

# AI 岗位基础面试问题

作者：孙峥  
专业：计算机技术  
邮箱：sunzheng2019@ia.ac.cn  
学校：中国科学院大学 (中国科学院)  
学院：人工智能学院 (自动化研究所)

2021 年 7 月 24 日

# Part I

## 基础数学问题

### Question 向量范数

简要叙述向量范数基本公式:  $\|x\|_p = (\sum_{i=1}^N |x_i|^p)^{\frac{1}{p}}$ , 当  $p$  为  $-\infty$  时, 结果为  $\min_i |x_i|$ ; 当  $p$  为  $\infty$  时, 结果为  $\max_i |x_i|$ 。(通过迫敛准则证明)

### Question 矩阵范数

先说明一些相关的知识点: 矩阵范数定义的时候, 有非负性, 绝对齐性, 三角不等式, 还比向量范数多一个相容性。然后引入矩阵的  $F$  范数,  $\|A\|_F^2 = \sum_{i,j=1}^n a_{ij}^2 = \text{tr}(A^T A)$ , 可以验证矩阵的  $F$  范数是矩阵范数。再引入矩阵的  $p$  范数,  $\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} = \max_{\|x\|_p=1} \|Ax\|_p$ , 容易证明这样定义的也是矩阵范数。由于是向量的  $p$  范数导出的矩阵的  $p$  范数, 所以此矩阵范数又称为算子范数(《泛函分析》中有定义)。

上述说明的矩阵范数有以下两个重要性质: (1) 矩阵的  $F$  范数和  $2-$  范数都与向量的  $2-$  范数相容; (2) 所定义的算子范数, 即  $p-$  范数都与向量的  $p-$  范数相容; (3) 任一矩阵范数, 一定存在与之相容的向量范数。

(1) 矩阵的  $1$ -范数:  $\|A\|_1 = \max_{x \neq 0} \frac{\|Ax\|_1}{\|x\|_1} = \max_{1 \leq j \leq n} \sum_{i=1}^n |a_{ij}|$ ; 矩阵每一列上的元素绝对值加和, 再从中取最大值。

(2) 矩阵的  $2$ -范数:  $\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} = \sqrt{\lambda_{\max}(A^T A)} = \sqrt{\max_{1 \leq i \leq n} |\lambda_i|}$ , 其中  $\lambda_i$  为  $A^T A$  的特征值。

(3) 矩阵的无穷范数:  $\|A\|_\infty = \max_{x \neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty} = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$ ; 矩阵每一行上的元素绝对值加和, 再从中取最大值。

1-范数和无穷范数的计算结果可以用‘一列无穷行’记忆, 多提一个矩阵的  $2,1$ -范数,  $\|W\| = \sum_{i=1}^m \sqrt{\sum_{j=1}^n w_{ij}^2}$ , 先计算每一行的  $2-$  范数, 变为向量后再计算向量的  $1-$  范数。

### Question 证明矩阵的 $2$ -范数

定义矩阵的范数:  $\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}$ ,  $A$  是对称正定阵, 证明  $\|A\|_2 = \lambda$ ( $\lambda$  是  $A$  的最大特征值)。

证: 网上可以查找到的证明过程都非常复杂, 需要  $A \geq B, A \leq B$ , 然后导出  $A = B$  的过程, 此处提供一种相对简单的方法。

假设  $A$  是一般矩阵,  $A^T A$  是对称半正定矩阵, 则  $\exists$  正交矩阵  $Q, s.t.$

$$A^T A = Q^T \Lambda Q, \Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n), \lambda_i \geq 0$$

且有:

$$\|Ax\|_2^2 = (Ax)^T (Ax) = x^T A^T A x = x^T Q^T \Lambda Q x = (Qx)^T \Lambda (Qx)$$

由于  $Q$  正交, 且  $\|x\|_2=1$ , 有  $\|Qx\|_2=1$ , 则:

$$\begin{aligned}\|A\|_2^2 &= \max_{\|x\|_2=1} \|Ax\|_2^2 \\ &= \max_{\|x\|_2=1} (Qx)^T \Lambda (Qx) \\ &= \max_{\|y\|_2=1} (y)^T \Lambda (y) \\ &= \max_{\|y\|_2=1} \sum_{i=1}^n y_i^2 \lambda_i \\ &= \lambda_1\end{aligned}$$

当  $A$  是对称正定阵时, 特征值均大于 0。 $A^T A$  可以视为  $f(A)g(A)$ , 其特征值的最大值为  $\lambda_1^2$ ,  $\lambda_1$  是  $A$  特征值的最大值, 证毕。证明过程中用到了正交矩阵不改变向量或矩阵的 2- 范数的性质。假设  $P, Q$  均为正交矩阵, 则  $\|A\|_2 = \|PA\|_2 = \|AQ\|_2 = \|P AQ\|_2$ , 即矩阵的 2- 范数和  $F$ - 范数是正交不变量, 但 1- 范数不是;

## Question 极大似然估计

设  $X = \{x_1, x_2, \dots, x_n\}$ , iid 服从  $U(0, k)$  的均匀分布, 求  $k$  的极大似然估计。

解: {求解极大似然估计, 应该先写出极大似然函数  $\ln(L(\theta))$ , 再对参数  $\theta$  求导即可, 必要时需要验证二阶导。}

$$\begin{aligned}f(X) &= \frac{1}{k^n}, 0 \leq x_i \leq k. \\ \ln L(k) &= -n \ln k, \ln L(k)' = -\frac{n}{k} < 0.\end{aligned}$$

不存在  $k$  的极大似然估计。

## Question 矩阵分解

### 叙述常用的矩阵分解

下面两种分解经常用于线性方程 (未知量等于方程个数) 的求解:

1. 设矩阵  $A \in R^{n \times n}$ , 若  $A$  能分解为一个下三角矩阵  $L$  和一个上三角矩阵  $U$  的乘积, 即  $A = LU$ , 则这种分解成为矩阵  $A$  的三角分解。当  $L$  为单位下三角矩阵 (主对角元素全为 1) 时称为 *Doolittle* 分解; 当  $U$  为单位上三角矩阵 (主对角元素全为 1) 时称为 *Cout* 分解
2. 设矩阵  $A \in R^{n \times n}$  为对称正定阵, 则存在一个非奇异下三角矩阵  $L$ , 使得  $A = LL^T$ , 当限定  $L$  的对角元素为正时, 这种分解是唯一的。

设矩阵  $A \in R^{n \times n}$  为对称正定阵, 则存在惟一的分解,  $A = LDL^T$ , 其中  $L$  是单位下三角矩阵,  $D$  为对角矩阵, 且  $D$  的对角元素都是正数。

下面的内容关于矩阵特征值和特征向量的求解:

1. 乘幂法: 计算矩阵的最大特征值及对应的特征向量, 可以接着用降价法求矩阵的次大特征值和相应的特征向量;
2. 反幂法: 计算非奇异矩阵按模最小特征值及其特征向量。
3. *Givens* 变换: 对于任意  $x = (x_1, x_2, \dots, x_n) \in R^n$ , 当  $x_i$  不等于零时, 可以经过一次平面旋转变换将其化为零, 并同时将第  $j$  个分量变为  $Gx_j = (x_i^2 + x_j^2)^{\frac{1}{2}}$ , 而其他分量保持不变。
4. *Jacobi* 方法是一种求实对称矩阵的全部特征值和相应特征向量的方法 (如果矩阵阶数不高可以使用)。

### 5. Householder 变换 (反射变换)

设  $w \in R^n$ , 且  $\|w\|_2 = 1$ , 则矩阵  $H = I - 2ww^T$  称为 Householder 矩阵或反射矩阵, 这里的  $I$  为  $n$  阶单位矩阵。 $\forall x = (x_1, x_2, \dots, x_n)^T \in R^n$ , 当其后边  $n - r + 1$  个分量不全为零时, 可以经过一次反射变换将其后  $n - r$  的分量化为 0, 且第  $r$  个分量变为:  $+(-)(\sum_{j=r}^n |x_j|^2)^{\frac{1}{2}}$ , 而其余分量保持不变。

*Givens* 变换是把向量中的一个元素变成 0, *Householder* 变换是把向量中多个元素变为 0

### 6. 矩阵的 QR 分解 (用来求特征值)

设  $A \in R^{m*n}$ , 则存在正交矩阵  $Q^{m*m}$  和上三角矩阵  $R^{m*n}$ , 使得  $A = QR$ , 并且在  $A$  是非奇异矩阵,  $R$  的对角元均大于零的条件下, 分解是唯一的。

一般的计算过程为: 先利用 *Householder* 变换将原矩阵化为上 *Hessenberg*(矩阵的下次对角线下方元素均为零), 对于  $n$  维 *Hessenberg* 矩阵, 通常用  $n - 1$  个 *Givens* 变换阵将它化为上三角阵, 此三角阵即为  $R^{m*n}$ , 前面的 *householder* 变换矩阵以及 *Givens* 变换矩阵连乘的结果, 是一个正交矩阵, 转置之后即为  $Q^{m*m}$ 。可使用  $QR$  分解求特征值, 求得特征值之后, 再利用反幂求相应的特征向量。

下面再补个一般的矩阵分解 (*schur* 分解):

1.  $A$  为实矩阵, 则  $A = U^T SU$ , 即  $A$  正交相似于  $S$ ,  $S$  对角线部分都是矩阵块, 矩阵块下方都是零 (对角线为块的上三角阵);

2.  $A$  为复矩阵,  $A = U^T SU$ ,  $A$ 酉相似于  $S$ ,  $S$  为上三角阵。

## Question 多元泰勒展开

### 多元泰勒展开和视频光流之间的关系

一个视频的光流是衡量视频相邻帧之间的变化量, 假设视频的规模是  $T * W * H$ , 则计算出来的光流也是  $T$  帧, 且尺寸是  $W * H$ , 但只有两通道, 分别存储每个像素在  $x$  方向和  $y$  方向的变化量, 具体原理如下 (先给出多元泰勒公式):

$$f(x + \delta x) = f(x) + f'(x)\delta x + \frac{f''(x)}{2!}(\delta x)^2 + o((\delta x)^2)$$

$$F(X + \delta X) = F(X) + \nabla F \cdot \delta X + \frac{1}{2}\delta X^T H \delta X$$

再考虑视频光流的计算, 视频光流的计算基于如下两个假设: (1) 连续的两帧图像之间, 目标像素灰度值不变; (2) 相邻的像素之间有相似的运动。假设某一帧图像在  $t$  时刻  $(x, y)$  处的像素值为  $I(x, y, t)$ ,  $d_t$  时间后, 该像素移动到  $(x + d_x, y + d_y)$ , 此时对应的像素值为  $I(x + d_x, y + d_y, t + d_t)$ , 根据假设有:

$$I(x, y, t) = I(x + d_x, y + d_y, t + d_t)$$

由于相邻帧的时间间隔非常小, 则位移也是无穷小量, 可以用多元泰勒公式进行展开:

$$I(x + d_x, y + d_y, t + d_t) = I(x, y, t) + \nabla I^T \delta X + \epsilon$$

$$= I(x, y, t) + (\frac{\partial I}{\partial x} * d_x + \frac{\partial I}{\partial y} * d_y + \frac{\partial I}{\partial t} * d_t + \epsilon)$$

两边同时除以  $\delta t$ :

$$\frac{\partial I}{\partial x} \frac{\delta x}{\delta t} + \frac{\partial I}{\partial y} \frac{\delta y}{\delta t} + \frac{\partial I}{\partial t} \frac{\delta t}{\delta t} + \frac{\epsilon}{\delta t} = 0$$

两边同时取极限:

$$f_x u + f_y v + f_t = 0$$

其中  $\frac{\partial I}{\partial x} = f_x$ ,  $\frac{\partial I}{\partial y} = f_y$ ,  $\frac{\partial I}{\partial t} = f_t$ ,  $\frac{\delta x}{\delta t} = u$ ,  $\frac{\delta y}{\delta t} = v$ , 整体称为光流方程,  $u, v$  是需要求解的量 (Lucas-Kanada 算法)。

## Question 最优化迭代方法

叙述最优化迭代方法, 如梯度下降, 牛顿法以及共轭梯度法等 (直接考虑多维情况)

### 1. 迭代方向

在最小化目标函数时, 迭代算法需要在当前点找到一个下降方向。定义: 在点  $x^k$  处, 对于向量  $d^k \in R^n 0$ , 若存在  $\bar{\lambda}$ , 使得对任意的  $\lambda \in (0, \bar{\lambda})$  都有  $f(x^k + \lambda d^k) < f(x^k)$ , 则称  $d^k$  为函数  $f$  在  $x^k$  处的一个下降方向。在此定义下, 如果  $f(x)$  是可微的 (可以进行一阶泰勒展开), 则:

$$f(x) = f(x_0) + \nabla f(x_0)^T (x - x_0) + O(\|x - x_0\|)$$

$x - x_0$  可以视为  $\lambda d^k$ , 取  $d^k = \nabla f(x_0)$ , 必有  $f(x) > f(x_0)$ , 所以当前点的梯度方向是一个上升方向, 即  $-\nabla f(x_0)$  是一个下降方向。

可以用上述推导过程说明梯度下降法的原理, 即用一阶泰勒展开来近似目标函数, 找到一个下降方向, 接下来证明梯度方向是最速下降方向。

### 2. 梯度方向是最速下降方向

方向导数定义为:

$$\frac{\partial f}{\partial \vec{l}} = \lim_{\|\vec{l}\| \rightarrow 0} \frac{f(x + \vec{l}) - f(x)}{\|\vec{l}\|},$$

结果是一个标量, 表示函数  $f$  沿着方向  $\vec{l}$  的变化率, 变化率越大表示函数变化越快, 对函数在  $x_0$  处的一阶泰勒展开求方向导数, 结果为:

$$\lim_{\|x - x_0\| \rightarrow 0} \frac{f(x) - f(x_0)}{\|x - x_0\|} = \lim_{\|x - x_0\| \rightarrow 0} \frac{\nabla f(x_0)^T (x - x_0)}{\|x - x_0\|} = \lim_{\|d^k\| \rightarrow 0} \frac{\nabla f(x_0)^T d^k}{\|d^k\|},$$

所以  $d^k = \nabla f(x_0)^T$  是一个最速方向, 结合 1 中负梯度是一个下降方向, 所以  $d^k = \nabla f(x_0)$  是一个最速下降方向。

### 3. 牛顿法

在介绍牛顿法用于最优化迭代算法之前, 先介绍牛顿法求解方程的零点。

#### (1) 牛顿法求解方程的根

牛顿法求解方程的根本还是使用函数在  $x_0$  处的一阶泰勒展开来近似:

$$f(x) = f(x_0) + f'(x_0)(x - x_0)$$

$$\text{令 } f(x) = 0, \text{ 即有 } x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

实际上就是  $x_0$  处的切线与  $x$  轴的交点作为下一个迭代点。

#### (2) 牛顿法用于最优化

和上面不一样的是, 牛顿法用于最优化方法时, 需要将函数  $f$  在  $x_0$  进行二阶泰勒展开:

$$f(x) = f(x_0) + \nabla f(x_0)^T (x - x_0) + \frac{1}{2}(x - x_0)^T G_0(x - x_0)$$

由一阶必要条件, 在极值点处应有  $\nabla f(x^*)^T = 0$

对二阶泰勒展开的结果求梯度 (假设  $G_0$  为对称正定阵):

$$\nabla f(x_0) + G_0(x - x_0) = 0$$

即:

$$x = x_0 - G_0^{-1} \nabla f(x_0).$$

### 4. 共轭梯度法

### 5. 拟牛顿法

## Part II

# 计算机算法设计与分析

首先介绍分治思想，求解问题的大概流程如下：

Q1: 从最简单的 *case* 入手；

Q2: 复杂问题，分解为 *sub-problems*。

如何分解：

1. 看 *Input*: 输入的关键数据结构 (DS, 包括数组、树、有向无环图、图、集合)，决定是否可分；
2. 看 *Output*: 决定能否把解合起来。

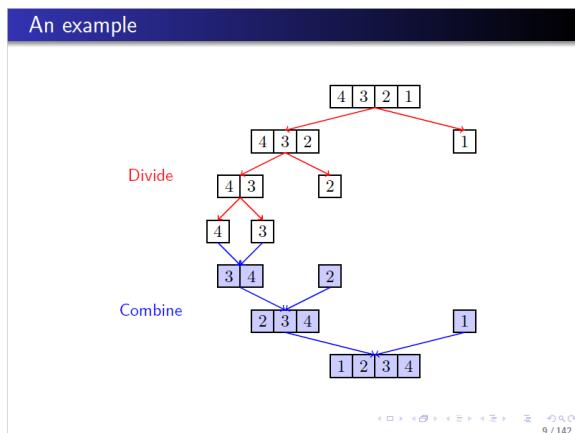
## Question 1

用时间复杂度尽可能少的算法来排序一个  $n$  个整数的数组。

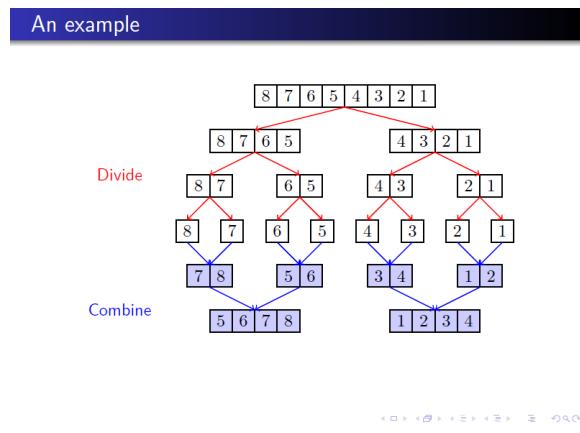
解：(1) 首先想到的是利用冒泡排序，利用两个 for 循环来排序数组，这种方法的时间复杂度是  $O(n^2)$ ，代码较简单，没有递归调用，略去；

(2) 采用 DC(divide and conquer) 思想，每次递归调用数组  $[0, n]$  的前  $n - 1$  个元素，再回溯合并，大致过程如下图所示 (选自卜东波老师上课的 slides)。

合并的时候将末尾的第  $n$  个元素插入前  $n - 1$  个元素当中，时间复杂度为  $O(n)$ ，所以有迭代式：



(a) 从倒数第二个元素位置一分为二



(b) 从中间的位置一分为二

图 1: 采用分治思想排序

$T(n) = T(n - 1) + O(n)$ , 简单推导：

$$\begin{aligned}
 T(n) &\leq T(n - 1) + cn \\
 &\leq T(n - 2) + c(n - 1) + cn \\
 &\leq \dots \\
 &\leq c(1 + 2 + 3 + \dots + n) \\
 &= O(n^2)
 \end{aligned}$$

