras: 1.99784 0.998829 -0.951687 para diff norm: 0.000266407 iter num: 24729
Paras: 1.99765 0.998724 -0.951707 para diff norm: 0.000220029 iter num: 24730
Paras: 1.99779 0.998928 -0.951696 para diff norm: 0.000248953 iter num: 24731
Paras: 1.99774 0.998884 -0.9517 para diff norm: 6.50401e-05 iter num: 24732
Paras: 1.99774 0.998884 -0.9517 para diff norm: 5.54896e-07 iter num: 24733
pass
solution:
1.99774
0.998884
-0.9517
Golden Solution:
2
1
-1
Press any key to continue . . .

```

3)

牛顿法

```

C:\Windows\system32\cmd.exe
Paras: 1.99954 1 -0.99909 para diff norm: 1.0188e-06 iter num: 7691
Paras: 1.99955 1 -0.999091 para diff norm: 1.01778e-06 iter num: 7692
Paras: 1.99955 1 -0.999091 para diff norm: 1.01676e-06 iter num: 7693
Paras: 1.99955 1 -0.999092 para diff norm: 1.01575e-06 iter num: 7694
Paras: 1.99955 1 -0.999093 para diff norm: 1.01473e-06 iter num: 7695
Paras: 1.99955 1 -0.999094 para diff norm: 1.01372e-06 iter num: 7696
Paras: 1.99955 1 -0.999095 para diff norm: 1.0127e-06 iter num: 7697
Paras: 1.99955 1 -0.999096 para diff norm: 1.01169e-06 iter num: 7698
Paras: 1.99955 1 -0.999097 para diff norm: 1.01068e-06 iter num: 7699
Paras: 1.99955 1 -0.999098 para diff norm: 1.00967e-06 iter num: 7700
Paras: 1.99955 1 -0.999099 para diff norm: 1.00866e-06 iter num: 7701
Paras: 1.99955 1 -0.999101 para diff norm: 1.00765e-06 iter num: 7702
Paras: 1.99955 1 -0.999101 para diff norm: 1.00664e-06 iter num: 7703
Paras: 1.99955 1 -0.999101 para diff norm: 1.00564e-06 iter num: 7704
Paras: 1.99955 1 -0.999102 para diff norm: 1.00463e-06 iter num: 7705
Paras: 1.99955 1 -0.999103 para diff norm: 1.00363e-06 iter num: 7706
Paras: 1.99955 1 -0.999104 para diff norm: 1.00262e-06 iter num: 7707
Paras: 1.99955 1 -0.999105 para diff norm: 1.00162e-06 iter num: 7708
Paras: 1.99955 1 -0.999106 para diff norm: 1.00062e-06 iter num: 7709
Paras: 1.99955 1 -0.999107 para diff norm: 9.99617e-07 iter num: 7710
pass
solution:
1.99955
1
-0.999107
Golden Solution:
2
1
-1
Press any key to continue . . .

```

对批量梯度下降法和随机梯度下降法的总结：

批量梯度下降---最小化所有训练样本的损失函数，使得最终求解的是全局的最优解，即求解的参数是使得风险函数最小，但是对于大规模样本问题效率低下。

随机梯度下降---最小化每条样本的损失函数，虽然不是每次迭代得到的损失函数都向着全局最优方向，但是大的整体的方向是向全局最优解的，最终的结果往往是在全局最优解附近，适用于大规模训练样本情况。

从实验数据来看，批量梯度下降法收敛到的是全局最优值，而随机梯度下降法收敛到最优值附件的地方。随机梯度下降法的好处是收敛远快于批量梯度下降法。

线性方程组求解转为成一个最小二乘问题

$$\mathbf{Y} = \mathbf{X} * \lambda$$

问题可以转化成寻找一组 λ 使得:

$$L(\lambda) = (\mathbf{X} * \lambda - \mathbf{Y})^T (\mathbf{X} * \lambda - \mathbf{Y}) \text{ 极小}$$

利用 $L(\lambda)$ 对向量 λ 求导 可以得到¹:

$$\frac{\partial L(\lambda)}{\partial \lambda} = 2\mathbf{X}^T (\mathbf{X}\lambda - \mathbf{Y}) = 0$$

我们也可以得到:

$$\frac{\partial^2 L(\lambda)}{\partial \lambda^2} = 2\mathbf{X}^T \mathbf{X}$$

$$\mathbf{X}(n+1) = \mathbf{X}(n) - f'(\mathbf{X}(n)) / f''(\mathbf{X}(n))$$

```
MatrixXd Iteration::BathGradDescent(MatrixXd paras, int iterator_num)
{
    int rows = m_input_data.rows();
    int cols = m_input_data.cols();
    MatrixXd init_paras = paras;
    if (m_input_data.cols() == paras.rows())
    {

        MatrixXd grad = (1.0 / rows)*m_input_data.transpose()*(m_output_data - m_input_data*init_paras);
        paras += m_learn_rate*grad;
    }
    else
    {
        //error
    }
    return paras;
}

MatrixXd Iteration::StocGradDescent(MatrixXd paras, int index)
{
    int rows = m_input_data.rows();
    int cols = m_input_data.cols();
    MatrixXd input_data_i(1,cols);
    MatrixXd output_data_i(1, 1);
    output_data_i(0, 0) = m_output_data(index,0);

    for (int i = 0; i < cols; i++)
    {
        input_data_i(0, i) = m_input_data(index, i);
    }

    if (m_input_data.cols() == paras.rows())
    {
        MatrixXd grad = (1.0 / rows) *input_data_i.transpose()*(input_data_i*paras - output_data_i);
        paras = paras - grad;
    }
}
```

```
    }
    else
    {
        //error
    }
    return paras;
}
```

牛顿法

牛顿法求零点

牛顿法是一种在实数域和复数域上近似求解方程的方法。方法使用函数 $f(x)$ 的泰勒级数的前面几项来寻找方程 $f(x) = 0$ 的根。¹牛顿法最大的特点就在于它的收敛速度很快。

由于牛顿法是基于当前位置的切线来确定下一次的位置，所以牛顿法又被很形象地称为是“切线法”。

步骤：

首先，选择一个接近函数 $f(x)$ 零点的 x_0 ，计算相应的 $f(x_0)$ 和切线斜率 $f'(x_0)$ 。然后计算穿过点 $(x_0, f(x_0))$ 并且斜率为 $f'(x_0)$ 的直线和 x 轴的交点的 x 轴坐标，也就是如下方程：

$$x * f'(x_0) + f(x_0) - x_0 * f'(x_0) = 0$$

我们将新求得的点的 x 坐标命名为 x_1 ，通常 x_1 会比 x_0 更接近方程 $f(x) = 0$ 的解。因此我们现在可以利用 x_1 开始下一轮迭代。迭代公式可化简为如下所示：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

已经证明，如果 $f'(x)$ 是连续的，并且待求的零点 x 是孤立的，那么在零点 x 周围存在一个区域，只要初始值 x_0 位于这个邻近区域内，那么牛顿法必定收敛。并且，如果 $f'(x)$ 不为0，那么牛顿法将具

有平方收敛的性能。粗略的说，这意味着每迭代一次，牛顿法结果的有效数字将增加一倍。下图为一个牛顿法执行过程的例子。下图为一个牛顿法执行过程的例子。



牛顿法求极值

对一个函数进行泰勒展开，保留到二阶。

$$f(x_n + \Delta x) \approx f(x_n) + f'(x_n)\Delta x + \frac{1}{2}f''(x_n)\Delta x^2$$

为得到最佳的 Δx ，我们需要对上式求导，并令其为0。

因此可以得到：

$$\Delta x = \frac{f'(x_n)}{f''(x_n)}$$

因此：

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

如果X是向量，则可以写成向量形式H是Hessian矩阵：

$$\mathbf{X}_{n+1} = \mathbf{X}_n - \frac{\nabla f(\mathbf{x}_n)}{H(\mathbf{x}_n)}$$

关于牛顿法和梯度下降法的效率对比：

从本质上讲，牛顿法是二阶收敛，梯度下降是一阶收敛，所以牛顿法就更快。如果更通俗地说的话，比如你想找一条最短的路径走到一个盆地的最底部，梯度下降法每次只从你当前所处位置选一个坡度最大的方向走一步，牛顿法在选择方向时，不仅会考虑坡度是否够大，还会考虑你走

了一步之后，坡度是否会变得更大。所以，可以说牛顿法比梯度下降法看得更远一点，能更快地走到最底部。（牛顿法目光更加长远，所以少走弯路；相对而言，梯度下降法只考虑了局部的最优，没有全局思想。）

根据wiki上的解释，从几何上说，牛顿法就是用一个二次曲面去拟合你当前所处位置的局部曲面，而梯度下降法是用一个平面去拟合当前的局部曲面，通常情况下，二次曲面的拟合会比平面更好，所以牛顿法选择的下降路径会更符合真实的最优下降路径。



注：红色的牛顿法的迭代路径，绿色的是梯度下降法的迭代路径。

牛顿法的优缺点总结：

优点：二阶收敛，收敛速度快；

缺点：牛顿法是一种迭代算法，每一步都要求求解目标函数的**Hessian**矩阵的逆矩阵，计算比较复杂。

2) 拟牛顿法(Quasi-Newton Methods)

拟牛顿法是求解非线性优化问题最有效的方法之一，于20世纪50年代由美国Argonne国家实验室的物理学家W.C.Davidon所提出来。Davidon设计的这种算法在当时看来是非线性优化领域最具创造性的发明之一。不久R. Fletcher和M. J. D. Powell证实了这种新的算法远比其他方法快速和可靠，使得非线性优化这门学科在一夜之间突飞猛进。

拟牛顿法的本质思想是改善牛顿法每次都要求解复杂的Hessian矩阵的逆矩阵的缺陷，它使用正定矩阵来近似Hessian矩阵的逆，从而简化了运算的复杂度。拟牛顿法和最速下降法一样只要求每一步迭代时知道目标函数的梯度。通过测量梯度的变化，构造一个目标函数的模型使之足以产生超线性收敛性。这类方法大大优于最速下降法，尤其对于困难的问题。另外，因为拟牛顿法不需要二阶导数的信息，所以有时比牛顿法更为有效。如今，优化软件中包含了大量的拟牛顿算法用来解决无约束，约束，和大规模的优化问题。

具体步骤：

拟牛顿法的基本思想如下。首先构造目标函数在当前迭代 x_k 的二次模型：

$$m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{p^T B_k p}{2}$$

$$p_k = -B_k^{-1} \nabla f(x_k)$$

这里 B_k 是一个对称正定矩阵，于是我们取这个二次模型的最优解作为搜索方向，并且得到新的迭代点：

$$x_{k+1} = x_k + \alpha_k p_k$$

其中我们要求步长 α_k 满足 Wolfe 条件。这样的迭代与牛顿法类似，区别就在于用近似的 Hesse 矩阵 B_k 代替真实的 Hesse 矩阵。所以拟牛顿法最关键的地方就是每一步迭代中矩阵 B_k 的更新。现在假设得到一个新的迭代 x_{k+1} ，并得到一个新的二次模型：

这个公式被称为割线方程。常用的拟牛顿法有 DFP 算法和 BFGS 算法。

[最优化方法:牛顿迭代法和拟牛顿迭代法](#)

拟牛顿法

拟牛顿法的历史

拟牛顿法是求解非线性优化问题最有效的方法之一，于20世纪50年代由美国Argonne国家实验室的物理学家W.C.Davidon所提出来。Davidon设计的这种算法在当时看来是非线性优化领域最具创造性的发明之一。不久R. Fletcher和M. J. D. Powell证实了这种新的算法远比其他方法快速和可靠，使得非线性优化这门学科在一夜之间突飞猛进。

拟牛顿法的本质思想是改善牛顿法每次需要求解复杂的Hessian矩阵的逆矩阵的缺陷，它使用正定矩阵来近似Hessian矩阵的逆，从而简化了运算的复杂度。拟牛顿法和最速下降法一样只要求每一步迭代时知道目标函数的梯度。通过测量梯度的变化，构造一个目标函数的模型使之足以产生超线性收敛性。这类方法大大优于最速下降法，尤其对于困难的问题。另外，因为拟牛顿法不需要二阶导数的信息，所以有时比牛顿法更为有效。如今，优化软件中包含了大量的拟牛顿算法用来解决无约束，约束，和大规模的优化问题。

具体步骤：

拟牛顿法的基本思想如下。首先构造目标函数在当前迭代 x_k 的二次模型：

$$m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{p^T B_k p}{2}$$

$$p_k = -B_k^{-1} \nabla f(x_k)$$

这里 B_k 是一个对称正定矩阵，于是我们取这个二次模型的最优解作为搜索方向，并且得到新的迭代点：

$$x_{k+1} = x_k + \alpha_k p_k$$

其中我们要求步长 α_k 满足Wolfe条件。这样的迭代与牛顿法类似，区别就在于用近似的Hesse矩阵 B_k 代替真实的Hesse矩阵。所以拟牛顿法最关键的地方就是每一步迭代中矩阵 B_k 的更新。现在假设得到一个新的迭代 x_{k+1} ，并得到一个新的二次模型：

这个公式被称为割线方程。常用的拟牛顿法有DFP算法和BFGS算法。

[最优化方法：牛顿迭代法和拟牛顿迭代法](#)

Quasi-Newton methods

BFGS

我们知道，在牛顿法中，我们需要求解二阶导数矩阵--Hessian阵，当变量很多时，求解Hessian阵势比较费时间的，Quasi-Newton法主要是在构造Hessian阵上下功夫，它是通过构造一个近似的Hessian阵，或者Hessian阵的逆，而不是解析求解或者利用差分法来求解这个Hessian阵。构造的Hessian阵通过迭代而改变。

比较出名的Quasi-Netwon方法有BFGS(以Charles George Broyden, Roger Fletcher, Donald Goldfarb and David Shanno命名)

在牛顿法中, k步搜寻步长与方向是 p_k , 满足下面方程

$$B_k p_k = -\nabla f(x_k)$$

B_k 就是近似的Hessian阵。下面我们讨论 B_k 如何变化,

我们要求 B_k 的更新满足quasi-Netwon条件

$$B_{k+1}(x_{k+1} - x_k) = \nabla f(x_{k+1}) - \nabla f(x_k)$$

这个条件就是简单的求 $f(x)$ 的二阶导数。

令:

$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k), \quad s_k = x_{k+1} - x_k, \text{ 因此 } B_{k+1} \text{ 满足 } B_{k+1} s_k = y_k$$

这就是割线方程(the secant equation), The curvature condition $s_k^T y_k > 0$ 需要满足。

k步的Hessian阵以如下方式更新,

$$B_{k+1} = B_k + U_k + V_k$$

为了保持 B_{k+1} 的正定性以及对称性。 B_{k+1} 可以取如下形式:

$$B_{k+1} = B_k + \alpha u u^T + \beta v v^T$$

选择 $u = y_k, v = B_k s_k$, 为了满足割线方程(the secant condition), 我们得到:

$$\alpha = \frac{1}{y_k^T s_k}$$

$$\beta = \frac{1}{s_k^T B_k s_k}$$

最终我们得到Hessian阵的更新方程:

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k^T}{s_k^T B_k s_k}$$

利用 Sherman–Morrison formula,

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}$$

其中A是可逆方阵, $1 + v^T A^{-1} u \neq 0$

可以方便的得到 B_{k+1} 的逆。

$$\begin{aligned} B_{k+1}^{-1} &= (I - \frac{s_k y_k^T}{y_k^T s_k}) B_k^{-1} (1 - \frac{y_k s_k^T}{y_k^T s_k}) + \frac{s_k s_k^T}{y_k^T s_k} \\ B_{k+1}^{-1} &= B_k^{-1} + \frac{(s_k^T y_k + y_k^T B_k^{-1} y_k)(s_k s_k^T)}{(s_k^T y_k)^2} - \frac{B_k^{-1} y_k s_k^T + s_k y_k^T B_k^{-1}}{s_k^T y_k} \end{aligned}$$

DFP

参考 DFP的矫正公式如下：

$$H_{k+1} = H_k + \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{s_k^T y_k}$$

当采用精确线搜索时，矩阵序列 H_k 的正定新条件 $s_k^T y_k > 0$ 可以被满足。但对于Armijo搜索准则来说，不能满足这一条件，需要做如下修正：

$$H_{k+1} = H_k - s_k^T y_k \leq 0$$

$$H_{k+1} = H_k + \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} - s_k^T y_k > 0$$

Broyden族算法

之前BFGS和DFP矫正都是由 y_k 和 $B_k s_k$ (或者 s_k 和 $H_k y_k$)组成的秩2矩阵。而Droyden族算法采用了BFGS和DFP校正公式的凸组合，如下：

$$\begin{aligned} H_{k+1}^\phi &= \phi_k H_{k+1}^{BFGS} + (1 - \phi_k) H_{k+1}^{DFP} \\ &= H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{s_k^T y_k} + \phi_k v_k v_k^T \end{aligned}$$

其中 $\phi_k \in [0, 1]$, v_k 由下式定义：

$$v_k = \sqrt{\frac{s_k}{y_k^T H_k y_k}} \left(\frac{s_k}{y_k^T s_k} - \frac{H_k y_k}{y_k^T H_k y_k} \right)$$

Trust-Region方法

Gauss-Netwon方法, L-M算法, Dog-Leg算法都是信赖域方法。下面主要讨论Trust-Region方法的原理。后面的分节讨论三种常见的Trust-Region方法。

Gauss-Netwon方法

对于函数 $f(x)$ 的小邻域展开

$$f(x + h) \simeq l(h) \equiv f(x) + J(x)h$$

$$F(x + h) \simeq L(h) \equiv \frac{1}{2}l(h)^T l(h) = \frac{1}{2}f^T f + h^T J^T f + \frac{1}{2}h^T J^T J h$$

我们需要选择步长 h_{gn} 来最小化 $L(h)$

$$h_{gn} = \operatorname{argmin}_h L(h)$$

$$(J^T J)h_{gn} = -J^T f$$

高斯-牛顿法

非线性回归中，损失函数可以表示成残差的形式：

$$L(x) = \sum_{i=1}^m r_i(x)^2$$

根据牛顿法：

$$x_{t+1} = x_t - H^{-1}g,$$

梯度表示为：

$$g_j = 2 \sum_i r_i \frac{\partial r_i}{\partial x_j}$$

对损失函数求二阶导，我们可以得到Hessian矩阵：

$$H_{jk} = 2 \sum_i \left(\frac{\partial r_i}{\partial x_j} \frac{\partial r_i}{\partial x_k} + r_i \frac{\partial^2 r_i}{\partial x_j \partial x_k} \right),$$

当残差 r_i 很小的时候，我们就可以去掉最后一项，因此

$$H_{jk} \approx 2 \sum_i J_{ij} J_{ik} \quad \text{With} \quad J_{ij} = \frac{\partial r_i}{\partial x_j}$$

这样Hessian就是半正定了。

因此参数迭代公式可以写成：

$$x_{t+1} = x_t - (J^T J)^{-1} J^T r,$$

另外一种解释

对于函数 $f(x)$ 的小邻域展开

$$f(x+h) \simeq l(h) \equiv f(x) + J(x)h$$

$$F(x+h) \simeq L(h) \equiv \frac{1}{2}l(h)^T l(h) = \frac{1}{2}f^T f + h^T J^T f + \frac{1}{2}h^T J^T J h$$

我们需要选择步长 h_{gn} 来最小化 $L(h)$

$$h_{gn} = \operatorname{argmin}_h L(h)$$

$$(J^T J)h_{gn} = -J^T f$$

L-M算法

方法用于求解非线性最小二乘问题，结合了梯度下降法和高斯-牛顿法。因此我们先简单介绍梯度下降法与Gauss-Netwon法。

梯度下降法

把损失函数展开到一阶。

$$f(x + \alpha \mathbf{d}) = f(x_0) + \alpha f'(x_0) \mathbf{d} + O(\alpha^2)$$

$$x_{t+1} = x_t - \alpha f'(x_t),$$

牛顿法

牛顿法 把cost function进行展开到二阶：

$$f(x_{t+1}) = f(x_t) + g(x_{t+1} - x_t) + \frac{1}{2}(x_{t+1} - x_t)^T H(x_{t+1} - x_t)$$

求导, $\frac{\partial f}{\partial x_t} = g + H(x_{t+1} - x_t)$, 让导数为0就有

$$x_{t+1} = x_t - H^{-1}g$$

要是H是正定的, 上面的就是凸函数, 也就一定有了最小值。可惜H不一定是正定的, 这就引导出了下面的方法

高斯-牛顿法

cost function可以表示成残差的形式：

$$L(x) = \sum_{i=1}^m r_i(x)^2$$

根据牛顿法：

$$x_{t+1} = x_t - H^{-1}g,$$

梯度表示为：

$$g_j = 2 \sum_i r_i \frac{\partial r_i}{\partial x_j}$$

对方程(4.9)求二阶导, 我们可以得到Hessian矩阵：

$$H_{jk} = 2 \sum_i \left(\frac{\partial r_i}{\partial x_j} \frac{\partial r_i}{\partial x_k} + r_i \frac{\partial^2 r_i}{\partial x_j \partial x_k} \right),$$

当残差 r_i 很小的时候, 我们就可以去掉最后一项, 因此

$$H_{jk} \approx 2 \sum_i J_{ij} J_{ik} \quad \text{With} \quad J_{ij} = \frac{\partial r_i}{\partial x_j}$$

这样Hessian就是半正定了。

因此参数迭代公式(4.10)可以写成：

$$x_{t+1} = x_t - (J^T J)^{-1} J^T r,$$

Levenberg-Marquardt

方法用于求解非线性最小二乘问题，结合了梯度下降法和高斯-牛顿法。

其主要改变是在Hessian阵中加入对角线项，加这一样可以保证Hessian阵是正定的。当然，这是一种L2正则化方式。

$$x_{t+1} = x_t - (H + \lambda I_n)^{-1} g,$$

$$x_{t+1} = x_t - (J^T J + \lambda I_n)^{-1} J^T r$$

L-M算法的不足点。

当 λ 很大时， $J^T J + \lambda I_n$ 根本没用，Marquardt为了让梯度小的方向移动的快一些，来防止小梯度方向的收敛，把中间的单位矩阵换成了 $diag(J^T J)$ ，因此迭代变成：

$$x_{t+1} = x_t - (J^T J + \lambda diag(J^T J))^{-1} J^T r$$

阻尼项(damping parameter) λ 的选择，Marquardt 推荐一个初始值 λ_0 与因子 $v > 1$ ，开始时，

$\lambda = \lambda_0$ ，然后计算cost functions，第二次计算 $\lambda = \lambda_0/v$ ，如果两者cost function都比初始点高，然后我们就增大阻尼项，通过乘以v，直到我们发现当 $\lambda = \lambda_0 v^k$ 时，cost function下降。

如果使用 $\lambda = \lambda_0/v$ 使得cost function下降，然后我们就把 λ_0/v 作为新的 λ

当 $\lambda = 0$ 时，就是高斯-牛顿法，当 λ 趋于无穷时，就是梯度下降法。如果使用 λ/v 没有使损失函数下降，使用 λ 导致损失函数下降，那么我们就继续使用 λ 做为阻尼项。

归一化的残差

如果我们假设数据中每一点的贡献是等权重，因此，我们有必要对每一项残差加一个权重，来归一化每一项残差。因此公式(4-9)可以变成，

cost function可以表示成残差的形式：

$$L(x) = \sum_{i=1}^m \lambda_i r_i(x)^2 = \sum_{i=1}^m \frac{1}{y_i^2} (y_i - f_i(x_i))^2$$

此时，Jacobia变成

$$H_{jk} \approx 2 \sum_i J_{ij} J_{ik} \quad \text{With} \quad J_{ij} = y_i \frac{\partial r_i}{\partial x_j}$$

$$J_S = \Sigma J, \Sigma_{ij} = \frac{1}{y_i} \delta_{ij}$$

L2正则化与Levenberg-Marquardt算法

我们在cost function加上L2正则项

$$L(\beta) = \sum_{i=1}^m [y_i - f(x_i, \beta)]^2 + \lambda \|\beta\|^2$$

可以表示成：

$$L(\beta) = L(\beta_0) + (\beta - \beta_0)^T \nabla_\beta L(\beta_0) + \frac{1}{2}(\beta - \beta_0)^T H(\beta - \beta_0) + \lambda \|\beta\|^2 + O(\beta^3)$$

求一阶导数：

$$L(\beta) = L(\beta_0) + (\beta - \beta_0)^T g + \frac{1}{2}(\beta - \beta_0)^T H(\beta - \beta_0) + \frac{1}{2}\lambda \|\beta\|^2 + O(\beta^3)$$

令其为0：

$$\frac{\partial L(\beta)}{\partial \beta} = g + H(\beta - \beta_0) + \lambda \beta = 0$$

就得到：

$$\beta = (H + \lambda I_n)^{-1} H \beta_0 - (H + \lambda I_n)^{-1} g, \text{ 算法流程如下：}$$

$$k := 0; x = x_0; v := 2; A := J(x)^T J(x); g := J(x)^T f(x); u := \tau * \max(a_{ii})$$

$$\text{found:}= (||g||_\infty \leq \epsilon_1)$$

while(not found) and $k < k_{max}$

k:=k+1,计算 h_{lm} 根据 $(A + uI)h_{lm} = -g$

if $\|h_{lm}\| \leq \epsilon(||x|| + \epsilon_2)$

 found := true

else:

$x_{new} := x + h_{lm}$

$\rho := (F(x) - F(x + h_{lm})) / (L(0) - L(h_{lm}))$

 if $\rho \geq 0$

$x := x_{new}; g := J(x)^T f(x)$

$A := J^T(x)J(x); g := J^T(x)f(x)$

 found:=(||g||_\infty \leq \epsilon_1)

$u := u * \max(1/2, 1 - (2\rho - 1)^3); v := 2$

 else:

$u := u * v; v = 2 * v$

```
def LM_Solver(self):
    init_p = 0.5 * np.ones((2, 1))
    init_p[0, 0] = 2.0
    init_p[1, 0] = 1.0
    epsilon_1 = 1e-15
    epsilon_2 = 1e-15
    epsilon_3 = 1e-15
    alpha = 1
    cost_history = np.zeros((self.training_steps, 3))
    Jaco = self.Jacobian(init_p)
```

```

Hessian = np.dot(Jaco.transpose(), Jaco)
resi = self.residual(init_p)
m_g = np.dot(Jaco.transpose(), resi)
cost_old = np.dot(resi.transpose(), resi)
epoch = -1
train_stop = -1
miu = Hessian.max()
lamda_v = 2

matrix_I = np.identity(Hessian.shape[0])
while epoch < self.training_steps - 1 and train_stop < 0:
    epoch += 1
    print "Epoch: ", epoch
    go_next_epoch = -1

    while go_next_epoch < 0:
        sigma_p = np.dot(np.linalg.inv(Hessian + miu*matrix_I), m_g)
        square_g = np.linalg.norm(sigma_p)
        if square_g <= epsilon_2*np.linalg.norm(init_p):
            train_stop = 1
            break
        else:
            new_p = init_p + sigma_p
            resi_new = self.residual(new_p)
            cost_new = np.dot(resi_new.transpose(), resi_new)
            frac = np.dot(sigma_p.transpose(), miu* sigma_p + m_g)
            rou = (cost_old - cost_new) / frac
            if rou > 0:
                go_next_epoch = 1
                init_p = new_p
                Jaco = self.Jacobian(init_p)
                Hessian = np.dot(Jaco.transpose(), Jaco)
                resi = self.residual(init_p)
                m_g = np.dot(Jaco.transpose(), resi)
                if (np.abs(m_g)).max < epsilon_1:
                    train_stop = 1
                    break
                miu *= max(1.0 / 3.0, 1 - pow(2 * rou - 1, 3))
                lamda_v = 2
                print init_p.transpose(), miu, cost_new
                cost_history[epoch, 0] = cost_new
                cost_history[epoch, 1] = miu
                cost_history[epoch, 2] = np.linalg.norm(m_g)
                cost_old = cost_new
            else:
                miu *= lamda_v
                # lamda_miu *= 0
                lamda_v *= 2
return cost_history, init_p

```


Dogleg算法

Powell's Dog Leg Method是一种信赖域方法

在Gauss-Netwon迭代中

$$J(x)h \simeq -f(x)$$

最陡的方向由下面公式给出：

$$h_{sd} = -g = -J(x)^T f(x)$$

但是这只是给出了方向，而没有给出步长。

考虑线性模型

$$f(x + \alpha h_{sd}) \simeq f(x) + \alpha J(x)h_{sd}$$

$$F(x + \alpha h_{sd}) \simeq \frac{1}{2} \|f(x) + \alpha J(x)h_{sd}\|^2 = F(x) + \alpha h_{sd}^T J(x)^T f(x) + \frac{1}{2} \alpha^2 \|J(x)h_{sd}\|^2$$

当 α 取如下值得时候，以上函数取最小值

$$\alpha = -\frac{h_{sd}^T J(x)^T f(x)}{\|J(x)h_{sd}\|^2} = \frac{\|g\|^2}{\|J(x)h_{sd}\|^2}$$

现在有两个步长的选择 $a = \alpha h_{sd}$ 以及 $b = h_{gn}$, Powell建议在信赖域半径是 Δ 的时候，步长可以如下选择

If $\|h_{gn}\| \leq \Delta$, $h_{dl} = h_{gn}$

Elseif: $\|\alpha h_{sd}\| \geq \Delta$, $h_{dl} = (\Delta / \|h_{sd}\|)h_{sd}$

else:

$$h_{dl} = \alpha h_{sd} + \beta(h_{gn} - \alpha h_{sd})$$

选择 β 使得 $\|h_{dl}\| = \Delta$

在L-M算法中我们定义了增益因子,

$$\rho = (F(x) - F(x + h_{dl}) / (L(0) - L(h_{dl})))$$

其中 L 是线性模型

$$L(h) = \frac{1}{2} \|f(x) + J(x)h\|^2$$

在L-M我们通过 ρ 来控制阻尼因子，在dog-leg算法中，我们通过它来控制步长

Dog Leg Method:

$$k := 0; x = x_0; \Delta := \Delta_0; g := J(x)^T f(x)$$

found:= ($\|f(x)\|_\infty \leq \epsilon_3$) or $\|g\|_\infty \leq \epsilon_1$

while(not found) and $k < k_{max}$

k:=k+1,计算 α

$$h_{sd} := -\alpha g; \text{ solve } J(x)h_{gn} \simeq -f(x)$$

计算 h_{dl}

```

if ||hdl|| ≤ ε(||x|| + ε2)
    found := true
else:
    xnew := x + hdl
    ρ := (F(x) - F(x + hdl)/(L(0) - L(hdl)))
    if ρ ≥ 0
        x := xnew; g := J(x)Tf(x)
        found:= (||f(x)||∞ ≤ ε3)or||g||∞ ≤ ε1
        if ρ ≥ 0.75
            Δ := max(Δ, 3 * ||hdl||)
        else if ρ ≤ 0.25
            Δ := Δ/2
    found:= Δ ≤ ε2(||x|| + ε)

```

```

def Dogleg_Solver(self):
    init_para = np.ones((2, 1))
    epsilon_1 = 1e-15
    epsilon_2 = 1e-15
    epsilon_3 = 1e-15
    alpha = 1
    cost_history = np.zeros((self.training_steps, 3))
    Jaco = self.Jacobian(init_para) //get Jacobian
    Hessian = np.dot(Jaco.transpose(), Jaco)
    resi = self.residual(init_para)
    m_g = np.dot(Jaco.transpose(), resi)
    cost_old = np.dot(resi.transpose(), resi)

    epoch = -1
    train_stop = -1
    delta = 0.01
    while epoch < self.training_steps - 1 and train_stop < 0:
        epoch += 1
        print "Epoch: ", epoch
        go_next_epoch = -1

        while go_next_epoch < 0:
            square_g = np.linalg.norm(m_g)
            square_Jg = np.linalg.norm(np.dot(Jaco, m_g))
            alpha = pow(square_g, 2) / pow(square_Jg, 2)

            h_sd = -alpha * m_g
            h_gn = np.dot(np.linalg.inv(Hessian), -m_g)
            norm_para = np.linalg.norm(init_para)

            ##calculate h_dl

```

```

    if np.linalg.norm(h_gn) <= delta:
        h_dl = h_gn
        hdl_type = 0
    elif np.linalg.norm(h_sd) >= delta:
        h_dl = (delta / np.linalg.norm(h_sd)) * h_sd
        hdl_type = 1
    else:
        beta = self._get_h_dl(h_sd, h_gn, delta)
        h_dl = h_sd + beta * (h_gn - h_sd)
        hdl_type = 2

        # #epsilon_2*norm_para:
    if np.linalg.norm(h_dl) < epsilon_2 * (norm_para + epsilon_2):
        train_stop = 1
        break
    else:
        para_0 = init_para + h_dl
        #cost_old = pow(np.linalg.norm(resi), 2)
        resi_new = self.residual(para_0)
        cost_new = np.dot(resi_new.transpose(), resi_new)
        #pow(np.linalg.norm(resi_new), 2)
        #h_sub = h_sd - 0.5 * np.dot(Hessian, h_dl)
        #frac = np.dot(h_dl.transpose(), h_sub)
        if hdl_type == 0:
            frac = cost_new
        elif hdl_type == 1:
            frac = delta/(2*alpha)*(2*np.linalg.norm(alpha*m_g) - delta)
        else:
            frac = 0.5*alpha*pow(beta, 2)*pow(np.linalg.norm(alpha*m_g), 2)
            frac += beta*(2 - beta)*cost_new
        rou = (cost_old - cost_new) /frac
        if rou > 0:
            go_next_epoch = 1
            init_para = para_0
            Jaco = self.Jacobian(init_para)
            Hessian = np.dot(Jaco.transpose(), Jaco)
            resi = self.residual(init_para)
            m_g = np.dot(Jaco.transpose(), resi)

            cost_history[epoch, 0] = cost_new
            cost_history[epoch, 1] = delta
            cost_history[epoch, 2] = np.linalg.norm(m_g)
            print para_0.transpose(), cost_new, delta

            if (np.abs(m_g)).max() < epsilon_1 or (np.abs(resi_new)).max() < epsilon_1:
                train_stop = 1
                break
            if rou > 0.75:
                delta = max(delta, 3 * np.linalg.norm(h_dl))
            elif rou < 0.25:
                delta /= 2.0
                if delta < epsilon_2:
                    break
            cost_old = cost_new

```