代码相对简单，略去；

(3) 和 (2) 中方法的分治一样，按照下标来分治，此时分治从该数组的中心位置一分为二，分别对两个子问题排序，分别排好序之后再回溯合并，大致过程如图所示（选自卜东波老师上课的 slides），这实际上就是归并排序（二路归并）。归并的过程可以简单描述为：先准备一个数组，数组容量是两个子问题的规模之和，比较  $a[i]$  和  $b[j]$  的大小，若  $a[i] \leq b[j]$ ，则将第一个有序表中的元素  $a[i]$  复制到  $r[k]$  中，并令  $i$  和  $k$  分别加上 1；否则将第二个有序表中的元素  $b[j]$  复制到  $r[k]$  中，并令  $j$  和  $k$  分别加上 1；如此循环下去，直到其中一个有序表取完；然后再将另一个有序表中剩余的元素复制到  $r$  中从下标  $k$  到最后的单元，大致过程如下（参考 <https://blog.csdn.net/daigualu/article/details/78399168>）。介绍完方法，下面给出实际的可运行代码（C++，在文件夹 code/MergeOrder 中），利用分治和归并排序的思想来排序某一数组，其中的数组规模和元素是自行输入，更加灵活。

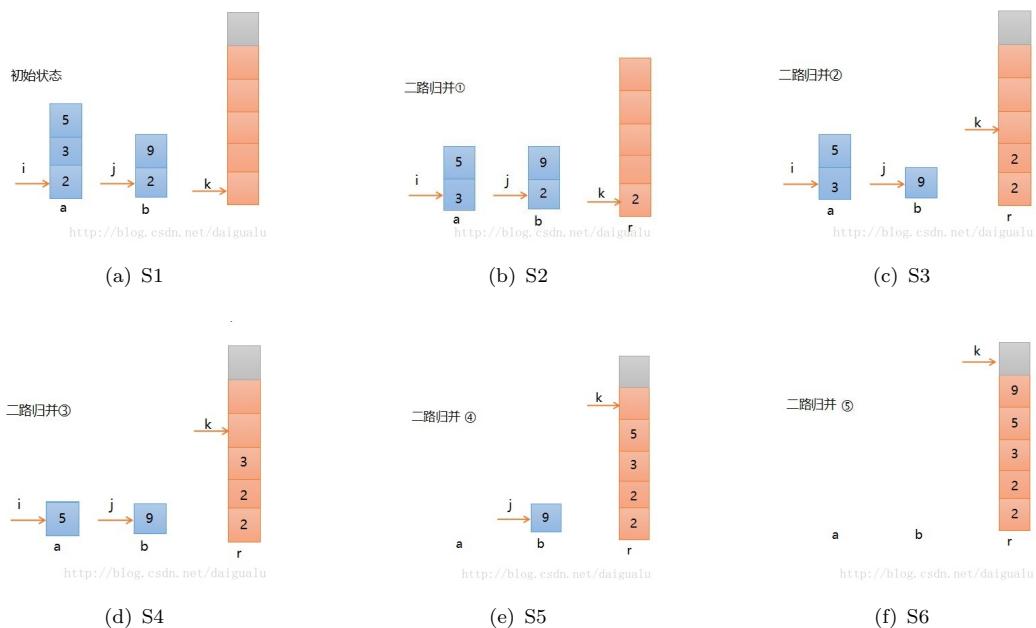


图 2: 二路归并过程

```

1 #include <iostream>
3 #include <stdio.h>
5 using namespace std;
7 void merge(int a[], int left, int mid, int right, int b[])
{
9     int i = mid;
11    int j = right;
13    int k = 0;
15    while (i >= left && j >= mid+1)
17    {
18        if (a[i] > a[j])
19        {
20            b[k++] = a[i--];
21        }
22        else
23        {
24            b[k++] = a[j--];
25        }
26    }
27    while (i >= left)
28    {
29        b[k++] = a[i--];
30    }
31    while (j >= mid+1)
32    {
33        b[k++] = a[j--];
34    }
35}
```

```

23     }
24     while ( i >= left )
25     {
26         b [k++] = a [i--];
27     }
28     while ( j >= mid+1 )
29     {
30         b [k++] = a [j--];
31     }
32     for ( i = 0; i < k; i++ )
33     {
34         a [right - i] = b [i];
35     }
36 }

37 void solve(int a[], int left , int right ,int b[])
38 {
39     if(right > left)
40     {
41         int mid = (right+left) / 2;
42         solve(a, left , mid,b);
43         solve(a, mid + 1, right ,b);
44         merge(a, left , mid, right ,b);
45     }
46 }
47

48 int main()
49 {
50     long int n;//数组维度
51     scanf("%d", &n);
52     int *a = new int[n];
53     int *b = new int[n];
54     for(long int i=0;i<n; i++)
55     {
56         scanf("%d", &a [i]); //scanf的速度要比cin的速度快
57     }
58
59     solve(a,0,n-1,b); //归并排序
60
61     for(int i = 0;i<n; i++)
62         cout<<a [i]<<'\n';
63     return 0;
64 }
65 }
```

Listing 1: 归并排序,C++

上述的代码过程中，两个子问题的归并实际上是从后向前的归并，下面给出从前向后的归并过程，且使用 vector 容器，注意引用符号的使用

```

1 #include<iostream>
2 #include<vector>
3
4 using namespace std;
5
6 class Solution
7 {
8 public:
9     void MergeOrder(vector<int>& v)
10    {
11        vector<int> tool_v(v.size(),0);
12        solve(v, 0,v.size()-1,tool_v);
13    }
14 private:
```

```

15 void solve(vector<int>& a, int left, int right, vector<int>& b)
16 {
17     if(right>left)
18     {
19         int mid = (left+right)/2;
20         solve(a, left, mid, b);
21         solve(a, mid+1, right, b);
22         merge(a, left, mid, right, b);
23     }
24 }
25 void merge(vector<int>& a, int left, int mid, int right, vector<int>& b)
26 {
27     int i = left;
28     int j = mid+1;
29     int k = 0;
30     while(i<=mid&&j<=right)
31     {
32         if(a[i]<a[j])
33             b[k++] = a[i++];
34         else
35             b[k++] = a[j++];
36     }
37     while(i<=mid)
38         b[k++] = a[i++];
39     while(j<=right)
40         b[k++] = a[j++];
41     for(int i=0;i<k;i++)
42         a[left+i] = b[i];
43 }
44 };
45
46 int main()
47 {
48     vector<int> v;
49     v.push_back(2);
50     v.push_back(3);
51     v.push_back(5);
52     v.push_back(1);
53     v.push_back(4);
54     v.push_back(6);
55     v.push_back(7);
56     Solution s;
57     s.MergeOrder(v);
58     for(int i=0;i<v.size();i++)
59         cout<<v[i]<<endl;
60     return 0;
61 }

```

## Question 2

*C++ 中输入二维 (多维) 数组的方法。(这不是个具体的问题，只是为了面试要求手写代码的时候可参考)。*

1. 使用 *C++* 中的 *vector* 数据结构, *vector* 是一个动态数组结构, 可以在其中添加或删除元素。在头文件中声明 `#include <vector>`, 定义一维数组 `vector < int > a;`, 定义二维数组 `vector < vector < int > > a;`, 注意最后两个尖括号之间应该有个空格, 使用方法如下:

(1) 数组规模较小时使用:

```

1 vector<vector<int> >vec;
2 vector<int>a;

```

```

3 a.push_back(1);
4 a.push_back(2);
5 vector<int>b;
6 b.push_back(3);
7 b.push_back(4);
8 vec.push_back(a);
9 vec.push_back(b);

```

(2) 数组规模较大，且不需要自行输入；

```

1 vector<vector<int>> arry(6); // 先确定数组的行数
2 for(int i=0;i<arry.size();i++)
3     arry[i].resize(8); // 确定每行的列数
4
5 for(int i=0;i<arry.size();i++)
6     for(int j=0;j<arry[0].size();j++)
7         arry[i][j]=i*j;

```

(3) 数组规模较大，且需要自行输入数组元素；

```

1 int m,n;
2 cin>>m>>n; // 输入时可以中间可以加空格
3 vector<vector<int>> arry;
4 for(int i=0;i<arry.size();i++)
5     for(int j=0;j<arry[0].size();j++)
6         cin>>arry[i][j]; // 输入时每行之间可以回车

```

2. 利用指针生成二维数组，数组名是实际上是一个指针，使用指针来分配指针，使用方法如下：

(1) 一维数组：

```

1 int arraysize; // 数组规模
2 scanf("%d",&arraysize); // 输入数组规模， scanf比cin快很多
3 int *arry=new int[arraysize]; // 数组名是指针
4 for(int count=0;count<arraysize;count++)
5     scanf("%d",&arry[count]);

```

(2) 二维数组

```

1 int row,col;
2 scanf("%d %d", &row, &col);
3 int **arry=new int*[row]; // 指向指针的指针， 申请row个指向int*的指针
4 for(int i=0;i<row;i++)
5 {
6     arry[i]=new int[col]; // arry每个元素都是指针
7     for(int j=0;j<col;j++)
8         scanf("%d",&arry[i][j]);
9 }

```

以上的代码在输入元素时，用的都是 `scanf` 函数，需要声明头文件 `#include <stdio.h>`。使用 `scanf` 函数要比 `cin` 快很多，在很多 OJ 题当中，当自己的算法时间不通过时，可以通过更换输入函数来使代码通过（个人经验）。

## Question 3

利用问题 1 和问题 2 中的方法，来解决数组逆序数计算问题（包括数组显著逆序数计算问题）。

解：这是第 1 题第 (3) 种方法归并排序的引用。计算（显著）逆序数：可以在归并排序一个数组时，进行（显著）逆序数的计算，显著逆序数就是  $a_i > k * a_j, i < j$ ，网上找到的逆序数计算的代码和显著逆序数的可能不一样。此处把逆序数的计算也当成显著逆序数的一种来统一计算。代码如下：

```
1 #include <iostream>
2 #include <stdio.h>
3 using namespace std;
4
5 long int merge(int a[], int left, int mid, int right, int b[])
6 {
7     int i = mid;
8     int j = right;
9     long int lcount = 0;
10    while (i >= left && j > mid)
11    {
12        if (a[i] > (long long) 3 * a[j])
13        {
14            lcount += j - mid;
15            i--;
16        }
17        else
18        {
19            j--;
20        }
21    }
22    i = mid;
23    j = right;
24    int k = 0;
25    while (i >= left && j > mid)
26    {
27        if (a[i] > a[j])
28        {
29            b[k++] = a[i--];
30        }
31        else
32        {
33            b[k++] = a[j--];
34        }
35    }
36    while (i >= left)
37    {
38        b[k++] = a[i--];
39    }
40    while (j > mid)
41    {
42        b[k++] = a[j--];
43    }
44    for (i = 0; i < k; i++)
45    {
46        a[right - i] = b[i];
47    }
48    return lcount;
49 }
50
51 long int solve(int a[], int left, int right, int b[])
52 {
53     long int cnt = 0;
54     if (right > left)
55     {
56         int mid = (right+left) / 2;
57         cnt += solve(a, left, mid, b);
58         merge(a, mid+1, right, b);
59     }
60 }
```

```

59     cnt += solve(a, mid + 1, right, b);
60     cnt += merge(a, left, mid, right, b);
61 }
62 return cnt;
63 }
64 long int InversePairs(int a[], int len)
65 {
66     int *b=new int[len];
67     long int count=solve(a,0,len-1,b);
68     delete [] b;
69     return count;
70 }
71 int main()
72 {
73     long int n;//数组维度
74     int *array;//数组
75     scanf("%d", &n);
76     array = new int[n];
77     for(long int i=0;i<n;i++)
78         scanf("%d", &array[i]);
79
80     long int count = InversePairs(array, n);
81     printf("%d", count);
82     return 0;
83 }

```

代码跟归并排序的过程差不多，不同的是在 *merge* 函数中，进行归并排序之前会计算两个子数组之间的显著逆序数个数（两个子数组内的显著逆序数由于递归调用已经计算完毕），就是 *merge* 函数中的第一个 *while* 循环，计算过后要将  $i, j$  设置成原来的值，再进行排序。此处应注意的是，网上关于逆序数（不是显著逆序数）的计算是边排序边计算逆序数，二者是一起的，原因就是顺序规则跟计算逆序数的规则是一致的。所以要计算逆序数，除了把上述代码中 *merge* 函数中第一个 *while* 循环里的 3 改成 1 之外，还可以在排序的同时计算逆序数，由于计算逆序数的代码网上可以很容易找到，此处略去。

## Question 4

### 快速排序算法

**解：**与归并排序按照数组的下标来分治不同，快速排序按照数组的值来分治，即选取 pivot 来排序一个数组。可以和后面的牛客网剑指 offer “最小的 K 个数”一题联系起来。首先设定一个分界值 pivot，通过该分界值将数组分成左右两部分。将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。然后，左边和右边的数据可以独立排序。对于左侧的数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数据也可以做类似处理。重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。

技术路线为：

(1) 设置两个变量  $i, j$ ，排序开始的时候： $i = 0, j = N - 1$ ; (2) 以第一个数组元素作为关键数据，赋值给  $key$ ，即  $key = A[0]$ ; (3) 从  $j$  开始向前搜索，即由后开始向前搜索 ( $j--$ )，找到第一个小于  $key$  的值  $A[j]$ ，将  $A[j]$  和  $A[i]$  的值交换；(4) 从  $i$  开始向后搜索，即由前开始向后搜索 ( $i++$ )，找到第一个大于  $key$  的  $A[i]$ ，将  $A[i]$  和  $A[j]$  的值交换；(5) 重复第 (3)(4) 步，直到  $i = j$ ，((3)(4) 步中，没找到符合条件的值，即 (4) 中  $A[j]$  不小于  $key$ , (4) 中  $A[i]$  不大于  $key$  的时候改变  $j, i$  的值，使得  $j = j - 1, i = i + 1$ ，直至找到为止。找到符合条件的值，进行交换的时候  $i, j$  指针位置不变。另外， $i == j$  这一过程一定正

好是  $i+$  或  $j-$  完成的时候，此时令循环结束)。

代码为：

```
1 #include<iostream>
2 #include<vector>
3
4 using namespace std;
5
6 class Solution{
7 public:
8     void QuickSortOrder(vector<int>& v)
9     {
10         solve(v, 0, v.size()-1);
11     }
12 private:
13     void solve(vector<int>& a, int start, int end)
14     {
15         if(start>=end)
16             return;
17
18         int i = start;
19         int j = end;
20         int pivot = a[i];
21
22         while(i<j)
23         {
24             while(a[j]>pivot&&j>i)
25                 j--;
26             swap(a[i], a[j]);
27
28             while(a[i]<pivot&&i<j)
29                 i++;
30             swap(a[i], a[j]);
31         }
32         solve(a, start, i-1);
33         solve(a, j+1, end);
34     }
35     void swap(int& x, int& y)
36     {
37         int temp = x;
38         x = y;
39         y = temp;
40     }
41 };
42
43 int main()
44 {
45     vector<int> v;
46     v.push_back(2);
47     v.push_back(3);
48     v.push_back(5);
49     v.push_back(1);
50     v.push_back(4);
51     v.push_back(6);
52     v.push_back(7);
53     Solution s;
54     s.QuickSortOrder(v);
55     for(int i=0;i<v.size();i++)
56         cout<<v[i]<<endl;
57     return 0;
58 }
```

## Question 5

计算分治问题时间复杂度的总结，主定理 (Master theorem)

解：第 1 题第 (3) 种方法和第 4 题分别介绍了排序一个数组的方法，一个是按照数组的下标分治，一个是按照数组的值来分治。但是没有分析两种方法的时间复杂度，下面给出一般的分治问题的复杂度分析方法。

假设某个问题的规模为  $n$ ，分成  $a$  个子问题，每个子问题的规模为  $\frac{n}{b}$ （这里的  $a, b$  不一定相等，因为子问题往往有重叠的部分，所以  $a \geq b$ ），有递推式： $T(n) = aT\left(\frac{n}{b}\right) + O(d^n)$ 。

结论见下图（卜东波老师的课上 slides）。

The slide has a dark blue header bar with the text "Master theorem". Below it is a white content area with a blue header "Theorem". The text states: "Let  $T(n)$  be defined by  $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$  for  $a > 1, b > 1$  and  $d > 0$ , then  $T(n)$  can be bounded by:" followed by three numbered cases. At the bottom right of the slide are navigation icons and the text "27 / 142".

**Theorem**

Let  $T(n)$  be defined by  $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$  for  $a > 1, b > 1$  and  $d > 0$ , then  $T(n)$  can be bounded by:

- ① If  $d < \log_b a$ , then  $T(n) = O(n^{\log_b a})$ ;
- ② If  $d = \log_b a$ , then  $T(n) = O(n^{\log_b a} \log n)$ ;
- ③ If  $d > \log_b a$ , then  $T(n) = O(n^d)$ .

图 3: Master theorem

计算该递推式：

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + O(d^n) \\ &\leq aT\left(\frac{n}{b}\right) + cn^d \\ &\leq a[aT\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^n] + cn^d \\ &\leq \dots \\ &\leq cn^d + ac\left(\frac{n}{b}\right)^d + a^2c\left(\frac{n}{b^2}\right)^d + \dots + a^{\log_b n - 1}c\left(\frac{n}{b^{\log_b n - 1}}\right)^d + a\log_b n \\ &\leq cn^d[1 + \frac{a}{b^d} + (\frac{a}{b^d})^2 + \dots + (\frac{a}{b^d})^{\log_b n - 1}] + a^{\log_b n} \end{aligned}$$

对上式中的等比项分类讨论:

(1)  $a < b^d$ , 即  $d > \log_b a$ , 以指数项的第一项计算, 则:

$$\begin{aligned} T(n) &\leq cn^d + a^{\log_b n} \\ &= cn^d + n^{\log_b a} \\ &= O(n^d) \end{aligned}$$

(2)  $a = b^d$ , 即  $d = \log_b a$ , 所有的指数项都要计算, 则:

$$\begin{aligned} T(n) &\leq cn^d \log_b n + a^{\log_b n} \\ &= cn^{\log_b a} \log_b n + a^{\log_b n} \\ &= O(cn^{\log_b a} \log_b n) \\ &= O(n^{\log_b a} \log n) \end{aligned}$$

(3)  $a > b^d$ , 即  $d < \log_b a$ , 以指数项的最后一项计算, 则:

$$\begin{aligned} T(n) &\leq cn^d \left(\frac{a}{b^d}\right)^{\log_b n - 1} + a^{\log_b n} \\ &= \frac{cn^d}{\frac{a}{b^d}} n^{\log_b \frac{a}{b^d}} + n^{\log_b a} \\ &= \frac{c}{a} (nb)^d n^{\log_b \frac{a}{b^d}} + n^{\log_b a} \\ &= \frac{cb^d}{a} n^d n^{\log_b a - \log_b b^d} + n^{\log_b a} \\ &= \frac{cb^d}{a} n^d \frac{n^{\log_b a}}{n^d} + n^{\log_b a} \\ &= \frac{cb^d}{a} n^{\log_b a} + n^{\log_b a} \\ &= O(n^{\log_b a}) \end{aligned}$$

命题证毕。

## Question 6

用时间复杂度尽可能少的算法找出一个数组中第  $k$  个小的元素

**解:** 首先想到的是对数组进行排序, 即可以找出数组中第  $k$  小的元素。上述已经介绍  $n$  个元素的数组最快的排序算法(归并排序和快速排序)时间复杂度为  $O(n \log n)$ , 此处介绍更快的解决此问题的算法, 时间复杂度为  $O(n)$ 。

快速排序使用了分治法的策略。它的基本思想是, 选择一个基准数(一般称之为枢纽元), 通过一趟排序将要排序的数据分割成独立的两部分: 在枢纽元左边的所有元素都不比它大, 右边所有元素都比它大, 此时枢纽元就处在它应该在的正确位置上了。在本问题中, 假设有  $N$  个数存储在数组  $a$  中。我们从  $a$  中随机找出一个元素作为枢纽元, 把数组分为两部分。其中左边元素都不比枢纽元大, 右边元素都不比枢纽元小。此时枢纽元所在的位置记为  $mid$ 。如果右半边(包括  $a[mid]$ )的长度恰好为  $k$ , 说明  $a[mid]$  就是需要的第  $k$  大元素, 直接返回  $a[mid]$ 。如果右半边(包括  $a[mid]$ )的长度大于  $k$ , 说明要寻找的第  $k$  大元素就在右半边, 往右半边寻找。如果右半边(包括  $a[mid]$ )的长度小于  $k$ , 说明要寻找的

第 k 大元素就在左半边，往左半边寻找。(下面代码没有调试过，可能会报错)

```
#include<iostream>
using namespace std;

int divide(int a[], int start, int end)
{
    if (start >= end)
    {
        return;
    }
    int i = start;
    int j = end;
    int pivot = a[start];

    while (i < j)
    {
        while (a[j] > pivot && j > i)
        {
            j--;
        }
        int temp1 = a[i];
        a[i] = a[j];
        a[j] = temp1;

        while (a[i] < pivot && i < j)
        {
            i++;
        }
        int temp2 = a[i];
        a[i] = a[j];
        a[j] = temp2;
    }
    return i; // 此时 i=j
}

int findKMax(int a[], int low, int high, int k)
{
    int mid = divide(a, low, high); // 包括 a[mid] 的右半边长度
    int length_of_right = high - mid + 1;
    if (length_of_right == k)
    {
        return a[mid];
    }
    else if (length_of_right > k)
    {
        // 右半边长度比 k 长，说明第 k 大的元素还在右半边，因此在右半边找
        return findKMax(a, mid + 1, high, k);
    }
    else
    {
        return findKMax(a, low, mid - 1, k - length_of_right);
    }
}

int main()
{
    int A[] = { 1, 2, 2, 2, 3, 3, 3 };
    int n = 7, k = 3;
    int result = findKMax(A, 0, n - 1, k);
    cout << "第" << k << "大的数字为" << result << endl;
    system("pause");
    return 0;
}
```

## Question 7

leetcode 第 33 题，搜索旋转排序数组。问题描述：假设按照升序排序的数组在预先未知的某个点上进行了旋转。例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2]。搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回-1。可以假设数组中不存在重复的元素，算法时间复杂度必须是  $O(\log n)$  级别。

```
1 public:
2     int search(vector<int>& nums, int target) {
3         //用二分法，先判断左右两边哪一边是有序的，再判断是否在有序的列表之内
4         if (nums.size() <= 0)
5             return -1;
6         int left = 0;
7         int right = nums.size() - 1;
8         int mid;
9         while(left < right){
10             mid = (right - left)/2 + left ;
11             if(nums[mid] == target)
12                 return mid;
13
14             // 如果中间的值大于最左边的值，说明左边有序
15             if(nums[mid] > nums[left])
16             {
17                 if (nums[left] <= target&&target <= nums[mid])
18                     right = mid;
19                 else
20                     // 这里 +1，因为上面是 <= 符号
21                     left = mid + 1;
22             }
23             // 否则右边有序
24             else
25             {
26                 // 注意：这里必须是 mid+1，因为根据我们的比较方式，mid属于左边的序列
27                 if (nums[mid+1] <= target&&target <= nums[right])
28                     left = mid + 1;
29                 else
30                     right = mid;
31             }
32         }
33         if(nums[left] == target)
34             return left;
35         else
36             return -1;
37     }
38 };
```

c

## Question 8

leetcode 第 153 题，寻找旋转排序数组中的最小值。问题描述：假设按照升序排序的数组在预先未知的某个点上进行了旋转。（例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2]）。找出其中最小的元素。可以假设数组中不存在重复元素。

```
1 class Solution {
2     public:
3         int findMin(vector<int>& nums) {
4             //用二分法，先判断左右两边哪一边是无序的，再判断是否在有序的列表之内
```

```

1   int min = nums[0];
2   if (nums.size() <= 0)
3       return -1;
4   int left = 0;
5   int right = nums.size() - 1;
6   int mid;
7   while(left < right)
8   {
9       if(right == left+1)
10      {
11          if(nums[right]<min)
12              min = nums[right];
13      }
14      mid = (right - left)/2 + left ;
15      if(nums[mid]<min)
16          min = nums[mid];
17
18      // 如果中间的值大于最左边的值，说明左边有序
19      if(nums[mid] > nums[left])
20      {
21          left = mid;// 此处与33题不一样，此处一定是mid，不能+1
22      }
23      // 否则右边有序
24      else
25      {
26          right = mid;
27      }
28  }
29  return min;
30}
31
32
33
34

```

与上一题不一样，这道题是要找旋转数组的最小值，最小值是要找无序的部分（最小值一定在无序的部分）。当只剩两个元素的时候，右边的数会比较小。或者写成下面形式：

```

1 class Solution {
2     public int findMin(int[] nums) {
3         int l = 0, r = nums.length - 1;
4         while (l < r) {
5             int mid = l + (r - l) / 2;
6             if (nums[mid] > nums[r])
7                 l = mid + 1;
8             else
9                 r = mid;
10        }
11        return nums[l];
12    }
13

```

## Question 9

leetcode 第题，寻找一个数组的众数。问题描述：

## Question 10

leetcode 第 200 题，计算岛屿的个数。问题描述：  
这道题实际上是计算连通域的个数，可以用 DFS 和 BFS 求解。

## Question 11

二叉树的构建 (递归与非递归方法), 先序中序后序遍历 (递归与非递归方法), 叶子节点计数, 深度计算等, 面试时手写代码用得上

先介绍递归方法:

```
#include <iostream>
2 #include <stdio.h>

4 using namespace std;

6 // 定义二叉树结构体
8 struct TreeNode{
10     int data;
11     TreeNode *lchild;
12     TreeNode *rchild;
13 };
14 /*
15 // 先序创建二叉树
16 TreeNode* CreateBiTree(TreeNode *T)
17 {
18     int ch;
19     cin >> ch;
20     if (ch == -1)
21     {
22         T = NULL;
23         return T;
24     }
25     else
26     {
27         T = new TreeNode;
28         T->val = ch;
29         cout << "input" << ch << "'s left son node:" ;
30         CreateBiTree(T->left);
31         cout << "input" << ch << "'s right son node:" ;
32         CreateBiTree(T->right);
33     }
34     return T;
35 }
36 */
37 // 上面这段创建树的代码创建好树再返回, 跟下面的相比少了指针

38 // 先序创建二叉树
39 void CreateBiTree(TreeNode **T)
40 {
41     int ch;
42     cin >> ch;
43     if (ch == -1)
44     {
45         *T = NULL;
46         return;
47     }
48     else
49     {
50         *T = new TreeNode;
51         (*T)->data = ch;
```

```

52     cout << "input" << ch << "'s left son node:";
53     CreateBiTree(&((*T)->lchild));
54     cout << "input" << ch << "'s right son node:";
55     CreateBiTree((&(*T)->rchild));
56 }
57     return;
58 }

//先序遍历
59 void PreOrderBiTree(TreeNode *T)
60 {
61     if (T == NULL)
62     {
63         return;
64     }
65     else
66     {
67         cout << T->data << " ";
68         PreOrderBiTree(T->lchild);
69         PreOrderBiTree(T->rchild);
70     }
71 }

//中序遍历
72 void MiddleOrderBiTree(TreeNode *T)
73 {
74     if (T == NULL)
75     {
76         return;
77     }
78     else
79     {
80         MiddleOrderBiTree(T->lchild);
81         cout << T->data << " ";
82         MiddleOrderBiTree(T->rchild);
83     }
84 }

//后序遍历
85 void PostOrderBiTree(TreeNode *T)
86 {
87     if (T == NULL)
88     {
89         return;
90     }
91     else
92     {
93         PostOrderBiTree(T->lchild);
94         PostOrderBiTree(T->rchild);
95         cout << T->data << " ";
96     }
97 }

//树的深度
98 int TreeDeep(TreeNode *T)
99 {
100     int deep = 0;
101     if (T != NULL)
102     {
103         int leftdeep = TreeDeep(T->lchild);
104         int rightdeep = TreeDeep(T->rchild);
105         deep = leftdeep >= rightdeep?leftdeep+1:rightdeep+1;
106     }
107     return deep;
108 }

```

```

118 //叶子节点个数
119 int LeafCount(TreeNode *T)
120 {
121     static int count;
122     if (T != NULL)
123     {
124         if (T->lchild == NULL && T->rchild == NULL)
125         {
126             count++;
127         }
128         LeafCount(T->lchild);
129         LeafCount(T->rchild);
130     }
131     return count;
132 }
133
134 int main()
135 {
136     cout << "输入第一个节点的值,-1表示没有儿子节点:" << endl;
137     TreeNode* T;
138     CreateBiTree(&T); //加个引用创建之后，T会变化；如果使用上面的，调用函数之后T仍然是空的
139     cout << T->data << endl;
140     system("pause");
141     return 0;
142 }
```

上面给出了两种构建二叉树的写法，第一种构建的二叉树是需要返回的，要注意指针和引用，原理与下面这段代码相同。

```

1 #include <iostream>
2 #include <stdio.h>
3
4 using namespace std;
5
6 void f1(int point)
7 {
8     point = 1;
9 }
10
11 void f2(int * point)
12 {
13     *point = 1;
14 }
15
16 int main()
17 {
18     int a = 0;
19     f1(a);
20     cout << a << endl;
21     f2(&a);
22     cout << a << endl;
23     system("pause");
24     return 0;
25 }
```

## Question 12

寻找二叉树上最远的两个节点的距离。

```
1 int maxDistance = 0;
2 int maxDistance(TreeNode *T)
3 {
4     int distl = 0;
5     int distr = 0;
6     if (T == NULL)
7     {
8         return 0;
9     }
10    if (T->lchild != NULL)
11        distl = maxDistance(T->lchild) + 1;
12    if (T->rchild != NULL)
13        distr = maxDistance(T->rchild) + 1;
14    if (distl + distr > maxDistance)
15        maxDistance = distl + distr;
16    return max(distl, distr);
17 }
```

介绍完分治思想部分的题，接下来的题将会面向动态规划。流程大致如下：

Q1: 从最简单的 case 入手；

Q2: 对大的 case 分解。

如何分解？这实际上是一个多步决策的过程：

1. 解能否逐步构造出来（类似于分治思想中的数据结构和解能否可分）；
2. 目标函数能够分解（与分治思想不同，分治没有目标函数）。

动态规划快的原因是：问题定义可能是指数级多的 case 找最优，DP 可以去除冗余，这一点在具体的问题中会体现。

## Question 13

矩阵乘法。问题描述： $n$  个矩阵  $A_1, A_2, A_3, \dots, A_n$  相乘，矩阵  $A_i$  的规模为  $p_{i-1} * p_i$ ，确定最优的运算顺序（结合律），使得整体运算次数最少（只看乘法，不看加法）。

解： $n$  个矩阵  $A_1, A_2, A_3, \dots, A_n$  相乘，有卡特兰数种运算情况 ( $G(n) = G(1)G(n-1) + G(2)G(n-2) + \dots + G(n-1)G(1)$ )。动态规划求解，即采用多步决策，设  $OPT(i, j)$  表示  $A_i A_{i+1} \dots A_{j-1} A_j$  的最优运算次数。假设从第  $k$  个位置一分为二，即  $(A_i, A_{i+1}, A_{i+2}, \dots, A_k)(A_{k+1}, A_{k+2}, \dots, A_j)$ ，则有递推式：

$$OPT(i, j) = OPT(i, k) + OPT(k+1, j) + p_{i-1} p_k p_j$$

如何选择中间的位置是一个枚举过程，对于每一个位置都要递归地去调用分成的两部分，然后每一部分再进行枚举过程，以此类推，伪代码如下：

此算法的时间复杂度为指数级的，证明如图 5。

指数级时间复杂度很大，实践中并不实用。观察上述递归过程，实际上有很多“冗余”，很多子问题  $OPT(i, j)$  在重复计算。 $i, j$  各有  $n$  种情况，共有  $O(n^2)$  个子问题。每个子问题的最优值可以先存储起来，以空间换时间。如果粗略估计，每个子问题又有  $n$  种划分，故时间复杂度为  $O(n^3)$ 。

Trial 1: Explore the recursion in the top-down manner

```

RECURSIVE_MATRIX_CHAIN( $i, j$ )
1: if  $i == j$  then
2:     return 0;
3: end if
4:  $OPT(i, j) = +\infty;$ 
5: for  $k = i$  to  $j - 1$  do
6:      $q = \text{RECURSIVE\_MATRIX\_CHAIN}(i, k)$ 
7:         +  $\text{RECURSIVE\_MATRIX\_CHAIN}(k + 1, j)$ 
8:         +  $p_{i-1} p_k p_j$ ;
9:     if  $q < OPT(i, j)$  then
10:         $OPT(i, j) = q$ ;
11:    end if
12: end for
13: return  $OPT(i, j)$ ;

```

- Note: The optimal solution to the original problem can be obtained through calling `RECURSIVE_MATRIX_CHAIN(1, n)`.



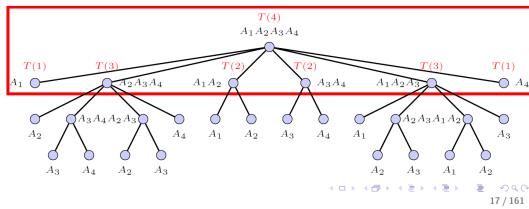
图 4: trial 1

However, this is not a good implementation

## Theorem

*Algorithm RECURSIVE-MATRIX-CHAIN costs exponential time.*

- Let  $T(n)$  denote the time used to calculate product of  $n$  matrices. Then  $T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$  for  $n > 1$ .



(a) 已知的条件

Proof

- We shall prove  $T(n) \geq 2^{n-1}$  using the substitution technique.
    - Basis:  $T(1) \geq 1 = 2^{1-1}$ .

$$\sum_{k=0}^{n-1} \alpha^k = \frac{1 - \alpha^n}{1 - \alpha}$$

$$\geq -1 + \sum_{k=1}^{\lfloor n/2 \rfloor} (I(k) + I(n-k) + 1) \quad (1)$$

$$= n + 2 \sum_{\substack{k=1 \\ \dots \\ n-1}} T(k) \quad (2)$$

$$\geq n + 2 \sum_{k=1}^{\lceil \log_2 n \rceil - 1} 2^{k-1} \quad (3)$$

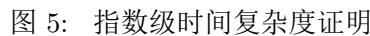
$$\geq n + 2(2^{n-1} - 1) \quad (4)$$

$$\geq n + 2^n - 2 \quad (5)$$

$$\geq 2^{n-1} \quad (6)$$



(b) 数学归纳法证明



## Question 14

背包问题。

假设当包空  $W$ , 前  $i$  个物品的最优价值为  $OPT(i, W)$ , 则有递推式:  $OPT(i, W) = \max\{OPT(i - 1, W), OPT(i - 1, W - w_i) + v_i\}$ , 下面给出伪代码及某个背包问题的示例:

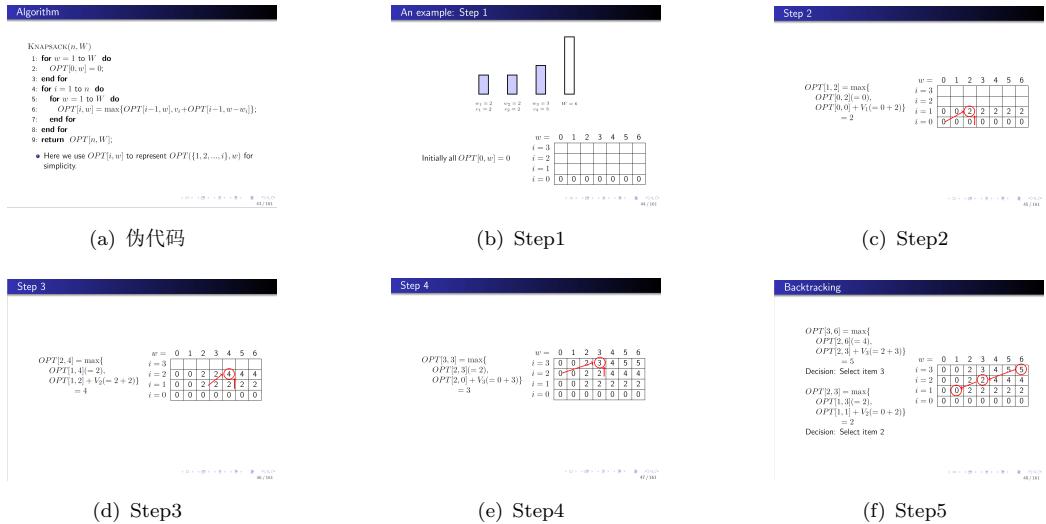


图 6: 伪代码及递归公式示例

可以看出求解背包问题的过程实际上就是按照递推公式求解一个二维矩阵的问题, 下面给出实际运行的代码。(code/Knapsack)

```

1 #include <iostream>
2 #include <string.h>
3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10    int k,W;// 物品的数量和包的总容量
11    cin>>k>>W;// 输入k和W
12    int w[k] = {0};// 第i个位置代表第i+1个物品的重量w(i+1)
13    int v[k] = {0};// 第i个位置代表第i+1个物品的价值v(i+1)
14    int OPT[k+1][W+1];
15    memset(OPT, 0, sizeof(OPT)); // 在头文件#include<string.h>中
16
17    for( int i=0;i<k; i++)
18    {
19        scanf("%d",&w[i]); // 输入各个物品的重量
20    }
21    for( int i=0;i<k; i++)
22    {
23        scanf("%d",&v[i]); // 输入各个物品的价值
24    }
25
26    // 动态规划求解, 实际上在计算一个二维矩阵
27    for( int i = 1;i<=k; i++)
28    {