```
return cost_history, init_para
```

共轭梯度法

CG方法分类

共轭梯度法可以用来处理线性方程组，尤其是大规模线性方程组，同时也可以用来处理非线性系统。处理线性系统问题的共轭梯度法在矩阵计算那节有过介绍，这里，我们再介绍一下，并把他推广到非线性情形。

非线性情形

由于任何非线性最优化问题，在解的附近可以做二次展开，如果省略常数项，则这个非线性函数可以用如下方程来表示。

$$f(x) = \frac{1}{2}x^T Ax - x^T b,$$

因此，讨论这个二次方程的优化问题就可以推广到非线性优化问题。

共轭梯度法

共轭梯度法(Conjugate Gradient)是介于最速下降法与牛顿法之间的一个方法，它仅需要一阶导数信息，但克服了最速下降法收敛慢的缺点，同时又避免了牛顿法需要储存和计算Hesse矩阵并求逆的缺点，共轭梯度法不仅是解决大型线性方程组最有用的方法之一，也是解大型非线性最优化问题最有效的算法之一。

向量共轭的定义：

若A是正定对称阵，若非0矢量u, v满足， $u^T A v = 0$ ，则称矢量u, v是A共轭的。

假设：

$P = [p_1, p_2, \dots, p_n]$ 是一组基于A的共轭矢量，则 $Ax = b$ 的解 x^* 可以表示为：

$x^* = \sum_{i=1}^n \alpha_i p_i$ ，因此基于基矢量展开，我们有：

$$Ax^* = \sum_{i=1}^n \alpha_i A p_i$$

把 x^* 代入 $f(x)$ ，求极小，我们可以得到每个基矢的分量 α_i 。

左乘 P_k^T ：

$$P_k^T Ax^* = \sum_{i=1}^n \alpha_i P_k^T A p_i$$

代入： $P_k^T Ax^* = b$, $u^T A v = \langle u, v \rangle_A$ ，利用 $i! = k$ 有 $\langle p_k, p_i \rangle_A = 0$ 就得到： $\alpha_i = \frac{b^T p_i}{p_i^T A p_i}$ 如果我

们的小心的选择 p_k ，为了获得解 x^* 的一个好的近似，我们并不需要所有的基向量。因此，我们把共轭梯度法当成一种迭代算法，它也允许我们近似 n 很大，以至于直接求解需要花费太多时间的系统。

我们给 x_* 一个初始猜测值 x_0 , 假设 $x_0 = \mathbf{0}$, 在求解的过程中, 我们需要一种标准来告诉我们是否我们的解更靠近真实的解 x_* , 实际上, 求解 $\mathbf{Ax} = \mathbf{b}$ 等价于求解如下二次函数的极小值:

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{x}^T \mathbf{b},$$

他存在唯一的最小值, 因为他的二阶导是正定的:

$$\mathbf{D}^2 f(\mathbf{x}) = \mathbf{A},$$

解就是一阶导数等于0的地方

$$\mathbf{Df}(\mathbf{x}) = \mathbf{Ax} - \mathbf{b},$$

假设第一个基矢 \mathbf{p}_0 就是 $f(x)$ 在 $\mathbf{x} = \mathbf{x}_0$ 处的负梯度, 我们可以假设 $\mathbf{x}_0 = \mathbf{0}$. 因此:

$$\mathbf{p}_0 = \mathbf{b} - \mathbf{Ax}_0.$$

令 \mathbf{r}_0 表示第 k 步的残差:

$$\mathbf{r}_k = \mathbf{b} - \mathbf{Ax}_k,$$

\mathbf{r}_k 是 $f(\mathbf{x})$ 在 $\mathbf{x} = \mathbf{x}_k$ 处的负梯度。

为了让 \mathbf{p}_k 与前面的所有 \mathbf{p}_i 相互共轭, \mathbf{p}_k 可以如下构造:

$$\mathbf{p}_k = \mathbf{r}_k - \sum_{i < k} \frac{\mathbf{p}_i^T \mathbf{A} \mathbf{r}_k}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}$$

沿着这个方向, 因此下一步的优化方向就是:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \text{ 满足:}$$

$$g'(\alpha_k) = 0$$

因此:

$$f(\mathbf{x}_{k+1}) = f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) =: g(\alpha_k),$$

$$\alpha_k = \frac{\mathbf{p}_k^T \mathbf{b}}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k} = \frac{\mathbf{p}_k^T (\mathbf{r}_k + \mathbf{A} \mathbf{x}_k)}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k} = \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}$$

算法:

$$\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$$

$$\mathbf{p}_0 := \mathbf{r}_0$$

$$k := 0$$

repeat:

$$\alpha_k := \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A} \mathbf{p}_k}$$

$$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

$$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k$$

if r_{k+1} is sufficiently small, then exit loop.

$$\beta_k := \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$$

$$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$$

$k := k + 1$ 最终的解就是 \mathbf{x}_{k+1}

CG方法的核心思想

通过构造一系列A-共轭的基矢量，比如k个，再把方程的解在这个基矢中做展开，算法的核心就是在于构造这些A-共轭的基矢量。

这个一步步构造共轭基矢量的方法与Gram-Schmidt方法没有区别，更进一步来说，Krylov子空间理论中构造正交基矢也用的是这种方法，Krylov构造的这些基矢最终构建的也是一个把原始矩阵相似变换为上Hessenberg矩阵的正交矩阵。最终，我们的解可以用这一组共轭的基矢来表示，而且我们需要的基矢数目不多，也就是我们迭代的步数不多，或者说只需要一个很小的Krylov子空间，因此CG方法是很高效的。因此CG方法也是Krylov子空间理论方法的一种。

一个核心问题就是我们只能处理对称矩阵A，因为我们需要矩阵是正定的，这样我们才能构建一组共轭的基矢。那我们对于非对称矩阵也可以使用类似于CG的方法吗？

次梯度法

对于不可导的函数，我们不能求导，因此可选方案是使用次梯度法。次梯度是一个集合，定义如下：

$$\partial f = [g | f() \geq f(x_0) + g^T(x - x_0), x \in \text{dom}f, f : R^n \rightarrow R]$$



上图中次梯度是一个区间，如 x_2 点的次梯度是过改点并处于曲线之下所有直线斜率构成的集合。

可以证明，在 x_0 点的次梯度是一个非闭空间 $[a, b]$ ， a, b 是单侧极限。

$$a = \lim_{x \rightarrow x_0^-} \frac{f(x) - f(x_0)}{x - x_0}$$

$$b = \lim_{x \rightarrow x_0^+} \frac{f(x) - f(x_0)}{x - x_0}$$

他们一定存在并且满足 $a \leq b$ ，所有次导数的集合 $[a, b]$ 称为函数 f 在 x_0 的次梯度。

例子： $y = |x|$ 在 $x=0$ 处的次梯度是 $[-1, 1]$ 。

次梯度法

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

其中 $g^{(k)} \in \partial f(x^{(k)})$

然而， $-g^{(k)}$ 可能不再是下降方向，所以常用的方式是一直保留最小的函数值，直到结果收敛。

$$f_{best}^{(k)} = \min((f_{best}^{(k)}, f(x^{(k)})))$$

随机坐标下降法

随机梯度法是对样本进行随机采样，当然我们也可以对模型的维度进行随机采样，这就是我们要介绍的随机坐标下降法。它的更新公式如下：

$$w_{t+1,j_t} = w_{t,j_t} - \eta_t \nabla_{j_t} f(w_t)$$

其中 j_t 表示第 t 次迭代中随机选取的维度标号， $\nabla_{j_t} f(w_t)$ 是损失函数对于模型 w_t 中第 j_t 个维度的偏导数。

随机坐标下降法流程

Initialize: w_0

Iterate: for $t = 0, 1, \dots, T-1$

 1. 随机选取一个维度 $j_t \in [1, 2, \dots, d]$

 2. 计算梯度 $\nabla_{j_t} f(w_t)$

 3. 更新参数：

$$w_{t+1,j_t} = w_{t,j_t} - \eta_t \nabla_{j_t} f(w_t)$$

end

有放回抽样

一方面，如果采样方法是有放回的，那么可以得到：

$$E_{j_t}[\nabla_{j_t} f(w_t)] = \frac{1}{d} \nabla f(w_t)$$

此处 $\nabla_{j_t} f(w_t)$ 是 d 维向量，其中第 j_t 个维度是 $\nabla_{j_t} f(w_t)$ ，其它维度是0。因此在期望方面，随机坐标梯度与梯度方面是一致的。

另一方面，对于线性模型，计算一个维度的梯度所需要的计算量只是计算整个梯度向量的 $1/d$ 。因此，随机坐标梯度可以作为原始梯度的高效替代品，尤其在维度较高时。

偏导数

定义：如果对任意模型 $w \in R^d$ ，对于维度 j 存在常数 β_j ，使得对任意 $\delta \in R$ 有下面不等式成立：

$$|\nabla_j f(w + \delta e_j) - \nabla_j f(w)| \leq \beta_j |\delta|$$

则称目标函数 f 对于维度 j 具有 β_j -Lipschitz连续的偏导数。

如果 f 对每个维度的偏导数都是Lipschitz连续的，我们记 $\beta_{max} = \max_{j=1,\dots,d} \beta_j$ 。

定理：假设目标函数 f 是 R^d 上的凸函数，并且具有 β_j -Lipschitz连续的偏导数，记：

$$w^* = \arg \min_{\|w\| \leq D} f(w),$$

当步长 $\eta = 1/\beta_{max}$ 时, 随机坐标下降法具有如下的次线性收敛速率:

$$Ef(w_T) - f(w^*) \leq \frac{2d\beta_{max}D^2}{T}$$

定理: 假设目标函数 f 是 R^D 上的 α -强凸函数, 并且具有 β_j -Lipschitz 连续的偏导数, 当步长

$\eta = 1/\beta_{max}$ 时, 随机坐标下降法具有如下的次线性收敛速率:

$$Ef(w_T) - f(w^*) \leq (1 - \frac{\alpha}{d\beta_{max}})^T (f(w_0) - f(w^*))$$

对比梯度下降法的收敛速率, 我们发现随机坐标下降法的收敛速度关于迭代次数的阶数与梯度下降法是一致的。从这个意义上讲, 随机坐标下降法的理论性质优于随机梯度下降法。¹

可分离情况

这种偏导数的计算量小于梯度的计算量的情形一般称为"可分离"情形, 对于线性模型, 常用损失函数都是可分离的。

随机块坐标下降法

Initialize: w_0

将 d 个维度均等分为 J 块。

Iterate: for $t = 0, 1, \dots, T-1$

1. 随机选取一个维度 $J_t \in [1, 2, \dots, J]$

2. 计算梯度 $\nabla_{J_t} f(w_t)$

3. 更新参数:

$$w_{t+1, J_t} = w_{t, J_t} - \eta_t \nabla_{J_t} f(w_t), j \in J_t$$

end

¹. 《分布式机器学习--算法, 理论与实践》刘铁岩 ↵

有约束最优化问题

有约束问题主要通过拉格朗日乘子与KKT乘子来转化为无约束优化问题。问题的关键是，在什么条件下可以通过乘子法来把有约束优化问题转化成无约束优化问题，因为如果我们能把转化成无约束优化问题，则求解无约束优化问题的方法都可以派上用场了。

常用的用于求解带约束问题的几种方法：

1. 对偶方法
2. 罚函数法
3. 障碍函数法
4. ADMM算法
5. 优化-最小化算法(Majorization-Minimization)¹

¹. 稀疏统计学习及其应用 ↵

拉格朗日乘子法

通过引入拉格朗日乘子，定义拉格朗日函数，通过求Lagrange函数关于原始变量的极小值来求得对偶函数。再求对偶函数关于Lagrange乘子的极大值。假设原始问题的最优解是 \mathbf{x}^* ,对偶问题的最优解是 λ^*, \mathbf{v}^* , 并且对偶间隙为0，则 $\mathbf{x}^*, \lambda^*, \mathbf{v}^*$ 满足如下的KKT条件。¹

通过KKT条件，我们可以通过求对偶问题的解来反求原始问题的解，因为原始问题的解和对偶问题的解满足KKT条件。一个典型的例子就是SVM，在机器学习，分类问题那一节有详细介绍。

Lagrange函数

考虑标准形式的优化问题(4.2.1),也称为原始问题。²

$$\min f_0(\mathbf{x})$$

$$\text{s.t. } f_i(\mathbf{x}) \leq 0, i=1,2,\dots,m$$

$$\text{s.t. } h_i(\mathbf{x}) = 0, i=1,2,\dots,p$$

其中，自变量 $\mathbf{x} \in \mathbf{R}^n$ 。问题的定义域 $D = \bigcap_{i=1}^m \text{dom} f_i \bigcap_{i=1}^p \text{dom} h_i$ 是非空集合，优化问题(不需要保证是凸优化)的最优值是 p^* 。

Lagrange对偶的基本思想是在目标函数中考虑问题4.2.1的约束条件，即添加约束条件的加权和，

得到增广的目标函数。定义问题4.2.1的Lagrange函数是 $L : \mathbf{R}^n \times \mathbf{R}^n \times \mathbf{R}^p \rightarrow R$:

$$\min L(x, \lambda, v) = f_0(x) + \sum_{i=1}^m \lambda_i f_i(x) + \sum_{i=1}^p v_i h_i(x)$$

其中定义域为 $\text{dom} L = D \times R^m \times R^p$ 。 λ_i 称为第*i*个不等式约束 $f_i(x) \leq 0$ 对应的Lagrange乘子， v_i 称为第*i*个等式约束 $h_i(x) = 0$ 对应的Lagrange乘子。向量 λ, \mathbf{v} 称为对偶变量或者问题4.2.1的Lagrange乘子向量。

Lagrange对偶函数

定义Lagrange对偶函数 $g : R^m \times R^p \rightarrow R$ 为Lagrange函数关于 \mathbf{x} 取得的最小值:即对

$\lambda \in R^m, v \in R^p$, 有:

$$g(\lambda, v) = \inf_{x \in D} L(x, \lambda, v) = \inf_{x \in D} (f_0(x) + \sum_{i=1}^m \lambda_i f_i(x) + \sum_{i=1}^p v_i h_i(x))$$

如果Lagrange函数关于 \mathbf{x} 无下界，则对偶函数取值为 $-\infty$ 。因为对偶函数是一族关于 λ, \mathbf{v} 的仿射函数的逐点下确界，所以即使原始问题4.2.1不是凸的，对偶函数也是凹的。

Lagrange对偶问题

通过上面定义的Lagrange函数, 我们可以可以定义对偶问题。³

maximise $g(\lambda, v)$

s.t. $\lambda \geq 0, v \geq 0$

对于不等式约束问题, 假设原始问题的最优解是 x^* , 对偶问题的最优解是 λ^*, v^* 。则 x^*, λ^*, v^* 满足如下的KKT条件。

¹. Stephen Boyd《凸优化》P235 ↪

². Luenberger, 叶荫宇《线性与非线性规划》第十四章 ↪

³. P.Bertsekas 《非线性规划》第三章 ↪

Karush-Kuhn-Tucker(KKT)条件

1. 先讨论一般带约束优化问题的数据描述
2. 然后引入其作用约束与不起作用约束的定义
3. 再引入正则点的定义
4. 接下来引入KKT条件
5. 针对局部极小问题，我们考虑优化函数的二阶导数问题。

一般形式的优化问题：

$$\text{minimize } f(\mathbf{x})$$

$$\text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0}$$

$$\text{s.t. } \mathbf{g}(\mathbf{x}) \leq \mathbf{0}$$

其中： $f : R^n \rightarrow R$, $\mathbf{h} : R^n \rightarrow R^m$, $m \leq n$; $\mathbf{g} : R^n \rightarrow R^p$;

针对这一问题，引入如下定义：

定义21.1. 对于一个不等式约束 $g_i(x) \leq 0$, 如果在 x^* 处, $g_i(x^*) = 0$, 那么称该不等式约束是 x^* 处的起作用约束。如果 $g_i(x^*) < 0$, 那么称该不等式约束是 x^* 处的不起作用约束。

定义21.2. 设 \mathbf{x}^* 满足 $\mathbf{h}(\mathbf{x}^*) = \mathbf{0}$, $\mathbf{g}(\mathbf{x}^*) \leq \mathbf{0}$, 设 $J(\mathbf{x}^*)$ 为起作用不等式约束下标集,

$$J(\mathbf{x}^*) = \{j : g_j(\mathbf{x}^*) = 0\}$$

如果： $\nabla h_i(\mathbf{x}^*)$, $\nabla g_j(\mathbf{x}^*)$, $i \leq i \leq m$, $j \in J(\mathbf{x}^*)$

是线性无关的，则称 \mathbf{x}^* 是一个正则点。

KKT条件：某个点是局部极小点所满足的一阶必要条件。

定理：KKT条件，设 $f, \mathbf{g}, \mathbf{h} \in C^1$, 设 \mathbf{x}^* 是问题 $\mathbf{h} = \mathbf{0}$, $\mathbf{g} \leq \mathbf{0}$ 的一个正则点和局部极小点，那么必然存在 $\lambda^* \in R^m$ 和 $\mathbf{u}^* \in R^p$ 使得以下条件成立：

$$\mathbf{u}^* \geq \mathbf{0}$$

$$Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$$

$$\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$$

λ^* 为拉格朗日乘子向量， \mathbf{u}^* 是 KKT 乘子向量，其元素分别成为拉格朗日乘子，KKT 乘子。

充分条件

上面讲了局部极小的必要条件，这里我们讨论局部极小的充分条件。然后我们就利用KKT条件去求不等式约束问题。

二阶充分必要条件；

定义如下矩阵：

$$L(\mathbf{x}, \lambda, \mathbf{u}) = F(\mathbf{x}) + \lambda \mathbf{H}(\mathbf{x}) + \mathbf{u} \mathbf{G}(\mathbf{x})$$

$F(\mathbf{x})$ 是 f 在 \mathbf{x} 处的Hessian矩阵；

$$\lambda \mathbf{H}(\mathbf{x}) = \lambda_1 H_1(\mathbf{x}) + \dots + \lambda_m H_m(\mathbf{x})$$

$$\mathbf{u} \mathbf{G}(\mathbf{x}) = u_1 G_1(\mathbf{x}) + \dots + u_p G_p(\mathbf{x})$$

其中 $G_k(\mathbf{x})$ 是 g_k 的Hessian矩阵。

起作用约束所定义曲面的切线空间：

$$T(\mathbf{x}^*) = \{y \in R^n : Dh(x^*)y = 0, Dg_j(x^*)y = 0, j \in J(x^*)\}.$$

定理：二阶必要条件：

如果 \mathbf{x}^* 是上面讨论的优化问题的极小点，那么存在 $\lambda^* \in R^m, \mathbf{u}^* \in R^p$ 使得：

1. $\mathbf{u}^* \geq 0,$

2. $Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$

3. $\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$

4. 对所有 $y \in T(\mathbf{x}^*, \mathbf{u}^*), y \neq 0$, 都有 $y^T L(\mathbf{x}, \lambda, \mathbf{u}) y > 0$

定理：二阶充分条件：

假设 $f, \mathbf{g}, \mathbf{h} \in C^2, \mathbf{x}^* \in R^n$ 是一个可行点，存在向量 $\lambda^* \in R^m, \mathbf{u}^* \in R^p$ 使得：

1. $\mathbf{u}^* \geq 0,$

2. $Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$

3. $\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$

4. 对所有 $y \in T(\mathbf{x}^*, \mathbf{u}^*), y \neq 0$, 都有 $y^T L(\mathbf{x}, \lambda, \mathbf{u}) y > 0$

那么 \mathbf{x}^* 是优化问题

$$\text{minimize } f(\mathbf{x})$$

$$\text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0}$$

$$\text{s.t. } \mathbf{g}(\mathbf{x}) \leq \mathbf{0}$$

的严格局部极小点，

$$T(\mathbf{x}^*) = \{y \in R^n : Dh(x^*)y = 0, Dg_j(x^*)y = 0, j \in J(x^*)\}.$$

其中：

$$J(\mathbf{x}^*, \mathbf{u}^*) = \{j : g_j(\mathbf{x}^*) = 0, \mathbf{u}^* > 0\}$$

罚函数法

在用SVM处理线性不可分分类的时候，我们引入了软间隔最大化，这就是一种罚函数法来处理带约束的优化问题。下面我们形式化这个处理带约束优化问题的方法。

一般的约束优化问题

考虑如下问题penalty_function_problem:

$$\begin{aligned} & \min f(\mathbf{x}) \\ & \text{s.t. } \mathbf{x} \in S \end{aligned}$$

其中 f 是 E^n 上的连续函数， S 是 E^n 上的约束集，在大部分应用中， S 由一系列函数约束隐式的定义，但是在这部分中，可以处理如上式中更一般的情形。外点法的思想在于将上式替换成如下形式的无约束问题。

$$\min f(\mathbf{x}) + cP(\mathbf{x})$$

其中 c 是正常数， P 是 E^n 上的函数，满足如下三个条件。

(1) P 是连续的函数。

(2) $P(\mathbf{x}) \geq 0$ 对任意的 $\mathbf{x} \in E^n$

(3) $P(\mathbf{x}) = 0$ 当且仅当 $\mathbf{x} \in S$

常见的不等式约束

假设 S 由一些列不等式约束定义：

$$S = [\mathbf{x} : g_i(\mathbf{x}) \leq 0], i = 0, 1, \dots, p$$

在这里，一个非常有用的罚函数为：

$$P(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^p (\max[0, g_i(\mathbf{x})])^2$$

这就是通过定义软间隔最大化时，用SVM处理不可分分类问题。

求解步骤

用罚函数求解原始问题的步骤如下：令 c_k , $k=1, 2, \dots$ 为区域无穷的序列，并且对任意 k ,

$c_{k+1} > c_k \geq 0$ 。定义函数：

$$q(c, \mathbf{x}) = f(\mathbf{x}) + cP(\mathbf{x})$$

对每个k求线性规划问题：

$$\min q(c_k, \mathbf{x})$$

得到最优解点 \mathbf{x}_k 。

有关解的定理

定理：令 c_k 是罚函数生成的序列。那么，序列的任何极限点都是penalty_function_problem的解。

精确惩罚函数

事实上，可以建立精确罚函数，使得对于有限的惩罚变量，罚函数问题能生成与原问题完全相同的解。对于这些函数，不需要为了求得正确解而求无穷序列产生的罚函数问题，然而，这些罚函数会引入一个新的问题--不可微性。

考虑如下问题PFP_3：

$$\begin{aligned} & \min f(\mathbf{x}) \\ & \text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0}, \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

惩罚函数取绝对值惩罚函数PFP_4：

$$P(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^m |h_i(\mathbf{x})| + \frac{1}{2} \sum_{i=1}^p \max[0, g_i(\mathbf{x})]$$

从而对某个正常数c，惩罚函数问题为：

$$\min f(\mathbf{x}) + cP(\mathbf{x})$$

精确惩罚函数定理

假设 x^* 满足约束规划问题general_penalty_function_problem局部极小的二阶充分条件。 λ, \mathbf{u} 是相应的拉格朗日乘子。则对于 $c > \max(|\lambda_i|, u_i : i = 1, 2, \dots, m, j = 1, 2, \dots, p)$ ， \mathbf{x}^* 也是绝对值惩罚函数问题PFP_4的局部最小点。

增广拉格朗日函数法¹

增广拉格朗日函数法是常用的用于求解非线性规划问题的方法，这种方法可以看作是罚函数法和局部对偶法的结合，两种方法的结合可以消除单独使用某一种方法所产生的缺陷。

等式约束

在等式约束问题中：

$$\begin{aligned} & \min f(\mathbf{x}) \\ \text{s.t. } & \mathbf{h}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in \Omega \end{aligned}$$

下，对于某个正常数c，增广拉格朗日函数为：

$$l_c(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda^T \mathbf{h}(\mathbf{x}) + \frac{1}{2}c|\mathbf{h}(\mathbf{x})|^2$$

可以从以下两种观点来看增广拉格朗日函数。

罚函数角度

对于固定的 λ 向量，增广拉格朗日函数就是如下规划问题：

$$\begin{aligned} & \min f(\mathbf{x}) \\ \text{s.t. } & \mathbf{h}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in \Omega \end{aligned}$$

的标准二次罚函数，引入二阶惩罚可以更好地收敛。该问题与原始问题等价。

对偶理论

从对偶理论的角度来看，增广拉格朗日函数就是如下问题：

$$\begin{aligned} & \min f(\mathbf{x}) + \frac{1}{2}c|\mathbf{h}(\mathbf{x})|^2 \\ \text{s.t. } & \mathbf{h}(\mathbf{x}) = \mathbf{0}, \mathbf{x} \in \Omega \end{aligned}$$

的标准拉格朗日函数。该问题与原始问题等价。

尽管原问题可能不是局部凸的，但是对于足够大的c，上面问题是局部凸的，特别是，拉格朗日函数在解点 \mathbf{x}^*, λ^* 处的Hessian矩阵是正定的。因此，对于足够大的c，局部对偶定理适合于上面的问题。

对上面问题应用对偶法，我们在 \mathbf{x}^*, λ^* 附件区域定义对偶函数：

$$\phi(\lambda) = \min[f(\mathbf{x}) + \lambda^T \mathbf{h}(\mathbf{x}) + \frac{1}{2}c|\mathbf{h}(\mathbf{x})|^2]$$

如果 $\mathbf{x}(\lambda)$ 为最小化上式右侧部分的向量，那么 $\mathbf{h}(\mathbf{x}(\lambda))$ 是上面对偶函数 ϕ 的梯度。因此，基本增广拉格朗日法采用如下的迭代过程。

$$\lambda_{k+1} = \lambda_k + c\mathbf{h}(\mathbf{x}(\lambda_k))$$

被看作是最大化对偶函数 ϕ 的一种最速上升法迭代，使用常数步长c。

不等式约束

上面主要讨论了等式约束的增广拉格朗日乘子法，其实我们可以轻松的推广到不等式约束的情形。

$$\begin{aligned} & \min f(\mathbf{x}) \\ & \text{s.t. } \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

我们假定问题存在良定的解 \mathbf{x}^* , \mathbf{x}^* 为约束条件的正则点，并且满足局部最小化问题的二解充分条件。这个问题也可以等价地写成等式约束问题：

$$\begin{aligned} & \min f(\mathbf{x}) \\ & \text{s.t. } \mathbf{g}(\mathbf{x}) + \mathbf{u} = \mathbf{0}, \mathbf{u} \geq \mathbf{0} \end{aligned}$$

1. 《线性与非线性规划》David G. Luenberger ↵

ADMM(乘子的交替方向法)¹

在分布式计算中,为了减少通讯成本,就需要我们减少求解问题的迭代次数;要降低迭代次数,就必然要求在一个迭代内完成更复杂的计算。因此有了ADMM。

考虑具有线性约束并且目标函数是两个分开函数之和形式的图最小化模型。

$$\begin{aligned} & \min f_1(\mathbf{x}^{(1)}) + f_2(\mathbf{x}^{(2)}) \\ \text{s.t. } & \mathbf{A}_1\mathbf{x}^{(1)} + \mathbf{A}_2\mathbf{x}^{(2)} = \mathbf{b} \\ & \mathbf{x}^{(1)} \in \Omega_1, \mathbf{x}^{(2)} \in \Omega_2 \end{aligned}$$

其中 $A_i \in E^{m \times n_i}, \{i=1,2\}, b \in E^m, \Omega_i \subset E^{n_i} (i=1,2)$ 是闭凸集;且 $f_i : E^{n_i} \rightarrow E, i=1,2$ 分别是 Ω_i 上的凸函数。因此上面问题的增广拉格朗日函数是:

$$l_c(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \lambda) = f_1(\mathbf{x}^{(1)}) + f_2(\mathbf{x}^{(2)}) + \lambda^T (\mathbf{A}_1\mathbf{x}^{(1)} + \mathbf{A}_2\mathbf{x}^{(2)} - \mathbf{b}) + \frac{c}{2} \|\mathbf{A}_1\mathbf{x}^{(1)} + \mathbf{A}_2\mathbf{x}^{(2)} - \mathbf{b}\|^2$$

我们假定上面的问题至少有一个最优解。

相比于乘子方法, ADMM是以另外的规则(近似)最小化 $l_c(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \lambda)$

$$\begin{aligned} \mathbf{x}_{k+1}^{(1)} &:= \arg \min_{x^{(1)} \in \Omega_1} l_c(\mathbf{x}^{(1)}, \mathbf{x}_k^{(2)}, \lambda_k) \\ \mathbf{x}_{k+1}^{(2)} &:= \arg \min_{x^{(2)} \in \Omega_2} l_c(\mathbf{x}_{k+1}^{(1)}, \mathbf{x}^{(2)}, \lambda_k) \\ \lambda_{k+1} &:= \lambda_k + c(A_1x_{k+1}^{(1)} + A_2x_{k+1}^{(2)} - b) \end{aligned}$$

这种思想是每一个更小的最优化问题都能被更有效地解决或者某些情形有接近的形式。

例子

ADMM最典型的例子就是加L1或者L2正则化的优化问题;下面以L1正则化为例。

用ADMM求解Lasso问题

lasso的拉格朗日形式可以等价表示成:

$$\min_{\beta \in R^p, \theta \in R^p} \left[\frac{1}{2N} \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \frac{1}{N} \lambda \|\theta\|_1 \right], \beta - \theta = 0$$

相应的增广拉格朗日函数为:

$$\min_{\beta \in R^p, \theta \in R^p} \left[\frac{1}{2N} \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \frac{1}{N} \lambda \|\theta\|_1 + \rho \|\beta - \theta\|_2^2 \right]$$

对于lasso, ADMM的更新公式为:

$$\beta^{t+1} = (\mathbf{X}^T \mathbf{X} + \rho \mathbf{I})^{-1} (\mathbf{X}^T \mathbf{y} + \rho \theta^t - u^t)$$

$$\theta^{t+1} = S_{\lambda/\rho}(\beta^{t+1} + u^t/\rho)$$

$$u^{t+1} = u^t + \rho(\beta^{t+1} - \theta^{t+1})$$

对 β 进行岭回归, 对 θ 进行的更新是软阈值, 对 u 采用线性更新。

最优化的优缺点及其改进方案

常见的损失函数

分类问题损失函数

y 是标签, f 是预测结果。

对于二分类问题 $Y = [-1, 1]$

0-1损失函数:

$$L_{0-1}(f, y) = 1_{fy \leq 0}$$

Hinge损失函数:

$$L_{hinge}(f, y) = \max(0, 1 - fy)$$

Logistic损失函数:

$$L_{logistic}(f, y) = \log_2(1 + \exp(-fy))$$

当预测值 $f \in [-1, 1]$ 时, 可以定义如下损失函数:

交叉熵(Cross Entropy):

$$L_{crossentropy} = -\log_2\left(\frac{1+fy}{2}\right)$$

回归问题损失函数

对于回归问题, $Y=R$, 我们希望 $f(x_i, \theta) \approx y_i$, 最常用的损失函数是平方损失函数:

$$L_{square}(f, y) = (f - y)^2$$

平方损失函数对异常点惩罚很大, 因此对异常点比较敏感。为了解决这个问题, 引入了绝对损失函数:

$$L_{square}(f, y) = |f - y|$$

但是绝对损失函数在0点不可以求导。

一个方法是在0点进行函数光滑化也就是在0点附件用平方函数代替。 $\delta > 0$

$$L_{square}(f, y) = \sqrt{(f - y)^2 + \delta^2}$$

或者用Huber损失函数:

Huber损失函数:

$$L_{square}(f, y) = \frac{1}{2}(f - y)^2, |f - y| < \delta$$

$$L_{square}(f, y) = \delta|f - y| - \frac{1}{2}\delta^2, |f - y| > \delta$$

误差大的时候选择线性惩罚, 误差小的时候选择平方惩罚。是一种Robust Regression损失函数。

还得补充与思考文本处理, 信号处理中的损失函数。

机器学习中哪些是凸优化问题，哪些是非凸优化问题

凸函数条件：Hessian矩阵为半正定。

逻辑回归，线性回归，SVM是凸优化问题。PCA，矩阵分解以及NN是非凸优化问题。

梯度下降

问题1：怎样验证自己代码中梯度公式的正确性？

$$\left| \frac{L(\theta + he_i) - L(\theta - he_i)}{2h} - \frac{\partial L(\theta)}{\partial \theta_i} \right| \approx Mh^2$$

若 $M \leq 10^7$, 则：

$$\left| \frac{L(\theta + he_i) - L(\theta - he_i)}{2h} - \frac{\partial L(\theta)}{\partial \theta_i} \right| \leq h$$

因此取 $h = 10^{-7}$ 看，看上式是否成立。不成立则 $M \geq 10^7$, 则取 $h = 10^{-8}$, 看上式得值是否变为原来的 10^{-2} 。如果是，则求导正确，不是则求导错误。

问题2：当数据量特别大的时候经典梯度下降法存在什么问题，需要怎么改进？

因为经典的梯度下降是对整个训练集进行的，如果训练集特别大，每次更新要针对整个数据集，因此需要很大的计算力，计算很费时间。

一般采用随机梯度下降法，每次训练一个数据。但是该方法很难收敛，会在极值点振荡，因此一般采用 mini-batch Gradient Descent 方法。一般 batch 的 size m 取 2 的整数次幂。

为了充分利用数据以及避免数据的特定顺序给算法收敛带来的印象，一般每次遍历数据前，对数据先随机排序，然后每次取 m 个数据，直到遍历所有数据。有必要再按新规则排序，重复以上步骤。
如何选择学习率 α ，为了加快学习速度与提高精度。一开始取大的学习速率，误差曲线进入平台期后，再减小学习速率做更精细的调整。

问题3：深度学习中常用的优化方法是随机梯度下降法，但是 SGD 偶尔也会失效，无法给出满意的训练结果，这是为什么？

当 SGD 遇到山谷的时候，由于沿山谷方向的梯度比较小，因此结果会在山谷方向来回移动，而不能像整体梯度下降一样能沿着山谷有一个确定方向的移动。其次是遇到平坦区域，有些方向的梯度接近于 0，因此需要很长世间才能走出这片区域。

问题4：为了改进 SGD，研究者都做了哪些改动？提出了哪些方法？他们各有哪些特点？

动量 (Momentum) 方法

为了解决SGD在山谷振荡，鞍点停滞的现象。我们引入了带动量的SGD。普通SGD是：

$$\theta_{t+1} = \theta_t - \eta g_t$$

带动量的SGD公式是：

$$v_t = \gamma v_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - v_t$$

通过 $0 < \gamma < 1$, 能实现梯度的累积, 也就是速度的累积, 因此能更好的冲出山谷与平坦(平原)区域。

AdaGrad方法

惯性的获得是基于历史的, 除了从过去的步伐中获得一股子冲劲, 我们还想知道不同参数更新的步伐, 这样, 对于更新频率低的参数能加快更新步伐, 而更新频率高的参数减小步伐。因此就有了AdaGrad方法, 他采用"历史梯度平均和"来衡量不同参数的梯度的稀疏性。, 取值越小表明越稀疏。更新公式如下:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{(\sum_{k=0}^t g_{k,i}^2 + \epsilon)^{1/2}} g_{t,i}$$

$\theta_{t+1,i}$ 表示t+1时刻, 参数向量的第i个参数, $g_{k,i}$ 表示k时刻, 参数向量的第i个方向。此外, 分母的求和新式实现了退火过程, 意味着随着时间推移, 学习速率越来越小, 保证算法最终收敛。

Adam方法

Adam方法将惯性保持与环境感知这两个有点集合在一起。

通过记录梯度的一阶矩来保持惯性, 通过记录二阶矩来实现环境感知能力。通过指数衰减平均技术来实现时间久远的梯度对当前平均值的贡献呈指数衰减。计算公式如下:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Adam更新公式:

$$\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{(\hat{v}_t + \epsilon)^{1/2}}$$

其中: $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$

问题5:L1正则化使得模型参数具有稀疏性的原理是什么？

角度1:通过做等高线图



由图可知, L1更容易在轴上与等高线相交, 而L2更容易在两轴之间相交。其实加L1,L2正则就是加一个约束, 也就是带约束的优化问题, 通过这个约束, 我们就知道了解空间的限定范围。

角度2: 贝叶斯先验

L1正则化相当于对模型参数引入了拉普拉斯先验, L2正则化相当于对模型参数引入了高斯先验。因为拉普拉斯先验分布中, 参数取值为0的概率更高, 因此更容易产生稀疏性。而高斯先验分布只会让所有参数趋于0, 并且在趋于0的不同位置是等概率的, 因此不会让参数等于0。

优化收敛位置

对于N个参数的系统(神经网络), 对应的Hessian阵是N阶的, 我们假定其本征值的正负概率都是0.5. 因此, 要保证是局部最小值, 则Hessian正定, 意味着所有的本征值都为正。其概率是 $\frac{1}{2^N}$ 。同样, 是局部最大值, 则Hessian阵负定, 概率也为 $\frac{1}{2^N}$ 。因此, 最有可能的是本征值有正有负, 也就是鞍点的情形。但是即使是普通的鞍点, 函数不会震荡, 而是马上逃离。实际的情形是, 函数收敛到了一个大的平坦区域, 其表现就是一阶导数很小, 以至于在训练结束的时候还没有逃离出这个平坦区域。那么, 我们得从数学上讨论这种平坦区域(平坦马鞍面)具有什么样的性质。可以看下面一个方程:

$$f(x, y) = \frac{\alpha}{2}x^2 - \frac{\beta}{2}y^2$$

$$f_x = \alpha x; f_y = -\beta y.$$
 假设 $\alpha > 0, \beta > 0$

(0, 0)点是唯一的鞍点。在x方向, 该点是局部最小值, 但在y方向, 是局部极大值, 如果 β 很小, 很小是它与学习率r相乘结果与1来说的。假设学习率是r一开始 $y = \Delta_0$, 则一次迭代后

$$\Delta_1 = \Delta_0 + r * \beta * \Delta_0 = (1 + r\beta)\Delta_0$$

$$\Delta_2 = \Delta_1 + r * \beta * \Delta_1 = (1 + r\beta)^2\Delta_0$$

n次迭代后:

如果训练总的次数为N, $r\beta \leq \frac{1}{N}$, 那N次迭代之后 $(1 + r\beta)^N\Delta_0 \leq (1 + \frac{1}{N})^N \approx e\Delta_0 = 2.71828\Delta_0$, 这定量的给出了在训练结束的时候, 函数能多大程度的逃离(0, 0)点。

这个模型可以推广到N维情形。假设Hessian阵有N-K个本征值非负, K个小于零。

考虑通过平移后, 鞍点处于(0, 0...0)点附近的二阶展开:

$$L(X) = L(0) + \sum_{i=K+1}^{i=N} \frac{\alpha_i}{2}x_i^2 - \sum_{i=1}^{i=K} \frac{\alpha_i}{2}x_i^2$$

如果满足所有的 $\alpha_i \ll 1, i \in [1, K]$, 这时候, 函数会在很长时间内, 局域在鞍点附件。

可以考虑的时, 怎么通过设计Cost Function来避免产生这些超级马鞍面。实际上是通过带冲量的方法来加速收敛。

这些超级马鞍面的产生要考虑激活函数吗, Relu比Sigmoid不容易产生这些马鞍面? 参考[知乎文章](#)

线性与非线性方程组解的稳定性分析

最速下降法中, Hessian矩阵的条件数决定了系统的收敛速度。这一节, 本来是自己想通过条件数来分析L-M算法解的稳定性的。

线性方程组解的稳定性分析

系统解的稳定性定义成系统小的扰动对系统解的影响, 。

Example 1:

$$\begin{bmatrix} 400 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

Solution:

$$x_1 = -100, x_2 = -200$$

If we give small change to matrix A, change A_{11} from 400 to 401 then:

$$\begin{bmatrix} 401 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

This time the solution is:

$$x_1 = 40000, x_2 = 79800$$

Ill-conditioned:

When the solution is highly sensitive to the values of the coefficient matrix A or the righthand side constant vector b, the equations are called to be ill-conditioned.

Condition Number

Let's linear equations:

$$Ax = b,$$

Let us investigate first, how a small change in the b vector changes the solution vector. x is the solution of the original system and let $x + \Delta x$ is the solution when b changes from b to $b + \Delta b$

Then we can write:

$$A(x + \Delta x) = b + \Delta b$$

or

$$Ax + A\Delta x = b + \Delta b$$

But because $Ax = b$, it follows that

$$A\Delta x = \Delta b$$

So:

$$\delta \mathbf{x} = \mathbf{A}^{-1} \Delta \mathbf{b}$$

Using the matrix norm properties:

$$\|\mathbf{Ax}\| \leq \|\mathbf{A}\| \|\mathbf{x}\|$$

We can get:

$$\|\mathbf{A}^{-1} \Delta \mathbf{b}\| \leq \|\mathbf{A}^{-1}\| \|\Delta \mathbf{b}\|.$$

Also, we can get:

$$\|\mathbf{b}\| = \|\mathbf{Ax}\| \leq \|\mathbf{A}\| \|\mathbf{x}\|$$

Using equations 2.5 and 2.6 we can get:

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\|\mathbf{A}^{-1}\| \cdot \|\Delta \mathbf{b}\|}{\|\mathbf{b}\|}$$

Let's define condition number as: $\mathbf{K}(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|$

we can rewrite equation 2.7 as:

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} \leq \mathbf{K}(\mathbf{A}) \cdot \frac{\|\Delta \mathbf{b}\|}{\|\mathbf{b}\|}$$

Now, let us investigate what happens if a small change is made in the coefficient matrix

A. Consider \mathbf{A} is changed to $\mathbf{A} + \Delta \mathbf{A}$ and the solution changes from \mathbf{x} to $\mathbf{x} + \Delta \mathbf{x}$

$$(\mathbf{A} + \Delta \mathbf{A})(\mathbf{x} + \Delta \mathbf{x}) = \mathbf{b},$$

we can obtain:

$$\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x} + \Delta \mathbf{x}\|} \leq \mathbf{K}(\mathbf{A}) \cdot \frac{\|\Delta \mathbf{A}\|}{\|\mathbf{A}\|}$$

$\mathbf{K}(\mathbf{A})$ is a measure of the relative sensitivity of the solution to changes in the right-hand side vector \mathbf{b} . When the condition number $\mathbf{K}(\mathbf{A})$ becomes large, the system is regarded as being illconditioned.

Matrices with condition numbers near 1 are said to be well-conditioned.

非线性方程组解的稳定性分析

For non-linear system,

最小二乘法求解线性, 非线性方程组

所有的线性方程组或者非线性方程组可以转化成一个最小二乘法问题, 使得拟合的方程与观察值之间的平方和最小。

$$\beta = \operatorname{argmin}_{\beta} S(\beta) = \operatorname{argmin}_{\beta} \sum_{i=1}^m [y_i - f(x_i, \beta)]^2$$

y_i 是观测值, x_i 是已知变量, 一共m组观测值。

当 $f(\mathbf{x}, \beta)$ 是线性方程组时

即 $f(\mathbf{X}, \beta) = \mathbf{A}\beta$

cost function 可以表示如下：

$$L(\beta) = ||\mathbf{Y} - \mathbf{A}\beta||^2$$

其中 $\mathbf{Y} = [y_1, y_2 \dots y_m]^T$

cost function 对 β

$$L(\beta) = \mathbf{Y}^T \mathbf{Y} - 2\mathbf{Y}^T \mathbf{A}\beta + \beta^T \mathbf{A}^T \mathbf{A}\beta$$

求导，

$$-2\mathbf{A}^T \mathbf{Y} + 2(\mathbf{A}^T \mathbf{A})\beta = 0$$

再让导数等于0, 就可以求得解为: β

$$\beta = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{Y}$$

我们接下来主要讨论非线性的情况：

也就是残差 $y_i - f(x_i, \beta)$ 不能表示成线性形式。

当然我们有很多方法来求解方程(4.1), 比如 梯度下降法, 牛顿法, 高斯-牛顿法, 以及 Levenberg-Marquardt 梯度下降法

把损失函数展开到一阶。

$$f(x + \alpha \mathbf{d}) = f(x_0) + \alpha f'(x_0) \mathbf{d} + O(\alpha^2)$$

$$x_{t+1} = x_t - \alpha f'(x_t),$$

牛顿法

把 cost function 进行展开到二阶：

$$f(x_{t+1}) = f(x_t) + g(x_{t+1} - x_t) + \frac{1}{2}(x_{t+1} - x_t)^T H(x_{t+1} - x_t)$$

求导, $\frac{\partial f}{\partial x_t} = g + H(x_{t+1} - x_t)$, 让导数为0就有

$$x_{t+1} = x_t - H^{-1}g$$

要是 H 是正定的, 上面的就是凸函数, 也就一定有了最小值。可惜 H 不一定是正定的, 这就引导出了下面的方法

高斯-牛顿法

cost function 可以表示成残差的形式：

$$L(x) = \sum_{i=1}^m r_i(x)^2$$

根据牛顿法：

$$v x_{t+1} = x_t - H^{-1}g,$$

梯度表示为：

$$g_j = 2 \sum_i r_i \frac{\partial r_i}{\partial x_j}$$

对方程(4.9)求二阶导，我们可以得到Hessian矩阵：

$$H_{jk} = 2 \sum_i \left(\frac{\partial r_i}{\partial x_j} \frac{\partial r_i}{\partial x_k} + r_i \frac{\partial^2 r_i}{\partial x_j \partial x_k} \right),$$

当残差 r_i 很小的时候，我们就可以去掉最后一项，因此

$$H_{jk} \approx 2 \sum_i J_{ij} J_{ik} \quad \text{With} \quad J_{ij} = \frac{\partial r_i}{\partial x_j}$$

这样Hessian就是半正定了。

因此参数迭代公式(4.10)可以写成：

$$x_{t+1} = x_t - (J^T J)^{-1} J^T r,$$

Levenberg-Marquardt

方法用于求解非线性最小二乘问题，结合了梯度下降法和高斯-牛顿法。

其主要改变是在Hessian阵中加入对角线项。当然，这是一种L2正则化方式。

$$x_{t+1} = x_t - (H + \lambda I_n)^{-1} g,$$

$$x_{t+1} = x_t - (J^T J + \lambda I_n)^{-1} J^T r$$

L-M算法的不足点。

当 λ 很大时， $J^T J + \lambda I_n$ 根本没用，Marquardt为了让梯度小的方向移动的快一些，来防止小梯度方向的收敛，把中间的单位矩阵换成了 $\text{diag}(J^T J)$ ，因此迭代变成：

$$x_{t+1} = x_t - (J^T J + \lambda \text{diag}(J^T J))^{-1} J^T r$$

阻尼项(damping parameter) λ 的选择，Marquardt 推荐一个初始值 λ_0 与因子 $v > 1$ ，开始时，

$\lambda = \lambda_0$ ，然后计算cost functions，第二次计算 $\lambda = \lambda_0/v$ ，如果两者cost function都比初始点高，然后我们就增大阻尼项，通过乘以 v ，直到我们发现当 $\lambda = \lambda_0 v^k$ 时，cost function下降。

如果使用 $\lambda = \lambda_0/v$ 使得cost function下降，然后我们就把 λ_0/v 作为新的 λ

当 $\lambda = 0$ 时，就是高斯-牛顿法，当 λ 趋于无穷时，就是梯度下降法。如果使用 λ/v 没有使损失函数下降，使用 λ 导致损失函数下降，那么我们就继续使用 λ 做为阻尼项。

归一化的残差

如果我们假设数据中每一点的贡献是等权重，因此，我们有必要对每一项残差加一个权重，来归一化每一项残差，这样做的目的是未来防止某些点(离群点，异常值点)的偏差过大，从而对模型结果造成显著影响，我们才假设每一点的误差贡献是一样的。因此公式(4.9)可以变成，cost function可以表示成残差的形式：

$$L(x) = \sum_{i=1}^m \lambda_i r_i(x)^2 = \sum_{i=1}^m \frac{1}{y_i^2} (y_i - f_i(x_i))^2$$

此时, Jacobia变成

$$H_{jk} \approx 2 \sum_i J_{ij} J_{ik} \quad \text{With} \quad J_{ij} = y_i \frac{\partial r_i}{\partial x_j}$$

$$J_S = \Sigma J, \Sigma_{ij} = \frac{1}{y_i} \delta_{ij}$$

Levenberg-Marquardt算法与L2正则化等价

我们在cost function加上L2正则项

$$L(\beta) = \sum_{i=1}^m [y_i - f(x_i, \beta)]^2 + \lambda \|\beta\|^2$$

可以表示成:

$$L(\beta) = L(\beta_0) + (\beta - \beta_0)^T \nabla_{\beta} L(\beta_0) + \frac{1}{2} (\beta - \beta_0)^T H (\beta - \beta_0) + \lambda \|\beta\|^2 + O(\beta^3)$$

求一阶导数:

$$\nabla_{\beta} L(\beta) = L(\beta_0) + (\beta - \beta_0)^T g + \frac{1}{2} (\beta - \beta_0)^T H (\beta - \beta_0) + \frac{1}{2} \lambda \|\beta\|^2 + O(\beta^3)$$

令其为0:

$$\frac{\partial L(\beta)}{\partial \beta} = g + H(\beta - \beta_0) + \lambda \beta = 0$$

就得到:

$$\beta = (H + \lambda I_n)^{-1} H \beta_0 - (H + \lambda I_n)^{-1} g,$$

当 λ 相对于H的本征值来说较小时, 则 $(H + \lambda I_n)^{-1} H = I$, 因此上式可以化为:

$$\beta = \beta_0 - (H + \lambda I_n)^{-1} g,$$

也就是通常的L-M算法。

第五章 机器学习

这章主要讨论传统的机器学习方法，涉及到分类，回归，聚类，降维，决策树与集成学习，图模型以及模型验证。本章序言要对这章介绍的方法起介绍作用，要通过语言使读者明白算法的原理，推到步骤以及算法之间的联系。完成理论的写作之后，你需要明白在实际问题中灵活运用理论或算法是更重要的一件事情。能否灵活的运用理论，创造性的发展理论靠的就是你的实战经验与理论功底。

机器学习原理

机器学习可以从多种视角去分析，比如从最优化视角，概率视角，贝叶斯视角等。本书尽力以最优化的视角来考虑机器学习问题与算法，当然，本书还远不完善，还在升级迭代的过程中。当前是尽力理解这三种视角的有点与局限性，也就是这三种方法的边界，以及与机器学习的交集，而后形成自己的认知。

模型分析

任何有监督的学习都可以通过偏差，方差，噪声来对模型进行分析。通过减小这三个指标中任何一项(除噪声，因为这是数据本身的问题，garbage in, garbage out)，都能提高模型的准确性。最典型的就是基于Bagging类方法能降低方差，基于Boosting类方法能降低偏差。

Bagging适合于High Variance & Low Bias 的模型。

Boosting适合于High Bias& Low Variance 的模型。

偏差，方差，噪声

偏差-方差分解可以用来对学习算法的期望泛化错误率进行拆分。同时，偏差-方差分解为我们设计模型与分析提供了一个比较清晰的方向。

假设测试样本是 \mathbf{x} ，令 y_D 为 \mathbf{x} 在数据集上的标记(可能存在标记错误的情况)， y 为 \mathbf{x} 的真实标记， $f(\mathbf{x}; D)$ 为训练集D上学得模型f在 \mathbf{x} 上的预测输出，以回归模型为例，学习算法的期望预测为：

$$\hat{f}(\mathbf{x}) = E_D[f(\mathbf{x}, D)]$$

方差是：

$$var(\mathbf{x}) = E_D[f(\mathbf{x}, D) - \hat{f}(\mathbf{x})]^2$$

噪声是：

$$\epsilon^2 = E_D[y_D - y]^2$$

输出期望与真实标记之间的差别成为偏差(Bias)，即：

$$bias^2(\mathbf{x}) = E_D[\hat{f}(\mathbf{x}) - y]^2$$

为方便讨论，假设噪声的期望为0；即： $E_D[y_D - y] = 0$ 。下面来对模型的期望泛化误差进行分解：

$$\begin{aligned} E(f; D) &= E_D[(f(\mathbf{x}, D) - y_D)^2] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}) + \hat{f}(\mathbf{x}) - y_D)^2] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y_D)^2] + E_D[2(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))(\hat{f}(\mathbf{x}) - y_D)] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y_D)^2] \end{aligned}$$

$$\begin{aligned}
&= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y + y - y_D)^2] \\
&= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y)^2] + E_D[(y - y_D)^2] + E_D[2(\hat{f}(\mathbf{x}) - y)(y - y_D)] \\
&= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y)^2] + E_D[(y - y_D)^2]
\end{aligned}$$

因此就得到：

$$E(f; D) = bias^2(\mathbf{x}) + var(\mathbf{x}) + \epsilon^2$$

也就是泛化误差分成偏差, 方差与噪声之和。我们设计模型分析时, 可以从这三方面去考虑。

噪声

为了消除噪声的影响, 我们需要对数据进行清洗, 进行预处理。在很大程度上就是为了得到更干净的数据。

Bias

偏差小, 说明的是模型很准, 可能有过拟合的倾向, 增加模型的复杂度可以使得bias减小。但是泛化能力变差, 也就是variance会大, 模型对数据很敏感。如果模型bias大, 可以通过对简单的模型进行boost, 来提升模型的准确性, 也就是Boost系列算法做的事情, 如: AdaBoost, GBDT, XGBoost。

方差

方差小, 说明模型很稳定, 也就是说模型的泛化能力好, 可能测试集上的数据相对于训练集来说有一些变化, 但是也不影响模型的输出结果, 此时, 可能模型欠拟合, 因此泛化能力好, 最典型的就是决策树桩, 它的方差小, 但是是欠拟合的。如果模型variance大, 可以通过训练多个模型, 并让各个模型之间的关联性很小, 通过求平均来减小Variance, 这就是集成模型中bagging系列算法做的事情, 如Bagging, Random Forest.

最大似然函数

输入: $X \in R^n$

输出: $Y \in (1, 2, \dots, K)$

条件概率: $P(X = x | Y = C_k) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \dots, X^{(n)} = x^{(n)} | Y = C_k)$, k=1,2,...,K

假设有N个样本点(X_i, Y_i), $i = 1, 2, \dots, N$

条件概率: $P(Y = y | X = x, \theta)$, 其中 θ 是模型的参数。

似然函数定义为: 每个样本点发生概率的乘积:

$$L(\theta) = \prod_{n=1}^N P(Y = Y_n | X = X_n, \theta)$$

我们需要求得一个模型，使得在所有样本点在该模型发生的几率极大，几率是联合几率，是每个样本发生几率的乘积。也就是极大化似然函数。

分类与回归问题似乎都可以转化成求解似然函数。

用极大似然函数求解回归问题

输入： $\mathbf{X} \in R^n$

输出： $Y \in R$

我们的模型是 $f(\mathbf{X}, \Theta)$ ，其中 Θ 是模型参数， \mathbf{X} 是输入变量。

假设我们的损失函数是平方误差，因此我们的cost function可以表示如下：

$$L(\mathbf{X}, \Theta) = \frac{1}{2} \sum_{n=1}^N (f(X_n, \Theta) - Y_n)^2$$

等价于：

$$e^{L(\mathbf{X}, \Theta)} = \prod_{n=1}^N \frac{1}{2} (f(X_n, \Theta) - Y_n)^2 = \prod_{n=1}^N e^{\frac{1}{2}(f(X_n, \Theta) - Y_n)^2}$$

实际上，最小化 $L(\mathbf{X}, \Theta)$ 等价于最小化 $e^{L(\mathbf{X}, \Theta)}$ ，等价于最大化 $e^{-L(\mathbf{X}, \Theta)}$ ，也就是等价于最大化

$$\Gamma(\Theta) = \prod_{n=1}^N e^{-\frac{1}{2}(f(X_n, \Theta) - Y_n)^2} = \prod_{n=1}^N P(X_n, \Theta)$$

其中

$$P(X_n, \Theta) = e^{-\frac{1}{2}(f(X_n, \Theta) - Y_n)^2}$$

定义成回归模型中样本 (X_n, Y_n) 发生的几率，这样我们就把回归问题与几率联系起来了。我们把求解损失函数最小，转化成极大化似然函数的求解。

对于不同的损失函数，我们都可以转化成对应的概率。

对于分类问题，怎么用一个统一的框架来解析？可以是基于概率的。

用极大似然函数求解多类分类问题

在这里我们以多项逻辑斯蒂回归为例，讨论怎么用最大化似然函数来求解这一类问题。

假设离散性随机变量 Y 的取值是 $(1, 2, \dots, K)$ ，输入变量 $\mathbf{X} \in R^n$

则，模型如下：

$$P(Y = k | \mathbf{x}) = \frac{\exp(\mathbf{w}_k \cdot \mathbf{x})}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}, k = 1, 2, \dots, K - 1$$

$$P(Y = K | \mathbf{x}) = \frac{1}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}$$

这里 $\mathbf{x} \in \mathbf{R}^{n+1}$ ； $\mathbf{w}_k \in \mathbf{R}^{n+1}$ 。

因此，总的似然函数可以表示为：

$$L(\mathbf{W}) = \prod_{n=1}^N P(Y = Y_n | X = X_n)$$

这样，我们需要求的参数是 $W = (\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_{n+1})$.

用极大似然与SVM来做多类分类问题

核心，定义出到超平面的距离，每一个类，一个超平面，然后定义距离，然后转化成概率，最终转化成极大似然函数求解。不过已经有了基于逻辑回归与极大似然来求解多分类的问题，因此已经建立起基于极大似然来解释分类与回归的问题，故是否还有必要基于极大似然与SVM来做多分类问题？其实没有必要。

麦克斯韦妖与信息熵

不存在一个监测系统，能区分粒子的速度而对粒子做出区分，使得原本温度相同的系统，自发的形成高温与低温子系统。因为在区分例子速度时，就是一个信息处理的过程，使得系统的信息熵减小。那么，怎么根据熵增原理，确定系统的熵减数量与检测系统需要付出的信息熵的数量，或者能量。

最大熵原理

极大似然与最大熵原理的等价性

对有限个状态的问题，可以建立起极大似然与最大熵原理之间的联系。推广到连续状态问题，先处理一维问题。假设后验分布是 $P(\mathbf{x}|\theta)$ 根据最大熵原理，需要满足：

$$E(\theta) = - \int P(\mathbf{x}|\theta) \ln P(\mathbf{x}|\theta)$$

$$\theta = \arg \max_{\theta} E(\theta)$$

需要加上这一节。

奥卡姆剃刀

Evaluation Metrics

二分类：AUC,

多分类：交叉熵(log loss)

回归：均方差

For any kind of machine learning problem, we must know how we are going to evaluate our results, or what the evaluation metric or objective is. For example in case of a skewed binary classification problem we generally choose area under the receiver operating characteristic

curve (ROC AUC or simply AUC). In case of multi-label or multi-class classification problems, we generally choose categorical cross-entropy or multiclass log loss and mean squared error in case of regression problems.

No FREE Lunch定理, 先验, 正则化

NO FREE LUNCH



图 1.4 没有免费的午餐. (黑点: 训练样本; 白点: 测试样本)

黑点是训练样本，白点是测试样本点。根据训练样本，我们得到模型A, B。如果测试点不同，则A, B的相对表现不同。问题的关键在于，我们先验的假设了(测试)样本点在所有空间是均匀分布的，因此，在训练集上训练出的所有模型的效果是一样的，因为你总有样本集是落在你模型预测的曲线之上的。

因此，如果样本点在整个数据空间是均匀分布的前提下，则不存在一个模型或算法在所有问题上比其它算法更优越。

但是现实遇到的问题是，数据的分布是受限制的，数据存在一个先验分布，因此会存在一些模型比其它模型更有效。这里NFL不再有是因为问题的前提--数据是均匀分布的--不再成立。因此，脱离场景谈方法是无效的，我们只能针对特定的问题，寻找最佳的优化方法。

正则化

没有免费的午餐定理告诉我们没有一个模型是对所有的样本是最优的¹，因此，对于特定的样本，我们需要把先验加进模型中(也就是损失函数中)，这个先验就是我们对这个样本的知识，比如样本数据的分布，样本的均值或者方差等(统计物理学中的系统的能量可以看成一种均值，统计物理学中有很多启发性的模型)。加到模型中的就是我们的正则项，正则项就是一个先验，是我们需要模型满足的约束。

一个问题是，正则化只能使我们的模型更稳定，不会更精确，那先验怎么作为正则化的形式加进模型，使得我们的模型更优越呢？

先验分布与熵值。

没有先验知识，样本在参数空间是均匀分布的，因此熵值很大；如果有强的先验，因此样本局限在参数空间的特定区域，因此系统具有的熵值比较小。

先验分布与损失函数。

知道预测误差的先验分布，才能选择合适的损失函数。知其然，才能知其所以然。

交叉熵

平方损失：

0-1损失：分类问题，

Log损失：

Hinge损失：

指数损失：

平方损失：等价于误差服从高斯分布的极大似然估计。适用于回归问题。

麦哈顿距离损失等价于误差服从拉普拉斯分布的极大似然估计。

0-1损失相当于误差服从

数据的误差的先验分布(统计学派观点？)：极大似然

模型参数的先验分布(贝叶斯观点)：正则化是模型复杂度的惩罚函数，真理是美的，简洁的。你可以自己定义模型的复杂度？问题是怎麽定义模型复杂度？(正则化是要求模型参数只能在0附近，但是现实可能是，真实的模型参数不在0附件，贝叶斯观点并不能与模型参数误差的先验分布联系起来，因为正则化假定模型参数的均值皆为0，但是贝叶斯观点并不要求这样。)

损失函数	预测误差的先验分布
平方损失	高斯分布
曼哈顿距离损失	拉普拉斯分布
0-1损失	二项分布

¹. Ian Goodfellow 《deep learning》 ↵

分类

分类的方法有

1. SVM
2. Logistic回归
3. K近邻
4. LDA(线性判别器)
5. 分类树

SVM, KKT条件与核函数方法

SVM的本质就是利用对偶方法求解带约束的凸优化问题。KKT条件保证了可以通过求解对偶问题来求解原始问题。要理解的是怎么利用对偶方法求解优化问题，这是求解优化问题的很重要的一种方法，SVM只是这种方法的一个典型例子。

核函数的作用的通过非线性变换，使得SVM可以用来处理非线性分类问题。

SVM

线性可分二分类问题

一组数据 $(x_i, y_i), i = 1, 2 \dots N, y_i \in +1, -1, x_i \in R^m$

SVM 算法是为了求得一个分割超平面 $wx + b = 0$ 使得所有的样本点到超平面的几何距离都不小于 γ , 且这个 γ 是最大的。

首先我们定义几何距离和函数距离:

几何距离: $\gamma_i = y_i(\frac{wx_i}{\|w\|} + \frac{b}{\|w\|})$

除以 $\|w\|$ 是因为 x, b 同时乘以一个因子, 超平面不变。

$$K(x_i, x_j) = \phi(x_i)\phi(x_j)$$



函数间距: $\gamma_i = y_i(wx_i + b)$

我们要使: $\gamma_i \geq \gamma$. 对任何的样本点 i 都成立, 并求得 γ 的最大值。故上面的问题可以用如下数学来刻画:

$$\max_{w,b} \gamma$$

$$\text{s.t. } y_i(\frac{wx_i}{\|w\|} + \frac{b}{\|w\|}) \geq \gamma$$

转化为函数间距:

$$\max_{w,b} \gamma$$

$$\text{s.t. } y_i(\mathbf{w}\mathbf{x}_i + b) \geq ||\mathbf{w}||\gamma = \hat{\gamma}$$

为了简化计算，我们取 $\hat{\gamma} = 1$

因此问题转化为：

$$\text{s.t. } \max_{\mathbf{w}, b} \frac{1}{||\mathbf{w}||}$$

$$\text{s.t. } y_i(\mathbf{w}\mathbf{x}_i + b) \geq 1, i = 1, 2, \dots, N$$

转化为求极小问题：

$$\min_{\mathbf{w}, b} \frac{1}{2} ||\mathbf{w}||^2$$

$$\text{s.t. } y_i(\mathbf{w}\mathbf{x}_i + b) \geq 1, i = 1, 2, \dots, N$$

问题转化成带约束的优化问题(在这里是凸优化问题)。带等式的约束问题可以通过引入拉格朗日乘子求解，带不等式约束的问题可以引入KKT乘子求解。下面一节我们讨论带约束的凸优化问题。

线性不可分与软间隔最大化

$$\min_{\mathbf{w}, b} \frac{1}{2} ||\mathbf{w}||^2 + C \sum_{i=1}^N \xi_i$$

$$\text{s.t. } y_i(\mathbf{w}\mathbf{x}_i + b) \geq 1 - \xi_i, i = 1, 2, \dots, N$$

$$\xi_i \geq 0, i = 1, 2, \dots, N$$

通过构造拉格朗日函数求对偶函数。原问题的解是 $(\mathbf{w}^*, \xi^*, b^*)$ ，对偶问题的解是 α^* 。

参考文献：

1. July · pluskid, 支持向量机通俗导论：

<https://raw.githubusercontent.com/liuzheng712/Intro2SVM/master/Intro2SVM.pdf>

Karush-Kuhn-Tucker(KKT)条件

1. 先讨论一般带约束优化问题的数据描述
2. 然后引入其作用约束与不起作用约束的定义
3. 再引入正则点的定义
4. 接下来引入KKT条件 5. 针对局部极小问题，我们考虑优化函数的二阶导数问题。

一般形式的优化问题：

$$\begin{aligned} & \text{minimize } f(\mathbf{x}) \\ & \text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \text{s.t. } \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

其中： $f: R^n \rightarrow R$, $\mathbf{h}: R^n \rightarrow R^m$, $m \leq n$; $\mathbf{g}: R^n \rightarrow R^p$;

针对这一问题，引入如下定义：

定义21.1. 对于一个不等式约束 $g_i(x) \leq 0$, 如果在 x^* 处, $g_i(x^*) = 0$, 那么称该不等式约束是 x^* 处的起作用约束。如果 $g_i(x^*) < 0$, 那么称该不等式约束是 x^* 处的不起作用约束。

定义21.2. 设 \mathbf{x}^* 满足 $\mathbf{h}(\mathbf{x}^*) = \mathbf{0}$, $\mathbf{g}(\mathbf{x}^*) \leq \mathbf{0}$, 设 $J(\mathbf{x}^*)$ 为起作用不等式约束下标集,

$$J(\mathbf{x}^*) = (j : g_j(\mathbf{x}^*) = 0)$$

如果： $\nabla h_i(\mathbf{x}^*)$, $\nabla g_j(\mathbf{x}^*)$, $i \leq i \leq m$, $j \in J(\mathbf{x}^*)$

是线性无关的，则称 \mathbf{x}^* 是一个正则点。

KKT条件：某个点是局部极小点所满足的一阶必要条件。

定理：KKT条件，设 $f, \mathbf{g}, \mathbf{h} \in C^1$, 设 \mathbf{x}^* 是问题 $\mathbf{h} = \mathbf{0}$, $\mathbf{g} \leq \mathbf{0}$ 的一个正则点和局部极小点，那么必然存在 $\lambda^* \in R^m$ 和 $\mathbf{u}^* \in R^p$ 使得以下条件成立：

$$\mathbf{u}^* \geq \mathbf{0}$$

$$Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$$

$$\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$$

λ^* 为拉格朗日乘子向量， \mathbf{u}^* 是KKT乘子向量，其元素分别成为拉格朗日乘子，KKT乘子。

充分条件

上面讲了局部极小的必要条件，这里我们讨论局部极小的充分条件。然后我们就利用KKT条件去求不等式约束问题。

二阶充分必要条件；

定义如下矩阵：

$$L(\mathbf{x}, \lambda, \mathbf{u}) = F(\mathbf{x}) + \lambda \mathbf{H}(\mathbf{x}) + \mathbf{u} \mathbf{G}(\mathbf{x})$$

$F(\mathbf{x})$ 是 f 在 \mathbf{x} 处的Hessian矩阵；

$$\lambda \mathbf{H}(\mathbf{x}) = \lambda_1 H_1(\mathbf{x}) + \dots + \lambda_m H_m(\mathbf{x})$$

$$\mathbf{uG}(\mathbf{x}) = u_1 G_1(\mathbf{x}) + \dots + u_p G_p(\mathbf{x})$$

其中 $G_k(\mathbf{x})$ 是 g_k 处的Hessian矩阵。

起作用约束所定义曲面的切线空间：

$$T(\mathbf{x}^*) = \{y \in R^n : Dh(x^*)y = 0, Dg_j(x^*)y = 0, j \in J(x^*)\}$$

定理：二阶必要条件：如果 \mathbf{x}^* 是上面讨论的优化问题的极小点，那么存在 $\lambda^* \in R^m, \mathbf{u}^* \in R^p$ 使得：

1. $\mathbf{u}^* \geq 0,$

2. $Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$

3. $\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$

4. 对所有 $\mathbf{y} \in T(\mathbf{x}^*, \mathbf{u}^*), \mathbf{y} \neq 0$, 都有 $\mathbf{y}^T L(\mathbf{x}, \lambda, \mathbf{u}) \mathbf{y} > 0$

定理：二阶充分条件：

假设 $f, \mathbf{g}, \mathbf{h} \in C^2, \mathbf{x}^* \in R^n$ 是一个可行点，存在向量 $\lambda^* \in R^m, \mathbf{u}^* \in R^p$ 使得：

1. $\mathbf{u}^* \geq 0,$

2. $Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$

3. $\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$

4. 对所有 $\mathbf{y} \in T(\mathbf{x}^*, \mathbf{u}^*), \mathbf{y} \neq 0$, 都有 $\mathbf{y}^T L(\mathbf{x}, \lambda, \mathbf{u}) \mathbf{y} > 0$

那么 \mathbf{x}^* 是优化问题

s.t. $\mathbf{h}(\mathbf{x}) = \mathbf{0}$

s.t. $\mathbf{g}(\mathbf{x}) \leq \mathbf{0}$

的严格局部极小点，

$$T(\mathbf{x}^*) = \{y \in R^n : Dh(x^*)y = 0, Dg_j(x^*)y = 0, j \in J(x^*)\}$$

其中： $J(\mathbf{x}^*, \mathbf{u}^*) = (j : g_j(\mathbf{x}^*) = 0), \mathbf{u}^* > 0$

学习的对偶算法

对于求极小问题：

$$\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2$$

s.t. $y_i(\mathbf{w} \mathbf{x}_i + b) \geq 1, i = 1, 2, \dots, N$

引入拉格朗日函数，对每个不等式约束引入拉格朗日乘子 $\alpha_i \geq 0, i = 1, 2, 3..., N$:

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^N \alpha_i y_i (\mathbf{w} \cdot \mathbf{x}_i + b) + \sum_{i=1}^N \alpha_i$$

其中 $\alpha = (\alpha_1, \alpha_2, ..., \alpha_N)^T$ 为拉格朗日乘子向量。

根据拉格朗日对偶性，原始问题的对偶问题是极大极小问题：

$$\max_{\alpha} \min_{\mathbf{w}, \mathbf{b}} L(\mathbf{w}, \mathbf{b}, \alpha)$$

现对 \mathbf{w}, \mathbf{b} 求极小，再对 α 求极大。

(1) 求 $\min_{\mathbf{w}, \mathbf{b}} L(\mathbf{w}, \mathbf{b}, \alpha)$ 。对 \mathbf{w}, \mathbf{b} 求导并令其为0，得到：

$$\begin{aligned}\frac{\partial L(w, b, \alpha)}{\partial w} &= \mathbf{w} - \sum_{i=1}^N \alpha_i y_i (\mathbf{x}_i) = 0 \\ \frac{\partial L(w, b, \alpha)}{\partial b} &= \sum_{i=1}^N \alpha_i y_i = 0\end{aligned}$$

把上面两个式子代入拉格朗日函数，可以得到原始函数的对偶函数，对偶函数只是拉格朗日乘子的函数：

$$\min_{\mathbf{w}, \mathbf{b}} L(\mathbf{w}, \mathbf{b}, \alpha) = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) + \sum_{i=1}^N \alpha_i$$

(2) 求 $\min_{\mathbf{w}, \mathbf{b}} L(\mathbf{w}, \mathbf{b}, \alpha)$ 对 α 的极大即是对偶问题。

$$\max_{\alpha} -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) + \sum_{i=1}^N \alpha_i$$

$$\text{s.t. } \sum_{i=1}^N \alpha_i y_i = 0$$

$$\alpha_i \geq 0, i=1, 2, \dots, N.$$

由于原始问题满足KKT条件，所以存在 w^*, α^* 使得 w^* 是原始问题的解， α^* 是对偶问题的解。

定理：由对偶问题求原始问题

假设 $\alpha^* = (\alpha_1^*, \alpha_2^*, \dots, \alpha_N^*)^T$ 是对偶问题的解，则存在一个指标 j 使得 $\alpha_j > 0$ ，使得原问题的解 \mathbf{w}^*, b 可由下面方程给出：

$$\begin{aligned}\mathbf{w}^* &= \sum_{i=1}^N \alpha_i^* y_i (\mathbf{x}_i) \\ b^* &= y_j - \sum_{i=1}^N \alpha_i^* y_i (\mathbf{x}_i \cdot \mathbf{x}_j)\end{aligned}$$

由此得到分离超平面：

$$\mathbf{w}^* \mathbf{x} + b^* = 0$$

分类决策函数：

$$\begin{aligned}f(x) &= \text{sign}(\mathbf{w}^* \mathbf{x} + b^*) \\ &= \text{sign}(\sum_{i=1}^N \alpha_i^* y_i (\mathbf{x}_i \cdot \mathbf{x}) + b^*)\end{aligned}$$

线性不可分与软间隔最大化

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i \\ \text{s.t.} \quad & y_i (\mathbf{w} \mathbf{x}_i + b) \geq 1 - \xi_i, i = 1, 2, \dots, N \\ & \xi_i \geq 0, i = 1, 2, \dots, N \end{aligned}$$

通过构造拉格朗日函数求对偶函数。原问题的解是 (w^*, ξ^*, b^*) , 对偶问题的解是 α 。

实际上与线性可分的求法是一致的, 只是对应的 ξ 会多出一个拉格朗日乘子, 在求解对偶问题的优化时, 多了一个约束。 \mathbf{w}^* 是唯一的, b 不唯一。

核函数方法

引入核函数

其实在我们求解对偶问题的最优化问题时，我们只需要求 $x_i x_j$,因此如果存在一种变换，使得：

$$\mathbf{x} \rightarrow \phi(\mathbf{x})$$

且

$$x_i x_j \rightarrow \phi(x_i) \phi(x_j) = K(x_i, x_j)$$

其实我们不需要显示的给出 $\phi(\mathbf{x})$ 的表达式，只需要知道 $K(x_i, x_j)$ 就可以了， $K(x_i, x_j)$ 就是核函数。
引入核函数，是因为我们可以利用核函数方法来解决非线性分类问题。

正定核函数

正定核的充要条件

设 $K: (X, X) \rightarrow \mathbb{R}$ 上的对称函数，则 $K(x, z)$ 为正定核函数的充要条件是对任意 $x_i \in X, i=1, 2, \dots, m, K(x, z)$ 对应的Gram矩阵：

$K = [K(x_i, x_j)]_{m \times m}$
的半正定矩阵。

常用的核函数

1. 线性核: $K(\mathbf{x}, \mathbf{z}) = \mathbf{x}_i^T \mathbf{x}_j$
2. 多项式核函数: $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x} \cdot \mathbf{z} + 1)^p$

分类决策函数是：

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^N \alpha_i^* y_i (\mathbf{x}_i^T \mathbf{x} + 1)^p + b^*\right)$$

1. 高斯核函数

$$K(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|^2}{2\sigma^2}\right)$$

分类决策函数是：

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^N \alpha_i^* y_i \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|^2}{2\sigma^2}\right) + b^*\right)$$

2. 拉普拉斯核函数

$$K(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|}{\sigma}\right) \sigma > 0$$

3. Sigmoid核函数

$$K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\mathbf{x}_i^T \mathbf{x}_j + \theta) \quad \beta > 0, \theta < 0$$

非线性分类问题

输入: 训练数据 $(x_i, y_i), i = 1, 2, \dots, N, y_i \in \{+1, -1\}, x_i \in R^m$

输出: 分类决策函数

(1) 选择适合的核函数 $K(x, z)$ 和适当的参数 C , 构造并求解优化问题:

$$\min_{\alpha} \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^N \alpha_i$$

$$\text{s.t. } \sum_{i=1}^N \alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C, i=1, 2, \dots, N.$$

求得最优解 $\alpha^* = (\alpha_1^*, \alpha_2^*, \dots, \alpha_N^*)^T$:

(2) 选择一个指标 j 使得 $0 < \alpha_j < C$, 计算

$$b^* = y_j - \sum_{i=1}^N \alpha_i^* y_i K(\mathbf{x}_i, \mathbf{x}_j)$$

(3) 构造分类决策函数:

$$f(x) = \text{sign}\left(\sum_{i=1}^N \alpha_i^* y_i K(\mathbf{x}_i, \mathbf{x}) + b^*\right)$$

逻辑回归

在这里我们以多项逻辑斯蒂回归为例，讨论怎么用最大化似然函数来求解这一类问题。

假设离散性随机变量Y的取值是(1,2,...,K),输入变量 $\mathbf{X} \in R^n$

则，模型如下：

$$P(Y = k|x) = \frac{\exp(\mathbf{w}_k \cdot \mathbf{x})}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}, k = 1, 2, \dots, K - 1$$
$$P(Y = K|x) = \frac{1}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}$$

这里 $\mathbf{x} \in \mathbf{R}^{n+1}$; $\mathbf{w}_k \in \mathbf{R}^{n+1}$.

因此，总的似然函数可以表示为：

$$L(\mathbf{W}) = \prod_{n=1}^N P(Y = Y_n | X = X_n)$$
 这样，我们需要求的参数是 $W = (\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_{n+1})$.

最简单的就是二项逻辑斯蒂回归，在这里 $K=2$: 似然函数可以写成如下简单的形式：

$$L(\mathbf{W}) = \prod_{n=1}^N [\pi(x_i)^{y_i}][1 - \pi(x_i)]^{1-y_i}$$

其中：
 $P(Y = 1|x) = \pi(x) = \frac{\exp(\mathbf{w} \cdot \mathbf{x})}{1 + \exp(\mathbf{w} \cdot \mathbf{x})}$
 $P(Y = 0|x) = 1 - \pi(x)$

当 $y_i = 0$ 时：

$$[\pi(x_i)^{y_i}][1 - \pi(x_i)]^{1-y_i} = 1 - \pi(x_i)$$
 当 $y_i = 1$ 时：

$$[\pi(x_i)^{y_i}][1 - \pi(x_i)]^{1-y_i} = \pi(x_i)$$

实际上分别是 $y_i = 0, y_i = 1$ 发生的概率。

一般连乘形式的似然函数容易发生溢出，故而一般求对数似然函数：

$$L(\mathbf{W}) = \sum_{n=1}^N [y_i \log \pi(x_i) + (1 - y_i) \log[1 - \pi(x_i)]]$$
$$= \sum_{n=1}^N [y_i(w \cdot x_i) + \log(1 + \exp(w \cdot x_i))]$$

求 $L(\mathbf{W})$ 的极大值，得到 \mathbf{w} 的估计值。

问题转化为以对数似然函数为目标函数的最优化问题，一般采用梯度下降法或者拟牛顿法求解。
(为啥不使用牛顿法？因为求Hessian矩阵比较麻烦，而拟牛顿法只需要构造Hessian阵就可以了，用迭代法求)

最大熵模型

最大熵原理

最大熵原理是概率模型学习的一个准则，最大熵原理认为，学习概率模型时，在所有可能的概率模型(分布)中，熵最大的模型是最好的模型。通常用约束条件来确定概率模型的集合。所以，最大熵原理也可以表述为在满足约束条件的模型集合中，选取熵最大的模型。

怎么建立起最大熵原理与最大似然原理之间的联系？

最大熵模型如下：

对于给定的训练数据集： $T = ((x_1, y_1), (x_2, y_2), \dots, (x_N, y_N))$

以及特征函数 $f_i(\mathbf{x}, \mathbf{y}), i = 1, 2, \dots, n$ ，最大熵模型的学习等价于约束最优化问题：

$$\max_{P \in C} H(P) = - \sum_{x,y} \hat{P}(x) P(y|x) \log P(y|x)$$

$$\text{s.t. } E_P(f_i) = E_{\hat{P}}(f_i), i = 1, 2, \dots, n$$

$$\sum_y P(y|x) = 1$$

可以转化为求极小问题：

$$\min_{P \in C} H(P) = - \sum_{x,y} \hat{P}(x) P(y|x) \log P(y|x)$$

$$\text{s.t. } E_P(f_i) - E_{\hat{P}}(f_i) = 0, i = 1, 2, \dots, n$$

$$\sum_y P(y|x) = 1$$

一般，带约束的优化问题可以通过引入拉格朗日乘子与KKT乘子来求解，也就是把原始问题转化为对偶问题来求解。在满足KKT条件时，对偶问题与原始问题等价：

首先引入拉格朗日乘子： w_0, w_1, \dots, w_n ，定义拉格朗日函数 $L(P, \mathbf{w})$ ；

$$\begin{aligned} L(P, \mathbf{w}) &= -H(P) + w_0(1 - \sum_y P(y|x)) + \sum_{i=1}^n w_i(E_P(f_i) - E_{\hat{P}}(f_i)) \\ &= - \sum_{x,y} \hat{P}(x) P(y|x) \log P(y|x) + w_0(1 - \sum_y P(y|x)) \\ &\quad + \sum_{i=1}^n w_i \left(\sum_{x,y} \hat{P}(x, y) f_i(x, y) - \sum_{x,y} \hat{P}(x) P(y|x) f_i(x, y) \right) \end{aligned}$$

最优化的原始问题是：

$$\min_{P \in C} \max_{\mathbf{w}} L(P, \mathbf{w})$$

最优化的对偶问题是：

$$\max_{\mathbf{w}} \min_{P \in C} L(P, \mathbf{w})$$

由于拉格朗日函数 $L(P, \mathbf{w})$ 是 P 的凸函数，因此原始问题与对偶问题的解释等价的。(为什么？用KKT条件能解析吗？)

令： $\Psi(\mathbf{w}) = \min_{P \in C} L(P, \mathbf{w})$

$\Psi(\mathbf{w})$ 称为对偶函数。同时，将其解记为：

$$P_w = \arg \min_{P \in C} L(P, \mathbf{w}).$$

具体计算就是 $L(P, \mathbf{w})$ 对 P 求导，并让其等于0。

$$\begin{aligned}\frac{\partial L(P, \mathbf{w})}{\partial P(y|x)} &= \sum_{x,y} \hat{P}(x) (\log P(y|x) + 1) - \sum_y w_0 - \sum_{x,y} (\hat{P}(x) \sum_{i=1}^n w_i f_i(x, y)) \\ &= \sum_{x,y} \hat{P}(x) (\log P(y|x) + 1 - w_0 - \sum_{i=1}^n w_i f_i(x, y))\end{aligned}$$

令其偏导数等于0，在 $\hat{P}(x) > 0$ 的情况下，解得：

$$P(y|x) = \exp(\sum_{i=1}^n w_i f_i(x, y)) + w_0 - 1$$

由于： $\sum_y P(y|x) = 1$ ，得：

$$P_w(y|x) = \frac{1}{Z_w(x)} \exp(\sum_{i=1}^n w_i f_i(x, y))$$

其中：

$$Z_w(x) = \sum_y \exp(\sum_{i=1}^n w_i f_i(x, y))$$

其中： $Z_w(x)$ 称为规范化因子， $f_i(x, y)$ 称为特征函数， w_i 是特征函数的权重。

其实这些就可以和统计物理学中的波尔兹曼分布联系在一起了，上面的推导也适用于波尔兹曼分布的推导。这也不奇怪，因为最大熵原理的理论就是脱胎于统计物理学。

之后再求解对偶问题的极大化问题：

$$\max_w \Psi(\mathbf{w})$$

将其解记作 w^* ，即：

$$w^* = \arg \max_w \Psi(\mathbf{w})$$

因为最大熵模型是一个带约束的优化问题，因此可任意通过定于拉格朗日函数，通过对变量求极值得到对偶函数，再通过对偶函数来求解。又因为在这里，对偶函数等价于对数似然函数，因此可以求对数似然函数的最大化。

模型学习的最优化算法

我们已知模型的概率分布：

$$P_w(y|x) = \frac{1}{Z_w(x)} \exp(\sum_{i=1}^n w_i f_i(x, y))$$

因此模型的对数似然函数是：

$$\begin{aligned}L(w) &= \sum_{x,y} P(x, y) \sum_{i=1}^n w_i f_i(x, y) - \sum_x P(x) \log Z_w(x) \\ L(w) &= \sum_{x,y} P(x, y) \sum_{i=1}^n w_i f_i(x, y) - \sum_x P(x) \log \sum_y \exp(\sum_{i=1}^n w_i f_i(x, y))\end{aligned}$$

求解上面函数的最大值，可以通过拟牛顿法或者改进的迭代尺度法来求解，当然最典型的求解方法是IIS(Improved Iterative Scaling)。(有时就加上这两个算法)

BFGS

输入：特征函数 f_1, f_2, \dots, f_n ; 经验分布 $P(x,y)$, 目标函数 $f(w)$, 梯度 $g(w) = \nabla f(w)$, 精度要求 ϵ 。

输出：最优化参数 w^* ; 最优模型 $P_{w^*}(y|x)$ 。

(1) 选定初始点 w^0 , 取 B_0 为正定对称函数, 置 $k=0$;

(2) 计算 $g_k = g(w^k)$, 若 $\|g_k\| < \epsilon$, 则停止计算, 得到 $w^* = w^k$; 否则转到(3)

(3) 由 $B_k p_k = -g_k$ 得出 p_k ;

(4) 一维搜索, 求 λ_k 使得:

$$(w^{(k)} + \lambda_k p_k) = \min_{\lambda \geq 0} f(w^{(k)} + \lambda p_k)$$

(5) 置: $w^{(k+1)} = w^{(k)} + \lambda_k p_k$

(6) 计算 $g_{k+1} = g(w^{(k+1)})$, 若 $\|g_{k+1}\| < \epsilon$, 则停止计算, 得 $w^* = w^{(k+1)}$, 否则按下面方式求 B_{k+1} 。

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T \delta_k} - \frac{B_k \delta_k \delta_k^T B_k^T}{\delta_k^T B_k \delta_k}$$

其中:

$$y_k = g_{k+1} - g_k$$

$$\delta_k = w^{(k+1)} - w^{(k)}$$

(7) 置 $k=k+1$, 转到(3)

K近邻

K-近邻不需要训练，因此是一种懒惰学习，急切学习(在训练时就分类)。算法的关键是度量(距离)的定义，K值得选择。

如果样本很大，类别不多，每个类别样本比例比较一致，且不同类别区别比较大的时候，如果给定一个分类错误的上限 ϵ ，我们没有必要遍历所有样本点找到K个近邻。因为，我们可以找到一个需要的样本上限N，N是 ϵ 的函数，然后我们从总的样本中挑选出N个样本，进行K紧邻就可以巨大的减小判别的时间。

K近邻的实现就是遍历所有样本，找出K个最近邻中所属类别最多的类，这就是该样本所属的类。

回归

1. 线性回归
2. Ridge回归
3. Lasso回归
4. 回归树
5. Ordinary Least Squares Regression(OLSR)

回归问题，可以通过最小二乘法来求解，最终是一个优化问题。因此重要的是怎么转化为一个优化问题，并通过优化方法来进行求解。

线性回归中，如果是满秩问题或者超定问题，我们可以通过正则方程求解，也就是矩阵求逆或者矩阵求伪逆。如果是欠定问题，则可以把正则化应用于回归分析中，主要涉及L1(Lasso Regression),L2(Ridge Regression),L1+L2(Elastic Net)正则化。

欠定问题在当前的一个核心，最典型的就是深度学习，一般模型的参数数目远大于数据的数目，因此需要一些正则化方法。比如L1(Lasso Regression),L2(Ridge Regression),L1+L2(Elastic Net)正则化，Early Stopping，因此NN不存在过拟合的原因是因为存在正则化。

多元回归分析

求解线性方程组，如果问题是为满秩，或者超定问题，可以通过矩阵求逆来解决。当然如果矩阵求逆太费时间，则可以通过共轭梯度法来求解。这一节不分析欠定问题。

多元回归方程

$$\hat{y} = \lambda_0 + \lambda_1 x_1 + \lambda_2 x_2 + \dots + \lambda_m x_m$$

其中 λ_j 是 x_j 的偏回归系数， \hat{y} 是样本的估计值。

根据最小值原理，可以求得偏回归系数。

$$L(\lambda) = \sum_{i=0}^{n-1} [y_i - (\lambda_0 + \lambda_1 x_{1i} + \lambda_2 x_{2i} + \dots + \lambda_m x_{mi})]^2$$

变量数为m,样本数为n. 最小二乘法的目的就是找到一组偏回归系数使得损失函数L极小，极端情况就是L=0，则所有样本估计值就是样本的观测值。这在满秩的情况下存在。一般只是求得L的极小值点。对损失函数求偏导，可以得到：

正则方程

实际问题可以转化成矩阵分析

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} & 1 \\ x_{21} & x_{22} & \cdots & x_{2m} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} & 1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} y_{1o} \\ y_{2o} \\ \vdots \\ y_{no} \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \\ \epsilon_0 \end{bmatrix}$$

即：

$$X * \lambda = Y + \epsilon$$

寻找一组 λ 使得 $L(\lambda) = (X * \lambda - Y)^T (X * \lambda - Y)$ 极小

利用 $L(\lambda)$ 对向量 λ 求导可以得到¹：

$$\frac{\partial L(\lambda)}{\partial \lambda} = 2X^T(X\lambda - Y) = 0$$

我们也可以得到：

$$\frac{\partial^2 L(\lambda)}{\partial \lambda^2} = 2X^T X$$

求得： $\lambda = (X^T X)^{-1} X^T Y$

$(X^T X)^{-1} X^T$ 也称为X的伪逆。

共轭梯度法

当矩阵X非常大的时候，时间复杂度是 $O(n^3)$ 的矩阵求逆方法太费时间，一般采用共轭梯度法来求解。这在矩阵计算那一章会讨论。

多项式拟合

函数的多项式表示方式：

$$y = \sum_{k=0}^m \lambda_k x^k$$

对于n组数据 (x_i, y_i) 若我们想用多项式去拟合这组数据，就是寻找使下列函数值极小的一组系数 λ

$$L(\lambda) = \frac{1}{2m} \sum_{i=0}^{n-1} (y_i - \sum_{k=0}^m \lambda_k x_i^k)^2$$

令： $\lambda = [\lambda_0, \lambda_1, \dots, \lambda_m]^T$

$$\text{s.t. } X_i = [x_i^0, x_i^1, \dots, x_i^m]$$

$$Y = [Y_0, Y_1, \dots, Y_{n-1}]^T$$

则上面损失函数可以表示为：

$$L(\lambda) = (X * \lambda - Y)^T (X * \lambda - Y)$$

对其求一阶导，结果为0：

$$\frac{\partial L(\lambda)}{\partial \lambda} = 0$$

最终方程可以化简成如下形式：

$$\begin{bmatrix} \sum_{j=0}^{n-1} x_j^0 & \sum_{j=0}^{n-1} x_j^1 & \cdots & \sum_{j=0}^{n-1} x_j^m \\ \sum_{j=0}^{n-1} x_j^1 & \sum_{j=0}^{n-1} x_j^2 & \cdots & \sum_{j=0}^{n-1} x_j^{m+1} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{j=0}^{n-1} x_j^m & \sum_{j=0}^{n-1} x_j^{m+1} & \cdots & \sum_{j=0}^{n-1} x_j^{2m} \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_m \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{n-1} y_j * x_j^0 \\ \sum_{j=0}^{n-1} y_j * x_j^1 \\ \vdots \\ \sum_{j=0}^{n-1} y_j * x_j^m \end{bmatrix}$$

即 $X * \lambda = Y$ 得到 $\lambda = X^{-1}Y$

若在方程上加上一个正则项，

$$L(\lambda) = \frac{1}{2m} \sum_{i=0}^{n-1} (y_i - \sum_{k=0}^m \lambda_k x_i^k)^2 + \sum_{k=0}^m \lambda_k^2$$

得到如下的矩阵：

$$\begin{bmatrix} \sum_{j=0}^{n-1} x_j^0 - 2n & \sum_{j=0}^{n-1} x_j^1 & \cdots & \sum_{j=0}^{n-1} x_j^m \\ \sum_{j=0}^{n-1} x_j^1 & \sum_{j=0}^{n-1} x_j^2 - 2n & \cdots & \sum_{j=0}^{n-1} x_j^{m+1} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{j=0}^{n-1} x_j^m & \sum_{j=0}^{n-1} x_j^{m+1} & \cdots & \sum_{j=0}^{n-1} x_j^{2m} - 2n \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_m \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{n-1} y_j * x_j^0 \\ \sum_{j=0}^{n-1} y_j * x_j^1 \\ \vdots \\ \sum_{j=0}^{n-1} y_j * x_j^m \end{bmatrix}$$

类似于Ridge回归：

但是对于加噪声参数的多项式数据，发现不加正则项能给出正确的结果，加正则项反倒不能，因为这是满秩问题，没必要加正则项加正则项的目的是为了解的稳定性而不是解的精确性，对于欠定问题，才可以加正则项来对解进行约束。

迭代法求解

求解上面的方程也可以使用迭代法求解，实际上就是最优化方法。

1. 矩阵求导术(上), <https://zhuanlan.zhihu.com/p/24709748> ↵

正则化回归

在线性回归分析中，如果是欠定问题，则可以把正则化应用于回归分析中，主要涉及L1(Lasso Regression),L2(Ridge Regression),L1+L2(Elastic Net)正则化。此外，Early Stopping也是一种正则化方法。

欠定问题在当前的一个核心，最典型的就是深度学习，一般模型的参数数目远大于数据的数目，因此需要一些正则化方法。比如L1(Lasso Regression),L2(Ridge Regression),L1+L2(Elastic Net)正则化，Early Stopping。

线性回归问题

对于线性系统，也就是预测函数：

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^p x_j \beta_j$$

然后通过最小二乘求解符合数据的模型。通过加正则项来约束模型的复杂度或者参数范围。对于非线性模型，只是把模型 $f(\mathbf{x})$ 换成你想要的非线性函数即可，比如N元M次多项式，选最简单的就是N元二次方程，这是最简单的非线性模型。

非线性回归问题

光学建模

物理学中有一些很有意思的非线性回归模型，比如光的衍射成像，光通过特定结构的衍射，反射成像，求解这些模型，第一步是建model，也就是通过求解Maxwell方程，得到非线性方程组，第二步就是通过实验数据来求解模型中的参数，这些参数有两类，一类是几何参数，也就是涉及到要处理问题的几何结构；第二类是物理参数，主要是描述系统材料属性的参数，比如折射率，反射率，光的波长等。

神经网络

一个典型的非线性回归例子就是深度学习，神经网络就是一个高度非线性的函数，非线性是由激活函数决定的，也就是激活函数是非线性函数。通过这个非线性函数，可以把输入的数据映射到输出的数据空间；高度的非线性是神经网络具有强大的非线性拟合能力的根本。神经网络的表达能力由下面的定理给出。

万能近似定理

万能近似定理(universal approximation theorem)(Hornik et al., 1989;Cybenko, 1989)表明,一个前馈神经网络如果具有线性输出层和至少一层具有任何一种“挤压”性质的激活函数(例如logistic sigmoid激活函数)的隐藏层,只要给予网络足够数量的隐藏单元,它可以以任意的精度来近似任何从一个有限维空间到另一个有限维空间的Borel 可测函数。

线性系统的Regularized Regression

这里讨论的还是线性系统，也就是预测函数：

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^p x_j \beta_j$$

然后通过最小二乘求解数据的模型。通过加正则项来约束模型的复杂度或者参数范围。对于非线性模型，只是把模型 $f(\mathbf{x})$ 换成你想要的非线性函数即可，比如N元M次多项式，选最简单的就是N元二次方程，这是最简单的非线性模型。

物理学中有一些很有意思的非线性回归模型，比如光的衍射成像，光通过特定结构的衍射，反射成像，求解这些模型，第一步是建model，也就是通过求解Maxwell方程，得到非线性方程组，第二步就是通过实验数据来求解模型中的参数，这些参数有两类，一类是几何参数，也就是设计到要处理问题的几何结构；第二类是物理参数，主要是描述系统材料属性的参数，比如折射率，反射率，光的波长等。

一个典型的非线性回归例子就是深度学习，神经网络就是一个高度非线性的函数，非线性是由激活函数决定的，也就是激活函数是非线性函数。通过这个非线性函数，可以把输入的数据映射到输出的数据空间；高度的非线性是神经网络具有强大的非线性拟合能力的根本。神经网络的表达能力由下面的定理给出。

万能近似定理

万能近似定理(universal approximation theorem)(Hornik et al., 1989;Cybenko, 1989)表明，一个前馈神经网络如果具有线性输出层和至少一层具有任何一种“挤压”性质的激活函数(例如logistic sigmoid激活函数)的隐藏层，只要给予网络足够数量的隐藏单元，它可以以任意的精度来近似任何从一个有限维空间到另一个有限维空间的Borel 可测函数。

正则化的目的是减少test误差而不是training误差，这是要明白的。

Ridge回归，Shrink与SVD

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij} \beta_j)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

等价于：

$$\begin{aligned} \hat{\beta}^{ridge} &= \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij} \beta_j)^2 \\ \text{s.t. } & \sum_{j=1}^p \beta_j^2 \leq t. \end{aligned}$$

上面的问题也等价于假定， y_i, β_i 服从如下分布：

$$y_i \in N(\beta_0 + \mathbf{x}_i^T \beta, \sigma^2)$$

$$\beta_i \in N(0, \tau^2)$$

其中: $\lambda = \sigma^2 / \tau^2$

因此RRS(Root sum square)可以写成如下形式:

$$RRS(\lambda) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\beta^T\beta$$

通过对 β 求导并令其为0, 可以得到:

$$\hat{\beta}^{ridge} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{y}$$

为了建立起L2与SVD之间的联系, 我们对 \mathbf{X} 进行SVD分解:

$$\mathbf{X} = \mathbf{UDV}^T$$

我们重写没有正则化的最小二乘拟合:

$$\mathbf{X}\beta^{ls} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{UU}^T\mathbf{y}$$

对于L2正则化的最小二乘法:

$$\mathbf{X}\beta^{ls} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{UD}(\mathbf{D}^2 + \lambda I)^{-1}\mathbf{DU}^T\mathbf{y}$$

$$= \sum_{j=1}^p \mathbf{u}_j \frac{d_j^2}{d_j^2 + \lambda} \mathbf{u}_j^T \mathbf{y}$$

从上面的分式 $\frac{d_j^2}{d_j^2 + \lambda}$ 可以知道, 对于 $d_j \ll \lambda$, 则相应的方向会收缩到0。可以认为L2就是对数据进行了

SVD分解后, 只保留了 $d_j > \lambda$ 的分量。

Lasso(L1)回归

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

等价于:

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2$$

$$\text{subject to : } \sum_{j=1}^p |\beta_j| \leq t.$$

L1更容易产生稀疏性, 因为椭圆与菱形更易在菱形顶点相交, 而与圆形不容易在坐标轴上相切。



L1稀疏性的数学解析

一个数学上更严格 的解析。。

上面的问题也等价于假定, y_i, β_i 服从如下分布:

$$y_i \in N(\beta_0 + x_i^T \beta, \sigma^2)$$

$$\beta_i \in (1/2\tau) \exp(-|\beta|/\tau)$$

其中: $\tau = 1/\lambda$

因此 β_i 更容易分布在0的附件, 也就是能级排斥(量子混沌里面的概念, L1是可积系统, L2是GOE)

最小二乘一般通过Cholesky分解($p^3 + Np^2/2$)或者QR分解(NP^2)来实现, 前者一般更快, 但是没有后者稳定。

聚类

聚类的方法有

1. K-means
2. 核K均值
3. 谱聚类
4. K中值

无监督学习中的EM算法是否属于聚类算法？

K均值

K-means是一种无监督的学习方法，通过定义距离与分类的数目K，样本归属于哪个类，取决于该样本距离K个类中心的距离，选择距离最小的类作为归属。随机选择K个起始点，通过迭代，最终收敛。其迭代过程就是求如下函数极小的过程。

$$E = \sum_{i=1}^K \sum_{x \in C_i} \|x - u_i\|^2$$

$$u_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

问题

一个主要的问题就是怎么选取类的数目K。

改进方案

k-means对初始值的设置很敏感，所以有了k-means++、intelligent k-means、genetic k-means。

k-means对噪声和离群值非常敏感，所以有了k-medoids和k-medians。

k-means只用于numerical类型数据，不适用于categorical类型数据，所以k-modes。

k-means不能解决非凸(non-convex)数据，所以有了kernel k-means

K-Means算法流程

输入：样本集D=(x_1, x_2, \dots, x_m)，聚类数目k。

从D中随机选取k个样本点作为中心，(u_1, u_2, \dots, u_k)

repeat:

 令 $C_i = 0$, ($1 \leq i \leq k$)

 for j=1,2,...,m,do

 样本点 x_j 到与各均值 u_i 的距离

$$d_{ij} = \|x_j - u_i\|_2$$

 根据距离确定样本点j所属于的类: $\lambda_j = \arg \min d_{ji}$

 将 x_j 划入所属的类 $C_{\lambda_j} = C_{\lambda_j} + u_i$

 end for

 for i=1,2,...,k,do

$$\text{计算 } u'_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

 if $u'_i \neq u_i$

```
     $\mathbf{u}_i = \mathbf{u}'_i$ 
else:
    not change
end for
until所有的均值向量都没有改变。
输出族划分C=( $C_1, C_2, \dots, C_k$ )
```

高斯混合模型

EM算法的引入

概率模型有时既含有可观测变量，又含有隐变量(潜在变量Latent variable)。如果概率模型的变量都是可观测变量，那么给定数据，就可以直接按极大似然估计方法，或者贝叶斯估计方法估计模型参数。

但是当模型含有隐变量时，就不能简单地使用这些估计方法，EM算法就是含有隐变量的概率模型参数的极大似然估计方法，或极大后验概率估计方法。

三硬币模型

为了直观的理解含有隐变量的模型，我们讨论抛硬币的例子。

三枚硬币A, B, C正面出现的正面的概率是 π, p, q 。进行如下实验：先掷硬币A，正面选硬币B，反面选硬币C；然后掷选出的硬币，掷硬币的结果出现正面记作1，反面记作0。独立地重复n次实验，得到一个观察序列如下：

1,1,0,1,0,0,1,0,1,1

三硬币模型可以写作：

$$\begin{aligned} P(y|\theta) &= \sum_z P(y, z|\theta) = \sum_z P(z|\theta)P(y|z, \theta) \\ &= \pi p^y (1-p)^{1-y} + (1-\pi)q^y (1-q)^{1-y} \end{aligned}$$

$\Theta = (\pi, p, q)$ 是模型参数，随机变量y的数据可以观察，就是B,C抛出的是正面还是反面；随机变量z的数据不可以观察，就是B,C哪枚硬币。我们需要求得就是模型的参数 $\Theta = (\pi, p, q)$ 。

最大似然函数

最大似然就是寻找一组模型参数，使得在该参数下，观测序列出现的概率极大。因此我们可以定义我们的目标函数就是在该模型参数之下，每个样本出现的概率的乘积。我们要最大化的就是这些概率的乘积。

将可观察数据表示为 $Y = (Y_1, Y_2, \dots, Y_n)^T$ ，不可观测数据表示为

$$Z = (Z_1, Z_2, \dots, Z_n)^T.$$

以最简单的二分类模型为例，则观察数据的似然函数是：

$$\begin{aligned} P(Y|\theta) &= \sum_Z P(Z|\theta)P(Y|Z, \theta) \\ &= \prod_{j=1}^n [\pi p^y (1-p)^{1-y} + (1-\pi)q^y (1-q)^{1-y}] \end{aligned}$$

考虑求模型参数 $\Theta = (\pi, p, q)$ 的极大似然估计，即：

$$\hat{\theta} = \arg \max_{\theta} \log(P(Y|\theta))$$

只能通过迭代来求解模型参数。EM算法就是求解该问题的一种方法。其实上面的处理的方法就是Logistic回归处理二分类的方法。

EM算法

E步：计算模型参数 $\pi^{(i)}, p^{(i)}, q^{(i)}$ 下观测数据 y_j 来自掷硬币B的概率是(无论是正面还是反面, 都可以用下面公式进行计算):

$$u^{(i+1)} = \frac{\pi^{(i)}(p^{(i)})^{y_j}(1-p^{(i)})^{1-y_j}}{\pi^{(i)}(p^{(i)})^{y_j}(1-p^{(i)})^{1-y_j} + (1-\pi^{(i)})(q^{(i)})^{y_j}(1-q^{(i)})^{1-y_j}}$$

M步：计算模型参数的新估计值：

之所以抛硬币B是因为抛硬币A出现了正面, 因此 π (抛A出现正面的几率如下计算)

$$\pi^{(i+1)} = \frac{1}{n} \sum_{j=1}^n u^{(i+1)}$$

在抛硬币B的前提下, 根据抛B得到正面的频率得到P(B正面出现的概率)

$$p^{(i+1)} = \frac{\sum_{j=1}^n u^{(i+1)} y_j}{\sum_{j=1}^n u^{(i+1)}}$$

在抛硬币C的前提下(1-来自抛B的概率), 根据抛C得到正面的频率得到P(C正面出现的概率)

$$q^{(i+1)} = \frac{\sum_{j=1}^n (1-u^{(i+1)}) y_j}{\sum_{j=1}^n (1-u^{(i+1)})}$$

EM算法流程

输入：观察变量数据Y, 隐变量数据Z, 联合分布 $P(Y, Z|\theta)$, 条件分布 $P(Z|Y, \theta)$;

输出：模型参数 θ .

(1)选择参数的初始值 $\theta^{(0)}$, 开始迭代；

(2)E步：记 $\theta^{(i)}$ 为第*i*次迭代参数 θ 的估计值, 在第*i*+1次迭代的E步, 计算 $\log P(Y, Z|\theta)$ 在 $Y, \theta^{(i)}$ 之下的对数期望：

$$Q(\theta, \theta^{(i)}) = E_z[\log P(Y, Z|\theta)|Y, \theta^{(i)}] = \sum_z P(Z|Y, \theta^{(i)}) \log P(Y, Z|\theta)$$

这里, $P(Y, Z|\theta^{(i)})$ 是在给定观测数据Y和当前的参数估计 $\theta^{(i)}$ 下隐变量数据Z的条件概率分布：

(3)M步(求导令其等于0得极大)：求使 $Q(\theta, \theta^{(i)})$ 极大的 θ , 确定第*i*+1次迭代的参数的估计值 $\theta^{(i+1)}$:

$$\theta^{(i+1)} = \arg \max_{\theta} Q(\theta, \theta^{(i)})$$

(4)重复第(2)(3)步, 直到收敛。

简言之：

E步：就是求在观测数据Y以及当前估计参数 $\theta^{(i)}$ 下的 $P(Y, Z|\theta)$ 的对数期望值 $Q(\theta, \theta^{(i)})$ 。

M步：就是求使得 $Q(\theta, \theta^{(i)})$ 极大的 θ ，再将求得的 θ 作为下一步 $\theta^{(i+1)}$ 。

Q函数

完全数据的对数似然函数 $\log P(Y, Z|\theta)$ 关于给定观测数据Y和当前参数 θ 下对未观察数据Z的条件概率分布 $\log P(Z|Y, \theta^{(i)})$ 的期望成为Q函数，即：

$$Q(\theta, \theta^{(i)}) = E_z[\log P(Y, Z|\theta)|Y, \theta^{(i)}]$$

高斯混合模型

高斯混合模型是指的具有如下形式的概率分布模型：

$$P(y|\theta) = \sum_{k=1}^K \alpha_k \phi(y|\theta_k)$$

其中 α_k 是系数 $\alpha_k \geq 0$, $\sum_{k=1}^K \alpha_k = 1$; $\phi(y|\theta_k)$ 是高斯分布函数, $\theta_k = (u_k, \sigma_k^2)$,

$$P(y|\theta) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{(y-u_k)^2}{2\sigma_k^2}\right)$$

称为第k个分模型。

高斯混合模型参数估计得EM算法

可以用EM算法来求解GMM的参数 α_k, u_k, σ_k 。

$\gamma_{jk} = 1$ 表示第j个观测来自第k个分模型, $\gamma_{jk} = 0$ 表示第j个观测不是来自第k个分模型。

有了观察数据 y_j 以及未观测数据 γ_{jk} ,

$\gamma_{jk} = 1$, 如果第j个观测来自第k个分模型。

$\gamma_{jk} = 0$, 其它情况。

则完全数据是：

$$(y_j, \gamma_{j1}, \gamma_{j2}, \dots, \gamma_{jK}), j = 1, 2, \dots, K$$

似然函数为：

$$\begin{aligned} P(y, \gamma|\theta) &= \prod_{j=1}^N P(y_j, \gamma_{j1}, \gamma_{j2}, \dots, \gamma_{jK}|\theta) \\ &= \prod_{k=1}^K \prod_{j=1}^N [\alpha_k \phi(y_j|\theta_k)]^{\gamma_{jk}} \\ &= \prod_{k=1}^K \alpha_k^{\gamma_{jk}} \prod_{j=1}^N [\phi(y_j|\theta_k)]^{\gamma_{jk}} \end{aligned}$$

以上对任何分布都实用，接下来具体到GMM：

$$\begin{matrix} n & (y - u) \\ & 228 \end{matrix} \quad \gamma$$

$$= \prod_{k=1}^K \alpha_k^{n_k} \prod_{j=1}^N \left[\frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{(y - u_k)^2}{2\sigma_k^2}\right) \right]^{\gamma_{jk}}$$

式中 $\sum_{j=1}^N \gamma_{jk} = n_k$, $\sum_{k=1}^K n_k = N$ 。

转化为完全数据的对数似然函数。

总结

隐变量模型

对于隐变量模型，我们观测的现象可能由不同的因素导致，但是每次的结果只是由一个因素导致。每个因素起作用的概率不同，所有因素起作用的概率之和就是1。

在这里，我们主要讨论的就是概率模型。我们需要求得是两组参数，一组参数是 $\theta_1, \theta_2, \dots, \theta_K$ ，每一个 θ_k 决定一个归一化的概率分布，比如高斯分布， K 表示有 K 个不同的高斯分布，另外一组参数就是每个分布的权重 $\alpha_1, \alpha_2, \dots, \alpha_K$ 。 K 还有另外一层含义，就是系统隐变量个数。因此，总的概率分布就是：

$$P(y|\Theta) = \sum_{k=1}^K \alpha_k \phi(y|\theta_k)$$

满足 $\sum_{k=1}^K \alpha_k = 1$ 。

$\phi(y|\theta_k)$ 是归一化的概率分布。比如高斯均值，方差(多维是协方差矩阵)不同的分布。或者是任意分布，比如在人脸识别中应用比较多的t-分布。

y 是我们观测到的数据，比如硬币的正反面，或者一幅图像(一个矩阵)。

EM算法

极大化对数似然函数。

完全数据的似然函数是：

$$\begin{aligned} P(y, \gamma|\theta) &= \prod_{j=1}^N P(y_j, \gamma_{j1}, \gamma_{j2}, \dots, \gamma_{jK} | \theta) \\ &= \prod_{k=1}^K \prod_{j=1}^N [\alpha_k \phi(y_j | \theta_k)]^{\gamma_{jk}} \\ &= \prod_{k=1}^K \alpha_k^{n_k} \prod_{j=1}^N [\phi(y_j | \theta_k)]^{\gamma_{jk}} \end{aligned}$$

式中 $\sum_{j=1}^N \gamma_{jk} = n_k$, $\sum_{k=1}^K n_k = N$ 。

完全数据的对数似然函数是：

$$\log P(y, \gamma|\theta) = \sum_{k=1}^K [n_k \log \alpha_k + \sum_{j=1}^N \gamma_{jk} \log [\phi(y_j | \theta_k)]]$$

极大似然与最大后验概率估计

最大似然估计是求参数 θ , 使似然函数 $P(y|\theta)$ 最大。最大后验概率估计则是求 θ 使得 $P(y|\theta)P(\theta)$ 最大。求得的 θ 不单单让似然函数最大, 要 θ 出现的概率也很大。其实对数化之后, 就是加上一个有关模型 θ 的正则项。也就是结构风险最小化就等价于最大后验概率估计。

因此, 加正则约束的优化问题, 实际上是最大化后验概率估计, 而不是最大化似然函数。比如加L2正则化, 就是认为, 参数先验的服从高斯分布。

MAP实际上在最大化

$$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)}$$

因为 $P(y)$ 是个固定值, 也就是 y 在实验中出现的次数, 比如硬币正面朝上的次数。所以可以去掉分母 $P(y)$ 。因此最大化后验概率估计就是最大化 $P(\theta|y)$ 。最大化 $P(\theta|y)$ 的意义很明确, y 出现了, 要求 θ 取什么值使得后验概率 $P(\theta|y)$ 最大。

那最大化 $P(\theta|y)$ 与最大化 $P(y|\theta)$ 的区别是什么?

MAP最大化 $P(\theta|y)$:也就是在已知的数据中, 求结构风险最小的模型参数。

MLP最大化 $P(y|\theta)$:也就是求使得数据出现概率最大的模型参数, 它相当于不对模型做惩罚, 结果可能是训练集上效果好, 测试集合上效果差。

降维

降维的方法有：

1. PCA
2. 核函数PCA
3. LDA
4. L1正则化
5. 局部保持映射
6. 拉普拉斯特特征映射

PCA

机器学习一般面对的是一个高维的问题，样本稀疏。容易面临严重的维数灾难。处理维数灾难的一个重要方法就是降维。确切的来说，有两种情形：

1. 基于距离的聚类问题。对于M个数据，定义距离后，M个数据张成一个MxM的距离矩阵，而且是对称矩阵。
2. 基于数据的低秩分解。对于M个数据，每个数据是N维，我们是对这个数据集张成的MxN维矩阵进行降维。

这就可以用到我们在矩阵计算中的一些方法，比如主成分分析(PCA)，实际上我们是通过SVD来实现PCA的，就是取最大的K个奇异值与奇异向量来实现矩阵的低秩分解。

下面我们主要考虑第一种情况，也就是处理聚类问题。

SVD

对于任何一个MxD的矩阵X可以做如下分解：

$$X = UDV^T$$

其中 $U^T U = I_D$, $V V^T = I_N$, D是MxD的对角矩阵。

$$X X^T = U D V^T V D^T U^T = U D D^T U^T$$

$$X^T X = V D^T U^T U D V^T = V D^T D V^T$$

是矩阵的本征值分解。因此可以通过上面两式来求解U, V。主要用到的是矩阵本征值，本征矢求解技巧。这都在矩阵计算(第五章)那一章讨论的内容。

对于我们的情况M=D, 因此

$$D^T = D$$

令 $B = X^T X$, 对B做特征值分解，得到

$$B = V \Lambda V^T,$$

假设Λ的矩阵元按如下方式排序：

$$\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_M$$

PCA

我们可以通过SVD来实现PCA，具体就是只取X的SVD中的前M个其一分量。

$$X = UDV^T \approx U_M D_M V_M^T$$

其中 U_M, D_M, V_M 对于矩阵X的前M个奇异值分量。

取前d个奇异向量。选择一个比例t, 一般取t = 95%, 使得。

$$\frac{\sum_{i=1}^d \lambda_i}{\sum_{i=1}^M \lambda_i} \geq t$$

PCA的好处

1. 通过降维, 使得样本的采样密度变大, 这是降维的主要目的¹
2. 当数据收噪声影响时, 小的特征值所对应的特征向量一般受噪声影响, 因此去掉这些量在一定程度上能起到去噪的作用。

PCA成立的条件

PCA依赖于以线性合并为基础的经验数据集(这里不讨论非线性PCA), PCA另外一个重要的假设是原始数据集是从高斯分布中抽取出来的。当这个假设不正确时, 就无法保证主要成分的有效性。这里我是有两个疑问的:

1. 高斯分布的矩阵元, 而产生的对称矩阵, 可以通过随机矩阵分析, 得到其展平的能谱时服从高斯正交分布, 进而简并的能级概率为0, 而保证最大的一些能级占的能量可以占到整个系统能量的很大比, 比如95%。但是非高斯分布就可以吗? 比如t-分布(在极端情况下就是高斯分布)
2. 一般推荐系统中的UI矩阵都是非负矩阵, 明显就不是服从高斯分布的, 为啥还可以用SVD? 能从正交变换来考虑吗?(解析就是, 只要高斯分布的均值就是数据的均值就可以了。但是购物的均值是有限的, 最大值确实无限的, 因此也不是完美的高斯分布。不是完美的高斯分布也没关系, 大致符合就可以了)。还是我们需要讨论服从泊松分布的随机矩阵的本征值问题?当然, 可以直接通过数值实验来验证。之所以要研究泊松分布, 使用时因为元素为0的概率最大。
3. 高斯分布是大自然中最常见的, 因为可以由中心极限定理得到高斯分布。

二项分布, 泊松分布于高斯分布

二项分布的期望是 np , 方差是 $np(1 - p)$ 。

当 $n \rightarrow \infty$ 时, p 很小, 二项分布就趋近于泊松分布, 也就是期望与方差一样都是 np ; p 不接近于0或者1时, 就趋近于均值为 np , 方差是 $np(1 - p)$ 的高斯分布。

因此, 即使是二项分布, 大量独立事件的总体效果就是高斯分布。

中心极限定理

中心极限定理: 设随机变量序列 X_i 相互独立, 具有相同的期望与方差, 也就是

$E(X_i) = u, D(X_I) = \sigma^2$, 令:

$$Y_n = X_1 + \dots + X_n,$$

$$Z_n = \frac{Y_n - \bar{Y}_n}{\sqrt{\text{Var}(Y_n)}},$$

则 $Z_n \rightarrow N(0, 1)$

因此独立同分布产生的数据的总的效果就是高斯分布。

1. 周志华《机器学习》

流形学习¹

流形学习是一类借鉴拓扑流形概念的降维方法，"流形"是在局部与欧氏空间同胚的空间。换言之，它在局部具有欧氏空间的性质，可以用欧式距离来计算距离。对于近邻，我们可以利用欧式距离来计算距离，而对于非近邻，我们可以利用测地线来计算距离，转化成计算两点之间最短路径，计算最短路径可以利用Dijkstra算法或者Floyd算法。

等度量映射

等度量映射(Isometric Mapping,简称IsoMap)的基本出发点是认为低维流形嵌入到高维空间后，直接在高维空间计算距离具有误导性，因为高维空间的直线距离在低维空间是不可达到。就如下图所示，低维空间的距离是"测地线geodesic"距离，是图中的红线是这两点的测地线，测地线是两点之间的本真距离。显然，直接在高维空间计算两点之间的距离是不恰当的。

我们必须通过流形在局部上与欧氏空间同胚这个性质，对每个点基于欧式距离找到近邻点，然后建立近邻连接图，再计算图中任意两点的距离，算法就是Dijkstra算法或者Floyd算法。

得到任意两点之间的距离之后，就可以利用MDS方法来获得样本点在低维空间的坐标。



图 10.7 低维嵌入流形上的测地线距离(红色)不能用高维空间的直线距离计算，但能用近邻距离来近似

局部线性嵌入(LLE)

与Isomap试图保持近邻样本之间的距离不同，局部线性嵌入(Locally Linear Embedding)试图保持领域内样本之间的线性关系。如下图所示，假定样本点 x_i 的坐标可以通过它的领域样本 x_j, x_k, x_l 的坐标通过线性组合的方式来重构出来。即：

$$\mathbf{x}_i = w_{ij}\mathbf{x}_j + w_{ik}\mathbf{x}_k + w_{il}\mathbf{x}_l$$

LLE希望上式的关系在低维空间得以保持。



图 10.9 高维空间中的样本重构关系在低维空间中得以保持

LLE为每个样本 \mathbf{x}_i 找到近邻下标集合 Q_i , 然后计算出基于 Q_i 中的样本点对 \mathbf{x}_i 进行线性重构的系数

\mathbf{w}_i :

$$\min_{\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_m} \sum_{i=1}^m \|\mathbf{x}_i - \sum_{j \in Q_i} w_{ij} \mathbf{x}_j\|_2^2$$

$$\text{s.t. } \sum_{j \in Q_i} w_{ij} = 1$$

其中 $\mathbf{x}_i, \mathbf{x}_j$ 均已知, 令 $C_{jk} = (\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_k)$ 。 w_{ij} 有闭式解:

$$w_{ij} = \frac{\sum_{k \in Q_i} C_{jk}^{-1}}{\sum_{l, s \in Q_i} C_{ls}^{-1}}$$

LLE在低维中保持 \mathbf{w}_i 不变, 于是 \mathbf{x}_i 对应的低维空间坐标 \mathbf{z}_i 可以通过下式求解:

$$\min_{\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_m} \sum_{i=1}^m \|\mathbf{z}_i - \sum_{j \in Q_i} w_{ij} \mathbf{z}_j\|_2^2$$

令 $\mathbf{Z} = (\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_m) \in R^{(d', m)}$, $(W)_{ij} = w_{ij}$ 。

$$\mathbf{M} = (\mathbf{I} - \mathbf{W})^T(\mathbf{I} - \mathbf{W}),$$

则关于 \mathbf{z}_i 的最优化问题可以转化为:

$$\min_{\mathbf{Z}} \text{tr}(\mathbf{Z} \mathbf{M} \mathbf{Z}^T),$$

$$\text{s.t. } \mathbf{Z} \mathbf{Z}^T = 1,$$

可以对上面你的 \mathbf{M} 进行特征值分解, \mathbf{M} 最小的 d' 个特征值对应的特征向量组成的矩阵就是 \mathbf{Z}^T

度量学习

决策树与集成学习

1. 决策树
 - i. ID3
 - ii. C4.5
 - iii. CART
 - iv. Decision Stump
2. 集成学习
 - i. Bagging
 - i. 随机森林
 - ii. Boost
 - i. Adaboost
 - ii. GBDT
 - iii. XGBoost
 - iv. LightGBM

Ensemble learning(团体学习)

利用多个学习方法来获得更好的预测能力，机器学习中的系综学习的样本是有限的，但是统计力学中的系综方法其样本是无限的。机器学习中的系综方法包括Bagging,boosting

Bagging算法(Bootstrap aggregating)

参考 是一种团体学习方法,可以看成是一种圆桌会议,或者投票选举的形式,其思想是“群众的眼光是雪亮的”,可以训练多个模型,之后将这些模型进行加权组合,一般这类方法的效果,都会好于单个模型的效果。在实践中,在特征一定的情况下,大家总是使用Bagging的思想去提升效果。算法步骤 给定一个大小为n的训练集D, Bagging算法从D中均匀、有放回地(即使用自助抽样法)选出m个大小为 n' 的子集 D_i ,作为新的训练集。在这m个训练集上使用分类、回归等算法,则可得到m个模型,同一个训练集中的成员可以有重重复,再通过取平均值、取多数票等方法,即可得到Bagging的结果。

Boosting

在Bagging方法中,我们假设每个训练样本的权重都是一致的;而Boosting算法则更加关注错分的样本,越是容易错分的样本,约要花更多精力去关注。对应到数据中,就是该数据对模型的权重越大,后续的模型就越要拼命将这些经常分错的样本分正确。最后训练出来的模型也有不同权重,所以boosting更像是会整,级别高,权威的医师的话语权就重些。训练:先初始化每个训练样本的权

重相等为 $1/d$, d 为样本数量;之后每次使用一部分训练样本去训练弱分类器,且只保留错误率小于0.5的弱分类器,对于分对的训练样本,将其权重调整为 $\text{error}(M_i)/(1-\text{error}(M_i))$,其中 $\text{error}(M_i)$ 为第*i*个弱分类器的错误率(降低正确分类的样本的权重,相当于增加分错样本的权重);

测试:每个弱分类器均给出自己的预测结果,且弱分类器的权重为 $\log(1-\text{error}(M_i))/\text{error}(M_i)$ 权重最高的类别,即为最终预测结果。

在adaboost中,弱分类器的个数的设计可以有多种方式,例如最简单的就是使用一维特征的树作为弱分类器。

adaboost在一定弱分类器数量控制下,速度较快,且效果还不错。

我们在实际应用中使用adaboost对输入关键词和推荐候选关键词进行相关性判断。随着新的模型方法的出现,adaboost效果已经稍显逊色,我们在同一数据集下,实验了GBDT和adaboost,在保证召回基本不变的情况下,简单调参后的Random Forest准确率居然比adaboost高5个点以上,效果令人吃惊。。。

Bagging和Boosting都可以视为比较传统的集成学习思路。现在常用的Random Forest, GBDT, GBRank其实都是更加精细化,效果更好的方法。后续会有更加详细的内容专门介绍。

决策树与集成学习

决策树是一种基本的回归与分类的方法。决策树由节点与边构成，节点分类内部节点与叶子节点；内部节点表示属性或者特征，叶子节点表示一个类。

决策树学习

训练集 $D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N))$ 其中 $x_i = (x_i^{(1)}, x_i^{(2)} \dots x_i^{(n)})$ 是 n 维变量， n 表示特征的数目， $y_i \in (1, 2 \dots K)$ 类标签。训练的本质是基于一定标准得到一组分类规则。我们需要得到一组与训练数据矛盾较小并且泛化能力也好的分类规则，其中泛化能力说的是在测试集上其错误率也小，错误率有不同的定义方式，可以是均方误差，也可以是似然函数，一般回归问题选择均方误差，分类问题选择似然函数，想想这是为什么？。

分类规则就是特征选择的过程，特征选择是基于一些可以量化的函数，一般基于信息熵增益最大或者信息熵增益率最大或者 Gini 系数下降最大的规则。因此，接下来我们要定义如下概念。

1. 信息熵
2. 条件熵
3. 信息增益
4. 信息增益率
5. Gini 系数

1. 信息熵

熵的概念来自于统计物理，描述微观系统的混乱程度，由 Rudolf Clausius 提出，

$$S = k_B \ln \Omega = -k_B \sum_i p_i \ln p_i$$

其中 k_B 是 Boltzmann constant， Ω 表示系统的可能状态数。

它表示的就是 $\ln(p_i)$ 的期望值。香农把这个概念引入信息学，定义了信息熵。

考虑一个只能取有限 n 个值的系统，值对应的就是状态，比如投硬币，只能取两个值，就是说只能有 2 个状态，因此 $n=2$ ，在统计物理中，状态对应的是系统的能级，也就是一个能级对应一个状态，系统处于不同能级的概率就是 p_i

$$P(X = x_i) = p_i, i = 1, 2..n$$

熵的定义如下： $H(x) = -\sum_{i=1}^n p_i \ln p_i$

若系统是确定的，则有一个 $p_i = 1$ ，其他的 $p_i = 0$ 等于 0，因此代入熵的公式可知，熵为 0。可以证明，对于 n 个状态的系统，系统的熵满足如下不等式：

$$0 \leq H(x) = -\sum_{i=1}^n p_i \ln p_i \leq \ln n.$$

2. 条件熵

联合概率说的是两个及两个以上系统随机变量共同发生的几率问题, 条件熵 $H(Y|X)$ 说的是随机变量X给定的情况下, 随机变量Y的条件熵。

计算如下:

$$H(Y|X) = \sum_{i=1}^{n=p} p_i H(Y|X=x_i)$$

这里, $p_i = P(X=x_i), i = 1, 2..n.$

$H(Y|X=x_i)$ 计算与上面的信息熵一样, 只是对于 $H(Y|X=x_i)$, 我们只计算 $X=x_i$ 的那些样本的熵, 因为我们会对所有X取不同值得样本进行求和, 因此会遍历整个样本。

3. 信息增益

信息增益是基于以上两个概念的, 是针对于特征而言的, 特征A对训练数据集D的信息增益 $g(D,A)$ 定义成经验熵 $H(D)$ 与特征A给定的条件下, D的经验条件熵 $H(D|A)$ 之差。即:

$$g(D, A) = H(D) - H(D|A).$$

如果我们知道数据集D的信息熵计算, 以及条件熵的计算, 则信息增益的计算不难。

4. 信息增益率

特征A对训数据集D的信息增益率 $g_R(D, A)$ 定义成信息增益 $g(D,A)$ 与特征A的经验熵 $H(A)$ 之比:

$$g_R(D, A) = \frac{g(D, A)}{H(A)}.$$

其中 $H(A) = \sum_{i=1}^{K=p} p_i \ln p_i$, K是特征A能取不同值得数目, p_i 是取不同值得概率。

他的计算基于信息增益与信息熵。

5. Gini系数

基尼系数定义: 对于分类问题, 假设有K个类, 样本点处于第k个类的概率是 p_k , 则概率分布的基尼系数定义成:

$$Gini(p) = \sum_{i=1}^{n=p} p_i(1-p_i) = 1 - \sum_{i=1}^{K=p} p_i^2$$

基尼系数描述的是不同类之间的不一致程度, 一个类与其它类的不一致概率和, 也就是不一致程度程度, 说的是从样本集合D中随便挑出两个样本, 他们属于不同类的概率。由下面函数刻画

$p_i(1-p_i)$. 对于分类问题, 我们希望最终的叶子节点里面, 所有的样本点尽可能使同一个类里面的, 也就是基尼系统尽可能趋近于0.

假设样本D中有K个类, C_k 是第k类的集合, $|C_k|$ 是集合 C_k 的大小, 因此样本集合的基尼系数定义成:

$$Gini(p) = 1 - \sum_{i=1}^{K=p} \left(\frac{|C_k|}{|D|}\right)^2$$

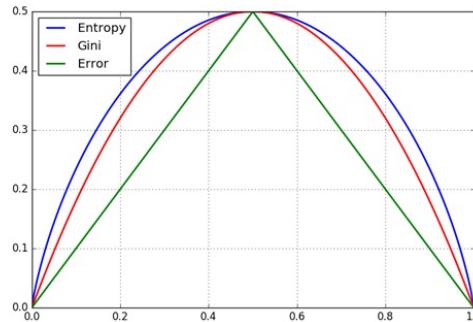
基于不同的指标, 比如信息增益, 信息增益率, 基尼系数, 我们会得到不同的决策树生成算法, 依次是ID3, C4.5, CART。

关于Gini系数的讨论(一家之言)

□ 考察Gini系数的图像、熵、分类误差率三者之间的关系

■ 将 $f(x) = -\ln x$ 在 $x=1$ 处一阶展开，忽略高阶无穷小，得到 $f(x) \approx 1-x$

$$\begin{aligned} H(X) &= -\sum_{k=1}^K p_k \ln p_k \\ &\approx \sum_{k=1}^K p_k (1-p_k) \end{aligned}$$



信息熵与基尼系数的函数大致一致，实际上在iris数据集中，生成决策树时，两者没有差别。

ID3算法

ID3算法基于信息增益最大，代码实现如下，它的缺点在于，它倾向于选择取值很多的特征，因为当特征能取很多值得时候，此时系统的不确定度降低，极端情况是，特征能取N个不同的值，N个训练集的数量，此时特征A条件熵为0。为了避免这种情况，而引入了C4.5算法。

假设特征A是信息增益最大的特征，当特征A取离散值时，假设特征A可以取K个不同值，我们选择A作为特征之后会生出K个节点，对于每个节点，我们要重复上面的步骤，继续寻找信息增益最大的特征，只是，这时候特征数目减小了1，减小的这个特征就是A特征本身。当一个节点，它的熵小于一定阈值的时候，我们就不再分割这个节点而是把他当成一个叶子节点。我们可以使用递归来实现ID3算法。

对于特征值取连续的情况，我们通过计算特征值大于阈值与小于阈值的两部分熵之和来计算熵。算法实现的时候要注意的一点就是，计算熵时，不能让概率等于0，否则会出错。