```

```

29     for( int j = 1;j<=W;j++)
30     {
31         if(j-w[i-1]<0)//如果没有这一条件，下面else中的语句很容易超出索引
32             OPT[i][j] = OPT[i-1][j];
33         else
34             OPT[i][j] = max(OPT[i-1][j],OPT[i-1][j-w[i-1]]+v[i-1]);
35     }
36 }
37 int O = OPT[k][W];
38 for( int i = 0;i<k+1;i++)
39 {
40     for( int j = 0;j<W+1;j++)
41     {
42         cout<<OPT[i][j]<<" ";
43     }
44     cout<<endl;
45 }
46 cout<<O<<endl;
47
48 return 0;
49 }
```

求解完背包问题，顺便解决一个类似的问题，问题描述：一台服务器，空间  $M$ ，内存  $N$ 。现在有若干个任务，每个任务需求  $X_i$  的空间和  $Y_i$  的内存，并且能服务  $U_i$  的人数，在服务器上如何分配任务可以使得同时服务的人数最多。

这道题跟背包问题一样，不过涉及到两个限制条件，所以需要计算一个三维矩阵，假设  $OPT(i, M, N)$  表示将空间  $M$ ， $N$  分配给前  $i$  个任务能够同时服务的最多人数。可以给出递归公式： $OPT(i, M, N) = \max\{OPT(i - 1, M, N), OPT(i - 1, M - m_i, N - n_i) + u_i\}$ ，下面给出代码（在文件 code/Maximum Number of Users）：

```

1 #include <iostream>
2 #include <string.h>
3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10     int k,M,N;//任务的数量和服务器的空间和内存
11     cin>>k>>M>>N;//输入 i 和 M,N
12     int m[k] = {0};//第i个位置代表第i+1个任务需求的空间m(i+1)
13     int n[k] = {0};//第i个位置代表第i+1个任务需求的内存(n(i+1))
14     int u[k] = {0};//第i个位置代表第i+1个任务可以服务的人数u(i+1)
15     int OPT[k+1][M+1][N+1];//三维数组
16     memset(OPT, 0, sizeof(OPT)); //在头文件#include<string.h>中
17
18     for( int i=0;i<k; i++)
19     {
20         scanf("%d",&m[i]); //输入各个任务的空间
21     }
22     for( int i=0;i<k; i++)
23     {
24         scanf("%d",&n[i]); //输入各个任务的内存
25     }
26     for( int i=0;i<k; i++)
27     {
28         scanf("%d",&u[i]); //输入各个任务可以服务的人数
29     }
30 }
```

```

32 //动态规划求解，实际上在计算一个三维矩阵
33 for(int i =1;i<=k; i++)
34 {
35     for(int j = 1;j<=M; j++)
36     {
37         for(int s = 1;s<=N; s++)
38         {
39             if((j-m[i-1]<0) || (s-n[i-1]<0))//如果没有这一条件，下面else中的语句很容易超出索引
40                 OPT[i][j][s] = OPT[i-1][j][s];
41             else
42                 OPT[i][j][s] = max(OPT[i-1][j][s],OPT[i-1][j-m[i-1]][s-n[i-1]]+u[i-1]);
43         }
44     }
45     int O = OPT[k][M][N];
46     cout<<O<<endl;
47     return 0;
48 }

```

上面的代码写的并不好，建议遇到不知道数组规模多大的时候采用动态数组的定义，即用指针来动态定义数组，可以改写成以下形式：

```

1 #include <iostream>
2 #include <string.h>
3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10    int k,M,N;//任务的数量和服务器的空间和内存
11    cin>>M>>N>>k;//输入k和M,N
12    int *m = new int [k];//第i个位置代表第i+1个任务需求的空间m(i+1)
13    int *n = new int [k];//第i个位置代表第i+1个任务需求的内存(n(i+1))
14    int *u = new int [k];//第i个位置代表第i+1个任务可以服务的人数u(i+1)
15    int ***OPT = new int **[k+1];//三维数组
16    for(int i=0;i<k; i++)
17    {
18        scanf("%d",&m[i]);//输入各个任务的空间
19        scanf("%d",&n[i]);//输入各个任务的内存
20        scanf("%d",&u[i]);//输入各个任务的服务的人数
21    }
22    for (int i = 0; i < k + 1; i++)
23    {
24        OPT[i] = new int*[M + 1];
25        for (int j = 0; j < M + 1; j++)
26        {
27            OPT[i][j] = new int[N + 1];
28            for (int s = 0; s < N + 1; s++)
29                OPT[i][j][s] = 0;
30        }
31    }
32
33 //动态规划求解，实际上在计算一个三维矩阵
34 for(int i =1;i<=k; i++)
35 {
36     for(int j = 1;j<=M; j++)
37     {
38         for(int s = 1;s<=N; s++)
39         {

```

```

42     if ((j-m[i-1]<0) || (s-n[i-1]<0)) //如果没有这一条件，下面else中的语句很容易超出索引
43         OPT[i][j][s] = OPT[i-1][j][s];
44     else
45         OPT[i][j][s] = max(OPT[i-1][j][s],OPT[i-1][j-m[i-1]][s-n[i-1]]+u[i-1]);
46     }
47   }
48 int O = OPT[k][M][N];
49
50 cout<<O<<endl;
51 return 0;
52 }
```

上述代码的问题实际开了个三维动态数组，而实际递归计算的时候，只用到了两层的二维数组，所以在计算的时候可以进一步简化，只用两个动态的二维数组就可以代替三维动态数组。代码如下：

```

1 #include <iostream>
2 #include <string.h>
3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10    int k,M,N;//任务的数量和服务器的空间和内存
11    cin>>M>>N>>k;//输入k和M,N
12    int *m = new int [k];//第i个位置代表第i+1个任务需求的空间m(i+1)
13    int *n = new int [k];//第i个位置代表第i+1个任务需求的内存(n(i+1))
14    int *u = new int [k];//第i个位置代表第i+1个任务可以服务的人数u(i+1)
15    int **OPT = new int *[M+1];//二维数组
16    int **newOPT = new int *[M+1];//二维数组
17    for(int i=0;i<k;i++)
18    {
19        scanf("%d",&m[i]);//输入各个任务的空间
20        scanf("%d",&n[i]);//输入各个任务的内存
21        scanf("%d",&u[i]);//输入各个任务的服务的人数
22    }
23
24    for (int i = 0; i < M + 1; i++)
25    {
26        OPT[i] = new int [N + 1];
27        newOPT[i] = new int [N + 1];
28        for (int j = 0; j < N + 1; j++)
29        {
30            OPT[i][j] = 0;
31            newOPT[i][j] = 0;
32        }
33    }
34
35 //动态规划求解，实际上在计算一个三维矩阵
36    for(int i =1;i<=k;i++)
37    {
38        for(int j = 1;j<=M;j++)
39        {
40            for(int s = 1;s<=N;s++)
41            {
42                if ((j-m[i-1]<0) || (s-n[i-1]<0)) //如果没有这一条件，下面else中的语句很容易超出索引
43                    newOPT[j][s] = OPT[j][s];
44                else
45                    newOPT[j][s] = max(OPT[j][s],OPT[j-m[i-1]][s-n[i-1]]+u[i-1]);
46            }
47        }
48    }
49
50 cout<<O<<endl;
51 return 0;
52 }
```

```

47     }
48 }
49 //新的矩阵代替旧的矩阵
50 for( int j = 1;j<=M; j++)
51 {
52     for( int s = 1;s<=N; s++)
53     {
54         OPT[ j ][ s ] = newOPT[ j ][ s ];
55     }
56 }
57
58 int O = newOPT[ M ][ N ];
59 delete []m;
60 delete []n;
61 delete []u;
62 for( int i = 0;i<M+1; i++)
63 {
64     delete []OPT[ i ];
65     delete []newOPT[ i ];
66 }
67 cout<<O<<endl;
68 system(" pause");
69 return 0;
}

```

## Question 15

**序列匹配。问题描述：**获得两个序列如 *occurranc* 和 *occurrence* 的最优匹配 (用打分函数衡量)。打分函数的设置非常重要，假设打分函数设置为：1. 两字母相同，加 1 分；2. 两字母不同，减 3 分；3. 插入或者删除，减 3 分 (如果打分函数设置不合理，那么下面描述的动态规划算法，将会毫无意义。所以打分函数的设置也是一个知识点，这个 pdf 中就不叙述了)。下图给出一个示例和递归公式的推导过程。

### Alignment is useful cont'd

- Application 2: In addition, we can also determine the most likely operations changing "OCCURRENCE" into "OCURRANCE".

- ① Alignment 1:

S':	0-CURRANCE
T':	OCCURRENCE

$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4$$

- ② Alignment 2:

S':	0-CURR-ANCE
T':	OCCURE-NCE

$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1$$

- Thus, the first alignment might describes the real generating process of "OCURRANCE" from "OCCURRENCE".

### The general form of sub-problems and recursions II

- Let's consider the first decision made for  $S_m$  in the optimal solution. There are three cases:

- $S_n$  comes from  $T_m$ : represented as aligning  $S_n$  with  $T_m$ . Then it suffices to align  $T[1..m-1]$  with  $S[1..n-1]$ .
- $S_n$  is an INSERTION: represented as aligning  $S_n$  with a space '|'. Then it suffices to align  $T[1..m]$  and  $S[1..n-1]$ .
- $S_n$  comes from  $T[1..m-1]$ : represented as aligning  $T_m$  with a space '|'. Then it suffices to align  $T[1..m-1]$  and  $S[1..n]$ .

Match/Mutation	Insertion	Deletion
S': $\boxed{O C U R R A N C E}$	S': $\boxed{O C U R R A N C E}$   S': $\boxed{O C U R R A N C E}$ -	
T': $\boxed{O C C U R R E N C E}$	T': $\boxed{O C C U R R E N C E}$ - T': $\boxed{O C C U R R E N C E}$	

(a) 示例

(b) 递归公式推导过程

图 7: 打分函数示例及递归公式推导过程

递归公式及伪代码如下：

### The general form of sub-problems and recursions III

- Summarizing these three examples of sub-problems, we can design the general form of sub-problems as: aligning a **prefix** of  $T$  (denoted as  $T[1..i]$ ) and **prefix** of  $S$  (denoted as  $S[1..j]$ ). Denote the optimal solution value as  $OPT(i, j)$ .

- We can prove the following optimal substructure property:

$$OPT(i, j) = \max \begin{cases} s(T_i, S_j) + OPT(i - 1, j - 1) \\ s(T'_i, S_j) + OPT(i, j - 1) \\ s(T_i, S'_j) + OPT(i - 1, j) \end{cases}$$

76 / 161

(a) 递归公式

77 / 161

### Needleman-Wunsch algorithm [1970]

```
NEEDLEMAN-WUNSCH( $T, S$ )
1: for  $i = 0$  to  $m$  do
2:    $OPT[i, 0] = -3 * i;$ 
3: end for
4: for  $j = 0$  to  $n$  do
5:    $OPT[0, j] = -3 * j;$ 
6: end for
7: for  $j = 1$  to  $n$  do
8:   for  $i = 1$  to  $m$  do
9:      $OPT[i, j] = \max\{OPT[i - 1, j - 1] + s(T_i, S_j), OPT[i - 1, j] - 3, OPT[i, j - 1] - 3\};$ 
10:  end for
11: end for
12: return  $OPT[m, n];$ 
```

Note: the first column is introduced to describe the alignment of prefixes  $T[1..i]$  with an empty sequence  $\epsilon$ , so does the first row.

(b) 伪代码

图 8: alignment 递归公式及伪代码

实际上递归公式就是在算一个二维矩阵，规模为  $(m + 1) * (n + 1)$ ，注意刚开始的时候会有一个空字符表示某个字母跟空字符匹配。在矩阵的最右下角，即  $(OPT(m + 1, n + 1))$  会得到最优匹配的得分。但是如何获得最优得分所对应的匹配，这里需要做一下回溯的过程。下面给出二维矩阵及回溯路线，路线中  $-2$  直接向上到  $1$ ，说明  $-2$  横向对应的  $c$  匹配了空字符。

### General cases

S: '' O C U R R A N C E	
T: ''	0 -3 -6 -9 -12 -15 -18 -21 -24 -27
O	-3 1 -2 -5 -8 -11 -14 -17 -20 -23
C	-6 -2 2 -1 -4 7 -10 -13 -16 -19
C	-9 -5 -1 1 -2 -5 -8 -11 -12 -15
U	-12 -8 -4 0 0 -3 -6 -9 -12 -13
R	-15 -11 -7 -3 1 1 -2 -5 -8 -11
R	-18 -14 -10 -6 -2 2 0 -3 -6 -9
E	-21 -17 -13 -9 -5 -1 1 -1 -4 -5
N	-24 -20 -16 -12 -8 -4 -2 2 -1 -4
C	-27 -23 -19 -15 -11 -7 -5 -1 3 0
E	-30 -26 -22 -18 -14 -10 -8 -4 0 4

Score:  $OPT("OC", "OCUR") = \max \begin{cases} OPT("O", "OCUR") & -3 (= -11) \\ OPT("O", "OCUR") & -1 (= -6) \\ OPT("O", "OCUR") & -3 (= -4) \end{cases}$   
Alignment:  $S' = OCUR$   
 $T' = OC--$

(a) 二维矩阵

### Find the optimal alignment via backtracking

S: '' O C U R R A N C E	
T: ''	0 -3 -6 -9 -12 -15 -18 -21 -24 -27
O	0 1 -2 -5 -8 -11 -14 -17 -20 -23
C	-6 -2 2 -1 -4 -7 -10 -13 -16 -19
C	-9 -5 -1 1 -2 -5 -8 -11 -12 -15
U	-12 -8 -4 0 0 -3 -6 -9 -12 -13
R	-15 -11 -7 -3 1 1 -2 -5 -8 -11
R	-18 -14 -10 -6 -2 2 0 -3 -6 -9
E	-21 -17 -13 -9 -5 -1 1 -1 -4 -5
N	-24 -20 -16 -12 -8 -4 -2 2 -1 -4
C	-27 -23 -19 -15 -11 -7 -5 -1 3 0
E	-30 -26 -22 -18 -14 -10 -8 -4 0 4

Optimal Alignment:  $S' = O-CURRANCE$   
 $T' = OCCURRENCE$

(b) 回溯路线

图 9: 二维矩阵及回溯路线

## Question 16

接着上一题，如果要匹配的两个字符串是两篇文章，那么计算时间和存储空间（需要存储一个二维矩阵）会变得非常巨大，这一题介绍高级动态规划，来优化上一题的算法

## Question 17

leetcode 第 1143 题，最长子序列。问题描述：此题中的最长子序列不是最长子字符串，比如  $text1 = "abcde"$ ,  $text2 = "ace"$ ，最长公共子序列是 "ace"，长度为 3。

解：最长子序列问题是序列匹配的特例，把序列匹配 (alignment) 中的打分函数设置如下：1. 两字母相同，加 1 分；2. 两字母不同，不加分；3. 插入或者删除，不加分，这样设置的打分函数的最终结果就是两个序列最长公共子序列的长度。递推公式的推导过程见下图：

下面准备编写这题的程序，先说明一些关于数据结构字符串 (string) 的相关知识点：

LONGEST COMMON SUBSEQUENCE problem

- The alignment of two sequences  $S[1..m]$  and  $T[1..n]$  reduces to finding the LONGEST COMMON SUBSEQUENCE of them when mutations are not allowed.
- Solution: the longest common subsequence of  $S$  and  $T$ . Let's describe the solving process as multi-stage decision-making process. At each stage, we decide whether align a letter  $S[i]$  with  $T[j]$  or not.
- Let's consider the first decision, i.e., whether we align  $S[m]$  with  $T[n]$  or not. We have two options:
  - Align  $S[m]$  and  $T[n]$  (when  $S[m] = T[n]$ ): Then the subproblem is how to find the longest common subsequence of  $S[1..m-1]$  and  $T[1..n-1]$ .
  - Do not align them: Then we have two subproblems: find the longest common subsequence of  $S[1..m]$  and  $T[1..n-1]$ , and find the longest common subsequence of  $S[1..m-1]$  and  $T[1..n]$ .

107 / 161

(a) 递推过程分析

Basic DP algorithm

- Summarizing these examples, we determine the general form of subproblems as: finding the longest common subsequences of  $S[1..i]$  and  $T[1..j]$  and denote the optimal value as  $OPT(i, j)$ .
- Then we have the following recursion:

$$OPT(i, j) = \begin{cases} 0 & \text{if } i = j = 0 \\ OPT(i-1, j-1) + 1 & \text{if } S[i] = T[j] \\ \max\{OPT(i-1, j), OPT(i, j-1)\} & \text{otherwise} \end{cases}$$

108 / 161

(b) 递推公式

图 10: 递推公式及其推导过程

1. 计算  $string str$  的长度，使用  $str.length()$  函数 (还有其他方法，比如  $str.size()$ ，可以自行百度);
2. 要求自行输入  $string str$ :
  - (1) 使用  $getline(cin, str)$  函数，这个函数包括在头文件  $\#include < string >$  中，或者直接  $cin >> str1 >> str2$ ; 这样输入的两个字符串之间可以用字符隔开;
  - (2) 使用  $scanf("%s", str1)$ , 或者  $scanf("%s", &str1)$ ;
  - (3) 定义一个字符数组  $char a[20]$ , 然后  $scanf("%s", a)$ , 或者  $scanf("%s", &a)$
3. 在字符串  $string str$  的指定位置前面插入字符串，使用  $str.insert(4, "hello")$ ;
4. 在字符串  $string str$  的指定的开始位置 (第一个参数) 替换掉指定长度 (第二个参数) 的字符串，使用  $str.replace(3, 4, "may")$ 。

下面给出实际可以运行的代码 (在文件 `code/LongestCommonSubsequence`) 和改进的代码 (数组采用动态定义):

```
#include <iostream>
1 #include <string>
2 #include <string.h>
3 #include <algorithm>
4 #include <stdio.h>
5
6 using namespace std;
7
8 int main()
9 {
10     string str1, str2;
11     getline(cin, str1);
12     getline(cin, str2);
```

```

14 int m = str1.length(); // 第一个字符串的长度
15 int n = str2.length(); // 第二个字符串的长度
16 int OPT[m+1][n+1]; // 0 矩阵
17 memset(OPT, 0, sizeof(OPT)); // 在头文件 #include<string.h> 中
18 // 计算二维数组，规模为 mn

19 for (int i=1;i<m+1;i++)
20 {
21     for (int j=1;j<n+1;j++)
22     {
23         if (str1[i-1]==str2[j-1])
24         {
25             OPT[i][j] = OPT[i-1][j-1] + 1;
26         }
27         if (str1[i-1]!=str2[j-1])
28         {
29             OPT[i][j] = max(OPT[i-1][j], OPT[i][j-1]);
30         }
31     }
32 }
33 for (int i=0;i<m+1;i++)
34 {
35     for (int j=0;j<n+1;j++)
36     {
37         cout<<OPT[i][j]<<" ";
38     }
39     cout<<endl;
40 }
41 printf("%d", OPT[m][n]);

42 return 0;
43 }
```

```

1 #include <iostream>
2 #include <string>
3 #include <string.h>
4 #include <algorithm>
5 #include <stdio.h>

6 using namespace std;

10 int func(string str1, string str2)
11 {
12     int m = str1.length();
13     int n = str2.length();
14     int **OPT=new int*[m+1]; // 0 矩阵
15     for (int i = 0; i < m + 1; i++)
16     {
17         OPT[i] = new int[n + 1];
18         for (int j = 0; j < n + 1; j++)
19             OPT[i][j] = 0;
20     }

22 // 计算二维数组，规模为 mn

24 for (int i=1;i<m+1;i++)
25 {
26     for (int j=1;j<n+1;j++)
27     {
28         if (str1[i-1]==str2[j-1])
29         {
30             OPT[i][j] = OPT[i-1][j-1] + 1;
31         }
32     }
33 }
```

```

32     }
33     if(str1[i-1]!=str2[j-1])
34     {
35         OPT[i][j] = max(OPT[i-1][j],OPT[i][j-1]);
36     }
37 }
38 return OPT[m][n];
39 }
40
int main()
{
    int k;//一组测试有k对序列
41 scanf("%d", &k);
    int *m = new int[k];
    int*n = new int[k];
    int*solution=new int[k];
42
43 for(int i=0;i<k; i++)
{
    cout<<"输入规模: "<<endl;
    scanf("%d%d",&m[i],&n[i]);
    string str1,str2;
    cout<<"输入字符串: "<<endl;
    cin >> str1 >> str2;//连续输入两个字符串，中间可以有空格
    solution[i] = func(str1,str2);
}
45 for(int i=0;i<k; i++)
    cout<<solution[i]<<endl;
46 delete []m;
47 delete []n;
48 delete []solution;
49 return 0;
50 }

```

上面的代码实际上是在打印了这样一个二维矩阵：

图中的最后一行和最后一列实际上可以省略，所以如果字符串的规模分别为  $m, n$ ，则二维矩阵  $OPT$

## An example

- The memoization table contains many cells with identical value, implying that most subproblems will be solved in the same way.
  - It suffices to calculate cells at limited **matching points**. The entire memoization table could be constructed using these cells.