```
import scipy
import numpy as np
from sklearn import tree
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

class DecisionTree(object):
    def __init__(self):
        self.training_steps = 50
```

```

def Entropy(self, Prob):
    entropy = 0
    for n in range(Prob.shape[0]):
        entropy -= Prob[n] * (np.log2(Prob[n])) ##Entropy
        #entropy += Prob[n] * (1 - Prob[n]) ##Gini
    return entropy

def distribution(self, feature_data, threshold_n):
    entropy = 0
    prob = 0.001*np.ones((3, 1))
    for n in range(0, threshold_n):
        for index in range(3):
            if np.abs(feature_data[n, 1] - index) < 0.01:
                prob[index] += 1
    prob = prob/threshold_n
    p_1 = (1.0*threshold_n)/feature_data.shape[0]
    H_1 = self.Entropy(prob)
    entropy += p_1*H_1

    prob_2 = 0.001*np.ones((3, 1))
    for n in range(threshold_n, feature_data.shape[0]):
        for index in range(3):
            if round(feature_data[n, 1]) == index:
                prob_2[index] += 1
    prob_2 = prob_2/(feature_data.shape[0] - threshold_n)
    H_2 = self.Entropy(prob_2)
    entropy += (1 - p_1) * H_2
    return entropy

def best_feature_threshold(self, data):
    best_threshold = 0
    best_feature = 0
    best_entropy = 100000
    feature_num = data.shape[1] - 1
    for feature_index in range(feature_num):
        a_arg = np.argsort(iris_data[:, feature_index])
        sorted_data = iris_data[a_arg]
        for n in range(1, sorted_data.shape[0]):
            threshold = (sorted_data[n-1, feature_index] + sorted_data[n, feature_index])/2
            feature_result_data = np.zeros((sorted_data.shape[0], 2))
            feature_result_data[:, 0] = sorted_data[:, feature_index]
            feature_result_data[:, -1] = sorted_data[:, -1]
            condition_entropy = self.distribution(feature_result_data, n)
            if condition_entropy < best_entropy:
                best_threshold = threshold
                best_entropy = condition_entropy
                best_feature = feature_index
    print best_entropy, best_feature, best_threshold
    return best_entropy, best_feature, best_threshold

if __name__ == '__main__':
    DT = DecisionTree()
    iris = load_iris()

```

```

clf = tree.DecisionTreeClassifier()
clf = clf.fit(iris.data, iris.target)
data = iris.data
target = iris.target
iris_data = np.zeros((data.shape[0], data.shape[1] + 1))
iris_data[:, 0:data.shape[1]] = data
iris_data[:, -1] = target
best_entropy, best_feature, best_threshold = DT.best_feature_threshold(iris_data)

```



例子

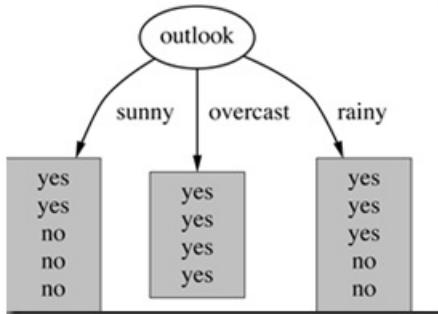
表1 天气预报数据集例子

Outlook	Temperature	Humidity	Windy	Play?
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rain	mild	high	false	yes
rain	cool	normal	false	yes
rain	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rain	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rain	mild	high	true	no

表中一共包含14个样本，包括9个正样本和5个负样本，并且是一个二分类问题，那么当前信息熵的计算如下²：

$$Ent(D) = -\frac{9}{14} \log_2 \frac{9}{14} - \frac{5}{14} \log_2 \frac{5}{14} = 0.940286$$

接下来以表中的Outlook属性作为分支标准，根据sunny、overcast、rain这三个属性值可将数据分为三类，如下图所示：



引入该分支标准后，数据被分成了3个分支，每个分支的信息熵计算如下：

$$H(D^{sunny}) = -\frac{2}{5} \log_2 \frac{2}{5} - \frac{3}{5} \log_2 \frac{3}{5} = 0.970951$$

$$H(D^{overcast}) = -\frac{4}{4} \log_2 \frac{4}{4} = 0$$

$$H(D^{rain}) = -\frac{2}{5} \log_2 \frac{2}{5} - \frac{3}{5} \log_2 \frac{3}{5} = 0.970951$$

因此，基于该分支标准所带来的信息增益为：

$$Gain(D, a) = Ent(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} Ent(D^v) = 0.940286 - \frac{5}{14} \cdot 0.970951 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.970951 =$$

假设以outlook为分支标准能带来最大的信息增益，则我们将数据集D一分为3(sunny, overcast, rain), D1,D2,D3;对D1,D2,D3分别重复原来D的过程，寻找一个集合Di(i=1,2,3)以及一个属性B，使得以B为属性分割Di的信息增益最大。每次分割一个叶子节点，把叶子节点作为根节点，选择的属性属性值作为新的叶子节点，来一步步的构建决策树。

C4.5

C4.5算法弥补了ID3算法的不足，通过使用信息增益率来作为特征的选择标准。

$$g_R(D, A) = \frac{g(D, A)}{H(A)}$$

其中 $H(A) = \sum_{i=1}^K p_i \ln p_i$, K是特征A能取不同值得数目, p_i 是取不同值得概率。也就是属性A固有的一个量度，如果属性A取值越多，该值越到，因此C4.5克服了ID3的两个缺陷：

1. 用信息增益选择属性时偏向于选择分枝比较多的属性值，即取值多的属性
2. 不能处理连续属性

模型的剪枝

为什么需要剪枝：提高泛化能力。

我们可以一步步的细分节点而使得系统的分类误差很小，但是这只是训练数据集上的结果，当我们把模型运用到测试集时，误差率会很高，这是因为模型过拟合你，我们可以通过对决策树进行剪枝来提高模型的泛化能力。

为了进行剪枝，我们得定义剪枝的标准，也就是定义损失函数，损失函数至少要包括两项，一个是模型在训练集上的误差，一个是模型的复杂程度。模型的复杂程度可以定义成与叶子节点正相关。

模型在训练集上的误差率可以如下定义。

$$C(T) = \sum_{t=1}^{t=|T|} N_t H_t$$

其中 N_t 是叶子节点t上的样本点数目, H_t 是叶子节点t的熵。

叶子节点的熵定义为:

$$H_t = -\sum_{k=1}^{k=K} \frac{N_{tk}}{N_t} \ln \frac{N_{tk}}{N_t},$$

其中 $k \in (1, 2 \dots K)$ 是样本结果可以取不同值的数目, 也就是标签有K类 N_t 是叶子节点t的样本数目,

N_{tk} 是叶子节点t中标签属于k类的数目。总体而言, $C(T)$ 刻画的是模型在训练集上的误差。结合这两部分, 我们可以定义剪枝的损失函数:

$$C_\alpha(T) = C(T) + \alpha|T|$$

其中正数 α 用来权衡模型的误差率与模型的复杂度。

输入: 计算生成的整个树T, 参数 α

输出: 修剪后的子树 T_α

(1)计算每个节点的经验熵

(2)递归地从树的叶节点向上回溯

设一组叶节点回溯到父节点之前与之后的整个树为 T_B, T_A , 对应的损失函数值分别为 $C_\alpha(T_B)$ 和

$C_\alpha(T_A)$, 如果:

$$C_\alpha(T_A) \leq C_\alpha(T_B)$$

则进行剪枝, 将父节点变成新的叶节点。

(3)返回(2), 直到不能继续为止, 得到损失函数最小的子树 T_α

局部加权平均

$$L(\theta) = \frac{1}{N} \sum_i w_i (\hat{y}_i - y_i)^2$$

其中: $w_i = \exp\left(\frac{x_i - \bar{x}}{2\sigma^2}\right)$

CART

决策树的生成就是递归的构建二叉决策树的过程。对回归树用平方误差最小的准则, 对分类树用基尼指数最小化准则, 进行特征选择, 生成二叉树。

CART包含分类树与回归树, 下面分别来讨论分类树与回归树

回归树的生成

假设X和Y分别为输入与输出变量，并且Y是连续变量，给定训练数据集

$$D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N)).$$

假设已将输入空间划分为M个单元 R_1, R_2, \dots, R_M ，并且每个单元 R_m 上有一个固定的输出值 c_m ，于是回归树模型可以表示为：

$$f(x) = \sum_{m=1}^{m=M} c_m I(x \in R_m)$$

当输入空间的划分确定时，可以用平方误差

$$\sum_{x_i \in R_m} (y_i - f(x_i))^2$$

来表示回归树对于训练数据的预测误差，用平方误差最小的准则来求解每个单元上的最优输出值。可以求得

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m).$$

问题是，怎么对输入的空间进行划分？

回归树可以处理离散特征与连续特征，对于连续特征，若这里按第j个特征的取值s进行划分，切分后的两个区域分别为：

$$R_1(j, s) = (x_i | x_i^j \leq s)$$

$$R_2(j, s) = (x_i | x_i^j > s)$$

若为离散特征，则找到第j个特征下的取值s：

$$R_1(j, s) = (x_i | x_i^j = s)$$

$$R_2(j, s) = (x_i | x_i^j \neq s)$$

分别计算 R_1, R_2 的类别估计 c_1, c_2 ，然后计算按照(j,s)切分后的损失：

$$\min_{j, s} [\sum_{x_i \in R_1} (y_i - c_1)^2 + \sum_{x_i \in R_2} (y_i - c_2)^2],$$

找到的是损失最小的(j,s)对即可，也就是说找到最优特征 j^* ，并找到最优特征的分割值 s^* 。递归执行(j,s)的选择过程，知道满足停止条件为止。递归的意思是，对分割出的两个区域 R_1, R_2 分别进行如上的步骤，再分割下去。

总结：

回归树算法：

输入：训练集 $D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N)).$

输出：回归树T

1. 求解选择的切分特征j与切分特征取值s，j将训练集D划分成两部分， R_1, R_2 ，

$$R_1(j, s) = (x_i | x_i^j \leq s)$$

$$R_2(j, s) = (x_i | x_i^j > s)$$

其中：

$$c_1 = \frac{1}{N_1} \sum_{x_i \in R_1} y_i$$

$$c_2 = \frac{1}{N_2} \sum_{x_i \in R_2} y_i$$

2. 遍历所有的可能解(j,s), 找到最优的(j^*, s^*), 最优解使得如下损失函数最小,

$$\min_{j,s} [\sum_{x_i \in R_1} (y_i - c_1)^2 + \sum_{x_i \in R_2} (y_i - c_2)^2]$$

按照最优特征(j^*, s^*)来切分即可

3. 递归调用1., 2. 步骤, 直到满足停止条件

4. 将输入空间划分为M个区域 R_1, R_2, \dots, R_M , 返回决策树T

$$f(x) = \sum_{m=1}^{m=M} \hat{c}_m I(x \in R_m)$$

回归树采用了分治策略, 对于无法用唯一的全局线性回归来优化的目标进行分而治之, 进而可以取得比较准确的结果, 但分段取均值并不是一明智的选择, 可以考虑将叶节点设置为一个线性函数, 这就是所谓的分段线性树模型。

分类树

分类树是CART中用来分类的, 不同于ID3与C4.5, 分类树采用基尼系数来选择最优的切分特征, 而且每次都是二分。

输入: 训练集 $D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N))$.

输出: 回归树T

1) 利用特征A的取值a将数据分为两部分, 计算 $A=a$ 时的基尼系数

$$Gini(D, A) = \frac{|D_1|}{D} Gini(D_1) + \frac{|D_2|}{D} Gini(D_2) \text{ 其中}$$

$$D_1(A, a) = (x_i | x_i^A = a)$$

$$D_2(A, a) = (x_i | x_i^A \neq a)$$

2) 对整个数据集中所有的可能特征A以及其可能取值a选取基尼系数最小的特征 A^* 与特征下的取值 a^* , 来将数据集切分, 将数据集 D_1, D_2 分到两个子节点中去。

3. 对子节点(D_1, D_2)递归调用1., 2. 步骤, 直到满足停止条件

4. 返回CART分类树T

CART的剪枝¹

剪枝基于如下的损失函数

$$C_\alpha(T) = C(T) + \alpha|T|$$

其中T为任意子树, $|T|$ 为子树的叶子结点个数, $C(T)$ 为对训练数据的预测误差(如基尼系数), 正数 α 用来权衡模型的误差率与模型的复杂度。

对于固定的 α , 存在唯一的最有子树 $C_\alpha(T)$.

Breiman等人证明, 可以用递归的方法对树进行剪枝, 将 α 从0开始增大, $0 \leq \alpha_1 \leq \alpha_2 \leq \dots \leq \alpha_n$.

得到一系列最优子树系列 (T_0, T_1, T_n) , 序列中的子树是嵌套的(?)。

具体的, 从整棵树 T_0 开始剪枝, 对 T_0 的任意内部节点 t , 以 t 为单节点树的损失函数是(也就是把 t 下面的所有节点减掉之后的损失函数):

$$C_\alpha(t) = C(t) + \alpha$$

以 t 为根结点的字数 T_t 的损失函数是(不进行剪枝时的损失函数):

$$C_\alpha(T_t) = C(T_t) + \alpha|T_t|$$

当 α 小于某个值时, 不剪枝会使得整体的损失函数较小, 也就是:

$$C_\alpha(T_t) < C_\alpha(t)$$

当 α 增大时, 在某一 α 有:

$$C_\alpha(T_t) = C_\alpha(t)$$

当 α 再增大时, 不等式反向, 只要 $\alpha = \frac{C(t) - C(T_t)}{|T_t| - 1}$, T_t 与 t 有相同的损失函数。

为此, 对 T_0 中每一个内部节点 t , 计算:

$$g(t) = \frac{C(t) - C(T_t)}{|T_t| - 1}$$

它表示剪枝之后整体损失函数减小的程度, 在 T_0 中减去 $g(t)$ 最小的 T_t , 将得到的子树作为 T_1 , 同时将最小的 $g(t)$ 设为 α_1 , T_1 为区间 $[\alpha_1, \alpha_2]$ 的最优子树。

如此剪枝下去, 在这一过程中, 不断的增加 α 的取值, 产生新的区间, 最终会的到一组决策树

(T_0, T_1, T_n) , 对应于椅子确定的权衡参数 (a_0, a_1, a_n) , 通过验证集合中每棵树的总体误差, 也就得到了最终的最优决策树 T^* .

T 是整颗决策树吗?

CART剪枝算法:

输入: CART算法生成的决策树 T_0

输出: 最优决策树 T_α

(1) 设 $k=0, T=T_0$

(2) 设 $\alpha = +\infty$

(3) 自下而上地对各内部结点 t 计算 $C(T_t), |T_t|$ 以及:

$$g(t) = \frac{C(t) - C(T_t)}{|T_t| - 1}$$
$$\alpha = \min(\alpha, g(t))$$

这里, T_t 表示以 t 为根结点的子树, $C(T_t)$ 是对训练数据的预测误差, $|T_t|$ 是 T_t 的叶节点个数。

(4) 自上而下地访问内部结点 t , 如果有 $g(t) = \alpha$, 进行剪枝, 并对叶节点 t 以多数表决法决定其类, 得到树 T 。

(5) 设 $k=k+1, \alpha_t = \alpha, T_k = T$

(6) 如果 T 不是由根结点单独生成的树, 则返回步骤(4)

(7) 采用交叉验证法在子树序列 T_0, T_1, \dots, T_n 中选取最优子树 T_α .

CART总结

CART(classification and regression tree)即分类回归树, 意思是它既可以用来做分类页可以用来做回归。

对分类问题, 我们基于基尼系数构造分类树, 其做法与ID3, C4.5一致。

对于回归问题, 我们基于平方误差最小, 构造回归树。构造就是, 基于一个特征(输入样本的一个维度), 选择最佳的切分点, 把当前数据集分成两部分, 分别求两部分输出值的质心(平均值), 再求两部分中的每一部分的输出值到该区域质心的平方误差, 对平方误差求和, 选择最佳切分点, 使得这两部分的误差和最小, 这就是该特征的最佳切分点, 再在该数据集中的所有特征中这个误差, 找到最佳的切分特征, 最后得到最佳特征以及最佳特征的切分点。基于这两个值, 把数据集分成两部分, 再对这两部分分别进行如上的操作(求最佳特征与该特征下的最佳切分点)

1. 决策树之 CART . ↵

2. 决策树(ID3、C4.5、CART) <https://blog.csdn.net/u010089444/article/details/53241218> ↵

集成学习

集成学习就是组合一系列的弱分类器生成强分类器。组合的弱分类器之间的关系有两种：

一种是彼此间相互依赖，一系列学习器之间需要串行生成，代表算法是Boosting系列算法，代表算法是AdaBoost与GBDT。

一种是学习器彼此间无关联，一些列算法可以并行的生成，代表算法是Bagging(Bootstrap Aggregating)和随机森林(Random Forest)系列。随机森林是时Bagging的进行版，随机表现在两个方面，一方面是取样本点是随机的，另一方面，选择特征也是随机的（比如总共有N个特征，随机选取小于N的K个特征）

还有一种是两者的结合物：stacking

PAC, 弱可学习, 强可学习

PAC(Probably approximately correct)

强可学习：如果存在一个多项式的学习算法学习呀，学习的正确率很高

弱可学习：如果存在一个多项式的学习算法学习呀，学习的正确率仅比随机猜测略好
在PAC学习框架下，一个概念是强可学习的充分必要条件是这个概念是弱可学习的。

集成学习的理论基础

我们考虑多个不相干分类器叠加处理二分类的问题， $y \in (-1, +1)$ 和真实的函数 f ，假设基分类器的错误率都是 ϵ ，即对每个分类器都有

$$P(h_i(x) \neq f(x)) = \epsilon$$

假设集成通过简单的投票表决结合 T 个基分类器，若半数的基分类器正确，则集成分类正确。

$$H(x) = \text{sign}(\sum_{i=1}^T h_i(x))$$

价格基分类器之间的错误率是相互独立的，则由Hoeffding不等式可知，集成的错误率为：

$$P(H(x) \neq f(x)) = \sum_{i=0}^{[T/2]} \binom{T}{k} (1 - \epsilon)^k \epsilon^{T-k} \leq \exp(-\frac{1}{2} T (1 - 2\epsilon)^2)$$

上式表明，随着集成中个体分类器数目 T 的增大，集成的错误率将指数下降，最终趋向于0。

Bagging

Bagging系列方法的原理图如下：



Bagging的本质思想是平均一个个多noisy(variance大)但是近似无偏的模型, 因此可以减少variance。树是bagging理想的候选者, 因为他们能抓住数据中复杂的相互作用结构。在大多数问题中, boosting相对于bagging有压倒性优势, 成为更好的选择。

Boosting通过弱学习者的时间演化, 成员投一个带权重的票。

$$\text{Regression: } \hat{f}_{rf}^B(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$$

因为从bagging中生成的树是id(identically distributed), B颗树的期望与单颗树的期望一样, 如果输出之间是IID(independent identically distributed), 每颗树的variance是 σ^2 , 则B颗树的variance是 $\frac{\sigma^2}{B}$, 假设树之间的关联系数是c, 则B颗树的variance是:

$$c\sigma^2 + \frac{1-c}{B}\sigma^2$$

如果c=1, 就回到iid情况, 如果c ≠ 0, 则上面的第一项不会随着B增大而减小, 只有第二项会随着B增大而减小, 因此我们要尽量使得树之间的关联系数c趋于0. 这可以通过随机化来实现, 对样本和对特征这两方面随机化。

问题: 怎么构造具有负关联系数的系统? $\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}^*(x)$

当B趋于无穷时, 上述表达式就是bagging 估计的一个Monte Carlo估计。

Random Forest

随机森林的两个随机:

1. 随机挑选一部分训练样本。
2. 随机挑选一部分特征。

这样保证每颗子树尽量是不相关的。因此误差也是不相关的, 就可以通过多个模型平均来减小

整个模型的误差。

参数推荐：

对于分类问题，默认m是 \sqrt{p} , 最小的节点数是1.

对于回归问题，默认的m是 $p/3$, 最小的节点尺寸是5.

OOB:out of box, bagging中有 $(1 - \frac{1}{N})^N = 1/e$ 的样本没有被选中，可以用来做验证，因此不需要交叉验证。当OOB误差稳定后，训练也就可以结束了。

Kernel random forest

Boosting

Boosting系列方法的原理图如下：



By 刘建平Pinard

在这里，我们将介绍GBDT, XGBoost以及LightGBM。

AdaBoost

Boosting系列算法的代表就是AdaBoost。

算法如下：

输入：训练数据集 $T = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N))$, 其中 $x_i \in X \subseteq R^n, y \in (-1, +1)$

输出：最终的分类器 $G(x)$

(1) 初始化训练数据集的权重分布

$$D_1 = (w_{11}, \dots, w_{1i}, \dots, w_{1N}),$$

$$w_{1i} = \frac{1}{N}, i = 1, 2, \dots, N$$

(2) 对 $m=1, 2, \dots, M$

使用具有权值分布 D_m 的训练数据集进行训练，得到基分类器

$$G_m(x) : x \in (-1, +1)$$

(b) 计算 $G_m(x)$ 在训练集上的分类误差率：

$$e_m = P(G_m(x_i) \neq y_i) = \sum_{i=1}^N w_{mi} I(G_m(x_i) \neq y_i)$$

(c) 计算 $G_m(x)$ 的系数

$$\alpha_m = \frac{1}{2} \log \frac{1-e_m}{e_m}$$

这里用的是自然对数。

(d)更新训练数据集的权值分布：

$$D_1 = (w_{m+1,1}, \dots, w_{m+1,i}, \dots, w_{m+1,N}),$$

$$w_{m+1,i} = \frac{w_{mi}}{Z_m} \exp(-\alpha_m y_i G_m(x_i)), i = 1, 2, \dots, N$$

这里, Z_m 是归一化因子:

$$Z_m = \sum_{i=1}^N w_{mi} \exp(-\alpha_m y_i G_m(x_i))$$

它使 D_{m+1} 成为一个概率分布。

(3)构建基本分类器的线性组合:

$$f(x) = \sum_{i=1}^N \alpha_m G_m(x)$$

得到最终分类器

$$G(x) = \text{sign}(f(x)) = \text{sign}\left(\sum_{i=1}^N \alpha_m G_m(x)\right)$$

由 α_m 的定义可以知道, 第m个分类器的误差率越小, 则 α_m 的值越大, 因此第m个分类器在总的分类器中占的比重越大。因此, AdaBoost会加大那些准确率很高的分类器的权重。

同时, 在第m+1个分类器求解时, 对于上一轮被分错的样本的权值会变大。

不改变所给的训练数据, 而不断的改变训练数据权值的分布, 使得训练数据在基本分类器的学习中起不同的作用, 这是AdaBoost的一个特点。

关键的两点:

1. 加大上一轮分错的样本的权重。
2. 加大分类错误率低的基本分类器的权重。

可以参考《统计机器学习》8.1节的例子来加深理解。

AdaBoost算法的训练误差分析

定理:(AdaBoost的训练误差界)

AdaBoost 算法最终分类器的训练误差界为:

$$\frac{1}{N} \sum_{i=1}^N I(G_m(x_i) \neq y_i) \leq \frac{1}{N} \sum_{i=1}^N \exp(-y_i f(x_i)) = \prod_m Z_m$$

定理(二分类问题AdaBoost的训练误差界)

$$\prod_m Z_m = \prod_{m=1}^M [2 \sqrt{\frac{e_m(1-e_m)}{1-4\gamma_m^2}}] = \prod_{m=1}^M \sqrt{\frac{1-4\gamma_m^2}{1-4\gamma_m^2}} \leq \exp\left(-2 \sum_{m=1}^M \gamma_m m^2\right)$$

其中: $\gamma_m = \frac{1}{2} - e_m$

梯度提升

提升树是以分类树或者回归树为基本分类器的提升方法, 提升树被认为是统计学习中性能最好的方法之一。

提升树实际采用加法模型(即基函数的线性组合)与前向分步算法。以决策树为基函数的提升方法称为提升树。对分类问题决策树是二叉分类树, 对回归问题决策树是二叉回归树。

决策树桩(decision stump): 可以看成是一个根节点直接连接两个叶节点的简单决策树。提升树模型可以表示为决策树的加法模型。

$$f_M(x) = \sum_{m=1}^M T(x; \Theta_m)$$

其中: $T(x, \Theta_m)$ 表示决策树; Θ_m 为决策树的参数; M 为树的个数。

梯度提升算法:

提升树算法采用前向分步算法。首先缺点初始提升树 $f_0(x) = 0$, 第 m 步的模型是:

$$f_m(x) = f_{m-1}(x) + T(x; \Theta_m)$$

其中, $f_{m-1}(x)$ 为当前模型, 通过经验风险极小化来确定下一刻决策树的参数 Θ_m

$$\hat{\Theta}_m = \operatorname{argmin}_{\Theta_m} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)).$$

对于一个训练数据集 $T = ((x_1, y_1), (x_2, y_2), \dots, (x_N, y_N))$, $x_i \in \chi \in R^N$, χ 为输入控件, $y_i \in R$, 为输出空间。如果将输入空间 χ 划分为 J 个互不相交的区域 R_1, R_2, \dots, R_J , 并且在每个区域上确定输出的常量 c_j , 那么树可以表示为:

$$T(x; \theta) = \sum_{j=1}^J c_j I(x \in R_j).$$

其中, 参数 $\Theta = ((R_1, c_1), (R_2, c_2), \dots, (R_J, c_J))$ 表示树的区域划分和各区域上的常数。 J 是回归树的复杂度, 即叶节点个数。

回归问题提升树使用以下前向分步算法:

$$f_0(x) = 0$$

$$f_m(x) = f_{m-1}(x) + T(x; \Theta_m), m = 1, 2, \dots, M$$

$$f_m(x) = \sum_{m=1}^M T(x; \Theta_m)$$

在前向分步算法的第 m 步, 给定当前模型 $f_{m-1}(x)$, 需求解:

$$\hat{\Theta}_m = \operatorname{argmin}_{\Theta_m} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)).$$

得到 $\hat{\Theta}_m$, 即第 m 颗树的参数。

采用平方误差损失函数时:

$$L(y, f(x)) = (y - f(x))^2$$

其损失变成:

$$L(y, f_{m-1}(x) + T(x; \Theta_m)) = (y - f_{m-1}(x) - T(x; \Theta_m))^2 = [r - T(x; \Theta_m)]^2$$

这里,

$$r = y - f_{m-1}(x)$$

是当前模型拟合数据的残差。所以, 对回归问题的提升树算法来说, 只需要简单地拟合当前模型的残差, 也就是下一步只是拟合前面一步的残差, 类似于 Resi-Net。

GBDT

当损失函数是平方损失和指数损失函数时，每一步优化是很简单的，但是对于一般损失函数而言，往往每一步优化并不容易。针对这一问题，Freidman提出了梯度梯度提升(gradient boosting)算法，这是利用最速下降法的近似方法，其关键是利用损失函数的负梯度在当前模型残差的近似值。而在平方损失中，拟合的是残差，在这里拟合的是模型上一步的梯度。

$$-[\frac{\partial L(y, f(x_i))}{\partial f(x_i)}]_{f(x)=f_{m-1}(x)}$$

作为回归问题提升算法中的残差的近似值，拟合一个回归树。

梯度提升树算法

输入：训练数据集 $T = ((x_1, y_1), (x_2, y_2), \dots, (x_N, y_N))$, $x_i \in \chi \in R^N$, χ 为输入, $y_i \in R$, 损失函数为 $L(y, f(x))$:

输出：回归树 $f(x)$.

(1) 初始化

$$f_0(x) = \operatorname{argmin}_c \sum_{i=1}^N L(y_i, c).$$

(2) 对 $m=1, 2, \dots, M$

(a) 对 $i=1, 2, \dots, N$, 计算

$$r_{mi} = -[\frac{\partial L(y, f(x_i))}{\partial f(x_i)}]_{f(x)=f_{m-1}(x)}$$

(b) 对 r_{mi} 拟合一个回归树，得到第 m 课树叶节点区域 R_{mj} , $j = 1, 2, \dots, J$

(c) 对 $j=1, 2, \dots, J$, 计算

$$c_{mj} = \operatorname{argmin}_c \sum_{x_i \in R_{mj}} L(y_i, f_{m-1}(x_i) + c).$$

$$(d) \text{更新 } f_m(x) = f_{m-1}(x) + \sum_{j=1}^J c_{mj} I(x \in R_{mj}), m = 1, 2, \dots, M$$

(3) 得到回归树

$$f(x) = f_M(x) = \sum_{m=1}^M \sum_{j=1}^J c_{mj} I(x \in R_{mj})$$

函数空间的数值优化：

XGBoost¹

模型复杂度惩罚是XGBoost相对于MART的提升。

$$\Omega(f^{(m)}) = \sum_{m=1}^M [\gamma T_m + \frac{1}{2} \lambda ||w_m||_2^2 + \alpha ||w_m||_1]$$

MART includes row subsampling, while XGBoost includes both row and column subsampling

Newton boosting

Input: Data set D, A loss function L, A base learner L_Φ , the number of iterations M. The learning rate η

1. Initialize $\hat{f}^{(0)}(x) = \hat{f}_{(0)}(x) = \hat{\theta}_0 = \operatorname{argmin}_\theta \sum_{i=1}^n L(y_i, \theta)$;
2. for m = 1,2,...,M do
3. $\hat{g}_m(x_i) = [\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}]_{f(x)=f^{(m-1)}(x)}$
4. $\hat{h}_m(x_i) = [\frac{\partial^2 L(y_i, f(x_i))}{\partial f(x_i)^2}]_{f(x)=f^{(m-1)}(x)}$
5. $\hat{\phi}_m = \operatorname{argmin}_{\phi \in \Phi} \sum_{i=1}^n \frac{1}{2} \hat{h}_m(x_i) [(-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}) - \phi(x_i)]^2$
6. $\hat{f}_m(x) = \eta \hat{\phi}_m(x);$
7. $\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x);$
8. end

$$\text{Output: } \hat{f}(x) = \hat{g}^{(M)}(x) = \sum_{m=1}^M \hat{f}_m(x)$$

对于没有惩罚项的Netwon tree boosting(NTB),每一次迭代都市最小化如下损失函数:

$$J_m(\phi_m) = \sum_{i=1}^n L(y_i, \hat{f}^{(m-1)}(x_i) + \phi_m(x_i))$$

对于梯度提升算法, 基函数是如下的树:

$$\phi_m(x) = \sum_{j=1}^T w_{jm} I(x \in R_{jm})$$

这里的T是第m颗树叶子节点的个数, w_{jm} 是第m颗树中第j个叶子节点的权重。

对上式进行二阶泰勒展开,并忽略掉常数项可以得到:

$$J_m(\phi_m) = \sum_{i=1}^n [\hat{g}_m(x_i) \phi_m(x_i) + \frac{1}{2} \hat{h}_m(x_i) \phi_m(x_i)^2]$$

代入:

$$\phi_m(x) = \sum_{j=1}^T w_{jm} I(x \in R_{jm})$$

$$J_m(\phi_m) = \sum_{i=1}^n [\hat{g}_m(x_i) \sum_{j=1}^T w_{jm} I(x \in R_{jm}) + \frac{1}{2} \hat{h}_m(x_i) (\sum_{j=1}^T w_{jm} I(x \in R_{jm}))^2]$$

由于每个样本点只能属于一个叶子节点, 因此:

$$I(x \in R_{jm}) I(x \in R_{im}) = \delta_{ij}$$

因此上面可以简化成:

$$\begin{aligned} J_m(\phi_m) &= \sum_{i=1}^n [\hat{g}_m(x_i) \sum_{j=1}^T w_{jm} I(x \in R_{jm}) + \frac{1}{2} \hat{h}_m(x_i) \sum_{j=1}^T w_{jm}^2 I(x \in R_{jm})] \\ &= \sum_{j=1}^T \sum_{i \in I_{jm}} [\hat{g}_m(x_i) w_{jm} + \frac{1}{2} \hat{h}_m(x_i) w_{jm}^2] \end{aligned}$$

定义:

$$G_{jm} = \sum_{i \in I_{jm}} \hat{g}_m(x_i)$$

$$H_{jm} = \sum_{i \in I_{jm}} \hat{h}_m(x_i)$$

因此, 可以把cost function写成:

$$\begin{aligned} J_m(\phi_m) &= \sum_{j=1}^T [G_{jm}w_{jm} + \frac{1}{2}H_{jm}w_{jm}^2] \\ &\geq -\sum_{j=1}^T \frac{1}{2} \frac{G_{jm}^2}{H_{jm}} \end{aligned}$$

成立条件是:

$$w_{jm} = -\frac{G_{jm}}{H_{jm}}, j = 1, 2, \dots, T$$

为了寻找最佳分裂点j, 也就是最大化如下的Gain:

$$Gain = \frac{1}{2} [\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}]$$

总结起来: Newton tree boosting算法如下:

Input: Data set D, A loss function L, A base learner L_Φ , the number of iterations M. The learning rate η

1. Initialize $f^{(0)}(x) = \hat{f}_{(0)}(x) = \hat{\theta}_0 = \operatorname{argmin}_\theta \sum_{i=1}^n L(y_i, \theta);$
2. for m = 1,2,...,M do
3. $\hat{g}_m(x_i) = [\frac{\partial L(y, f(x_i))}{\partial f(x_i)}]_{f(x)=f^{(m-1)}(x)}$
4. $\hat{h}_m(x_i) = [\frac{\partial^2 L(y, f(x_i))}{\partial f(x_i)^2}]_{f(x)=f^{(m-1)}(x)}$
5. Determin the structure $\hat{R}_{jm}, j = 1, \dots, T$ by selecting splits which maximize

$$Gain = \frac{1}{2} [\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}]$$

6. Determine the leaf weights $w_{jm}, j = 1, \dots, T$ for the learnt structure by

$$\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}, j = 1, 2, \dots, T$$

7. $\hat{f}_m(x) = \eta \sum_{j=1}^T \hat{w}_{jm} I(x \in \hat{R}_{jm});$

8. $\hat{f}^m(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x);$

9. end

$$\text{Output: } \hat{f}(x) = \hat{g}^{(M)}(x) = \sum_{m=1}^M \hat{f}_m(x)$$

L2正则化

当我们对损失函数加入树的复杂度以及树的权重的L2正则化的时候(不考虑L1),

$$\begin{aligned} J_m(\phi_m) &= \sum_{j=1}^T [G_{jm} w_{jm} + \frac{1}{2}(H_{jm} + \lambda)w_{jm}^2] \\ &\geq - \sum_{j=1}^T \frac{1}{2} \frac{G_{jm}^2}{H_{jm} + \lambda} \end{aligned}$$

成立条件是:

$$w_{jm} = -\frac{G_{jm}}{H_{jm} + \lambda}, j = 1, 2, \dots, T$$

为了寻找最佳分裂点j, 也就是最大化如下的Gain, 也就是原来的cost 减去分裂后的cost:

$$Gain = \frac{1}{2} [\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}] - \lambda$$

RF,GBDT,XGBOOST,LightGBM比较

RF

随机森林(RF)是Bagging的扩展变体, 它在以决策树为基学习器构建Bagging集成的基础上, 进一步在决策树的训练过程中引入随机特征选择, 主要包含两个特征:

1. 随机的有放回的样本选择
2. 随机选择特征

其原理是通过减少通过投票来减少整体的方差。虽然每个基分类器的bias大, 但是通过多个彼此不相关的基分类器能减少总体的方差。

优点:

1. 数据集合上表现良好, 相对于其他算法有较大的优势(训练速度, 预测准确度)
2. 能够处理高纬度的数据, 并且不用特征选择, 而且在训练完后, 给出特征的重要性
3. 容易做并行处理

缺点:

1. 在噪声较大的数据集上, 分类与回归问题会出现过拟合。

GBDT

传统的Boosting算法, 比如AdaBoost主要关注的是正确错误样本的权重, 而GBDT的每一次计算都是为了减少上一次的残差, 而为了消除残差, 我们可以在残差减小的梯度方向建立模型, 在 GradientBoost中, 每个新的模型的建立时为了使得之前的模型的残差往梯度下降的方法。

在GradientBoosting算法中, 关键即使利用损失函数的负梯度方向在当前模型的值作为残差的近似值, 进而拟合棵CART回归树。

GBDT会累加所有树的结果, 而这种累加是无法通过分类完成的, 因此GBDT的树都是CART回归树, 而不是分类树(尽管GBDT调整后也可以用于分类, 但不代表GBDT的树是分类树)

GBDT的优点:

GBDT的性能在RF的基础上又有一步提升, 因此其优点也很明显:

1. 它能灵活的处理各种类型的数据
2. 在相对较少的调参时间下, 预测的准确度较高

缺点:

由于它是Boosting, 因此基学习器之间存在串行关系, 难以并行训练数据。

XGBoost

XGBoost相对于GBDT的优势在于：

1. 传统的GBDT在优化的时候，之利用了一阶信息，而XGBoost对代价函数进行二阶泰勒展开，利用了二阶信息²
2. XGBoost在模型中加入了正则项，用来控制模型的复杂程度，使得学出来的模型更简单，防止过拟合。
3. 列抽样，XGBoost借鉴了RF做法，支持列抽样，不仅防止过拟合，而且能减少计算量。
4. 对缺失值的处理(Why?)。对于特征的值有缺失的样本，XGBoost还可以自动学习出它的分裂方向；
5. XGBoost工具支持并行(Why?)。注意XGBoost的并行不是tree粒度的并行，XGBoost也是一次迭代完才能进行下一次迭代的(第t次迭代的代价函数里包含了前面t-1次迭代的预测值)。
XGBoost的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序(因为要确定最佳分割点)，XGBoost在训练之前，预先对数据进行了排序，然后保存为block结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个block结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行¹。

LightGBM

LightGBM的优化点

1. 采用直方图算法⁴
2. 树的生长策略优化
3. 相对于xgboost和GBDT, LightGBM提出了两个新方法，使得LightGBM的效率要显著要高于GBDT和xgboost。这两种新方法是：Gradient-based One-Side Sampling (GOSS: 基于梯度的one-side采样) 和Exclusive Feature Bundling (EFB: 互斥的特征捆绑)

这两种新方法：

1. GOSS(从减少样本角度)：排除大部分小梯度的样本，仅用剩下的样本计算信息增益。
2. EFB(从减少特征角度)：捆绑互斥特征，也就是他们很少同时取非零值(也就是用一个合成特征代替)。

GBDT是基于决策树的集成算法，采用前向分布算法，在每次迭代中，都是通过负梯度拟合残差，从而学习一颗决策树，最耗时的步骤就是找最优划分点。一种流行的方法就是预排序，核心是在已经排好序的特征值上枚举所有可能的特征点。另一种改进则是直方图算法，它把连续特征值划分到k个桶中去，划分点则在这k个点中选取。 $k \leq d$ 所以在内存消耗和训练速度都更佳，且在实际的数据集上表明，离散化的分裂点对最终的精度影响并不大，甚至会好一些。原因在于决策树本身就是一个弱学习器，采用Histogram算法会起到正则化的效果，有效地防止模型的过拟合。LightGBM也是基于直方图的。

直方图算法(Histogram)

直方图算法是先把连续的浮点特征值离散化成k个整数，同时构造一个宽度为k的直方图。遍历数据时，根据离散化后的值作为索引在直方图中累积统计量，当遍历一次数据后，直方图累积了需要的统计量，然后根据直方图的离散值，遍历寻找最优的分割点。它的优点如下：直方图只需对直方图统计量计算信息增益，相比较于预排序算法每次都遍历所有的值，信息增益的计算量要小很多。通过利用叶节点的父节点和相邻节点的直方图的相减来获得该叶节点的直方图，从而减少构建直方图次数，提升效率。存储直方图统计量所使用的内存远小于预排序算法。

树的生长策略优化

LightGBM 通过 leaf-wise (best-first) 策略来生长树。它将选取具有最大信息增益最大的叶节点来生长。当生长相同的叶子时，leaf-wise 算法可以比 level-wise 算法减少更多的损失。
当数据较小的时候，leaf-wise 可能会造成过拟合。所以，LightGBM 可以利用额外的参数 `max_depth` 来限制树的深度并避免过拟合（树的生长仍然通过 leaf-wise 策略）。

GOSS(Gradient-based One-Side Sampling)

在AdaBoost中采用权重很好诠释了样本的重要性，GBDT没有这种权重，但是我们注意到每个数据样本的梯度可以被用来做采样的信息。也就是，如果一个样本的梯度小，那么表明这个样本已经训练好了，它的训练误差很小了，我们可以丢弃这些数据。当然，改变数据分布会造成模型的精度损失。GOSS则通过保存大梯度样本，随机选取小梯度样本，并为其弥补上一个常数权重。这样，GOSS更关注训练不足的样本，同时也不会改变原始数据太多。²³

EFB(Exclusive Feature Bundling)

高维数据一般是稀疏的，可以设计一种损失最小的特征减少方法。并且，在稀疏特征空间中，许多特征都是互斥的，也就是它们几乎不同时取非0值。因此，我们可以安全的把这些互斥特征绑到一起形成一个特征，然后基于这些特征束构建直方图，这样又可以加速了。
EFB算法可以把很多特征绑到一起，形成更少的稠密特征束，这样可以避免对0特征值的无用的计算。加速计算直方图还可以用一个表记录数据的非0值。

1. RF、GBDT、XGBoost面试级整理

[https://blog.csdn.net/qq_28031525/article/details/70207918] ↵

2. 『论文阅读』LightGBM原理-LightGBM

[<https://blog.csdn.net/shine19930820/article/details/79123216>] ↵

3. LightGBM原理之论文详解 <https://zhuanlan.zhihu.com/p/35155992> ↵

4. 机器学习——LightGBM <https://www.cnblogs.com/wkslearner/p/9333168.html> ↵

生成算法

1. 贝叶斯方法
2. HMM
3. DMM-HMM混合系统

贝叶斯方法

贝叶斯公式

首先我们讨论贝叶斯公式，以及先验分布于后验分布¹。

$$p(\theta|\mathbf{X}) = \frac{p(\mathbf{X}|\theta)p(\theta)}{p(\mathbf{X})}$$

其中： $p(\theta)$ 是先验分布， $p(\mathbf{X}|\theta)$ 是似然值， $p(\theta|\mathbf{X})$ 是后验分布。在贝叶斯体系中，模型参数 θ 不再认为是固定不变的量，而是服从一定分布的随机变量。在没有数据支持的情况下，我们对其有一个假设性的分布 $p(\theta)$ ，这个分布称为先验分布，而在观测到数据集 $X = [x_1, \dots, x_n]$ 以后，根据数据集上表现出来的似然值 $p(\mathbf{X}|\theta)$ ，可以得到调整后的后验分布 $p(\theta|\mathbf{X})$ 。我们想要的求得的模型参数，要使得观测数据集出现的概率极大。这就是求参数 θ 的极大似然方法。通常通过迭代方法来求极大化 $L(\theta|X)$ ，比如EM算法。

概率统计模型有两个常见的任务：一是参数估计，二是预测，也就是在给定一组训练数据 \mathbf{X} ，评估某新的观测数据 \mathbf{o} 的概率。

若干常见模型估计方法

模型估计方法	参数估计	预测
最大似然估计	$\hat{\theta}_X^{ML} = \arg \min_{\theta} p(\mathbf{X} \theta)$	$p(\mathbf{o} \mathbf{X}) = p(\mathbf{o} \hat{\theta}_X^{ML})$
贝叶斯方法	$p(\theta \mathbf{X}) = p(\mathbf{X} \theta)p(\theta)$	$p(\mathbf{o} \mathbf{X}) = \int p(\mathbf{o} \theta)p(\theta \mathbf{X})d\theta$
最大后验概率方法	$\hat{\theta}_X^{MAP} = \arg \min_{\theta} p(\theta \mathbf{X})$	$p(\mathbf{o} \mathbf{X}) = p(\mathbf{o} \hat{\theta}_X^{MAP})$

朴素贝叶斯

朴素贝叶斯是基于特征条件独立假设。输入变量中所有特征之间是独立的因此，联合条件概率是各个条件概率的乘积。

输入： $X \in R^n$

输出： $Y \in (1, 2, \dots, K)$

条件概率： $P(X = x|Y = C_k) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \dots, X^{(n)} = x^{(n)}|Y = C_k)$, $k=1,2,\dots,K$

条件独立假设：

$$P(X = x|Y = y) = P(X = x|Y = C_k) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \dots, X^{(n)} = x^{(n)}|Y = C_k) \\ = \prod_{j=1}^n P(X^{(j)} = x^{(j)}|Y = C_k)$$

在上面的公式中，先验概率计算如下：

$$P(Y = C_k) = \frac{\sum_{i=1}^N I(y_i=c_k)}{N}, k = 1, 2, \dots, K$$

条件概率计算

假设第j个特征 $x^{(j)}$ 的可能取值集合是 $(a_{j1}, a_{j2}, \dots, a_{js_j})$ ，条件概率计算如下：

$$P(X^{(j)} = a_{jl}|Y = C_k) = \frac{\sum_{i=1}^N I(x_i^{(j)} = a_{jl}, y_i = c_k)}{\sum_{i=1}^N I(y_i = c_k)}$$

$$j = 1, 2, \dots, n; l = 1, 2, \dots, S_j; k = 1, 2, \dots, K$$

后验概率计算

基于贝叶斯定理计算后验概率：

$$P(Y = C_k|X = x) = \frac{P(X=x|Y=C_k)P(Y=C_k)}{\sum_k P(X=x|Y=C_k)P(Y=C_k)}$$

把条件独立假设代入可以得到：

$$P(Y = C_k|X = x) = \frac{P(Y=C_k) \prod_{j=1}^n P(X^{(j)} = x^{(j)}|Y = C_k)}{\sum_k P(Y=C_k) \prod_{j=1}^n P(X^{(j)} = x^{(j)}|Y = C_k)}$$

因为分母对不同的类别k都是一样的，因此只需要求分子就可以了。计算不同的类别k对应的值，分类结果对应概率最大的。

因此朴素贝叶斯可以表示为：

$$y = f(x) = \arg \max_{c_k} P(Y = C_k) \prod_{j=1}^n P(X^{(j)} = x^{(j)}|Y = C_k)$$

Laplace Smoothing

在实际计算条件概率的时候，或出现0的情况，这会影响后面的后验概率的计算。一般通过Laplace Smoothing来处理。

$$P(X^{(j)} = a_{jl}|Y = C_k) = \frac{\sum_{i=1}^N I(x_i^{(j)} = a_{jl}, y_i = c_k) + \lambda}{\sum_{i=1}^N I(y_i = c_k) + s_j \lambda}$$

$$\lambda > 0, j = 1, 2, \dots, n; l = 1, 2, \dots, S_j; k = 1, 2, \dots, K$$

基本上到此，朴素贝叶斯的介绍也就完结了，朴素贝叶斯不需要计算任何参数，计算简单，缺点是分类性能不一定高。

极大似然估计

极大似然估计是试图在所有的模型参数可能取值中，找到一个能使得数据出现的可能性最大的值。

贝叶斯学派与频率学派

以上帝抛N次硬币为例，统计出现正面朝上的次数与概率：

频率学派：认为上帝只有一枚硬币，硬币出现朝上的概率等于当抛的次数N趋于无穷的时候，出现的正面朝上的次数n除以N。频率学派认为上帝只有一枚硬币。

贝叶斯学派：认为上帝有无限多的硬币，所有这些硬币出现正面朝上的概率p满足一个分布P(p)，就是先验分布。每一组N个数据是一枚硬币抛出来的，也就是一组数据对应一个模型，这个模型由p决定。当出现下一组数据的时候，上帝就换了另外一枚硬币，也就是另外一个模型。

1. 《计算广告》刘鹏 p166 ↵

隐马尔科夫模型

定义

HMM类似于量子力学中的路径积分。马尔科夫模型是关于时序的概率模型，描述由一个隐藏的马尔科夫链生成不可观测的状态随机序列，再由各个状态生成一个观测而产生观测随机序列的过程。隐藏的马尔科夫链随机生成的状态的序列成为状态序列(state sequence);每个状态生成一个观测，而由此产生的观测的随机序列成为观测序列(observation sequence).序列的每一个位置又可以看成一个时刻。

马尔科夫模型由初始概率分布，状态转移概率分布以及观测概率分布确定。

设Q是所有可能的状态集合，V是所有可能的观测的集合。

$$Q = (q_1, q_2, \dots, q_N), V = (v_1, v_2, \dots, v_M)$$

其中N是可能的状态数，M是可能的观测数。

I是长度为T的状态序列，O是对应的观测序列。

$$I = (i_1, i_2, \dots, i_T), O = (o_1, o_2, \dots, o_T)$$

A是状态转移矩阵：

$$A = [a_{ij}]_{N \times N}$$

其中：

$$a_{ij} = P(i_{t+1} = q_j | i_t = q_i), i = 1, 2, \dots, N, j = 1, 2, \dots, N$$

是在t时刻处于 q_i 的条件下在时刻t+1转移到 q_j 的概率。

B是观察概率矩阵：

$$B = [b_j(k)]_{N \times M}$$

其中：

$$b_j(k) = P(o_t = v_k | i_t = q_j), k = 1, 2, \dots, M; j = 1, 2, \dots, N$$

是在时刻t处于状态 q_j 的条件下生成观测 v_k 的概率。

π 是初始状态概率向量：

$$\pi = (\pi_i)$$

其中：

$$\pi_i = P(i_1 = q_i), i = 1, 2, \dots, N$$

是在时刻t=1处于状态 q_i 的概率。

马尔科夫模型 λ 可以用三元符号来表示：

$$\lambda = (A, B, \pi)$$

A,B, π 称为马尔科夫模型的三要素。

马尔科夫模型的两个假设：

(1)齐次马尔科夫假设。任一时刻只与前一时刻有关系

$$P(i_t | i_{t-1}, o_{t-1}, \dots, i_1, o_1) = P(i_t | i_{t-1}), t = 1, 2, \dots, T$$

(2)观测独立性假设：任一时刻的观测只依赖于该时刻的马尔科夫链的状态，而与其他观测以及状态没有关系。

$$P(o_t | i_T, o_T, \dots, i_{t+1}, o_{t+1}, i_t, i_{t-1}, o_{t-1}, i_1, o_1) = P(o_t | i_t)$$

HMM的三个问题

1. 概率计算问题：给定模型 $\lambda = (A, B, \pi)$ ，和观测序列 $O = o_1, o_2, \dots, o_T$ ，计算在模型 λ 下观测序列 O 出现的概率 $P(O|\lambda)$ 。
2. 学习问题：已知观测序列 $O = o_1, o_2, \dots, o_T$ ，估计模型 $\lambda = (A, B, \pi)$ ，使得 $P(O|\lambda)$ 最大。即用极大似然法的方法估计参数。
3. 预测问题（也称为解码（decoding）问题）：已知观测序列 $O = o_1, o_2, \dots, o_T$ 和模型 $\lambda = (A, B, \pi)$ ，求给定观测序列条件概率 $P(I|O)$ 最大的序列 $I = (i_1, i_2, \dots, i_T)$ ，即给定观测序列，求最有可能的对应的状态序列。

概率计算问题

直接计算方法：

问题：给定模型 $\lambda = (A, B, \pi)$ ，和观测序列 $O = o_1, o_2, \dots, o_T$ ，计算在模型 λ 下观测序列 O 出现的概率 $P(O|\lambda)$ 。

对于状态序列 $I = (i_1, i_2, \dots, i_T)$ 的概率是：

$$P(I|\lambda) = \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \dots a_{i_{T-1} i_T},$$

对于上面这种状态序列，产生观测序列 $O = o_1, o_2, \dots, o_T$ 的概率是：

$$P(O|I, \lambda) = b_{i_1}(o_1) b_{i_2}(o_2) \dots b_{i_T}(o_T),$$

因此 I 和 O 的联合概率是：

$$P(O, I|\lambda) = P(O|I, \lambda) P(I|\lambda) = b_{i_1}(o_1) b_{i_2}(o_2) \dots b_{i_T}(o_T) \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \dots a_{i_{T-1} i_T}$$

对所有可能的 I 求和，得到：

$$P(O|\lambda) = \sum_I P(O, I|\lambda)$$

直接计算，时间复杂度是 $O(TN^T)$ 。为了降低计算的时间复杂度，引入了前向算法。

前向算法

给定模型 λ , 定义到时刻t部分观测序列为 $O = o_1, o_2, \dots, o_T$ 且状态为 q_i 的概率为前向概率。记作:

$$\alpha_t(i) = P(o_1, o_2, \dots, o_t, i_t = q_i | \lambda)$$

有了前向概率就可以通过递归的方法来计算任何一个观测序列的概率。就如下图所示:

前向算法使用前向概率的概念, 记录每个时间下的前向概率, 使得在递推计算下一个前向概率时, 只需要上一个时间点的所有前向概率即可。原理上也是用空间换时间。这样的时间复杂度是 $O(N^2 T)$ 。



上面就是状态数目为5, 时间 $T=4$ 的图。一个问题是, 前向算法是否可以转化为矩阵求解问题?

$$\alpha_t = (\alpha_t(1), \alpha_t(2), \dots, \alpha_t(N))^T$$

初始状态是 π , 转移矩阵是 A , 因此时刻t的概率分布是一个 $N \times M$ 的矩阵, 也就是N个状态, 每个状态有M个分立值, 计算如下: 问题是怎么降低运算复杂度? N^T 的复杂度是 $O(N^{2.373} \log T)$

后向概率

$$\beta_t(i) = P(o_{t+1}, o_{t+2}, \dots, o_T | i_t = q_i, \lambda)$$

定义 $\beta_T(i) = 1, i = 1, 2, \dots, N$

对 $t = T - 1, T - 2, \dots, 1$ 有:

$$\begin{aligned} \beta_t(i) &= \sum_{j=1}^N a_{ij} b_j(o_{t+1}) \beta_{t+1}(j), i = 1, 2, \dots, N \\ \beta_t &= A^{T-t} \prod_{j=t+1}^T xB[o_j], t = T - 1, T - 2, \dots, 1 \end{aligned}$$

$$\beta_t = (\beta_t(1), \beta_t(2), \dots, \beta_t(N))^T$$

一些概率与期望值

(1) 给定模型 λ 以及观测 O , 在时刻t处于状态 q_i 的概率为:

$$\gamma_t(i) = P(i_t = q_i | O, \lambda) = \frac{P(i_t = q_i, O | \lambda)}{P(O | \lambda)}$$

因为 $\alpha_t(i)\beta_t(i) = P(i_t = q_i, O | \lambda)$

所以: $\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)}$

(2) 给定模型 λ 以及观测 O , 在时刻t处于状态 q_i 且在t+1时刻处以 q_j 的概率为:

$$\xi_t(i, j) = P(i_t = q_i, i_{t+1} = q_j | O, \lambda)$$

$$\xi_t(i, j) = \frac{P(i_t = q_i, i_{t+1} = q_j, O | \lambda)}{P(O | \lambda)} = \frac{P(i_t = q_i, i_{t+1} = q_j, O | \lambda)}{\sum_{i=1}^N \sum_{j=1}^N P(i_t = q_i, i_{t+1} = q_j, O | \lambda)}$$

而:

$$P(i_t = q_i, i_{t+1} = q_j, O | \lambda) = \alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)$$

所以:

$$\xi_t(i, j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}$$

(3) 将 $\gamma_t(i), \xi_t(i, j)$ 对不同时刻t求和, 可以得到一些有用的期望:

(a) 在观测 O 下状态i出现的期望值:

$$\sum_{i=1}^T \gamma_t(i)$$

(b) 在观测 O 下由状态i转移的期望值:

$$\sum_{i=1}^{T-1} \gamma_t(i)$$

(c) 在观测 O 下由状态i转移到状态j的期望值:

$$\sum_{i=1}^{T-1} \xi_t(i, j)$$

学习算法

监督学习方法

我们有观测序列和对应的状态序列 $[(O_1, I_1), (O_2, I_2), \dots, (O_S, I_S)]$

怎么可以通过统计方法来得到转移矩阵A, 观测概率B, 以及初始状态概率 π 。

$$\hat{a}_{ij} = \frac{A_{ij}}{\sum_{j=1}^N A_{ij}}, i = 1, 2, \dots, N; j = 1, 2, \dots, N$$

$$\hat{b}_j(k) = \frac{B_{jk}}{\sum_{k=1}^M B_{jk}}, j = 1, 2, \dots, N; k = 1, 2, \dots, M$$

初始状态概率 π_i 的估计值 $\hat{\pi}_i$ 为S个样本中初始状态为 q_i 的频率。

人工标注的成本很高，因此就会利用非监督学习的方法。

Baum-Welch算法思想

假设给定训练数据只包含S个长度为T的观察序列 (O_1, O_2, \dots, O_s) 而没有对应的状态序列，目标是学习HMM模型 $\lambda = (A, B, \pi)$ 的参数。我们将观测序列数据堪称观察数据O,状态序列数据看作不可观测的隐数据I,那么隐马尔科夫模型事实上是一个含有隐变量的概率模型：

$$P(O|\lambda) = \sum_I P(O|I, \lambda)P(I|\lambda)$$

他的参数学习可以由EM算法来实现。

1. 确定完全数据的对数似然函数

所有观测数据写成 $O = (o_1, o_2, \dots, o_T)$,所有隐数据写成 $I = (i_1, i_2, \dots, i_T)$,完成数据是

$(O, I) = (o_1, o_2, \dots, o_T, i_1, i_2, \dots, i_T)$.完全数据的对数似然函数是 $\log P(O, I|\lambda)$

2. EM算法的E步, 求Q函数 $Q(\lambda, \hat{\lambda})$

$$Q(\lambda, \hat{\lambda}) = E_I [\log P(O, I|\lambda)|O, \hat{\lambda}]$$

$$Q(\lambda, \hat{\lambda}) = \sum_I \log P(O, I|\lambda)P(O, I|\hat{\lambda})$$

其中, $\hat{\lambda}$ 是隐马尔科夫模型参数的当前估计值, λ 是要极大化的马尔科夫模型参数。

$$P(O, I|\lambda) = \pi_{i_1} b_{i_1}(o_1) b_{i_2}(o_2) \dots b_{i_T}(o_T) \pi_{i_1} a_{i_1 i_2} a_{i_2 i_3} \dots a_{i_{T-1} i_T}$$

于是函数 $Q(\lambda, \hat{\lambda})$ 可以写成：

$$Q(\lambda, \hat{\lambda}) = \sum_I \log \pi_{i_1} P(O, I|\hat{\lambda}) + \sum_I \left(\sum_{t=1}^{T-1} \log a_{i_t i_{t+1}} \right) P(O, I|\hat{\lambda}) + \sum_I \left(\sum_{t=1}^{T-1} \log b_{i_t}(o_t) \right) P(O, I|\hat{\lambda})$$

式中求和都是对所有训练数据的序列总长度T进行的。

3. EM算法的M步：极大化Q函数 $Q(\lambda, \hat{\lambda})$ 求模型参数A,B,λ。极大化，满足梯度为0，也就对A, B, λ中的元素分别求导令其为0。

在 $Q(\lambda, \hat{\lambda})$ 中的三项中，分别只包含了 $\pi, a_{ij}, b_j(k)$,因此求解起来比较方便。此外，我们还可以利用概率和为1的限定条件，也就是利用拉格朗日乘子来进行计算。

对 $Q(\lambda, \hat{\lambda})$ 的第一项，我们求 λ :

$$\sum_I \log \pi_{i_1} P(O, I|\hat{\lambda}) = \sum_{i=1}^N \log \pi_i P(O, i_1 = i|\hat{\lambda})$$

因为 π_i 满足约束条件 $\sum_{i=1}^N \pi_i = 1$, 利用拉格朗日乘子, 写出拉格朗日函数:

$$\sum_{i=1}^N \log \pi_i P(O, i_1 = i | \hat{\lambda}) + \gamma \left(\sum_{i=1}^N \pi_i - 1 \right)$$

对其求偏导并令其为0:

$$\frac{\partial}{\partial \pi_i} \left[\sum_{i=1}^N \log \pi_i P(O, i_1 = i | \hat{\lambda}) + \gamma \left(\sum_{i=1}^N \pi_i - 1 \right) \right] = 0$$

得到:

$$P(O, i_1 = i | \hat{\lambda}) + \gamma \pi_i = 0$$

对 i 求和得到 γ :

$$\gamma = -P(O | \hat{\lambda})$$

代入上式, 得到:

$$\pi_i = \frac{P(O, i_1 = i | \hat{\lambda})}{P(O | \hat{\lambda})}$$

对 $Q(\lambda, \hat{\lambda})$ 的第二项, 我们求 a_{ij} :

$$\sum_I \left(\sum_{t=1}^{T-1} \log a_{i_t i_{t+1}} \right) P(O, I | \hat{\lambda}) = \sum_{i=1}^N \sum_{j=1}^{T-N} \sum_{t=1}^{T-1} \log a_{ij} P(O, i_t = i, i_{t+1} = j | \hat{\lambda})$$

利用约束条件 $\sum_{i=1}^N a_{ij} = 1$, 利用拉格朗日乘子可以得到:

$$a_{ij} = \frac{\sum_{t=1}^{T-1} P(O, i_t = i, i_{t+1} = j | \hat{\lambda})}{\sum_{t=1}^{T-1} P(O, i_t = i | \hat{\lambda})}$$

对 $Q(\lambda, \hat{\lambda})$ 的第三项, 我们求 $b_j(k)$:

$$\sum_I \left(\sum_{t=1}^{T-1} \log b_{i_t}(o_t) \right) P(O, I | \hat{\lambda}) = \sum_{j=1}^N \sum_{t=1}^T \log b_j(o_t) P(O, i_t = j | \hat{\lambda})$$

利用 $\sum_{k=1}^M b_j(k) = 1$. 注意只有在 $o_t = v_k$ 时, $b_j(o_t)$ 对 $b_j(k)$ 的偏导数才不为0, 以 $I(o_t = v_k)$ 表示,

求得:

$$b_j(k) = \frac{\sum_{t=1}^T P(O, i_t = j | \hat{\lambda}) I(o_t = v_k)}{\sum_{t=1}^T P(O, i_t = j | \hat{\lambda})}$$

$\hat{\lambda}$ 是当前 (A, B, π) 的估计值, 因此可以通过迭代, 得到 (A, B, π) 的收敛解。

Baum-Welch算法

输入: 观测数据 $O = (o_1, o_2, \dots, o_T)$;

输出: 隐马尔科夫模型的参数

(1) 初始化

对 $n=0$, 选取 $a_{ij}^{(0)}, b_j(k)^{(0)}, \pi_i^{(0)}$ 得到模型 $\lambda^{(0)} = (A^{(0)}, B^{(0)}, \pi^{(0)})$

(2). 递推对 $n=1, 2, \dots$,

$$a_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \xi_t(i)}$$

$$b_j(k) = \frac{\sum_{t=1, o_t=v_k}^T \gamma_t(j)}{\sum_{t=1}^T \gamma_t(j)}$$

$$\pi_i = \gamma_1(i)$$

右端的值按照观测 $O = (o_1, o_2, \dots, o_T)$ 和模型 $\lambda^{(n)} = (A^{(n)}, B^{(n)}, \pi^{(n)})$, 以及式中 $\xi_t(i, j)$ 按照“一些概率与期望值”中定义进行计算。

(3) 终止, 得到模型参数 $\lambda^{(n+1)} = (A^{(n+1)}, B^{(n+1)}, \pi^{(n+1)})$

预测算法

前面已经讨论了给定观测序列 $O = (o_1, o_2, \dots, o_T)$ 的概率计算问题以及通过观测序列学习模型参数问题, 还余下HMM三个问题中的给定观测下隐变量的预测问题。

前面, 我们在“一些概率与期望值”中通过前向概率和后向概率, 得到了当前模型参数以及观察序列下, 某一时刻隐变量的取值概率 $\xi_t(i)$ 。

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)}$$

计算每一时刻 t 最可能出现的状态 i , 然后得到最有可能出现的序列。这就是近似算法.

近似算法

$$i_t^* = \arg \max_{1 \leq i \leq N} [\gamma_t(i)], t = 1, 2, \dots, T$$

从而得到状态序列 $I^* = (i_1^*, i_2^*, \dots, i_T^*)$ 。近似算法简单, 但是会预测出一些实际上不发生的状态, 比如转移矩阵等于0的两个相邻状态。尽管如此, 近似算法还是有用的。

维特比算法思想

维特比算法是用动态规划解马尔科夫模型预测问题, 即用动态规划求概率最大路径(最优路径), 这时, 一条路径对应一个状态序列。

根据动态规划原理, 最优路径具有这样的特性: 如果最优路径在时刻 t 通过节点 i_t^* , 那么这一路径从结点 i_t^* 到终点 i_T^* 的部分路径, 对于从 i_t^* 到 i_T^* 的所有可能部分路径来说, 必须是最优的。因为假如不是这样, 那么从 i_t^* 到 i_T^* 就有另外一条更好的部分路径存在, 如果他和从 i_t^* 到 i_T^* 的部分路径连接起来, 就会形成一条比原来的路径更优的路径, 这是矛盾的。

根据这一原理, 我们首先定义两个变量 δ, ψ , 定义在时刻 t 状态为 i 的所有单个路径 (i_1, i_2, \dots, i_t) 中概

率最大值为：

$$\delta_t(i) = \max_{i_1, i_2, \dots, i_{t-1}} P(i_t = i, i_{t-1}, i_{t-2}, \dots, i_1, o_t, \dots, o_1 | \lambda), i = 1, 2, \dots, N$$

由定义可以得到变量 δ 的递推公式：

$$\begin{aligned} \delta_{t+1}(i) &= \max_{i_1, i_2, \dots, i_t} P(i_{t+1} = i, i_{t-1}, i_{t-2}, \dots, i_1, o_{t+1}, \dots, o_1 | \lambda) \\ &= \max_{1 \leq j \leq N} [\delta_t(j) a_{ji}] b_i(o_{t+1}), i = 1, 2, \dots, N; t = 1, 2, \dots, T-1 \end{aligned}$$

定义在时刻t状态i的所有单个路径 $(i_1, i_2, \dots, i_{t-1}, i)$ 中概率最大的路径的第t-1个节点为：

$$\psi_t(i) = \arg \max_{1 \leq j \leq N} [\delta_{t-1}(j) a_{ji}], i = 1, 2, \dots, N$$

维特比算法步骤

输入：模型 $\lambda = (A, B, \pi)$ 和观测数据 $O = (o_1, o_2, \dots, o_T)$ ；

输出：最优化路径 $I^* = (i_1^*, i_2^*, \dots, i_T^*)$ 。

(1). 初始化

$$\delta_1(i) = \pi_i b_i(o_1), i = 1, 2, \dots, N$$

$$\psi_1(i) = 0, i = 1, 2, \dots, N$$

(2). 递推，对 $t=2, 3, \dots, T$

$$\delta_{t+1}(i) = \max_{1 \leq j \leq N} [\delta_t(j) a_{ji}] b_i(o_{t+1}), i = 1, 2, \dots, N$$

$$\psi_t(i) = \arg \max_{1 \leq j \leq N} [\delta_{t-1}(j) a_{ji}], i = 1, 2, \dots, N$$

(3). 终止

$$P^* = \max_{1 \leq j \leq N} \delta_T(i)$$

$$i_T^* = \arg \max_{1 \leq j \leq N} \delta_T(i)$$

(4) 最优路径回溯。对 $t=T-1, T-2, \dots, 1$

$$i_t^* = \psi_{t+1}(i_{t+1}^*)$$

求得最优路径 $I^* = (i_1^*, i_2^*, \dots, i_T^*)$.

DNN-HMM混合系统

DNN-HMM混合系统利用DNN强大的表示学习能力以及HMM的序列化建模能力，他在大词汇连续语音识别任务重的表现远优于传统的(GMM-)HMM.

图模型

图模型，主要分为有向图与无向图；一些特例就是链模型与树模型。具体应用的图模型有隐马尔科夫模型，马尔科夫树，马尔科夫随机场，卡尔曼滤波器。要明白的主要还是数学化后的图模型需要求解的是什么量，以及利用什么方法去求解，先明白大的原理，再看具体的应用，比如链模型与树模型，这些更细的模型又是为了处理什么现实问题而引入的呢？这些都要作为自己心中的问题，从书中去寻找答案。

这一节很重要，这些方法将被用到图像处理和视觉模型，推理中去。现在城市大脑的研究，基于的就是大规模的图推理。要写的话，这一节会很长，有一本一千多页的书，就专讲概率图模型的。有空要把这些方法都掌握了。再把这些方法灵活的运用到其它地方去。更重要的是打通各学科之间的隔阂，因为无向图模型就是来自于统计物理中的Ising模型。



可以看看知乎上这篇文章¹。HMM与Kalman filter的图模型一样，只是前者未知量是离散值，后者未知量是连续值。

有向图模型

有向图模型也称为贝叶斯网络，它可以将具有有向无环图(DAG)形式的联合概率分布的因子分解表示为条件分布的乘积。

$$Pr(x_{1\dots N}) = \prod_{n=1}^N Pr(x_n | x_{pa[n]})$$

其中 $[x_n]_{n=1}^N$ 表示联合分布中的随机变量，同时函数 $pa[n]$ 返回变量 x_n 父节点的索引值。

无向图模型

无向图模型利用势函数 $\phi_c[x_{1\dots N}]$ 的乘积来表示变量 $[x_n]_{n=1}^N$ 的概率分布。结果如下：

$$Pr(x_{1\dots N}) = \frac{1}{Z} \prod_{c=1}^C \phi_c[x_{1\dots N}]$$

配分函数：

$$Z = \sum_{x_1} \sum_{x_2} \dots \sum_{x_N} \prod_{c=1}^C \phi_c[x_{1\dots N}]$$

在我们后面的讨论中，将介绍怎么从Ising模型一步步过度到其他的无向图模型。

链模型

链模型只是图模型的特例。

有向链模型

无向链模型

树模型

主要涉及到隐马尔科夫模型(HMM),条件随机场(CRF),LDA(Latent Dirichlet Allocation)主题模型以及近似推断方法。

问题，方法与应用

我们需要求解的就是最大后验概率MAP，我们求解的方法就是EM算法，Viterbi算法，还有就是需要一些采样方法MCMC。图模型可以应用到推理(手势识别，城市大脑，语音识别，图像处理等各方面，自己要好好整理)。重要的是把我这套方法的原理，而后整理能用到什么问题中去，再推广到其它问题中去。

¹. 概率图模型体系:HMM、MEMM、CRF <https://zhuanlan.zhihu.com/p/33397147> ↵

马尔科夫链蒙特卡洛方法

[参考这篇文章](#)

基本思想

对于一个给定的概率分布 $P(X)$, 若要得到其样本, 通过上述的马尔科夫链的概念, 我们可以构造一个转移矩阵 P 的马尔科夫链, 使得该MC的平稳分布是 $P(X)$, 这样, 无论初始状态是何值, 比如是 x_0 , 那么随着马尔科夫过程的转移, 得到了一系列的状态值, 如 $x_0, x_1, \dots, x_n, x_{n+1}, \dots$, 如果这个马尔科夫过程在第n步已经收敛, 那么分布 $P(X)$ 的样本就是 x_n, x_{n+1}, \dots 。

细致平衡条件

假设一个各态便利的马尔科夫过程, 其转移矩阵是 P , 分布是 $\pi(x)$, 若满足:

$$\pi(i)P_{i,j} = \pi(j)P_{j,i}$$

则 $\pi(x)$ 是马尔科夫链的平稳分布, 上式称为细致平衡条件。

Metropolis采样算法

Metropolis采样算法的基本原理

假设需要从目标概率密度 $p(\theta)$ 中进行采样, 同时, θ 满足 $-\infty < \theta < \infty$. Metropolis采样算法根据马尔科夫链去生成一个序列:

$$\theta^{(1)} \rightarrow \theta^{(2)} \rightarrow \dots \theta^{(t)} \rightarrow$$

其中, $\theta^{(t)}$ 表示的是马尔科夫链在第t时刻的状态。

在Metropolis采样算法过程中, 首先初始化状态值 $\theta^{(1)}$, 然后利用一个已知的分布 $q(\theta|\theta^{(t-1)})$ 生成一个新的候选状态 $\theta^{(*)}$, 随后根据一定的概率选择接受这个新值还是拒绝这个新值, 在Metropolis采样中, 概率为:

$$\alpha = \min\left(1, \frac{p(\theta^{(*)})}{p(\theta^{(t-1)})}\right)$$

这样的过程一直持续到采样过程的收敛, 当收敛以后, 样本 $\theta^{(t)}$ 即为目标分布 $P(\theta)$ 中的样本。

Metropolis采样算法的流程

基于以上的分析，可以总结出如下的Metropolis采样算法的流程
初始化时间t=1

设置u的值，并初始化初始状态 $\theta^{(t)} = u$

重复以下的过程：

令t=t+1 从已知分布 $q(\theta|\theta^{(t-1)})$ 中生成一个候选状态 $\theta^{(*)}$

计算接受的概率： $\alpha = \min\left(1, \frac{p(\theta^{(*)})}{p(\theta^{(t-1)})}\right)$

从均匀分布Uniform(0,1)生成一个随机值a。如果 $a < \alpha$, 则接受新生成的值： $\theta^{(t)} = \theta^{(*)}$; 否

则 $\theta^{(t)} = \theta^{(t-1)}$ 直到t=T.

Metropolis采样算法的解释

要证明Metropolis采样算法的正确性，最重要的是要证明构造的马尔科夫过程满足细致平衡条件，即：

$$\pi(i)P_{i,j} = \pi(j)P_{j,i}$$

对于上面的过程，分布为 $p(\theta)$ ，从状态i转移到状态j的转移概率为：

$$P_{i,j} = \alpha_{i,j}Q_{i,j}$$

其中， $Q_{i,j}$ 为上述已知分布，且是对称的分布 $Q_{i,j} = Q_{j,i}$ 。即：

$q(\theta = \theta^{(t)} | \theta^{(t-1)}) = q(\theta = \theta^{(t-1)} | \theta^{(t)})$ 接下来，我们需要证明Metropolis采样算法中构造的马尔科夫链满足细致平衡条件：

$$\begin{aligned} p(\theta^{(i)} P_{i,j}) &= p(\theta^{(i)} \alpha_{i,j} Q_{i,j}) \\ &= p(\theta^{(i)} \min\left(1, \frac{p(\theta^{(j)})}{p(\theta^{(i)})}\right) Q_{i,j}) \\ &= \min(p(\theta^{(i)} Q_{i,j}), p(\theta^{(j)} Q_{i,j})) \\ &= p(\theta^{(j)} \min\left(\frac{p(\theta^{(i)})}{p(\theta^{(j)})}, 1\right) Q_{j,i}) \\ &= p(\theta^{(i)} \alpha_{i,j} Q_{i,j}) \\ &= p(\theta^{(j)} P_{j,i}) \end{aligned}$$

因此，通过上面方法构造出来的马尔科夫链满足细致平衡条件。

MCMC的使用场景

MCMC——Metropolis-Hastings算法

MCMC——Gibbs Sampling算法

统计推断

贝叶斯方法把参数视为随机变量，这些随机变量具有先验分布，表示这些随机变量在人们心目中的取值。我们可以从贝叶斯的角度，来对正则化进行解释。

L1正则化

当我们认为模型的拟合的残差符合高斯分布，而模型的参数 β_i 服从独立同分布的拉普拉斯分布，也就是：

$$\mathbf{y} | \beta, \lambda, \sigma \sim N(\mathbf{X}^T \beta, \sigma^2 I_{N \times N})$$

$$\beta_i | \lambda, \sigma \sim \prod_{j=1}^p \frac{\lambda}{2\sigma} \exp(-\frac{\lambda}{\sigma} |\beta_j|)$$

很容易证明， $\beta | \mathbf{y}, \lambda, \sigma$ 的负对数后验概率密度为：

$$\frac{1}{2\sigma^2} \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \frac{\lambda}{\sigma} \|\beta\|_1$$

特征工程

特征工程是什么

有这么一句话在业界广泛流传：数据和特征决定了机器学习的上限，而模型和算法只是逼近这个上限而已。那特征工程到底是什么呢？顾名思义，其本质是一项工程活动，目的是最大限度地从原始数据中提取特征以供算法和模型使用。通过总结和归纳，人们认为特征工程包括以下方面：

特征工程

特征选择方式

参考知乎上[这篇文章](#)

特征选择与特征提取是特征工程中的两大核心问题。特征选择是指选择获得相应模型和算法最好性能的特征集，工程上常用的方法如下：

1. 计算每一个特征与相应变量的相关性。一般是算皮尔逊系数和互信息系数。皮尔逊系数只能衡量线性相关性，互信息系数能够很好地度量各种相关性，但是计算相对复杂，很多toolkit里面包含了这个工具（如sklearn的MINI），得到相关性之后就可以排序选择特征了。
2. 构建单个特征的模型，通过模型的准确性为特征排序，借此来选择特征。
3. 通过L1正则项来选择特征，L1正则方法具有稀疏解的特性，因此天然具备特征选择的特性。
4. 训练能够对特征打分的预选模型：RF和LR等都对模型的特征打分，通过打分获得相关性后再训练最终模型。
5. 通过特征组合后回来选择特征：如对用户ID和用户特征的组合来获得较大的特征集再来选择特征，这种做法在推荐系统和广告系统中比较常见，这也是所谓亿级甚至十亿集特征的主要来源，原因是用户数据比较稀疏，特征组合能同时兼顾全局模型和个性化模型。
6. 通过深度学习来进行特征选择/目前这种手段

总体而言，有两种方法：

基于方差的方法(PCA)，基于相关性方法。

1. 相关性：一般用皮尔逊相关系数来定义。
2. 方差法：去掉值变化不大的特征，因为可以把这些特征看成常数值，他们对分类或者回归结果影响不大。

特征选择

1. Filter方法：自变量与目标变量之间的关联
 - i. Pearson相关系数
 - ii. 卡方检测
 - iii. 信息增益，互信息(MIC)
2. Wrapper方法：通过目标函数来决定是否加入一个变量
 - i. 迭代：产生特征子集，评价
 - ii. 完全搜索
 - iii. 启发式搜索
 - iv. 随机搜索(GA, SA)
3. Embedded方法：学习器自身自动选择特征
 - i. 正则化L1, L2
 - ii. 决策树(熵-信息增益，基尼系数)

iii. 深度学习

Filter

Pearson相关系数

Pearson相关系数(取值在[-1,1]之间, 大于0是正相关, 小于0是负相关, 等于0就没有相关性。只对具有线性相关性的变量有效, 不能处理具有非线性关系的两组变量)

互信息(MIC)

$$MIC : I(X, Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log\left(\frac{p(x, y)}{p(x)p(y)}\right)$$

对于线性与非线性关系的数据都实用。

距离相关系数(Distance Correlation)

为了克服Pearson相关系数只对具有线性关系的变量起作用而引入。

样本: $(X_k, Y_k), k = 1, 2, \dots, n$

定义距离矩阵:

$$a_{j,k} = \|\mathbf{X}_j - \mathbf{X}_k\|, j, k = 1, 2, \dots, n$$

$$b_{j,k} = \|\mathbf{Y}_j - \mathbf{Y}_k\|, j, k = 1, 2, \dots, n$$

$\|\cdot\|$ 是欧氏距离。取所有的双中心距离。

$$\mathbf{A}_{j,k} := a_{j,k} - \hat{a}_{j..} - \hat{a}_{..k} + \hat{a}_{...}$$

$$\mathbf{B}_{j,k} := b_{j,k} - \hat{b}_{j..} - \hat{b}_{..k} + \hat{b}_{...}$$

$\hat{a}_{j..}$ 是j-th row的平均, $\hat{a}_{..k}$ 是k-th column的平均, $\hat{a}_{...}$ 是全局平均。

定义The squared sample distance covariance:

$$dCov_n^2(X, Y) := \frac{1}{n^2} \sum_{j=1}^n \sum_{k=1}^n A_{jk} B_{jk}$$

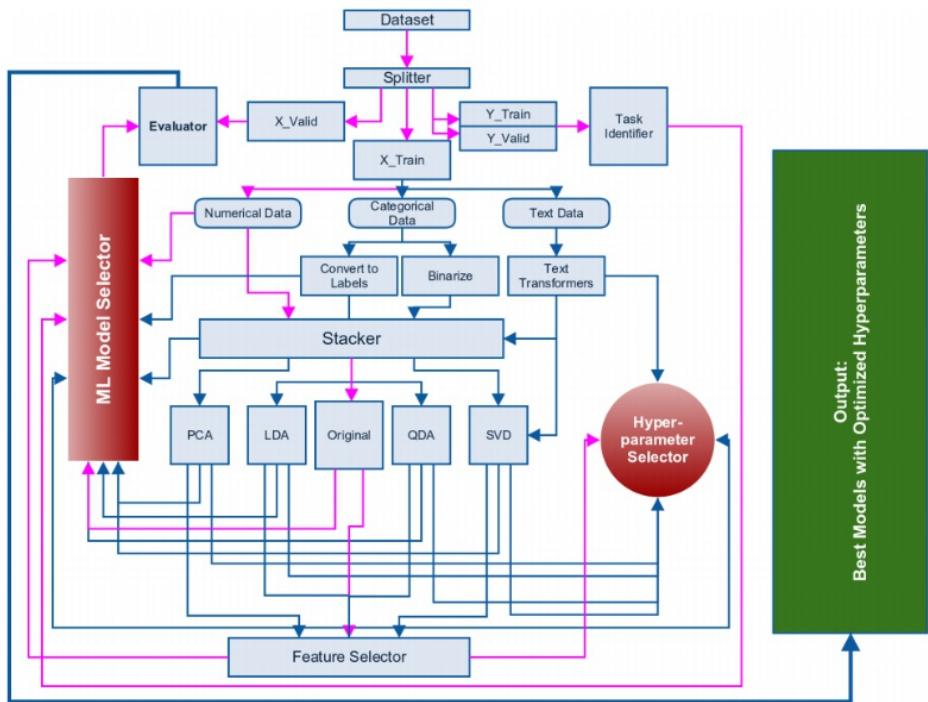
虽然MIC与距离相关系数能处理具有线性与非线性关系的变量之间的相关性, 但是Pearson还是不可替代的, 第一, Pearson系数计算速度快;第二, 相对于其它两种取值在[0,1], Pearson取值[-1,1], 正负表关系的正负, 绝对值表示强度。前提就是两个变量是线性相关的。

特征工程小结

- 特征工程: 利用数据领域的相关知识来创建能够使机器学习算法达到最佳性能的特征的过程。
- 特征构建: 是原始数据中人工的构建新的特征。
- 特征提取: 自动地构建新的特征, 将原始特征转换为一组具有明显物理意义或者统计意义或核的特征。
- 特征选择: 从特征集合中挑选一组最具统计意义的特征子集, 从而达到降维的效果。
 重要性排名: 特征构建>特征提取>特征选择

ML框架¹

参考



特征工程



模型的超参数调节

Model	Parameters to optimize	Good range of values
Linear Regression	<ul style="list-style-type: none">• fit_intercept• normalize	<ul style="list-style-type: none">• True / False• True / False
Ridge	<ul style="list-style-type: none">• alpha• Fit_intercept• Normalize	<ul style="list-style-type: none">• 0.01, 0.1, 1.0, 10, 100• True/False• True/False
k-neighbors	<ul style="list-style-type: none">• N_neighbors• p	<ul style="list-style-type: none">• 2, 4, 8, 16• 2, 3
SVM	<ul style="list-style-type: none">• C• Gamma• class_weight	<ul style="list-style-type: none">• 0.001, 0.01.....10...100...1000• 'Auto', RS*• 'Balanced', None
Logistic Regression	<ul style="list-style-type: none">• Penalty• C	<ul style="list-style-type: none">• L1 or L2• 0.001, 0.01.....10...100
Naive Bayes (all variations)	NONE	NONE
Lasso	<ul style="list-style-type: none">• Alpha• Normalize	<ul style="list-style-type: none">• 0.1, 1.0, 10• True/False
Random Forest	<ul style="list-style-type: none">• N_estimators• Max_depth• Min_samples_split• Min_samples_leaf• Max features	<ul style="list-style-type: none">• 120, 300, 500, 800, 1200• 5, 8, 15, 25, 30, None• 1, 2, 5, 10, 15, 100• 1, 2, 5, 10• Log2, sqrt, None
Xgboost	<ul style="list-style-type: none">• Eta• Gamma• Max_depth• Min_child_weight• Subsample• Colsample_bytree• Lambda• alpha	<ul style="list-style-type: none">• 0.01, 0.015, 0.025, 0.05, 0.1• 0.05-0.1, 0.3, 0.5, 0.7, 0.9, 1.0• 3, 5, 7, 9, 12, 15, 17, 25• 1, 3, 5, 7• 0.6, 0.7, 0.8, 0.9, 1.0• 0.6, 0.7, 0.8, 0.9, 1.0• 0.01-0.1, 1.0, RS*• 0, 0.1, 0.5, 1.0 RS*

NLP 特征工程



特征工程可以参考[这篇文章](#)

<http://blog.kaggle.com/2016/07/21/approaching-almost-any-machine-learning-problem-abhishek-thakur/>

机器学习的本质是数据加算法，算法就是数学函数，方程与优化。

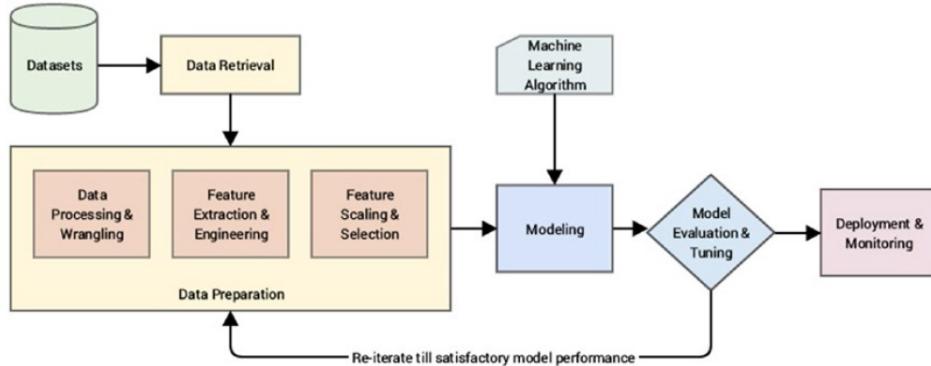


Figure 4-1. Revisiting our standard Machine Learning pipeline

Feature extraction and engineering

标准的机器学习pipeline

特征提取, 标度, 选择的pipeline。

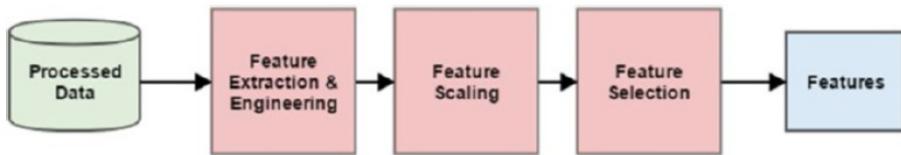


Figure 4-2. A standard pipeline for feature engineering, scaling, and selection

特征提取和特征工程是整个机器学习pipeline中最重要的步骤。

特征工程就是通过特殊领域的知识与特殊的技能, 把数据转化成特征。数据科学家把他们时间的80%花在特征工程上。花在模型搭建与评估上的时间相对的少, 但是也是重要的。深度学习(CNN,RNN,LSTM)区别于机器学习是其拥有机器学习没有的特征自动提取功能。

One-Hot编码在类别数据中工作的非常好, 但是当类别很多时会造成灾难, 此时需要Bin-Counting Scheme., Feature Hashing Scheme.

文本数据处理

Bag of Words Model.对次进行统计

Bag of N-Grams Model, 对N个连续的词进行统计, 比如两个或三个, Bi-Grams, Tri-Grams. 在NLP中, Tri-Grams就已经足够了。

词袋模型只考虑词的统计特性, 但是有些词在所有文本中本身就具有高的频率, 因此就有了TF-IDF model(Term Frequency-Inverse Document Frequency)

$$tfidf(w, D) = tf(w, D) * idf(w, D) = f(w, D) * \log\left(\frac{C}{df(w)}\right)$$

tf-idf(w,D)是词w在文本D中TF-IDF分数。tf(w,D)表示词w在文档D中的频率, idf(w,D)是词w的逆文档频率, C是该语料库中的文本数。TF-IDF模型可以用来做为文本关键词提取。以及文本相似性计算(一般用余弦相似性), 因此可以用来做聚类, 适用于无监督学习。

LDA(Latent Dirichlet Allocation)作用在TF-IDF矩阵上, 分解成document-topic matrix与topic-term matrix。