图 11: 最长公共子序列的动态决策过程

的规模必须为  $m + 1, n + 1$ , 然后利用递推公式来计算二维数组中的各个数值, 最后  $OPT[m][n]$  表示两个字符串的最长公共子序列的长度。下面再给出 *leetcode* 上的类函数的写法(面试时手写代码, 都要写成这种形式。上面那种要输入完整信息的是比较旧的 OJ 测试使用的):

```

1 class Solution {
2     public:
3         int longestCommonSubsequence(string text1, string text2) {
4             int m = text1.length(); // 第一个字符串的长度
5             int n = text2.length(); // 第二个字符串的长度
6             int OPT[m+1][n+1]; // 0 矩阵
7             memset(OPT, 0, sizeof(OPT)); // 在头文件 #include<string.h> 中
8             // 计算二维数组, 规模为 mn
9             for (int i=1; i<m+1; i++) {
10                 for (int j=1; j<n+1; j++) {
11                     if (text1[i-1]==text2[j-1])
12                     {
13                         OPT[i][j] = OPT[i-1][j-1] + 1;
14                     }
15                     if (text1[i-1]!=text2[j-1])
16                     {
17                         OPT[i][j] = max(OPT[i-1][j], OPT[i][j-1]);
18                     }
19                 }
20             }
21             return OPT[m][n];
22         }
23     };
24 }
```

上述代码对应的代码在 *leetcode* 中可以通过, 但是有的 OJ 测试系统中, 对算法的时间和空间复杂度要求非常高, 这种算法不一定能通过, 还可以进一步优化。实际上在计算的时候, 可以像之前背包系列问题那样, 每次运算只涉及到两列或两层, 不用开二维或三维数组。

## Question 18

上一题求最长公共子序列长度中的方法需要计算的量有点多, 浪费了太多的时间和空间, 实际上可以只计算变化的点处的值。

## Question 19

*leetcode* 第 198 题。问题描述: 一个窃贼偷窃一排房子, 每个房子里有一定价值的物品, 但是不同连续偷连续两个房子, 会触发警报, 求窃贼可以偷的最高价值。如果房子改成一圈又如何?

房子并列:

1. 递推式: 房子排成一排, 设  $OPT(n)$  表示偷前  $n$  个房子的最大价值,  $v_n$  表示第  $n$  个房子的价值, 则有递推式:  $OPT(n) = \max\{OPT(n-1), OPT(n-2) + v_n\}$ 。在编程的时候可以开很大的内存来存储  $OPT$ , 或者只用两个变量来计算。

实际代码:

```

1 class Solution {
2     public:
3         int rob(vector<int>& nums) {
4             int k = nums.size();
5             int *OPT = new int [k+1];
6             for(int i = 0;i<k+1;i++)
7             {
8                 OPT[ i ] = 0;
9             }
10            OPT[1] = nums[0];
11            //动态规划求解
12            for(int i = 2;i<k+1;i++)
13            {
14                OPT[ i ] = max(OPT[ i - 1 ],nums[ i - 1 ]+OPT[ i - 2 ]);
15            }
16
17            return OPT[k];
18        }
19    };

```

```

1 class Solution {
2     public:
3         int rob(vector<int>& nums) {
4             int k = nums.size();
5             int curmax = 0;
6             int premax = 0;
7             //动态规划求解
8             for(int i = 0;i<k;i++)
9             {
10                 int temp = curmax;
11                 curmax = max(premax+nums[ i ],curmax);
12                 premax = temp;
13             }
14
15             return curmax;
16         }
17    };

```

伪代码：(只写后一种方法的伪代码)

```

1 rob(nums)
2     k = nums.size();
3     premax = 0;
4     curmax = 0;
5     for i=0 to k do
6         temp = curmax;
7         curmax = max(premax+nums[ i ],curmax);
8         premax = temp;
9     end for
10    return curmax;

```

房子围成一圈：

1. 递推式：将房子围成一圈的问题改成房子排两排的问题。因为第 0 个房子和第  $n$  个房子靠近，所以这两个房子不能同时偷，以此将原问题分成两部分，房子围成一圈时偷窃的所有方法的最大价值 =  $\max\{\text{第 } 0 \text{ 个房子不偷时的所有方法的最大价值}, \text{第 } n \text{ 个房子不偷时的最大价值}\}$ ，然后两个子问题就可以

用房子单排的方法来解答。

2. 代码及伪代码:

```
robs(nums)
2   k=nums.size();
3   nums1 = nums[1,2,3,...,k-1];
4   nums2 = nums[2,3,4,...,k];
5   v1 = rob(nums1);
6   v2 = rob(nums2);
7   return max(v1,v2)
```

```
1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         int k = nums.size();
5         if (k == 0)
6         {
7             return 0;
8         }
9         if (k==1)
10        {
11            return nums[0];
12        }
13        vector<int> v1(nums.begin(),nums.begin() + k-1); // 取vector中一部分元素
14        vector<int> v2(nums.begin() + 1,nums.begin() + k);
15        int vv1 = robp(v1);
16        int vv2 = robp(v2);
17        return max(vv1,vv2);
18    }
19    int robp(vector<int>& num) {
20        int k = num.size();
21        if(k==0)
22        {
23            return 0;
24        }
25        int curmax = 0;
26        int premax = 0;
27        // 动态规划求解
28        for(int i = 0;i<k; i++)
29        {
30            int temp = curmax;
31            curmax = max(premax+num[i],curmax);
32            premax = temp;
33        }
34        return curmax;
35    }
};
```

## Question 20

leetcode 第 334 题。问题描述：在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

1. 递推式

设  $d(p)$  表示根节点为  $p$  的二叉树最优偷盗的金额,  $v(d)$  表示节点  $d$  的价值。分析一个子二叉树  $\{p, l, r, ll, lr, rl, rr\}$ , 容易得到递推式:  $d(p) = \max\{d(l) + d(r), d(ll) + d(lr) + d(rl) + d(rr) + v(d), d(l) + d(rl) + d(rr), d(r) + d(ll) + d(lr)\}$ , 简单分析  $d(r) \geq d(rl) + d(rr)$ ,  $d(l) \geq d(ll) + d(lr)$ , 所以递推式只涉及两项  $d(p) = \max\{d(l) + d(r), d(ll) + d(lr) + d(rl) + d(rr) + v(d)\}$ 。

## 2. 代码及伪代码

实际可运行代码:

```

1 #include <iostream>
2 #include <stdio.h>
3 #include <algorithm>
4
5 using namespace std;
6
7 //定义二叉树结构体
8 struct TreeNode{
9     int data;
10    TreeNode *lchild;
11    TreeNode *rchild;
12};
13
14 //先序创建二叉树
15 void CreateBiTree(TreeNode **T)
16 {
17     int ch;
18     cin >> ch;
19     if (ch == -1)
20     {
21         *T = NULL;
22         return;
23     }
24     else
25     {
26         *T = new TreeNode;
27         (*T)->data = ch;
28         cout << "input" << ch << "'s left son node:" ;
29         CreateBiTree(&((*T)->lchild));
30         cout << "input" << ch << "'s right son node:" ;
31         CreateBiTree(&((*T)->rchild));
32     }
33     return;
34 }
35
36 //用一个数组分别记录偷根节点和不偷根节点时的最大值
37 class Solution {
38 public:
39     int rob(TreeNode* root) {
40         int * res = doRob(root);
41         return max(res[0],res[1]);
42     }
43
44     int * doRob(TreeNode * root)
45     {
46         int * res = new int [2];
47         res[0] = 0;
48         res[1] = 0;
49         if (root == NULL)
50             return res;
51         int* left = doRob(root->lchild);
52         int * right = doRob(root->rchild);
53         //不偷根节点, 最大值为两个子树的最大值之和
54         res[0] = max(left[0],left[1])+max(right[0],right[1]);
55         //偷根节点, 最大值为两个子树不包含根节点的最大值加上根节点的值
56     }
57 }
```

```

56     res[1] = left[0] + right[0] + root->data;
57     return res;
58 }
59 }
60
62 int main()
{
63     cout << "输入第一个节点的值,-1表示没有儿子节点:" << endl;
64     TreeNode* T;
65     CreateBiTree(&T); // 加个引用创建之后, T会变化; 如果使用上面的, 调用函数之后T仍然是空的
66     Solution s;
67     int opt = s.rob(T);
68     cout << opt << endl;
69     system("pause");
70     return 0;
71 }
72 }
```

伪代码:

```

rob(TreeNode *p)
2     if p==NULL
3         return 0;
4     else
5         d(p) = max(d(l)+d(r),d(l1)+d(lr)+d(r1)+d(rr)+v(d))
6     end if
```

## Question 21

有向无环图中的单元最短路径，考虑无负圈和无负边两种情况。

1. 若有向无环图中没有负圈时，分为无圈和有圈(无负圈)，具体看下面两种情况。对于有向无环图来说，

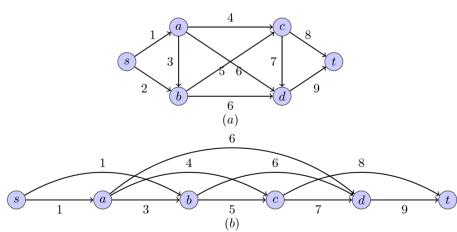
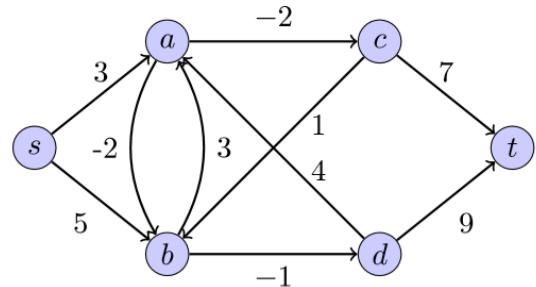


图 3.23: 有向无环图及结点的拓扑排序示例

(a) 无圈



(b) 有圈 (无负圈)

图 12: 有向无环图

可以将所有结点排成一个结点序列，使得每个结点都在其前驱结点之后，即边的方向都是从左向右，这个操作称为线性化 (Linearization)，也称为拓扑排序 (Topological sorting)。这样做的好处是，DAG 相当于是一个数组， $U$  在图中可以到达  $V$ ，则在数组中节点  $U$  在节点  $V$  前面。计算某个节点  $V$  时， $V$  的所有前驱节点都已经计算完毕。

(1). 上面无圈的图中计算起始节点  $s$  到某个节点  $v$  的最短路径，设为  $d(v)$ 。假设求解节点  $s$  到节点  $c$

的最短距离，则由动态规划， $d(c) = \min\{d(a) + r_{ac}, d(b) + r_{bc}\}$ ，依次递归求解最短距离。

(2). 对于有圈的情况，会变得稍微复杂点。比如依次求解  $d(a), d(b)$ ,  $d(a) = \min\{3, 3+d(b), 4+d(d)\}$ ,  $d(b) = \min\{5, -2+d(a), 1+d(c)\}$ ，这样递归求解的时候  $d(a)$  和  $d(b)$  会产生依赖关系。原因是子问题定义的太粗了，需要添加限制条件来细化子问题，使得解没有循环依赖性，加个解的属性做限制。

现在限制路径上的节点不能超过  $k$  个(共有  $k$  个节点)，则路径中一定没有圈。设  $OPT(v, k)$  表示从初始节点  $s$  至多经过  $k$  步到达节点  $v$  的最短路径，则有动态规划递推式： $OPT(v, k) = \min\{OPT(v, k-1), \min_{(v,w) \in E} \{OPT(w, k-1) + d(v, w)\}\}$

对递推式的理解：实际上至多  $k$  步应该包含至多  $k$  步，至多  $k-1$  步，至多  $k-2$  步等等，但是由于在求解至多  $k-1$  步时，即  $OPT(v, k-1)$  时，按照给定的递推式  $OPT(v, k-1) = \min\{OPT(v, k-2), \min_{(v,w) \in E} \{OPT(w, k-2) + d(v, w)\}\}$ ，已经包含了  $OPT(v, k-2)$ ，所以只需分为递推式的两种情况即可。

2. 若有向无环图中不仅没有负圈，而且没有负边的情况请参考贪心算法部分。

## Question 22

上一题中提到子问题定义太粗容易造成递归循环，这一题再给出隐马模型中将子问题定义的细的例子，并简要介绍维特比 (Viterbi) 算法。

隐马尔可夫模型有真实状态值和观测状态值， $a_{kl}$  表示真实状态从  $k$  到  $l$ ,  $a_k$  表示初始真实状态为  $k$ ,  $e_k(b)$  表示真实状态为  $k$  的情况下，观测状态为  $b$ 。解码问题是已知观测序列，寻找最可能的真实状态序列。

1. 定义子问题如下： $v_i = \max_{x_1, x_2, x_3, \dots, x_i} p(x_1, x_2, x_3, \dots, x_i, y_1, y_2, y_3, \dots, y_i)$ ，发现并无递归规律可循，因为子问题定义的范围很大；

2. 定义子问题如下： $v_i(k) = \max_{x_1, x_2, x_3, \dots, x_{i-1}} p(x_1, x_2, x_3, \dots, x_i = k, y_1, y_2, y_3, \dots, y_i)$ ，则可以使用动态规划算法 (Viterbi 算法) 求解，递推公式为： $v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))$ 。

### Viterbi's decoding algorithm: recursion

- First we rewrite  $\max_X P(X, Y)$  as:

$$\max_{x_n} \max_{x_{n-1}} \dots \max_{x_1} e_{x_n}(y_n) a_{x_{n-1}x_n} e_{x_{n-1}}(y_{n-1}) \dots a_{x_1x_2} e_{x_1}(y_1) a_{0x_1}$$

- Let denote  $v_i(k)$  as

$$\max_{x_{i-1}} \dots \max_{x_1} e_k(y_i) a_{x_{i-1}k} e_{x_{i-1}}(y_{i-1}) \dots a_{x_1x_2} e_{x_1}(y_1) a_{0x_1}$$

Then we have:

$$\max_X P(X, Y) = \max_k v_n(k)$$

- We can also observe the following recursion:

$$v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))$$

### Viterbi's decoding algorithm

```

VITERBIDECODING( Y, a, e )
1: Initialize  $v_1(k) = a_{0k} e_k(y_1)$  for all state  $k$ ;
2: for  $i = 2$  to  $n$  do
3:   for each state  $k$  do
4:      $v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))$ ;
5:      $ptr_i(k) = \text{argmax}_l (a_{lk} v_{i-1}(l))$ ;
6:   end for
7: end for
8:  $P(X^*, Y) = \max_k (v_n(k))$ ;
9:  $x_n^* = \text{argmax}_k (v_n(k))$ ;
10: for  $i = n-1$  to 1 do
11:    $x_i^* = ptr_{i-1}(x_{i+1}^*)$ ;
12: end for
13: return  $X$ ;

```

(a) 推导过程

(b) Viterbi 算法

图 13: Viterbi 算法及其推导

## Question 23

给定正整数  $n$ ，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于  $n$ 。你需要让组成和的完全平方数的个数最少。解法一：动态规划

```

1 class Solution {
2     public:
3     int numSquares(int n)
4     {
5         vector<int> dp;
6         for (int i=0;i<=n; i++)
7         {
8             dp.pushback(10); // 先赋值较大的数
9         }
10        dp[0] = 0;
11        for (int i=1;i<=n; i++)
12        {
13            int j=1;
14            while(i-j*j>=0)
15            {
16                dp[i]=min(dp[i],dp[i-j*j]+1);
17                j++;
18            }
19        }
20        return dp[n];
21    }
22}

```

### 解法二：数学理论支持

(1) Lagrange 四平方定理：任何一个正整数都可以表示成不超过四个整数的平方之和。

(2) 满足四数平方和定理的数  $n$ (这里要满足由四个数构成，小于四个不行)，必定满足  $n = 4^a(8b + 7)n$

```

1 bool is_sqrt(long long n)
2 {
3     int m = sqrt(n);
4     if (m*m == n)
5         return true;
6     else
7         return false;
8 }
9 int solve(long long n)
10 {
11     if (is_sqrt(n))
12         return 1;
13     while (n % 4 == 0)
14     {
15         n /= 4;
16         if (n % 8 == 7)
17             return 4;
18         for (int i = 0; i*i < n; i++)
19         {
20             if (is_sqrt(n - i*i))
21                 return 2;
22         }
23     }
24     return 3;
25 }

```

介绍完动态规划算法，下面的题将面向贪心算法

## Question 23

区间调度问题，输入: $n$  个活动  $A = \{A_1, A_2, \dots, A_n\}$ ，每个活动都要占用资源。活动  $A_i$  使用资源区间为  $[S_i, F_i]$ 。活动  $A_i$  产生收益  $W_i$ 。求最优的区间调度，使得收益最大。

解：这道题经常出现在 google 的面试题当中，分两种情况讨论。当收益  $W_i$  不尽相同时采用动态规划，都相同时采用贪心算法。为了简单起见，所有的活动  $A_1, A_2, \dots, A_n$  都已经按照结束时间顺序排列。