```
In [11]: from sklearn.decomposition import LatentDirichletAllocation  
....  
....: lda = LatentDirichletAllocation(n_topics=2, max_iter=100, random_state=42)  
....: dt_matrix = lda.fit_transform(tv_matrix)  
....: features = pd.DataFrame(dt_matrix, columns=['T1', 'T2'])  
....: features  
Out[11]:  
      T1      T2  
0  0.190615  0.809385  
1  0.176860  0.823140  
2  0.846148  0.153852  
3  0.815229  0.184771  
4  0.180563  0.819437  
5  0.839140  0.160860
```

利用LDA把tf-idf矩阵分解成两个主题T1,T2,可以看出0, 1, 4成一类, 2, 3, 5成一类。

图像数据

```
Canny edge detection算法;from skimage.feature import canny。  
Object Detection:HOG(Histogram of Oriented gradients)  
from skimage.feature import hog
```

Feature scaling

Feature selection

准确率与召回率

True Positive(TP):实际是正例并且我们分类成正例的样本数。
False Positive(FP):实际是负例但是我们分类成正例的样本数。
True Negative(TN):实际是负例并且我们分类成也是负例的样本数。
False Negative(FN):实际是正例但是我们分类成也是负例的样本数。

Accuracy

$$Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$$

Precision

只针对于正样本或者误分为正样本的样本，在不均衡样本中比较有用，比如人群中癌症阳性的比例很小，因此如果预测人群没有患癌则获得的accuracy很高，但是precision是0.。

$$Precision = \frac{TP}{TP+FP}$$

Recall

$$Recall = \frac{TP}{TP+FN}$$

F1 Score

$$F1Score = \frac{2*Precision*Recall}{Precision+Recall}$$

超参数finetune

两种方法：Grid 搜索与随机搜索。

Factorization Machines(FM)

这一节讨论点击率预测原发，常用的CTR预估算法有FM, FFM, DeepFM。

要明白FM与SVD之间的联系。

原理就是，每个特征 x 可以用隐变量来表示，一般隐变量维数为30-40，也就是说，每一个特征可以在一个30-40维的低维空间来表示。因此，可以与Krylov子空间，SVD联系起来。都是需求超大特征空间的低维表示或者低秩分解。

FM

一般的线性模型为：

$$y = w_0 + \sum_{i=1}^n w_i x_i$$
 一般模型中，各个特征是独立考虑的，没有考虑特征之间的相互关系。如果

考虑特征 x_i, x_j 之间的相互关系，模型修改如下：

$$y = w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j$$

如果系统的特征比较多的话，计算复杂度会大大提升。为了降低时间复杂度，我们引入了辅助向量 latent vector

$$\mathbf{V}_i = [v_{i1}, v_{i2}, \dots, v_{ik}]^T$$

$$w_i = \sum_{f=1}^k v_{if}$$

$$w_{i_1, \dots, i_{o^*}} = \sum_{f=1}^{k_{o^*}} \prod_{j=1}^{o^*} v_{i_j, f}$$

辅助变量是描述变量之间的相关性。模型修改如下：

$$y = w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n (V_i, V_j) x_i x_j$$

以上就是FM模型。k是超参数，一般娶30或者40。时间复杂度是 $O(kn^2)$ ，可通过如下方式化简。

$$\begin{aligned} & \sum_{i=1}^n \sum_{j=i+1}^n (V_i, V_j) x_i x_j \\ &= \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n (V_i, V_j) x_i x_j - \frac{1}{2} \sum_{i=1}^n (V_i, V_i) x_i x_i \\ &= \frac{1}{2} \sum_{f=1}^k \left(\sum_{i=1}^n v_{if} x_i \right)^2 - \sum_{i=1}^n v_{if}^2 x_i^2 \end{aligned}$$

通过对每个特征引入latent vector \mathbf{V}_i ，并对公式进行化简，可以把时间复杂度降为 $O(kn)$ 。

Field-aware Factorization Machines(FFM)

模型评估

这一章主要介绍一些模型评估的方法，比如Cross-Validation, ROC, AUC
模型的性能¹：

1. 错误率/精度(accuracy)
2. 准确率(查准率, precision)/召回率(查全率, recall)
3. P-R曲线, F1度量
4. ROC曲线/AUC(最常用) 参考这篇文章：[机器学习模型性能评估方法笔记](#)

错误率/精度(accuracy)

假设我们拥有m个样本个体，那么该样本的错误率为：

$$e = \frac{1}{m} \sum_{i=1}^m I(f(x_i) \neq y_i)$$

或者：

$$e = \int_{x \sim D} I(f(x) \neq y) p(x) dx$$

准确率/召回率(查全率)

准确率=预测为真且实际也为真的个体个数 / 预测为真的个体个数

召回率 = 预测为真且实际也为真的个体个数 / 实际为真的个体个数

我们将准确率记为P, 召回率记为R, 定义, TP(true positive), FP(false positive), FN(false negative), TN(true negative)。则有：

$$P = \frac{TP}{TP+FP}$$

$$R = \frac{TP}{TP+FN}$$

ROC曲线/AUC

ROC曲线则是以假正例率FPR为横轴, 真正例率TPR为纵轴。其中

$$FPR = \frac{FP}{FP+TN}$$

$$TPR = \frac{TP}{TP+FN}$$

我们可以看到真正例率与召回率是一样的, 那么ROC曲线图如下图所示:

¹. [机器学习模型性能评估方法笔记](#) ↵

正则化方法原理

Ridge回归, Shrink与SVD

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

等价于：

$$\begin{aligned} \hat{\beta}^{ridge} &= \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2 \\ &\text{subject to : } \sum_{j=1}^p \beta_j^2 \leq t. \end{aligned}$$

上面的问题也等价于假定, y_i, β_i 服从如下分布:

$$y_i \in N(\beta_0 + x_i^T \beta, \sigma^2)$$

$$\beta_i \in N(0, \tau^2)$$

其中: $\lambda = \sigma^2 / \tau^2$

因此RRS(Root sum square)可以写成如下形式:

$$RRS(\lambda) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\beta^T\beta$$

通过对 β 求导并令其为0, 可以得到:

$$\hat{\beta}^{ridge} = (\mathbf{X}^T \mathbf{X} + \lambda I)^{-1} \mathbf{X}^T \mathbf{y}$$

为了建立起L2与SVD之间的联系, 我们对 \mathbf{X} 进行SVD分解:

$$\mathbf{X} = \mathbf{U} \mathbf{D} \mathbf{V}^T$$

我们重写没有正则化的最小二乘拟合:

$$\mathbf{X}\beta^{ls} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} = \mathbf{U} \mathbf{U}^T \mathbf{y}$$

对于L2正则化的最小二乘法:

$$\begin{aligned} \mathbf{X}\beta^{ls} &= (\mathbf{X}^T \mathbf{X} + \lambda I)^{-1} \mathbf{X}^T \mathbf{y} = \mathbf{U} \mathbf{D} (\mathbf{D}^2 + \lambda \mathbf{I})^{-1} \mathbf{D} \mathbf{U}^T \mathbf{y} \\ &= \sum_{j=1}^p \mathbf{u}_j \frac{d_j^2}{d_j^2 + \lambda} \mathbf{u}_j^T \mathbf{y} \end{aligned}$$

从上面的分式 $\frac{d_j^2}{d_j^2 + \lambda}$ 可以知道, 对于 $d_j \ll \lambda$, 则相应的方向会收缩到0。可以认为L2就是对数据进行了SVD分解后, 只保留了 $d_j > \lambda$ 的分量。

Lasso(L1)回归

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

等价于：

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2$$

$$subject to: \sum_{j=1}^p |\beta_j| \leq t.$$

L1更容易产生稀疏性,因为椭圆与菱形更易在菱形顶点相交,而与圆形不容易在坐标轴上相切。



L1稀疏性的数学解析

一个数学上更严格 的解析。。上面的问题也等价于假定, y_i, β_i 服从如下分布:

$$y_i \in N(\beta_0 + x_i^T \beta, \sigma^2)$$

$$\beta_i \in (1/2\tau) \exp(-|\beta|/\tau)$$

其中: $\tau = 1/\lambda$

因此 β_i 更容易分布在0的附件, 也就是能级排斥(量子混沌里面的概念, L1是可积系统, L2是GOE)

最小二乘一般通过Cholesky分解($p^3 + Np^2/2$)或者QR分解(NP^2)来实现, 前者一般更快, 但是没有后者稳定。

机器学习任务

机器学习score函数设置：

1. 回归, 分类, 决策树, 深度神经网络, 图模型, 概率统计, 最优化方法
2. 分类, 聚类, 特征选择, 降维等数据挖掘技术
3. 机器学习, 概率统计, 最优化
4. GBDT, LR, LTR, 特征提取
5. 推荐系统基本算法: LR, GBDT, SVD/SVD++, FM/FFM具体实用场景与变形
6. 对分类, 回归, 聚类, 标注等统计机器学习问题有深入研究, 熟悉常用模型: LR, KNN, Naive bayes, rf, GBDT, SVM, PCA, SVD, kmeans, kmodes等
7. 精通主流机器学习算法, 对贝叶斯, 随机森林, SVM, 神经网络, 聚类, PCA等有深入研究
8. 熟悉数据分析思路, 熟悉经典的数据挖掘, 机器学习算法, 如: LR, 决策树, BP神经网络, SVM等浅层学习, 熟悉集成算法, 诸如bagging, boosting, 了解深度学习CNN, RNN, 熟悉使用python, 了解pandas, sklearn, xgboost
9. 熟悉常用的最优化算法设计与实现, 对于非凸优化问题有深入的理解(并行模拟退火, 蒙特卡洛等优化方法)
10. 机器学习算法: 贝叶斯, 聚类, 逻辑回归, SVM, GBDT, RF

总结:

1. 分类:
 - i. SVM,
 - ii. LR
2. 回归:
3. 集成学习:
 - i. bagging
 - ii. Boosting
4. 工具
 - i. sklearn, xgboost, tensorflow

分类总结:

机器学习

1. Regression:
 - i. Ordinary Least Squares Regression(OLSR)
 - ii. Linear Regression
 - iii. Logistic Regression
 - iv. Stepwise Regression

- v. Multivariate Adaptive Regression Splines
- vi. Locally Estimated Scatterplot Smoothing
- 2. Regularization Algorithms
 - i. Ridge Regression
 - ii. Least Absolute Shrinkage and Selection Operator(LASSO)
 - iii. Least-Angle Regression(LARS)
- 3. Ensemble Algorithms
 - i. Boosting
 - ii. Booststrapped Aggregation(Bagging)
 - iii. Adaboost
 - iv. Stacked Generalization(Blending)
 - v. Gradient Boosting Machines(GBM)
 - vi. Gradient Boosted Regression Trees(GBRT)
 - vii. Random Forest
- 4. Decision Tree Algorithms
 - i. Classification and Regression Tree(CART)
 - ii. Iterative Dichotomiser3(ID3)
 - iii. C4.5 and C5.0
 - iv. Chi-Squared Automatic Interaction Detection(CHAID)
 - v. Decision Stump
 - vi. M5
 - vii. Conditional Decision Trees
- 5. Dimensionality Reduction Algorithms
 - i. Principle Component Analysis(PCA)
 - ii. Principle Component Regression(PCR)
 - iii. Sammon Mapping
 - iv. Multidimensional Scaling(MDS)
 - v. Projection Pursuit
 - vi. Linear Discriminant Analysis(LDA)
 - vii. Mixture Discriminant Analysis(MDA)
 - viii. Quadratic Discriminant Analysis(QDA)
 - ix. Flexible Discriminant Analysis(FDA)
- 6. Bayesian Algorithms
 - i. Naive Bayes
 - ii. Gaussian Naive Bayes
 - iii. Multinomial Naive Bayes
 - iv. Averaged One-Dependence Estimators(AODE)
 - v. Bayesian Belief Network(BBN)
 - vi. Bayesian Network(BN)
- 7. Clustering Algorithms
 - i. K-Means
 - ii. K-Medians

- iii. Expectation Maximisation(EM)
 - iv. Hierarchical
8. Instance-Based Algorithms
- i. K-Nearest Neighbor(KNN)
 - ii. Learning Vector Quantization(LVQ)
 - iii. Self-Organizing Map(SOM)
 - iv. Locally Weighted Learning(LWL)
9. Graphical Models
- i. Bayesian Network
 - ii. Markov random field
 - iii. Chain Graphs
 - iv. Ancestral Graph
10. Association Rule Learning Algorithms
- i. Apriori Algorithm
 - ii. Eclat Algorithm
 - iii. FP-growth

深度学习

- 1. Deep Learning
 - i. Deep Boltzmann Machine(DBM)
 - ii. Deep Belief Networks(DBN)
 - iii. Convolutional Neural Network(CNN)
 - iv. Stacked Auto-Encoders
- 2. Artificial Neural Network
 - i. Perception
 - ii. Back-Propagation
 - iii. Radial Basis Function Network(RBFN)

机器学习问题

1. 过拟合与欠拟合, 交差验证的目的, 超参数搜索方法, EarlyStopping
2. L1正则和L2正则的做法, 正则化背后的思想, BatchNorm, Covariance Shift, L1正则产生稀疏解的原理
3. 逻辑回归为何是线性模型, LR如何解决低维不可分, 从图模型角度看LR,
4. 和朴素贝叶斯和无监督
5. 几种参数估计方法MLE, MAP, 贝叶斯的联系与区别
6. 简单说下SVM的支持向量, KKT, 何为对偶, 核的通俗理解
7. GBDT,随机森林能否并行, 问问bagging, boosting
8. 生成模型, 判别模型举个例子
9. 聚类方法的掌握, 问问kmeans的EM推导思路, 谱聚类和Graph-cut的理解
10. 梯度下降类方法和牛顿类方法的区别, 随便问问Adam, L-BFGS的思路
11. 半监督的思想, 问问一些特定半监督算法是如何利用无标签数据的, 从MAP角度看半监督
12. 常见的分类模型的评价指标(顺便问问交叉熵, ROC如何绘制, AUC的物理含义, 类别不均匀样本)

神经网络

1. CNN中卷积操作和卷积核作用, maxpooling作用
2. 卷积层与全连接层的联系
3. 梯度爆炸与消失的概念(顺便问问神经网络权重初始化的方法, 为何能减缓梯度爆炸与消失, CNN中有哪些解决方法, LSTM如何解决的, 如何梯度裁剪, dropout如何用在RNN系列网络中, dropout如何防止过拟合)
4. 为何卷积可以用在图像, 语音, 语句上, 顺便问问channel在不同类型数据源中的含义

自然语言处理, 推荐系统

1. CRF跟逻辑回归, 最大熵模型的关系
2. CRF的优化方法, CRF和MRF的联系, HMM与CRF的关系(顺便问问朴素贝叶斯和HMM的联系, LSTM+CRF用于序列标注的原理, CRF的点函数和边函数, CRF的经验分布)
3. wordEmbedding的几种常用方法和原理(language model, perplexity评价指标, word2vec跟Glove的异同)
4. Topic model说一说, 为何CNN能用在文本分类, syntactic 和semantic问题举例,
5. 常见的sentence embedding方法, 注意力机制(注意力机制的几种不同情形, 为何引入, seq2seq原理)
6. 序列标注的评价指标, 语义消歧的做法, 常见的跟word有关的特征
7. factorization machine, 常见矩阵分解模型
8. 如何把分类模型用于商品推荐(包括数据集划分, 模型验证等)
9. 序列学习, wide & deep model(顺便问问为何wide和deep)

第六章 深度学习

深度学习正在深刻的影响一些传统的领域，比如计算机视觉，语音识别与自然语言处理，本章的最终目的就是介绍深度学习在这些领域的成就，当然，很多方式是深度学习与传统方法的结合。

本章另外一个任务就是对计算机视觉，语音识别与自然语言处理这三个领域的传统与现代方法的梳理，现代方法主要是指深度学习，传统方法主要是指机器学习。其中，概率图模型以前在这三大领域发挥了重大的作用，是处理这三大领域问题的基本方法。因此，自己要加强对概率图模型的理解与应用。

自己知道方法与原理后，要能快速的实现这些模型，这才是自己欠缺的一个关键能力，必须加强。这一章主要讨论深度学习模型。主要包含CNN, RNN。

CNN侧重于空间，主要用在图像分类模型，比如LeNet-5,AlexNet,VGG,GoogLeNet,Residual Net,Dense Net,Two path Net。

RNN侧重于时间序列，主要用于语音识别，自然语言处理，包含的网络有传统的RNN, LSTM以及GRU。

DNN(MLP)主要指有多个隐藏层的神经网络，也称为多层感知机(Multi-layer perceptron,MLP)。在介绍完CNN与RNN之后，就是介绍这两种网络的实际用途了，比如RNN在语音识别方面的应用。GAN是一大块，但是不知道能用在什么地方，有空也必须添加进去。

Deep learning 与传统的神经网络的区别

传统神经网络采用BP算法，而深度学习采用逐层训练机制，这解决了以前至多只能训练六七层的问题。

BP算法存在的问题：

1. 梯度越来越稀疏，从顶层往下，误差校正信号越来越小；
2. 收敛到局部最小值：尤其从远离最优区域开始的时候。
3. 只能使用有标签数据

Deep learning训练过程：

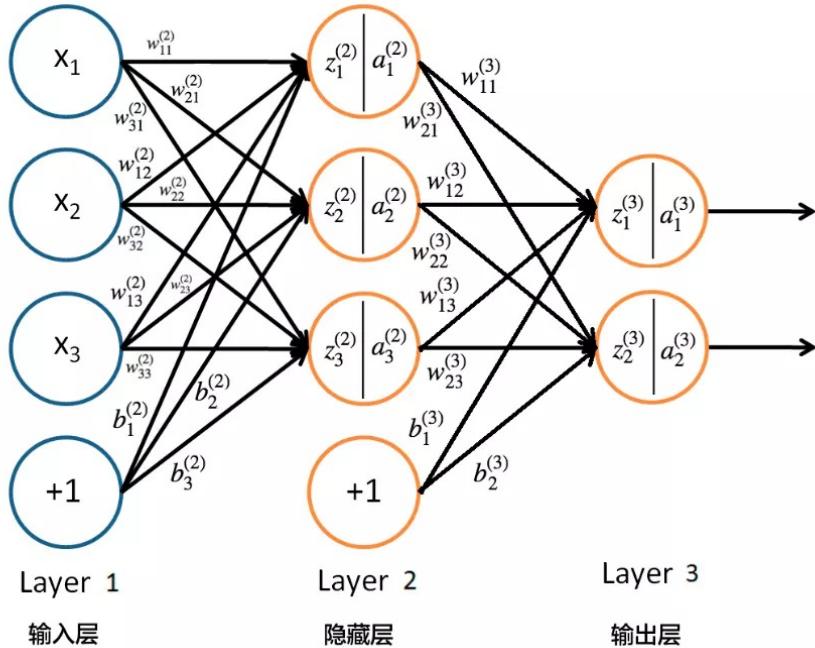
- 1.首先逐层构建单层网络，这样每次都是训练一层神经网络。
- 2.当所有层训练完之后，Hilton使用wake-sleep进行调优。

自己动手写神经网络

这里需要实现NN, CNN,RNN, 区别在于是否加卷积。对于NN, 要保证层数不说限制, 每层的神经元不受限制, 只处理一维输入,输出也就一维,也就是分类问题;CNN处理二维输入(图像), 设计卷积和卷积核, 实现的例子就是LeNet-5。RNN就是实现最简单的RNN。三者通用的东西是什么?可以提出来, 我们需要用到哪些模块?这个心里是要清楚的。

DNN

首先讨论普通的DNN



n_l 是层神经元数目;

$W^{(l)} \in R^{n_l \times n_{l-1}}$ 是第l-1层到第l层的权重矩阵。

$W_{ij}^{(l)}$ 表示第l-1层的j神经元到l层的i神经元之间的权重。

$f()$ 是激活函数。

$\mathbf{b}^{(l)} = [b_1^{(l)}, b_2^{(l)}, \dots, b_{n_l}^{(l)}] \in R^{n_l}$ 表示第l-1层到第l层的偏置。

$\mathbf{z}^{(l)} = [z_1^{(l)}, z_2^{(l)}, \dots, z_{n_l}^{(l)}] \in R^{n_l}$ 表示第l层神经元的状态。

$\mathbf{a}^{(l)} = [a_1^{(l)}, a_2^{(l)}, \dots, a_{n_l}^{(l)}] \in R^{n_l}$ 表示第l层神经元的激活值。

信息前向传播

因此我们根据下面的公式计算每层的神经元状态与输出值，计算从第2层到最后一层L。

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

对于第L层，最终输出 $\mathbf{a}^{(L)}$ 。信息传播如下：

$$\mathbf{x} = \mathbf{a}^{(1)} \rightarrow \mathbf{z}^{(2)} \rightarrow \mathbf{a}^{(2)} \rightarrow \dots \rightarrow \mathbf{a}^{(L-1)} \rightarrow \mathbf{z}^{(L)} \rightarrow \mathbf{a}^{(L)} = \mathbf{y}$$

误差反向传播

训练数据为 $(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), \dots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})$ ，即N个数据，假设输出是 n_L 维的，也就是：

$$\mathbf{y}^{(i)} = (y_1^{(i)}, y_2^{(i)}, \dots, y_{n_L}^{(i)})^T \in R^{n_L \times 1}$$

对于第i个数据，假设误差是均方误差， $\mathbf{o}^{(i)}$ 是输出。

$$E_i = \frac{1}{2}(\mathbf{y}^{(i)} - \mathbf{o}^{(i)})^2 = \frac{1}{2} \sum_{k=1}^{n_L} (y_k^{(i)} - o_k^{(i)})^2$$

因此对总的N个数据。总的误差表示如下：

$$E = \frac{1}{N} \sum_{i=1}^N E_i$$

对输出层权重参数更新

$$\frac{\partial E}{\partial w_{ij}^{(n_L)}} = \frac{\partial E}{\partial z_i^{(n_L)}} \frac{\partial z_i^{(n_L)}}{\partial w_{ij}^{(n_L)}} = \delta_i^{(n_L)} a_j^{(n_L-1)}$$

其中：

$$\delta_i^{(n_L)} = \frac{\partial E}{\partial z_i^{(n_L)}} = -(y_i - a_i^{(n_L)}) f'(z_i^{(n_L)})$$

因此：

$$\delta^{(n_L)} = -(\mathbf{y} - \mathbf{a}^{(n_L)}) * \mathbf{f}'(\mathbf{z}^{(n_L)})$$

其中*表示向量对应元素相乘。

对隐含层权重参数的更新

对于隐藏层有：

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} a_j^{(l-1)}$$

$$\delta_i^{(l)} = \frac{\partial E}{\partial z_i^{(l)}} = \sum_{j=1}^{n_{l+1}} \frac{\partial E}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}}$$

$$\frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = w_{ji}^{(l+1)} f'(z_i^{(l)})$$

因此有：

$$\delta_i^{(l)} = \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} w_{ji}^{(l+1)} f'(z_i^{(l)}) = \left(\sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} w_{ji}^{(l+1)} \right) f'(z_i^{(l)})$$

因此写成矢量想成形式：

$$\delta^{(l)} = -((\mathbf{W}^{(l+1)})^T \delta^{(l+1)}) * \mathbf{f}'(\mathbf{z}^{(l)})$$

因此：

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \left(\sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} w_{ji}^{(l+1)} \right) f'(z_i^{(l)}) a_j^{(l-1)}$$

对输出层与隐含偏置求导数

$$\frac{\partial E}{\partial b_i^{(l)}} = \frac{\partial E}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} = \delta_i^{(l)}$$

BP四个核心公式

整理起来，BP算法有如下四个核心公式：

$$\begin{aligned}\delta_i^{(L)} &= -(y_i - a_i^{(L)}) f'(z_i^{(L)}) \\ \delta_i^{(l)} &= \left(\sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} w_{ji}^{(l+1)} \right) f'(z_i^{(l)}) \\ \frac{\partial E}{\partial b_i^{(l)}} &= \delta_i^{(l)} \\ \frac{\partial E}{\partial w_{ij}^{(l)}} &= \delta_i^{(l)} a_j^{(l-1)}\end{aligned}$$

两个辅助公式

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

参数更新

$$\begin{aligned}W^{(l)} &= W^{(l)} - \frac{u}{N} \sum_{i=1}^N \frac{\partial E_i}{\partial W^{(l)}} \\ \mathbf{b}^{(l)} &= \mathbf{b}^{(l)} - \frac{u}{N} \sum_{i=1}^N \frac{\partial E_i}{\partial b^{(l)}}\end{aligned}$$

代码实现

如下是一个输入两个变量，只含有4个神经元的隐含层以及一个输出变量的网络(2,4,1)。

```
import numpy as np

class BP_network(object):
    def __init__(self):
        self.w2 = np.random.random((4, 2))
        self.w3 = np.random.random((1, 4))
        self.b2 = np.random.random((4, 1))
        self.b3 = np.random.random((1, 1))
        self.a1 = np.zeros((2, 1))
        self.z2 = np.zeros((4, 1))
        self.a2 = np.zeros((4, 1))
        self.z3 = np.zeros((1, 1))
        self.a3 = np.zeros((1, 1))
        self.delta3 = np.zeros((1, 1))
        self.delta2 = np.zeros((4, 1))
        self.batch_size = 100
        self.leak_rate = 0.01
        self.learn_rate = 0.1

    def grad(self, x):
        result = np.zeros(x.shape)
        for n in range(x.shape[0]):
            if x[n] > 0:
                result[n] = 1
            else:
                result[n] = self.leak_rate
        return result

    def activate_function(self, x):
        result = np.zeros(x.shape)
        for n in range(x.shape[0]):
            if x[n] > 0:
                result[n] = x[n]
            else:
                result[n] = self.leak_rate*x[n]
        del x
        return result

    def forward_prop(self, input_data):
        self.a1 = input_data
        self.z2 = np.dot(self.w2, self.a1) + self.b2
        self.a2 = self.activate_function(self.z2)

        self.z3 = np.dot(self.w3, self.a2) + self.b3
        self.a3 = self.activate_function(self.z3)
        return self.a3

    def back_prop(self, out_put, y_target):
        self.delta3 = -(y_target - out_put)*self.grad(self.z3)
```

```

        self.delta2 = np.multiply(np.dot(self.w3.T, self.delta3), self.grad(self.z2))
        self.w3 -= self.learn_rate*np.dot(self.delta3, self.a2.T)
        self.b3 -= self.learn_rate*self.delta3
        self.w2 -= self.learn_rate*np.dot(self.delta2, self.a1.T)
        self.b2 -= self.learn_rate*self.delta2

    def train(self, max_epoch, input_data, output_data):
        data_size = len(input_data)
        for epoch_n in range(max_epoch):
            n = np.random.randint(0, input_data.shape[0])
            x = input_data[n, :]
            x = np.reshape(x, (x.shape[0], 1))
            y = output_data[epoch_n%data_size, :]
            output = self.forward_prop(x)
            error = (output[0] - y[0])*(output[0] - y[0])
            print("Iter ", epoch_n, error, n)
            self.back_prop(output, y)

    if __name__ == '__main__':
        bp = BP_network()
        # z = x + 2y
        input_data = np.array([[1, 1], [1, 0], [0, 1], [1, 0.5], [1.5, 0.5]])
        output_data = np.array([[3], [1], [2], [2], [2.5]])
        bp.train(100, input_data, output_data)
        result = bp.forward_prop(np.reshape([1, 1], (2, 1)))
        print("predict is ", result)

```

CNN

现在我们讨论卷积神经网络的误差怎么反向传播，以及参数的梯度怎么计算。主要涉及到的概念就是卷积计算，pooling，步长，边界补齐。

下图就是一个典型的CNN¹。



卷积计算

假设 I 就是输入的图像, $K \in R^{k_1 \times k_2}$ 的卷积核, 则卷积如下计算:

$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i-m, j-n)K(m, n)$$

$$= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n)K(-m, -n)$$

卷积和Cross-correlation是一样的, 当 $K(-m, -n) == K(m, n)$;

首先通过数学来描述每一层网络:

对于图像, 我们输入是一个高H, 长W和通道C=3的张量, 比如图像

$I \in R^{H \times W \times C}$; 对于D个Filters, 我们有 $K \in R^{k_1 \times k_2 \times C \times D}$, 以及偏置 $b \in R^D$ 。

因此通过卷积操作之后的输出是:

$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \sum_{c=1}^C K_{m,n,c} \cdot I_{i+m, j+n, c} + b$$

对于灰度图C=1:

$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} K_{m,n} \cdot I_{i+m, j+n} + b$$

记号

为了方便我们的计算, 我们约定使用如下计算:

1. l 表示低 l 层, $l=1$ 是第一层, $l=L$ 是最最后一层
2. 输入的维度是 $H \times W$, 用 i, j 表示其迭代子
3. 卷积核 w 的维度是 $k_1 \times k_2$, m, n 是他们的迭代子
4. $w_{m,n}^l$ 是连接第 l 层神经元与第 $l-1$ 层之间的权值矩阵, 也就是卷积核。
5. b^l 是第 l 层的偏置
6. l 层的输入 $x_{i,j}^l$ 与第 $l-1$ 层的输出 $O_{i,j}^{l-1}$ 之间关系是:

$$x_{i,j}^l = \sum_m \sum_n w_{m,n}^l o_{i+m, j+n}^{l-1} + b^l$$

7. $O_{i,j}^{l-1}$ 是 l 层的输出:

$$O_{i,j}^{l-1} = f(x_{i,j}^l)$$

8. $f(\cdot)$ 是激活函数

l 层的输入输出

$$x_{i,j}^l = \text{rot}_{180^\circ} \{w_{m,n}^l\} * o_{i,j}^{l-1} + b_{i,j}^l$$

$$x_{i,j}^l = \sum_m \sum_n w_{m,n}^l o_{i+m, j+n}^{l-1} + b_{i,j}^l$$

$$o_{i,j}^l = f(x_{i,j}^l)$$

误差

神经网络的输出值是 y_p , 相应的目标值是 t_p 。因此误差是:

$$E = \frac{1}{2} \sum_p (t_p - y_p)^2$$

误差反向

我们所有要计算的参数就是每层卷积核与偏置。

我们需要通过改变卷积核中的一个矩阵元 $w_{m'n'}$ 来观察最终的误差怎么改变, 再反过来调节卷积核个的元素大小, 最终使得网络输出值与相应的目标值之间的误差极小。

总的来说, CNN与普通的DNN的求法思想是一样的, CNN特别之处不是对整张图求和。而只是对卷积范围内的区域求和。



$H \times W$ 的图像经过 $k_1 \times k_2$ 的卷积核作用之后, 得到的图像大小是 $(H - k_1 + 1) \times (W - k_2 + 1)$ 。因此相应的梯度可以由链式法则表示:

$$\begin{aligned} \frac{\partial E}{\partial w_{m',n'}} &= \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x_{i,j}^l} \frac{\partial x_{i,j}^l}{\partial w_{m',n'}} \\ &= \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta_{i,j}^l \frac{\partial x_{i,j}^l}{\partial w_{m',n'}} \end{aligned}$$

因为:

$$x_{i,j}^l = \sum_m \sum_n w_{m,n}^l o_{i+m,j+n}^{l-1} + b_{i,j}^l \text{ 代入上式得到:}$$

$$\frac{\partial x_{i,j}^l}{\partial w_{m',n'}} = \frac{\partial}{\partial w_{m',n'}} (\sum_m \sum_n w_{m,n}^l o_{i+m,j+n}^{l-1} + b^l)$$

得到:

$$\frac{\partial x_{i,j}^l}{\partial w_{m',n'}} = \frac{\partial}{\partial w_{m',n'}} (w_{0,0} b_{0,0} o_{i+0,j+0}^{l-1} + \dots + w_{m',n'}^l o_{i+m',j+n'}^{l-1} + \dots + b^l)$$

$$= \frac{\partial}{\partial w_{m',n'}}^l (w_{m',n'}^l o_{i+m',j+n'}^{l-1}) = o_{i+m',j+n'}^{l-1}$$

最终得到误差对卷积核的梯度是：

$$\begin{aligned} \frac{\partial E}{\partial w_{m',n'}} &= \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta_{i,j}^l o_{i+m',j+n'}^{l-1} \\ &= \text{rot}_{180^\circ} \{ \delta_{i,j}^l \} * o_{m',n'}^{l-1} \end{aligned}$$

误差对每输入的导数

$$\delta_{i,j}^l = \frac{\partial E}{\partial x_{i,j}^l}$$

可以由链式法则来求，并建立起 $\delta_{i,j}^l$ 与 $\delta_{i,j}^{l+1}$ 之间的关系：

$$\frac{\partial E}{\partial x_{i',j'}^l} = \sum_{i,j \in Q} \frac{\partial E}{\partial x_Q^{l+1}} \frac{\partial x_Q^{l+1}}{\partial x_{i',j'}^l} = \sum_{i,j \in Q} \delta_Q^{l+1} \frac{\partial x_Q^{l+1}}{\partial x_{i',j'}^l}$$

只有有限区域Q中的 $\delta_{i,j}^{l+1}$ 对 $\delta_{i,j}^l$ 有影响，Q的大小就是卷积核的大小。

$$\begin{aligned} \frac{\partial E}{\partial x_{i',j'}^l} &= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \frac{\partial E}{\partial x_{i'-m,j'-n}^{l+1}} \frac{\partial x_{i'-m,j'-n}^{l+1}}{\partial x_{i',j'}^l} \\ &= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \delta_{i'-m,j'-n}^{l+1} \frac{\partial x_{i'-m,j'-n}^{l+1}}{\partial x_{i',j'}^l} \end{aligned}$$

相邻两层输入之间的导数

$$\begin{aligned} \frac{\partial x_{i'-m,j'-n}^{l+1}}{\partial x_{i',j'}^l} &= \frac{\partial}{\partial x_{i',j'}^l} \left(\sum_{m'} \sum_{n'} w_{m',n'}^{l+1} o_{i'-m+m',j'-n+n'}^l + b^{l+1} \right) \\ &= \frac{\partial}{\partial x_{i',j'}^l} \left(\sum_{m'} \sum_{n'} w_{m',n'}^{l+1} f(x_{i'-m+m',j'-n+n'}^l) + b^{l+1} \right) \end{aligned}$$

得到：

$$\begin{aligned} \frac{\partial x_{i'-m,j'-n}^{l+1}}{\partial x_{i',j'}^l} &= \frac{\partial}{\partial x_{i',j'}^l} \left(w_{m',n'}^{l+1} f(x_{0-m+m',0-n+n'}^l) + \dots + w_{m',n'}^{l+1} f(x_{i',j'}^l) + \dots + b^{l+1} \right) \\ &= \frac{\partial}{\partial x_{i',j'}^l} (w_{m',n'}^{l+1} f(x_{i',j'}^l)) = w_{m',n'}^{l+1} \frac{\partial}{\partial x_{i',j'}^l} (f(x_{i',j'}^l)) \\ &= w_{m',n'}^{l+1} f'(x_{i',j'}^l) \end{aligned}$$

最终得到误差对输入的导数：

$$\begin{aligned} \frac{\partial E}{\partial x_{i',j'}^l} &= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \delta_{i'-m,j'-n}^{l+1} w_{m',n'}^{l+1} f'(x_{i',j'}^l) \\ &= \text{rot}_{180^\circ} \left\{ \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \delta_{i'+m,j'+n}^{l+1} w_{m',n'}^{l+1} \right\} f'(x_{i',j'}^l) \\ &= \delta_{i',j'}^{l+1} * \text{rot}_{180^\circ} \{ w_{m',n'}^{l+1} \} f'(x_{i',j'}^l) \end{aligned}$$

贝叶斯网络(信念网络)

主要是基于波尔兹曼分布或者说波尔兹曼机的模型，也是一种MRF模型，能量模型。

¹. <https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/>

networks/ ↵

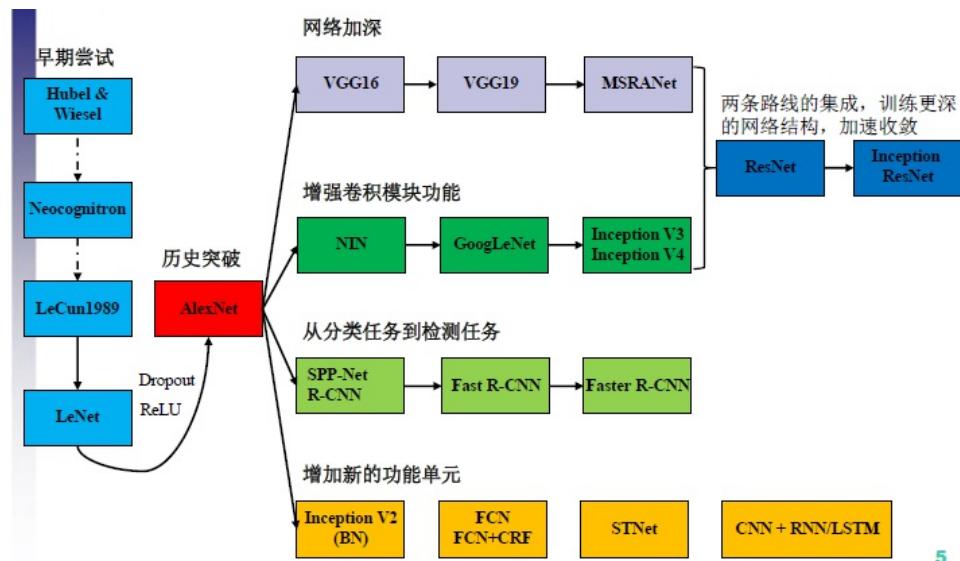
CNN

CNN主要是处理空间数据，比如图像数据。CNN主要包含LeNet-5,AlexNet,GoogLeNet,VGG,Residual Net,Dense Net。

CNN核心：局部感受野，权值共享，时间或空间采样这三种思想来保证某种程度的平移，尺度，形变不变性。

介绍每种模型的时候，要说明这种模型的优点，最好能阐述为啥会有这种优点。

CNN的演化²



5

CNN基础知识

这里，我们将讨论CNN中用到的基础知识，比如：卷积，stride, padding, pooling, 激活函数，误差反向传播，局部感知，权值共享。

如何理解CNN中的卷积？

CNN相对于全连接网络的好处

减少模型参数数量，减少计算时间。

CNN的卷积核与特征提取

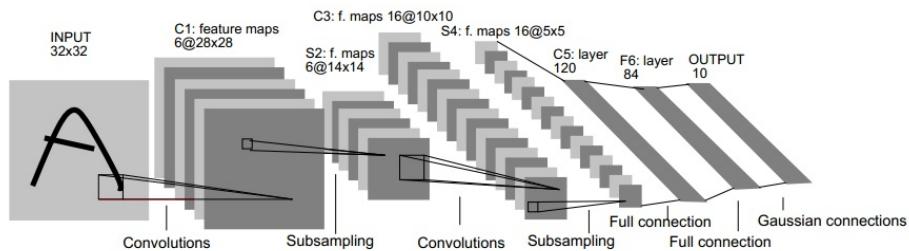
CNN中卷积核数目以及参数数据的计算。根据的是相邻两层feature maps个数，以及每个map的大小来计算出所用卷积核的数目。

CNN的卷积核大小为啥是奇数？

不能保持对称性，比如原来图像长度宽是 $M \times M$ ，卷积核大小是 $n \times n$ ，则卷积作用后图像大小是 $(M-n+1) \times (M-n+1)$ ，如果 n 是偶数，则 $n-1$ 是奇数，因此处理后的图像不再对称了。

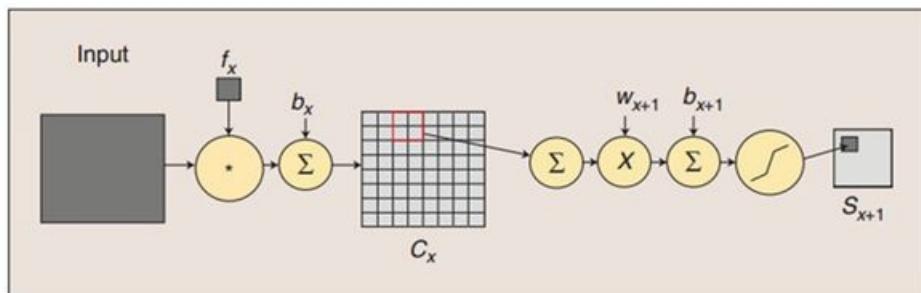
在这里，我们将要计算模型中，两层之间的参数数目。

如何计算CNN两层之间参数数目



C1层是卷积层，单通道下用了6个卷积核，这样就得到了6个feature map，其中每个卷积核的大小为 5×5 ，用每个卷积核与原始的输入图像进行卷积，这样feature map的大小为 $(32-5+1) \times (32-5+1) = 28 \times 28$ ，所需要的参数的个数为 $(5 \times 5+1) \times 6 = 156$ (其中 5×5 为卷积模板参数，1为偏置参数)，连接数为 $(5 \times 5+1) \times 28 \times 28 \times 6 = 122304$ (其中 28×28 为卷积后图像的大小)。

S2层为 pooling 层，也可以说是池化或者特征映射的过程，拥有6个 feature map，每个feature map的大小为 14×14 ，每个feature map的隐单元与上一层C1相对应的feature map的 2×2 单元相连接，这里没有重叠。计算过程是： 2×2 单元里的值相加然后再乘以训练参数 w ，再加上一个偏置参数 b (每一个feature map共享相同 w 和 b)，然后取sigmoid 值，作为对应的该单元的值。所以S2层中每 feature map 的长宽都是上一层C1的一半。S2层需要 $2 \times 6 = 12$ 个参数，连接数为 $(4+1) \times 14 \times 14 \times 6 = 5880$ 。注：这里池化的过程与ufldl教程中略有不同。下面为卷积操作与池化的示意图：



计算CNN相邻两层之间参数数目，假设上层有大小是 $N \times M \times M$ ，其中 n 是通道数目， $M \times M$ 是图像大小。如果连接下一层的卷积核大小是 $n \times n$ ，我们可以对上一层的 a 个通道做卷积，则每次操作需要 a 个卷积核，再加上一个偏置，则需要 $a \times n \times n + 1$ 个参数，假设下一层有 K 个通道，则两层之间所有的参数数目是 $K(a \times n \times n + 1)$ 。这里 a 可以取不同值，也可以是不同值的组合，比如在LeNet-5中，S2到C3之

间就存在不同通道的组合, C3的前6个特征图以S2中3个相邻的特征图子集为输入(1->2->3->4->5->6->1)。接下来6个特征图以S2中4个相邻特征图子集为输入。然后的3个以不相邻的4个特征图子集为输入。最后一个将S2中所有特征图为输入。这样C3层有 $6(3*25+1)+6*(4*25+1)+3*(4_25+1)+(25*6+1)=1516$ 个可训练参数和 $10*10*1516=151600$ 个连接。

CNN怎么实现误差反向传播

CNN的pooling的作用

降维与信息提取。

下采样与上采样的误差怎么传递与反向？

[卷积神经网络\(CNN\)反向传播算法](#)

[CNN中一些特殊环节的反向传播](#)

CNN网络中另外一个不可导的环节就是Pooling池化操作, 因为Pooling操作使得feature map的尺寸变化, 假如做 $2 \times 22 \times 22 \times 2$ 的池化, 假设那么第I+1I+1I+1层的feature map有16个梯度, 那么第III层就会有64个梯度, 这使得梯度无法对位的进行传播下去。其实解决这个问题的思想也很简单, 就是把1个像素的梯度传递给4个像素, 但是需要保证传递的loss(或者梯度)总和不变。根据这条原则, mean pooling和max pooling的反向传播也是不同的。

1、mean pooling

mean pooling的前向传播就是把一个patch中的值求取平均来做pooling, 那么反向传播的过程也就是把某个元素的梯度等分为n份分配给前一层, 这样就保证池化前后的梯度(残差)之和保持不变, 还是比较理解的。

mean pooling比较容易让人理解错的地方就是会简单的认为直接把梯度复制N遍之后直接反向传播回去, 但是这样会造成loss之和变为原来的N倍, 网络是会产生梯度爆炸的。

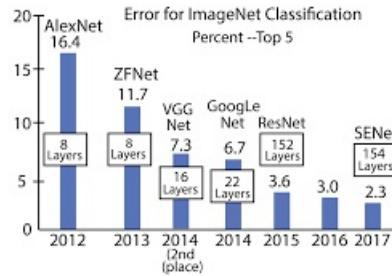
2、max pooling

max pooling也要满足梯度之和不变的原则, max pooling的前向传播是把patch中最大的值传递给后一层, 而其他像素的值直接被舍弃掉。那么反向传播也就是把梯度直接传给前一层某一个像素, 而其他像素不接受梯度, 也就是为0。所以max pooling操作和mean pooling操作不同点在于需要记录下池化操作时到底哪个像素的值是最大, 也就是max id, 这个可以看caffe源码的pooling_layer.cpp, 下面是caffe框架max pooling部分的源码

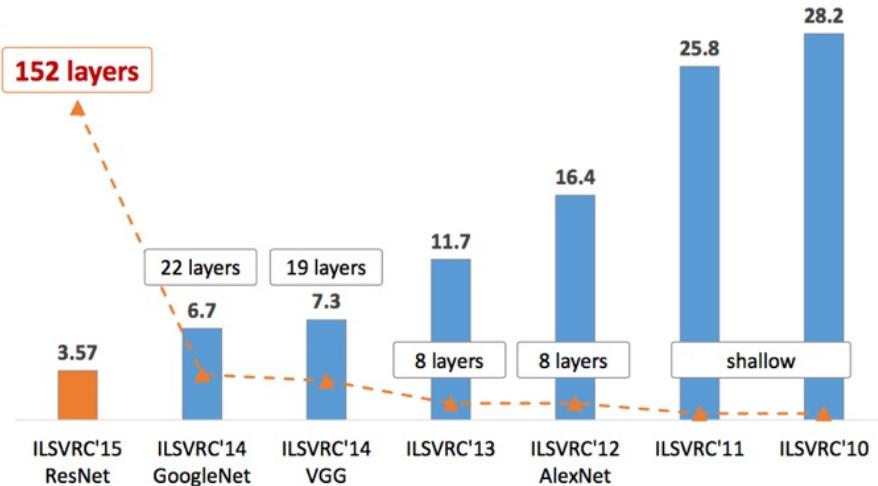
```
// If max pooling, we will initialize the vector index part.  
if (this->layer_param_.pooling_param().pool() == PoolingParameter_PoolMethod_MAX && top.size()  
    == 1)  
{  
    max_idx_.Reshape(bottom[0]->num(), channels_, pooled_height_, pooled_width_);  
}
```

源码中有一个max_idx_的变量，这个变量就是记录最大值所在位置的，因为在反向传播中要用到，那么假设前向传播和反向传播的过程就如下图所示

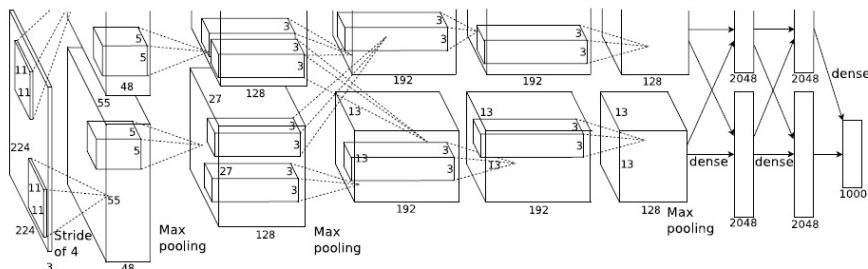
ImageNet top5 error record history



网络层数随历史的变化。



AlexNet



AlexNet的优点,

1. 使用ReLU作为激活函数，消除了sigmoid,tanh造成的梯度消失问题。同时也减小了激活函数的计算量。
2. 百万数据和多GPU训练减小了计算时间。
3. 局部相应归一化(LRN,Local Response Normalization)
4. 重叠池化，也就是pooling的尺寸小于步长。
5. 通过数据增强(裁剪，平移，尺度变换，水平翻转，加随机光照)与Dropout降低过拟合

模型的改进(Simons)

是否可以通过逐渐加深网络来实现浅层的网络而具有强大的功能。具体步骤就是：

1. 一个浅层的网络进行end-end的训练，训练到一定程度的时候，进行第二步
 2. 沿用第一步的权重，通过以Residual Block来微调网络，减小误差，优化方法是SGD，但是我们要保证，每增加一层网络，总的模型的效果会更好。
 3. 循环第二步，直到结果符合我们期望的为止(误差到noise层)
- 理论上，这类似于机器学习中的Adaboost，是否可以分析它的误差会指数衰减？希望能像ML算法一样，能有数学理论来分析深度学习。通过理论指导，像搭积木一样来构建深度神经网络。
- 这种方式与现在的逐层训练有啥区别吗？

Dense Net

DenseNet的示意图，主要表现任意两层间可以有short connection。

在ResNets中，只在相邻两层有skip connection.在激活函数是恒等函数的前提下，有：

$$x_l = H_l(x_{l-1}) + x_{l-1}$$

对于Dense Net, l层可以与0, 1, ..., l-1中任意多层相连。因此有：

$$x_l = H_l(x_0, x_1, \dots, x_{l-1})$$

DenseNet的一个优点是网络更窄，参数更少，大部分原因是得益于这种dense block的设计，后面有提到在dense block中每个卷积层的输出feature map数量都很小(小于100)，而不像其他网络一样动不动就是几百上千的宽度。同样这种连接方式使得特征和梯度的传递更加有效，网络也更容易训练。前面提到过梯度消失问题是网络深度越深的时候越容易出现，原因就是输入信息和梯度信息在很多层之间传递导致的，而现在这种dense connection相当于每一层都直接连接input和loss，因此就可以减轻梯度消失现象，这样更深网络不是问题。

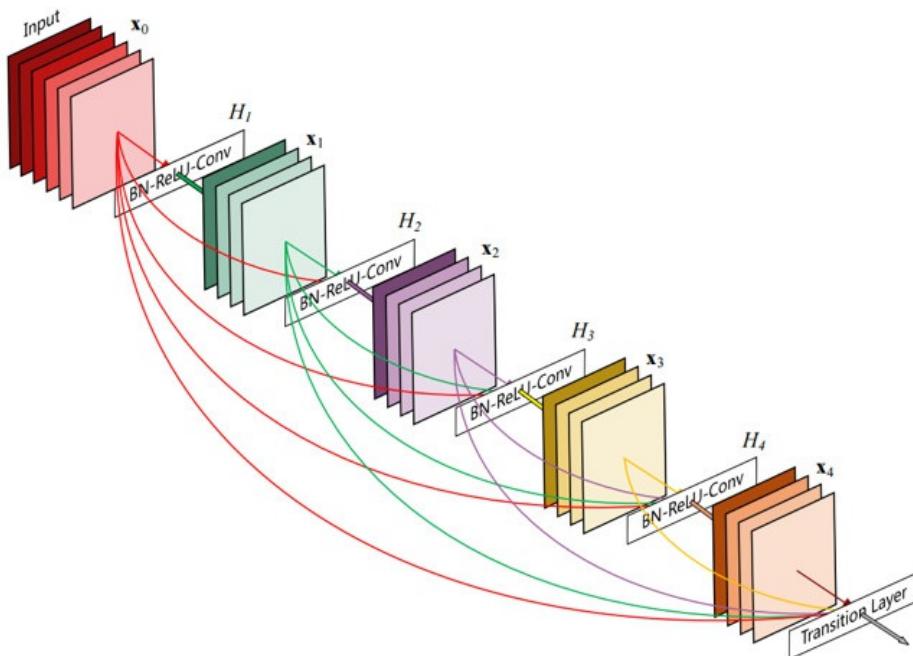
Dense net主要通过建立不同层之间的直接连续，充分利用了feature，进一步减轻了梯度消失问题，加深了网络深度，而且训练效果非常好。另外利用bottleneck layer,translation layer以及较小的

对于Dense Net, l层可以与0, 1, ..., l-1中任意多层相连。因此有：

$$x_l = H_l(x_0, x_1, \dots, x_{l-1})$$

DenseNet的一个优点是网络更窄，参数更少，大部分原因是得益于这种dense block的设计，后面有提到在dense block中每个卷积层的输出feature map数量都很小(小于100)，而不像其他网络一样动不动就是几百上千的宽度。同样这种连接方式使得特征和梯度的传递更加有效，网络也更容易训练。前面提到过梯度消失问题是网络深度越深的时候越容易出现，原因就是输入信息和梯度信息在很多层之间传递导致的，而现在这种dense connection相当于每一层都直接连接input和loss，因此就可以减轻梯度消失现象，这样更深网络不是问题。

Dense net主要通过建立不同层之间的直接连续，充分利用了feature，进一步减轻了梯度消失问题，加深了网络深度，而且训练效果非常好。另外利用bottleneck layer, translation layer以及较小的growth rate使得网络变窄，参数减小，有效抑制了过拟合，同时减小了计算量。



Dense Blocks之间通过pooling与convolution来改变图像尺寸。

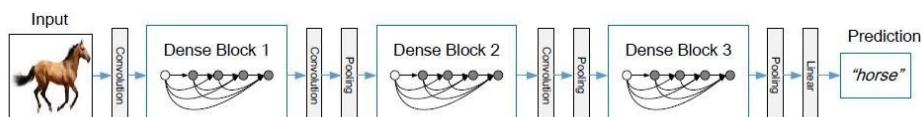


Figure 2. A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.

<http://blog.csdn.net/u014380165>

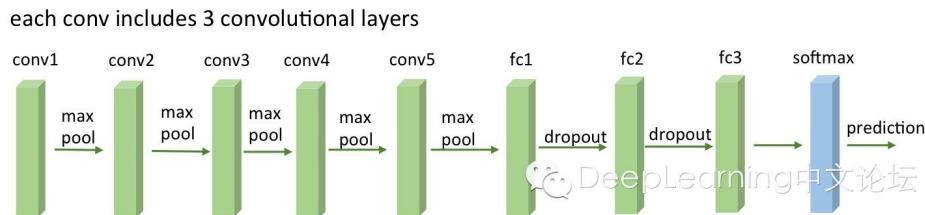
问题集

1. 为什么 1×1 的卷积核可以用来降维？

VGG

相对于以前的网络而言，VGG与GoogLeNet的网络变得更深了。VGGNet在整个网络中使用 3×3 的小感受野，以步长1进行逐像素卷积，因此两个 3×3 卷积相当于一个 5×5 ，三个 3×3 卷积核相当于一个 7×7 的卷积核，这样大大减小了模型的参数个数。使用了小尺寸的卷积核，增加了网络深度并不会带来明显的参数膨胀，却能在更深的网络中获得更高的精度。

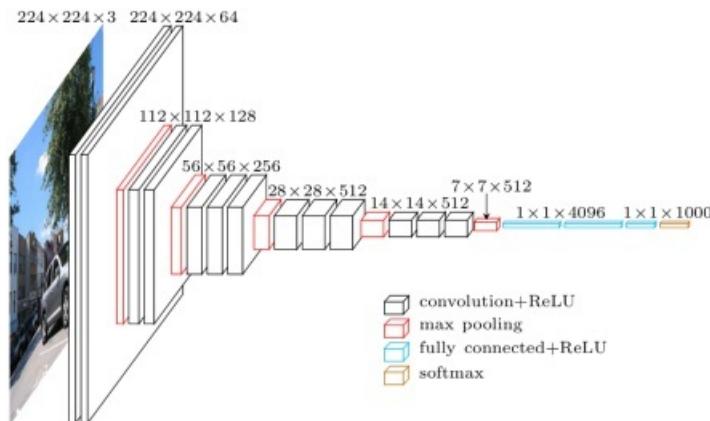
VGG结构



VGG16包含16层，VGG19包含19层。一系列的VGG在最后三层的全连接层上完全一样，整体结构上都包含5组卷积层，卷积层之后跟一个MaxPool。所不同的是5组卷积层中包含的级联的卷积层越来越多。

图像大小变化

AlexNet中每层卷积层中只包含一个卷积，卷积核的大小是 7×7 。在VGGNet中每层卷积层中包含2~4个卷积操作，卷积核的大小是 3×3 ，卷积步长是1，池化核是 2×2 ，步长为2。VGGNet最明显的改进就是降低了卷积核的尺寸，增加了卷积的层数。



VGG11-VGG19

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

知乎 @annulus

原理

- VGG16相比AlexNet的一个改进是采用连续的几个 3×3 的卷积核代替AlexNet中的较大卷积核(11×11 , 7×7 , 5×5)。
比如用2个 3×3 代替一个 5×5 ²

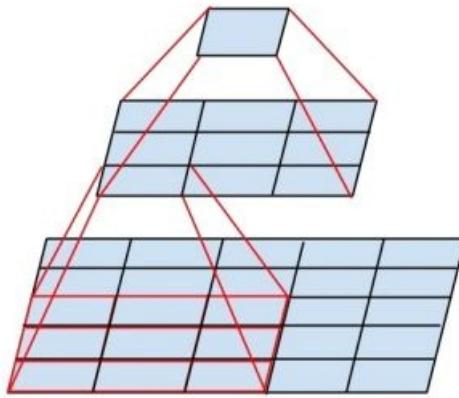


Figure 1. Mini-network replacing the 5×5 convolutions.

2. 在VGGNet的卷积结构中，引入 1×1 的卷积核，在不影响输入输出维度的情况下，引入非线性变换，增加网络的表达能力，降低计算量。
3. 训练时，先训练级别简单(层数较浅)的VGGNet的A级网络，然后使用A网络的权重来初始化后面的复杂模型，加快训练的收敛速度。
4. VGG的多尺度训练
VGGNet使用了Multi-Scale的方法做数据增强，将原始图像缩放到不同尺寸S，然后再随机裁切224'224的图片，这样能增加很多数据量，对于防止模型过拟合有很不错的效果。实践中，作者令S在[256,512]这个区间内取值，使用Multi-Scale获得多个版本的数据，并将多个版本的数据合在一起进行训练。VGG作者在尝试使用LRN之后认为LRN的作用不大，还导致了内存消耗和计算时间增加¹。

相对于AlexNet的改进

1. 去掉了LRN层，作者发现深度网络中LRN的作用并不明显，干脆取消了
2. 采用更小的卷积核- 3×3 ，Alexnet中使用了更大的卷积核，比如有 7×7 的，因此VGG相对于Alexnet而言，参数量更少
3. 池化核变小，VGG中的池化核是 2×2 , stride为2, Alexnet池化核是 3×3 , 步长为2

VGG的意义³

特征提取器

众所周知，VGG是一个良好的特征提取器，其与训练好的模型也经常被用来做其他事情，比如计算perceptual loss(风格迁移和超分辨率任务中)，尽管现在resnet和inception网络等等具有很高的精度和更加简便的网络结构，但是在特征提取上，VGG一直是一个很好的网络，所以说，当你的某些

任务上resnet或者inception等表现并不好时，不妨试一下VGG，或许会有意想不到的结果。
VGG之所以是一个很好的特征提取器，除了和它的网络结构有关，我认为还和它的训练方式有关系，VGG并不是直接训练完成的，它使用了逐层训练的方法。

分析到这里可以得出结论，VGG对于Alexnet来说，改进并不是很大，主要改进就在于使用了小卷积核，网络是分段卷积网络，通过maxpooling过度，同时网络更深更宽。

1. VGGNet网络结构 <https://blog.csdn.net/dcrmg/article/details/79254654> ↵
2. 一文读懂VGG网络 <https://zhuanlan.zhihu.com/p/41423739> ↵
3. VGG网络结构分析 https://blog.csdn.net/qq_25737169/article/details/79084205 ↵

GoogleNet

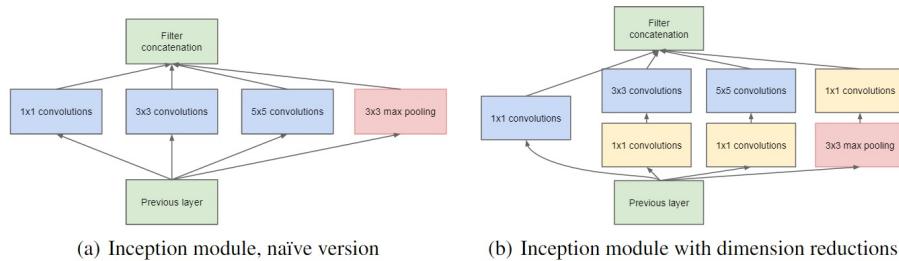
特点：

1. 不使用全连接层，而是用平均池化代替全连接层，减小模型的参数数量，这种想法来自于NIN。
2. 采用一种高效的机器视觉深度神经网络结构，称为“Inception”，这种模块化结构方便增加与修改，现在inception版本到了V6/V7。GoogLeNet中采用了9个inception模块。
3. 为了避免梯度小时，网络额外增加了2个辅助的softmax用于向前传导梯度。

Inception v1模型¹

1. Inception V1：它将 1×1 , 3×3 , 5×5 的conv和 3×3 的pooling, 堆叠在一起，一方面增加了网络的width, 另一方面增加了网络对尺度的适应性；
2. Inception先通过一个 1×1 的低channel的卷积核，来对后面的大的卷积核起降维作用，进而控制参数量。

GoogLeNet的参数：可以看出里面用 7×7 的平均池代替了全连接。



1×1 的卷积核与Inception³

1. 实现跨通道的交互和信息整合

对于 $n \times n$ ($n > 1$) 的卷积核，我们通常还要考虑pad(边缘补0的个数), stride(每次卷积移动的步长)。但是当尺寸是 1×1 时，对于single channel而言就相当于对原特征的scala操作；但是我们一般遇到的都是multi-channel的情况，此时我们便可以根据自己的需要定义卷积核的个数，从而进行降(升)维。如上面所说，如果将它看作cross channel的pooling操作，它还能帮我们得到在同一位置不同通道之间进行特征的aggregation。

2. 进行卷积核通道数的降维和升维 左边是naive的inception模块，右边是加入 1×1 convolution进行降维的inception。假设输入的feature map是 $28 \times 28 \times 192$ ，其中192表示通道数目。卷积核大小以及卷积通道数(包括三种卷积核，分别是 $1 \times 1 \times 64$, $3 \times 3 \times 128$, $5 \times 5 \times 32$)，右图中在 3×3 , 5×5 convolution前新加入的 1×1 的卷积核为96和16通道的，在max pooling后加入的 1×1 卷积为32通道。

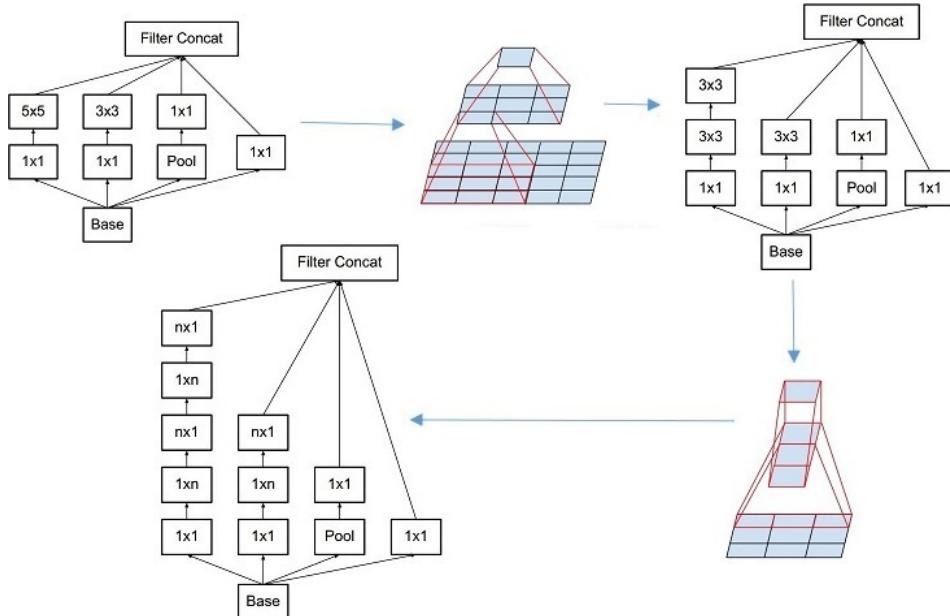
那么图a该层的参数为 $(1 \times 1 \times 192 \times 64) + (3 \times 3 \times 192 \times 128) + (5 \times 5 \times 192 \times 32)$

图b该层的参数为 $(1 \times 1 \times 192 \times 64) + (1 \times 1 \times 192 \times 96) + (1 \times 1 \times 192 \times 16) + (3 \times 3 \times 96 \times 128) + (5 \times 5 \times 16 \times 32) + (1 \times 1 \times 192 \times 32)$ 。比较可知，模型参数减少了。

Inception v2模型

一方面加入了BN层，减少了Internal Covariate Shift(内部neuron的数据分布发生变化)，使每一层的输出都规范化到一个 $N(0, 1)$ 的高斯；

另外一方面学习VGG用2个 3×3 的conv替代inception模块中的 5×5 ，既降低了参数数量，也加速计算；



使用 3×3 的已经很小了，那么更小的 2×2 呢？ 2×2 虽然能使得参数进一步降低，但是不如另一种方式更加有效，那就是Asymmetric方式，即使用 1×3 和 3×1 两种来代替 3×3 的卷积核。这种结构在前几层效果不太好，但对特征图大小为12~20的中间层效果明显。

Inception v3模型

v3一个最重要的改进是分解(Factorization)，将 7×7 分解成两个一维的卷积($1 \times 7, 7 \times 1$)， 3×3 也是一样($1 \times 3, 3 \times 1$)，这样的好处，既可以加速计算(多余的计算能力可以用来加深网络)，又可以将1个conv拆成2个conv，使得网络深度进一步增加，增加了网络的非线性，还有值得注意的地方是网络输入从 224×224 变为了 299×299 ，更加精细设计了 $35 \times 35 / 17 \times 17 / 8 \times 8$ 的模块。

Inception v4模型

v4研究了Inception模块结合Residual Connection能不能有改进？发现ResNet的结构可以极大地加速训练，同时性能也有提升，得到一个Inception-ResNet v2网络，同时还设计了一个更深更优化的Inception v4模型，能达到与Inception-ResNet v2相媲美的性能。

后续改进的版本总结：

1. Inception-v2在之前的版本中主要加入了batch Normalization;另外也借鉴VGGNet的思想,用两个3x3的卷积代替5x5的卷积,不仅降低了训练参数,也提升了速度。
2. Inception-3在v2的基础上进一步分解大的卷积,比如把nxn的卷积拆分成两个一维卷积: $1 \times n, n \times 1$ 。
3. Inception-V4借鉴了ResNet可以构建更深层网络的思想,设计了一个更深更优化的模型。

不同Inception模块复杂度

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2	64	192					112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

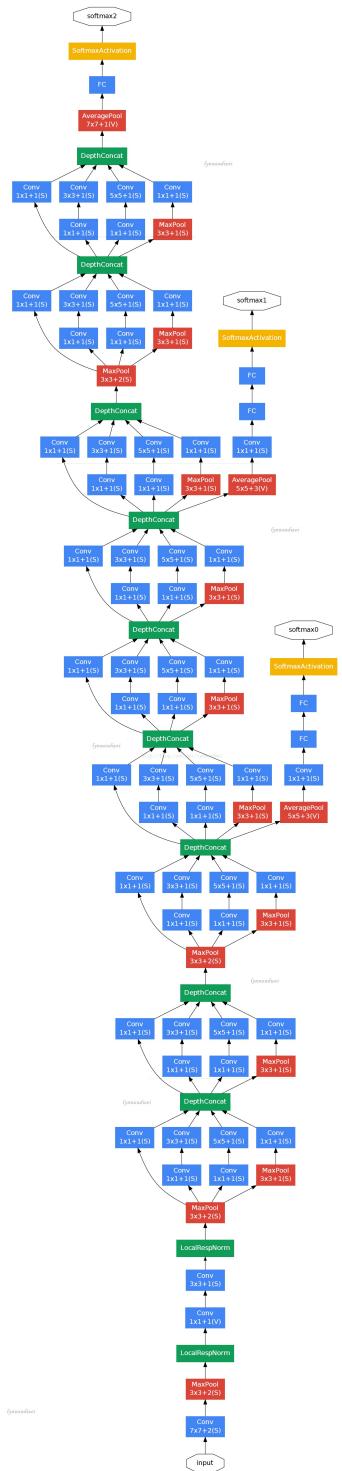
Auxiliary Classifiers²

需要注意的是,为了避免梯度消失,网络额外增加了2个辅助的softmax用于向前传导梯度。文章中说这两个辅助的分类器的loss应该加一个衰减系数,实际测试的时候,这两个额外的softmax会被去掉。

在GoogLeNet中,使用了多余的在底层的分类器,直觉上可以认为这样做可以使底层能够在梯度下降中学的比较充分,但在实践中发现两条:

1. 多余的分类器在训练开始的时候并不能起到作用,在训练快结束的时候,使用它可以有所提升最底层的那个多余的分类器去掉以后也不会有损失。
2. 以为多余的分类器起到的是梯度传播下去的重要作用,但通过实验认为实际上起到的是regularizer的作用,因为在多余的分类器前添加dropout或者batch normalization后效果更佳。

GoogleNet structure



1. 深入浅出——网络模型中Inception的作用与结构全解析深入浅出——网络模型中
Inception的作用与结构全解析 <https://blog.csdn.net/u010402786/article/details/52433324> ↵
2. Inception in CNN <https://blog.csdn.net/stdcoutzyx/article/details/51052847> ↵
3. 1*1的卷积核与Inception <https://blog.csdn.net/a1154761720/article/details/53411365> ↵

Residual NN

Residual NN提出来的背景：

随着网络变深，训练误差与测试误差得提高了。这是违反我们的训练的初衷的，因为即使我们把36层后面的网络变成恒等映射，效果也不会变差。

存在这种随着网络层数增加，会出现如下两个问题：

1. 梯度消失或者爆炸，导致训练难以收敛。然而梯度消失/爆炸的问题，很大程度上可以通过标准的初始化和正则化层来基本解决，确保几十层的网络能够收敛（用SGD+反向传播）¹。
2. 随着深度增加，模型的训练误差与测试误差会迅速下滑，这样一种退化，并不是过拟合导致的，并且增加更多的层匹配深度模型，会导致更多的训练误差。这种现象在CIFAR-10和ImageNet中都有提及。下图就是一个典型的例子。

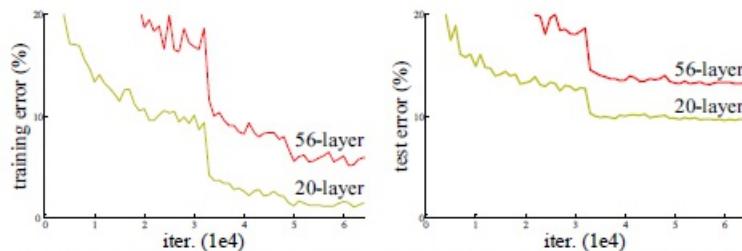


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Residual-NN的实现

参考He Kaiming的这篇文章

Residual Block的设计如下：

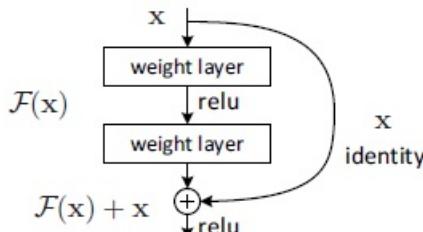


Figure 2. Residual learning: a building block.

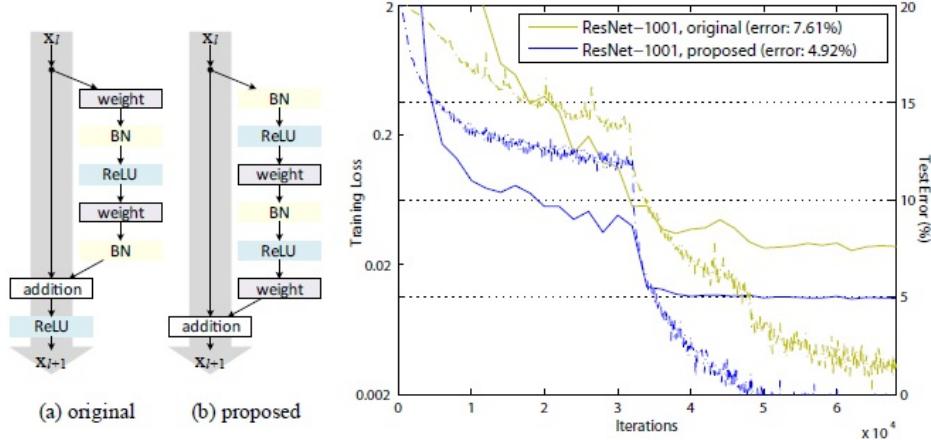
对于每一个Residual Units：

$$y_l = h(x_l) + F(x_l, W_l)$$

$$x_{l+1} = f(y_l)$$

其中函数h一般取恒等映射 $h(x_l) = x_l$, 实验发现f取恒等映射能实现最快的误差下降与最低的训练误差;f是激活函数, 取ReLU。F的残差函数。

一个推荐的网络结构:



如果激活函数f取恒等映射 $y_l = f(y_l)$, 则有:

$$\begin{aligned} x_{l+1} &= x_l + F(x_l, W_l) \\ x_L &= x_l + \sum_{i=l}^{L-1} F(x_i, W_i) \end{aligned}$$

因此对于误差反向传播:

$$\frac{\partial \epsilon}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L} \frac{\partial x_L}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L} \left(1 + \frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F(x_i, W_i)\right)$$

因为 $\frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F(x_i, W_i)$ 不会常等于-1, 因此梯度不会消失, 即使当权重非常小的时候。

因此信号对于正向与反向传播, 可以直接的从一个单元传到另外一个单元。条件就是skip connection 函数h, 激活函数f是恒等映射。函数h是恒等映射时保证梯度不消失或者爆炸的关键。他们使用的mini-batch是128, 用了2GPUs, 权重衰减是0.0001, 动量是0.9, 权重初始化了。详细的可以看上面给的论文链接。权重每30个epoch变成原来的十分之一。

Residual NN模型效果:

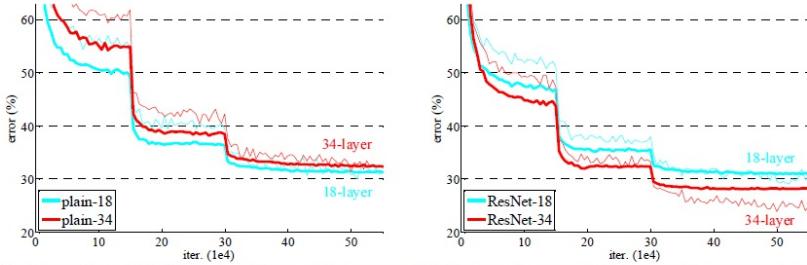


Figure 4. Training on ImageNet. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Resi-NN能解决以往随着模型加深训练与测试误差变大的现象。与其它网络的结果比较。

CIFAR-10	error (%)	CIFAR-100	error (%)
NIN [13]	8.81	NIN [13]	35.68
DSN [14]	8.22	DSN [14]	34.57
FitNet [15]	8.39	FitNet [15]	35.04
Highway [7]	7.72	Highway [7]	32.39
ELU [12]	6.55	ELU [12]	24.28
FitResNet, LSUV [16]	5.84	FitNet, LSUV [16]	27.66
ResNet-110 [1] (1.7M)	6.61	ResNet-164 [1] (1.7M)	25.16
ResNet-1202 [1] (19.4M)	7.93	ResNet-1001 [1] (10.2M)	27.82
ResNet-164 [ours] (1.7M)	5.46	ResNet-164 [ours] (1.7M)	24.33
ResNet-1001 [ours] (10.2M)	4.92 (4.89 ± 0.14)	ResNet-1001 [ours] (10.2M)	22.71 (22.68 ± 0.22)
ResNet-1001 [ours] (10.2M) [†]	4.62 (4.69 ± 0.20)		

¹. [译] Deep Residual Learning for Image Recognition (ResNet)

<https://www.jianshu.com/p/f71ba99157c7> ↵

RNN

建模序列化数据的一种主流深度学习模型。

设计目的

捕获长距离输入之间的依赖。

背景：

传统的前馈神经网络一般输入的都是一个定长的向量，无法处理变长的序列信息，即使通过一些方法把序列处理成定长的向量，模型也很难捕捉序列中的长距离依赖关系。RNN则通过将神经元串行起来处理序列化的数据。由于每个神经元能用它的内部变量保存之前输入的序列信息，因此整个序列被浓缩成抽象的表示，并可以据此进行分类或生成新的序列。近年来，得益于计算能力的大幅度提升和模型的改进，RNN在很多领域取得了突破性的进展——机器翻译，图像描述，推荐系统，智能聊天机器人，自动作词作曲等。

RNN, LSTM, GRU比较

传统RNN vs 门控RNN

传统RNN的问题：长期依赖会造成梯度消失（多数情况下）或梯度爆炸（少数情况下）。

解决以上问题的方案：

1. 以新的方法改善或者代替传统的SGD方法，如Bengio提出的clipped gradient；
2. 设计更为精妙的activation function或recurrent unit，如LSTM和GRU。原因：LSTM和GRU都能通过各种Gate将重要特征保留，保证其在long-term传播的时候也不会被丢失；还有一个作用就是有利于BP的时候不容易梯度消失。

传统RNN和门控RNN的不同点：

1. 传统RNN会每一步都重写“记忆”，而门控RNN可以在某些步骤保持原有“记忆”；
2. 门控RNN收敛速度更快，泛化能力更好。

LSTM vs GRU

相同点：都属于“加模型”

不同点：

1. LSTM可控“记忆”的曝光度，而GRU完全暴露“记忆”，是不可控的；
2. LSTM的新“记忆”是与forget gate独立的，而GRU的新“记忆”受update gate控制

从LSTM和GRU的公式里面可以看出，都会有门操作，决定是否保留上时刻的状态和是否接收此时刻的外部输入，LSTM是用遗忘门和输入门来做到的，GRU则是只用一个更新门(z_t)。这种设计有两种解释，一种解释是说，网络是能很容易记住长依赖问题。即前面很久之前出现过一个重要的特征，如果遗忘门或者更新门选择不重写内部的memory，那么网络就会一直记住之前的重要特征，那么会对当前或者未来继续产生影响。另一点是，这种设计可以不同状态之间提供一条捷径，那么梯度回传的时候不会消失的太快，因此减缓了梯度消失带来的训练难问题。

LSTM 和GRU的不同点。首先LSTM有一个输出门来控制memory content的曝光程度，而GRU则是直接输出。另外一点是要更新的new memory content的来源不同， \hat{h}_t 会通过重置门控制从 h_{t-1} 中得到信息的力度，而 \hat{c}_t 则没有，而是直接输入 h_{t-1} 。

注意力机制

1 RNN-LSTM-GRU-最小GRU <https://www.jianshu.com/p/166db8ab3cef>

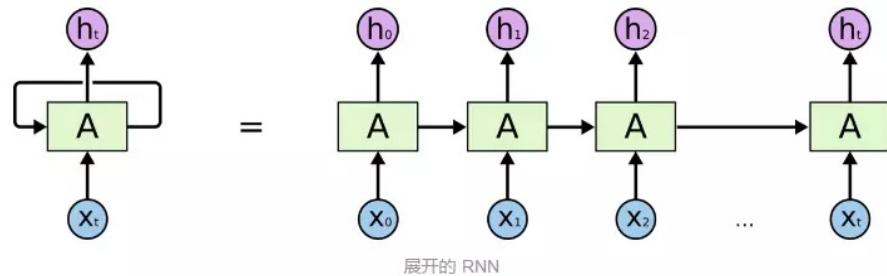
2 RNN LSTM与GRU深度学习模型学习笔记 <https://blog.csdn.net/softee/article/details/54292102>

3 RNN以及LSTM的介绍和公式梳理 https://blog.csdn.net/Dark_Scope/article/details/47056361

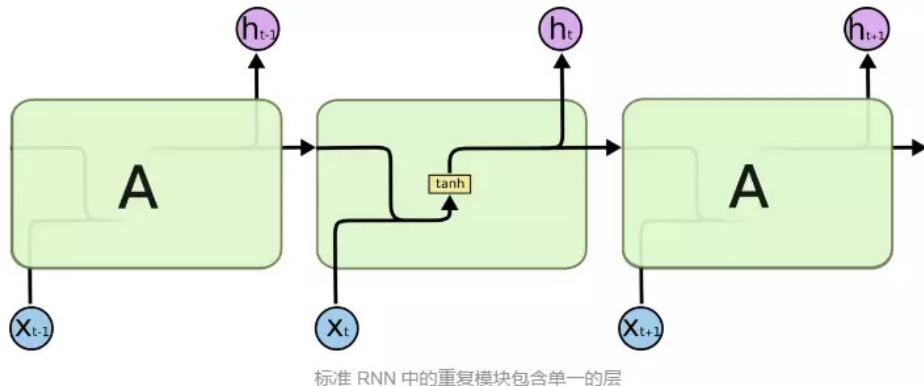
循环神经网络(RNN)

最简单的RNN

网上最易懂得结构图¹:

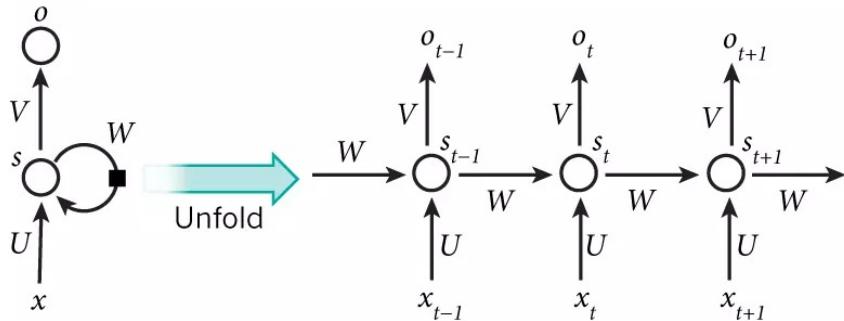


一个简单的RNN内部的结构是:



展开图:

如果我们把上面的图展开，循环神经网络也可以画成下面这个样子：



现在看上去就比较清楚了，这个网络在t时刻接收到输入 x_t 之后，隐藏层的值是 s_t ，输出值是 o_t 。关键一点是， s_t 的值不仅仅取决于 x_t ，还取决于 s_{t-1} 。我们可以用下面的公式来表示循环神经网络的计算方法：

$$o_t = g(Vs_t) \quad (式1)$$

$$s_t = f(Ux_t + Ws_{t-1}) \quad (式2)$$

式1是输出层的计算公式，输出层是一个全连接层，也就是它的每个节点都和隐藏层的每个节点相连。 V 是输出层的权重矩阵， g 是激活函数。式2是隐藏层的计算公式，它是循环层。 U 是输入 x 的权重矩阵， W 是上一次的值 s_{t-1} 作为这一次的输入的权重矩阵， f 是激活函数。

从上面的公式我们可以看出，循环层和全连接层的区别就是循环层多了一个权重矩阵 W 。

RNN主要由矩阵 U, W, V ,以及激活函数 f, g 决定。

图中左边是RNN的一个基本模型，右边是模型展开之后的样子。展开是为了与输入样本匹配。假若输入时汉语句子，每个句子最长不超过20(包含标点符号)，则把模型展开20次²。

1. x_t 代表输入序列中的第 t 步元素，例如语句中的一个汉字。一般使用一个one-hot向量来表示，向量的长度是训练所用的汉字的总数(或称之为字典大小)，而唯一为1的向量元素代表当前的汉字。
2. s_t 代表第 t 步的隐藏状态，其计算公式为 $s_t = \tanh(Ux_t + Ws_{t-1})$ 。也就是说，当前的隐藏状态由前一个状态和当前输入计算得到。考虑每一步隐藏状态的定义，可以把 s_t 视为一块内存，它保存了之前所有步骤的输入和隐藏状态信息。 s_{-1} 是初始状态，被设置为全0。
3. o_t 是第 t 步的输出。可以把它看作是对第 $t+1$ 步的输入的预测，计算公式为： $o_t = \text{softmax}(Vs_t)$ 。可以通过比较 o_t 和 x_{t+1} 之间的误差来训练模型。

U, V, W 是RNN的参数，并且在展开之后的每一步中依然保持不变。这就大大减少了RNN中参数的数量。

一个例子

假设我们要训练的中文样本中一共使用了3000个汉字，每个句子中最多包含50个字符，则RNN中每个参数的类型可以定义如下。

1. $x_t \in R^{3000}$, 第t步的输入, 是一个one-hot向量, 代表3000个汉字中的某一个。
2. $o_t \in R^{3000}$, 第t步的输出, 类型同 x_t 。
3. $s_t \in R^{50}$, 第t步的隐藏状态, 是一个包含50个元素的向量。RNN展开后每一步的隐藏状态是不同的。
4. $U \in R^{50 \times 3000}$, 在展开后的每一步都是相同的。
5. $V \in R^{3000 \times 50}$, 在展开后的每一步都是相同的。
6. $W \in R^{50 \times 50}$, 在展开后的每一步都是相同的。

其中 x_t 是输入, U, V, W 是参数, s_t 是由输入和参数计算所得到的隐藏状态, 而 o_t 则是输出。 s_t 和 o_t 的计算公式已经给出, 为清晰起见, 重新写出。

$$s_t = \tanh(Ux_t + Vs_{t-1})$$

$$o_t = \text{softmax}(Vs_t)$$

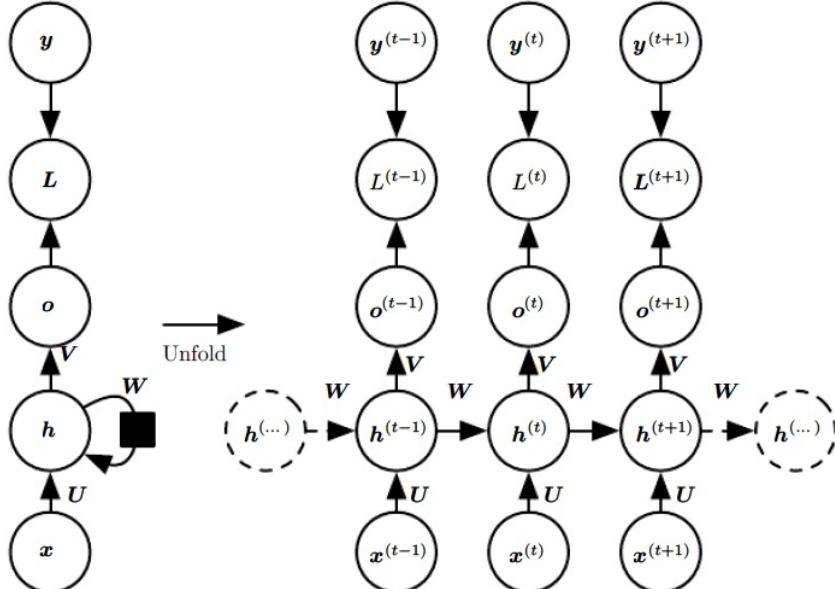
要训练的就是参数矩阵 U, V, W

RNN Goodfellow

循环神经网络中一些重要的设计模式包含如下几种:

(1) 每个时间步都有输出, 并且隐藏单元之间有循环连接的循环网络, 如下图所示。

RNN的输入到隐藏的连接由权值矩阵 U 参数化, 隐藏到隐藏的循环由权重矩阵 W 参数化, 隐藏到输出层的连接由权重矩阵 V 参数化。 $L^{(t)}$ 是损失函数。



前向传播公式：假设激活函数是tanh，我们有如下的更新方程：

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a})$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)})$$

其中 \mathbf{b}, \mathbf{c} 是偏置向量， $\mathbf{U}, \mathbf{V}, \mathbf{W}$ 是权重矩阵。

这一循环网络将一个输入序列映射到相同长度的输出序列。与 \mathbf{x} 序列配对的 \mathbf{y} 的总的损失就是所有时间步的损失之和。例如， $L^{(t)}$ 为给定的 $x^{(1)}, x^{(2)}, \dots, x^{(t)}$ 后 $y^{(t)}$ 的负对数似然，则：

$$L((x^{(1)}, x^{(2)}, \dots, x^{(\tau)}), (y^{(1)}, y^{(2)}, \dots, y^{(\tau)})) = \sum_t L^{(t)}$$

$$= - \sum_t \log p_{\text{model}}(y^{(t)} | (x^{(1)}, x^{(2)}, \dots, x^{(t)})$$

其中 $p_{\text{model}}(y^{(t)} | (x^{(1)}, x^{(2)}, \dots, x^{(t)})$ 需要读取模型输出向量 $\hat{\mathbf{y}}^{(t)}$ 对应于 $\mathbf{y}^{(t)}$ 的项。

应用于展开图且代价为 $O(\tau)$ 的反向传播算法是通过时间反向传播(back-propagation through time, BPTT)。

(2) 每个时间步都产生一个输出，只有当前时刻的输出到下个时刻的隐藏单元之间有循环连接的循环网络。如下图所示：

该图中，RNN被训练将特定输出值放入 \mathbf{o} 中，并且 \mathbf{o} 是被允许传播到未来的唯一信息。此处没有从 \mathbf{h} 前向传播的直接连接。之前的 \mathbf{h} 仅通过产生的预测间接地连接到当前。 \mathbf{o} 通常缺乏过去的重要信息，除非它非常高维并且内容丰富。这使得该图中的RNN不那么强大，但是它更容易训练，因为每个时间步可以与其他时间步分离训练，允许训练期间更多的并行化。

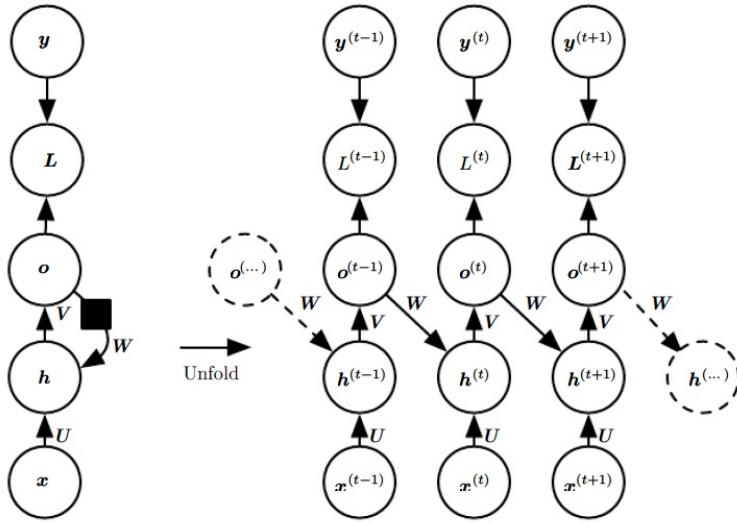


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t , the input is \mathbf{x}_t , the hidden layer activations are $\mathbf{h}^{(t)}$, the outputs are $\mathbf{o}^{(t)}$, the targets are $\mathbf{y}^{(t)}$ and the loss is $L^{(t)}$. (Left) Circuit diagram.

(3) 隐藏单元之间存在循环连接，但读取整个序列后产生单个输出的循环网络，如下图所示：
这样的网络可以用于概括序列并产生用于进一步处理的固定大小的表示。在结束处可能存在目标，或者通过更下游模块的反向传播来获得输出 $\mathbf{o}^{(t)}$ 上的梯度。

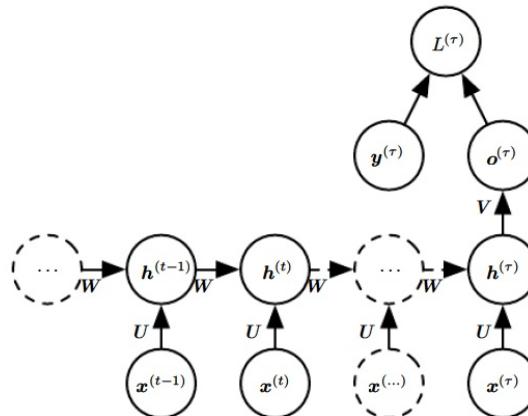
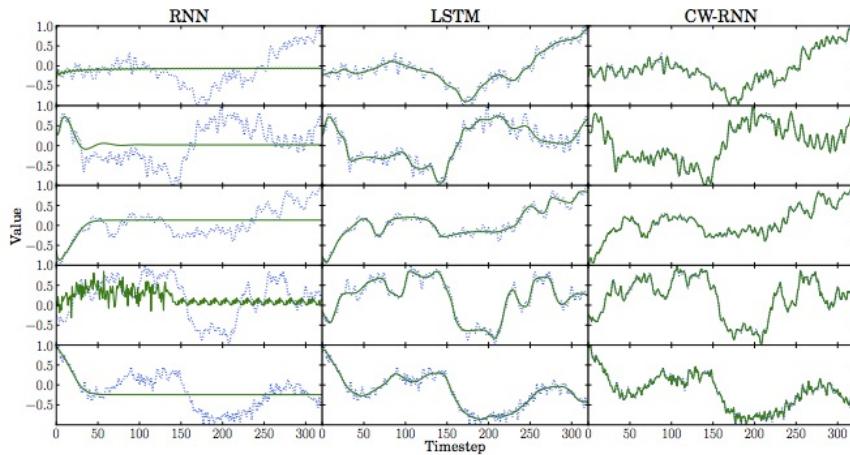


Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $\mathbf{o}^{(t)}$ can be obtained by back-propagating from further downstream modules.

Clockwork RNNs

CW-RNNs, 时钟频率驱动循环神经网络



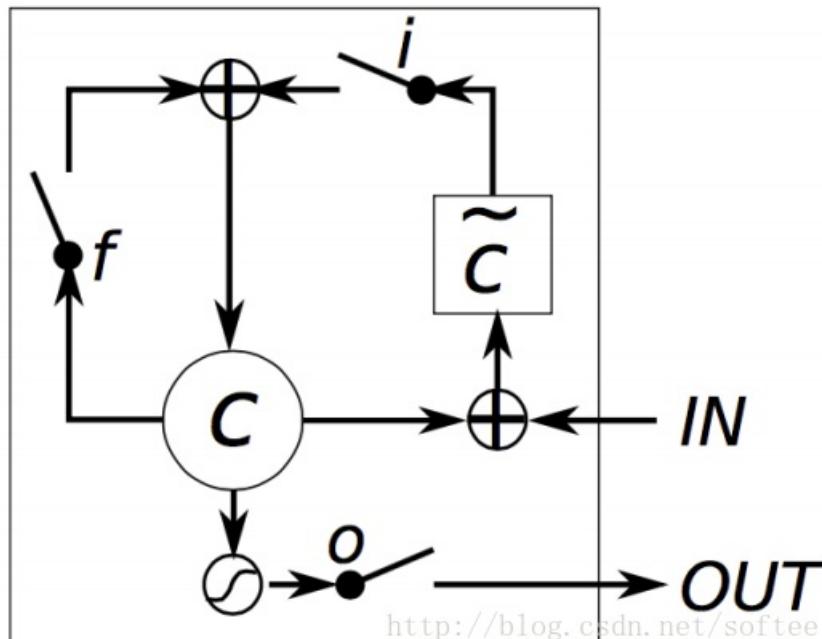
上图中，绿色实线是预测结果，蓝色散点是真实结果。每个模型都是对前半部分进行学习，然后预测后半部分。LSTMs模型类似滑动平均，但是CW-RNNs效果更好。其中三个模型的输入层、隐藏层、输出层的节点数都相同，并且只有一个隐藏层，权值都使用均值为0，标准差为0.1的高斯分布进行初始化，隐藏层的初始状态都为0，每一个模型都使用Nesterov-style momentum SGD(Stochastic Gradient Descent, 随机梯度下降算法)进行学习与优化

LSTM

由于Vanilla RNN具有梯度消失问题，对长关系的依赖的建模能力不够强大，也就是很长时刻以前的输入，对现在的网络影响非常小，后向传播那些梯度也很难影响很早以前的输入，即会出现梯度消失的问题。而LSTM通过构建一些门，让网络能记住那些非常重要的信息，而这个核心的结构就是cell state。比如遗忘门，来选择性清空过去的记忆和更新较新的信息。

两种常见的LSTM结构。

第一种是带遗忘门的Traditional LSTM。



公式如下：

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$

$$h_t = o_t * \sigma_h(c_t)$$

"**"表示向量中对应元素相乘。 c_t 是cell中t时刻保存的信息， h_t 表示t时刻的输出。 f_t, i_t, o_t, c_t, h_t 都是向量。

前三行是三个门，分别是遗忘门 f ，输入门 i ，输出门 o ，输入都是 x_t, h_{t-1} ，只是参数不同，然后要经过一个激活函数把值缩放到 $[0, 1]$ 之间。

第四行 c_t 是 cell state, 由上一时刻的 c_{t-1} 和输入得到, 两者相互独立。如果遗忘门 f_t 取 0 的话, 也就是调节参数 W_f, U_f 使得对于任意的输入 x_t 与上一次的输出 h_{t-1} 都有 f_t 趋于 0 向量, 那么上一时刻的状态就会全部被清空, 然后只关注此时刻的输入, 这与传统的 RNN 相似, 只是多了一个 σ_h 操作。输入门 i_t 决定是否接收此时刻的输入。最后输出门 o_t 决定是否输出 cell state。

$$\hat{c} = \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \text{ 因此有 } c_t = f_t * c_{t-1} + i_t * \hat{c}.$$

这样一来 cell state 的更新来源就明显了, 一部分是上时刻的自己, 一部分是 new memory content, 而且两个来源是相互独立地由两个门控制的。遗忘门控制是否记住以前的那些特征, 输入们决定是否接受当前的输入。后面可以看到 GRU 其实把这两个门合二为一了。

总结

加入输入门与遗忘门, 就是为了控制上一时刻与当前时刻的内容在 cell state c_t 中的比列, f_t, i_t 的值介于 [0, 1], 当取 0 的时候, 就表示对应的内容不添加在 cell state 中。因此, 当前 cell state 的内容由上一时刻 cell state 中的内容以及这一时刻的输入决定²。

Peephole LSTM

第二种是带遗忘门的 Peephole LSTM, 公式如下:

$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$

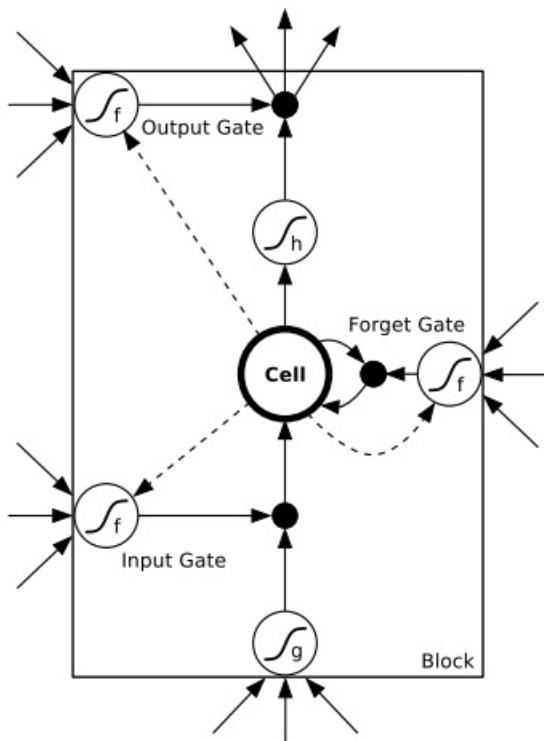
$$c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$$

$$h_t = o_t * \sigma_h(c_t)$$

和上面的公式比较, 发现只是把 h_{t-1} 换成了 c_{t-1} , 即三个门的输入都改成了 $[x_t, c_{t-1}]$, 因为是从 cell

state里取得信息，所以叫窥视管(peephole)。

把这两种结构结合起来，可以用如下图描述：



图中连着门的那些虚线都是peephole。三个输入都是 x_t, h_{t-1}, c_{t-1}

GRU

参考 [循环神经网络\(RNN, Recurrent Neural Networks\)介绍](#)

GRU这个结构2014年才出现，结构与LSTM类似，效果一样，但是精简一些，参数更少。公式如下：

$$z_t = \sigma(W_z x_t + U_z h_{t-1})$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1})$$

$$\hat{h}_t = \tanh(W x_t + U(r_t * h_{t-1}))$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \hat{h}_t$$

四行的解释如下：

z_t 是update gate, 更新activation时的逻辑门。

r_t 是reset gate, 决定candidate activation时, 是否要放弃以前的activate h_t

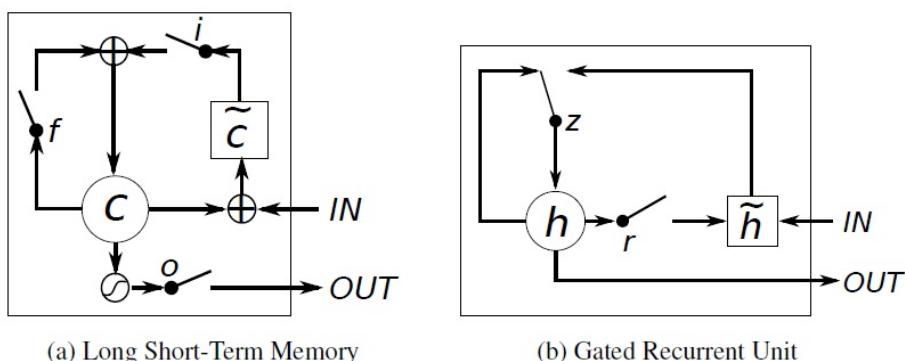
\hat{h}_t 是candidate activation, 接收 x_t, h_{t-1}

h_t 是activation, 是GRU的隐层, 接收 h_{t-1}, \hat{h}_t

GRU相对于LSTM, 它取消了输出门。由更新门与重置门共同控制怎么从上一刻的的隐藏状态到当前的隐藏状态 h_t 。

其中 r_t 用来控制需要保留多少之前的记忆, 比如如果 r_t 为0, 那么 \hat{h}_t 只包含当前词的信息。 z_t 控制需要从前一时刻的隐藏层 h_{t-1} 中遗忘多少信息, 需要加入多少当前时刻的隐藏层信息 \hat{h}_t , 最后得到 h_t 。

如果reset门取1而update门取1, 则退化到RNN。



现在看这图, 就清晰很多, i, o, f 都是门向量, 作用就是控制流过这些门的流量, 使得通过向量各个分量的通过门之后, 其分量的值变为原来的 x 倍, $x \in [0, 1]$ 是门的系数。就连输出也有一个输出门向量来控制。

计算机视觉

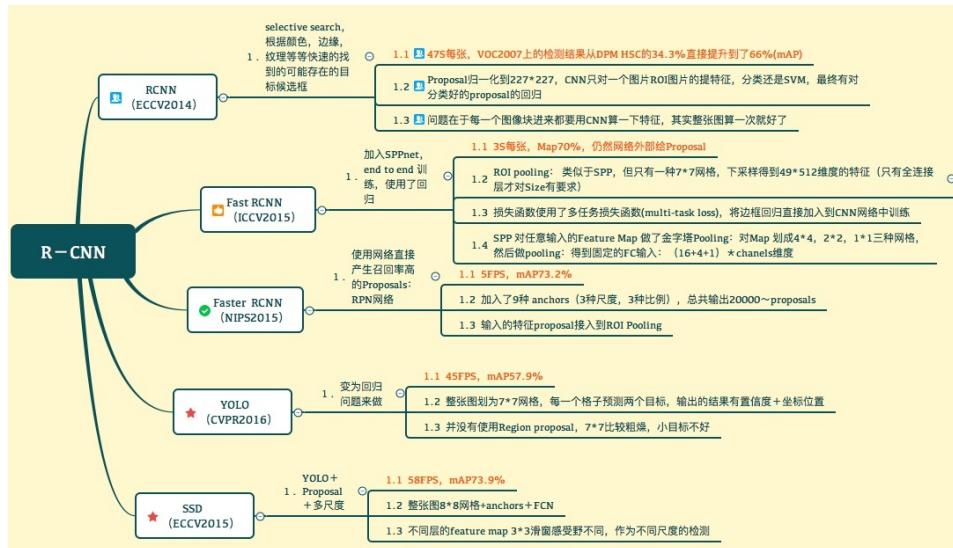
在CNN那一节，我们主要讨论了利用CNN来做图像分类。在这一节，我们要讨论计算机视觉中另外两大核心问题，目标检测与语义分割。

目标检测

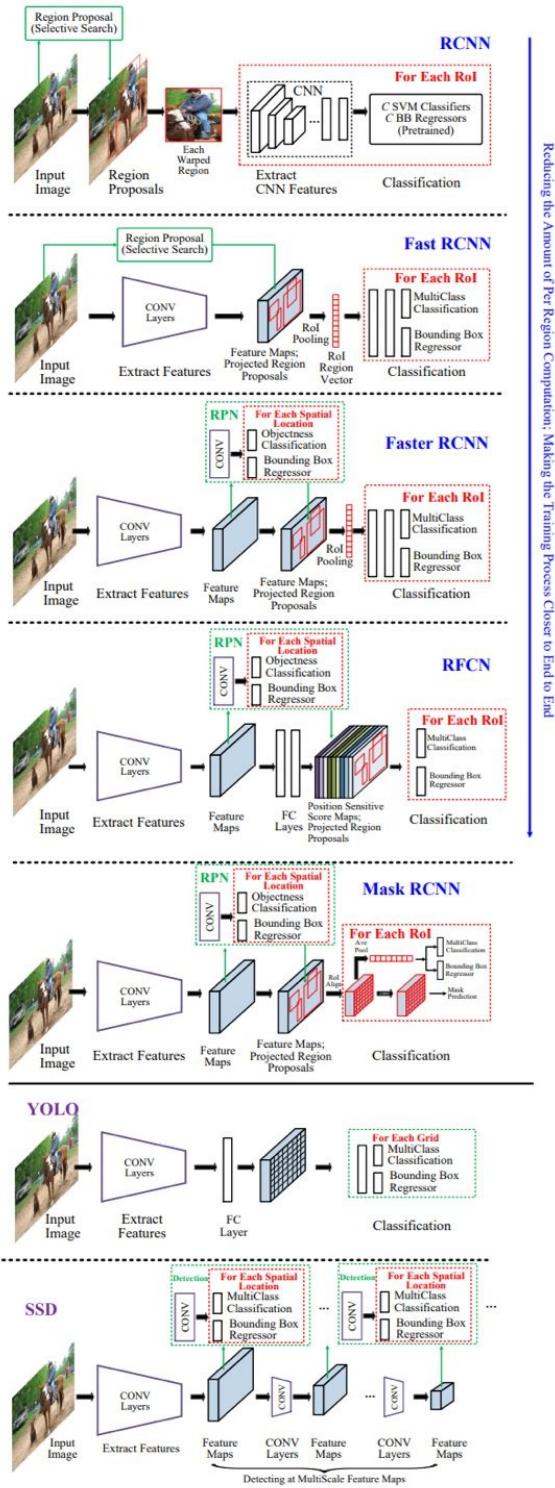
传统的目标检测流程

1. 区域选择(穷举策略:采用滑动窗口, 且设置不同的大小, 不同的长宽对图像进行遍历, 时间复杂度高)
2. 特征提取(SIFT, HOG等; 形态多样性, 光照变化多样性, 背景多样性使得特征鲁棒性差)
3. 分类器分类(主要有SVM, Adaboost等)

从R-CNN开始的基于深度学习的目标检测方法的发展历史。



Faster R-CNN, R-FCN和SSD是三种目前最优且应用最广泛的目标检测模型。其他流行的模型通常与这三者类似, 都依赖于深度CNN(如:ResNet, Inception等)来进行网络初始化, 且大部分遵循同样的proposal/分类管道。



基于区域提名的方法

R-CNN,SPP-net,Fast R-CNN,Faster R-CNN,R-FCN 最后总结一下各大算法的步骤：

RCNN

- 1.在图像中确定约1000-2000个候选框 (使用选择性搜索Selective Search)
- 2.每个候选框内图像块缩放至相同大小, 并输入到CNN内进行特征提取
- 3.对候选框中提取出的特征, 使用分类器判别是否属于一个特定类
- 4.对于属于某一类别的候选框, 用回归器进一步调整其位置

Fast R-CNN

- 1.在图像中确定约1000-2000个候选框 (使用选择性搜索Selective Search)
- 2.对整张图片输进CNN, 得到feature map
- 3.找到每个候选框在feature map上的映射patch, 将此patch作为每个候选框的卷积特征输入到SPP layer和之后的层
- 4.对候选框中提取出的特征, 使用分类器判别是否属于一个特定类
- 5.对于属于某一类别的候选框, 用回归器进一步调整其位置

Faster R-CNN

- 1.对整张图片输进CNN, 得到feature map
- 2.卷积特征输入到RPN, 得到候选框的特征信息
- 3.对候选框中提取出的特征, 使用分类器判别是否属于一个特定类
- 4.对于属于某一类别的候选框, 用回归器进一步调整其位置

R-CNN

R-CNN(Region-based Convolutional Neural Network)是深度学习在目标检测的开山之作。下面先介绍R-CNN。

R-CNN主要基于 R-CNN(Selective Search + CNN + SVM)

端对端的方法

YOLO(You Only Look Once),SSD(Single Shot MultiBox Detector)

语义分割

全卷积网络(FCN)

FCN, DeconvNet, SegNet, DilatedConvNet

CRF/MRF的使用

DeepLab, CRFasRNN, DPN

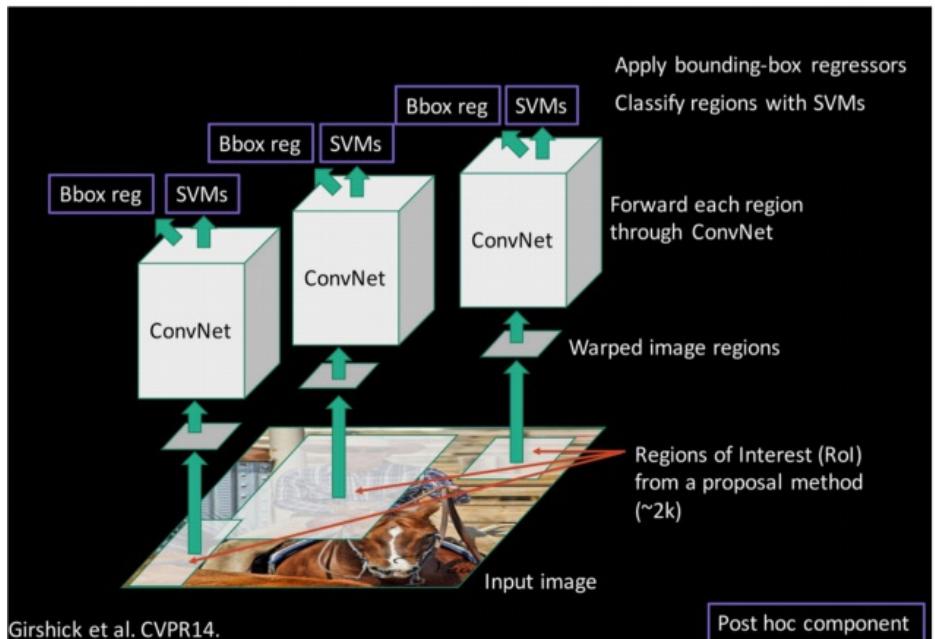
实例分割

Musk R-CNN

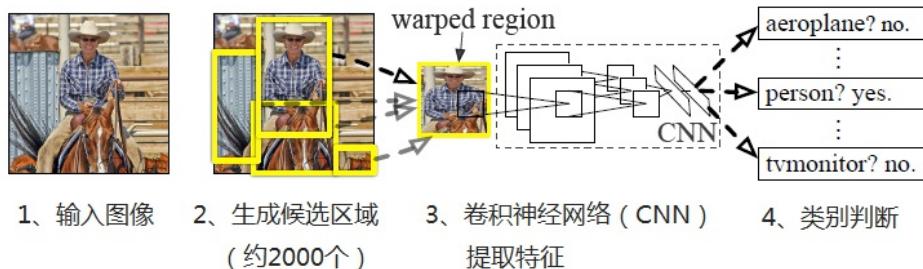
R-CNN

R-CNN(Region CNN, 区域卷积神经网络)可以说是利用深度学习进行目标检测的开山之作, 作者 Ross Girshick多次在PASCAL VOC的目标检测竞赛中折桂, 2010年更是带领团队获得了终身成就奖, 如今就职于Facebook的人工智能实验室(FAIR)¹。

R-CNN原理示意图



R-CNN算法的流程如下：



- 1、输入图像
- 2、每张图像生成1K~2K个候选区域
- 3、对每个候选区域，使用深度网络提取特征(AlexNet、VGG等CNN都可以)
- 4、将特征送入每一类的SVM分类器，判别是否属于该类
- 5、使用回归器精细修正候选框位置

下面分别展开来讲：

1、生成候选区域

使用Selective Search(选择性搜索)方法对一张图像生成约2000-3000个候选区域，其思路如下：

- (1) 使用一种过分割手段，将图像分割成小区域
- (2) 查看现有小区域，合并可能性最高的两个区域，重复直到整张图像合并成一个区域位置。优先合并以下区域：

- 颜色(颜色直方图)相近的
- 纹理(梯度直方图)相近的
- 合并后总面积小的
- 合并后，总面积在其BBOX中所占比例大的

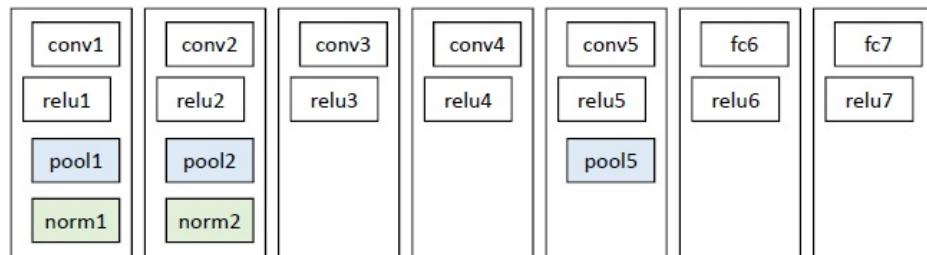
在合并时须保证合并操作的尺度较为均匀，避免一个大区域陆续“吃掉”其它小区域，保证合并后形状规则。

- (3) 输出所有曾经存在过的区域，即所谓候选区域

2、特征提取

使用深度网络提取特征之前，首先把候选区域归一化成同一尺寸 227×227 。

使用CNN模型进行训练，例如AlexNet，一般会略作简化，如下图：

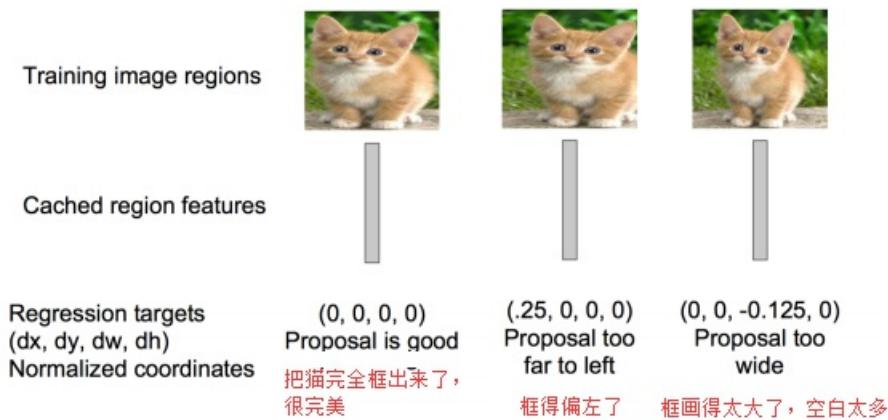


3、类别判断

对每一类目标，使用一个线性SVM二类分类器进行判别。输入为深度网络(如上图的AlexNet)输出的4096维特征，输出是否属于此类。

4、位置精修

目标检测的衡量标准是重叠面积:许多看似准确的检测结果,往往因为候选框不够准确,重叠面积很小,故需要一个位置精修步骤,对于每一个类,训练一个线性回归模型去判定这个框是否框得完美,如下图:



R-CNN将深度学习引入检测领域后,一举将PASCAL VOC上的检测率从35.1%提升到53.7%。

1. 大话目标检测经典模型(RCNN、Fast RCNN、Faster RCNN)
<https://my.oschina.net/u/876354/blog/1787921> ↵

Fast-RCNN

继2014年的R-CNN推出之后，Ross Girshick在2015年推出Fast R-CNN，构思精巧，流程更为紧凑，大幅提升了目标检测的速度。

Fast R-CNN主要解决R-CNN的以下问题

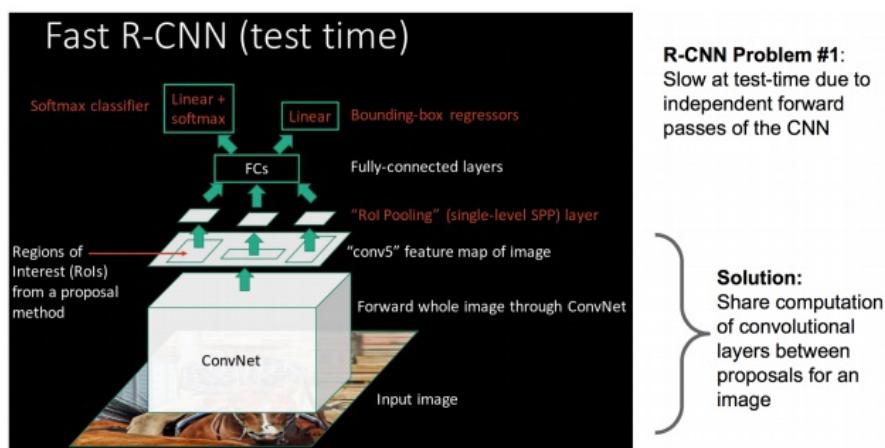
1. 训练，测试时速度慢

R-CNN的一张图像内候选框之间存在大量重叠，提取特征操作冗余。而Fast R-CNN将整张图像归一化后直接送入深度网络，紧接着送入从这幅图像上提取出的候选区域。这些候选区域的前几层特征不需要再重复计算。

2. 训练所需空间大

R-CNN中独立的分类器和回归器需要大量特征作为训练样本。Fast R-CNN把类别判断和位置精调统一用深度网络实现，不再需要额外存储。

Fast R-CNN原理示意图



R-CNN虽然不再像传统方法那样穷举，但R-CNN流程的第一步中对原始图片通过Selective Search提取的候选框region proposal多达2000个左右，而这2000个候选框每个框都需要进行CNN提特征+SVM分类，计算量很大，导致R-CNN检测速度很慢，一张图都需要47s。

有没有方法提速呢？答案是有的，这2000个region proposal不都是图像的一部分吗，那么我们完全可以对图像提一次卷积层特征，然后只需要将region proposal在原图的位置映射到卷积层特征图上，这样对于一张图像我们只需要提一次卷积层特征，然后将每个region proposal的卷积层特征输入到全连接层做后续操作。

但现在的问题是每个region proposal的尺度不一样，而全连接层输入必须是固定的长度，所以直接这样输入全连接层肯定是不行的。SPP Net恰好可以解决这个问题。

SPP Net¹

SPP: Spatial Pyramid Pooling(空间金字塔池化)

众所周知, CNN一般都含有卷积部分和全连接部分, 其中, 卷积层不需要固定尺寸的图像, 而全连接层是需要固定大小的输入。

所以当全连接层面对各种尺寸的输入数据时, 就需要对输入数据进行crop(crop就是从一个大图扣出网络输入大小的patch, 比如227×227), 或warp(把一个边界框bounding box的内容resize成227×227)等一系列操作以统一图片的尺寸大小, 比如224_224(ImageNet)、32_32(LenNet)、96*96等。

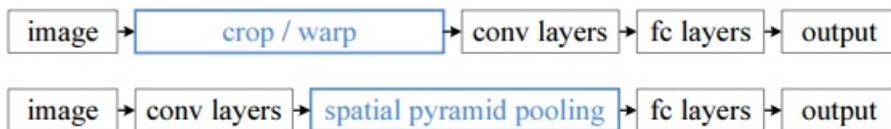


所以才如你在上文中看到的, 在R-CNN中, “因为取出的区域大小各自不同, 所以需要将每个Region Proposal缩放(warp)成统一的227x227的大小并输入到CNN”。

但warp/crop这种预处理, 导致的问题要么被拉伸变形、要么物体不全, 限制了识别精确度。没太明白? 说句人话就是, 一张16:9比例的图片你硬是要Resize成1:1的图片, 你说图片失真不?

SPP Net的作者Kaiming He等人逆向思考, 既然由于全连接FC层的存在, 普通的CNN需要通过固定输入图片的大小来使得全连接层的输入固定。那借鉴卷积层可以适应任何尺寸, 为何不能在卷积层的最后加入某种结构, 使得后面全连接层得到的输入变成固定的呢?

这个“化腐朽为神奇”的结构就是spatial pyramid pooling layer。下图便是R-CNN和SPP Net检测流程的比较:



它的特点有两个:

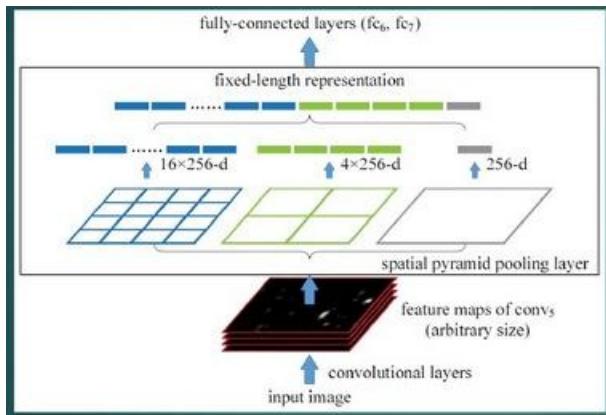
1.结合空间金字塔方法实现CNNs的多尺度输入。

SPP Net的第一个贡献就是在最后一个卷积层后, 接入了金字塔池化层, 保证传到下一层全连接层的输入固定。

换句话说, 在普通的CNN机构中, 输入图像的尺寸往往是固定的(比如224*224像素), 输出则是一个固定维数的向量。SPP Net在普通的CNN结构中加入了ROI池化层(ROI Pooling), 使得网络的输入图像可以是任意尺寸的, 输出则不变, 同样是一个固定维数的向量。

简言之, CNN原本只能固定输入、固定输出, CNN加上SSP之后, 便能任意输入、固定输出。神奇吧?

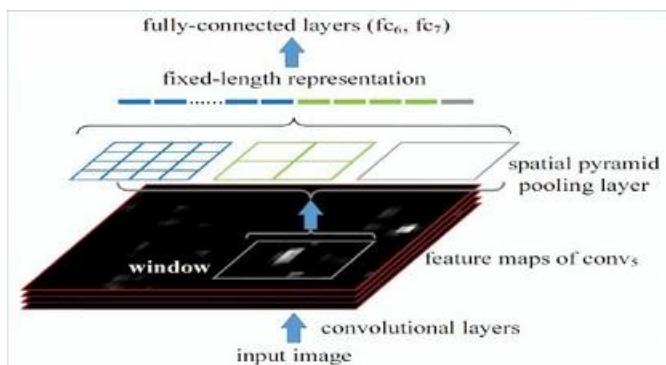
ROI池化层一般跟在卷积层后面，此时网络的输入可以是任意尺度的，在SPP layer中每一个pooling的filter会根据输入调整大小，而SPP的输出则是固定维数的向量，然后给到全连接FC层。



2. 只对原图提取一次卷积特征

在R-CNN中，每个候选框先resize到统一大小，然后分别作为CNN的输入，这样是很低效的。而SPP Net根据这个缺点做了优化：只对原图进行一次卷积计算，便得到整张图的卷积特征feature map，然后找到每个候选框在feature map上的映射patch，将此patch作为每个候选框的卷积特征输入到SPP layer和之后的层，完成特征提取工作。

如此这般，R-CNN要对每个区域计算卷积，而SPPNet只需要计算一次卷积，从而节省了大量的计算时间，比R-CNN有一百倍左右的提速。

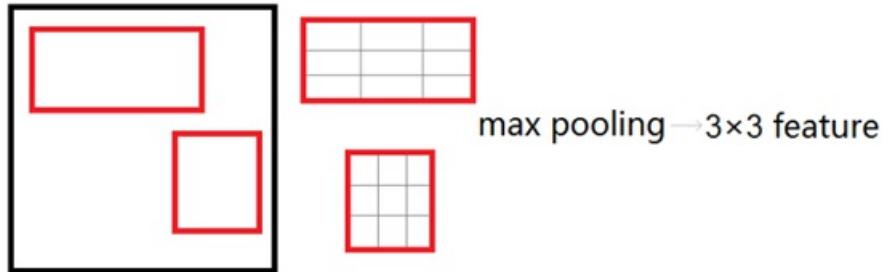


1. 在特征提取阶段²

通过CNN（如AlexNet）中的conv、pooling、relu等操作都不需要固定大小尺寸的输入，因此，在原始图片上执行这些操作后，输入图片尺寸不同将会导致得到的feature map（特征图）尺寸也不同，这样就不能直接接到一个全连接层进行分类。

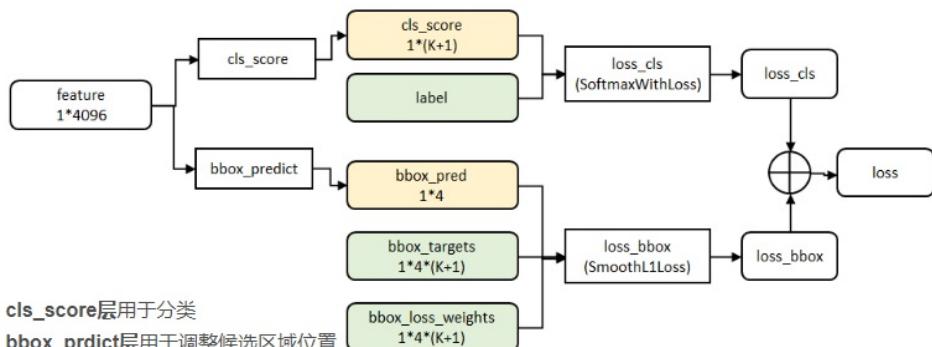
ROI Pooling

在Fast R-CNN中，作者提出了一个叫做ROI Pooling的网络层，这个网络层可以把不同大小的输入映射到一个固定尺度的特征向量。ROI Pooling层将每个候选区域均匀分成 $M \times N$ 块，对每块进行max pooling。将特征图上大小不一的候选区域转变为大小统一的数据，送入下一层。这样虽然输入的图片尺寸不同，得到的feature map（特征图）尺寸也不同，但是可以加入这个神奇的ROI Pooling层，对每个region都提取一个固定维度的特征表示，就可再通过正常的softmax进行类型识别。



2、在分类回归阶段

在R-CNN中，先生成候选框，然后再通过CNN提取特征，之后再用SVM分类，最后再做回归得到具体位置(bbox regression)。而在Fast R-CNN中，作者巧妙的把最后的bbox regression也放在了神经网络内部，与区域分类合并成为了一个multi-task模型，如下图所示：



实验表明，这两个任务能够共享卷积特征，并且相互促进。

Fast R-CNN很重要的一个贡献是成功地让人们看到了Region Proposal+CNN（候选区域+卷积神经网络）这一框架实时检测的希望，原来多类检测真的可以在保证准确率的同时提升处理速度。

总结

Fast-RCNN的优势：

1. 加入ROI Pooling层使得每个提名区域的输出大小一致
2. 把分类任务与位置回归任务放在一个网络里面，同时训练，损失函数是分类损失函数加上回归损失函数。加快了训练过程。

Fast R-CNN和R-CNN相比，训练时间从84小时减少到9.5小时，测试时间从47秒减少到0.32

秒，并且在PASCAL VOC 2007上测试的准确率相差无几，约在66%-67%之间。

	R-CNN	Fast R-CNN
Faster!	Training Time: (Speedup)	84 hours 1x
	Test time per image (Speedup)	47 seconds 1x
FASTER!		9.5 hours 8.8x
		0.32 seconds 146x

1. 一文读懂目标检测：R-CNN、Fast R-CNN、Faster R-CNN、YOLO、SSD

https://blog.csdn.net/v_JULY_v/article/details/80170182 ↵

2. 大话目标检测经典模型(RCNN、Fast RCNN、Faster RCNN)

<https://my.oschina.net/u/876354/blog/1787921> ↵

Faster-RCNN

Faster R-CNN更快更强

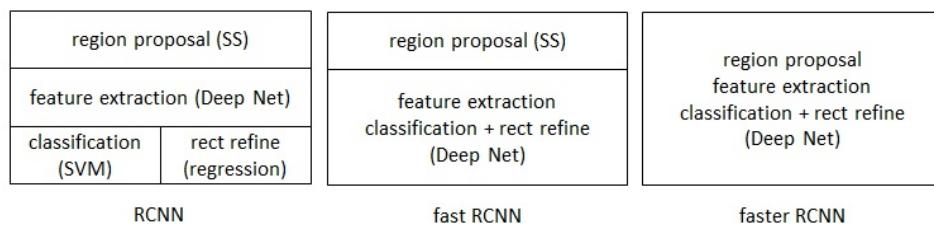
继2014年推出R-CNN, 2015年推出Fast R-CNN之后, 目标检测界的领军人物Ross Girshick团队在2015年又推出一力作:Faster R-CNN, 使简单网络目标检测速度达到17fps, 在PASCAL VOC上准确率为59.9%, 复杂网络达到5fps, 准确率78.8%。

在Fast R-CNN还存在着瓶颈问题:Selective Search(选择性搜索)。要找出所有的候选框, 这个也非常耗时。那我们有没有一个更加高效的方法来求出这些候选框呢?

在Faster R-CNN中加入一个提取边缘的神经网络, 也就说找候选框的工作也交给神经网络來做了。

这样, 目标检测的四个基本步骤(候选区域生成, 特征提取, 分类, 位置精修)终于被统一到一个深度网络框架之内。

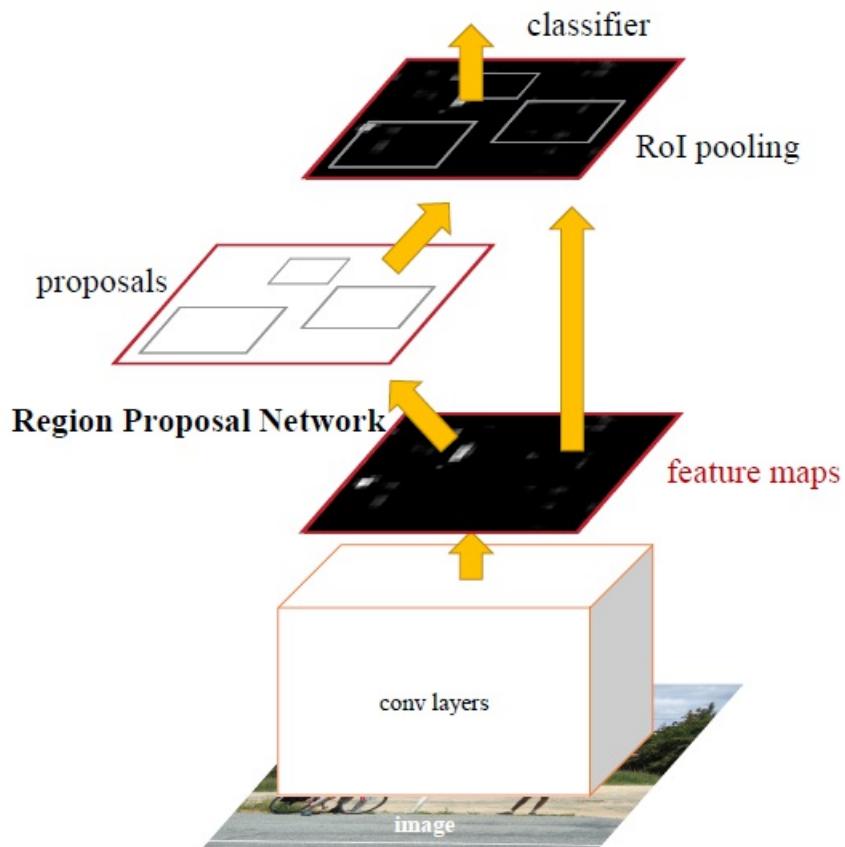
如下图所示:



Faster R-CNN可以简单地看成是“区域生成网络+Fast R-CNN”的模型, 用区域生成网络(Region Proposal Network, 简称RPN)来代替Fast R-CNN中的Selective Search(选择性搜索)方法。

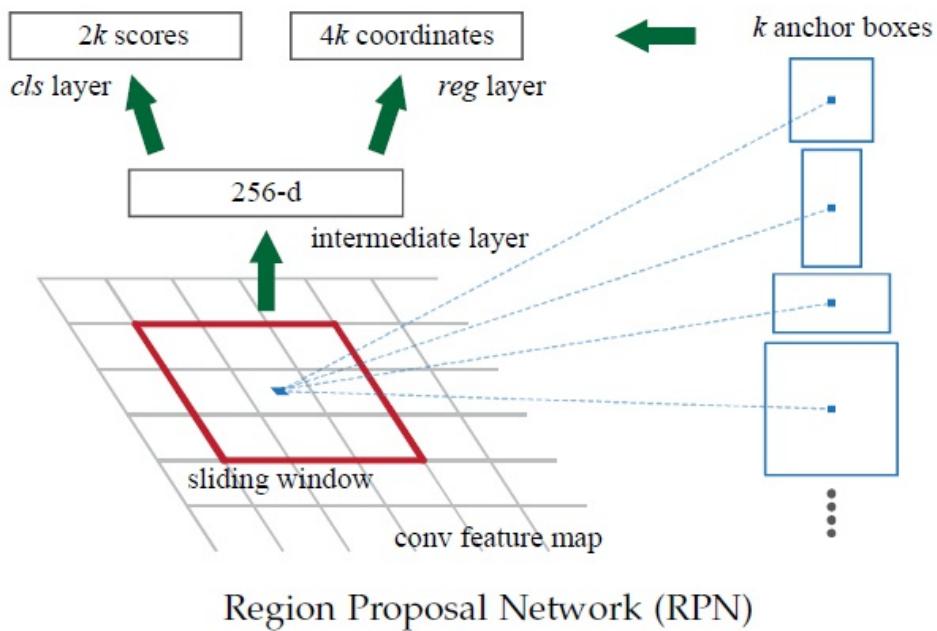
如下图:

Faster R-CNN:



RPN

RPN如下图:



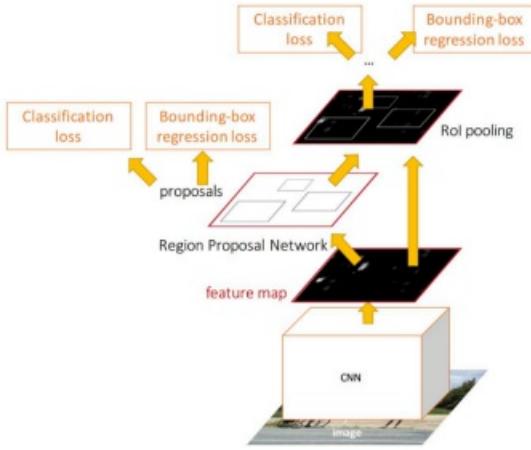
RPN的工作步骤如下：

- 在feature map(特征图)上滑动窗口
- 建一个神经网络用于物体分类+框位置的回归
- 滑动窗口的位置提供了物体的大体位置信息
- 框的回归提供了框更精确的位置

联合四个损失函数的网络¹

一种网络，四个损失函数；

- RPN classification(anchor good/bad)
- RPN regression(anchor->proposal)
- Fast R-CNN classification(over classes)
- Fast R-CNN regression(proposal ->box)



总结

Faster R-CNN设计了提取候选区域的网络RPN，代替了费时的Selective Search(选择性搜索)，使得检测速度大幅提升，下表对比了R-CNN、Fast R-CNN、Faster R-CNN的检测速度：

	R-CNN	Fast R-CNN	Faster R-CNN
Test time per image (with proposals)	50 seconds	2 seconds	0.2 seconds
(Speedup)	1x	25x	250x
mAP (VOC 2007)	66.0	66.9	66.9

¹. 一文读懂目标检测：R-CNN、Fast R-CNN、Faster R-CNN、YOLO、SSD
https://blog.csdn.net/v_JULY_v/article/details/80170182 ↵

YOLO

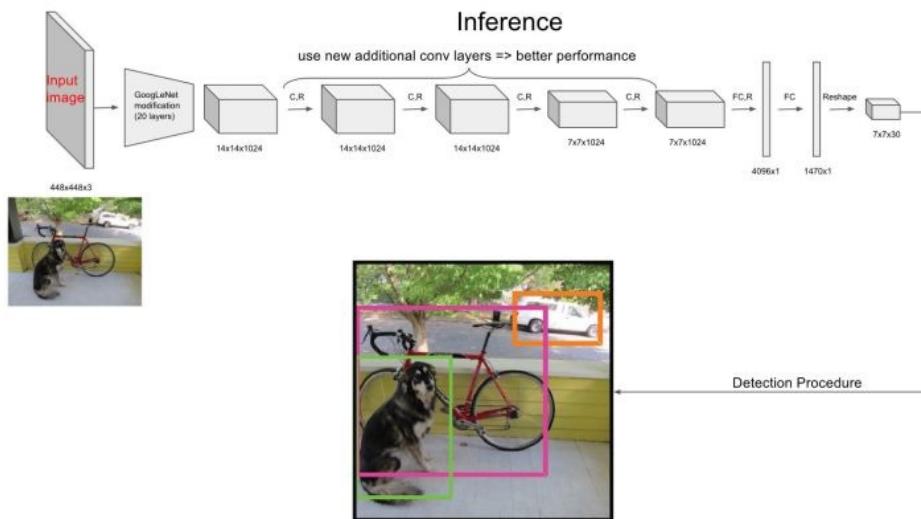
目标检测算法可以分为two stage与one stage两类算法：

two stage就是先产生region proposal, 再在region proposal上做分类与回归, 典型的就是RCNN的算法, 如: R-CNN, Fast R-CNN, Faster R-CNN, 它的优点是准确率高, 但是速度慢。

one stage算法仅仅使用一个CNN网络直接预测不同目标的类别与位置, 典型的算法是YOLO, SSD。它的准确度要低一些, 但是它的速度快。

目标检测的一个实际应用场景就是无人驾驶, 如果能够在无人车上装载一个有效的目标检测系统, 那么无人车将和人一样有了眼睛, 可以快速地检测出前面的行人与车辆, 从而作出实时决策¹。

YOLO的原理图



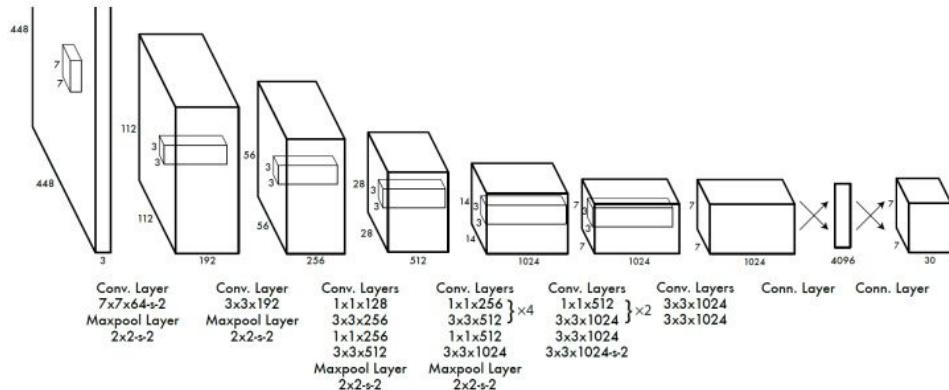
原理解析

整体上, YOLO采用的是一个单独的CNN实现end-to-end的目标检测。通过输入一张448x448的图片到CNN, 先经过GoogLeNet的前二十个卷积层, 再通过一些卷积层与两个全连接层, 最终输出的是7x7x30的张量。7x7表示49个区域, 30=20+2x5, 表示二十个类别的概率, 2表示两个边界框, 5表示每个边界框的四个坐标以及一个置信度。

网络设计

Yolo采用卷积网络来提取特征, 然后使用全连接层来得到预测值。网络结构参考GooLeNet模型, 包含24个卷积层和2个全连接层, 如图8所示。对于卷积层, 主要使用1x1卷积来做channel reduction, 然后紧跟3x3卷积。对于卷积层和全连接层, 采用Leaky ReLU激活函数: $\max(x, 0.1x)$ 。

但是最后一层却采用线性激活函数。



输出参数解析

每个单元格需要预测 $(B \times 5 + C)$ 个值。如果将输入图片划分为 $S \times S$ 网格，那么最终预测值为 $S \times S \times (B \times 5 + C)$ 大小的张量。整个模型的预测值结构如下图所示。对于PASCAL VOC数据，其共有20个类别，如果使用 $S=7, B=2$ ，那么最终的预测结果就是 $7 \times 7 \times 30$ 大小的张量。在下面的网络结构中我们会详细讲述每个单元格的预测值的分布位置。

损失函数

YOLO算法将目标检测看成回归问题，所以采用的是均方差损失函数。但是对不同的部分采用了不同的权重值。首先区分定位误差和分类误差。

对于定位误差，即边界框坐标预测误差，采用较大的权重 $\lambda_{coord} = 5$ 。

然后其区分不包含目标的边界框与含有目标的边界框的置信度，对于前者，采用较小的权重值

$\lambda_{noobj} = 0.5$ 。其它权重值均设为1。然后采用均方误差，其同等对待大小不同的边界框，但是实际上较小的边界框的坐标误差应该要比较大的边界框要更敏感。为了保证这一点，将网络的边界框的

宽与高预测改为对其平方根的预测，即预测值变为 $(x, y, \sqrt{w}, \sqrt{h})$ 。

最终的损失函数：

$$\begin{aligned}
& \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
& + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
& + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2 \\
& + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2 \\
& + \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

上面两行是坐标误差，中间两行是IOU误差，最下面一行是分类误差。

其中第一项是边界框中心坐标的误差项， $\mathbb{1}_{ij}^{\text{obj}}$ 指的是第 i 个单元格存在目标，且该单元格中的第 j 个边界框负责预测该目标。第二项是边界框的高与宽的误差项。第三项是包含目标的边界框的置信度误差项。第四项是不包含目标的边界框的置信度误差项。而最后一项是包含目标的单元格的分类误差项， $\mathbb{1}_i^{\text{obj}}$ 指的是第 i 个单元格存在目标。这里特别说一下置信度的target值 C_i ，如果是不存在目标，此时由于 $\text{Pr}(\text{object})=0$ ，那么 $C_i = 0$ 。如果存在目标， $\text{Pr}(\text{object})=1$ ，此时需要确定 $\text{IOU}_{\text{pred}}^{\text{truth}}$ ，当然你希望最好的话，可以将IOU取1，这样 $C_i = 1$ ，但是在YOLO实现中，使用了一个控制参数`rescore`(默认为1)，当其为1时，IOU不是设置为1，而是计算`truth`和`pred`之间的真实IOU。不过很多复现YOLO的项目还是取 $C_i = 1$ ，这个差异应该不会太影响结果吧。

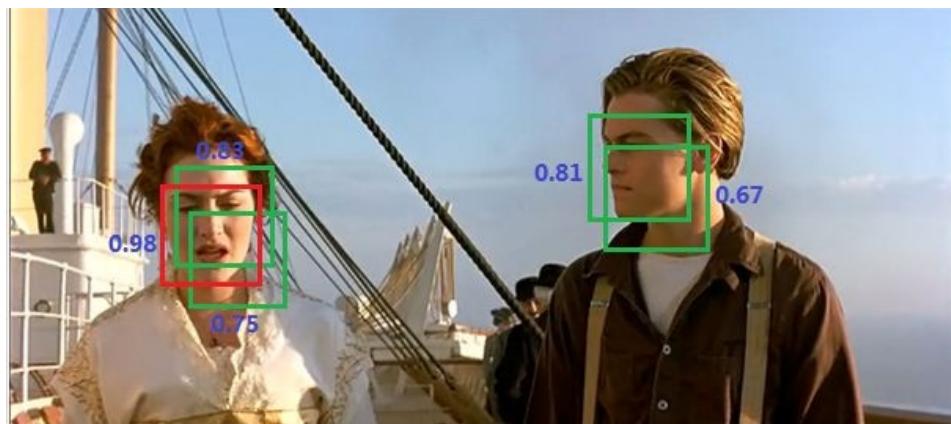
输出参数解析

具体来说，Yolo的CNN网络将输入的图片分割成 $S \times S$ 网格，然后每个单元格负责去检测那些中心点落在该格子内的目标，如图6所示，可以看到狗这个目标的中心落在左下角一个单元格内，那么该单元格负责预测这个狗。每个单元格会预测 B 个边界框(bounding box)以及边界框的置信度(confidence score)。所谓置信度其实包含两个方面，一是这个边界框含有目标的可能性大小，二是这个边界框的准确度。前者记为 $\text{Pr}(\text{object})$ ，当该边界框是背景时(即不包含目标)，此时 $\text{Pr}(\text{object})=0$ 。而当该边界框包含目标时， $\text{Pr}(\text{object})=1$ 。边界框的准确度可以用预测框与实际框(ground truth)的IOU(intersection over union, 交并比)来表征，记为 $\text{IOU}_{\text{pred}}^{\text{truth}}$ 。因此置信度可以

定义为 $Pr(object) * IOU_{pred}^{truth}$ 。很多人可能将Yolo的置信度看成边界框是否含有目标的概率，但是其实它是两个因子的乘积，预测框的准确度也反映在里面。边界框的大小与位置可以用4个值来表征： (x, y, w, h) ，其中 (x, y) 是边界框的中心坐标，而 w 和 h 是边界框的宽与高。还有一点要注意，中心坐标的预测值 (x, y) 是相对于每个单元格左上角坐标点的偏移值，并且单位是相对于单元格大小的，单元格的坐标定义如图6所示。而边界框的 w 和 h 预测值是相对于整个图片的宽与高的比例，这样理论上4个元素的大小应该在 $[0, 1]$ 范围。这样，每个边界框的预测值实际上包含5个元素： (x, y, w, h, c) ，其中前4个表征边界框的大小与位置，而最后一个值是置信度。

网络预测

在说明Yolo算法的预测过程之前，这里先介绍一下非极大值抑制算法（non maximum suppression, NMS），这个算法不单单是针对Yolo算法的，而是所有的检测算法中都会用到。NMS算法主要解决的是一个目标被多次检测的问题，如图11中人脸检测，可以看到人脸被多次检测，但是其实我们希望最后仅仅输出其中一个最好的预测框，比如对于美女，只想要红色那个检测结果。那么可以采用NMS算法来实现这样的效果：首先从所有的检测框中找到置信度最大的那个框，然后挨个计算其与剩余框的IOU，如果其值大于一定阈值（重合度过高），那么就将该框剔除；然后对剩余的检测框重复上述过程，直到处理完所有的检测框。Yolo预测过程也需要用到NMS算法。



下面就来分析Yolo的预测过程，这里我们不考虑batch，认为只是预测一张输入图片。根据前面的分析，最终的网络输出是 $7 \times 7 \times 30$ ，但是我们可以将其分割成三个部分：类别概率部分为 $[7, 7, 20]$ ，置信度部分为 $[7, 7, 2]$ ，而边界框部分为 $[7, 7, 2, 4]$ （对于这部分不要忘记根据原始图片计算出其真实值）。然后将前两项相乘（矩阵 $[7, 7, 20]$ 乘以 $[7, 7, 2]$ 可以各补一个维度来完成 $[7, 7, 20] \times [7, 7, 2, 1]$ ）可以得到类别置信度值为 $[7, 7, 2, 20]$ ，这里总共预测了 $7 \times 7 \times 2 = 98$ 个边界框。

所有的准备数据已经得到了，那么我们先说第一种策略来得到检测框的结果，我认为这是最正常与自然的处理。首先，对于每个预测框根据类别置信度选取置信度最大的那个类别作为其预测标签，经过这层处理我们得到各个预测框的预测类别及对应的置信度值，其大小都是 $[7, 7, 2]$ 。一般情况下，会设置置信度阈值，就是将置信度小于该阈值的box过滤掉，所以经过这层处理，剩余的是置信度比较高的预测框。最后再对这些预测框使用NMS算法，最后留下来的才是检测结果。一个值得注

意的点是NMS是对所有预测框一视同仁，还是区分每个类别，分别使用NMS。Ng在deeplearning.ai中讲应该区分每个类别分别使用NMS，但是看了很多实现，其实还是同等对待所有的框，我觉得可能是不同类别的目标出现在相同位置这种概率很低吧。

小结

YOLO将目标检测任务转换成一个回归问题，大大加快了检测的速度，使得YOLO可以每秒处理45张图像。而且由于每个网络预测目标窗口时使用的是全图信息，使得false positive比例大幅降低（充分的上下文信息）。

但是YOLO也存在问题：没有了Region Proposal机制，只使用7*7的网格回归会使得目标不能非常精准的定位，这也导致了YOLO的检测精度并不是很高。

2. 目标检测|YOLO原理与实现 <https://zhuanlan.zhihu.com/p/32525231> ↵

3. 一文读懂目标检测：R-CNN、Fast R-CNN、Faster R-CNN、YOLO、SSD

https://blog.csdn.net/v_JULY_v/article/details/80170182 ↵

SSD

SSD: Single Shot MultiBox Detector

YOLO使用整图特征在 7×7 的粗糙网格内回归对目标的定位并不是很精准。那是不是可以结合Region Proposal的思想实现精准一些的定位？SSD结合YOLO的回归思想以及Faster R-CNN的anchor机制做到了这点。

步骤

SSD仅需要一张输入图像和训练所需要的每个目标的 ground truth, 而输出为位置偏移量、置信度和分类概率。首先通过对输入图像的一系列卷积操作生成不同大小的特征图, 比如上图中的 8×8 和 4×4 大小特征图。然后对这些每个特征图执行 3×3 的卷积核来评估此前生成的默认边界框, 这种默认边界框可理解为先验框, 类似于此前 Faster R-CNN 的 anchor boxes。对这些边界框同时执行预测, 主要预测两个量: 边界框的偏移量和目标物体的分类概率值, 最后使用非极大值抑制来选取高于阈值的边界框。

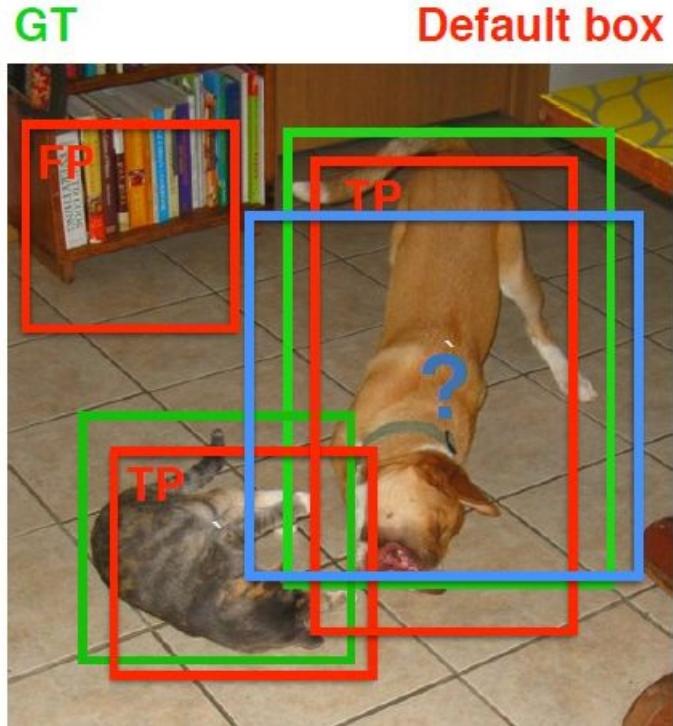
所以, SSD 的分步步骤如下:

1. 对输入图像执行一系列卷积生成不同大小的特征图
2. 对每个特征图都执行 3×3 卷积来评估默认边界框
3. 对每个边界框预测偏移量和分类概率
4. 执行NMS非极大值抑制

先验框匹配¹

在训练过程中, 首先要确定训练图片中的ground truth(真实目标)与哪个先验框来进行匹配, 与之匹配的先验框所对应的边界框将负责预测它。在Yolo中, ground truth的中心落在哪个单元格, 该单元格中与其IOU最大的边界框负责预测它。但是在SSD中却完全不一样, SSD的先验框与ground truth的匹配原则主要有两点。首先, 对于图片中每个ground truth, 找到与其IOU最大的先验框, 该先验框与其匹配, 这样, 可以保证每个ground truth一定与某个先验框匹配。通常称与ground truth匹配的先验框为正样本(其实应该是先验框对应的预测box, 不过由于是一一对应的就这样称呼了), 反之, 若一个先验框没有与任何ground truth进行匹配, 那么该先验框只能与背景匹配, 就是负样本。一个图片中ground truth是非常少的, 而先验框却很多, 如果仅按第一个原则匹配, 很多先验框会是负样本, 正负样本极其不平衡, 所以需要第二个原则。第二个原则是:对于剩余的未匹配先验框, 若某个ground truth的 IOU 大于某个阈值(一般是0.5), 那么该先验框也与这个ground truth进行匹配。这意味着某个ground truth可能与多个先验框匹配, 这是可以的。但是反过来却不可以, 因为一个先验框只能匹配一个ground truth, 如果多个ground truth与某个先验框 IOU 大于阈值, 那么先验框只与IOU最大的那个先验框进行匹配。第二个原则一定在第一个原则之后进行, 仔细考虑一下这种情况, 如果某个ground truth所对应最大 IOU 小于阈值, 并且所匹配的先验框却与另外一个ground truth的 IOU 大于阈值, 那么该先验框应该匹配谁, 答案应该是前者, 首先要确保某个ground truth一定有一个先验框与之匹配。但是, 这种情况我觉得基本上是不存在的。由于先验

框很多，某个ground truth的最大 IOU 肯定大于阈值，所以可能只实施第二个原则既可以了，这里的TensorFlow版本就是只实施了第二个原则，但是这里的Pytorch两个原则都实施了。图8为一个匹配示意图，其中绿色的GT是ground truth，红色为先验框，FP表示负样本，TP表示正样本。



尽管一个ground truth可以与多个先验框匹配，但是ground truth相对先验框还是太少了，所以负样本相对正样本会很多。为了保证正负样本尽量平衡，SSD采用了hard negative mining，就是对负样本进行抽样，抽样时按照置信度误差(预测背景的置信度越小，误差越大)进行降序排列，选取误差的较大的top-k作为训练的负样本，以保证正负样本比例接近1:3。

SSD网络结构

SSD采用VGG16作为基础模型，然后在VGG16的基础上新增了卷积层来获得更多的特征图以用于检测。SSD的网络结构如图5所示。上面是SSD模型，下面是Yolo模型，可以明显看到SSD利用了多尺度的特征图做检测。模型的输入图片大小是 300×300 (还可以是 512×512)，其与前者网络结构

没有差别，只是最后新增一个卷积层，本文不再讨论)。

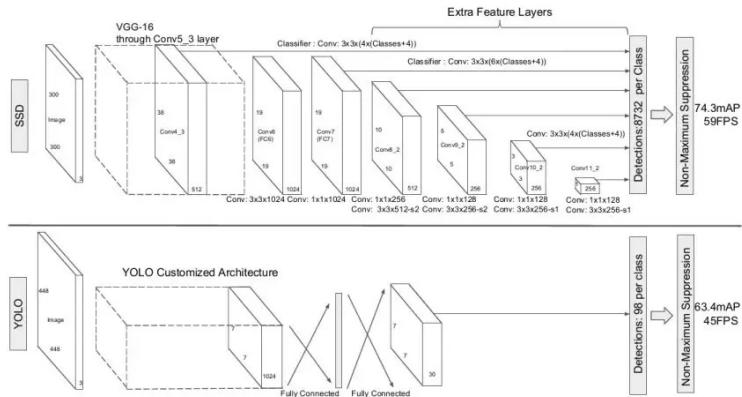


Fig. 2: A comparison between two single shot detection models: SSD and YOLO [5]. Our SSD model adds several feature layers to the end of a base network, which predict the offsets to default boxes of different scales and aspect ratios and their associated confidences. SSD with a 300×300 input size significantly outperforms its 448×448 YOLO counterpart in accuracy on VOC2007 test while also improving the speed.

损失函数

训练样本确定了，然后就是损失函数了。损失函数定义为位置误差 (localization loss, loc) 与置信度误差 (confidence loss, conf) 的加权和¹：

$$L(x, c, l, g) = \frac{1}{N} (L_{conf}(x, c) + \alpha L_{loc}(x, c, g))$$

权重系数 α 通过交叉验证设置为1。

其中 N 是先验框的正样本数量。这里 $x_{ij}^p \in \{1, 0\}$ 为一个指示参数，当 $x_{ij}^p = 1$ 时表示第 i 个先验框与第 j 个 ground truth 匹配，并且 ground truth 的类别为 p。c 为类别置信度预测值。l 为先验框的所对应边界框的位置预测值，而 g 是 ground truth 的位置参数。对于位置误差，其采用 Smooth L1 loss，定义如下：

$$L_{loc}(x, l, g) = \sum_{i \in Pos} \sum_{m \in \{cx, cy, w, h\}} x_{ij}^k \text{smooth}_{L1}(l_i^m - \hat{g}_j^m)$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx}) / d_i^w \quad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy}) / d_i^h$$

$$\hat{g}_j^w = \log \left(\frac{g_j^w}{d_i^w} \right) \quad \hat{g}_j^h = \log \left(\frac{g_j^h}{d_i^h} \right)$$

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

对于置信度偏差采用softmax:

$$L_{conf}(x, c) = - \sum_{i \in Pos}^{N} x_{ij}^p \log(\hat{c}_i^p) - \sum_{i \in Neg} \log(\hat{c}_i^0) \quad \text{where} \quad \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)}$$

数据扩增

采用数据扩增(Data Augmentation)可以提升SSD的性能, 主要采用的技术有水平翻转(horizontal flip), 随机裁剪加颜色扭曲(random crop & color distortion), 随机采集块域(Randomly sample a patch)(获取小目标训练样本)

预测过程

预测过程比较简单, 对于每个预测框, 首先根据类别置信度确定其类别(置信度最大者)与置信度值, 并过滤掉属于背景的预测框。然后根据置信度阈值(如0.5)过滤掉阈值较低的预测框。对于留下的预测框进行解码, 根据先验框得到其真实的位置参数(解码后一般还需要做clip, 防止预测框位置超出图片)。解码之后, 一般需要根据置信度进行降序排列, 然后仅保留top-k(如400)个预测框。最后就是进行NMS算法, 过滤掉那些重叠度较大的预测框。最后剩余的预测框就是检测结果了。

结果

SSD与其它检测算法的对比结果(在VOC2007数据集)如表所示, 基本可以看到, SSD与Faster R-CNN有同样的准确度, 并且与Yolo具有同样较快地检测速度。

Method	mAP	FPS	batch size	# Boxes	Input resolution
Faster R-CNN (VGG16)	73.2	7	1	~ 6000	~ 1000 × 600
Fast YOLO	52.7	155	1	98	448 × 448
YOLO (VGG16)	66.4	21	1	98	448 × 448
SSD300	74.3	46	1	8732	300 × 300
SSD512	76.8	19	1	24564	512 × 512
SSD300	74.3	59	8	8732	300 × 300
SSD512	76.8	22	8	24564	512 × 512

1. 目标检测|SSD原理与实现 <https://zhuanlan.zhihu.com/p/33544892> ↵

2. 一文读懂目标检测:R-CNN、Fast R-CNN、Faster R-CNN、YOLO、SSD
https://blog.csdn.net/v_JULY_v/article/details/80170182 ↵

无人驾驶

无人驾驶的主要内容：

1. 传感：通过雷达，激光，摄像头，GPS以及各种传感器来收集数据
2. 感知：通过传感收集的数据进行处理
 - i. 定位
 - ii. 物体识别与跟踪
3. 决策：基于数据做出决策
 - i. 行为预测
 - ii. 路径规划
 - iii. 避障

感知

物体识别

无人驾驶，首先就是定位问题，要知道自己在哪儿，让自己处于安全的区域内，因此，首先就是需要检测马路上的斑马线。

线检测

Hough变换

直线检测最经典的方法就是Hough变换，Hough变换在图像处理那一章节已经介绍了，因此这不介绍原理了。主要介绍OpenCV中的HoughLinesP函数：

C++: void HoughLinesP(InputArray image, OutputArray lines, double rho, double theta, int threshold, double minLineLength=0, double maxLineGap=0)

Parameters:

image – 8-bit, single-channel binary source image. The image may be modified by the function.

lines – Output vector of lines. Each line is represented by a 4-element vector (x_1, y_1, x_2, y_2) ,

where (x_1, y_1) and (x_2, y_2) are the ending points of each detected line segment.

rho – Distance resolution of the accumulator in pixels.

theta – Angle resolution of the accumulator in radians.

threshold – Accumulator threshold parameter. Only those lines are returned that get enough votes ($> \text{threshold}$).

minLineLength – Minimum line length. Line segments shorter than that are rejected.

maxLineGap – Maximum allowed gap between points on the same line to link them.

例子：

```
rho = 2 # distance resolution in pixels of the Hough grid
```

```

theta = np.pi / 180 # angular resolution in radians of the Hough grid
threshold = 12 # minimum number of votes (intersections in Hough grid cell)
min_line_length = 200 # minimum number of pixels making up a line
max_line_gap = 200 # maximum gap in pixels between connectable line segments

```

Canny edge detector

C++: void Canny(InputArray image, OutputArray edges, double threshold1, double threshold2, int apertureSize=3, bool L2gradient=false)

Parameters:

image – single-channel 8-bit input image.

edges – output edge map; it has the same size and type as image.

threshold1 – first threshold for the hysteresis procedure.

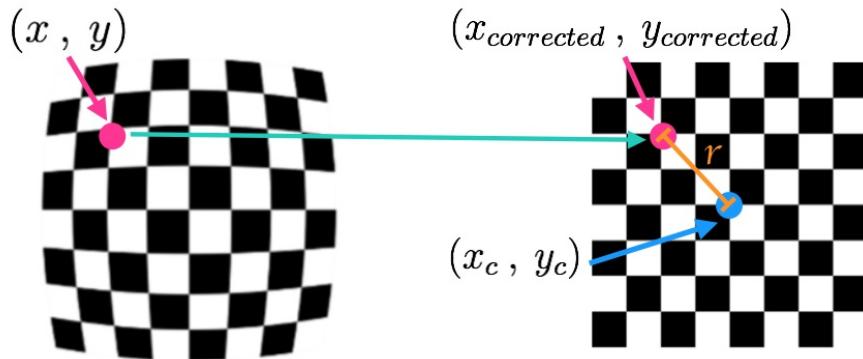
threshold2 – second threshold for the hysteresis procedure.

apertureSize – aperture size for the Sobel() operator.

L2gradient – a flag, indicating whether a more accurate $L_2\text{norm} = \sqrt{(dI/dx)^2 + (dI/dy)^2}$ should be used to calculate the image gradient magnitude (L2gradient=true), or whether the default $L_1\text{norm} = |dI/dx| + |dI/dy|$ is enough (L2gradient=false).

Distortion 与 Calibrating Camera

To ensure that the geometrical shape of objects is represented consistently, no matter where they appear in an image.



(x_c, y_c) 是扭曲中心, r 是原始图像中一点到扭曲中心的距离。

radial distortion

径向扭曲, 具有旋转不变性, 因此对x,y的修正是一样的。

$$x_{distored} = x_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

$$y_{distored} = y_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

tangential distortion

$$x_{corrected} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y + [2p_2xy + p_1(r^2 + 2y^2)]$$

calibrateCamera

To computer the transformation between 3D object points in the world and 2D image points

我们一般使用棋盘模型来的得到这些扭曲系数(k_1, k_2, k_3, p_1, p_2)

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size, None, None),
其中mtx is camera matrix, mtx, dist is distortion coefficients. 具体而言, imgpoints是真实的扭曲的图像中的点的3D坐标, objpoints是没有扭曲的棋盘坐标, 是2D的, 通过这两者之间, 可以建立一个映射关系, 返回ret, mtx, dist, rvecs, tvecs这些参数。

其中imgpoints可以通过OpenCV的函数来搜索到扭曲图像中的角点, ret, corners = cv2.findChessboardCorners(gray, (8,6), None)。

再用dst = cv2.undistort(img, mtx, dist, None, mtx)就可以恢复图像了。

Perspective Transform

To transform an image such that we are effectively viewing objects from a different angle or direction.