(1) 当收益  $W_i$  不尽相同时，设  $OPT(i)$  表示任务  $A_1, A_2, \dots, A_i$  的最优调度， $pre(i)$  表示任务  $A_i$  开始之前就结束的所有任务，则有动态规划递推式： $OPT(i) = \max\{OPT(i-1), OPT(pre(i)) + W_i\}$

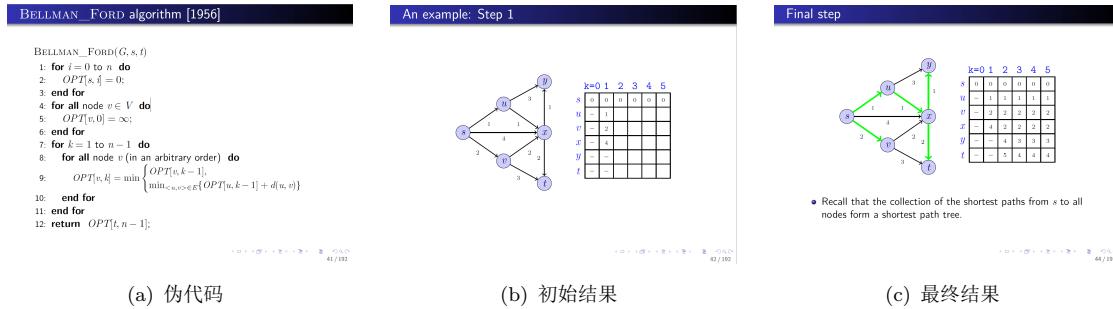
(2) 当收益  $W_i$  都相同时，每次选择的规则是依次选结束最早的活动（或者最晚开始的活动，原理一样），就可以在给定的资源线上尽可能开展最多的活动，即最高的收益。贪心算法每次的选择都是最优的结果。

总结：贪心算法与动态规划算法有不同的地方。(1) 动态规划算法需要回溯，贪心算法不需要；(2) 动态规划算法需要递归地算全局最优，贪心算法是每一步取局部最优，不需要递归，最终结果还是全局最优；(3) 每一个贪心算法背后，都有一个较笨拙的动态规划算法。

## Question 24

最短路径问题。对于有向无环图，当没有负圈时可以采用动态规划求解最短路径；当没有负边时，可以采用动态规划或者贪心算法。

无负圈情况在动态规划部分已经总结，这里介绍无负边的求解方法。无负边首先也可以使用动态规划求解，递推公式与无负圈相同， $OPT(v, k) = \min\{OPT(v, k-1), \min_{(u,v) \in E} OPT(u, k-1) + d(u, v)\}$ 。从伪代码中可以看出，初始状态要将第一列赋值无穷大，第一行赋值 0，然后根据递推公式一列一列



(a) 伪代码

(b) 初始结果

(c) 最终结果

图 14: 伪代码及实例运算过程

地计算。

现在使用贪心算法（戴斯彻算法，Dijkstra 算法）求解无负边的情况，引入集合  $S$ 。每次计算时，在集合  $S$  中的节点不用计算，不在  $S$  中的只计算  $s$  往外一步的最短节点，将其加入  $S$  中。直接给出伪代码。什么时候使用贪心算法求解有最优解？（目标函数是线性的，并且限制条件是 matroid 的，翻译为拟阵。）

## Question 25

例题：给定一组向量  $\{v_1, v_2, \dots, v_n\}$ ，每个向量都有相应的权重  $\{w_1, w_2, \dots, w_n\}$ ，求这组向量中的极大无关组使得权重最大。首先考虑动态规划算法： $OPT(1, 2, \dots, i) = \max\{OPT(1, 2, \dots, i-1), OPT(i^C) + w_i\}$ ， $OPT(i^C)$  表示与  $i$  线性无关的向量集合。

## Dijkstra's algorithm [1959]

```
DIJKSTRA( $G, s, t$ )
1:  $d(s) = 0$ ; //  $d(u)$  stores upper bound of the shortest distance from  $s$ 
   to  $u$ ;
2: for all node  $v \neq s$  do
3:    $d(v) = +\infty$ ;
4: end for
5:  $S = \{\}$ ; // Let  $S$  be the set of explored nodes;
6: while  $S \neq V$  do
7:   Select the unexplored node  $v^*$  ( $v^* \notin S$ ) that minimizes  $d(v)$ ;
8:    $S = S \cup \{v^*\}$ ;
9:   for all unexplored node  $v$  adjacent to an explored node do
10:     $d(v) = \min\{d(v), \min_{u \in S} \{d(u) + d(u, v)\}\}$ ;
11: end for
12: end while

● Lines (9 – 11) are called “relaxing”. That is, we test whether the
shortest-path to  $v$  found so far can be improved by going through  $u$ ,
and if so, update  $d(v)$ .
● When  $d_{u,v} = 1$  for any edge  $< u, v >$ , the Dijkstra's algorithm
reduces to BFS, and thus can be treated as weighted BFS.
```

55 / 192

图 15: Dijkstra 算法

如果  $v$  不是零向量，且权重最大，则最优解中一定含有  $v$ 。使用贪心算法求解：将向量按照权重下降排列，依次选择。如果某个向量与之前的向量线性相关，则跳过，即可获得最优解。

(上述这个问题还有个近似的形式：考虑一个图，每条边都有权重，选择一些边使得权重最大且图中没有圈。)

## Part III

# 机器学习与深度学习基础模型与算法

机器学习模型主要分为两大类：生成式模型和判别式模型。

生成式模型是研究某一类的样本，研究这一类样本的特性利用极大似然方法： $\max_{\theta} p(D|(M(\theta)))$ ，即在什么参数的情况下，能够产生某一类的样本。比如贝叶斯公式  $p(w_i|x) = \frac{p(x|w_i)p(w_i)}{p(x)}$  中的  $p(x|w_i)$  可以根据极大似然估计得到参数值，进而得到分布。

判别式模型是研究所有的样本，利用极大后验方法： $\max_{\theta} p((M(\theta))|D)$ ，即在给定的所有样本的情况下，参数为多少的概率最大。比如深度学习中的神经网络，根据所有的样本来更新参数。

机器学习部分

## Question 分类任务评估方法

叙述常用的分类任务评估指标 1. 模型评价术语

现在假设我们的分类目标只有两类，记为正例 (positive) 和负例 (negative) 分别是：

- (1) True positives(TP): 被正确地划分为正例的个数，即实际为正例且被分类器划分为正例的实例数；
- (2) False positives(FP): 被错误地划分为正例的个数，即实际为负例但被分类器划分为正例的实例数；
- (3) False negatives(FN): 被错误地划分为负例的个数，即实际为正例但被分类器划分为负例的实例数；
- (4) True negatives(TN): 被正确地划分为负例的个数，即实际为负例且被分类器划分为负例的实例数。

		预测类别			
		Yes	No	总计	
实际类别	Yes	TP	FN	P (实际为 Yes)	
	No	FP	TN	N (实际为 No)	
	总计	P' (被分为 Yes)	N' (被分为 No)	P+N	

图 16: 混淆矩阵

## 2. 评价指标

### (1) 正确率 (accuracy)

正确率是最常见的评价指标,  $\text{accuracy} = (\text{TP} + \text{TN}) / (\text{P} + \text{N})$ , 正确率是被分对的样本数在所有样本数中的占比, 通常来说, 正确率越高, 分类器越好;

### (2) 灵敏度 (sensitivity)

$\text{sensitivity} = \text{TP}/\text{P}$ , 表示的是所有正例中被分对的比例, 衡量了分类器对正例的识别能力;

### (3) 特异性 (specificity)

$\text{specificity} = \text{TN}/\text{N}$ , 表示的是所有负例中被分对的比例, 衡量了分类器对负例的识别能力;

### (4) 精度 (precision)

$\text{precision} = \text{TP}/(\text{TP} + \text{FP})$ , 精度是精确性的度量, 表示被分为正例的示例中实际为正例的比例;

### (5) 召回率 (recall)

召回率是覆盖面的度量, 度量有多个正例被分为正例,  $\text{recall} = \text{TP}/(\text{TP} + \text{FN}) = \text{TP}/\text{P} = \text{sensitivity}$ , 可以看到召回率与灵敏度是一样的。精度和召回率反映了分类器分类性能的两个方面。如果综合考虑查准率与查全率, 可以得到新的评价指标 F1-score, 也称为综合分类率:

$$F1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

## 3. 曲线

ROC 曲线是 (Receiver Operating Characteristic Curve, 受试者工作特征曲线) 的简称, 是以灵敏度(即真阳性率, 所有正例中被分对的比例,  $\text{TP}/\text{P}$ )为纵坐标, 以 1 减去特异性 ( $1 - \text{TN}/\text{N} = \text{FP}/\text{N}$ , 即假阳性率  $\text{FP}/\text{N}$ , 所有负例中被分为正的比例, 即负例中被分错的比例) 为横坐标绘制的性能评价曲线。可以将不同模型对同一数据集的 ROC 曲线绘制在同一笛卡尔坐标系中, ROC 曲线越靠近左上角, 说明其对应模型越可靠。也可以通过 ROC 曲线下面的面积 (Area Under Curve, AUC) 来评价模型, AUC 越大, 模型越可靠。

解释: ROC 曲线纵坐标为  $\text{TP}/\text{P}$ , 横坐标为  $\text{FP}/\text{N}$ , 已知  $\text{TP}/\text{P} + \text{FN}/\text{P} = 1$ ,  $\text{FP}/\text{N} + \text{TN}/\text{N} = 1$ , 此时如图中加一条曲线  $x + y = 1$ , 即添加条件  $\text{TP}/\text{P} + \text{FP}/\text{N} = 1$ , 可以得出  $\text{FN}/\text{P} = \text{FP}/\text{N}$ ,  $\text{TN}/\text{N} = \text{TP}/\text{P}$ , 即在相交点有正例和负例有相同的正确率和错误率, 如果相交的点越往左上, 说明横坐标越小, 纵坐标越大, 即正负例错误率都较小, 模型较好。

PR 曲线是 Precision Recall Curve 的简称, 描述的是 precision 和 recall 之间的关系, 以 recall 为横坐标, precision 为纵坐标绘制的曲线。该曲线的所对应的面积 AUC 实际上是目标检测中常用的评价指标平均精度 (Average Precision, AP)。AP 越高, 说明模型性能越好。

# Question 线性回归和逻辑回归

## 1. 线性回归

假设线性回归函数为  $f_{\theta}(x) = \theta^T x$ , 其中  $\theta = (\theta_1, \theta_2, \dots, \theta_n)$ , 损失函数默认为平方误差函数 (假设有  $m$  个样本):

$$J(\theta) = \frac{1}{2m} \sum_{k=1}^m [f_{\theta}(x^{(k)}) - y^{(k)}]^2$$

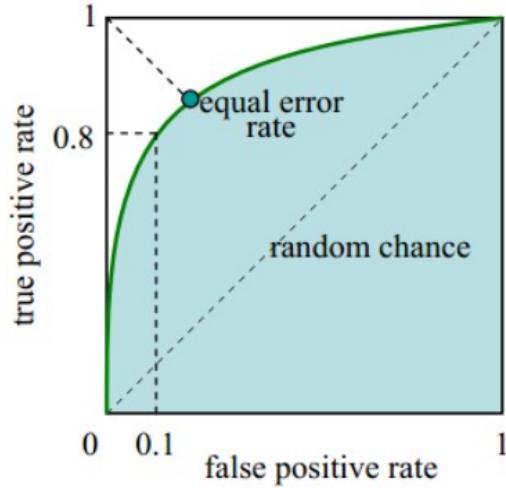


图 17: ROC 曲线

求解的方法有最小二乘法 (将  $\theta$  作为未知量, 用样本数据将目标函数写成  $\|A\theta - b\|^2$  的形式, 进而对  $\theta$  求一阶导, 令一阶导为 0, 得到  $\theta = (A^T A)^{-1} A^T b$ ) 和梯度下降法。(过程都比较简单, 可以面试现场直接推出来)

## 2. 逻辑回归

逻辑回归应用于二分类, 在线性回归的基础上添加一个 sigmoid 函数  $g(x) = \frac{1}{1+e^{-x}}$ ,  $f_\theta(x) = g(\theta^T x)$ , 使得函数值域从  $(-\infty, +\infty)$  变成  $(0, 1)$ , 该激活函数有性质  $y' = y(1 - y)$ 。(注: 深度学习中 sigmoid 可用于多标签分类, softmax 用于多元分类, 也包括二分类) 分类任务损失函数一般采用交叉熵函数  $J(\theta) = -\frac{1}{m} \sum_{k=1}^m [y^{(k)} \ln[f_\theta(x^{(k)})] + (1 - y^{(k)}) \ln[1 - f_\theta(x^{(k)})]]$ 。

关于交叉熵函数的推导:

逻辑回归主要采用极大似然函数方法来解, 连续随机变量的极大似然估计将各个样本的值代入概率密度函数, 再相乘取对数即可。离散样本的极大似然函数只需将各个样本取值的概率求出来, 然后乘在一起, 就是离散情况下的极大似然函数。

逻辑回归模型下, 单个样本的概率密度为  $p(x^{(i)}) = f_\theta(x^{(i)})^y [1 - f_\theta(x^{(i)})]^{1-y}$ , 随后将所有样本的概率乘一起,  $\ln(L(\theta)) = \prod_{k=1}^m f_\theta(x^{(k)})^y [1 - f_\theta(x^{(k)})]^{1-y}$ , 再取对数结果为  $\sum_{k=1}^m [y^{(k)} \ln[f_\theta(x^{(k)})] + (1 - y^{(k)}) \ln[1 - f_\theta(x^{(k)})]]$ , 该似然函数需要取最大值, 转换成代价函数需要加负号取最小值, 再对样本取平均, 即:  $J(\theta) = -\frac{1}{m} \sum_{k=1}^m [y^{(k)} \ln[f_\theta(x^{(k)})] + (1 - y^{(k)}) \ln[1 - f_\theta(x^{(k)})]]$ 。

该代价函数对  $\theta$  求导:

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta} &= -\frac{1}{m} \sum_{k=1}^m [y^{(k)} \frac{1}{f_\theta(x^{(k)})} - (1-y^{(k)}) \frac{1}{1-f_\theta(x^{(k)})}] \frac{\partial f_\theta(x^{(k)})}{\partial \theta} \\
&= -\frac{1}{m} \sum_{k=1}^m [y^{(k)} \frac{1}{f_\theta(x^{(k)})} - (1-y^{(k)}) \frac{1}{1-f_\theta(x^{(k)})}] \frac{\partial g(\theta^T x)}{\partial \theta} \\
&= -\frac{1}{m} \sum_{k=1}^m [y^{(k)} \frac{1}{f_\theta(x^{(k)})} - (1-y^{(k)}) \frac{1}{1-f_\theta(x^{(k)})}] g(\theta^T x) [1-g(\theta^T x)] \frac{\partial \theta^T x}{\partial \theta} \\
&= -\frac{1}{m} \sum_{k=1}^m [y^{(k)} \frac{1}{f_\theta(x^{(k)})} - (1-y^{(k)}) \frac{1}{1-f_\theta(x^{(k)})}] f_\theta(x^{(k)}) [1-f_\theta(x^{(k)})] x^{(k)} \\
&= -\frac{1}{m} \sum_{k=1}^m [y^{(k)} (1-f_\theta(x^{(k)})) - (1-y^{(k)}) f_\theta(x^{(k)})] x^{(k)} \\
&= -\frac{1}{m} \sum_{k=1}^m [y^{(k)} - f_\theta(x)] x^{(k)}
\end{aligned}$$

当模型预测结果  $\theta x^{(k)}$  为 0 时, 即  $g(\theta x) = \frac{1}{2}$ , 从梯度公式中可以看出, 此时与标签  $y^{(k)}$ (0 或 1) 的差值最大, 这一点和 sigmoid 图像上表现的一致。

## Question 贝叶斯决策理论

由于贝叶斯决策比较理论, 也比较好理解, 这里不过多介绍。

## Question

### 概率密度估计

概率密度估计主要分为参数概率密度估计和非参数概率密度估计。参数概率密度估计包括极大似然估计和贝叶斯估计, 极大似然估计一般都是假设样本服从某个分布, 然后估计参数的准确数值; 贝叶斯估计直接估计样本服从的分布。

非参数概率密度估计方法基本思路: 为了估计样本  $x$  处的概率密度函数  $p(x|w_i)$ , 构造一系列包含点  $x$  的区域  $R_1, R_2, \dots, R_n, \dots$ , 记  $v_n$  为  $R_n$  的体积,  $k_n$  为落在  $R_n$  中的样本个数,  $p_n(x)$  表示对  $p(x|w_i)$  的第  $n$  次估计,  $p_n(x) = \frac{k_n/n}{v_n}$ .

由上式, 有两种构造序列的方法来估计概率密度函数:

(1) parzen 窗估计: 固定局部区域体积  $v$ ,  $k$  变化。 $V_n = \frac{1}{\sqrt{v_n}}$ , 实际上是每一次估计时体积不变。体积会逐渐收敛, 要求  $k_n$  和  $k_n/n$  能够保证  $p_n(x)$  能够收敛到  $p(x)$ ;

(2) k 近邻估计: 固定局部样本数  $k$ ,  $V$  变化。 $K_n = \sqrt{n}$ , 实际上是每一次估计时局部样本数目不变。区域会逐渐生长。

计算:

(1) parzen:

$R_n$  是一个  $d$  维的超立方体, 宽度为  $h_n$ ,  $v_n = h_n^d$ ( $x$  是中心点). 如果  $x_i$  落在  $v_n$  中,  $\phi(\frac{x-x_i}{h_n}) = 1$ , 否则为 0, 则  $k_n = \sum_{i=1}^n \phi(\frac{x-x_i}{h_n})$ . 推出  $p_n(x) = \frac{1}{n} \sum_{i=1}^n \frac{1}{v_n} \phi(\frac{x-x_i}{h_n})$ .