```
offset = 100 # offset for dst points
# Grab the image shape
img_size = (gray.shape[1], gray.shape[0])

# For source points I'm grabbing the outer four detected corners
src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]])
# For destination points, I'm arbitrarily choosing some points to be
# a nice fit for displaying our warped result
# again, not exact, but close enough for our purposes
dst = np.float32([[offset, offset], [img_size[0]-offset, offset], [img_size[0]-offset, img_size[1]-offset],
                  [offset, img_size[1]-offset]])
# Given src and dst points, calculate the perspective transform matrix
M = cv2.getPerspectiveTransform(src, dst)
# Warp the image using OpenCV warpPerspective()
warped = cv2.warpPerspective(undist, M, img_size)
```

注意dst中对应于src中的四点可以任意, getPerspectiveTransform中注意参数顺序。

Sobel

```
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
```

```
# Calculate the gradient magnitude
gradmag = np.sqrt(sobelx**2 + sobely**2)
# Rescale to 8 bit
scale_factor = np.max(gradmag)/255
gradmag = (gradmag/scale_factor).astype(np.uint8)
binary_output = np.zeros_like(gradmag)
binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])] = 1
```

Color and Gradient

HSV color space (hue, saturation, and value), and **HLS** space (hue, lightness, and saturation).
S channel相对于其它channel(R,G,B,L,h,v)对于阈值分割更robust。

Lane Finding

上面我们已经介绍过了直线检测，现在我们主要讨论相对复杂的曲线检测。检测的步骤如下：

- 通过棋盘图像计算相机参数与扭曲系数
- 对原始图像进行扭曲矫正
- 通过颜色变换，以及梯度等方法来得到一个基于阈值的二进制图像
- 应用perspective transform来得到一张矫正的二进制图(“birds-eye-view”)
- 得到路道的像素并且找到路道的边界
- 确定路道的曲率并且车辆相对于中心的位置
- Wrap the detected lane boundaries back onto the original image
- Output visual display of the lane boundaries and numerical estimation of lane curvature and vehicle position

任务

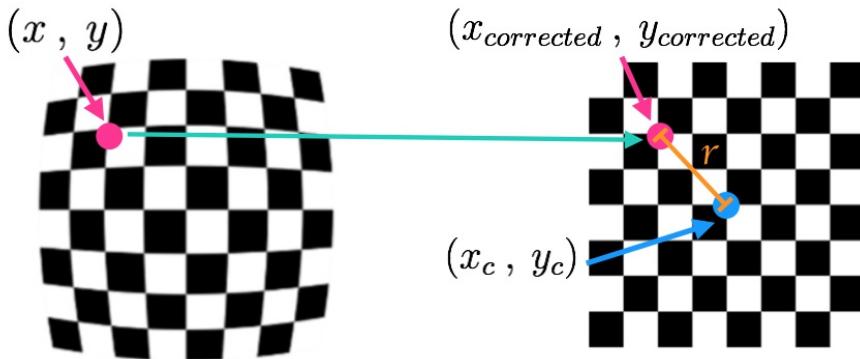
1. 明白相机参数，扭曲系数的直观图像与怎么计算这些参数？
2. 怎么挑选颜色变换来得到鲁棒的图像？
3. 怎么进行perspective transform(透视变换)？
4. 除了基于Histogram的极值与卷积之外，还有什么其它方式来检测路道？
5. 怎么计算路道的曲率以及怎么快速的代码实现？
6. 对于崎岖，被遮挡的山道怎么有效的进行路道的识别？考虑特殊的情形，也就是可见度不高，不明显的路道怎么有效的确定路道？
7. 如果你自己有一辆无人车，你怎么基于特定场景(工厂环境)设计鲁棒，计算量极小的算法来实现道路检测，自动驾驶？
8. 明白无人驾驶在哪些方面无深度学习而不可？

Camera Calibration

由于相机棱镜的不完美，导致了图形的径向扭曲，由于相机位置不正，导致了图像的切向扭曲，因为我们有必要对现实扭曲的图像进行矫正来得到完美的图像。

To ensure that the geometrical shape of objects is represented consistently, no matter where

they appear in an image.



(x_c, y_c) 是扭曲中心, r 是原始图像中一点到扭曲中心的距离。

radial distortion

径向扭曲, 具有旋转不变性, 因此对x,y的修正是一样的。通过一些高阶的修正来还原模型, 我们需要求得这些修正的系数。

$$x_{distored} = x_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

$$y_{distored} = y_{ideal}(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

tangential distortion

$$x_{corrected} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y + [2p_2xy + p_1(r^2 + 2y^2)]$$

Calibrating Camera

To computer the transformation between 3D object points in the world and 2D image points

我们一般使用棋盘模型来的得到这些扭曲系数(k_1, k_2, k_3, p_1, p_2)

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img_size, None, None),
其中mtx is camera matrix, mtx, dist is distortion coefficients. 具体而言, imgpoints是真实的扭曲的图像中的点的3D坐标, objpoints是没有扭曲的棋盘坐标, 是2D的, 通过这两者之间, 可以建立一个映射关系, 返回ret, mtx, dist, rvecs, tvecs这些参数。

其中imgpoints可以通过OpenCV的函数来搜索到扭曲图像中的角点, ret, corners = cv2.findChessboardCorners(gray, (8,6), None)。

再用dst = cv2.undistort(img, mtx, dist, None, mtx)就可以恢复图像了。

具体步骤如下:

Finding chessboard corners (for an 8x6 board):

```
ret, corners = cv2.findChessboardCorners(gray, (8,6), None)
```

Drawing detected corners on an image:

```
img = cv2.drawChessboardCorners(img, (8,6), corners, ret)
```

Camera calibration, given object points, image points, and the **shape of the grayscale image**:

```
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)
```

Undistorting a test image:

```
dst = cv2.undistort(img, mtx, dist, None, mtx)
```

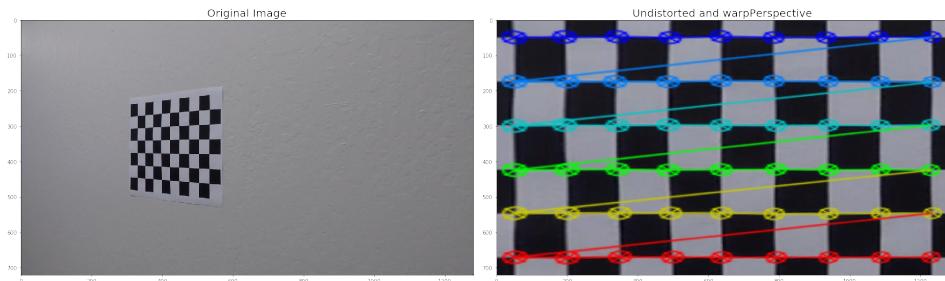
Perspective Transform

To transform an image such that we are effectively viewing objects from a different angle or direction.

```
offset = 100 # offset for dst points
# Grab the image shape
img_size = (gray.shape[1], gray.shape[0])

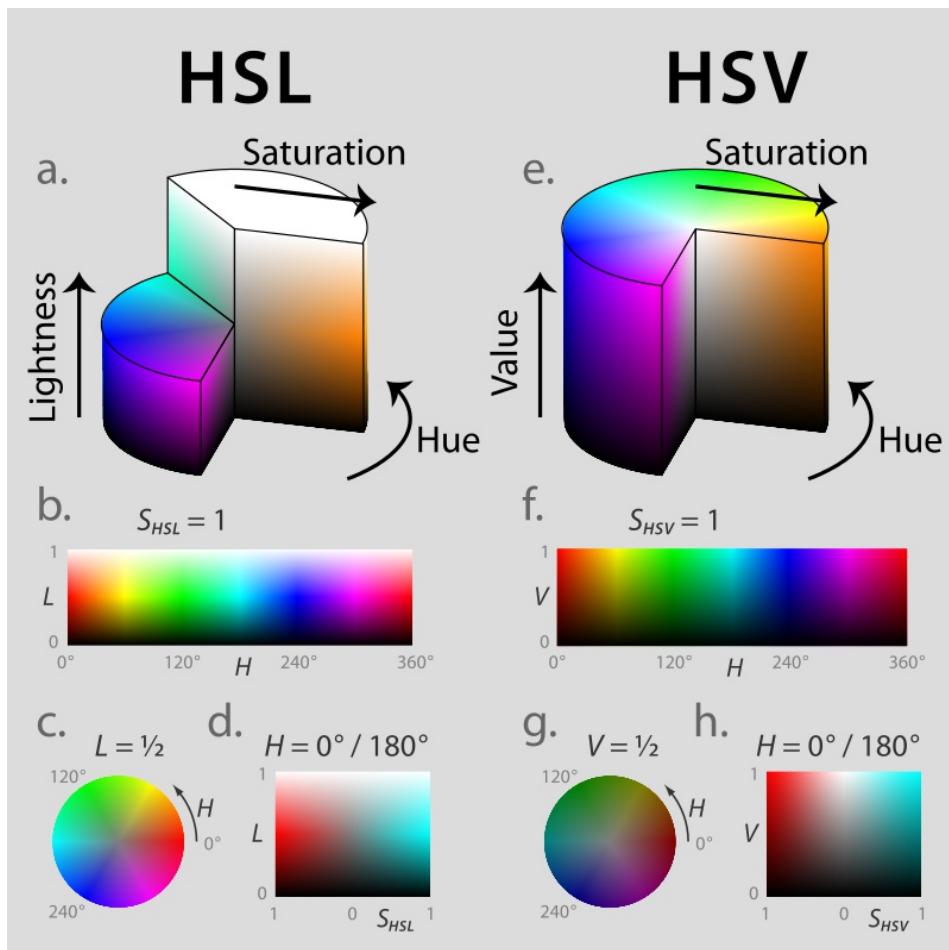
# For source points I'm grabbing the outer four detected corners
src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]])
# For destination points, I'm arbitrarily choosing some points to be
# a nice fit for displaying our warped result
# again, not exact, but close enough for our purposes
dst = np.float32([[offset, offset], [img_size[0]-offset, offset], [img_size[0]-offset, img_size[1]-offset],
                  [offset, img_size[1]-offset]])
# Given src and dst points, calculate the perspective transform matrix
M = cv2.getPerspectiveTransform(src, dst)
# Warp the image using OpenCV warpPerspective()
warped = cv2.warpPerspective(undist, M, img_size)
```

注意dst中对应于src中的四点可以任意, getPerspectiveTransform中注意参数顺序。



Color transformation, HSV and HLS

彩色图像可以用RGB来表示, 也可以用HSV空间(hue色相, saturation饱和度, and value明度), 或者HLS空间(hue色相, lightness亮度, and saturation饱和度)来表示。



HSL(a~d)和HSV(e~h)。上半部分(a、e)：两者的3D模型截面。下半部分：将模型中三个参数的其中之一固定为常量，其它两个参数的图像。(图像引用自维基百科中文版)
由OpenCV的cv2.imread()可以得到RGB图像，再由

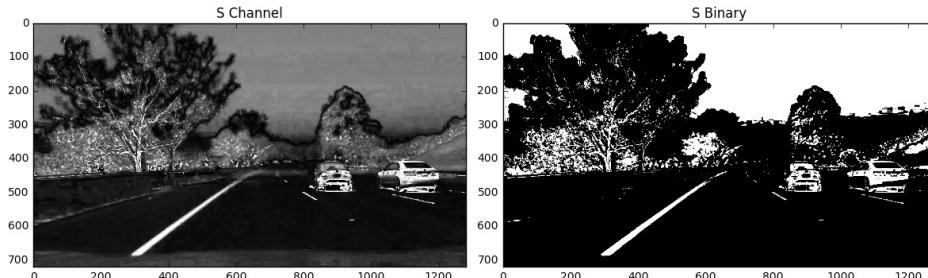
```
hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
H = hls[:, :, 0]
L = hls[:, :, 1]
S = hls[:, :, 2]
```

可以把RGB转换成HLS图像。

HLS and Color Thresholds

实验发现，对于目标检测而言，对S通道进行阈值分割能得到更Robust的结果，特殊情况例外，比如R,G,B单色的目标物体。

```
thresh = (90, 255)
binary = np.zeros_like(S)
binary[(S > thresh[0]) & (S <= thresh[1])] = 1
```



Gradient Thresholds

我们可以通过sobel算子来求不同方向的梯度，可以用来检测方向以及通过阈值来进行分割。
强度阈值：

$$absobel_x = (sobel_x^2)^{1/2}$$

$$absobel_y = (sobel_y^2)^{1/2}$$

$$absobel_{xy} = (sobel_x^2 + sobel_y^2)^{1/2}$$

角度阈值： $\arctan(sobel_y / sobel_x)$

Sobel Operator

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

```
gray = cv2.cvtColor(im, cv2.COLOR_RGB2GRAY)
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1)
abs_sobelx = np.absolute(sobelx)
scaled_sobel = np.uint8(255*abs_sobelx/np.max(abs_sobelx))
```

```
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=sobel_kernel)
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=sobel_kernel)
# Calculate the gradient magnitude
gradmag = np.sqrt(sobelx**2 + sobely**2)
# Rescale to 8 bit
scale_factor = np.max(gradmag)/255
gradmag = (gradmag/scale_factor).astype(np.uint8)
binary_output = np.zeros_like(gradmag)
binary_output[(gradmag >= mag_thresh[0]) & (gradmag <= mag_thresh[1])] = 1
```

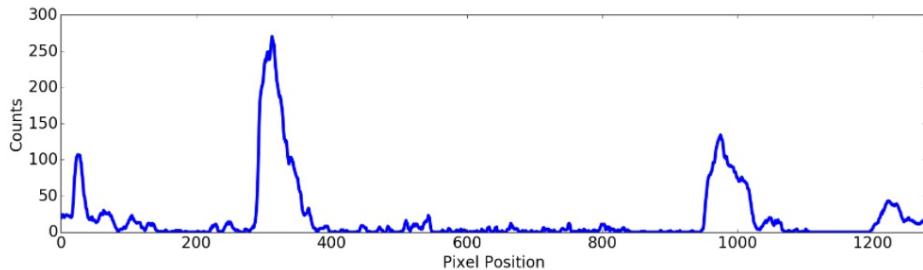
Advanced Topic for Finding Lines

Histogram Peaks



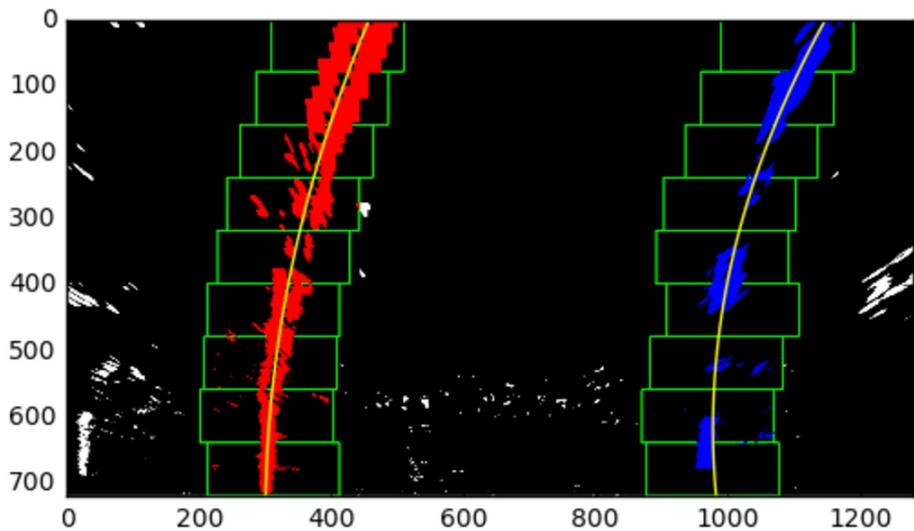
```
import numpy as np
import matplotlib.pyplot as plt

histogram = np.sum(img[img.shape[0]//2:,:,:], axis=0)
plt.plot(histogram)
```



Sliding Window

如果我们把图像切分成小的window, 通过上面的Histogram Peaks找到每个小window的peak, 也就是对于的路线最可能过的点, 做一条曲线来拟合这些点, 这条曲线就是lane line。如下图中的黄线。下图中的绿色小窗口的宽度就是margin宽度, 我们认为lane line位于margin区域内。



```

import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import cv2

# Load our image
binary_warped = mpimg.imread('warped_example.jpg')

def find_lane_pixels(binary_warped):
    # Take a histogram of the bottom half of the image
    histogram = np.sum(binary_warped[binary_warped.shape[0] // 2:, :], axis=0)
    # Create an output image to draw on and visualize the result
    out_img = np.dstack((binary_warped, binary_warped, binary_warped))
    # Find the peak of the left and right halves of the histogram
    # These will be the starting point for the left and right lines
    midpoint = np.int(histogram.shape[0] // 2)
    leftx_base = np.argmax(histogram[:midpoint])
    rightx_base = np.argmax(histogram[midpoint:]) + midpoint

    # HYPERPARAMETERS
    # Choose the number of sliding windows
    nwindows = 9
    # Set the width of the windows +/- margin
    margin = 100
    # Set minimum number of pixels found to recenter window
    minpix = 50

    # Set height of windows - based on nwindows above and image shape
    window_height = np.int(binary_warped.shape[0] // nwindows)
    # Identify the x and y positions of all nonzero pixels in the image
    nonzero = binary_warped.nonzero()
    nonzeroy = np.array(nonzero[0])

```

```

nonzerox = np.array(nonzero[1])
# Current positions to be updated later for each window in nwindows
leftx_current = leftx_base
rightx_current = rightx_base

# Create empty lists to receive left and right lane pixel indices
left_lane_inds = []
right_lane_inds = []

lefty = []
righty = []

# Step through the windows one by one
for window in range(nwindows):
    # Identify window boundaries in x and y (and right and left)
    win_y_low = binary_warped.shape[0] - (window + 1) * window_height
    win_y_high = binary_warped.shape[0] - window * window_height
    #### TO-DO: Find the four below boundaries of the window ####
    win_xleft_low = leftx_current - margin # Update this
    win_xleft_high = leftx_current + margin # Update this
    win_xright_low = rightx_current - margin # Update this
    win_xright_high = rightx_current + margin # Update this

    # Draw the windows on the visualization image
    cv2.rectangle(out_img, (win_xleft_low, win_y_low),
                  (win_xleft_high, win_y_high), (0, 255, 0), 2)
    cv2.rectangle(out_img, (win_xright_low, win_y_low),
                  (win_xright_high, win_y_high), (0, 255, 0), 2)

    #### TO-DO: Identify the nonzero pixels in x and y within the window ####
    good_left_inds = None
    good_right_inds = None

    histogram2 = np.sum(binary_warped[win_y_low:win_y_high, :], axis=0)

    good_left_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
                      (nonzerox >= win_xleft_low) & (nonzerox < win_xleft_high)).nonzero()
    )[0]

    good_right_inds = ((nonzeroy >= win_y_low) & (nonzeroy < win_y_high) &
                      (nonzerox >= win_xright_low) & (nonzerox < win_xright_high)).nonzero()
    )[0]

    if len(good_left_inds) > minpix:
        leftx_current = np.int(np.mean(nonzerox[good_left_inds]))
    if len(good_right_inds) > minpix:
        rightx_current = np.int(np.mean(nonzerox[good_right_inds]))

    # Append these indices to the lists
    left_lane_inds.append(good_left_inds)
    right_lane_inds.append(good_right_inds)

    #### TO-DO: If you found > minpix pixels, recenter next window ####
    #### (`right` or `leftx_current`) on their mean position ####
    pass # Remove this when you add your function

```

```

# Concatenate the arrays of indices (previously was a list of lists of pixels)
try:
    left_lane_inds = np.concatenate(left_lane_inds)
    right_lane_inds = np.concatenate(right_lane_inds)
except ValueError:
    # Avoids an error if the above is not implemented fully
    pass

# Extract left and right line pixel positions
leftx = nonzerox[left_lane_inds]
lefty = nonzeroy[left_lane_inds]
#leftx = left_lane_inds
#rightx = right_lane_inds
rightx = nonzerox[right_lane_inds]
righty = nonzeroy[right_lane_inds]

return leftx, lefty, rightx, righty, out_img


def fit_polynomial(binary_warped):
    # Find our lane pixels first
    leftx, lefty, rightx, righty, out_img = find_lane_pixels(binary_warped)

    ### TO-DO: Fit a second order polynomial to each using `np.polyfit` #####
    left_fit = np.polyfit(lefty, leftx, 2)
    right_fit = np.polyfit(righty, rightx, 2)

    # Generate x and y values for plotting
    ploty = np.linspace(0, binary_warped.shape[0] - 1, binary_warped.shape[0])
    try:
        left_fitx = left_fit[0] * ploty ** 2 + left_fit[1] * ploty + left_fit[2]
        right_fitx = right_fit[0] * ploty ** 2 + right_fit[1] * ploty + right_fit[2]
    except TypeError:
        # Avoids an error if `left` and `right_fit` are still none or incorrect
        print('The function failed to fit a line!')
        left_fitx = 1 * ploty ** 2 + 1 * ploty
        right_fitx = 1 * ploty ** 2 + 1 * ploty

    ## Visualization ##
    # Colors in the left and right lane regions
    out_img[lefty, leftx] = [255, 0, 0]
    out_img[righty, rightx] = [0, 0, 255]

    # Plots the left and right polynomials on the lane lines
    plt.plot(left_fitx, ploty, color='yellow')
    plt.plot(right_fitx, ploty, color='yellow')

    return out_img

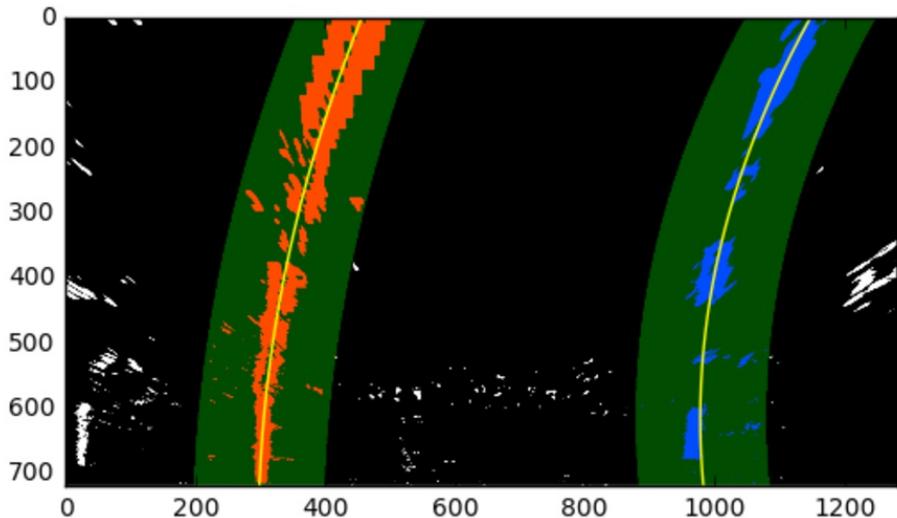

out_img = fit_polynomial(binary_warped)

plt.imshow(out_img)
plt.show()

```

Search from Prior

这是基于先验(前面的位置与方向的方法), 具体就是, 根据以前的运动曲线, 比如在小范围内可以拟合成二次曲线, 再基于当前的位置, 我们就可以确定下一时刻大致的运动范围, 因此我们不需要检验所有与前进方向垂直的区域, 只需要检验Margin区域来得到下一位置的Lane line的位置。通过一步步的迭代, 更新拟合曲线的参数与位置, 就可以得到整条Lane Line。



可行区域填充

得到Lane Lines之后, 我们使用dstack是二进制图像变成三维图像, 再用彩色填充道路之间的区域。

再用getPerspectiveTransform得到一个逆矩阵, 从而把bird view变成原始空间的图像。

再叠加原始的图像, 得到对道路进行填充的图像。

src不必要太精确, 只需是包含原始道路区域的梯形, 而dst坐标是特定矩形的四个顶点。当然, 求src的精确方法是, 通过卷积, 来求得道路的终点。

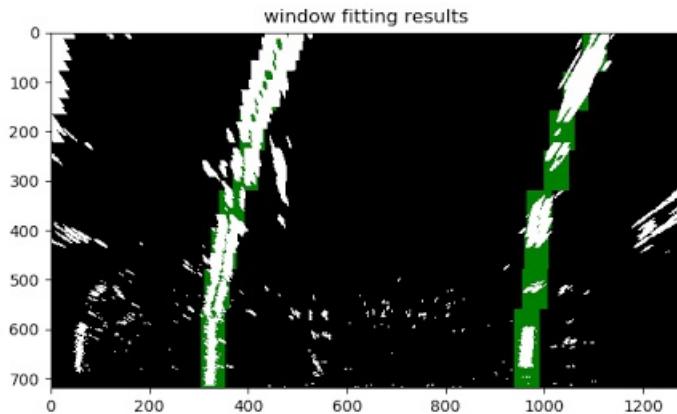
```
Minv = cv2.getPerspectiveTransform(dst, src)

warp_zero = np.zeros_like(combined_binary).astype(np.uint8)
color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
pts_left = np.array([np.flipud(np.transpose(np.vstack([left_fitx, lefty])))])
pts_right = np.array([np.transpose(np.vstack([right_fitx, righty]))])
pts = np.hstack((pts_left, pts_right))
cv2.polylines(color_warp, np.int_(pts), isClosed=False, color=(0,0,255), thickness = 40)
cv2.fillPoly(color_warp, np.int_(pts), (0,255, 0))
newwarp = cv2.warpPerspective(color_warp, Minv, (combined_binary.shape[1], combined_binary.shape[0]))
result = cv2.addWeighted(mpimg.imread(image), 1, newwarp, 0.5, 0)
```

Convolution Methods

我们也可以利用卷积来寻找Lane Line的位置，具体步骤就是。

对一个小的窗口，比如高度为80，长度L就是图像x轴长度的一个窗口区域，对这个窗口区域做Histogram，得到一个大小为L的一维数组，一个卷积核对这个数组做卷积，位于我们感兴趣的Margin区域内的Peak位置，就是Lane Line上的点。把这些点连起来就可以得到整条Lane Line。



Measuring Curvature

对于曲线：

$$x = f(y) = Ay^2 + By + C$$

曲线 $x = f(y)$ 的曲率半径定义为：

$$R_{curve} = \frac{(1 + (\frac{dy}{dx})^2)^{3/2}}{|\frac{d^2x}{dy^2}|}$$

对于我们的例子有：

$$f'(y) = \frac{dy}{dx} = 2Ay + B$$

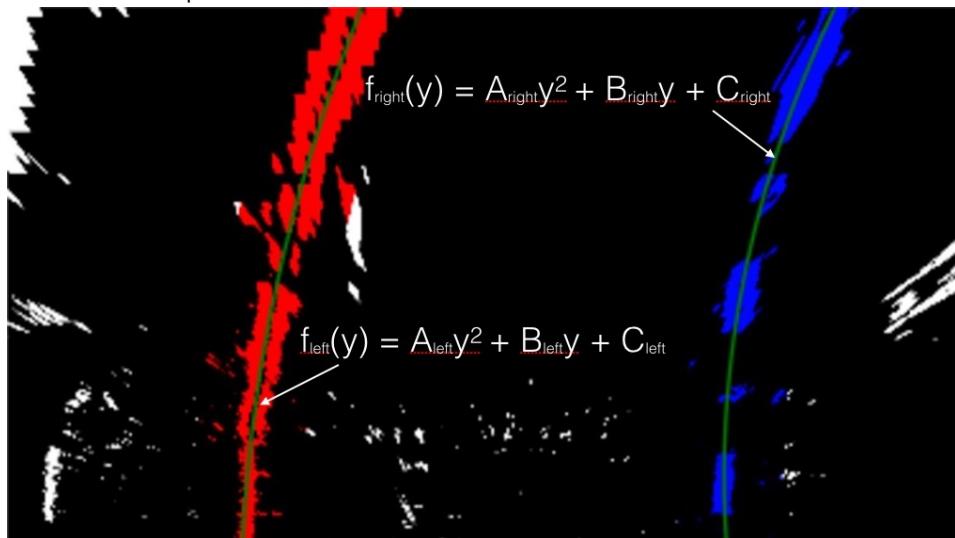
$$f''(y) = \frac{d^2x}{dy^2} = 2A$$

因此， (y, x) 点的曲率半径为：

$$R_{curve} = \frac{(1 + (2Ay + B)^2)^{3/2}}{|2A|}$$

无标度化

当然，我们还可以进行无标度化，来得到一个不带单位的方程，具体就是对x,y除以一个单位因子，这个因子就是每个pixel对应的物理长度。



在实际中，我们需要确保两条线之间的距离在一定的区间，必须要有这个约束。

Tensorflow

pooling layers

Advantages:

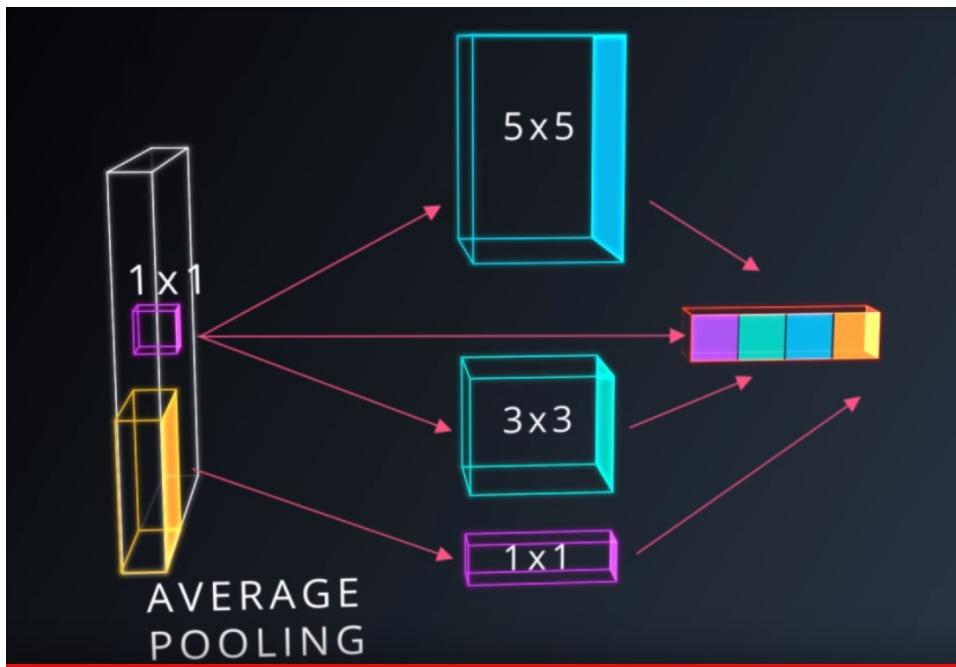
Decrease the size of the output and prevent overfitting. Preventing overfitting is a consequence of reducing the output size, which in turn, reduces the number of parameters in future layers.

Recently, pooling layers have fallen out of favor. Some reasons are:

1. Recent datasets are so big and complex we're more concerned about underfitting.
2. Dropout is a much better regularizer.
3. Pooling results in a loss of information. Think about the max pooling operation as an example.
4. We only keep the largest of n numbers, thereby disregarding n-1 numbers completely.

```
input = tf.placeholder(tf.float32, (None, 4, 4, 5))
filter_shape = [1, 2, 2, 1]
strides = [1, 2, 2, 1]
padding = 'VALID'
pool = tf.nn.max_pool(input, filter_shape, strides, padding)
```

Inception



深度学习成功的关键

神经网络成功的原因¹

1. 有逐层的处理
2. 有特征的内部变化
3. 有足够的模型复杂度

为什么深度学习现在才流行

模型的复杂度对应的就是模型的学习能力。越复杂的模型，学习能力会越强。哪为什么以前我们不用复杂的模型，现在却用了？主要有如下三点：

1. 我们今天有更大的数据
2. 有强大的计算设备
3. 有很多有效的训练技巧，比如逐层训练

这就解释了为啥以前不用复杂的神经网络，而现在用了。此外，由带隐藏层神经网络的万有逼近定理，可以知道神经网络的强大学习能力。

深度学习相对于以前的机器学习的优点是，以前需要我们手动的设计特征，现在深度学习的端对端学习把特征学习和分类器学习器联系在一起了。

深度学习的逐层抽象

这里主要讨论CNN。

只需要一层隐藏层的神经网络也可以逼近任何连续函数，哪为啥需要深度神经网络呢？这就是浅层神经网络不能实现的逐层抽象。深度神经网络学到的特征是分层的，比如最常见的图像识别中，最底层的是像素特征，再上一层是边特征，再上层是由边组成的局部轮廓，最上面是全局特征。逐层抽象是深度神经网络相对于浅层神经网络的一大优势。

其实我自己对逐层抽象的理解就是从表示方式来说，逐层抽象需要的神经元会远少于单隐藏层的神经网络。比如要表示一幅 $n \times n$ 的图像，单隐藏层的神经网络需要 n^2 个神经元。而如果是双层网络的话，我们可以做类似于PCA或者SVD的操作，一层表示X轴对应的隐变量成分，一层表示Y轴的隐变量成分，因此，我们需要的神经元数目只是正比于n而不是当隐藏层的 n^2 。因此，逐层抽象能简化模型的复杂度。

一个问题是怎用神经网络来做SVD？也就是做推荐。

当然，你或许也知道一些机器学习方法也有类似的逐层抽象的模型，比如Boosting模型，他们为啥没有取得深度学习的成功？其原因就是，第一，复杂度还不够，第二，更关键的是，它始终在原始空间里面做事情，所有的这些学习器都在原始特征空间，中间没有进行任何特征变换。原始空间的理解就是，拿图像识别来说，它只是通过决策树来一层层的逼近目标，通过信息熵或者基尼系数来做决策，它不会去对图像做抽象，只有像素级别的逼近。

深度学习成功的关键

主要包含这么几点：

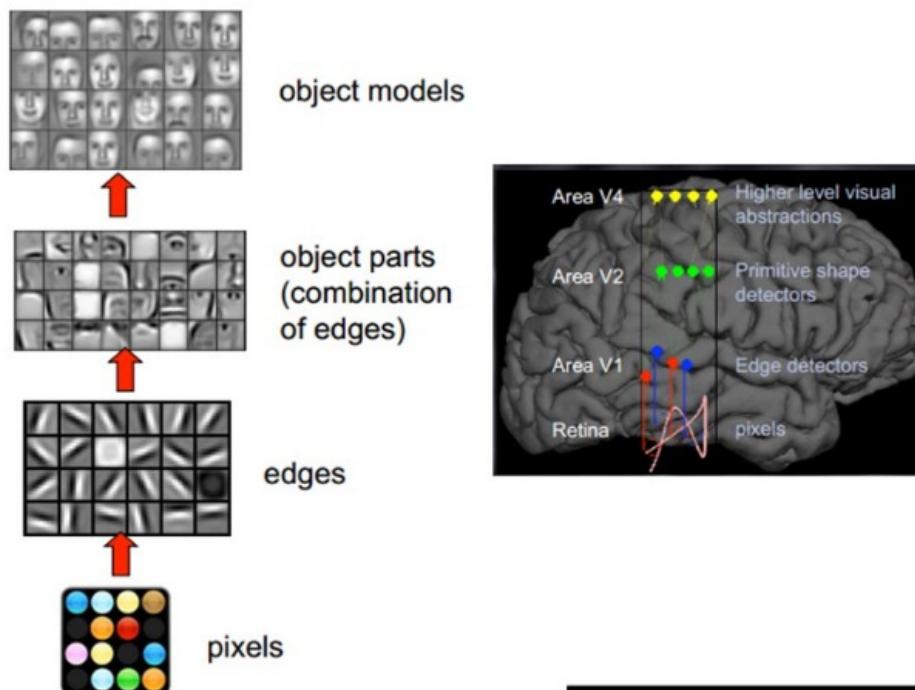
1. 逐层的处理，也就是逐层抽象
2. 有一个内部的特征变换
3. 有足够的复杂度

深度学习能够成功的领域

就目前而言，深度学习在图像、视频、语音方面取得巨大的成功，而在其他问题上，比如Kaggle上的机票、订旅馆、推荐方面，还是传统机器学习方法占主导。若要总结深度学习能在哪些领域/问题方面能够成功，那就是处理的这些对象，是由一些很底层的结构的，这些底层的结构是在我们处理的任务上市通用的，也就是局部的时间平移不变性；

对于图像而言，图像的底层的边缘特征很难随时间而变。对语音，我们处理的也是一段时间，一个地区的语音任务。对于不同地区，就涉及到方言的问题，恐怕就没有通用的模型了，我们必须对不同的语言，不同的方言训练不同的模型。

人对于推荐系统，这些对象，就是人类的喜好，很容易随着时间而变迁，因此很难有通用的模型。对于这些问题，现在深度学习并没有比传统机器学习，比如Random Forest, GBDT强，是否将来的神经网络可以比这些传统的机器学习方法好？还是机器学习方法根本上就很难胜任这些问题，探索这些更本质的问题，是我们需要重视的一个方向，这个问题没有弄清楚，那么我们把深度学习用在这些问题，可能就是走了一条错误的路上。



深度学习的缺点

1. 参数调节很难，在不同任务，比如跨图像与语音就需要重新调节参数。
2. 结果重复性差。数据与方法相同，超参数不一样，结果就不一样。

一些要做的事情

深度神经网络是建立在可微函数与BP算法的基础之上的，那么能不能用不可微的函数把神经网络做的更深呢？这时就不能用BP算法来训练了。

现实的很多问题就不是可微的，因此我们需要发展非可微的深层网络。此外，我们还可以借鉴几十年来机器学习的成功模型，作为构建深度模型的基础，这些模型中有一些就不是可微的。通过这样来构建出能在当前深度学习效果不如传统机器学习的问题上比机器学习更胜一筹。

gcForest

比如周志华gcForest，它是一个基于树模型的方法，主要是借用集成学习的很多想法。其次在很多不同的任务上，它的模型得到的结果和深度神经网络是高度相似的，除了一些大规模的图像等等。在其他的任务上，特别是跨任务表现非常好，我们可以用同样一套参数，用在不同的任务中得到不错的性能，就不需要逐任务的慢慢调参数。

训练CNN的技巧

1. 从优化角度
 - i. BP算法
 - ii. SGD, Momentum, Adam系列
2. 从模型泛化误差角度：防止过拟合
 - i. 正则化方法
 - i. L1/L2
 - ii. 提前终止
 - ii. 扩大数据量方法
 - i. 数据增强：旋转，对折
 - ii. 对比度减弱/模糊化，对比度增强，加噪声
 - iii. 模型集成方法
 - i. Bagging，结合多个模型得到模型平均，减少泛化误差
 - ii. Dropout
3. 从训练角度：防止梯度消失
 - i. ReLU
 - ii. Batch Normalization
4. 增加非线性或者模型表达能力
 - i. 激活函数：Sigmoid, ReLU
5. 针对CNN的
 - i. Pooling，减小表达空间尺寸，从而减小网络参数与计算时间，防止过拟合
 - ii. 卷积：稀疏连接，权值共享，减小模型参数

要思考一个问题，怎么基于这些方法，重构出CNN, RNN, 以及怎么利用这些技巧来设计新的模型？

1. 周志华：满足这三大条件，可以考虑不用深度神经网络 <https://36kr.com/p/5129474.html> ↵

先验与算法设计

这一节是目的是总结怎么通过先验来设计算法。根据“没有免费午餐定理”我们知道，不存在一个普适的算法在所有场合下都表现很好，因此我们必须根据样本的先验知识来选择模型与设计算法。因此，我们到底该怎么设计我们的算法，也就是基于不同的问题背景，得出先验知识，然后把先验知识作为我们设计算法的基本条件。我们首先要分析的是一些比较成功的网络，比如CNN, RNN, 可能还有一些机器学习模型；分析它怎么利用起了先验知识而获得了相比于其它模型更好的成果。因此，这节的核心是先验与算法设计。最终的目的是通过分析已有的成功经验，借鉴这些经验而把它用在未来的问题上面去，也就是《论语》中的“告诸往而知来”。

NO FREE LUNCH

没有免费的午餐定理：没有一个模型是对所有的样本是最优的。

比如下面的例子：

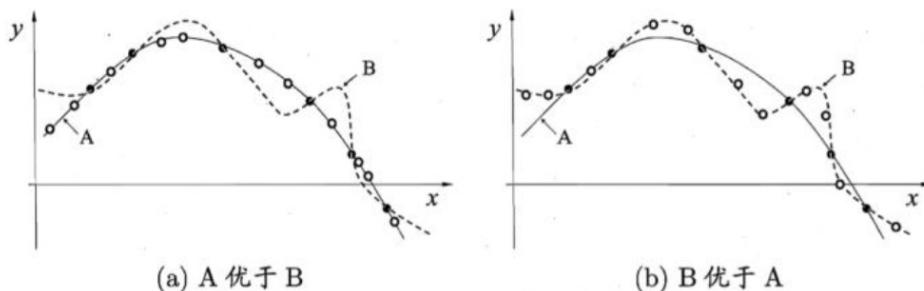


图 1.4 没有免费的午餐。(黑点: 训练样本; 白点: 测试样本)

黑点是训练样本，白点是测试样本点。根据训练样本，我们得到模型A, B。如果测试点不同，则A, B的相对表现不同。问题的关键在于，我们先验的假设了(测试)样本点在所有空间是均匀分布的，因此，在训练集上训练出的所有模型的效果是一样的，因为你总有样本集是落在你模型预测的曲线之上的。

因此，如果样本点在整个数据空间是均匀分布的前提下，则不存在一个模型或算法在所有问题上比其它算法更优越。

但是现实遇到的问题是，数据的分布是受限制的，数据存在一个先验分布，因此会存在一些模型比其它模型更有效。这里NFL不再有是因为问题的前提--数据是均匀分布的--不再成立。因此，脱离场景谈方法是无效的，我们只能针对特定的问题，寻找最佳的优化方法。

正则化

没有免费的午餐定理告诉我们没有一个模型是对所有的样本是最优的¹，因此，对于特定的样本，我们需要把先验加进模型中(也就是损失函数中)，这个先验就是我们对这个样本的知识，比如样本数据的分布，样本的均值或者方差等(统计物理学中的系统的能量可以看成一种均值，统计物理学中有很多启发性的模型)。加到模型中的就是我们的正则项，正则项就是一个先验，是我们需要模型

满足的约束。

一个问题是，正则化只能使我们的模型更稳定，不会更精确，那为啥还需要正则项？先验怎么作为正则化的形式加进模型，使得我们的模型更优越呢？实际上，优化不加正则项的损失函数就是求得极大似然；优化加正则项的损失函数就是最大后验概率估计。后者考虑了模型的风险，而前者没有考虑模型的风险。因此，加不加正则项，取决于我们是否需要考虑模型的风险，一般只有处理under-fitting(欠定)问题，也就是变量多余方程数目时，我们才加正则项，对于超定方程，我们根本就不需要加正则项，我们只需要用最小二乘就可以解决。

最后还留下一个问题，就是损失函数怎么设计，也就是范数怎么选取， L_1, L_2, L_∞ ，究竟选取哪一个？这个就涉及到模型预测误差的先验分布了，看是拉普拉斯分布(L_1)，高斯分布(L_2)还是其它的？nuclear 范数，无穷范数对应的先验是什么？

极大似然与最大后验概率估计

最大似然估计是求参数 θ ，使似然函数 $P(y|\theta)$ 最大。最大后验概率估计则是求 θ 使得 $P(y|\theta)P(\theta)$ 最大。求得的 θ 不单单让似然函数最大，要 θ 出现的概率也很大。其实对数化之后，就是加上一个有关模型 θ 的正则项。也就是结构风险最小化就等价于最大后验概率估计。

因此，加正则约束的优化问题，实际上是最大化后验概率估计，而不是最大化似然函数。比如加L2正则化，就是认为，参数先验的服从高斯分布。

MAP实际上在最大化

$$P(\theta|y) = \frac{P(y|\theta)P(\theta)}{P(y)}$$

因为 $P(y)$ 是个固定值，也就是 y 在实验中出现的次数，比如硬币正面朝上的次数。所以可以去掉分母 $P(y)$ 。因此最大化后验概率估计就是最大化 $P(\theta|y)$ 。最大化 $P(\theta|y)$ 的意义很明确， y 出现了，要求 θ 取什么值使得后验概率 $P(\theta|y)$ 最大。

那最大化 $P(\theta|y)$ 与最大化 $P(y|\theta)$ 的区别是什么？

MAP最大化 $P(\theta|y)$ ：也就是在已知的数据中，求结构风险最小的模型参数。

MLP最大化 $P(y|\theta)$ ：也就是求使得数据出现概率最大的模型参数，它相当于不对模型做惩罚，结果可能是训练集上效果好，测试集合上效果差。

一些技术

这里，有些是技术，有些是先验知识，但更多的是先验知识。

1. 误差反向传播，梯度下降方法
2. 梯度消失：Relu, leak-Relu
3. 循环网络，矩阵相乘与梯度爆炸：
4. Pooling：特性提取，几何变换不变性
5. Dropout与Bagging：
6. 正则化
7. 卷积
8. 批量归一化与泛化能力

如何设计一个卷积神经网络来实现图像分类

BP网络

在本章第一节，我们介绍了如何实现一个误差反向的神经网络。因此，我们可以把一张图转化为一个向量，输入到BP网络中，再根据总的类别来设置输出向量的长度，比如1000个分类就让输出的维度是1000。这样我们可以实现一个很简单的用于图像分类的神经网络。

当然这只是一个很粗浅的模型，因为它没有利用到图像固有的一些特性，或者说图像识别这个问题背后的先验知识。

先验与熵值

卷积¹

全连接认为下层的输出与上层的所有元素都有关系，上层的每个元素的作用是均等的。而卷积操作，基于这样一个假设，网络下一层的元素只与上一层网络的部分元素有关（稀疏连接，权值共享）。这相当于全连接来说，卷积相当于加了一个矩形窗，这是一个很强的先验。但正是由于有这个先验，使得我们的计算量远远小于全连接网络。

卷积导致了参数共享，而参数共享的先验就是平移不变性。

或者从另外一个角度来说，卷积，这种稀疏连接的方式，区别于全连接在于，它认为下一层每个神经元只与上一层特定区域的神经元有相互作用，或者说，与其它神经元的关联权重为0，也就是从全局作用变成局部作用，就像从一个方差很大的高斯分布变成一个方差很小的高斯分布。这是一个强先验，所谓的强先验就是具有小的熵值，弱先验具有大的熵值。强先验对应更确定的知识，未知性小，而弱先验就是未知性大，极端的就是没有先验，也就是说不对模型做任何假设，比如全连接网络。

池化

能显著降低参数数量，还能保持平移，伸缩，旋转操作的不变性。

¹. 《深度学习》9.6节 ↵

模型压缩

传统模型计算量

Model	Model Size(MB)	Million Mult-Adds	Million Parameters
AlexNet ^[1]	>200	720	60
VGG16 ^[2]	>500	15300	138
GoogleNet ^[3]	~50	1550	6.8
Inception-v3 ^[4]	90-100	5000	23.2

MobileNet:

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
GoogleNet	69.8%	1550	6.8
VGG 16	71.5%	15300	138

不同压缩方法在ImageNet上的对比实验结果

CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD [5]	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning [11]	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression [10]	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	50x	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	363x	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	510x	57.5%	80.3%

权值值化

模型剪裁

SqueezeNet

[SqueezeNet](#)

#

第七章 推荐系统

本章的结构如下：

1. 推荐方法的分类
 - i. 协同过滤方法
 - ii. 基于内容的推荐
 - iii. 基于知识的推荐
2. 点击率预测
3. 推荐系统算法的评估

推荐系统在数学上就是一个超大规模稀疏矩阵的分解与重构问题，推荐实际上就是寻找User-Item矩阵中本该不为0而实际为0的矩阵元。

对矩阵分块，实现的就是聚类功能，可以有基于Item的聚类，也可以有基于User的聚类，因此，要研究怎么有效的对矩阵进行分块。

推荐系统中的Matrix Factorization(MF)或者Dimensionality Reduction，实际上就是数学SVD，因此推荐系统中这些算法也称为SVD模型。

推荐算法分类¹

1. 协同过滤推荐算法 CF:Collaborative Filtering	基于记忆 (Memory-based CF)	基于用户(User-based),相似度 基于物品(Item-based), 相似度
	基于模型(Model-based CF)	隐因子模型(SVD,PCA), 朴素贝叶斯分类
2. 基于内容的推荐算法 (CB:Content-Based)	基于信息检索	TF-IDF 基于Rocchio的CB推荐算法 基于决策树，基于线性分类，基于朴素贝叶斯，基于KNN
3. 基于知识的推荐算法 (KB:Knowledge-based)	关联知识与关联规则挖掘	Apriori
	约束知识与约束推荐算法	约束推荐算法

基于内容的推荐：基于用户的浏览记录，购买记录建立用户画像。基于物品的描述信息，提取物品特征，建立物品画像。

不同推荐算法的优缺点

1. 协同过滤推荐算法:CF的优点:不需要知道物品的内容描述,容易实现用户潜在兴趣的挖掘;缺点:新用户/新物品的冷启动问题,推荐的可解性差。
2. 基于内容的推荐算法:CB的优点,用户间独立性,可解性,新物品的冷启动容易解决。缺点,新用户的冷启动问题难以解决,对用户潜在的兴趣挖掘难以实现,难以提取物品的特征。
3. 基于知识的推荐算法:KB推荐方法中的应用局限。

不同推荐算法之间的融合

怎么通过解决一些常见的推荐系统问题,比如冷启动?

推荐系统算法的评估

矩阵分解

U-I矩阵是一个大的稀疏矩阵,我们需要寻找矩阵的一个低秩分解,实际上就是一个奇异值分解(SVD)。

$$V = UM$$

评分矩阵 $V \in \mathbb{R}^{n \times m}$, SVD实际上就是去找到两个矩阵: $U \in \mathbb{R}^{f \times n}$, $M \in \mathbb{R}^{f \times m}$, 其中矩阵U表示User 和 feature 之间的联系,矩阵V表示 Item 和 feature 之间的联系。 $f \ll m, f \ll n$ 。这样就完成了矩阵的分解。实际计算中,我们并不是通过常规的求本征值本征矢的SVD来做分解,而是通过最小二乘加正则约束来实现分解。

SVD存在的问题

会出现矩阵元为负数的非现实情况。因此有了非负矩阵分解。

一些需要处理的问题

1. 怎么有效的对矩阵进行低秩分解?能否针对不同场景总结出不同的分解方法?
2. 怎么设定求解的损失函数?是欧氏距离而是L1距离?这得回到我们的业务要求,如果U-I矩阵的矩阵元就是消费的金额,是否损失函数中曼哈顿距离比欧氏距离更好?
3. 正则项怎么加?
4. 隐变量个数怎么选择?
5. 怎么加显式反馈数据(explicit feedbacks)和隐式反馈(implicit feedbacks)数据?
6. 怎么设置损失函数?损失函数应该包括哪些项目?
7. 怎么有效的为矩阵分块?矩阵分块实际上实现的就是聚类。

¹. 牛温佳《用户网络行为画像》[←](#)

计算广告学概论¹

一些概念

CPM:Cost per Mille,千次展览付费

RPM:Revenue per Mille千次展览收益

CPC:Cost per Click按点击付费

CPT:Cost Per Time按时间付费

CPS:Cost Per Sale按销售额付费

CPA:Cost per Action按转化付费

ROI:Return On Investment 投入产出比

RTB:Real Time Bidding

ADN:ad Net-work 广告网络

Auction-based advertising:竞价广告

Search ad:搜索广告

GSP:Generalized Second Price广义第二高价

DSP:Demand Side Platform 需求方平台

优化目标

eCPM:excepted Cost Per Mille,千次展示期望收入

ePCM = 点击率x点击价值

流量塑形:在有些情况下,我们可以主动地影响流量,以利于合约的达成。

搜索广告是典型的竞价广告,是整个在线广告中份额最大的部分。

定价问题

常见的两种:GSP与基于纳什均衡的VCG(Vickrey-Clarke-Groves)定价策略。

GSP(广义第二高价):对赢得每一个位置的广告主,都按照他下一位的广告位置出价来收取费用。

基于纳什均衡。

ePCM等于点击率乘以出价 $r = u \cdot v$

一般有如下变形: $r = u^k \cdot v$ 其中k是价格挤压(Squashing)因子.

重定向

把那些曾经对广告主服务发生明确兴趣的用户找出来,向他们投放该广告主的广告。

常用的相似度

余弦相似度

1. 刘鹏《计算广告--互联网商业变现的市场与技术》 [↵](#)

协同过滤算法

协同过滤主要利用用户群过去的行为与意见来预测当前用户最可能喜欢的东西。他的优点是，只需要知道用户-物品评分矩阵，不需要知道物品的任何信息。下面就是一个典型的例子，我们给出用户-物品矩阵，用已知的数据来预测未知的数据：¹

	物品1	物品2	物品3	物品4	物品5
Alice	5	3	4	4	?
用户1	3	1	2	3	3
用户2	4	3	4	3	5
用户3	3	3	1	5	4
用户4	1	5	5	2	1

我们需要利用用户1, 2, 3, 4的数据，以及Alice的部分数据来预测Alice对物品5的评分。有不同的方法来预测

Alice对物品5的评分，我们可以根据Alice已有的数据，来计算Alice与其它用户的Pearson的相关系数，寻找相似度最高的用户，再给出一个评分。

基于用户的最近邻推荐

Pearson相似系数

$U = [u_1, \dots, u_n]$ 代表用户集, $P = [p_1, p_2, \dots, p_m]$ 代表物品集合, 用R代表评分项 $r_{i,j}$ 的 $n \times m$ 评分矩阵, \bar{r}_a 代表用户a的平均评分。

$$sim(a, b) = \frac{x}{y}$$

$$x = \sum_{p \in P} (r_{a,p} - \bar{r}_a)(r_{b,p} - \bar{r}_b)$$

$$y = (\sum_{p \in P} (r_{a,p} - \bar{r}_a)^2)^{1/2} (\sum_{p \in P} (r_{b,p} - \bar{r}_b)^2)^{1/2}$$

只针对a,b已有的公共数据进行计算。比如Alice与用户1只计算物品1, 2, 3, 4上的Pearson相似系数。

基于待计算用户最相似的N个用户(样本)进行预测，基于以下公式预测用户a对物品p的评分：

$$pred(a, p) = \bar{r}_a + \frac{x}{y}$$

$$x = \sum_{b \in N} sim(a, b) * (r_{b,p} - \bar{r}_b)$$

$$y = \sum_{b \in N} sim(a, b)$$

一般用户的近邻数目选择在20-50之间。

基于用户的最近邻推荐的缺点是，由于大型商城用户太多，计算最近邻太费时间，而且用户的偏好数据容易变更。因此有下面的基于物品的最近邻推荐，物品的属性数据很难变更。

基于物品的最近邻推荐

通常我们用余弦相似度来计算用品a,b的相似度：

$$sim(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{ab}}{|\mathbf{a}||\mathbf{b}|}$$

为了考虑到用户的评价评分，因此有了下面的改进的余弦相似度：

$$sim(a, b) = \frac{x}{y}$$

$$x = \sum_{u \in U} (r_{u,a} - \bar{r}_u)(r_{u,b} - \bar{r}_u)$$

$$y = (\sum_{u \in U} (r_{u,a} - \bar{r}_u)^2 \sum_{u \in U} (r_{u,b} - \bar{r}_u)^2)^{1/2}$$

其中U是所有同时给物品a和b评分的用户集合。

通过物品之间的相似度来预测用户对未评分物品的评分，在上面例子中，就是通过物品5与物品1, 2, 3, 4的相似度，以及Alice对物品1, 2, 3, 4的评分来预测Alice对物品5的评分：

$$pred(u, p) = \frac{x}{y}$$

$$x = \sum_{i \in ratedItems(u)} sim(i, p) * r_{u,i}$$

$$y = \sum_{i \in ratedItems(u)} sim(i, p)$$

u为Alice, p是物品5, RatedItems(u)为物品1, 2, 3, 4。实际中ratedItems(u)不会是所有物品，近邻的规模也是一个固定值。

基于模型的推荐技术

协同过滤方法主要介绍两种：

1. SVD
2. pLSA

¹. 《推荐系统》Dietmar Jannach, P9 ↪

FM(Factorization Machines),FFM

FM

一般的线性模型为:

$$y = w_0 + \sum_{i=1}^n w_i x_i$$
 一般模型中, 各个特征是独立考虑的, 没有考虑特征之间的相互关系。如果

考虑特征 x_i, x_j 之间的相互关系, 模型修改如下:

$$y = w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j$$

如果系统的特征比较多的话, 计算复杂度会大大提升。为了降低时间复杂度, 我们引入了辅助向量 latent vector

$$\mathbf{V}_i = [v_{i1}, v_{i2}, \dots, v_{ik}]^T$$

, 辅助变量是描述变量之间的相关性。模型修改如下:

$$y = w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n (V_i, V_j) x_i x_j$$

以上就是FM模型。k是超参数, 一般娶30或者40。时间复杂度是 $O(kn^2)$, 可通过如下方式化简。

$$\begin{aligned} & \sum_{i=1}^n \sum_{j=i+1}^n (V_i, V_j) x_i x_j \\ &= \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n (V_i, V_j) x_i x_j - \frac{1}{2} \sum_{i=1}^n (V_i, V_i) x_i x_i \\ &= \frac{1}{2} \sum_{f=1}^n \left(\sum_{i=1}^n v_{if} x_i \right)^2 - \sum_{i=1}^n v_{if}^2 x_i^2 \end{aligned}$$

通过对每个特征引入latent vector \mathbf{V}_i , 并对公式进行化简, 可以把时间复杂度降为 $O(kn)$.

隐语义模型

LFM(Latent factor model): 通过隐含特征(Latent Factor)联系用户兴趣和物品.

LFM通过如下公式计算用户u对物品i的兴趣:

$$Preference(u, i) = r_{ui} = p_u^T q_i = \sum_{k=1}^K p_{u,k} q_{i,k}$$

公式中 $p_{u,k}$ 和 $q_{i,k}$ 是模型的参数, 其中 $p_{u,k}$ 度量了用户u的兴趣和第k个隐类的关系。而 $q_{i,k}$ 都凉了第k个隐类和物品i之间的关系。

$p_{u,k}, q_{i,k}$ 通过如下方式求解:

$$C = \sum_{(u,i) \in K} (r_{ui} - \hat{r}_{ui})^2 = \sum_{(u,i) \in K} (r_{ui} - \sum_{k=1}^K p_{u,k} q_{i,k})^2 + \lambda ||p_u||^2 + \lambda ||q_i||^2$$

SVD++

指标

1. 准确率
2. 召回率
3. 覆盖率
4. 多样性

非负矩阵分解

$N * p$ 数据矩阵 \mathbf{X} 近似表示为：

$$\mathbf{X} \approx \mathbf{WH}$$

其中: $\mathbf{W} \in R^{N \times r}$, $\mathbf{H} \in R^{r \times p}$, $r \leq \max(N, p)$, 我们假定 $x_{ij}, w_{ik}, h_{kj} \geq 0$.

矩阵 \mathbf{W}, \mathbf{H} 通过最大化下面似然函数确定：

$$L(\mathbf{W}, \mathbf{H}) = \sum_{i=1}^N \sum_{j=1}^p [x_{ij} \log(\mathbf{WH})_{ij} - (\mathbf{WH})_{ij}]$$

这个从一个 x_{ij} 满足均值为 $(\mathbf{WH})_{ij}$ 的泊松分布的模型的似然函数。

通过梯度下降法可以求解。

SVD在推荐系统的应用

假设我们现在有评分矩阵 $V \in \mathbb{R}^{n \times m}$, SVD实际上就是去找到两个矩阵: $U \in \mathbb{R}^{f \times n}$, $M \in \mathbb{R}^{f \times m}$, 其中矩阵U表示 User 和 feature 之间的联系, 矩阵V表示 Item 和 feature 之间的联系。

大家这时候肯定会想, 我们的评分矩阵里面一般会有很多缺失值, 那要怎么去得到 U 和 M 呢? 实际上, 这两个矩阵是通过学习的方式得到的, 而不是直接做矩阵分解。我们定义如下的损失函数:

$$E = \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^m I_{ij} (V_{ij} - p(U_i, M_j))^2 + \frac{k_u}{2} \sum_{i=1}^n \|U_i\|^2 + \frac{k_m}{2} \sum_{j=1}^m \|M_j\|^2$$

其中 $p(U_i, M_j)$ 表示我们对用户 i 对 物品 j 的评分预测:

$$p(U_i, M_j) = U_i^T M_j$$

梯度下降:

$$-\frac{\partial E}{\partial U_i} = \sum_{j=1}^m I_{ij} ((V_{ij} - p(U_i, M_j)) M_j) - k_u U_i$$

$$-\frac{\partial E}{\partial M_j} = \sum_{i=1}^n I_{ij} ((V_{ij} - p(U_i, M_j)) U_i) - k_m M_j$$

实际上, 用户的评分与用户的习惯, 以及该物品的总体评分水平有关, 因此可以加上用户的均值与物品的均值。

$$\hat{r}_{ui} = \mu + b_i + b_u + \mathbf{q}_i^T \mathbf{p}_u$$

μ : 训练集中所有记录的评分的全局平均数。在不同网站中, 因为网站定位和销售的物品不同, 网站的整体评分分布也会显示出一些差异。比如有些网站中的用户就是喜欢打高分, 而另一些网站的用户就是喜欢打低分。而全局平均数可以表示网站本身对用户评分的影响。

b_u : 用户偏置(user bias)项。这一项表示了用户的评分习惯中和物品没有关系的那种因素。比如有些用户就是比较苛刻, 对什么东西要求都很高, 那么他的评分就会偏低, 而有些用户比较宽容, 对什么东西都觉得不错, 那么他的评分就会偏高。

b_i : 物品偏置(item bias)项。这一项表示了物品接受的评分中和用户没有什么关系的因素。比如有些物品本身质量就很高, 因此获得的评分相对都比较高, 而有些物品本身质量很差, 因此获得的评分相对都会比较低。

这个时候我们的损失函数变为:

$$E = \sum_{(u,i) \in k} (r_{ui} - \mu - b_i - b_u - \mathbf{q}_i^T \mathbf{p}_u) + \lambda (\|\mathbf{p}_u\|_2 + \|\mathbf{q}_i\|_2 + b_u^2 + b_i^2)$$

SVD与神经网络实现

SVD实现维数约化, 以及SVD与神经网络的实现。SVD与非线性维数约化, SVD与离群点检测。

非负矩阵分解

这是个有界优化问题

$$\begin{aligned} \min_{W,H} \quad & f(W, H) \equiv \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^m (V_{ij} - (WH)_{ij})^2 \\ \text{subject to} \quad & W_{ia} \geq 0, H_{bj} \geq 0, \quad \forall i, a, b, j. \end{aligned}$$

所以最直观的最简单的方法就是

方法一：

$$\begin{aligned} W^{k+1} &= \max(0, W^k - \alpha_k \nabla_W f(W^k, H^k)), \\ H^{k+1} &= \max(0, H^k - \alpha_k \nabla_H f(W^k, H^k)), \end{aligned}$$

张量分解

Tucker分解可以用于张量分解，不止是有矩阵分解，还有张量分解，那么本征值怎么推广到张量？
张量分解有什么用？

¹ [SVD在推荐系统中的应用](#)

基于内容的推荐算法

对物品特征的描述称为“内容”。协同过滤方法不需要用到物品的任何描述信息，如果我们想通过物品的特性和用户的偏好记录来进行推荐，协同过滤方法是不可以做到的，因此有了我们这节要讨论的基于内容的推荐方法。尽管这种方法必须依赖于关于物品和用户偏好的额外信息，但是它不需要巨大的用户群体或者评分记录，也就是说，只有一个用户也可以产生推荐列表。

关键

1. 建立物品画像。
2. 建立用户画像。
3. 怎么通过一组矢量来刻画物品/用户

怎么通过物品的内容描述来确定物品的相似程度，也就是文本分类问题；通过文本分类来进行物品的分类。也就是，把基于内容推荐的算法变成一个文本分类的问题，因此我们主要会利用信息检索和信息过滤领域中的技术。比如TF-IDF，代表词频和反文档频率；向量空间文档模型。

TF-IDF

TF-IDF(Term Frequency-Inverse Document Frequency)基于这样一个假设：若一个词在目标文档中出现的频率高且在其他文档中出现的频率低，那么这个词就可以用来区分出目标文档。可以通过如下方式进行计算。IDF一般去如下的表达式。

$$TF - IDF(w, D) = TF(w, D) * IDF(w, D) = f(w, D) * \log\left(\frac{C}{DF(w)}\right)$$

TF-IDF(w,D)是词w在文本D中TF-IDF分数。

TF(w,D)表示词w在文档D中的频率，也就是词w在文档D中出现的频率除以文档D中词的总数。

IDF(w,D)是词w的逆文档频率，C是该语料库中的文本数，DF(w)是语料库中含有词w的文本数目。

TF-IDF模型可以用来做为文本关键词提取，以及文本相似性计算(一般用余弦相似性)，因此可以用来做聚类，适用于无监督学习。

以电影为例，将某电影所有的文档，比如某电影所有的评语，通过TF-IDF进行矢量化，再与特定的系数相乘求和，得到这电影的综合评估向量，再与电影的基本属性结合，构建电影的物品画像。

对所有的电影建立物品画像之后，就可以计算电影之间的相似度。再建立用户画像，就可以为用户做出推荐¹。

¹. 《用户网络行为画像》牛温佳 P101 ↪

基于知识的推荐

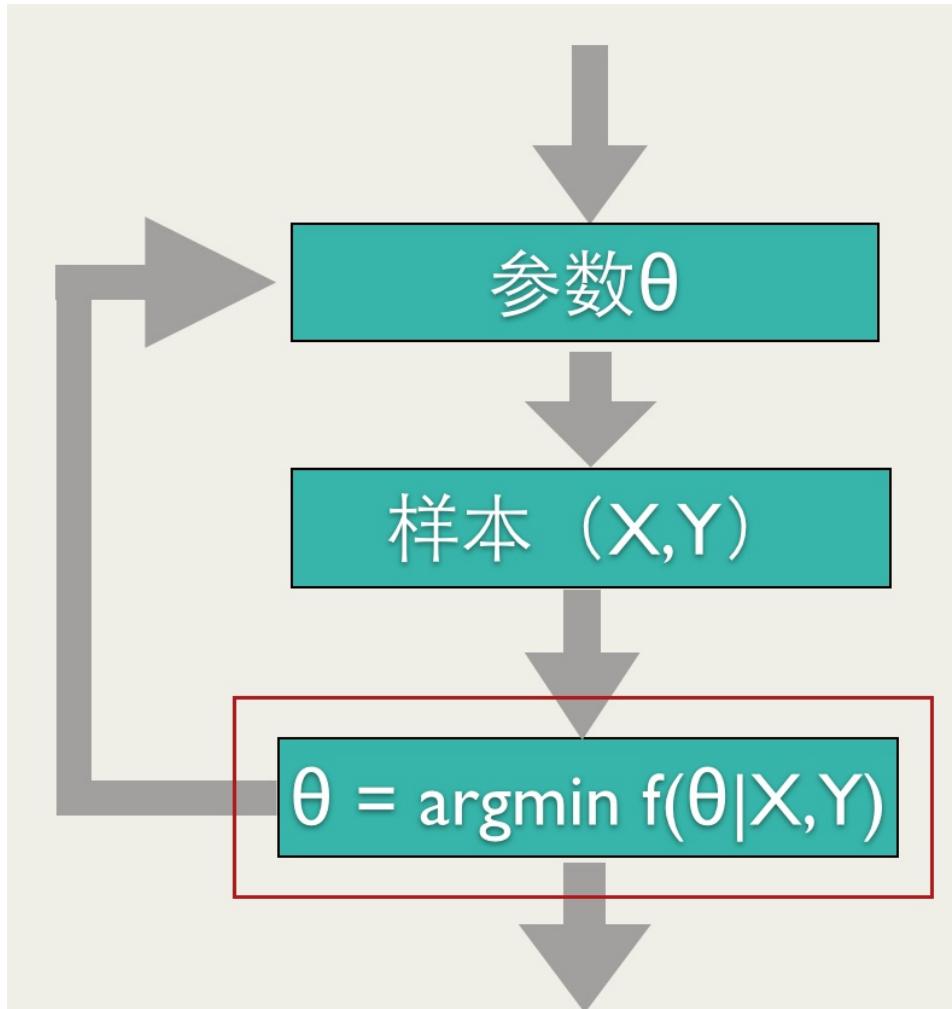
协同过滤和基于知识的推荐系统各有优势，但是在很多情况下这些方法并不是最好的。比较典型的是，我们并不会频繁的购买房屋、汽车或者计算机。所以在这种情况下，纯粹的CF系统会由于评分数据很少而效果不好。此外，时间跨度因素的作用也很重要。比如，5年前对计算机的评分内容对基于内容推荐来说就不太合适了，对汽车等更新换代比较快的产品就是这样，因为用户偏好会随着生活方式或者家庭状况的改变而改变。最后，在一些更为复杂的产品领域，比如汽车，用户经常希望明确定义他们的需求，比如：“汽车的最高价是x，汽颜色应该是黑色的”。这样需求的形式化处理并不是纯粹协同过滤和基于内容的推荐架构所擅长解决的。

基于知识的推荐系统可以帮助我们解决上面的问题。由于不需要评分数据就能推荐，因此不存在冷启动问题。推荐结果不依赖单个用户评分：要么是以用户需求和产品之间相似度的形式，要么是根据明确的规则。

基于知识推荐系统的两种基本类型是基于约束推荐和基于实例推荐。

在线学习

传统的训练方法，模型上线后，更新的周期会比较长(一般是一天，效率高的是一个小时)，这种模型上线后，一般是静态的(一段时间内不会改变)，不会与线上的状态有任何的互动，假设预测错了，只能在下一次更新的时候完成更正。online Learning会及时做出修正。因此，online learning能够更加及时地反应线上变化。



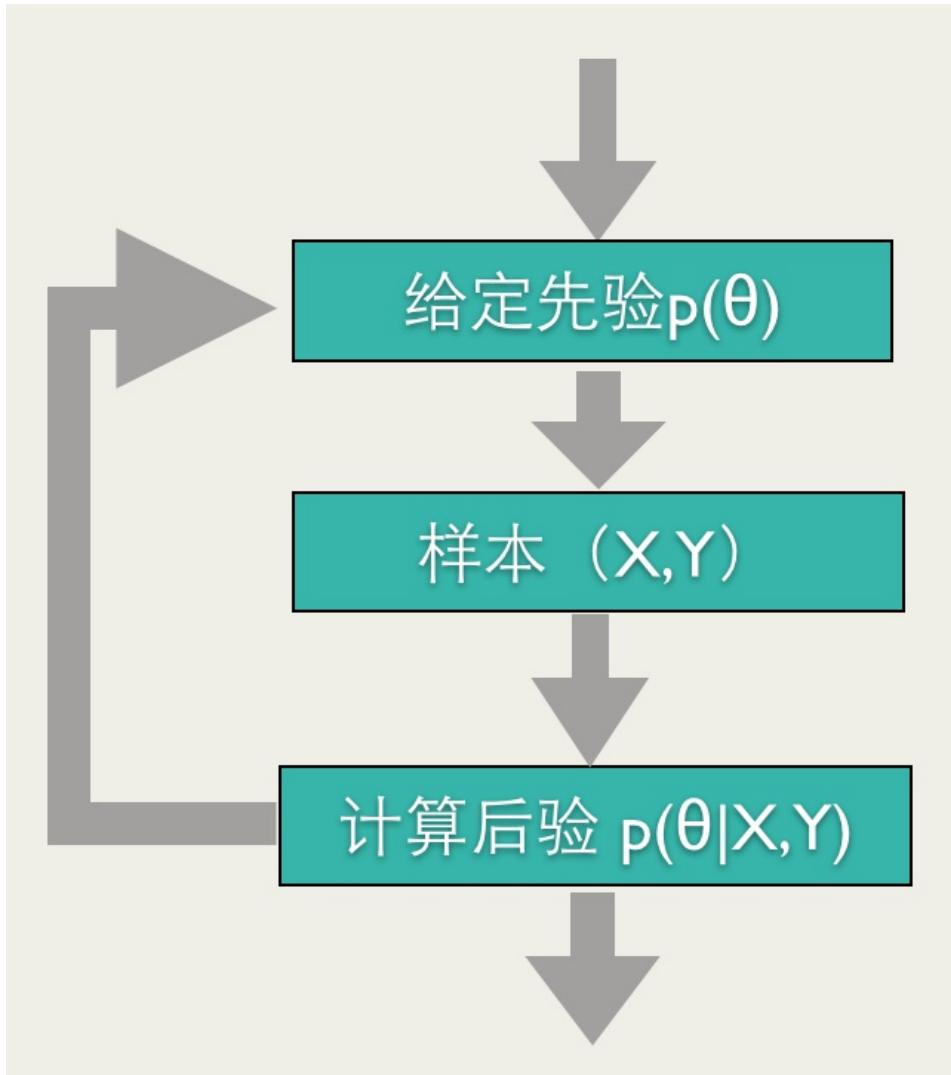
如上图所示，Online learning训练过程也需要优化一个目标函数(红框标注的)，但是和其他的训练方法不同，online learning要求快速求出目标函数的最优解，最好能有解析解。

怎么实现online learning

上面说online learning要求快速求出目标函数的最优解。要满足这个要求，一般的做法有两种：Bayesian Online Learning和Follow The Regularized Leader。下面分别介绍这两种做法的思路。

Bayesian Online Learning

贝叶斯方法的online learning的训练方法：给定参数先验，根据反馈计算后验，将其作为下一次预测的先验，然后再根据反馈计算后验，如此进行下去，如下图所示：



这和什么方法比较类似呢？

稀疏低秩分解

范数

L0,L1,L2范数¹

L0范数表示向量中非0元素的个数。如果我们对一个矩阵加L0约束，也就是要求矩阵为0的元素越多越好。也就是希望矩阵是稀疏的。

L1范数是指向量中各个元素绝对值之和。也称为“稀疏规则算子”(Lasso regularization)。加L1范数能实现权值稀疏。

由于L0范数很难优化求解(NP难问题)，而且L1范数十L0范数的最优凸近似。(n=1是保证 L^n 范数是凸函数的最小整数)。因此一般用L1来实现稀疏性。

参数稀疏性带来的好处就是特征选择与强的可解释性。

L2范数叫做Ridge回归，他能防止模型过拟合，因为他尽量让 $\|W\|_2$ 尽量小，但与L1不同，他不会让参数等于0。因为参数越小，模型越简单，因此不容易产生过拟合。

L2好处是能防止过拟合，还可以使得条件数很多的Hessian阵优化更容易，也就是使得原来不正定的矩阵变得正定，这里的矩阵一般就是Hessian矩阵。增量拉格朗日乘子法就是这么做的，L-M算法也是这么做的。

$$\min \|x\|_0 \quad \text{在一定条件下, 以概率1意义下等价} \quad \min \|x\|_1$$
$$\text{s.t. } Ax = b \quad \text{http://blog.csdn.net/u012543333/article/details/75833330} \quad \text{s.t. } Ax = b$$

核范数

核范数 $\|W\|_*$ 是指矩阵奇异值之和，也就是迹trace，英文是Nuclear Norm。它的作用就是为了实现Low-Rank(低秩)。所谓的秩就是线性方程组中独立的线性方程组数目。矩阵的秩就是矩阵的非0奇异值的个数。因此，加核范数能实现低秩矩阵。对于对角矩阵，核范数就是L1范数。

如下是核范数的两个应用。

矩阵填充

也就是推荐系统中，在User-Item矩阵中，有些用户没有购买一些items，那么我们是否该向他们推荐这些items，这在数学上是矩阵填充问题，在推荐系统中就是向从没有购买过某商品的用户推荐某商品。

我们可以用低秩约束来实现矩阵填充，因为我们的先验是，矩阵各行(列)之间是高度相关的，因此矩阵是低秩的。

鲁棒PCA

主成分分析，可以有效的找出数据中最主要的元素和结构，去除噪声与冗余，将原来的复杂数据降维，揭露隐藏在复杂数据背后的简单结构。最简单的主成分分析就是PCA了。PCA的目的就是为了找到主元，最大程度的去除冗余和噪音的干扰。

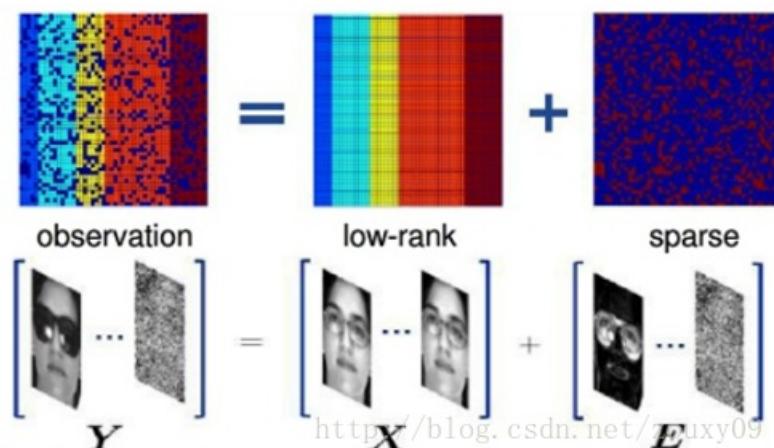
鲁棒主成分分析(Robust PCA)考虑的是这样一个问题：一般我们的数据矩阵 X 会包含结构信息，也包含噪声。那么我们可以将这个矩阵分解为两个矩阵，一个是低秩的(由于内部有一定的结构信息，造成各行和各列之间是线性相关的)，另一个是稀疏的(由于含有噪声，而噪声是稀疏的)。则鲁棒主成分分析可以写成以下的优化问题：

$$\min_{A, E} \text{rank}(A) + \lambda \|E\|_0; \text{ s.t. } X = A + E$$

与经典PCA问题一样，鲁棒PCA本质上也是寻找数据低维空间上的最佳投影问题。对于低秩数据观测矩阵 X ，假设 X 收到随机(稀疏)噪声的影响，则 X 的低秩就会破坏，使 X 变成满秩的。所以我们需要将 X 分解成包含其真实结构的低秩矩阵和稀疏噪声矩阵之和。为了找到低秩矩阵，实际上就找到了矩阵的本质低维空间。那有了PCA，为什么还需要有这个Robust PCA，Robust在哪？因为PCA假设我们的数据的噪声是高斯分布的，对于大的噪声或者严重的离群点，PCA就会被它影响，导致无法工作。而Robust PCA则不存在这个假设。它只是假设噪声是稀疏的，而不管噪声的强弱如何。由于rank和L0范数在优化上存在非凸和非光滑特性，所以我们一般将它转化成求解以下一个松弛的凸优化问题：

$$\min_{A, E} \|A\|_* + \lambda \|E\|_1; \text{ s.t. } X = A + E$$

说个应用吧。考虑同一副人脸的多幅图像，如果将每一副人脸图像看成是一个行向量，并将这些向量组成一个矩阵的话，那么可以肯定，理论上，这个矩阵应当是低秩的。但是，由于在实际操作中，每幅图像会受到一定程度的影响，例如遮挡，噪声，光照变化，平移等。这些干扰因素的作用可以看做是一个噪声矩阵的作用。所以我们可以把我们的同一个人脸的多个不同情况下的图片各自拉长一列，然后摆成一个矩阵，对这个矩阵进行低秩和稀疏的分解，就可以得到干净的人脸图像(低秩矩阵)和噪声的矩阵了(稀疏矩阵)，例如光照，遮挡等等。至于这个的用途，你懂得。



背景建模

背景建模的最简单情形是从固定摄像机的视频中分离出背景和前景。我们将视频图像序列的每一帧图像像素值拉成一个列向量，那么多个帧就是多个列向量，就组成一个观测矩阵。由于背景比较稳定，图像序列帧与帧之间具有极大的相似性，所以仅由背景像素组成的矩阵具有低秩性质。同时由于前景是移动的物体，占据的像素比例较小，故前景像素组成的矩阵具有稀疏性。视频观测矩阵就是这两种特性矩阵的叠加，因此，可以说视频背景建模实现的过程就是低秩矩阵恢复的过程。

Video D Low-rank appx. A Sparse error E

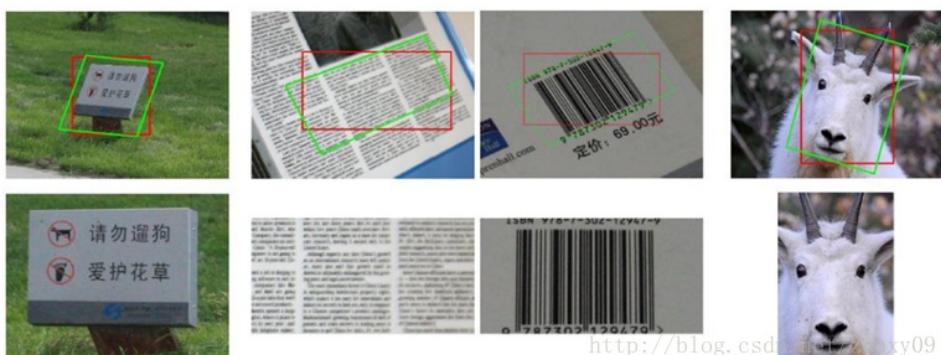


变换不变低秩纹理(TILT)

以上介绍的是针对图像的低秩逼近算法，仅仅考虑图像样本之间像素的相似性，却没有考虑到图像作为二维的像素集合，其本身具有的规律性。事实上，对未加旋转的图像，由于图像的对称性和自相似性，我们可以将其看做一个带噪声的低秩矩阵。当图像由端正发生旋转时，图像的对称性就会被破坏，也就是说各行像素间的线性相关性被破坏，因此矩阵的秩就会增加。

低秩纹理映射算法(Transform Invariant Low-Rank Textures, TILT)是一种用低秩性与噪声的稀疏性进行低秩纹理恢复的算法。它的思想是通过几何变换 τ 把D所代表的图像区域校正成正则的区域，如具有横平竖直，对称等特性，这些特性就可以通过低秩性来进行刻画。

纯背景(众数)+背景+前景：如有多个纯背景怎么办？
(群论可以排上用场了，低秩怎么与对称性建立联系？)



矩阵恢复的条件

假设从 $p \times p$ 矩阵中均匀随机地抽取 N 个元素，对于 p 维矩阵，秩为 r ，当 N 为多大时，就可以用如下的核范数松弛形式来精确的恢复该矩阵？

$$\text{minimize} \|M\|_*, \text{ 约束为: } m_{ij} = z_{ij}, (i, j) \in \Omega$$

其中 $\|M\|_*$ 是核范数，即 M 的奇异值之和。考虑到噪声，则对观测到的值直接建模并不现实，下面是一种更实际的方法：

$$\min_M \frac{1}{2} \sum_{(i,j) \in \Omega} (z_{ij} - m_{ij})^2 + \lambda \|M\|_*$$

任何方法（不仅是核范数松弛法）都需要 $N > p \log p$ 个观测值才可以准确的恢复该矩阵，即使秩为1的矩阵。

对于给定秩为 r 的 $p \times p$ 矩阵，需要大约 $O(rp)$ 个参数才能精确恢复，因为有 $O(r)$ 个奇异向量，每个向量的维度是 p 。

一般而言，合成和采样如果满足不等式：

$$N \geq Crp \log p$$

则用核范数松弛方法精确恢复矩阵的概率很高。

谱正则化²

矩阵 Z 的低秩填充，也就是求解如下问题：

$$\text{minimizerank}(M), \text{ 约束为: } m_{ij} = z_{ij}, (i, j) \in \Omega$$

上面是一个非凸目标函数，可以松弛为凸形式，原子范数是矩阵秩的凸松弛：

$$\text{minimize} \|M\|_*, \text{ 约束为: } m_{ij} = z_{ij}, (i, j) \in \Omega$$

上面没有考虑到噪声，考虑到噪声，则对观测到的值直接建模并不现实，下面是一种更实际的方法，是对上式得松弛版本：

$$\min_M \frac{1}{2} \sum_{(i,j) \in \Omega} (z_{ij} - m_{ij})^2 + \lambda \|M\|_*$$

这称为谱正则化。这种修改所得到的解 Z 并不能精确拟合观测值。但是在观测值含有噪声的情况下，这种方式能减少过拟合。

下面介绍求解上面问题的一般过程。

基于Soft-Impute的矩阵填充

首先定义几个符号，观测到的元素子集用 Ω 表示，由此定义投影算子 $P_\Omega : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}$ 为：

$$[P_\Omega(Z)]_{ij} = z_{ij}, \text{ if } (i, j) \in \Omega$$

$$[P_\Omega(Z)]_{ij} = 0, \text{ other}$$

即 P_Ω 会用0代替 Z 中的缺失值，只留下观测值。有了这个定义，就可以得到等式：

$$\sum_{(i,j) \in \Omega} (z_{ij} - m_{ij})^2 = \|P_\Omega(Z) - P_\Omega(M)\|_F^2$$

矩阵W的秩为r, 则相应的奇异值分解 $W = UDV^T$, 这里定义它的软阈值化版本:

$$S_\lambda(W) = U D_\lambda V^T, \text{ 其中 } D_\lambda = \text{diag}[(d_1 - \lambda)_+, \dots, (d_r - \lambda)_+]$$

Soft-Impute算法:

1. 初始化 $Z^{old} = 0$, 创建递减的 $\lambda_1 > \lambda_2 > \dots > \lambda_k$ 。
2. 对于每个 $k=1, 2, \dots, K$, 令 $\lambda = \lambda_k$, 并进行下面的迭代, 直到收敛:
 - i. 计算 $\hat{Z}_\lambda \leftarrow S_\lambda(P_\Omega(Z) + P_\Omega^+(Z^{old}))$
 - ii. 更新 $Z^{old} \leftarrow \hat{Z}_\lambda$
3. 输出序列 $\hat{Z}_{\lambda_1}, \dots, \hat{Z}_{\lambda_K}$

在计算中, 可以通过如下分解来优化计算:

$$P_\Omega(Z) + P_\Omega^+(Z^{old}) = (P_\Omega(Z) - P_\Omega^+(Z^{old})) + Z^{old}$$

右边第一部分是稀疏的, 第二部分是SVD的软阈值, 是低秩的。

可以证明, 这个迭代算法能收敛到问题:

$$\min_{M \in R^{m \times n}} \frac{1}{2} \|P_\Omega(Z) - P_\Omega(M)\|_F^2 + \lambda \|M\|_*$$

的解, 这是目标函数的另外一种表示。

终极目的

想办法把Krylov子空间用到矩阵计算上面去。

RIP

RIP: Restricted Isometric Property受限等容性条件。

调和分析是下一个需要看的数学方向, 因为这涉及到压缩感知这一块。调和分析起源于 Euler, Fourier 等著名科学家的研究, 主要涉及算子插值方法、极大函数方法、球调和函数理论、位势理论、奇异积分以及一般可微函数空间等。经过近200年的发展, 已经成为数学中的核心学科之一, 在偏微分方程、代数数论中有广泛的应用。

1. 参考 机器学习中的范数规则化之(二)核范数与规则项参数选择

<https://blog.csdn.net/zouxy09/article/details/24972869>

2. 《稀疏统计学习及其应用》7.3

LTR

LTR or machine-learned ranking(MLR)是运用机器学习,典型的监督,半监督或者强化学习来为信息检索系统构建排序模型。

排序学习可以在信息检索(IR),NLP, DM等领域被广泛应用,典型的应用有文献检索,专家检索系统,定义查询系统,协同过滤,问答系统,关键词提取,文档摘要还有机器翻译等。

互联网搜索方面的一个新趋势是使用机器学习方法去自动的建立评价模型 $f(q,d)$ 。

对于标注训练集,选定LTR方法,确定损失函数,以最小化损失函数为目标进行优化即可得到排序模型的相关参数,这就是学习过程。预测过程就是将待预测结果输入到学习到的排序模型中,即可以得到结果的相关得,利用该得分进行排序即可得到待预测结果的最终顺序。

CTR预估

CTR预估中GBDT与LR融合方案

Building Real World Intelligent Systems

情感分析(Sentiment analysis)常用的模型

- Bing Liu's Lexicon,具体的可以参看[这个网站](#)
- MPQA Subjectivity Lexicon
- Pattern Lexicon
- SentiWordNet Lexicon
- VADER Lexicon
- AFINN Lexicon是最简单与最流行的lexicons.

文本分类系统的搭建

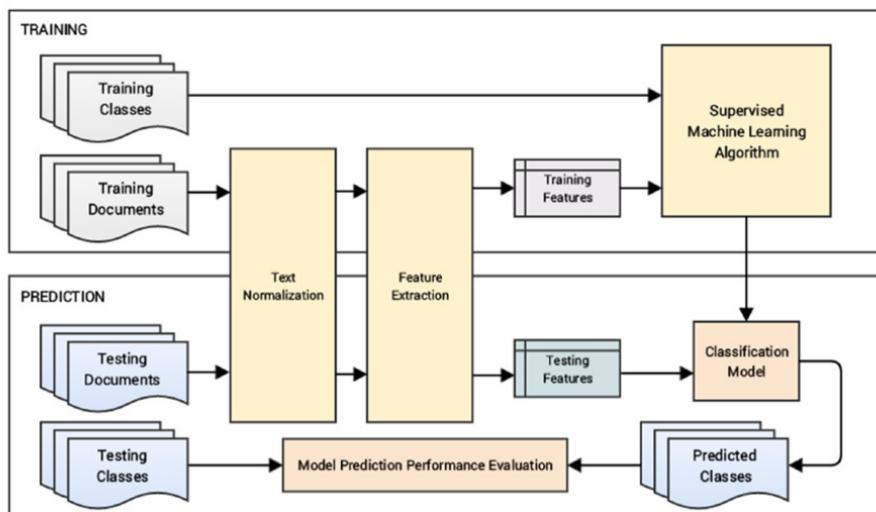


Figure 7-4. Blueprint for building an automated text classification system (Source: Text Analytics with Python, Apress 2016)

Sihouette Score

K-means找到最佳clusters数目的标准。

Kernel里面有详细的教程与讨论。

第八章 项目工作

这章主要通过Kaggle实战来提升自己的运用的能力，最主要的还是特征提取的能力。

深度学习在晶圆检测中的应用研究

要从这个项目中深挖出整个深度学习，也就是从这个项目中涉及到CNN, RNN,DNN.以及训练技巧以及ML。图像处理与信号处理。也就是要体现出深度与广度。三段经历代表深度学习，编程，最优化(机器学习)三大领域，要会发现亮点，也就是自己的特色。注意深度，附带是广度。

项目背景与目的

通过深度学习来提升光刻模型的性能。随着芯片制造中芯片节点(晶体管栅极的最小线宽(栅宽))变小，光成像过程中光的波动性越来越显著，几何光学要被波动光学所修正。光刻成像的模型也越来越复杂，但是，即使模型越来越复杂，我们通过光刻的成像模型也与显示的成像结果存在一些差距，尤其在边角处。这些由成像模型导致的错误，很大部分可以由有经验的工程师来辨别出来，这些错误区别于真实的印刷错误。我们的目的是想通过深度学习来减小我们的模型错误而保留真实的印刷错误。可以认为是把模型导致的错误当成噪声，我们需要抑制这些噪声，而把真正的印刷错误当成真实的信号来对待，我们需要保留这些信号而抑制噪声。

一个问题是：不加特殊的处理，你在抑制噪声的时候，也同时抑制了真正的信号。那你的网络要怎么处理才能保留住信号而抑制噪声？

问题的延伸

我们真正想要找的是印刷错误(真正的错误)，而模型导致的错误是虚假的错误。因此，我们可以当成一个二分类问题去求解。但是不能漏掉一个真实的错误。

背景

随着半导体工艺中，芯片中的尺寸越来越小，光学衍射效应越来越明显，单纯的几何光学(初中物理老师告诉你，光是直线传播的，这其实说的是，光是粒子，但光通过狭窄的缝隙会发生衍射效应，这时，光不是直线传播的了)以及不适用，要考虑光的衍射效应。光通过Musk就相当于一次傅里叶变化，从时域变到了频率域。再通过棱镜，又相当于一次傅里叶变换，从频率域变到时域。但棱镜的尺寸有限，因此只接收了低频波，因此，用图像处理的话来说，棱镜的作用就是一个低通滤波器。去掉了高频部分，因此投影到wafer(晶圆)上的图案不再菱角分明，而是在边界处会比较圆滑。

如果没有一些傅里叶光学的背景，你们听起来会比较费劲，但是如果你们有图像处理的背景的话，就记住把Musk上的图案刻蚀到晶圆上就是经历了两次傅里叶变换再加一个低通滤波器。

光通过单缝后，其频谱是连续的，但是经过周期性(结构)缝，其频谱就是离散的，如果为了节约棱镜的开支，就只能取很低的频谱。

光学建模是个逆问题(逆问题的例子，就是CT，通过收集到的信号来反推身体的内部结构)。(怎么把逆问题与深度学习，机器学习联系起来？)，再通俗点，它实际上和机器学习，深度学习要处理的问题是一样的，就是求解一个模型，就是train一个model。正向过程就是你已经训练好了一个模型

(取啥例子？)，然后输入一张图，看它的分类是啥，这很容易，难得是，我有一堆分类好的图片，让你训练一个model。逆问题就是一个优化问题，因此就涉及到一些优化算法，这时候考点就来了？有哪些优化算法，总结下，有两种，线性搜索方法 (linear search) 以及 trust region。但是对于这个求逆问题而言，是很复杂的，直接离散的话，参数基本在10的15次方以上，总之，这个求逆问题是很难的，即使求出来，也不会很准。因此，我们有两个研究的方向。直接通过光学图像（也就是将要刻蚀在wafer上的电路板）来反求印刷这张图像的模板 (Musk)，其实这个光刻就像拍照片，拍的物体（比如风景）就是Musk，wafer就相当于胶卷，wafer上的图像就相当于风景在胶卷上的投缘。

第一个方向是，直接通过光学图像（也就是将要刻蚀在wafer上的电路板）来反求印刷这张图像的模板 (Musk)，我们的模型就是卷积网络，因为多层网络可以用来近似任何一个函数（通用近似定理 (Universal approximation theorem, 一译万能逼近定理)）指的是：如果一个前馈神经网络具有线性输出层和至少一层隐藏层，只要给予网络足够数量的神经元，便可以实现以足够高精度来逼近任意一个在 R^n 的紧子集 (Compact subset) 上的连续函数。只要神经元足够多，再加一个激活函数），当然可以用来近似这个求逆过程。但是这条路风险很大，因此，我们有了第二条研究的路径，就是修正原有的物理模型，使得它更powerful，也就是说，我们仍然需要建立物理model，来进行求逆过程，得到一个物理的model，只是，这个model不是足够强大，会有一些缺陷，因此，我们想通过一个CNN来修正这个model，来让我们的模型更有效。这就是大致的背景。背景讲清楚了，我们就该讲我们是怎么做的了。

搭建GPU环境，装显卡驱动（显卡驱动，蓝屏），cuda，cudnn，但是发现不能使用apt-get install东西，重装了很多次系统，装了不同的ubuntu系统，从12.04到16.04的五个不同版本，折腾了一周。（因此因为公司的IP保护，无线网卡解决问题）

我们一开始做了些尝试，用了CNN，VGG，遇到的问题（图像预处理，亚像素的偏离），数据集不够，因为真实数据都是机密 (IP)，不可以轻易拿到。自己去切割图像，通过有经验的人工来做分类。再通过翻转这些操作来增加数据集。此外，数据集也是很讲究的，你得保证同一数据集中的物理参数，焦距，偏振方向都是一致的，焦距不同，得到的图像模糊程度不同。

很长一段时间，我们都在用用记忆卷积，反卷积的深度神经网络，都没发现很好的效果。后来使用了pre-trained model，因为我们的数据量少（30000张图），但是经过CNN处理过的图像，并不比没有处理的图像好，这就尴尬了，因此，我们一直加深网络。

工具

Quardo 4000; Tensorflow

图像预处理

主要困难还是在于图像的收集，我们可以通过降低判定real-false的阈值来增加模型错误的图像数量，我们还需要人工来判定哪些错误是真实的印刷错误，哪些是由模型导致的错误。而人工标注是一件很费时间的事情。

我们输入的是我们模型产生的图像，而我们的target是真是拍摄到的芯片图像。这些都是数字图像，我们必须确保输入与输出的图像是同一个地方的，也就是图像必须真正的对齐，不能差任何一点，半个pixel也不能（差一点就会导致两图相减在边缘地方出现巨变），但是最后发现我们的图像差了半个pixel，我们必须通过信号处理的方法来对齐我们的输入与输出图像。这些是数据预处理过程中遇到的真实问题。

数据集

图像大小:48*48,

数据集:一共31400张图片, 训练集90%, 一共28260张图, 测试集10%, 一共3140张图

batch size大小5.

一个epoch(训练完28260张图)的时间是600s,即10分钟左右。

```
#batch_size =5
#learn_rate = 0.0001
net = tflearn.input_data(shape=[None, 48, 48, 1])
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 2, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 3, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 2, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 1, 64)
net = tflearn.conv_2d(net, 1, 1)
Training Step: 1000 | total loss: 6.56844 | time: 118.655s
| Adam | epoch: 001 | loss: 6.56844 - acc: 0.7033 | val_loss: 8.57394 - val_acc: 0.6723 -- it
er: 05000/28260
--
Training Step: 2000 | total loss: 6.26063 | time: 234.816s
| Adam | epoch: 001 | loss: 6.26063 - acc: 0.6907 | val_loss: 8.11802 - val_acc: 0.6762 -- it
er: 10000/28260
--
Training Step: 3000 | total loss: 5.61156 | time: 351.264s
| Adam | epoch: 001 | loss: 5.61156 - acc: 0.7085 | val_loss: 7.88475 - val_acc: 0.6794 -- it
er: 15000/28260
--
Training Step: 4000 | total loss: 6.61628 | time: 469.786s
| Adam | epoch: 001 | loss: 6.61628 - acc: 0.7000 | val_loss: 9.18316 - val_acc: 0.6804 -- it
er: 20000/28260
--
Training Step: 5000 | total loss: 5.81782 | time: 586.015s
| Adam | epoch: 001 | loss: 5.81782 - acc: 0.6987 | val_loss: 7.58499 - val_acc: 0.6852 -- it
er: 25000/28260
--
Training Step: 6000 | total loss: 5.99239 | time: 50.356ss
| Adam | epoch: 002 | loss: 5.99239 - acc: 0.7008 | val_loss: 6.78956 - val_acc: 0.6852 -- it
er: 01740/28260
```

网络选择

从一开始的普通CNN

```
#batch_size =5
#learn_rate = 0.0001
net = tflearn.input_data(shape=[None, 48, 48, 1])
net = tflearn.conv_2d(net, 64, 1)
```

```

net = tflearn.residual_block(net, 2, 64)
net = tflearn.residual_block(net, 1, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 1, 64)
net = tflearn.residual_block(net, 2, 64)
net = tflearn.conv_2d(net, 1, 1)
Training Step: 1000 | total loss: 6.41052 | time: 88.615s

```

到pre-trained VGG

熟知VGG16的人知道, VGG16输入的图像是3通道大小是224x224的图像, 因此一般使用VGG也需要相同大小的输入, VGG16经过5次大小为2的pooling, 最终变成 $7 \times 7 \times 512 = 25088$, 7是图像大小, 512是通道数目, 再后面就是3个全连接层, 在第五个pooling层与第一个全连接层之间, 我们必须保证图像的大小, 而在此之前, 所有的参数只是一些特征(卷积核)的参数, 无关图像大小, 因此我们可以把VGG的前面提取特征的部分复用到任何其它数据集, 而不管输入图像大小, 只要保证图像大小大于最大核的尺寸, 当然, 有些pooling层就没有了, 你们可能认为这样做之后会使得VGG的性能降低, 这我不好否认, 不过, 你确实可以通过增减pooling层数目, 使得VGG能适用于输入图像尺寸任意的图像, 也可以适用于单通道, 因为你可以把一个通过重复三次, 来近似RGB图像。tensorlayer恢复以前模型中有一个函数就是tl.files.assign_params, 它可以用来指定一些层的参数, Assign the given parameters to the TensorLayer network. 这可以用复用以前模型提取的特征。我们这里只用了前面两个卷积层的参数。

```

# 1st parameter
tl.files.assign_params(sess, [load_params[0]], network)
# the first three parameters
tl.files.assign_params(sess, load_params[:3], network)

```

```

#####Extract VGG16 weights#####
import numpy as np
weight_fileVGG =
'/home/lipp/Work/tensorlayer/LoadTrainedWeights/UseVGGWeights/vgg16_weights
.npz'
weightsVGG = np.load(weight_fileVGG)
keysVGG = sorted(weightsVGG.keys())
VGG2EUV = []
for i, k in enumerate(keysVGG):
    print(i, k, np.shape(weightsVGG[k]))
    VGG2EUV+=[weightsVGG[k]]
np.savez('WeightsVGG2EUV.npz',VGG2EUV)

#####
#Using the pre-trained weights#####
weight_fileVGG = '/home/lipp/Work/tensorlayer/LoadTrainedWeights/UseVGGWeights/Wei
ghtsVGG2EUV.npz'
weightsVGG = np.load(weight_fileVGG)
AllWeightEUV = weightsVGG['arr_0']
tl.files.assign_params(sess, AllWeightEUV[0:4], network)
#####the first 2 conv layers use the pre-trained weights#####

```

VGG16

VGG16的名字由来是因为卷积层加上全连接层一共16层，5个pooling层不计算在内，因为pooling层没有权重参数。根据每层参数数目与参数类型float32 = 4 byte,因此整个网络的参数大小是500多M。

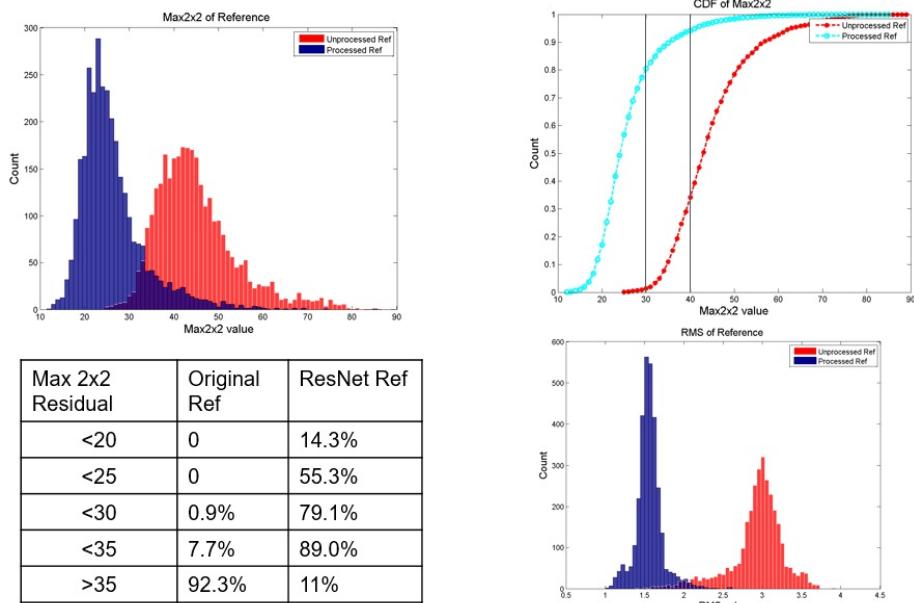
```
0 conv1_1_W (3, 3, 3, 64) float32
1 conv1_1_b (64,) float32
2 conv1_2_W (3, 3, 64, 64) float32
3 conv1_2_b (64,) float32
4 conv2_1_W (3, 3, 64, 128) float32
5 conv2_1_b (128,) float32
6 conv2_2_W (3, 3, 128, 128) float32
7 conv2_2_b (128,) float32
8 conv3_1_W (3, 3, 128, 256) float32
9 conv3_1_b (256,) float32
10 conv3_2_W (3, 3, 256, 256) float32
11 conv3_2_b (256,) float32
12 conv3_3_W (3, 3, 256, 256) float32
13 conv3_3_b (256,) float32
14 conv4_1_W (3, 3, 256, 512) float32
15 conv4_1_b (512,) float32
16 conv4_2_W (3, 3, 512, 512) float32
17 conv4_2_b (512,) float32
18 conv4_3_W (3, 3, 512, 512) float32
19 conv4_3_b (512,) float32
20 conv5_1_W (3, 3, 512, 512) float32
21 conv5_1_b (512,) float32
22 conv5_2_W (3, 3, 512, 512) float32
23 conv5_2_b (512,) float32
24 conv5_3_W (3, 3, 512, 512) float32
25 conv5_3_b (512,) float32
26 fc6_W (25088, 4096) float32
27 fc6_b (4096,) float32
28 fc7_W (4096, 4096) float32
29 fc7_b (4096,) float32
30 fc8_W (4096, 1000) float32
31 fc8_b (1000,) float32
```

TOTAL memory: 24M 4 bytes ~= 93MB / image (only forward! ~2 for bwd)

TOTAL params: 138M parameters

效果

通过我们的ResidualNN, 我们能很大的降低我们模型得到的图像与实际拍摄到的图像之间的差别, 也就是在Max2x2。(RMS怎么定义忘记了?)



原因是, 我们输入的图像已经很接近输出的图像, 因此, 我们只需要训练残差就可以了。

图像与信号处理

Residual NN

Residual NN提出来的背景:

随着网络变深, 训练误差与测试误差得提高了。这是违反我们的训练的初衷的, 因为即使我们把26层后面的网络变成恒等映射, 效果也不会变差。

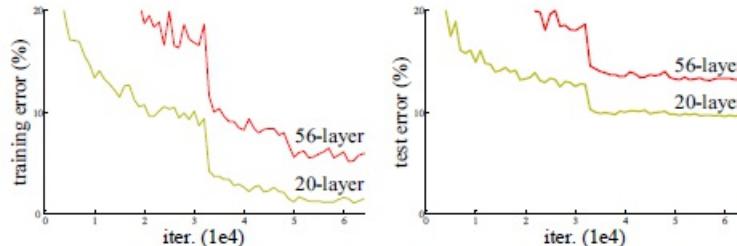


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

存在这种随着网络层数增加，会出现如下两个问题：

1. 梯度消失或者爆炸，导致训练难以收敛。这个问题可以通过normalized initialization 和 intermediate normalization layers解决。
2. 随着深度增加，模型的训练误差与测试误差会迅速下滑，这不是overfit造成的。这种现象在 CIFAR-10和ImageNet中都有提及。

Residual-NN的实现

参考He Kaiming的这篇文章

Residual Block的设计如下：

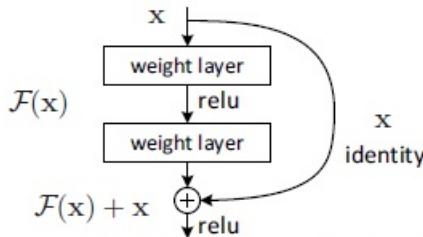


Figure 2. Residual learning: a building block.

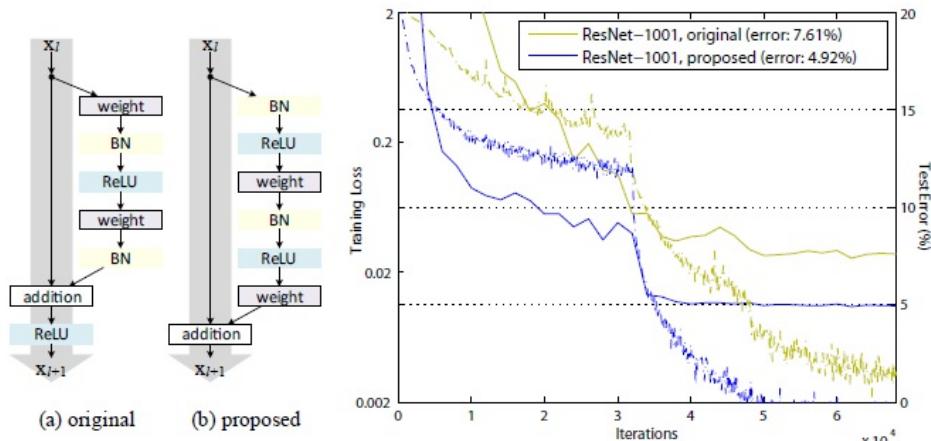
对于每一个Residual Units：

$$y_l = h(x_l) + F(x_l, W_l)$$

$$x_{l+1} = f(y_l)$$

其中函数h一般取恒等映射 $h(x_l) = x_l$ ，实验发现f取恒等映射能实现最快的误差下降与最低的训练误差；f是激活函数，取ReLU。F的残差函数。

一个推荐的网络结构：



如果激活函数f取恒等映射 $y_l = f(y_l)$ ，则有：

$$x_{l+1} = x_l + F(x_l, W_l)$$

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i, W_i)$$

因此对于误差反向传播：

$$\frac{\partial \epsilon}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L} \frac{\partial x_L}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L} \left(1 + \frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F(x_i, W_i) \right)$$

因为 $\frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F(x_i, W_i)$ 不会常等于 -1，因此梯度不会消失，即使当权重非常小的时候。

因此信号对于正向与反向传播，可以直接的从一个单元传到另外一个单元。条件就是 skip connection 函数 h ，激活函数 f 是恒等映射。函数 h 是恒等映射时保证梯度不消失或者爆炸的关键。他们使用的 mini-batch 是 128，用了 2 个 GPUs，权重衰减是 0.0001，动量是 0.9，权重初始化了。详细的可以看上面给的论文链接。权重每 30 个 epoch 变成原来的十分之一。

Residual NN 模型效果：

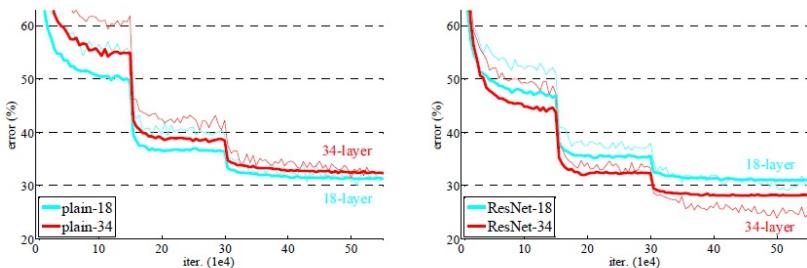


Figure 4. Training on ImageNet. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Resi-NN 能解决以往随着模型加深训练与测试误差变大的现象。与其它网络的结果比较。

CIFAR-10	error (%)	CIFAR-100	error (%)
NIN [13]	8.81	NIN [13]	35.68
DSN [14]	8.22	DSN [14]	34.57
FitNet [15]	8.39	FitNet [15]	35.04
Highway [7]	7.72	Highway [7]	32.39
ELU [12]	6.55	ELU [12]	24.28
FitResNet, LSUV [16]	5.84	FitNet, LSUV [16]	27.66
ResNet-110 [1] (1.7M)	6.61	ResNet-164 [1] (1.7M)	25.16
ResNet-1202 [1] (19.4M)	7.93	ResNet-1001 [1] (10.2M)	27.82
ResNet-164 [ours] (1.7M)	5.46	ResNet-164 [ours] (1.7M)	24.33
ResNet-1001 [ours] (10.2M)	4.92 (4.89 ± 0.14)	ResNet-1001 [ours] (10.2M)	22.71 (22.68 ± 0.22)
ResNet-1001 [ours] (10.2M) [†]	4.62 (4.69 ± 0.20)		

模型的改进 (Simons)

是否可以通过逐渐加深网络来实现浅层的网络而具有强大的功能。具体步骤就是：

1. 一个浅层的网络进行 end-end 的训练，训练到一定程度的时候，进行第二步
2. 沿用第一步的权重，通过以 Residual Block 来微调网络，减小误差，优化方法是 SGD，但是我们要保证，每增加一层网络，总的模型的效果会更好。

3. 循环第二步，直到结果符合我们期望的为止(误差到noise层)

理论上，这类似于机器学习中的Adaboost，是否可以分析它的误差会指数衰减？希望能像ML算法一样，能有数学理论来分析深度学习。通过理论指导，像搭积木一样来构建深度神经网络。

Fast项目

要从这个项目中体现出你对语言的驾驭能力，也就是编程能力。各大语言的区别，主要是C++，Java与python。OOP，还有就是模式设计这些。

C++基础

1. 指针与引用
2. C++的OOP
3. C++的继承与多态
4. C++与Java的最大区别
5. 智能指针

Windows编程

1. Windows的消息机制
2. 多线程与同步
3. 进程间通讯
4. Socket

软件架构

1. 泛型编程
2. STL--标准模板库
3. 模式设计

算法与数据结构

1. 数据结构
2. 排序算法

Miura项目

要从这个项目中你对这最优化问题的理解与实践。也就是能对不同的最优化算法能如数家珍，并能知道其适用场景。从最优化扯到ML, DL。还是多与机器学习靠拢，这就形成自己的三个方面。

最优化

一阶方法--梯度下降法

1. 随机梯度下降法
2. 动量方法
3. Adagrad
4. Adam
5. 特殊的一阶方法--共轭梯度法

二阶方法

1. 牛顿法
2. quasi-Netwon法
 - i. BFGS
 - ii. DFP

线性搜索

Trust-region方法

1. L-M算法
2. Dog-Leg算法

约束最优化问题

1. 拉格朗日乘子理论
2. 罚函数法
3. 对偶性与方法

Titanic

Variable Description

Survived: Survived (1) or died (0)
Pclass: Passenger's class
Name: Passenger's name
Sex: Passenger's sex
Age: Passenger's age
SibSp: Number of siblings/spouses aboard
Parch: Number of parents/children aboard
Ticket: Ticket number
Fare: Fare
Cabin: Cabin
Embarked: Port of embarkation

常用的数据清洗技巧

导入数据

```
train = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\train.csv")
test = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\test.csv")
test_Y = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\gender_submission.csv")
```

导入库函数

```
import warnings
warnings.filterwarnings('ignore')
# Handle table-like data and matrices
import numpy as np
import pandas as pd

# Modelling Algorithms
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier , GradientBoostingClassifier

# Modelling Helpers
from sklearn.preprocessing import Imputer , Normalizer , scale
from sklearn.cross_validation import train_test_split , StratifiedKFold
```

```

from sklearn.feature_selection import RFECV

# Visualisation
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
import seaborn as sns

# Configure visualisations
%matplotlib inline
mpl.style.use('ggplot')
sns.set_style('white')
pylab.rcParams[ 'figure.figsize' ] = 8 , 6

```

数据的统计属性

```

train.shape ##数据的维度
train.head ##显示所有数据

```

相关系数与统计属性

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
PassengerId	1.000000	-0.005007	-0.035144	0.036847	-0.057527	-0.001652	0.012658
Survived	-0.005007	1.000000	-0.338481	-0.077221	-0.035322	0.081629	0.257307
Pclass	-0.035144	-0.338481	1.000000	-0.369226	0.083081	0.018443	-0.549500
Age	0.036847	-0.077221	-0.369226	1.000000	-0.308247	-0.189119	0.096067
SibSp	-0.057527	-0.035322	0.083081	-0.308247	1.000000	0.414838	0.159651
Parch	-0.001652	0.081629	0.018443	-0.189119	0.414838	1.000000	0.216225
Fare	0.012658	0.257307	-0.549500	0.096067	0.159651	0.216225	1.000000

关联系数, 分布, 类别图

```

def plot_correlation_map( df ):
    corr = train.corr()
    _ , ax = plt.subplots( figsize =( 12 , 10 ) )

```

```

cmap = sns.diverging_palette( 220 , 10 , as_cmap = True )
_ = sns.heatmap(
    corr,
    cmap = cmap,
    square=True,
    cbar_kws={ 'shrink' : .9 },
    ax=ax,
    annot = True,
    annot_kws = { 'fontsize' : 12 }
)

def plot_distribution( df , var , target , **kwargs ):
    row = kwargs.get( 'row' , None )
    col = kwargs.get( 'col' , None )
    facet = sns.FacetGrid( df , hue=target , aspect=4 , row = row , col = col )
    facet.map( sns.kdeplot , var , shade= True )
    facet.set( xlim=( 0 , df[ var ].max() ) )
    facet.add_legend()

def plot_categories(fd, cat, target, **kwargs):
    row = kwargs.get('row', None)
    col = kwargs.get('col', None)
    facet = sns.FacetGrid(fd, row = row, col = col)
    facet.map(sns.barplot, cat, target)
    facet.add_legend()

plot_correlation_map(train)
plot_distribution( train , var = 'Age' , target = 'Survived' , row = 'Sex' )
plot_categories(train, cat = 'Embarked', target = 'Survived')

```

plot_correlation_map(train)



类标签数字化

```
sex = pd.Series( np.where( train.Sex == 'male' , 1 , 0 ) , name = 'Sex' )
```

通过pandas产生一个新的数组，它是把原来sex中的male转化成1，其它的sex类别转化成0。

对类标签添加前缀

```
embarked = pd.get_dummies( train.Embarked , prefix='Embarked' )
```

原来Embarked的类标签是C, Q, S。现在添加了前缀Embarked_，并且把原来的一个类分成了三个类。

```
In [47]: embarked.head()
```

```
Out[47]:
```

	Embarked_C	Embarked_Q	Embarked_S
0	0	0	1
1	1	0	0
2	0	0	1
3	0	0	1
4	0	0	1

缺失值填充

通过均值来填充

```
imputed = pd.DataFrame()
imputed[ 'Age' ] = train.Age.fillna( train.Age.mean() )
imputed[ 'Fare' ] = train.Fare.fillna( train.Fare.mean() )
imputed[ 'Pclass' ] = train.Pclass.fillna( train.Pclass.mean() )
```

数据拼接

```
DataSet = pd.concat([imputed, embarked, sex, train.Pclass, train.SibSp, train.Parch],axis = 1)
```

DataSet.shape就是(891,10)

构造训练数据

```
train_X = DataSet[0:891]
train_Y = train.Survived
```

模型选择与训练

```
model = RandomForestClassifier(max_depth=3, max_features='auto',n_estimators=100)
model.fit(train_X, train_Y)
print model.score(train_X, train_Y),model.score(test_X, test_Y)

0.8305274971941639 0.8851674641148325
```

当你选定一个模型后，就根据这个模型，进行相应的参数输入，最基本的包括输入(X,Y)，其它的就是模型参数的设定，这个你可以根据sklearn的API进行设置。当然有目的性的调参考验的就是你的理论功底了。

总的效果

```
import warnings
warnings.filterwarnings('ignore')
# Handle table-like data and matrices
import numpy as np
import pandas as pd

from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier , GradientBoostingClassifier

# Modelling Helpers
from sklearn.preprocessing import Imputer , Normalizer , scale
from sklearn.cross_validation import train_test_split , StratifiedKFold
from sklearn.feature_selection import RFECV
import xgboost as xgb

train = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\train.csv")
test = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\test.csv")
test_Y = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\gender_submission.csv")

imputed_train = pd.DataFrame()
imputed_train[ 'Age' ] = train.Age.fillna( train.Age.mean() )
imputed_train[ 'Fare' ] = train.Fare.fillna( train.Fare.mean() )
imputed_train[ 'Pclass' ] = train.Pclass.fillna( train.Pclass.mean() )
embarked_taint = pd.get_dummies(train.Embarked, prefix = 'Embarked')
sex_train= pd.Series( np.where( train.Sex == 'male' , 1 , 0 ) , name = 'Sex' )
DataSet_train = pd.concat([imputed_train, embarked_taint, sex_train, train.Pclass, train.SibSp, train.Parch],axis = 1)

imputed_test = pd.DataFrame()
imputed_test[ 'Age' ] = test.Age.fillna( train.Age.mean() )
imputed_test[ 'Fare' ] = test.Fare.fillna( train.Fare.mean() )
imputed_test[ 'Pclass' ] = test.Pclass.fillna( train.Pclass.mean() )
embarked_test = pd.get_dummies(test.Embarked, prefix = 'Embarked')
sex_test = pd.Series( np.where( test.Sex == 'male' , 1 , 0 ) , name = 'Sex' )
DataSet_test = pd.concat([imputed_test, embarked_test, sex_test, test.Pclass, test.SibSp, test.Parch],axis = 1)

test_X = DataSet_test[0:418]
train_X = DataSet_train[0:891]
train_Y = train.Survived
train_X.shape,train_Y.shape,test_X.shape,test_Y.shape
```

最终结果

使用SVM在Titanic数据上，训练集效果依次是线性SVM, GBDT,LR,RF,DT,KNN。最诡异的是，有事测试集上准确率比训练集上高。

对于随机森林RF，会发现，一开始随着树的深度增加，RF整体的准确率会上升，也就是说，RF需要准确性高的(或者说bias小的)分类器做基函数，对于GBDT，发现随深度增加，准确率下降，也就是GBDT需要深度浅，variance很小的树作为分类器，这符合他们的原理。即RF基于bias小的基分类器，通过增加树的数目来降低variance，boosting基于variance小的基分类器，通过boost来降低bias。

但是有个问题是，RF在100棵树时效果最好，选择1000, 10000都没有100的效果好。

```
In [78]: model = GaussianNB()
model.fit(train_X, train_Y)
print "train score = ", Model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score = 0.7710437710437711
test score = 0.7870813397129187

In [79]: model = DecisionTreeClassifier()
model.fit(train_X, train_Y)
print "train score = ", Model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score = 0.9820426487093153
test score = 0.8110047846889952

In [81]: model = RandomForestClassifier(max_depth=6, min_samples_leaf = 1, max_features='auto',n_estimators=100)
model.fit(train_X, train_Y)
print "train score = ", Model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score = 0.8664421997755332
test score = 0.8947368421052632

In [77]: model = LogisticRegression()
model.fit(train_X, train_Y)
print "train score = ", Model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score = 0.8024691358024691
test score = 0.9497607655502392

In [80]: model = GradientBoostingClassifier(max_depth=1,min_samples_leaf=3)
model.fit(train_X, train_Y)
print "train score = ", Model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score = 0.8148148148148148
test score = 0.9784688995215312

In [82]: model = SVC(kernel = 'linear')
model.fit(train_X, train_Y)
print "train score = ", model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score = 0.7867564534231201
test score = 1.0
```

模型融合

对多个模型的结果取平均(投票)，作为最终的结果。比如拿SVM, LR, RF, GBDT的平均结果作为最终结果。

用几个模型筛选出较为重要的特征

```
def get_top_n_features(titanic_train_data_X, titanic_train_data_Y, top_n_features):
    # 随机森林
    rf_est = RandomForestClassifier(random_state=42)
    rf_param_grid = {'n_estimators': [500], 'min_samples_split': [2, 3], 'max_depth': [20]}
    rf_grid = model_selection.GridSearchCV(rf_est, rf_param_grid, n_jobs=25, cv=10, verbose
```

```

se=1)
rf_grid.fit(titanic_train_data_X,titanic_train_data_Y)
#将feature按Importance排序
feature_imp_sorted_rf = pd.DataFrame({'feature': list(titanic_train_data_X), 'importance': rf_grid.best_estimator_.feature_importances_}).sort_values('importance', ascending=False)
)
features_top_n_rf = feature_imp_sorted_rf.head(top_n_features)['feature']
print('Sample 25 Features from RF Classifier')
print(str(features_top_n_rf[:25]))


# AdaBoost
ada_est = ensemble.AdaBoostClassifier(random_state=42)
ada_param_grid = {'n_estimators': [500], 'learning_rate': [0.5, 0.6]}
ada_grid = model_selection.GridSearchCV(ada_est, ada_param_grid, n_jobs=25, cv=10, verbose=1)
ada_grid.fit(titanic_train_data_X, titanic_train_data_Y)
#排序
feature_imp_sorted_ada = pd.DataFrame({'feature': list(titanic_train_data_X), 'importance': ada_grid.best_estimator_.feature_importances_}).sort_values('importance', ascending=False)
)
features_top_n_ada = feature_imp_sorted_ada.head(top_n_features)['feature']


# ExtraTree
et_est = ensemble.ExtraTreesClassifier(random_state=42)
et_param_grid = {'n_estimators': [500], 'min_samples_split': [3, 4], 'max_depth': [15]}
)
et_grid = model_selection.GridSearchCV(et_est, et_param_grid, n_jobs=25, cv=10, verbose=1)
et_grid.fit(titanic_train_data_X, titanic_train_data_Y)
#排序
feature_imp_sorted_et = pd.DataFrame({'feature': list(titanic_train_data_X), 'importance': et_grid.best_estimator_.feature_importances_}).sort_values('importance', ascending=False)
)
features_top_n_et = feature_imp_sorted_et.head(top_n_features)['feature']
print('Sample 25 Features from ET Classifier:')
print(str(features_top_n_et[:25]))


# 将三个模型挑选出来的前features_top_n_et合并
features_top_n = pd.concat([features_top_n_rf, features_top_n_ada, features_top_n_et], ignore_index=True).drop_duplicates()

return features_top_n

```

Customer Segmentation

制定简单的方案花不了多少时间，尤其在具有理论基础的前提下。花时间的是怎么用代码去实现你的方案。只有在对语言(这个工具)非常熟悉的时候你才能快速的实现你的demo。

问题与技术路线

1. 要处理的问题是什么?
 - i. 客户分割
2. 要实现什么结果?
 - i. 把客户分割成不同的几个类别
3. 数据要怎么处理?
 - i. 数据清洗(去除错误的项目, 比如购物数量小于0, 无意义的项目, 比如用户ID是nan项)
 - ii. 以客户的ID作为标识, 来对客户进行分类
4. 怎么实现我们的目的?
 - i. 通过客户的三个特征(RFM, Recency, Frequency, Monetary Value), 再用K-means进行聚类
 - ii. Recency:客户最近多长时间购物过
 - iii. Frequency:客户购物的频率
 - iv. Monetary value:客户一共花了多少钱

groupby("CustomerID").sum().reset_index() 的作用就是根据CustomerID进行聚类, 再对同一CustomerID的amount进行求和。

```
customer_monetary_val = cs_df[['CustomerID', 'amount']].groupby("CustomerID").sum().reset_index()
customer_history_df = customer_history_df.merge(customer_monetary_val, how='outer')
customer_freq = cs_df[['CustomerID', 'amount']].groupby("CustomerID").count().reset_index()
customer_freq.rename(columns={'amount': 'frequency'}, inplace=True)
customer_history_df = customer_history_df.merge(customer_freq, how='outer')

print cs_df.shape, '\n', customer_history_df.head(5)
(354345, 11)
   CustomerID  recency    amount  frequency
0      12346.0    326.0  77183.60         1
1      12747.0     2.0   4196.01        103
2      12748.0     1.0   33719.73       4596
3      12749.0     4.0   4090.88        199
4      12820.0     3.0    942.34         59
```

对数化这是那个指标

```
from sklearn import preprocessing
customer_history_df['recency_log'] = customer_history_df['recency'].apply(math.log)
customer_history_df['amount_log'] = customer_history_df['amount'].apply(math.log)
```

```
customer_history_df['frequency_log'] = customer_history_df['frequency'].apply(math.log)
feature_vector = ['recency_log', 'amount_log', 'frequency_log']
X_subset = customer_history_df[feature_vector].as_matrix()
scaler = preprocessing.StandardScaler().fit(X_subset)
X_scaled = scaler.transform(X_subset)
```

要注意，如下两种表达方式是一样的。

```
recency_log_1 = customer_history_df['recency'].apply(math.log)
recency_log_2 = customer_history_df.recency.apply(math.log)
```

然后做直方图，也就是统计amount_log在每个值区间的出现的概率。用n,bins, patchs接收返回的数据。

```
X = customer_history_df.amount_log
n, bins, patchs = plt.hist(X,1000,facecolor='green',alpha = 0.75)
plt.xlabel('Log of Sales Amount')
plt.ylabel('Probability')
plt.title(r'Histogram of Log Transformed Customer Monetary Value')
plt.grid(True)
plt.show()
```

画RFM图，进行聚类。

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111,projection='3d')

xs = customer_history_df.recency_log
ys = customer_history_df.frequency_log
zs = customer_history_df.amount_log
ax.scatter(xs,ys,zs,s=5)

ax.set_xlabel('Recency')
ax.set_ylabel('Frequency')
ax.set_zlabel('Monetary')
```

用KMeans来进行聚类

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
import matplotlib.cm as cm

X = X_scaled

cluster_centers = dict()

for n_clusters in range(3,6,2):
```

```

fig, (ax1, ax2) = plt.subplots(1, 2)
#ax2 = plt.subplot(111, projection='3d')
fig.set_size_inches(18, 7)
ax1.set_xlim([-0.1, 1])
ax1.set_ylim([0, len(X) + (n_clusters + 1) * 10])

clusterer = KMeans(n_clusters=n_clusters, random_state=10)
cluster_labels = clusterer.fit_predict(X)

silhouette_avg = silhouette_score(X, cluster_labels)
cluster_centers.update({n_clusters :{
                            'cluster_center':clusterer.cluster_centers_,
                            'silhouette_score':silhouette_avg,
                            'labels':cluster_labels}
                        })

sample_silhouette_values = silhouette_samples(X, cluster_labels)
y_lower = 10
for i in range(n_clusters):
    ith_cluster_silhouette_values = \
        sample_silhouette_values[cluster_labels == i]

    ith_cluster_silhouette_values.sort()

    size_cluster_i = ith_cluster_silhouette_values.shape[0]
    y_upper = y_lower + size_cluster_i

    color = cm.nipy_spectral(float(i) / n_clusters)
    ax1.fill_betweenx(np.arange(y_lower, y_upper),
                      0, ith_cluster_silhouette_values,
                      facecolor=color, edgecolor=color, alpha=0.7)

    ax1.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
    y_lower = y_upper + 10 # 10 for the 0 samples

ax1.set_title("The silhouette plot for the various clusters.")
ax1.set_xlabel("The silhouette coefficient values")
ax1.set_ylabel("Cluster label")
ax1.axvline(x=silhouette_avg, color="red", linestyle="--")
ax1.set_yticks([])
ax1.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
colors = cm.nipy_spectral(cluster_labels.astype(float) / n_clusters)
feature1 = 1
feature2 = 2
ax2.scatter(X[:, feature1], X[:, feature2], marker='.', s=30, lw=0, alpha=0.7,
            c=colors, edgecolor='k')

centers = clusterer.cluster_centers_
ax2.scatter(centers[:, feature1], centers[:, feature2], marker='o',
            c="white", alpha=1, s=200, edgecolor='k')
for i, c in enumerate(centers):
    ax2.scatter(c[feature1], c[feature2], marker='%' % i, alpha=1,
                s=50, edgecolor='k')
ax2.set_title("The visualization of the clustered data.")
ax2.set_xlabel("Feature space for the 1st feature i.e. monetary value")

```

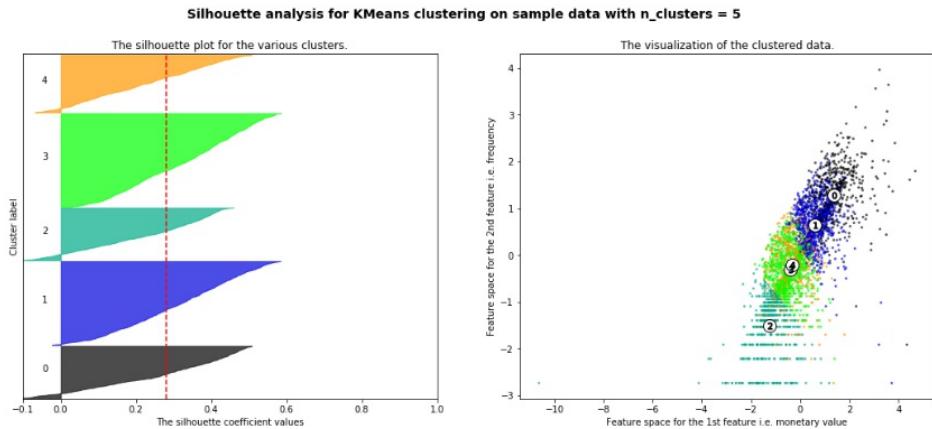
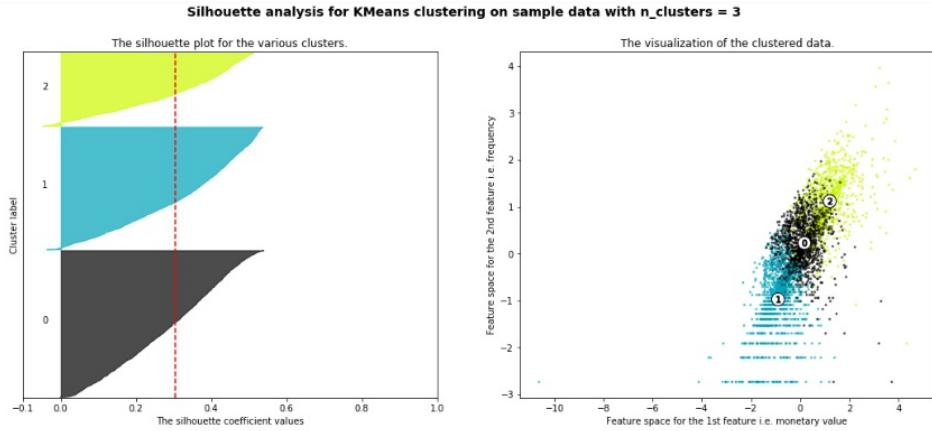
```

ax2.set_ylabel("Feature space for the 2nd feature i.e. frequency")
plt.suptitle(("Silhouette analysis for KMeans clustering on sample data "
             "with n_clusters = %d" % n_clusters),
             fontsize=14, fontweight='bold')
plt.show()

```

对于聚类要注意这个指标:silhouette score

聚类图如下。



然后对不同的特征画进行分析, 画最小值, 最大值, 25%, 中值, 75%。

traces是一个list, 通过append来添加元素, 把y,name,cls添加到traces里面。

plotly的使用可以参看[官方教程](#), plotly是一个完全开源的项目。

`np.percentile(y0, cutoff_quantile)`表示取从小到大, 在cutoff_quantile%处的值, cutoff_quantile为百分位值。

```

import plotly as py
import plotly.graph_objs as go
py.offline.init_notebook_mode()

```

```

x_data = ['Cluster 1','Cluster 2','Cluster 3','Cluster 4', 'Cluster 5']
cutoff_quantile = 100
field_to_plot = 'recency'

y0 = customer_history_df[customer_history_df['num_cluster5_labels']==0][field_to_plot].values
y0 = y0[y0<np.percentile(y0, cutoff_quantile)]
y1 = customer_history_df[customer_history_df['num_cluster5_labels']==1][field_to_plot].values
y1 = y1[y1<np.percentile(y1, cutoff_quantile)]
y2 = customer_history_df[customer_history_df['num_cluster5_labels']==2][field_to_plot].values
y2 = y2[y2<np.percentile(y2, cutoff_quantile)]
y3 = customer_history_df[customer_history_df['num_cluster5_labels']==3][field_to_plot].values
y3 = y3[y3<np.percentile(y3, cutoff_quantile)]
y4 = customer_history_df[customer_history_df['num_cluster5_labels']==4][field_to_plot].values
y4 = y4[y4<np.percentile(y4, cutoff_quantile)]
y_data = [y0,y1,y2,y3,y4]

colors = ['rgba(93, 164, 214, 0.5)', 'rgba(255, 144, 14, 0.5)', 'rgba(44, 160, 101, 0.5)', 'rgba(255, 65, 54, 0.5)', 'rgba(207, 114, 255, 0.5)', 'rgba(127, 96, 0, 0.5)']
traces = []

for xd, yd, cls in zip(x_data, y_data, colors):
    traces.append(go.Box(
        y=yd,
        name=xd,
        boxpoints=False,
        jitter=0.5,
        whiskerwidth=0.2,
        fillcolor=cls,
        marker=dict(
            size=2,
        ),
        line=dict(width=1),
    ))

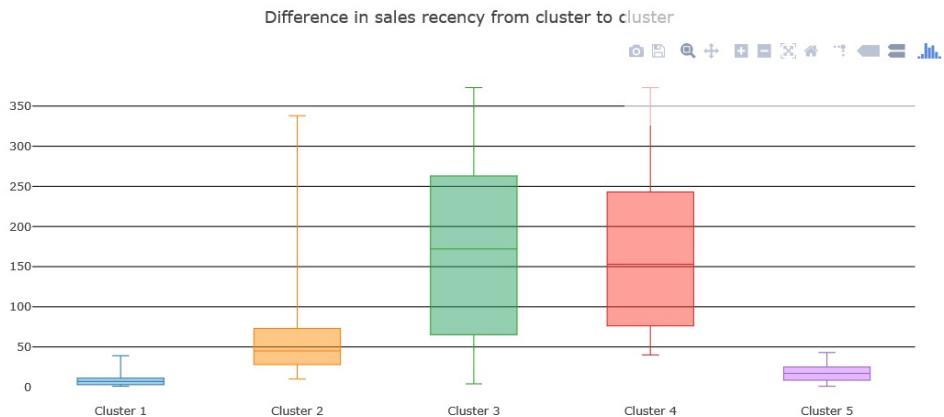
layout = go.Layout(
    title='Difference in sales {} from cluster to cluster'.format(field_to_plot),
    yaxis=dict(
        autorange=True,
        showgrid=True,
        zeroline=True,
        dtick=50,
        gridcolor='black',
        gridwidth=0.1,
        zerolinecolor='rgb(255, 255, 255)',
        zerolinewidth=2,
    ),
    margin=dict(
        l=40,
        r=30,
        b=80,
        t=100,
    ),
    paper_bgcolor='white',
    plot_bgcolor='white',
    showlegend=False
)

```

```

)
fig = go.Figure(data=traces, layout=layout)
py.offline.iplot(fig)

```



Effective Cross Selling

问题与技术路线

1. 要处理的问题是什么?
 - i. 市场篮分析
2. 要实现什么结果?
 - i. 寻找客户潜在的购买模式, 来制定销售策略, 增加销售量。
3. 数据要怎么处理?
 - i. 数据清洗(去除错误的项目, 比如购物数量小于0, 无意义的项目, 比如用户ID是nan项)
4. 怎么实现我们的目的?
 - i. 关联规则挖掘(Association Rule-Mining)
 - ii. FP Growth算法

concepts

Itemset: 存在的物品组合, 比如{A,B,C,...,X}

Support: 特定组合物品组合占所有交易的比列。

n = number of transactions with beer and diaper;

N = total transactions;

$$supp(\text{beer}, \text{diaper}) = \frac{n}{N}$$

Confidence: 一种规则的置信度。

for a rule which statrs{beer->diaper}

$$\text{confidence}(\text{beer}, \text{diaper}) = \frac{\text{supp}(\text{beer and diaper})}{\text{supp}(\text{beer})}$$

Lift: the lift of rule {X->Y}

$$\text{lift}(X -> Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X) * \text{supp}(Y)}$$

Frequent itemset: Frequent itemsets are itemsets whose support is greater than a user defined support threshold.

任务

ToDo

- Tensorflow - char embedding layer
- Tensorflow - Pointer Layer
- 机器学习 - 专题-XGBoost
- 深度学习 - GAN 与 RL 基本模型
- CRF 条件随机场 / 图模型
- Linux Shell 常用命令
 - wc/sed/awk/grep/sort/uniq/paste/cat/head/tail
- DL - 专题 - 加速方法
- 机器学习基础相关
 - 深度学习与机器学习的异同及联系
- 机器学习实践相关
 - 数据预处理
 - 缺失值
 - 特征工程, 特征选择的标志(为什么选择这些特征)
 - XGB 调参的方法、LightGBM、CatBoost
 - gbdt,xgboost,lgbm的区别
 - 模型融合的方法
 - 数据可视化分析 - 流形学习
- 算法专题-排列与组合
- 吉布斯采样
- 算法-矩阵乘法
 - 稀疏矩阵
- 显著性检测
- 机器学习的体系、学派
- 朴素贝叶斯与逻辑回归的区别
- 为什么 RNN 中采用 tanh 作为激活函数
 - | <https://www.zhihu.com/question/61265076>
- 修订 FastText 中词向量生成的描述
- 自定义用于快速测试代码的数据结构及相关方法
- 并查集
- 什么是梯度消失
- HR面常见问题
- 超参数调优
- 机器学习实战-部分代码(K-Means等)
- 判断相似二叉树及优化

- $\max(x, y)$ 的期望
- 加速网络收敛的方法
- 回归树、基尼指数
- 判断两个链表是否相交(CSDN)
- 一阶二阶泰勒展开
- 梯度下降往往没有指向

问题求解系统

分类,回归,支持向量机,决策树,集成学习与随机森林,维数约化,特征工程

深度学习的发展脉络

写这一节的目的是为了对深度学习的发展有一个认识。可以写成一部深度学习的发展史。里面要描述出当时存在的问题，以及为解决这些问题而发展出什么模型；为解决不同领域的问题如图像识别方向，语音识别方向，自然语言处理方向而发展出的不同模型。以及为解决特定领域的问题而发展特定的方法，比如从RNN到LSTM。最总会成为一个树形结构。

另外一个目的就是要时刻对最前沿的领域有一个了解，及时去拜读arxiv上最新的深度学习的论文。了解当前存在的主要问题是什么，然会看存在的可能的解决方案是什么？自己有什么想法等。看CVPR的前三名，就基本对计算机视觉的最新工作有个基本的了解了。

格式：论文是为了解决什么问题，解决的怎么样了，相对于其它方法的优势是什么，附上论文的地址。

第九章 问题求解系统

这一章，主要是构建自己的ML, DL问题求解系统，基于scikit-learn与tensorflow，目的是通过构建自己的问题系统，来方便的训练与使用模型；最终会把这个系统打造成自己的AutoML系统，当然不止是ML，还有DL。需要包含如下几部分。这个系统可以参照[阿里的云栖](#)。最终，本书中的理论都是这个系统的注脚。但是，这回事一项很浩瀚的工程。

1. Scikit-learn的代码框架
2. tensorflow的代码框架
3. 自己系统的框架
 - i. 数据清洗
 - ii. 模型的选择
 - iii. 模型的验证
 - iv. 模型的使用
4. 系统处理的问题
 - i. 基于sklearn, 对不同的主流方向(主流问题, 如分类, 回归, 降维, 特征工程, 推荐系统), 构建不同的父类
 - ii. 数据清洗是一个麻烦的工程, 这个工程也该包含在我们的系统里面
 - iii. 接下来就是模型的选择
 - iv. 特征工程是一个大的方面
 - v. 模型结果的验证
5. 问题求解系统的几大部分
 - i. 数据预处理
 - ii. 特征工程
 - iii. 统计分析
 - iv. 机器学习
 - v. 深度学习
 - vi. 将包含的部分
 - i. 时间序列分析
 - ii. 文本分析
 - iii. 网络分析
 - iv. 推荐系统
 - v. 自动参数调节
6. 需要思考的问题
 - i. 最终是否要做成一个PC端的应用程序？
 - ii. 用什么语言写做这个界面？
 - iii. 底层用什么语言实现？
 - iv. AutoML的当前市场, 未来的趋势以及怎么盈利？
7. 思考自己的框架怎么给业界带来便利, 加速业界各项研究的发展?
 - i. 自己框架怎么让机器学习, 语音识别, NLP的研究变得更容易？
 - ii. 怎么解决非一二线互联网工作没有优秀的ML, DL人才的问题？了解他们应用AI存在的困

难，以及探索解决之道。(PC端这样方便)

8. 实现问题求解系统中将要遇到的困难？只有当自己能预测自己的进度的时候，也就表明自己对整个系统成竹在胸了。每一步需要的知识，怎么建构都有自己的想法的是，自己的解决方案的时候，余下的就只是体力活了。最终，自己的系统是不是会像keras一样，只是封装了sklearn, tensorflow。

最终要实现的框架，基于这个框架能够快速的在特殊的场景下搭建起一个平台。

Sklearn介绍

sklearn的算法与模块

如下图所示, sklearn的定位是作为数据挖掘与数据分析的有效工具。它提供了这四大类算法, 分类, 回归, 聚类, 降维; 两个模块: 模型选择, 预处理。每类算法包含有不同的算法。

1. 分类
 - i. SVM, 最近邻, 随机森林
2. 回归
 - i. SVR, ridge regression, Lasso regression
3. Clustering
 - i. k-Means,spectral clustering,man-shift
4. 降维
 - i. PCA,特征提取, 非负矩阵分解
5. 模型选择
 - i. grid search,cross validation
6. Preprocessing
 - i. 预处理, 特征提取

User Guide

细节的东西我们可以看scikit-learn的User Guide。

User Guide包含如下几大模块:

1. Supervised learning
2. Unsupervised learning
3. Model selection and evaluation
4. Dataset transformations
5. Dataset loading utilities
6. Computing with scikit-learn

Sklearn的代码架构

API Reference

1. `sklearn.base` : **Base classes and utility functions**
2. `sklearn.calibration` : **Probability Calibration**
3. `sklearn.cluster` : **Clustering**
4. `sklearn.compose` : **Composite Estimators**
5. `sklearn.covariance` : **Covariance Estimators**
6. `sklearn.cross_decomposition` : **Cross decomposition**
7. `sklearn.datasets` : **Datasets**
8. `sklearn.decomposition` : **Matrix Decomposition**
9. `sklearn.discriminant_analysis` : **Discriminant Analysis**
10. `sklearn.dummy` : **Dummy estimators**
11. `sklearn.ensemble` : **Ensemble Methods**
12. `sklearn.exceptions` : **Exceptions and warnings**
13. `sklearn.feature_extraction` : **Feature Extraction**
14. `sklearn.feature_selection` : **Feature Selection**
15. `sklearn.gaussian_process` : **Gaussian Processes**
16. `sklearn.isotonic` : **Isotonic regression**

17. `sklearn.impute` : **Impute**
18. `sklearn.kernel_approximation` **Kernel Approximation**
19. `sklearn.kernel_ridge` **Kernel Ridge Regression**
20. `sklearn.linear_model` : **Generalized Linear Models**
21. `sklearn.manifold` : **Manifold Learning**
22. `sklearn.metrics` : **Metrics**
23. `sklearn.mixture` : **Gaussian Mixture Models**
24. `sklearn.model_selection` : **Model Selection**
25. `sklearn.multiclass` : **Multiclass and multilabel classification**
26. `sklearn.multioutput` : **Multioutput regression and classification**
27. `sklearn.naive_bayes` : **Naive Bayes**
28. `sklearn.neighbors` : **Nearest Neighbors**
29. `sklearn.neural_network` : **Neural network models**
30. `sklearn.pipeline` : **Pipeline**
31. `sklearn.preprocessing` : **Preprocessing and Normalization**
32. `sklearn.random_projection` : **Random projection**
33. `sklearn.semi_supervised` **Semi-Supervised Learning**
34. `sklearn.svm` : **Support Vector Machines**
35. `sklearn.tree` : **Decision Trees**

36. `sklearn.utils` : Utilities

python类的继承

```
class Person(object):
    pass

class Child(Person):          # Child 继承 Person
    pass

May = Child()
Peter = Person()

print(isinstance(May,Child))      # True
print(isinstance(May,Person))      # True
print(isinstance(Peter,Child))     # False
print(isinstance(Peter,Person))     # True
print(issubclass(Child,Person))    # True
```

`class Child(Person)`表示`Child`继承于`Person`, 任何类最终都可以追溯到根类`object`, 这与Java,C#是一样的。

Tensorflor介绍

Tensorflow的代码架构

常用数据处理技巧

文本数据

技巧总结

1. 分割含有子特征的特征，让每个子特征成为一个特征。
2. 时间数据则分割成天，星期，月，年等小特征来处理。
3. 缺失值补充，根据先验知识，补成0或者均值。
4. 类别数据通过onehot来进行处理。(还有更多需要考虑的，哪些情况下不能这么做)
5. 最终所有特征必须是数值数据或者布尔型数据。
6. 运用一个算法去得到一个预测模型。
7. 选取最佳的特征

文本数据统计

统计文本中词出现的频率，具体一个场景就是统计购物历史记录中的每类物品总的销售量。这些技巧都要固化到自己软件系统中。

`set()`是不重复集合，相当于字典的key。第一个函数得到字典的key。第二个函数建立一个list(),然后每一次购物记录作为一列。这次购物中有特定key则设为1, 否则为0.

Google Analytics Customer Revenue Prediction

1 Introduction

Here is an Exploratory Data Analysis for the Google Analytics Customer Revenue Prediction competition within the R environment. For this EDA in the main we will use `tidyverse` packages. Also for modelling we will use `glmnet`, `xgboost` and `keras` packages.

Our task is to build an algorithm that predicts the natural log of the sum of all transactions per user. Thus, for every user in the test set, the target is:

$$y_{user} = \sum_{i=1}^n transaction_{user_i}$$

$$target_{user} = \ln(y_{user} + 1).$$

Submissions are scored on the root mean squared error, which is defined as:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2},$$

where \hat{y} is the predicted revenue for a customer and y is the natural log of the actual revenue value.

Let's prepare and have a look at the dataset.