(2) k 近邻估计:

$k_i$ : 区域内第  $i$  类的样本数;

- $k$ : 区域内所有的样本数;  
 $n_i$ : 所有样本中第  $i$  类的样本数;  
 $n$ : 所有样本数。

$k$  近邻后验概率密度估计: (计算方式有两种)

$$1. p_n(x, w_i) = \frac{k_i/n}{v}$$

$$p_n(w_i|x) = \frac{p_n(x, w_i)}{p_n(x)} = \frac{p_n(x, w_i)}{\sum_{j=1}^c p_n(x, w_j)} = \frac{\frac{k_i/n}{v}}{\sum_{j=1}^c \frac{k_j/n}{v}} = \frac{k_i}{k}$$

$$2. p_n(x|w_i) = \frac{k_i/n}{v}, p(w_i) = \frac{n_i}{n}, p(x) = \frac{k/n}{v}$$

$$p_n(w_i|x) = \frac{p_n(x|w_i)p(w_i)}{p(x)} = \frac{k_i}{k}$$

## Question 聚类

数据聚类方法分为均值聚类, 分级聚类和谱聚类三种。(这里将重点介绍一些数学方面的推导过程, 而不是像市场课程只介绍技术路线。)

### 一. 均值聚类

技术路线很简单, 开始随机初始化  $k$  个聚类中心, 依次遍历所有样本, 将每个样本归到最近的聚类中心一类, 求该类的均值作为新的聚类中心; 再重新遍历样本, 这样一直迭代直至收敛。下面将介绍为什么采取均值作为新的聚类中心。

从总体样本的概率密度函数出发:  $p(x|\theta) = \sum_{j=1}^c p(x|w_j, \theta_j)P(w_j)$ ,  $P(w_j)$  是各个类别的先验概率, 称为混合比例。这里的主要任务是要估计  $\theta$ , 一旦  $\theta$  得到估计, 可以将上述混合密度分解为多个已知的密度成分, 并且可以采用最大化后验概率来确定样本的类别, 考虑最大似然估计。

## 8.4 最大似然估计

• 目标: 估计一个  $\hat{\theta}$  使  $p(D|\theta)$  最大。

— 考虑对数似然 (log-likelihood):

$$\begin{aligned} f_{lh}(\theta) &= \ln(p(D|\theta)) \\ &= \sum_{k=1}^n \ln(p(x_k|\theta)) \\ &= \sum_{k=1}^n \ln \left( \sum_{j=1}^c p(x_k|\omega_j, \theta_j) P(\omega_j) \right) \end{aligned}$$

$f_{lh}(\theta)$  对参数  $\theta$  的梯度(假定参数独立):

$$\begin{aligned} \nabla_{\theta} f_{lh}(\theta) &= \sum_{k=1}^n \frac{1}{p(x_k|\theta)} \nabla_{\theta} p(x_k|\theta) \\ &= \sum_{k=1}^n \frac{1}{p(x_k|\theta)} \nabla_{\theta} (p(x_k|\omega_i, \theta_i) P(\omega_i)) \\ &= \sum_{k=1}^n \frac{P(\omega_i)}{p(x_k|\theta)} \nabla_{\theta} (p(x_k|\omega_i, \theta_i)) \\ &= \sum_{k=1}^n \frac{P(\omega_i)}{p(x_k|\theta)} p(x_k|\omega_i, \theta_i) \nabla_{\theta} \ln(p(x_k|\omega_i, \theta_i)) \\ &= \sum_{k=1}^n \frac{P(\omega_i, x_k|\theta)}{p(x_k|\theta)} \nabla_{\theta} \ln(p(x_k|\omega_i, \theta_i)) \\ &= \sum_{k=1}^n P(\omega_i | x_k, \theta) \nabla_{\theta} \ln(p(x_k|\omega_i, \theta_i)) \end{aligned}$$

$$p(x_k|\theta) = \sum_{j=1}^c p(x_k|\omega_j, \theta_j) P(\omega_j)$$

仅考虑包含  $\theta_i$  的项

相等

(a) 似然函数

(b) 似然函数的梯度

图 18: 似然函数及梯度

这里推导过程中的  $P(w_i|x_k, \theta)$  不是概率密度, 而是将  $x_k$  代入  $p(w_i|x, \theta)$  函数中的取值。令上述  $c$  个方程等于 0(可以用牛顿法直接求解  $\theta$ ), 以及添加先验概率条件, 可以得到两个条件。

推导过程如下, 采用拉格朗日乘子法求解:

现在做四个假设:

1. 样本服从的先验概率密度为高斯分布, 即:  $\ln p(x|w_i, \mu_i) = -\ln((2\pi)^{\frac{d}{2}} |\Sigma|^{\frac{1}{2}}) - \frac{1}{2}(x - \mu_i)^T \Sigma^{-1}(x - \mu_i)$ ;
2. 各个类别出现的先验概率相等;

## 8.4 最大似然估计

- 令梯度等于零，可得如下  $c$  个方程：

$$\sum_{k=1}^n P(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \nabla_{\boldsymbol{\theta}_i} \ln(P(\mathbf{x}_k | \omega_i, \hat{\boldsymbol{\theta}}_i)) = 0, \quad i=1, 2, \dots, c$$

- 求解上述方程可得待估计的  $\hat{\boldsymbol{\theta}}$ 。

- 进一步：**当未知量中包含先验概率  $P(\omega_i)$ （即混合比例）时，应限制如下两个条件：

$$P(\omega_i) \geq 0, \quad i=1, 2, \dots, c, \quad \text{且} \quad \sum_{j=1}^c P(\omega_j) = 1.$$

## 8.4 最大似然估计

- 实际上，如果似然函数可微，且  $P(\omega_i) \neq 0$ ，那么  $P(\omega_i)$  和  $\hat{\boldsymbol{\theta}}_i$  必然同时满足以下条件：

$$\text{条件1: } \hat{P}(\omega_i) = \frac{1}{n} \sum_{k=1}^n \hat{P}(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}})$$

$$\text{条件2: } \sum_{k=1}^n \hat{P}(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \nabla_{\boldsymbol{\theta}_i} \ln(P(\mathbf{x}_k | \omega_i, \hat{\boldsymbol{\theta}}_i)) = 0, \quad i=1, 2, \dots, c$$

$$\text{其中, } \hat{P}(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}}) = \frac{p(\mathbf{x}_k | \omega_i, \hat{\boldsymbol{\theta}}) \hat{P}(\omega_i)}{\sum_{j=1}^c p(\mathbf{x}_k | \omega_j, \hat{\boldsymbol{\theta}}_j) \hat{P}(\omega_j)} = \frac{p(\mathbf{x}_k, \omega_i | \hat{\boldsymbol{\theta}})}{p(\mathbf{x}_k | \hat{\boldsymbol{\theta}})}$$

全概率公式

(a)  $c$  个方程及添加的约束条件

(b) 得到的两个条件

图 19: 要求解的方程以及求解的结果

### 关于条件1的证明

- 首先，考虑所有变量时对数似然函数可以写成：

$$f_n(\mathbf{0}, \boldsymbol{\alpha}) = \sum_{k=1}^n \ln(p(\mathbf{x}_k | \mathbf{0})) = \sum_{k=1}^n \ln \left( \sum_{j=1}^c p(\mathbf{x}_k | \omega_j, \mathbf{0}_j) P(\omega_j) \right)$$

$$= \sum_{k=1}^n \ln \left( \sum_{j=1}^c p(\mathbf{x}_k | \omega_j, \mathbf{0}_j) \alpha_j \right)$$

其中引入新记号： $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_c]^T = [P(\omega_1), P(\omega_2), \dots, P(\omega_c)]^T$

- 然后，拉格朗日函数可以写成：

$$L(\mathbf{0}, \boldsymbol{\alpha}) = f_n(\mathbf{0}, \boldsymbol{\alpha}) + \lambda \left( \sum_{j=1}^c \alpha_j - 1 \right) \quad \text{且} \quad \sum_{j=1}^c \alpha_j = 1$$

拉格朗日乘子

### 关于条件1的证明(续)

- 求目标函数关于变量的偏导数，并令其等于0：

$$\frac{\partial L(\mathbf{0}, \boldsymbol{\alpha})}{\partial \alpha_i} = \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0}_i)}{p(\mathbf{x}_k | \mathbf{0})} + \lambda = 0, \quad i=1, 2, \dots, c$$

- 在方程的两边乘以  $\alpha_i$ ，并将  $c$  个方程相加，可得

$$\sum_{i=1}^c \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0}) \alpha_i}{p(\mathbf{x}_k | \mathbf{0})} + \lambda \sum_{i=1}^c \alpha_i = 0$$

$$\Rightarrow \lambda = -\sum_{i=1}^c \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0}) \alpha_i}{p(\mathbf{x}_k | \mathbf{0})} = -\sum_{i=1}^c \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0}) \alpha_i}{p(\mathbf{x}_k | \mathbf{0})}$$

$$= -\sum_{i=1}^c \frac{p(\mathbf{x}_i | \mathbf{0})}{p(\mathbf{x}_i | \mathbf{0})} = -n$$

### 关于条件1的证明(续)

- 最后由如下公式

$$\frac{\partial L(\mathbf{0}, \boldsymbol{\alpha})}{\partial \alpha_i} = \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0})}{p(\mathbf{x}_k | \mathbf{0})} + \lambda = 0, \quad i=1, 2, \dots, c$$

- 可得

$$\sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0}) \alpha_i}{p(\mathbf{x}_k | \mathbf{0})} = n \alpha_i, \quad i=1, 2, \dots, c$$

$$\Rightarrow \alpha_i = \frac{1}{n} \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0})}{p(\mathbf{x}_k | \mathbf{0})} = \frac{1}{n} \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \mathbf{0}) p(\omega_i)}{p(\mathbf{x}_k | \mathbf{0})}$$

$$= \frac{1}{n} \sum_{k=1}^n P(\omega_i | \mathbf{x}_k, \mathbf{0})$$

因此，条件1得证。

图 20: 使用拉格朗日乘子法证明条件一的过程

3. 每个样本以概率 1 属于一个类;
  4. 协方差矩阵为一个很小的数  $\epsilon$  乘以单位阵。
- 在这四个假设之下, 来求解参数  $\mu_i = \frac{1}{n_i} \sum_{x_k \in w_i} x_k, i = 1, 2, \dots, c$ , 得到 K-均值聚类算法。

## 二. 分级聚类

分级聚类将树的每一个叶子节点当成一个样本, 归到同一个根节点的视为一类。主要分类方式分为两种, 一种是先考虑样本之间的距离, 合并了一些之后, 需要考虑类与类之间的距离 (自底向上); 第二种是从所有样本中逐渐细分样本 (自顶向下)。K-均值聚类需要事先知道聚类的个数, 分级聚类貌似可以避免, 但是从哪层 level 划分也是需要探究的, 聚类个数仍然是个问题。

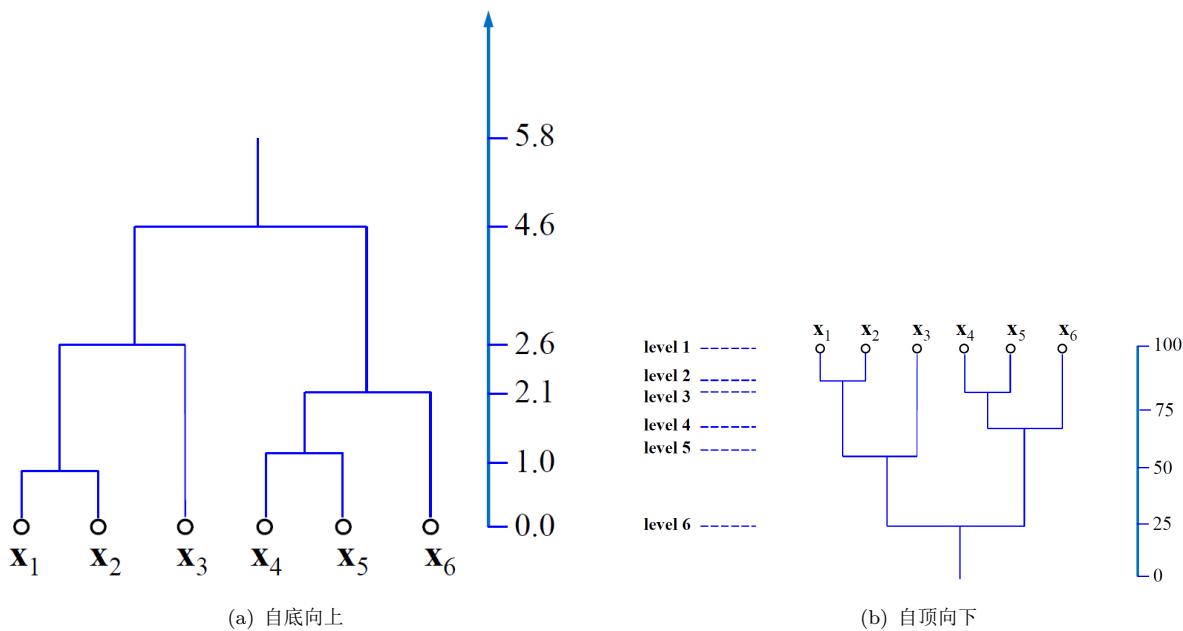


图 21: 两种分级聚类的方式

## 三. 谱聚类

联系均值聚类方法, 均值聚类假设样本分布服从高斯分布, 默认均值是聚类中心。如果不同类别的样本分布在多个同心圆上, 均值分类将失效。因为同心圆的均值都在同一点, 这样所有类别都被分成同一类。

定理: 设  $G$  为一个具有非负连接权重的无向图, 由  $G$  导出的拉普拉斯矩阵  $L$  的零特征值重数等于图  $G$  的连通子图的个数  $k$ 。这个定理将聚类与图论联系起来。证明过程略

使用图论来求做谱聚类, 首先要将样本点转换成一张图, 构造图的方式有全连接或局部连接。局部连接中需要用到  $k$  近邻 ( $\epsilon$  邻域) 来构造图, 即对每个数据点  $x_i$ , 首先在所有样本中找出不包含  $x_i$  的  $k$  个最邻近的样本点, 然后  $x_i$  与每个邻近样本点均有一条边相连, 从而完成图构造。图构造好之后要考虑图的 Laplace 矩阵 (后续的操作都在围绕着图的 Laplace 矩阵操作)。

图的 Laplace 矩阵目前有三种主流的构造方式:

1. 直接使用图的 Laplace 矩阵  $L = D - W$ , 其中  $D$  是图的度矩阵,  $W$  是图的邻接矩阵;
2.  $L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$ ,  $D$  为度矩阵,  $W$  为邻接矩阵;
3.  $L_{rw} = D^{-1} L = I - D^{-1} W$ .

Laplace 矩阵构造好之后, 需要求解 Laplace 矩阵的最小的  $k$  个特征值对应的特征向量, 将这  $k$  个特征向量依次排列组合在一起, 得到  $n * k$  的矩阵, 则每一行的  $1 * k$  向量将作为该样本点的新特征值, 然

后再进行均值聚类。所以谱聚类的过程实际上是一个特征选择 + 均值聚类的过程。分别对应下面三个算法过程。

```
Un-normalized (classical) Spectral Clustering—Algorithm 1
1 input: similarity matrix  $\mathbf{W}$ , number  $k$  of clusters
2 compute the un-normalized Laplacian matrix  $\mathbf{L} = \mathbf{D} - \mathbf{W}$ 
3 compute the first  $k$  eigenvectors  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$  of the  $\mathbf{L}$ 
4 let  $\mathbf{U} \in \mathbb{R}^{n \times k}$  be the matrix containing the vectors  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$ , namely,  $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k] \in \mathbb{R}^{n \times k}$ 
5 for  $i = 1, 2, \dots, n$ , let  $\mathbf{y}_i \in \mathbb{R}^k$  be the vector corresponding to the  $i$ -th row of  $\mathbf{U}$ .
6 cluster the points  $\{\mathbf{y}_i\}_{i=1,\dots,n}$  in  $\mathbb{R}^k$  with k-means algorithm into clusters  $A_1, A_2, \dots, A_k$ 
7 output  $A_1, A_2, \dots, A_k$ .
```

(a) algorithm1

```
Normalized Spectral Clustering—Algorithm 2 (Shi 算法)
1 input: similarity matrix  $\mathbf{W}$ , number  $k$  of clusters
2 compute  $\mathbf{L}_{sym} = \mathbf{D}^{-1/2} \mathbf{L} \mathbf{D}^{-1/2}$ 
3 compute the first  $k$  eigenvectors  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$  of  $\mathbf{L}_{sym}$ 
4 Let  $\mathbf{U} \in \mathbb{R}^{n \times k}$  be the matrix containing the vectors  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$ , namely,  $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k] \in \mathbb{R}^{n \times k}$ 
5 for  $i = 1, 2, \dots, n$ , let  $\mathbf{y}_i \in \mathbb{R}^k$  be the vector corresponding to the  $i$ -th row of  $\mathbf{U}$ .
6 cluster the points  $\{\mathbf{y}_i\}_{i=1,\dots,n}$  in  $\mathbb{R}^k$  with k-means algorithm into clusters  $A_1, A_2, \dots, A_k$ 
7 output  $A_1, A_2, \dots, A_k$ 
```

(b) algorithm2

```
Normalized Spectral Clustering—Algorithm 3 (Ng 算法)
1 input: similarity matrix  $\mathbf{W}$ , number  $k$  of clusters
2 compute  $\mathbf{L}_{sym} = \mathbf{D}^{-1/2} \mathbf{L} \mathbf{D}^{-1/2}$ 
3 compute the first  $k$  eigenvectors  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$  of  $\mathbf{L}_{sym}$ 
4 Let  $\mathbf{U} \in \mathbb{R}^{n \times k}$  be the matrix containing the vectors  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k$ , namely,  $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k] \in \mathbb{R}^{n \times k}$ 
5 form the matrix  $\mathbf{T} \in \mathbb{R}^{n \times k}$  from  $\mathbf{U}$  by normalizing the rows to norm 1, namely, set  $t_{ij} = u_{ij} / \sqrt{\sum_{m=1}^n u_{im}^2}$ 
6 for  $i = 1, 2, \dots, n$ , let  $\mathbf{y}_i \in \mathbb{R}^k$  be the vector corresponding to the  $i$ -th row of  $\mathbf{T}$ .
7 cluster the points  $\{\mathbf{y}_i\}_{i=1,\dots,n}$  in  $\mathbb{R}^k$  with k-means algorithm into clusters  $A_1, A_2, \dots, A_k$ 
8 output  $A_1, A_2, \dots, A_k$ 
```

On spectral clustering: analysis and an algorithm, NIPS, 2002.