```
# Necessary libraries
import os # it's a operational system library, to set some informations
import random # random is to generate random values
```

```

import pandas as pd # to manipulate data frames
import numpy as np # to work with matrix
from scipy.stats import kurtosis, skew # it's to explore some statistics of numerical values

import matplotlib.pyplot as plt # to graphics plot
import seaborn as sns # a good library to graphic plots
import squarify # to better understand proportion of categories - it's a treemap layout algorithm

# Importing libraries to use on interactive graphs
from plotly.offline import init_notebook_mode, iplot, plot
import plotly.graph_objs as go

import json # to convert json in df
from pandas.io.json import json_normalize # to normalize the json file

# to set a style to all graphs
plt.style.use('fivethirtyeight')
init_notebook_mode(connected=True)

```

```

def cross_selling(file_path_name):
    grocery_items = set()
    with open(file_path_name) as f:
        reader = csv.reader(f, delimiter=",")
        for i, line in enumerate(reader):
            grocery_items.update(line)
    output_list = list()
    with open(file_path_name) as f:
        reader = csv.reader(f, delimiter=",")
        for i, line in enumerate(reader):
            row_val = {item: 0 for item in grocery_items}//set item to 0 as initial value
            row_val.update({item: 1 for item in line})//set to 1 if item in line
            output_list.append(row_val)
    grocery_df = pd.DataFrame(output_list)
    return grocery_df

if __name__ == '__main__':
    file_path_name = "C:\Users\pili\GitBook\Kaggle\OnlineRetailTransactions\grocery_dataset.txt"
    cross_selling(file_path_name)

```

```

item_summary_df = grocery_df.sum().sort_values(ascending=False).reset_index()
item_summary_df.rename(columns={item_summary_df.columns[0]: 'item_name', item_summary_df.columns[1]: 'item_count'}, inplace=True)

```

降序排序，再更换columns的名字。

```

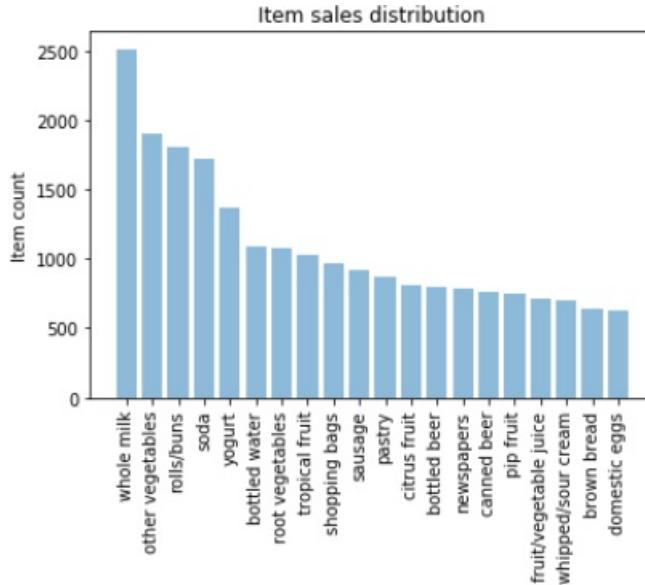
import matplotlib.pyplot as plt
objects = (list(item_summary_df['item_name'].head(20)))
y_pos = np.arange(len(objects))

```

```

performance = list(item_summary_df['item_count'].head(20))
plt.bar(y_pos, performance, align='center', alpha=0.5)
plt.xticks(y_pos, objects, rotation='vertical')
plt.ylabel('Item count')
plt.title('Item sales distribution')
plt.show()

```



带多个标签数据的处理 1

```

cf_pd.geoNetwork[0]
'{"continent": "Asia", "subContinent": "Western Asia", "country": "Turkey", "region": "Izmir"
, "metro": "(not set)", "city": "Izmir", "cityId": "not available in demo dataset", "networkD
omain": "ttnet.com.tr", "latitude": "not available in demo dataset", "longitude": "not availa
ble in demo dataset", "networkLocation": "not available in demo dataset"}'

```

带多个标签数据的处理 2

```

cf_pd.totals[0]
'{"visits": "1", "hits": "1", "pageviews": "1", "bounces": "1", "newVisits": "1"}'

```

数据处理技巧

1. 把所有子标签都提取出来做一个标签。一个标签就是一个feature
 - i. 怎么自动的提取子标签？
2. 列出所有的feature名字(.columns)
3. 如果一个feature只有一个类，则舍去
4. 对于时间这个特征，可以使用datetime把他分成年，月，日，星期，hour等多个特征来处理(最

后原始时间这个特征要删除)

i. 含有时间的feature怎么自动分割成小的特征？

缺失值的比例

```
def missing_values(data):
    total = data.isnull().sum().sort_values(ascending=False)
    percent = (data.isnull().sum()/data.isnull().count()*100).sort_values(ascending=False)
    df = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])

    print("Total columns at least one Values:")
    print(df[~(df['Total'] == 0)])
    print("\n Total of Sales % of Total: ", round((df_train[df_train['totals.transactionRevenue'] != np.nan]['totals.transactionRevenue'].count() / len(df_train['totals.transactionRevenue']) * 100),4))
    return total, percent
```

时间特征处理

```
from datetime import datetime
def date_process(df):
    df['date'] = pd.to_datetime(df['date'], format="%Y%m%d")
    df['_weekday'] = df['date'].dt.weekday
    df['_day'] = df['date'].dt.day
    df['_month'] = df['date'].dt.month
    df['_year'] = df['date'].dt.year
    df['_visitHour'] = (df['visitStartTime'].apply(lambda x:str(datetime.fromtimestamp(x).hour))).astype(int)
    return df
```

找到只有一个类的数据并删除

```
discovering_consts = [col for col in df_train.columns if df_train[col].nunique() == 1]
df_train.drop(discovering_consts, axis=1, inplace=True)
```

画图:

```
# setting the figure size of our plots
plt.figure(figsize=(14,5))

# Subplot allow us to plot more than one
# in this case, will be create a subplot grid of 2 x 1
plt.subplot(1,2,1)
# setting the distribution of our data and normalizing using np.log on values highest than 0
and +
# also, we will set the number of bins and if we want or not kde on our histogram
ax = sns.distplot(np.log(df_train[df_train['totals.transactionRevenue'] > 0]['totals.transactionRevenue'] + 0.01), bins=40, kde=True)
ax.set_xlabel('Transaction RevenueLog', fontsize=15) #setting the xlabel and size of font
ax.set_ylabel('Distribution', fontsize=15) #setting the ylabel and size of font
```

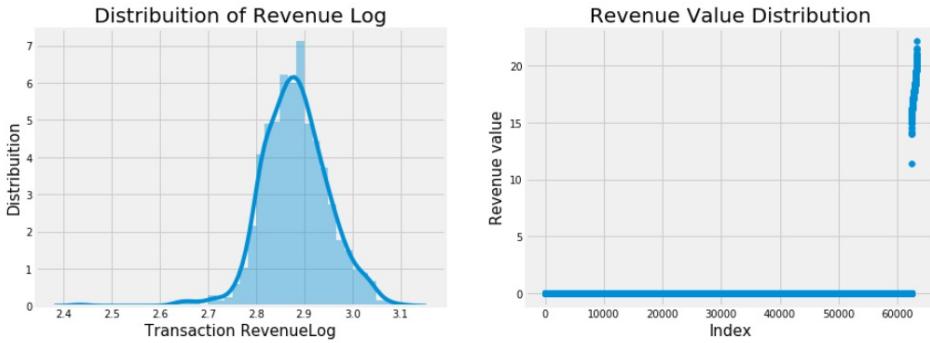
```

ax.set_title("Distribution of Revenue Log", fontsize=20) # setting the title and size of font

# setting the second plot of our grid of graphs
plt.subplot(1,2,2)
# ordering the total of users and setting the values of transactions to understanding
plt.scatter(range(df_train.shape[0]), np.sort(df_train['totals.transactionRevenue'].values))
plt.xlabel('Index', fontsize=15) # xlabel and size of words
plt.ylabel('Revenue value', fontsize=15) # ylabel and size of words
plt.title("Revenue Value Distribution", fontsize=20) # Setting Title and fontsize

plt.show()

```



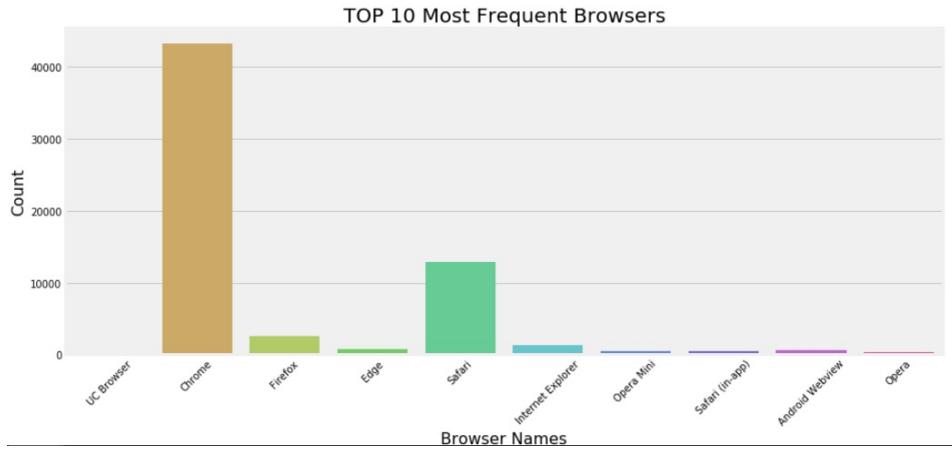
Season CountPlot

```

plt.figure(figsize=(14,6))
df_top10 = df_train['device.browser'].value_counts()[:10].index.values
#df_train['device.browser'].isin(df_top10)
sns.countplot(df_train[df_train['device.browser'].isin(df_top10)]['device.browser'], palette="hls")
plt.title("TOP 10 Most Frequent Browsers", fontsize=20) # Adding Title and setting the size
plt.xlabel("Browser Names", fontsize=16) # Adding x label and setting the size
plt.ylabel("Count", fontsize=16) # Adding y label and setting the size
plt.xticks(rotation=45) # Adjust the xticks, rotating the labels

plt.show() #use plt.show to render the graph that we did above

```



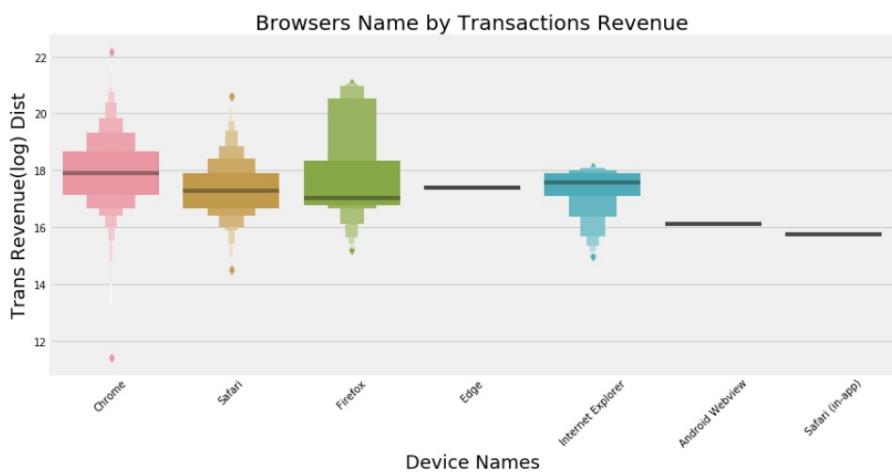
boxenplot

```

plt.figure(figsize=(13,6))
df_top10 = df_train['device.browser'].value_counts()[:10].index.values
g1 = sns.boxenplot(x='device.browser',y='totals.transactionRevenue',
                    data=df_train[(df_train['device.browser'].isin(df_top10))&df_train['totals.transactionRevenue']>0])
g1.set_title('Browsers Name by Transactions Revenue', fontsize=20) # title and fontsize
g1.set_xticklabels(g1.get_xticklabels(), rotation=45) # It's the way to rotate the xticks when we use variable to our graphs
g1.set_xlabel('Device Names', fontsize=18) # Xlabel
g1.set_ylabel('Trans Revenue(log) Dist', fontsize=18) #Ylabel

plt.show()

```



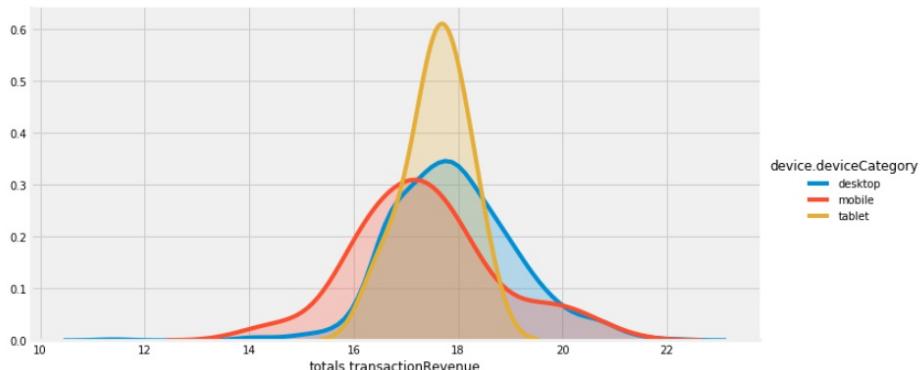
FacetGrid

```
(sns.FacetGrid(df_train[df_train['totals.transactionRevenue'] > 0],
```

```

        hue='device.deviceCategory', height=5, aspect=2)
    .map(sns.kdeplot, 'totals.transactionRevenue', shade=True)
    .add_legend()
)
plt.show()

```



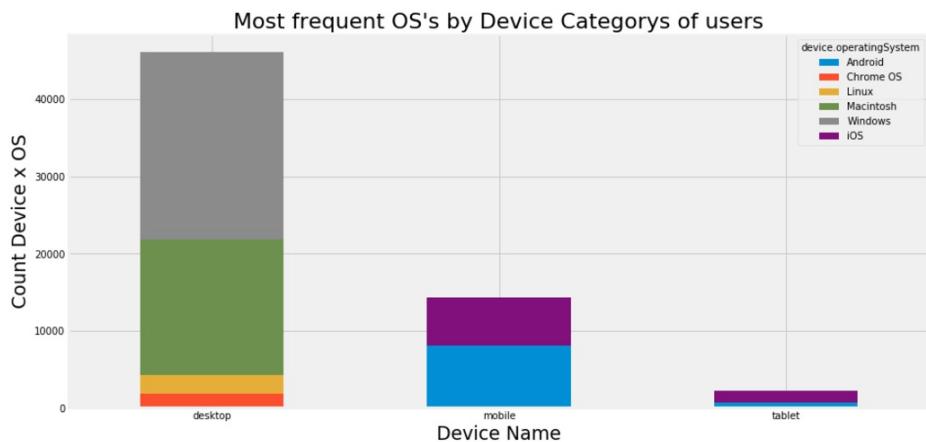
Crosstab

```

# At index I will use isin to substitute the loop and get just the values with more than 1%
crosstab_eda = pd.crosstab(index=df_train['device.deviceCategory'], # at this line, I am usin
g the isin to select just the top 5 of browsers
                            columns=df_train[df_train['device.operatingSystem']]\n
                                         .isin(df_train['device.operatingSystem'])\n
                                         .value_counts()[:6].index.values)[['device.\noperatingSystem']]
# Ploting the crosstab that we did above
crosstab_eda.plot(kind="bar",      # select the bar to plot the count of categoricals
                  figsize=(14,7), # adjusting the size of graphs
                  stacked=True)   # code to unstack
plt.title("Most frequent OS's by Device Categories of users", fontsize=22) # adjusting title a
nd fontsize
plt.xlabel("Device Name", fontsize=19)           # adjusting x label and fontsize
plt.ylabel("Count Device x OS", fontsize=19)     # adjusting y labe
l and fontsize
plt.xticks(rotation=0)                         # Adjust the xticks, rotati
ng the labels

plt.show() # rendering

```



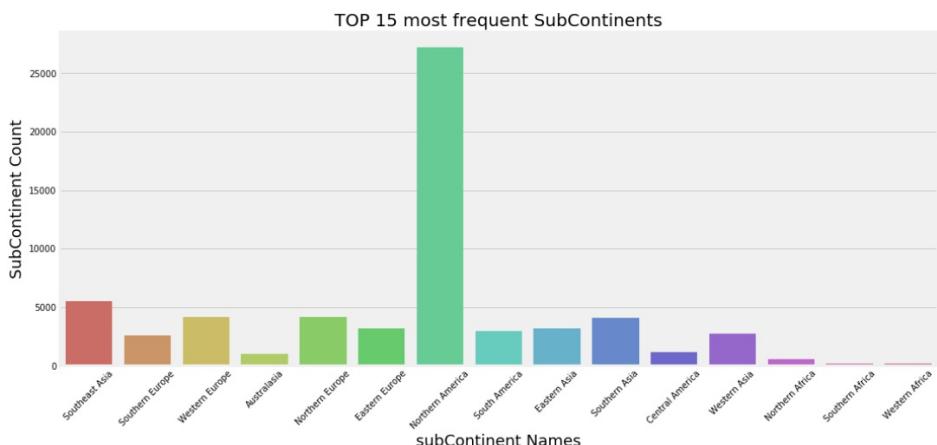
Countplot

```
plt.figure(figsize=(16,7))

sns.countplot(df_train[df_train['geoNetwork.subContinent']\
                     .isin(df_train['geoNetwork.subContinent'].value_counts()[:15].index.v\
lues)]\

[ 'geoNetwork.subContinent'], palette="hls")
plt.title("TOP 15 most frequent SubContinents", fontsize=20) # seting the title size
plt.xlabel("subContinent Names", fontsize=18) # seting the x label size
plt.ylabel("SubContinent Count", fontsize=18) # seting the y label size
plt.xticks(rotation=45) # Adjust the xticks, rotating the labels

plt.show()
```



Crosstab

```
date_sales = ['_visitHour', '_weekday'] #seting the desired
```

```

cm = sns.light_palette("green", as_cmap=True)
pd.crosstab(df_train[date_sales[0]], df_train[date_sales[1]],
            values=df_train["totals.transactionRevenue"], aggfunc=[np.sum]).style.background_
gradient(cmap = cm)

# tab.columns.levels[1] = ["Sun", "Mon", "Thu", "wed", "Thi","Fri","Sat"]

```

	sum							
_weekday	0	1	2	3	4	5	6	
_visitHour								
0	126.386	192.059	143.162	36.2319	37.3215	53.0282	142.051	
1	138.872	87.9446	120.248	70.4552	37.2508	70.3504	106.245	
2	122.176	160.589	54.7113	139.921	125.765	17.7265	116.501	
3	107.211	124.656	84.0083	76.1021	74.1997	15.76	36.4601	
4	71.6829	107.381	90.8094	69.3424	49.719	16.8251	17.6372	
5	105.335	70.0461	36.3155	53.4049	52.448	53.8893	72.7108	
6	34.9277	70.9624	16.7699	36.6059	17.6348	0	52.4337	
7	17.7273	35.4628	0	18.3798	0	0	16.79	
8	54.4742	18.619	0	33.8402	0	0	0	
9	0	19.2371	0	0	0	0	0	
10	0	17.597	0	17.0134	0	17.8254	0	
11	18.9495	18.2831	19.9462	17.5434	53.2772	0	0	
12	0	18.747	74.8096	58.3735	55.9478	0	52.9778	
13	181.915	93.8184	50.857	71.523	107.002	0	34.476	
14	87.7575	36.8734	123.865	155.946	142.324	72.1576	0	
15	211.405	122.643	183.972	205.398	274.288	32.8824	85.0231	
16	91.8596	123.522	85.8605	182.576	178.588	82.5616	86.4576	
17	214.562	167.027	236.304	167.496	88.2338	18.0095	53.8049	
18	288.439	230.905	235.754	308.098	183.177	0	0	
19	174.786	202.682	123.821	175.253	143.794	107.968	53.4519	
20	160.666	249.791	183.774	105.292	170.235	53.6175	51.9169	
21	199.911	85.0781	93.2237	200.392	126.461	141.345	72.2549	
22	172.337	143.755	90.8017	101.408	188.296	17.0162	85.5684	
23	87.7206	88.2007	107.127	105.673	219.07	110.034	85.0856	

```

number_of_colors = 20 # total number of different colors that we will use

# Here I will generate a bunch of hexadecimal colors
color = ["#" + ''.join([random.choice('0123456789ABCDEF') for j in range(6)]) for i in range(number_of_colors)]
country_tree = df_train["geoNetwork.country"].value_counts() #counting the values of Country

print("Description most frequent countrys: ")

```

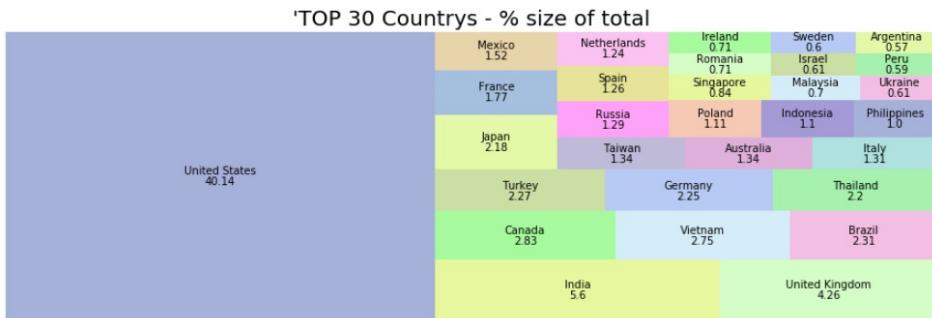
```

print(country_tree[:15]) #printing the 15 top most

country_tree = round((df_train["geoNetwork.country"].value_counts()[:30] \
                     / len(df_train['geoNetwork.country'])) * 100),2)

plt.figure(figsize=(14,5))
g = squarify.plot(sizes=country_tree.values, label=country_tree.index,
                   value=country_tree.values,
                   alpha=.4, color=color)
g.set_title("TOP 30 Countrys - % size of total", fontsize=20)
g.set_axis_off()
plt.show()

```



plot country map

画这幅图时会出现问题，就是不显示，需要关掉笔记再打开。

```

# Counting total visits by countrys
countMaps = pd.DataFrame(df_train['geoNetwork.country'].value_counts()).reset_index()
countMaps.columns=['country', 'counts'] #renaming columns
countMaps = countMaps.reset_index().drop('index', axis=1) #reseting index and droping the col
umn

data = [ dict(
    type = 'choropleth',
    locations = countMaps['country'],
    locationmode = 'country names',
    z = countMaps['counts'],
    text = countMaps['country'],
    autocolorscale = False,
    marker = dict(
        line = dict (
            color = 'rgb(180,180,180)',
            width = 0.5
        )),
    colorbar = dict(
        autotick = False,
        tickprefix = '',
        title = 'Number of Visits'),
) ]

```

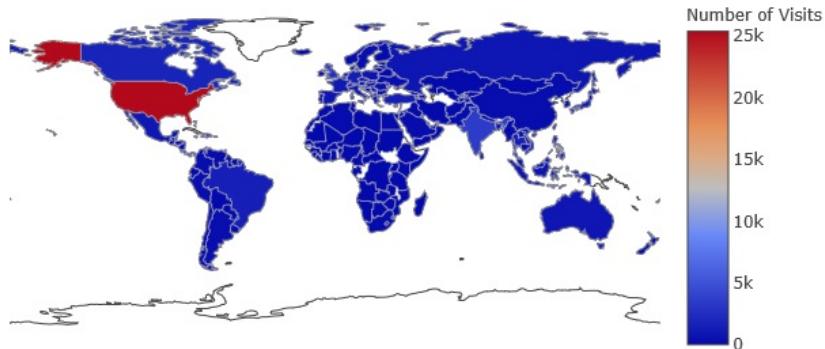
```

layout = dict(
    title = 'Couting Visits Per Country',
    geo = dict(
        showframe = False,
        showcoastlines = True,
        projection = dict(
            type = 'Mercator'
        )
    )
)

figure = dict( data=data, layout=layout )
iplot(figure, validate=False, filename='map-countrys-count')

```

Couting Visits Per Country



Pie

```

def PieChart(df_colum, title, limit=15):
    """
    This function helps to investigate the proportion of visits and total of transction revenue
    by each category
    """

    count_trace = df_train[df_colum].value_counts()[:limit].to_frame().reset_index()
    rev_trace = df_train.groupby(df_colum)[ "totals.transactionRevenue"].sum().nlargest(10).to
    _frame().reset_index()

    trace1 = go.Pie(labels=count_trace['index'], values=count_trace[df_colum], name= "% Acess
es", hole= .5,
                    hoverinfo="label+percent+name", showlegend=True, domain= {'x': [0, .48]}, 
                    marker=dict(colors=color))

```

```

trace2 = go.Pie(labels=rev_trace[df_column],
                 values=rev_trace['totals.transactionRevenue'], name="% Revenue", hole=.5
                ,
                hoverinfo="label+percent+name", showlegend=False, domain= {'x': [.52, 1]}

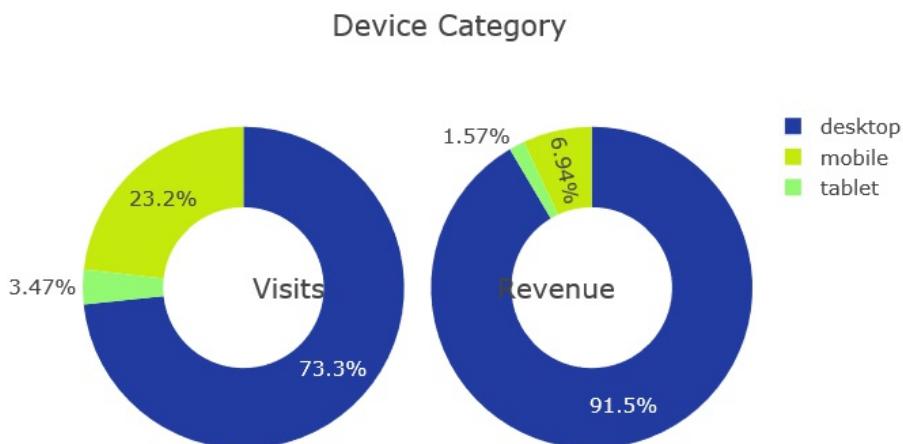
)

layout = dict(title=title, height=450, font=dict(size=15),
              annotations=[dict(
                dict(
                  x=.25, y=.5,
                  text='Visits',
                  showarrow=False,
                  font=dict(size=20)
                ),
                dict(
                  x=.8, y=.5,
                  text='Revenue',
                  showarrow=False,
                  font=dict(size=20)
                )
              ])
            )

fig = dict(data=[trace1, trace2], layout=layout)
iplot(fig)

```

PieChart("device.deviceCategory", "Device Category")



NLP常用处理技巧

特征构造

矢量化

1. 通过求每一个词的tf-idf值，用tf-idf值代替单词，将文本转化为向量。
2. 将tf-idf值小于一定阈值的词删掉，这样来降低维度。
3. 用词汇表将每一个文本转化为维度一样的向量，并且非0值即为单词所对应的tf-idf值。

PageRank算法

思想：

1. 链接数量。一个网页被越多的其它网页链接，说明这个网页越重要。
2. 链接质量。一个网页被一个越高权值的网页链接，也能表明这个网页越重要。贡献的权值等于这个网页的权值除以这个网页的出链数目。

$In(V_i)$ 是 V_i 的入链集合，代表有多少其它网页链接它； $Out(V_i)$ 是 V_i 的出链集合，也就是引用了哪些其它网页。网页 V_i 的得分为：

$$S(V_i) = \sum_{j \in In(V_i)} \left(\frac{1}{Out(V_j)} S(V_j) \right)$$

为了防止有些网页的权值为0，则加入一个阻尼项。

$$S(V_i) = (1 - d) + d \sum_{j \in In(V_i)} \left(\frac{1}{Out(V_j)} S(V_j) \right)$$

不需要借助语料库，单篇文章提取关键词的算法TextRank的原理与PageRank一致，因为是借鉴PageRank算法的。

第十章 面试

这章涉及到C++,传统算法, ML, DL方面的常见问题。

概率, 统计, 推断是很重要的一个方向, 包括贝叶斯方法,MCMC这些, 到时有必要增加一章。概率统计是计算机视觉的基础。

机器学习类面试问题集

算法理论方面

- 2.1 LR, SVM, KNN, GBDT, XGB推导, 算法细节(LR为何是sigmod, 理论推导出sigmod,KNN距离度量方式, XGBoost为什么要用二阶信息不用一阶, LR和SVM对比, GBDT和XGB和LightGBM对比)。
- 2.2 CNN DNN RNN 细节以及相关问题(pool层, 激活函数, 梯度消失弥散问题, LSTM结构图, 深度网络优势及缺点)。
- 2.3 常见排序算法的复杂度和一些细节以及改进优化。
- 2.4 树模型建模过程。
- 2.5 特征选择方法。
- 2.6 模型训练停止方法。
- 2.7 正则化作用。
- 2.8 模型效果评价指标。
- 2.9 AUC理解和计算方法。
- 2.10 Hadoop, Hive, Spark相关理论。
- 2.11 L_BFGS, DFP推导。
- 2.12 弱分类器组合成强分类器的理论证明。
- 2.13 FM, FMM, Rank_SVM算法细节。
- 2.14 map_reduce基本概念以及常见处理代码。
- 2.15 过拟合的解决方法。
- 2.16 各个损失函数之间区别。
- 2.17 L1,L2正则化相关问题。

前向神经网络问题

激活函数的作用

增加非线性，增强学习能力

为啥使用收敛慢的(随机)梯度下降法

因为只需要计算一阶导数，而不需要计算二阶，因为几十万以上的参数，计算二阶导数太费时间，内存也是个问题(除非用L-BFGS)

非线性激活函数在神经网络中的作用

1. 线性激活函数的劣势：如果只是线性激活函数，则多层网络与单层网络等价，因为每多一层，只是相当于乘以一个矩阵。
2. 非线性激活函数的好处：能把原来线性不可解的问题变为高纬度空间中的线性可解问题。线性可解的意思是：
 - i. 对于分类问题，则原本线性不可分的在新的空间中线性可分了；
 - ii. 对于回归问题，非线性激活函数的存在使得我们可以逼近任意的连续函数。
3. 获得更强大的函数拟合能力，万有近似定理。

为啥Sigmoid和Tanh激活函数会导致梯度消失的现象？

因为在参数很大或者很小的时候，他们的导数会趋于0，因此造成了梯度消失。

ReLU系列激活函数相对于Sigmoid和Tanh激活函数的有点是什么？它们有什么局限性以及如何改进？

优点

- (1)计算梯度简单，因为Sigmoid和Tanh涉及到指数运算
- (2)ReLU能有效解决后两者的问题，提供相对宽的激活边界
- (3)ReLU的单侧抑制提供了网络的稀疏表达能力(Why？因为他可以使一部分神经元的输出为0，因此这些神经元不起作用。)

缺点

ReLU的局限性在于其训练过程中会导致神经元死亡的问题，由于激活函数是 $mf(x) = \max(0, z)$ 导致梯度在经过该ReLU单元时，被置为0。且之后再也不能被任何数据激活，因为流经这个神经元的梯度永远为0，因此不对任何数据产生响应。因此会导致一定比例的神经元不可逆的死亡，进而参数梯度永远无法更新，整个训练过程失败。

为了解决这一问题，人们设计了Leaky ReLU(LReLU)其表达式是：

$$f(z) = z; z > 0$$

$$f(z) = az; z \leq 0$$

一般 a 是一个很小的正数，但是怎么选取 a 也是一个问题。为解决这个问题，人们发明了参数化的PReLU(Parametric ReLU)，它与LReLU的主要区别是，斜率参数 a 作为网络中一个可学习的参数。而另一个LReLU的变种增加了“随机化”机制，具体的，在训练过程中，斜率 a 作为一个满足某种分布的随机采样；测试时再固定下来。Random ReLU(RReLU)在一定程度上能起到正则化的作用。

平方误差损失函数和交叉熵损失函数分别适合什么场景？

平方损失函数：回归问题，输出是连续值，并且最后一层不含Sigmoid或者Softmax激活函数的NN。

因为会导致梯度很小而发生梯度消失的问题

交叉熵：分类问题，

写出多层感知机的平方误差和交叉熵损失函数

给定包含 m 个样本的集合 $((x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)}))$,

代价函数

平方误差其整体代价函数是：

$$\begin{aligned} J(W, b) &= [\frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)})] + \frac{\lambda}{2} \sum_{l=1}^{N-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (W_{ij}^{(l)})^2 \\ &= [\frac{1}{m} \sum_{i=1}^m \frac{1}{2} \|y^{(i)} - L_{W,b}(x^{(i)})\|^2] + \frac{\lambda}{2} \sum_{l=1}^{N-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (W_{ij}^{(l)})^2 \end{aligned}$$

二分类交叉熵误差其整体代价函数是：

$$\begin{aligned} J(W, b) &= -[\frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)})] + \frac{\lambda}{2} \sum_{l=1}^{N-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (W_{ij}^{(l)})^2 \\ &= -[\frac{1}{m} \sum_{i=1}^m (y^{(i)} \ln o^{(i)} + (1 - y^{(i)}) \ln(1 - o^{(i)}))] + \frac{\lambda}{2} \sum_{l=1}^{N-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (W_{ij}^{(l)})^2 \end{aligned}$$

多分类交叉熵误差其整体代价函数是：

$$\begin{aligned} J(W, b) &= -[\frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)})] + \frac{\lambda}{2} \sum_{l=1}^{N-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (W_{ij}^{(l)})^2 \\ &= -[\frac{1}{m} \sum_{i=1}^m y_k^{(i)} \ln o_k^{(i)}] + \frac{\lambda}{2} \sum_{l=1}^{N-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (W_{ij}^{(l)})^2 \end{aligned}$$

其中 $o_k^{(i)}$

代表第*i*个样本的预测属于类别*k*的概率， $y_k^{(i)}$ 为实际的概率(如果第*i*个样本的真实类别为*k*，则

$y_k^{(i)} = 1$ ，否则为0)。

梯度计算公式

第(l)层的参数为 $W^{(l)}, b^{(l)}$, 每一层的线性变换为:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{x}^{(l)} + \mathbf{b}^{(l)}$$

输出为:

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

其中f是非线性激活函数(比如Sigmoid, Tanh, ReLu等); $\mathbf{a}^{(l)}$ 直接作为下一层的输入.即:

$\mathbf{x}^{(l+1)} = \mathbf{a}^{(l)}$ 。我们通过批量梯度下降法来优化网络参数。

$$\mathbf{W}_{ij}^{(l)} = \mathbf{W}_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(\mathbf{W}, \mathbf{b}),$$

$$\mathbf{b}_i^{(l)} = \mathbf{b}_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(\mathbf{W}, \mathbf{b}),$$

问题的核心是求 $\frac{\partial}{\partial W_{ij}^{(l)}} J(\mathbf{W}, \mathbf{b}), \frac{\partial}{\partial b_i^{(l)}} J(\mathbf{W}, \mathbf{b})$

然后从最后一层, 反向迭代。

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(\mathbf{W}, \mathbf{b}) = \frac{\partial J(\mathbf{W}, \mathbf{b})}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial \mathbf{W}_{ij}^{(l)}}$$

神经网络训练技巧

主要涉及的问题是“过拟合”:解决方法包括数据集增强, 正则化, 模型集成。其中dropout是模型集成方法中最高效与常用的技巧。此外, DNN训练中设计的手动调参, 如学习率, 权重衰减系数, Dropout比例等, 这些参数的选择会显著影响模型最终的训练效果。Batch Normalization(BN)方法有效规避了这些复杂参数对网络训练产生的影响, 在加速训练收敛的同时也提升了网络的泛化能力。

神经网络训练时是否可以将全部参数初始化为0?

因为所有参数都是相同的值, 所有神经元都是对称的, 因此没有办法打破这种对称, 导致无论学习多久, 所有参数依然是相同的。因此我们需要随机地初始化NN参数的值, 来打破这种对称性。

为啥Dropout可以抑制过拟合? 它的工作原理和实现?

还得先介绍Dropout的实际步骤, 也就是训练的时候是多少个神经元? 预测的时候是多少个?
Dropout作用于小批量训练数据, 由于其随机丢弃部分神经元的机制, 相当于每次迭代都是在训练不同结构的神经网络, 类比于Bagging方法, Dropout可被认为是一种实用的大规模深度神经网络的模型集成算法。这是由于传统意义上的Bagging涉及多个模型的同时训练和测试评估, 当网络与参数规模庞大时, 这种集成方式需要消耗大量的运算时间与空间。Dropout在小批量级别上的操作, 提供了一种轻量级的Bagging集成近似, 能够实现指数级数量神经网络的训练与评测。

Dropout在具体实现中, 某个神经元以概率p被丢弃, 因此N个神经元在Dropout下相当于是 2^N 个模型的集成。这 2^N 个模型可认为是原始网络的子网络, 它们共享部分权值, 并且具有相同的网络层

数，而模型的整体参数数目不变，这就大大简化了运算。对于任意神经元，每次训练中都与一组随机挑选的不同的神经元集合共同进行优化，这个过程会减弱全体神经元之间的联合适应性，减小过拟合的风险，增强泛化能力。

批量归一化的基本动机与原理是什么？在卷积神经网络中如何使用？

神经网络训练过程的本质是学习数据分布，如果训练数据与测试数据的分布不同将大大降低网络的泛化能力，因此我们需要在训练开始前对所有输入数据进行归一化处理。

归一化方法是针对每一批数据（比如m个样本），在网络的每一层的输入之前增加归一化处理（均值为0，方差为1），将所有批处理数据强制在统一的数据分布下，即对该层的任一神经元（假设是第k维） $\hat{x}^{(k)}$ 采用如下公式：

$$\hat{x}^{(k)} = \frac{\bar{x}^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

其中 $x^{(k)}$ 为该层第k个神经元的原始输入数据， $E[x^{(k)}]$ 为这一批输入数据在第k个神经元的均值， $\sqrt{Var[x^{(k)}]}$ 为这一批数据在k个神经元的标准差。

卷积操作的本质特性包括系数交互和参数共享，具体解释这两种特性以及作用。

稀疏交互

对于全连接网络，如果第k层有m个神经元，第k+1层有n个神经元，这两层间的权重数目是mn，对于卷积网络，如果卷积核大小为k，则这两层间的权重个数是kn，因为后面一层，每一个神经元只与前一层k个神经元相连。一般k远远小于m，因此相对于全连接网络而言，参数数量大大减少了，因此具有稀疏交互的特征。

稀疏交互的物理意义是，通常图像、文本、语音等现实世界中的数据都具有局部的特征结构，我看可以先学习局部的特征，再将局部的特征组合起来形成复杂和抽象的特征。以人脸识别为例，最底层的神经元可以检测出各个角度的边缘特征，中间层的神经元可以将边缘组合起来得到眼睛、鼻子、嘴巴等复特征；最后，位于上层的神经元可以根据各个器官的组合检测出人脸的特征。

参数共享

参数共享是指在同一个模型的不同模块中使用相同的参数，它是卷积运算的固有属性。在学习的过程中，我们学习到的就是卷积核（的参数）。根据参数共享的思想，我们只需要学习一组参数集合，而不需要针对每个位置的每个参数都进行优化，从而大大降低了模型的储存需求。

参数共享的物理意义是使得卷积具有平移不变性。假如图像中有一只猫，那么无论它出现图像中的什么位置，都应该将它识别为猫，也就是说神经网络的输出对于平移变换具有不变性。

常用的池化操作有哪些？池化的作用是什么？

常用的池化操作主要针对非重叠区域，包括均值（mean pooling）、最大池（max pooling）。mean pooling是用一个区域的平均值代替这个区域，能够抑制由于领域大小受限造成估计值方差增大的现象，特点是背景的保留效果更好；max pooling是用一个区域的最大值代替这个区域本身，能够抑制网络参数误差造成估计均值偏移现象，特点是更好地提取纹理信息。

池化操作的本质是降采样。除了能显著降低参数数量，还能够保持对平移，伸缩，旋转操作的不变性。平移不变性是对小量平移而言，尺度一般是pooling的尺寸。伸缩(尺度)不变性，因为pooling就是一个能保持区域主要信息(如最大值)的压缩变换，因此具有伸缩不变性。旋转不变性，在配合多重pooling以后，对于max pooling，对于旋转的图像，我们仍能得到相同的结果，比如图像数字5。

卷积神经网络如何用于文本分类任务？

假设文本总单词量是N，每个单词可以转化为一个K维向量，向量维度K可以预先在其它语料库中获得，也可以作为未知的参数有网络训练得到。这样就组成了一个NXK的输入矩阵，可以看成是一个图像。

卷积层

在这个矩阵上，我们引入hXK大小的卷积核，进行卷积操作：

$$c_i = f(w * x_{i:i+h-1+b})$$

卷积后得到一个N-h+1维度大小的特征向量。通过不同大小的卷积核，提炼出不同的特征向量，构成卷积层的输出。

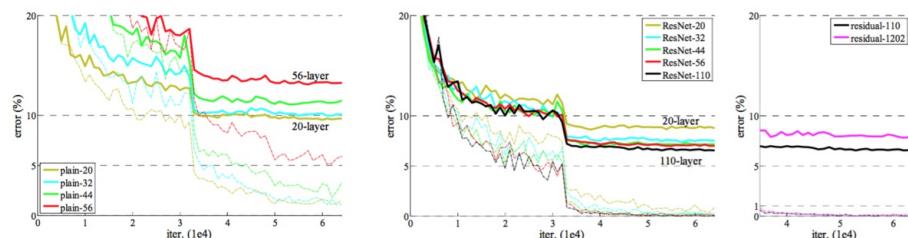
池化层

选用K-Max进行池化，也就是对每个卷积核作用之后得到的特征向量，进行k-Max池化，也就是选择特征向量中最大的K个特征，这样，每个特征向量转化成一个k维向量，效果就是，通过池化把不同长度的句子转化成定长的向量表示。若有M个卷积核，池化后就得到一个kXM的矩阵，k=1的话就是，1-Max，最终一个文本对应一个1XM的向量。

后面的网络结构就与具体的任务有关了，如果是人本分类，就最后接入一个全连接层，并使用softmax激活函数输出每个类别的概率。

ResNet的提出背景和核心理论是什么？

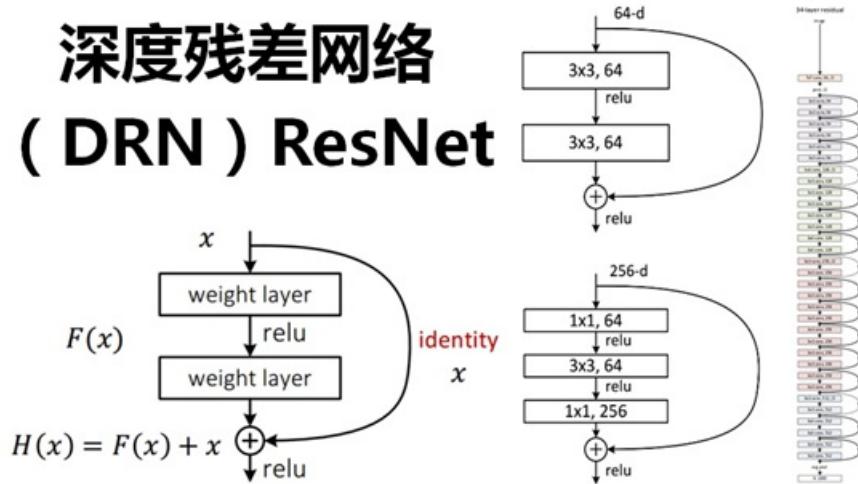
ResNet的提出背景是解决或缓解深层的神经网络训练中的梯度消失问题。当时的一个问题是，层数更深的神经网络反而会有更大的训练误差。这种反常，很大程度上归结于深度神经网络的梯度消失问题。随着网络层数的增加，很误差反向传递时，在每层间是相乘的，N层网络，从输出到输入，就是N次连乘，因此很容易造成梯度的消失与膨胀，影响参数的学习。下图就是传统深度神经网络与残差网络的对比。



如下图所示，输入 x 经过两个神经网络的变换得到 $F(x)$ ，同时也短接到两层之后，最后这个包含两层神经网络模块输出 $H(x)=F(x)+x$ ；这样一来， $F(x)$ 被设计为只需要拟合输入 x 与目标输出 $\hat{H}(x)$ 的残差 $\hat{H}(x) - x$ ，残差网络的名字因此而来。如果某一层的输出已经很好的拟合了期望结果，那么多加入一层不会使得模型变差，因为该层的输出将直接被短接到两层之后，直接相当于学习了一个恒等映射，而跳过的两层只需要拟合上层输出和目标之间的残差即可。

ResNet可以有效改善深层的神经网络学习问题，使得训练更深的网络成为可能。

深度残差网络 (DRN) ResNet



基本C++问题

指针与引用的区别

指针是一个变量，该变量储存的是一个地址，指向内存中的一个内存单元；而引用只是原来变量的一个别名。

数组名与指针的区别

数组名a可以当成数字首地址的指针,除如下两种情况, sizeof(a),&a.

1.取sizeof得到不同结果;对数组取sizeof得到整个数组的长度;对指针取sizeof得到指针本身所占空间的大小。

2.指针可以pointer++, 数组名不可以。

3.指针可以赋值再指向别人, 数组名不可以。

实现String类的构造函数, 析构函数和赋值函数

```
#pragma once
class AString
{
public:
    AString(const char* str = NULL);
    AString(const AString &other);
    ~AString(void);
    AString & operator=(const AString &other);
private:
    char *m_String;
};
```

```
#include "StdAfx.h"
#include "AString.h"
#include <iostream>
using namespace std;

AString::~AString(void)
{
    cout<<"Destructing"<<endl;
    if(m_String != NULL)
    {
        delete [] m_String;
        m_String = NULL;
    }
}

AString::AString(const char* str)
{
```

```

cout<<"Constructing"<<endl;
if(str == NULL)
{
    m_String = new char[1];
    *m_String = '\0';
}
else
{
    m_String = new char[strlen(str)+1];
    strcpy(m_String, str);
}
}

AString::AString(const AString &other)
{
    cout<<"Constructing copy"<<endl;
    m_String = new char[strlen(other.m_String)+1];
    strcpy(m_String, other.m_String);
}

AString & AString::operator=(const AString &other)
{
    cout<<"Operator = Function:"<<endl;
    if(this == &other)
    {
        return *this;
    }
    delete []m_String;
    m_String = new char[strlen(other.m_String)+1];
    strcpy(m_String, other.m_String);
    return *this;
}

```

```

#include "StdAfx.h"
#include "AString.h"
#include <iostream>
using namespace std;

void main()
{
    cout<<"Step 1"<<endl;
    AString a("hello");
    cout<<"Step 2"<<endl;
    AString b("world");
    cout<<"Step 3"<<endl;
    AString c(a);
    cout<<"Step 4"<<endl;
    c = b;
    cout<<"Step 5"<<endl;
}

```

The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The window contains the following text:

```
Step 1
Constructing
Step 2
Constructing
Step 3
Constructing copy
Step 4
Operator = Function:
Step 5
Destructing
Destructing
Destructing
Press any key to continue . . .
```

字符串指针赋值问题

如果指针没有分配空间，则会出现0xC0000005: Access violation writing location

```
void main(int n)
{
    //char pStr[] = "abc";//right
    //char pStr[4] = "abc";//right
    //char pStr[3] = "abc";//wrong,need >=4
    char* pStr = "abc"; //wrong, 没有分配空间,
    char temp = 'b';
    pStr[0] = temp;
    printf("%s\n",pStr);
}
```

析构函数能否写成虚函数

1. 基类的可以
2. 子类的不可以，会造成内存泄漏。

sizeof

定义一个空的类型，里面没有任何成员变量和成员函数，但是对该类型求sizeof，得到的结果是多少？

答案：1；

不是0的原因是，当我们声明该类型的实例时，它必须在内存中有一定的空间，否则无法使用这些实例，至少占多少内存，由编译器决定，在VS中，每个空类型的实例占1字节的空间。

面试官：如果在该类型里面添加一个构造函数和析构函数，再对该类型求sizeof，得到的结果又是

多少？

应聘者：1。因为调用构造函数与析构函数只需要知道函数的地址即可，而这些函数的地址只与类型相关，与类型的实例无关，编译器也不会因为这两个函数而在实例中添加额外的信息。

面试官：那如果把析构函数标记为虚函数呢？

应聘者：编译器一旦发现一个类型中有虚函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向该虚函数表的指针，在32位的机器上，一个指针占4字节，因此求sizeof是4，不是4+1，因为只是指向虚函数表的指针的大小，在64位机器上，一个指针占8字节，因此求sizeof是8。

数组与指针的sizeof

sizeof对数组，得到整个数组所占空间的大小。

sizeof对指针，得到指针本身所占空间的大小。

数组的名字就是一个指针，该指针指向数组的第一个元素，我们可以用一个指针来访问数组。也就是数组名加元素下标来访问数组元素或者获取数组元素的值。

```
int data1[] = {1,4,7,9,6};
cout<<sizeof(data1)<<endl;
cout<<"1st of data1:"<<*(data1)<<endl;
cout<<"2st of data1:"<<*(data1 + 1)<<endl;
cout<<"1st of data1:"<<*(data1+3)<<endl;

20
1st of data1:1
2st of data1:4
1st of data1:9
```

class与struct区别

C++中Class成员变量或者成员函数默认private,而struct成员变量默认为public.

C#中，class与struct的成员变量与成员函数都默认是private.

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myints;
    std::cout << "size of myints:" << myints.size() << std::endl;
    std::cout << "sizeof(myints):" << sizeof(myints) << std::endl;
    return 0;
}
size of myints: 5
sizeof(myints): 20
```

String 类

string的构造、拷贝构造、析构、赋值、输出、比较、字符串加、长度、子串

数据结构

数据结构一直是技术面试的重点，大多数面试题都是围绕数组，字符串，链表，树，栈以及队列这几种常用的数据结构展开的，因此每一个面试者都必须熟练掌握着集中数据结构。

数组和字符串是两种最基本的数据结构，他们用连续内存分别储存数据和字符。链表和树是面试中出现频率最高的数据结构，由于操作链表和树需要大量的指针，应聘者在解决相关问题的时候一定要留意代码的鲁棒性，否则容易出现程序崩溃的问题。栈是一个与递归密切相关的数据结构，同样队列也与广度优先遍历算法密切相关，深入理解这两种数据结构能帮助我们解决很多算法问题。

数组

它占用一块连续的内存空间并按照顺序存储数据，创建数据时，我们需要首先指定数组的容量大小，然后根据大小分配内存。即使我们只在数组中存储一个数字，也需要为所有的数据预先分配内存，因此数组的空间效率不是很好，经常会有空闲的区域没有得到充分利用。

数组中的内存是连续的，因此我们可以根据下标在O(1)时间读/写任何元素，因此它的时间效率是很高的。我们可以根据数组时间效率高的优点来实现简单的哈希表，把数组的下标设为哈希表的键值(key)，数组中的数据设为哈希表的value。这样就可以在O(1)时间内实现查找，从而快速，高效地解决很多问题。

Vector

针对数组空间效率不高的问题，人们有设计实现了多种动态数组，比如STL中的vector，为了避免浪费，我们首先为数组开辟较小的空间，然后往数组中添加数据，当数据的数目超过数字的容量时，我们再重新分配一个更大的空间(STL中vector每次扩容时，新的容量都是前一次的两倍)，把之前的数据复制到新的数组中，再把之前的内存释放，这样就减小内存的浪费。每次扩容都需要大量的额外操作，因此尽量减小改变数组容量大小的次数。

简单设计Vector：一开始，新建一个大小为N的数组(弄成数组是保证内存是连续的)，当vector中的元素内存超出开始的数组内存大小时，就新建一个大小为2N的数值，把原来的数组copy到新的数组，再释放原来的内存。用这种节奏来创建更大的数组来容纳更多的vector。

```
void CList::ReplaceEmpty(char str[], int length)
{
    if(str == NULL || length <=0)
        return;
    int count = 0;
    int num = 0;
    while(str[num] != '\0')
    {
        if(str[num] == ' ')
            count++;
        num++;
    }
    int newStringLength = num + 2*count;
    int newNum = newStringLength;
    if(newStringLength >= length)
        return;
```

```

for(int n = num; n >= 0; n--)
{
    if(str[n] == ' ')
    {
        str[newNum--] = '0';
        str[newNum--] = '2';
        str[newNum--] = '%';
    }
    else
        str[newNum--] = str[n];
}
}

```

二叉树：

树的遍历

1. 前序遍历: 先访问根结点, 再访问左子结点, 最后访问右子结点。
2. 中序遍历: 先访问左子结点, 再访问根结点, 最后访问右子结点。
3. 后序遍历: 先访问左子结点, 再访问右子结点, 再访问根结点。

前中后表示访问根结点的时间。

- 前序遍历: 先访问根结点, 再访问左子结点, 最后访问右子结点。
图 2.5 中的二叉树的前序遍历的顺序是 10、6、4、8、14、12、16。
- 中序遍历: 先访问左子结点, 再访问根结点, 最后访问右子结点。
图 2.5 中的二叉树的中序遍历的顺序是 4、6、8、10、12、14、16。
- 后序遍历: 先访问左子结点, 再访问右子结点, 最后访问根结点。
图 2.5 中的二叉树的后序遍历的顺序是 4、8、6、12、16、14、10。

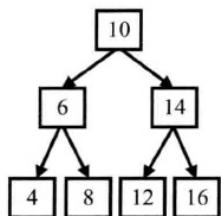


图 2.5 一个二叉树的例子

根据前序, 中序来重建树:

如图 2.7 所示，前序遍历序列的第一个数字 1 就是根结点的值。扫描中序遍历序列，就能确定根结点的值的位置。根据中序遍历特点，在根结点的值 1 前面的 3 个数字都是左子树结点的值，位于 1 后面的数字都是右子树结点的值。

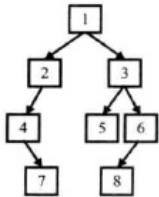


图 2.6 根据前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}重建的二叉树

前序中，第一个点就是根结点，根据根结点的值，在中序中找到根结点的值，则根结点之前的是左子树，根结点之后的是右子树。根据中序排列中左子树的大小(L)，在前序排列中找到左子树(从1到L)，因此对于左子树，又得到了前序和中序排列。对右子树也可以得到前序与中序排列。由此递归实现。

由于在中序遍历序列中，有 3 个数字是左子树结点的值，因此左子树总共有 3 个左子结点。同样，在前序遍历的序列中，根结点后面的 3 个数字就是 3 个左子树结点的值，再后面的所有数字都是右子树结点的值。这样我们就在前序遍历和中序遍历两个序列中，分别找到了左右子树对应的子序列。

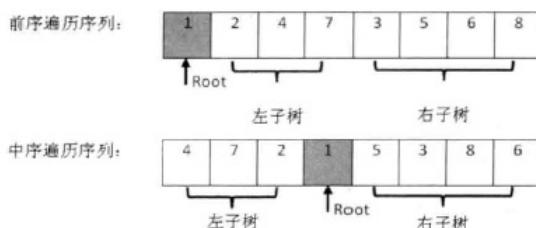


图 2.7 在二叉树的前序遍历和中序遍历的序列中确定根结点的值、左子树结点的值和右子树结点的值

```

BinaryTreeNode* CTree::ConstructBinaryTree(int* preorderS, int* preorderEnd, int* inorderS, int
* inorderEnd)
{
    int rootValue = preorderS[0];
    int* rootNode = inorderS;

    while(rootNode <= inorderEnd && *rootNode != rootValue) //find root in inorder
    {
        ++rootNode;
    }
}
  
```

```

    }
    int LTLen = rootNode - inorderS;
    BinaryTreeNode* RootTree = new BinaryTreeNode();
    RootTree->m_nValue = rootValue;
    if(LTLen >0)
    {
        RootTree->m_pLeft = ConstructBinaryTree(preorderS+1, preorderS+LTLen, inorderS, rootNode-1);
    }
    if(LTLen < preorderEnd - preorderS)
    {
        RootTree->m_pRight = ConstructBinaryTree(preorderS+LTLen + 1, preorderEnd, rootNode+1, inorderEnd);
    }
    return RootTree;
}

void CTree::PreOrderPrint(BinaryTreeNode* BTree)
{
    if(BTree != NULL)//print root first
    {
        m_preorder.push_back(BTree->m_nValue);
        cout<<BTree->m_nValue<<endl;
    }
    if(BTree->m_pLeft != NULL)
    {
        PreOrderPrint(BTree->m_pLeft);
    }
    if(BTree->m_pRight != NULL)
    {
        PreOrderPrint(BTree->m_pRight);
    }
}

void CTree::InOrderPrint(BinaryTreeNode* BTree)
{
    if(BTree->m_pLeft != NULL)
    {
        InOrderPrint(BTree->m_pLeft);
    }
    if(BTree != NULL)//print root second
    {
        m_preorder.push_back(BTree->m_nValue);
        cout<<BTree->m_nValue<<endl;
    }
    if(BTree->m_pRight != NULL)
    {
        InOrderPrint(BTree->m_pRight);
    }
}

void CTree::PostOrderPrint(BinaryTreeNode* BTree)
{
    if(BTree->m_pLeft != NULL)
    {

```

```

        PostOrderPrint(BTree->m_pLeft);
    }

    if(BTree->m_pRight != NULL)
    {
        PostOrderPrint(BTree->m_pRight);
    }

    if(BTree != NULL)//print root last
    {
        m_postorder.push_back(BTree->m_nValue);
        cout<<BTree->m_nValue<<endl;
    }
}

```

< >

栈与队列

栈:先进后出, 压入push,弹出pop

队列:后进先出, 压入push,弹出pop

用两个栈来实现队列。模板类的实现要在.h里面

```

#pragma once
#include <stack>
using namespace std;

template <class T>
class CStackDeque
{
public:
    CStackDeque(void){};
    ~CStackDeque(void){};
    void appendTail(const T& node);
    T deleteHead();

private:
    stack<T> m_stack1;
    stack<T> m_stack2;
};

template<class T>
void CStackDeque<T>::appendTail(const T& node)
{
    m_stack1.push(node);
}

template<class T>
T CStackDeque<T>::deleteHead()
{
    if(m_stack2.size() <= 0)
    {
        while(m_stack1.size() > 0)
    }
}

```

```

    {
        T& node = m_stack1.top();
        m_stack1.pop();
        m_stack2.push(node);
    }
}

if(m_stack2.size() <= 0)
    throw new exception("queue is empty!");

T node2 = m_stack2.top();
m_stack2.pop();
return node2;
}
void main()
{
    CStackDeque<int> queue;
    queue.appendTail(1);
    int value = queue.deleteHead();
}

```

排序算法

核心是在数组中选择一个值，重新排列数组，使得该值左边的元素都小于该值，右边的元素都大于该值。分成两组后，再继续这种操作，以此递归下去，最终得到全排序。

下面都是对数组操作，因此要注意下标问题。

```

int CSort::Partition(int data[], int length, int start, int end)
{
    if(start >= end)
        return start;
    if(data == NULL || start < 0 || end > length || length <= 0)
        throw new exception("Invaidate parameters!");
    int small = start;
    srand((unsigned)time(NULL)); //random seed
    int randInt = rand()%(end - start);
    int value = data[start + randInt];
    cout<<"Value: "<<value<<endl;
    swap(data[start+randInt],data[end]); //all element start from start,
    for(int index = start; index < end; index++)
    {
        if(data[index] < value)
        {
            swap(data[index], data[small]);
            small++;
        }
    }
    swap(data[small], data[end]);
    return small;
}

void CSort::QuicikSort(int data[], int length, int start, int end)

```

```

{
    if(end <= start)
        return;

    int index = Partition(data,length,start,end);
    if(index>start)
    {
        Quicksort(data,length,start, index-1);
    }
    if(index < end)
    {
        Quicksort(data, length, index + 1 ,end);
    }
}

```

与, 或, 异或

负数的二进制表示

举例:

-5 在计算机中表达为: 11111111 11111111 11111111 11111011。转换为十六进制:

0xFFFFFFFF。

-1在计算机中如何表示:

先取1的原码:00000000 00000000 00000000 00000001

得反码: 11111111 11111111 11111111 11111110

得补码: 11111111 11111111 11111111 11111111

可见, -1在计算机里用二进制表达就是全1。16进制为:0xFFFFFFFF。

注:十六进制前缀是0x。

表 2.1 与、或、异或的运算规律

与 (&)	$0 \& 0 = 0$	$1 \& 0 = 0$	$0 \& 1 = 0$	$1 \& 1 = 1$
或 ()	$0 0 = 0$	$1 0 = 1$	$0 1 = 1$	$1 1 = 1$
异或 (^)	$0 ^ 0 = 0$	$1 ^ 0 = 1$	$0 ^ 1 = 1$	$1 ^ 1 = 0$

左移运算符 $m << n$ 表示把 m 左移 n 位。左移 n 位的时候, 最左边的 n 位将被丢弃, 同时在最右边补上 n 个 0。比如:

$00001010 << 2 = 00101000$

$10001010 << 3 = 01010000$

计算二进制中1的个数

```

int CRecursion::CountNumber1(int n)
{
    int count = 0;
    while(n)
    {
        if(n&1)

```

```

        count++;
    n = n>>1;
}
return count;
}

```

链表

在O(1)时间内删除一个节点

方法:通过用节点下一个节点来覆盖这个节点,因此不需要查询将被删除节点前面的所有节点。

问题:

如果你传入一个指针,实际上你传入的就是一个物理地址,对于链表而言,你不能在链表头插入一个元素,因为最终出来的时候,指针(的地址)一直没变,因此技术你在函数里面对指针进行赋值,但是出了函数,有没有效果了。

一种会出错的删除做法:

```

void CList::DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted)
{
    if(!pListHead || !pToBeDeleted)
        return;

    if(pToBeDeleted->m_pNext == NULL)
    {
        pToBeDeleted = NULL;
    }
    if(pToBeDeleted->m_pNext != NULL)
    {

        /*pToBeDeleted->m_value = pToBeDeleted->m_pNext->m_value;/出错的做法
        pToBeDeleted->m_pNext = pToBeDeleted->m_pNext->m_pNext;
        delete pToBeDeleted->m_pNext;***因为删除了pToBeDeleted->m_pNext,因此索引到此,访问了空地
        址,程序会崩溃。正确的做法如下

        ListNode* pNext = pToBeDeleted->m_pNext;//将要被删除的节点,在删除之前,复制它的值
        pToBeDeleted->m_value = pToBeDeleted->m_pNext->m_value;
        pToBeDeleted->m_pNext = pToBeDeleted->m_pNext->m_pNext;
        delete pNext;
    }
}

```

对数组中奇偶数据分类排序

```

void CCArry::ReorderOddEven(int* data, int length, bool (*Func)(int))
{

```

```

if(data==NULL || length <= 0)
    return;
int* pBegin = data;
int* pEnd = data + length - 1;

while(pBegin < pEnd)
{
    while(!Func(*pBegin))
    {
        pBegin++;
    }

    while(Func(*pEnd))
    {
        pEnd--;
    }

    int temp = *pEnd;
    *pEnd = *pBegin;
    *pBegin = temp;
}

void main()
{
    const int length = 6;
    int data[] = {4,1,5,3,6,7};
    CCArrary* test_arr = new CCArrary();
    test_arr->ReorderOddEven(data, length, IsEven());//函数指针的调用
    for(int i = 0; i < length; i++)
    {
        cout<<data[i]<<endl;
    }
}
bool IsEven(int value)
{
    return (value&1) == 0;
}

```

为啥要用C++智能指针

1. 内存泄漏：是指操作系统将空间分配给你，但是那个空间被你申请后并没有被使用，并且没有相应的释放语句。也就是该空间不再被任何指针或引用所引用，成为一个幽灵空间，操作系统以为你在控制它，但其实你并没有控制它。
2. 重复释放：程序通过free或delete语句释放已经不属于该程序的空间。这是很危险的，因为第二次free的空间已经被别的程序所使用。所以C++中把这种错误当成了致命错误。
栈上的空间是由系统分配的。堆上的空间是由用户自己分配的。通过new创建，free释放。

C++什么时候该使用new？该注意什么来防止内存泄漏？

打印1到最大的n位数，比如n=2,怎打印1到99.

```

void PrintToMaxofNDigits(int n); //输入要打印的位数
void PrintNumber(char* number); //打印字符串
bool Increment(char* number); //每次增加1看是否越界
void CRecursion::PrintToMaxofNDigits(int n)
{
    if(n <= 0)
        return;

    char *number = new char[n+1];
    memset(number, '0', n);
    number[n] = '\0';
    while(!Increment(number))
    {
        PrintNumber(number);
    }
    delete []number;
}

bool CRecursion::Increment(char* number)
{
    bool isOverFlow = false;
    int numLength = strlen(number);
    int nTakeOver = 0;
    for(int i = numLength - 1; i >=0; i--)
    {
        int nSum = number[i] - '0' + nTakeOver;
        if(i == numLength -1)
            nSum++;

        if(nSum >= 10)
        {
            if(i == 0)
            {
                isOverFlow = true;
            }
            else
            {
                number[i] = nSum - 10 + '0';
                nTakeOver = 1;
            }
        }
        else
        {
            number[i] = nSum + '0';
            break;
        }
    }
    return isOverFlow;
}

void CRecursion::PrintNumber(char* number)
{
    bool isBeginning0 = true;
    int nLength = strlen(number);
}

```

```

for(int i = 0; i < nLength; ++i)
{
    if(isBeginning0 && number[i] != '0')
        isBeginning0 = false;
    if(!isBeginning0)
    {
        printf("%c",number[i]);
    }
}
printf("\n");
}

```

printf("%c",number[i]);打印字符

接下来是递归算法：

通过递归，打印只针对于个位数，也就是index == length -1的位置。

```

void CRecursion::PrintToMaxofNDigitsRecur(int n)
{
    if(n <= 0)
        return;

    char* number = new char[n+1];
    number[n] = '\0';
    for(int i = 0; i < 10; i++)
    {
        number[0] = i+'0';
        PrintToMaxofNDigitsRecursively(number, n, 0);
    }
    delete[] number;
}

void CRecursion::PrintToMaxofNDigitsRecursively(char* number, int length, int index)
{
    if(index == length -1)
    {
        PrintNumber(number);
        return;
    }

    for(int i = 0; i < 10; i++)
    {
        number[index+1] = i + '0';
        PrintToMaxofNDigitsRecursively(number, length, index + 1);
    }
}

```

输入两颗二叉树A和B，判断B是不是A的子结构。

通过递归实现。如果两个头节点相同，则比较两者的左右子树。如果不相同，则把A的左右子树当成树，看他们是否含有B。

```

bool CTree::HasSubtree(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2)
{
    bool result;
    if(pRoot2 == NULL)
        return true;
    if(pRoot1 == NULL)
        return false;

    if(pRoot1->m_nValue == pRoot2->m_nValue)
    {
        result = DoesTree1HaveTree1(pRoot1, pRoot2);
    }
    if(!result)
    {
        result = DoesTree1HaveTree1(pRoot1->m_pLeft, pRoot2);
    }
    if(!result)
    {
        result = DoesTree1HaveTree1(pRoot1->m_pRight, pRoot2);
    }
    return result;
}

bool CTree::DoesTree1HaveTree1(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2)
{
    bool result;
    if(pRoot2 == NULL)
        return true;
    if(pRoot1 == NULL)
        return false;

    if(pRoot1->m_nValue != pRoot2->m_nValue)
    {
        return false;
    }

    return DoesTree1HaveTree1(pRoot1->m_pLeft, pRoot2->m_pLeft)&&DoesTree1HaveTree1(pRoot1->m_pRight, pRoot2->m_pRight);
}

```

输入两个整数序列，第一个是栈的压入序列，第二个是不是栈的弹出序列

比如压入序列是1, 2, 3, 4, 5;则4, 5, 3, 2, 1是可能的弹出序列，而4, 3, 5, 1, 2不是弹出序列。

```

bool CTree::IsPopOrder(const int* pPush, const int* pPop, int nLength)
{
    stack<int> stackData;
    const int* pNextPush = pPush;
    const int* pNextPop = pPop;
    int pushLength = 0;
    while(pushLength < nLength)

```

```
{  
    int topData = *pNextPop;  
    while(*pNextPush != topData)  
    {  
        if(pNextPush == NULL)//no this element in the push list  
            return false;  
        stackData.push(*pNextPush);  
        pNextPush++;  
        pushLength++;  
    }  
    pNextPush++;  
    pushLength++;  
    pNextPop++;  
}  
while(!stackData.empty())  
{  
    if(stackData.top() != *pNextPop)  
    {  
        return false;  
    }  
    stackData.pop();  
    pNextPop++;  
}  
return true;  
}
```

标准模板库

STL这节包含两部分，一个是数据结构，一个是算法。对于这些数据结构，一方面要知道有哪些数据结构，然后要知道每一种数据结构的特征与功能，能实现哪些操作，然后是我们要知道每种数据结构使用的场景。然后要知道这些数据结构实现的原理是什么？比如红黑树这些。最终就是知道原理以后，我们能基于特定任务而创建自己的数据结构。

algorithms，要知道有哪些算法，这些算法的操作对象是什么(迭代器)？哪些数据结构。然后就是熟悉常用的一些算法。

Boost库包含150多个库，其中就有线性代数库uBLAS，Boost是STL的高阶版，当前自己用不到，因此，了解了STL就足够了。如果要用线性代数库，则Eigen可能是一个好的选择。遇到问题，需要开发库，原则来说先看已有的库，基本上已有的库就呢个满足自己的需求了。

参考这篇文章 [C++ STL 一般总结](#)

STL中体现了泛型程序设计的思想，泛型是一种软件的复用技术。

STL的六大组件：

容器(Container)

是一种数据结构，如list,vector,deques,以模板的方法提供，为了访问容器中的数据，可以使用由容器类输出的迭代器。

顺序容器：vector,list deque,string.

关联容器：set,multiset, map, multimap,hash_set,hash_map,hash_multiset,hash_multimap

杂项：stack,queue,valarray,bitset

迭代器(Iterator)

提供了访问容器中对象的方法，例如，可以使用一堆迭代器指定list或vector中的一定范围的对象。迭代器如同一个指针。事实上，C++指针也是一种迭代器。但是迭代器也可以是那些定义了operator*()以及其它类似于指针的操作符地方的类对象。

算法(algorithm)

是用来操作容器中数据的模板函数。列例如，STL中的sort()来对一个vector中的数据进行排序。用find()来搜索一个list中的对象，函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高级复杂容器的任何数据结构上使用。

仿函数(Function object)

仿函数(functor)又称为函数对象(function object)，其实就是重载了()操作符的struct，没有什么特别的地方。

迭代适配器(adaptor)

空间配置器(allocator)

其中主要工作包括1.对象的创建与销毁, 2.内存的获取与释放

STL底层数据结构实现

容器的基本特征

概念：容器是储存其他对象的对象。被储存的对象必须是同一类型。

基本特征：以下用X表示容器类型（后面会讲到），T表示储存的对象类型（如int）；a和b表示为类型X的值；u表示为一个X容器的标识符（如果X表示vector<int>，则u是一个vector<int>对象。）

表达式	返回类型	说明	复杂度
X::iterator	指向T的迭代器类型	满足正向迭代器要求的任何迭代器	编译时间
X u		创建一个名为u的空容器	固定
X()		创建一个匿名空容器	固定
X u(a)		同X u(a);	线性
a.begin()	迭代器	返回指向容器第一个元素的迭代器	固定
a.end()	迭代器	返回指向超尾值的迭代器	固定
a.size()	无符号整型	返回元素个数	固定
a.swap()	void	交换a和b内容	固定
a == b	可转换为bool	如果a和b长度相当且每个元素都相等，则为真	线性
a != b	可转换为bool	返回！(a == b)	线性

顺序容器的基本特征

概念：序列是对基本容器的一种改进，在保持其基础功能上增加一些我们需要的更为方便的功能。

要求：序列的元素必须是严格的线性顺序排序。因此序列中的元素具有确定的顺序，可以执行将值插入到特定位置、删除特定区间等操作。

序列容器基本特征：以下用t表示类型为T（储存在容器中的值的类型）的值，n表示整数，p、q、i和j表示迭代器。

表达式	返回类型	说明
X a(n,t)		声明一个名为a的由n个t值组成的序列
X(n,t)		创建一个由n个t值组成的匿名序列
X a(i,j)		声明一个名为a的序列，并将其初始化为区间[i,j)的内容
X(i,j)		创建一个匿名序列，并将其初始化为区间[i,j)的内容
a.insert(p,t)	迭代器	将t插入到p的前面
a.insert(p,n,t)	void	将n个t插入到p的前面
a.insert(p,i,j)	void	将区间[i,j)的元素插入到p前面
a.erase(p)	迭代器	删除p所指向的元素
a.erase(p,q)	迭代器	删除区间[p,q]中的元素
a.clear()	void	清空容器

不同容器特有的特征

不同容器特有的特征：

表达式	返回类型	含义	支持的容器
a.front()			vector、list、deque
a.back()			vector、list、deque
a.push_front(t)			list、deque
a.push_back(t)			vector、list、deque
a.pop_front(t)			list、deque
a.pop_back(t)			vector、list、deque
a[n]			vector、deque
a.at(t)			vector、deque

data structure

1. array
2. vector
3. deque
4. list
5. stack
6. queue
7. set
8. map
9. pair

Set

set的特性是所有的元素都会根据键值自动排序，set的元素不像map那样同时拥有实值(value)和键值(key)，set元素的键值就是实值，实值就是键值。Set不允许两个元素拥有相同的键值，不呢个通过迭代器修改set元素的值。

multiset和set的唯一区别在于multiset允许键值重复。

algorithms

常规问题

数值的整数次方

$$a^n = a^{(n-1)/2}a^{(n-1)/2}, a \text{ is even}$$

$$a^n = a^{(n-1)/2}a^{(n-1)/2}a, a \text{ is odd}$$

```
double Chapter3::Power(double base, int exponent)
{
    if(base <= 0)
    {
        this->bmInvideSetting = false;
        return 0;
    }
    if(exponent == 0)
    {
        return 1;
    }

    if(exponent < 0)
    {
        base = 1.0/base;
        exponent = -exponent;
    }
    double result = Power(base, exponent>>1);
    result *= result;
    if(exponent & 0x1 == 1)
        result *= base;
    return result;
}
```

数组奇偶分类

奇数在前，偶数在后排列，复杂度 $O(n)$

```
void Chapter3::RecordOddEven(int *pData, unsigned int length)
{
    if(pData == NULL || length == 0)
        return;
    int *pStart = pData;
    int *pEnd = pData + length - 1;
    int temp;
    while(pStart < pEnd)
    {
        if(*pStart & 1 == 1)
        {
            pStart++;
        }
        else
        {
            temp = *pStart;
            *pStart = *pEnd;
            *pEnd = temp;
            pStart++;
            pEnd--;
        }
    }
}
```

```

        }
    else
    {

        temp = *pEnd;
        *pEnd = *pStart;
        *pStart = temp;
        pEnd--;
    }
}
}

```

pStart < pEnd比大小，就是比较内存中位置先后。

一次查找链表倒数第k个节点

```

ListNode* Chapter3::FindKthToTail(ListNode* pHeadList, unsigned int k)
{
    if(pHeadList == NULL)
        return NULL;
    ListNode *KthNode = pHeadList;
    int num = 0;

    while(++num<k)
    {
        pHeadList = pHeadList->m_pNext;
        if(pHeadList ==NULL)
            return NULL;
    }
    while(pHeadList != NULL)
    {
        pHeadList = pHeadList->m_pNext;
        if(pHeadList ==NULL)
            return KthNode;
        KthNode = KthNode->m_pNext;
    }
}

```

```

//test FindKthToTail
int _tmain(int argc, _TCHAR* argv[])
{
    Chapter3* chp = new Chapter3();
    ListNode* p;
    ListNode* head = new ListNode();
    p = head;
    for(int n = 0; n < 10; n++)
    {
        ListNode* node = new ListNode();
        node->m_value = n;
        p->m_pNext = node;
    }
}

```

```

    p = node;
}
p->m_pNext = NULL;
int k_th = 2;
ListNode *kthNode = chp->FindKthToTail(head, k_th);
cout << "Last "<< k_th << " is:"<<kthNode->m_value<<endl;
return 0;
}

```

树

二叉搜索树的后序遍历

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果，是返回true，否则返回false。假设输入的数组的任意两个数字都不相同。

二叉搜索树的特征是结点的值大于所有左子树的值，且小于所有右子树的值。如果是后序遍历，则最后一个数是根节点的值，这个值通过一刀把它前面的数组切分成左右两部分，左部分的值都小于该值，为树的左子树，右边部分的值都大于该值，是右子树。如果这个划分不存在，则输入数组不是某二叉树的后序遍历。

例如输入数组{5, 7, 6, 9, 11, 10, 8}，则返回 true，因为这个整数序列是图 4.6 二叉搜索树的后序遍历结果。如果输入的数组是{7, 4, 6, 5}，由于没有哪棵二叉搜索树的后序遍历的结果是这个序列，因此返回 false。

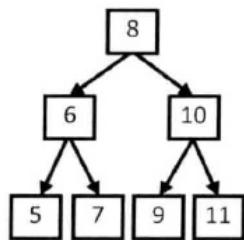


图 4.6 后序遍历序列 5、7、6、9、11、10、8 对应的二叉搜索树

```

bool CTree::VerifySequenceOnBST(int sequence[], int length)
{
    if(sequence == NULL || length< 0)
        return false;

    int rootValue = sequence[length -1];
    int leftTreeLength = 0;
    for(int i = 0; i < length; i++)//finding the point can devide the sequence into 2 parts
    {
        if(sequence[i] > rootValue && leftTreeLength == 0)
        {
            leftTreeLength = i+1;
            break;
        }
    }
    for(int i = leftTreeLength; i < length; i++)
    {
        if(sequence[i] < rootValue)
            return false;
    }
}

```

```

        }
    }
    //make sure the right part large than the root value
    for(int i = leftTreeLength; i < length; i++ )
    {
        if(sequence[i] < rootValue)
            return false;
    }

    bool left = true;
    bool right = true;
    if(leftTreeLength > 0)
        left = VerifySequenceOnBST(sequence, leftTreeLength);

    if(length - 1 - leftTreeLength > 0)
        left = VerifySequenceOnBST(sequence + leftTreeLength, length - leftTreeLength - 1);

    return (left&&right);
}

```

二叉树中和为某一值的路径

输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。
通过递归来实现。求和到叶结点，看是否符合。符合则打印，不符合则把改节点弹出并返回上一级看右结点是否符合。都不符合则再返回上一级，看右边的子树。

```

void CTree::FindPath(BinaryTreeNode* pRoot, int exceptedSum)
{
    if(pRoot == NULL)
        return;

    vector<int> treePath;
    int currentSum = 0;
    FindPath(pRoot->m_pRight,treePath,currentSum,exceptedSum);
}

void CTree::FindPath(BinaryTreeNode* pRoot, vector<int>& path, int currentSum, int exceptedSum)
{
    currentSum += pRoot->m_nValue;
    if(currentSum > exceptedSum)//can't delete this node, because m_nValue can be negative.
    {
        pRoot =NULL;
        return;
    }
    path.push_back(pRoot->m_nValue);
    if(currentSum == exceptedSum && pRoot->m_pLeft == NULL && pRoot->m_pRight == NULL)
    {
        for(int i=0 ; i< path.size(); i++)
        {
            printf("%c\t",path[i]);
        }
    }
}

```

```

        }
        printf("\n");
    }
    if(pRoot->m_pLeft != NULL)
    {
        FindPath(pRoot->m_pLeft,path,currentSum,exceptedSum);
    }
    if(pRoot->m_pRight != NULL)
    {
        FindPath(pRoot->m_pRight,path,currentSum,exceptedSum);
    }
    path.pop_back();
}

```

从上到下打印二叉树

通过deque实现，在pop_front的同时，把该结点的左右子节点push_back到队列的尾部。这样打印完整个deque

```

void CTree::PrintFromTopToBottom(BinaryTreeNode* pRoot)
{
    if(!pRoot)
        return;

    deque<BinaryTreeNode*> dequeTreeNode;
    dequeTreeNode.push_back(pRoot);
    while(dequeTreeNode.size())
    {
        BinaryTreeNode* currentNode = dequeTreeNode.front();
        dequeTreeNode.pop_front();
        printf("%c",currentNode->m_nValue);

        if(currentNode->m_pLeft)
            dequeTreeNode.push_back(currentNode->m_pLeft);

        if(currentNode->m_pRight)
            dequeTreeNode.push_back(currentNode->m_pRight);
    }
}

```

之字形打印整个二叉树

通过两个stack来实现，比如第1层存于stack1，在打印第一层时，把第一层的子节点(也就是dierceng)存于stack2,打印完stack1,此时stack为空，再打印stack2,,在打印stack2时，把第二层的子节点(也就是第三层)存于stack1。以此循环下去，直到stack1,stack2为空为止。

复杂链表的复制

请实现函数ComplexListNode *Clone(ComplexListNode *pHead),复制一个复杂链表在复杂链表中除了有一个m_pNext指针指向下一个节点,还有一个m_pSibling指针指向链表中的任意节点或者nullptr。节点的C++定义如下。

分三步:

- 1.在原来链表每个结点后面复制前面的节点, 做成一个map,也就是hash表。这样原来每个节点N后面都跟着一个N'。
- 2.为链表中新加的每个N'链接指针m_pSibling。
- 3.把链表结点根据序号的奇偶性分成两部分, 这样就得到了我们想要的clone的链表。返回表头即可。

二叉搜索树与双向链表

将一个二叉搜索树转化成一个排序的双向链表。要求不能添加任何新的结点, 只能调整树中结点指针的指向。如图:

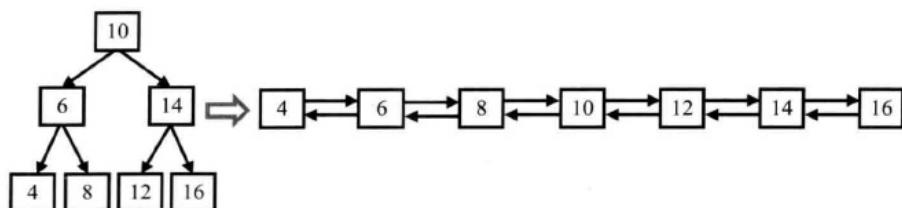


图 4.12 一棵二叉搜索树及转换之后的排序双向链表

通过根结点, 把树分为两部分, 对每部分都转化成一个双向链表, 再最终把这两部分与根节点拼起来, 就得到最终的双向链表。问题转化为两个问题, 1.怎么把树分成两部分再实现递归;2.怎么办把根结点与两个排序好的链表进行链接。

```
BinaryTreeNode* CTree::Convert(BinaryTreeNode* pRoot)
{
    if(pRoot == NULL)
        return;

    BinaryTreeNode* pLastNodeInList = NULL;
    ConvertNode(pRoot, &pLastNodeInList);
    BinaryTreeNode* pHeadofList = pLastNodeInList;
    while(pHeadofList!=NULL &&pHeadofList->m_pLeft !=NULL)
    {
        pHeadofList = pHeadofList->m_pLeft;
    }
    return pHeadofList;
}

void CTree::ConvertNode(BinaryTreeNode* pRoot, BinaryTreeNode** pLastNodeInList)
{
    if(pRoot == NULL)
        return;

    BinaryTreeNode* pCurrentNode = pRoot;
```

```

if(pCurrentNode->m_pLeft!=NULL)
    ConvertNode(pCurrentNode->m_pLeft, pLastNodeInList);
    //左结点转化为双向链表
pCurrentNode->m_pLeft = *pLastNodeInList;//当前节点的右节点连接到双向链表
if(*pLastNodeInList != NULL)
    (*pLastNodeInList)->m_pRight = pCurrentNode;
    //当前节点变成链表最后一个节点
*pLastNodeInList = pCurrentNode;
if(pCurrentNode->m_pRight!=NULL)
    ConvertNode(pCurrentNode->m_pRight, pLastNodeInList);
}

```

字符串排列

输入一个字符串，打印出该字符串中字符的所有排列，比如abc，则所有排列为：
abc,acb,bca,bac,cba,cab.

```

void CRecursion::Permutation(char* pStr)
{
    if(pStr == nullptr)
        return;
    Permutation(pStr,pStr);
}

void CRecursion::Permutation(char* pStr, char* pBegin)
{
    if(*pBegin == '\0')
        printf("%s\n",pStr);
    else
    {
        for(char* pch = pBegin; *pch != '\0'; ++pch)
        {
            char temp = *pch;//pch与pBegin置换
            *pch = *pBegin;
            *pBegin = temp;
            Permutation(pStr, pBegin+1);
            temp = *pch;//还原到原来的样子
            *pch = *pBegin;
            *pBegin = temp;
        }
    }
}

void main(int n)
{
    CRecursion recu;
    char pStr[] = "abc";
    recu.Permutation(pStr);
    printf("Print end!\n");
}

```

序列化二叉树

请实现两个函数，分别用来序列化与反序列化二叉树。若结点的子节点为空，则用\$来代替，如果两个子节点都不存在，则用两个\$来代替。节点之间用","隔开。

为确保根节点在最前面，采取前序遍历来序列化数据。ostream用来文件写操作，从内存中写入到硬盘中。

反序列化则把字符串重构成二叉树。根据读取的是数还是\$来判定是否需要再建子节点。建完左子节点再建右子节点。

```
void CTree::Serialize(BinaryTreeNode* pRoot, ostream& stream)
{
    if(pRoot == NULL)
    {
        stream << "$,";
        return;
    }
    stream << pRoot->m_nValue << ',';
    Serialize(pRoot->m_pLeft, stream);
    Serialize(pRoot->m_pRight, stream);
}

void CTree::DeSerialize(BinaryTreeNode** pRoot, istream& stream)
{
    int number;
    if(ReadStream(stream, &number))
    {
        *pRoot = new BinaryTreeNode();
        (*pRoot)->m_nValue = number;
        (*pRoot)->m_pLeft = nullptr;
        (*pRoot)->m_pRight = nullptr;

        DeSerialize(&(*pRoot)->m_pLeft, stream);
        DeSerialize(&(*pRoot)->m_pRight, stream);
    }
}

bool CTree::ReadStream(istream& stream, int* number)
{
    if(stream.eof())
        return false;

    char buffer[32];
    buffer[0] = '\0';

    char ch;
    stream >> ch;
    int i = 0;
    while(!stream.eof() && ch != ',')
    {
        buffer[i++] = ch;
        stream >> ch;
    }
    bool isNumber = false;
```

```

    if(i>0 && buffer[0] != '$')
    {
        *number = atof(buffer);
        isNumber = true;
    }
    return isNumber;
}

```

寻找数组中第K大的数据

[参考无序整数数组中找第k大的数](#)

利用快速排序的思想。从数组中随机挑选一个数S,把数组分成大于S的数组SL, 以及小于等于S的数组SR,第一次时间复杂度为N, 如果SL数组大小大于K, 则要找的数据在SL中, 否则在SR中。假设在SL中, 我们再一次分割数组, 这一次平均时间复杂度为N/2, 如果有第三次划分, 怎第三次平均时间复杂度是N/4,这样下来总的时间是2N,因此时间复杂度是O(N).

寻找最小的K个数

利用堆排序来实现时间复杂度为O(Nlog(K+1))的算法。建立大小为K的堆, 则查找的时间复杂度是O(K+1)。

一个问题是怎么维护一个最大堆？最大堆可以在O(1)时间内找到最大值, 在O(log K)时间内插入。下面实现利用了快速排序中的二分法。

```

void CCArry::SmallestKInList(int* numbers, int length, int K)
{
    if(numbers == NULL || length <= 0)
        return;

    int middle = length >> 1;
    int start = 0;
    int end = length -1;
    int index = 0;
    CSort* sort = new CSort();
    while(end > K-1)
    {
        index = sort->Partition(numbers,length,start,end);
        if(index == K-1)
            break;
        else if(index > K -1)
            end = index -1;
        else
            start = index + 1;
    }
    delete sort;
    return;
}

```

寻找频率超过半数的数

实际上就是中位数

```
int CCArry::MoreThanHalfNumSort(int* numbers, int length)
{
    if(numbers == NULL || length <= 0)
        return 0;

    int middle = length >> 1;
    int start = 0;
    int end = length - 1;
    int index = 0;
    CSort* sort = new CSort();

    while(start < end)
    {
        index = sort->Partition(numbers, length, start, end);
        if(index == middle)
            return numbers[middle];
        else if(index > middle)
            end = index - 1;
        else
            start = index + 1;
    }
    return numbers[middle];
}
```

连续子数组最大和。

输入译者整型数组，其中数组里面有正数也有负数，数组中一个或者连续的多个数组组成一个子数组。求所有子数组的和。

通过定义一个中间变量，子数组以*i*为终点的连续子数组的最大和*f[i]*，则我们要求的连续子数组最大和也就是*f[i]*在*i=1,...,n*中的最大值。因为*f[i]*可以用如下递归公式简单的实现。因为问题可以简单的求解。

$$f[i] = f[i - 1] + d[i], \text{ if } f[i-1] > 0$$

$$f[i] = d[i], \text{ if } f[i-1] \leq 0$$

结束