(c) algorithm3

图 22: 三种构造 Laplace 矩阵方式分别对应的算法过程

## Question 集成学习

**Boosting** Boosting 是集成学习中的常用技术手段，通过改变训练样本的权重，学习多个分类器，并将这些分类器进行线性组合，以达到非线性分类器的效果。Adaboost 是 Boosting 方法中最具代表性的方法，Adaboost 算法在深度学习流行以前是生物特征识别（特别是人脸识别）的主要方法。  
Adaboost 核心思想：在分类问题中，它通过改变训练样本的权重，学习多个分类器，并将这些分类器进行组合，提高分类性能。将弱分类器的线性组合为强分类器，给每个样本多设置一个权重，计算所得的误差函数也是各个样本的误差乘以权重得到，而不是均值误差。当某次分类，一些样本被分错时，则分错的样本权重需要提高。在 Adaboost 当中，有两个权重，一个是样本的权重，一个是分类器的权重。

1. 在每轮训练中，如何改变训练数据的权值或分布？

提高那些被前一轮弱分类器分错的样本的权重，降低已经被正确分类的样本的权重。错分的样本将在下一轮弱分类器中得到更多关注。于是分类问题被一系列弱分类器“分而治之”。

2. 如何将一系列的弱分类器组合成一个强分类器？

关于弱分类器的组合，Adaboost 的做法是：采用加权（多数）表决的方法。具体地，加大分类错误率较小的弱分类器的权重，使其在表决中起更大的作用。

具体算法流程如下：

下面给出一个例子：

## Question SVM

### 支持向量机

支持向量机是 1995 年-2005 年十年期间非常流行的方法，经典的 BP 算法于 1980 年提出，然后受限于计算能力，当时只能训练 3-5 层的网络，这时 SVM 和 Boosting 出现，因为有理论支撑。

1. 首先介绍 VC 维的概念：对于一个模型，它可以以 100% 的精度二分类  $n$  个样本，最大的  $n$  称为它的 VC 维。VC 维实际上刻画了一个模型的复杂程度，一个分类  $n$  维样本的线性分类器参数有  $n+1$  个，其 VC 维也是  $n+1$ 。但是参数的个数和 VC 维并不等同，如  $f(x) = Asin(wx)$  参数只有两个，但是它可以分开任意个样本，所以 VC 维是无穷大；

## 8.8 Adaboost

- Adaboost算法步骤

- 输入训练数据集:  
 $T = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
- 输入弱学习算法
- (1) 初始化训练数据的权重分布  
 $D_1 = \{w_{11}, w_{12}, \dots, w_{1n}\}$ ,  $w_{1j} = 1/n$ ,  $j = 1, \dots, n$
- (2) 对  $m = 1, 2, \dots, M$
- (2a) 使用具有权重分布  $D_m$  的训练数据, 学习基本分类器  
 $G_m(x): X \rightarrow \{-1, +1\}$

(a) Step1

### Adaboost算法步骤(续)

- (2b) 计算  $G_m(x)$  在训练数据集上的分类错误率(加权):  
 $c_m = P(G_m(x_i) \neq y_i) = \sum_{i=1}^n w_{im} I(G_m(x_i) \neq y_i)$
- (2c) 计算  $G_m(x)$  的贡献系数:  
 $\alpha_m = \frac{1}{2} \ln \frac{1-c_m}{c_m}$

$\alpha_m$  表示  $G_m(x)$  在最终分类器中的重要性。当  $c_m \leq 0.5$  时,  $\alpha_m \geq 0$ 。同时,  $\alpha_m$  将随着  $c_m$  的减小而增大。

所以, 分类误差率越小的基本分类器在最终分类器中的作用越大。

(b) Step2

### Adaboost算法步骤(续)

- (2d) 更新训练数据集的权重分布:  
 $D_{m+1} = \{w_{m+1,1}, w_{m+1,2}, \dots, w_{m+1,n}\}$
- 具体计算如下:
- $$w_{m+1,j} = \frac{w_{mj}}{Z_m} \times \begin{cases} \exp(-\alpha_m), & \text{if } G_m(x_j) = y_j \\ \exp(\alpha_m), & \text{if } G_m(x_j) \neq y_j \end{cases}$$
- 若正确分类, 减少权重; 否则, 增加权重
- $$= \frac{w_{mj}}{Z_m} \times \exp(-\alpha_m y_j G_m(x_j))$$
- 其中,  $Z_m$  是规范化因子, 它使  $D_{m+1}$  成为一个概率分布:
- $$Z_m = \sum_{j=1}^n w_{mj} \exp(-\alpha_m y_j G_m(x_j))$$

(c) Step3

### Adaboost算法步骤(续)

- (3) 构建基本分类器的线性组合:

$$f(x) = \sum_{m=1}^M \alpha_m G_m(x)$$

对于两类分类问题, 得到最终的分类器:

$$G(x) = \text{sign}(f(x)) = \text{sign}\left(\sum_{m=1}^M \alpha_m G_m(x)\right)$$

(d) Step4

图 23: Adaboost 算法流程

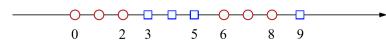
## 8.8 Adaboost

$$\text{sign}(x-v)$$

- 例子:** 给定如表所示训练数据。假设弱分类器由“如果  $x < v$ , 则  $x$  属于第一类; 如果  $x > v$ , 则  $x$  属于第二类”产生, 其阈值使该分类器在训练数据集上分类误差率最低。试用Adaboost算法学习一个强分类器。

序号	1	2	3	4	5	6	7	8	9	10
x	0	1	2	3	4	5	6	7	8	9
y	1	1	1	-1	-1	-1	1	1	1	-1

上表中: 共10个一维空间的样本, x表示样本, y表示标签。



李航著:《统计学习方法》(第8章), 清华大学出版社, 2012

图 24: 例题

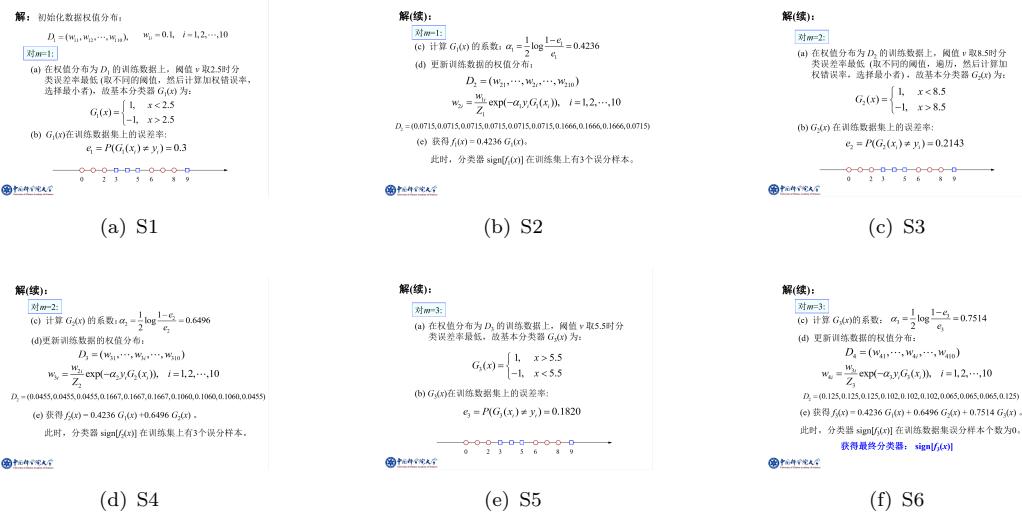


图 25: 解答过程

2. 在二分类线性分类器中, 对于一条直线, 会出现 Large Margin, 即样本点离这条直线的最小距离。Margin 越大, 则该分类器的 VC 维就越小, 泛化能力就越好;

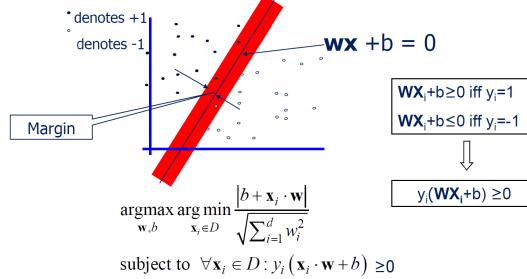
3. SVM 分为 Hard-Margin SVM(两类完全可分) 和 Soft-Margin SVM(两类不完全可分)。

(1) Hard-Margin SVM:

样本完全可分, 只需最大化样本离分界面的最小距离即可, 目标函数及限制条件可以写为:

可以化简为:

### Estimate Margin (Method 1)



- Min-max problem  $\rightarrow$  game problem

26

图 26: Hard-Margin SVM 目标函数及限制条件

都是二次规划问题 (目标函数二次, 限制条件一次), 求解方法已经非常成熟。

(2) Soft-Margin SVM:

实际情况中样本并不是完全可分的, 这时引入松弛变量, 即允许分错, 但是又不允许太多。目标函数即限制条件可以写为:

进一步分析, 引入的松弛变量实际上就是一个  $hinge(x) = \max\{1 - x, 0\}$ 。 $\epsilon \geq 1 - y(wx + b) = 1 - yf(x)$ :

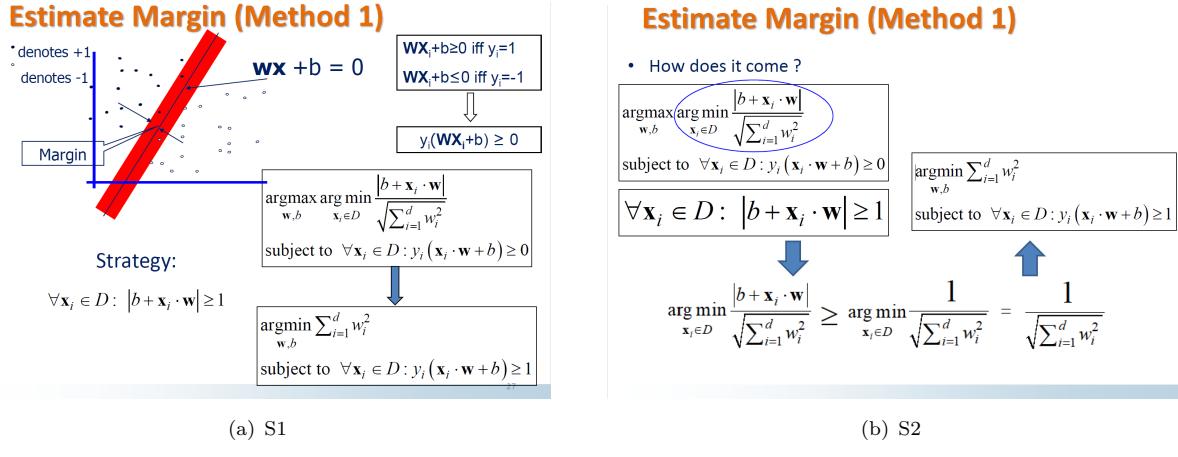


图 27: 化简

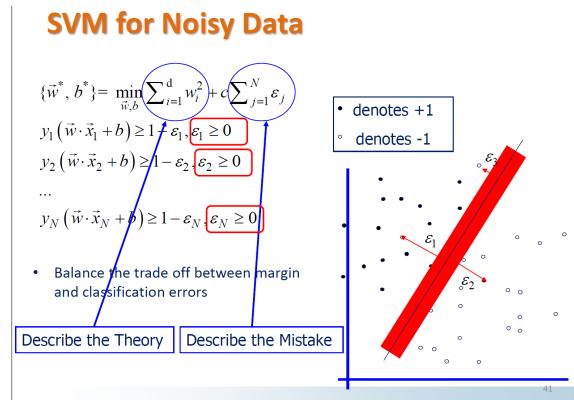


图 28: Soft-Margin SVM 目标函数及限制条件

- (1) 若  $yf(x) \geq 1$ , 即  $1 - yf(x) \leq 0$ , 则分类正确, 此时  $\epsilon = 0$ ;  
(2) 若  $yf(x) \leq 1$ , 即  $1 - yf(x) \geq 0$ , 则分类错误或者样本点在 Margin 里面, 此时  $\epsilon > 0$ , 且  $\epsilon$  越大, 越表示分错, 代价应当越大;  
综上应有:  $\epsilon = \text{hinge}(yf(x))$ , 函数图像如下:

将目标函数改良为:

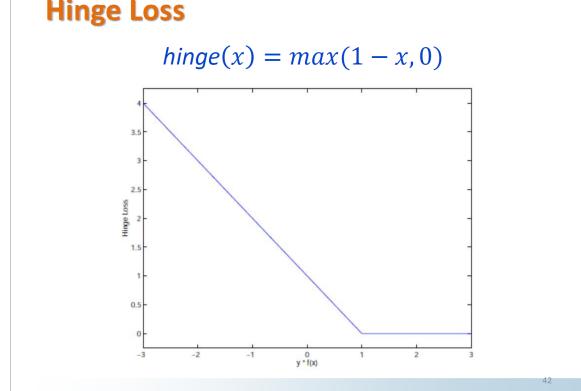


图 29: hinge(x)

上述改良过程主要有两个要点, 一是通过引入松弛变量, 给损失函数添加了  $\text{relu}(\text{hinge}(x))$  函数; 二是

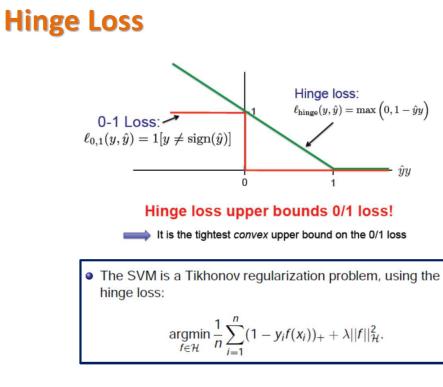


图 30: 改良的目标函数

使用正则项。

4. 不管是 Hard-Margin SVM 还是 Soft-Margin SVM 他们的对偶问题 (后续详细介绍对偶问题) 都是一样的:

每个样本点都有一个参数  $\alpha$ :

- (1)  $\alpha = 0$ , 对应样本点在边界之外, 分类正确;
- (2)  $0 < \alpha < C$ , 对应样本点在边界上, 且为支撑向量;
- (3)  $\alpha = C$ , 对应样本点在边界内, 且为支撑向量;

从上述过程可以看出, 支持向量机实际上只能解决线性问题, 但是通过核函数的引进, 非线性问题通过升维可以变为线性问题。支持向量机通过将传统的求解模型 (有参数  $W, b$ ) 转换到其对偶形式 (只有参数  $\alpha$ , 每一个样本都有一个参数  $\alpha_i$ ), 这样避免了维数灾难的问题, 因为在对偶形式中  $\alpha$  只与样本个数有关, 并且解是全局的。最后给出一个实例:

## Dual Problem

- We can transform the problem to its dual

$$\begin{aligned} \max. W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to } \alpha_i &\geq 0, \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

Dot product of X

$\alpha$ 's → New variables  
(Lagrangian multipliers)

- This is a convex quadratic programming (QP) problem
  - Global maximum of  $\alpha_i$  can always be found
  - well established tools for solving this optimization problem

- Note:  $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$

49

## Dual Problem for Soft-Margin SVM

- The dual of the problem is

$$\begin{aligned} \max. W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to } C &\geq \alpha_i \geq 0, \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

- $\mathbf{w}$  is also recovered as  $\mathbf{w} = \sum_{j=1}^s \alpha_j y_j \mathbf{x}_j$
- The only difference with the linear separable case is that there is an upper bound  $C$  on  $\alpha_i$
- Once again, a QP solver can be used to find  $\alpha_i$  efficiently!!!

53

(a) Hard-Margin SVM

(b) Soft-Margin SVM

图 31: 对偶问题

支持向量机中的核方法实际上是在将数据升维，一系列样本在低维空间中不能用线性分类器分开，需

### Example

- Suppose we have 5 1D data points  
 $x_1=1, x_2=2, x_3=4, x_4=5, x_5=6$ , with  $y_1=1, 2, 6$  as class 1 and 4, 5 as class 2
- We use the polynomial kernel of degree 2  
 $\langle \mathbf{x}, \mathbf{y} \rangle = [\mathbf{xy}]^T$   
 $\mathbf{c}$  is set to 100
- We first find  $\alpha_i$  ( $i=1, \dots, 5$ ) by

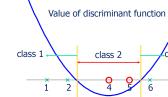
$$\begin{aligned} \max. \quad & \sum_{i=1}^5 \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^5 \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \mathbf{x}_j + 1)^2 \\ \text{subject to } 100 &\geq \alpha_i \geq 0, \sum_{i=1}^5 \alpha_i y_i = 0 \end{aligned}$$

### Example

- By using a QP solver, we get  
 $\alpha_1=0, \alpha_2=2.5, \alpha_3=0, \alpha_4=7.333, \alpha_5=4.833$   
→ Verify that the constraints are indeed satisfied
- The support vectors are  $(x_2, y_2), (x_4, y_4), (x_5, y_5)$
- The discriminant function is

$$\begin{aligned} f(x) &= 3.5(2x+1)^2 + 7.333(5x+1)^2 + 4.833(3x+1)^2 + 4 \\ &= 0.6667x^2 - 5.333x + 9 \\ \mathbf{b} &\text{ is recovered by solving } \alpha_2 \text{ or by } \alpha_4 \text{ or by } \alpha_5 \text{ as } x_2, x_4, x_5 \text{ lie on } \mathbf{w}^\top \mathbf{x} + \mathbf{b} = 0 \text{ all give } \mathbf{b}=9 \\ \Rightarrow f(x) &= 0.6667x^2 - 5.333x + 9 \end{aligned}$$

### Example



(a) S1

(b) S2

(c) S3

图 32: 实例

要通过核方法做升维 (对偶问题中只出现了样本的核函数形式)，升维到高维空间中就可以用线性分类器来做分类

关于如何加速训练 SVM，有两点改进：

- 只有支持向量会影响决策边界：训练时采用组块策略，不是一次选择所有样本。而是每次添加一些新样本，确定支撑向量，再添加新样本，再确定新样本，一直到支撑向量不变为止，即收敛；
- 当达到最优结果时，此时 KKT 条件是满足的：每次添加完新数据之后，用序列最小化优化方法 (SMO) 来计算  $\alpha$ ，可以每次选择计算两个  $\alpha$ 。很多文献中有关于如何选择两个乘子的方法，如果每次随机选择  $\alpha$ ，会导致收敛速度缓慢。

上述介绍了支持向量机分类两类问题，对于多分类问题，有 *oneV Sall*, *oneV Sone*, *DAGSVM*, *halfV Shalf*, *LatticeSVM* 等，或者基于二分类的理论直接建立多分类的理论。

关于深度学习与支持向量机的区别：

- 支持向量机中的特征升维，是直接学习核函数  $K(x, y) = \phi(x)\phi(y)$ ，而深度学习是直接学习函数  $\phi(x)$  和  $\phi(y)$ ；(2)SVM 的目标函数是凸函数，在学习的时候可以直接到达全局最优解；深度学习不是凸函数，需要一些迭代策略 (如带冲量项的梯度下降)；
- 深度学习的解一般不是最优解，但时常有效的原因：SVM 实际上是对问题的一种近似，但是可以精确求解；深度学习虽然不是问题的最优解，但是对问题的直接求解，会跳过支持向量机由于模型近似带来的固有误差。

# Question

对偶问题。对偶问题的实质是要给目标函数估计一个界：最小化一个目标函数，对偶问题是估计该目标函数的最大下界；最大化一个目标函数，对偶问题是估计该目标函数的最小上界。

例如，原问题是  $\min g(x)$ ，则对偶问题可以写成： $\max f(y)$ ，其中：对于任意一个  $x$ ， $g(x)$  都是  $f(y)$  的上界；对于任何一个  $y$ ， $f(y)$  都是  $g(x)$  的下界。

## 一、线性规划的对偶问题：

对于一个无约束的目标函数，采用求驻点（即求导）方法即可求得最优解，如果添加约束条件，可以使用 lagrange 方法将问题转换为一个无约束的问题，再进行求导。具体转换方式如下：

转换后的无约束目标函数是不可导的，无法直接求解。结合对偶问题的描述，求解一个目标函数的最

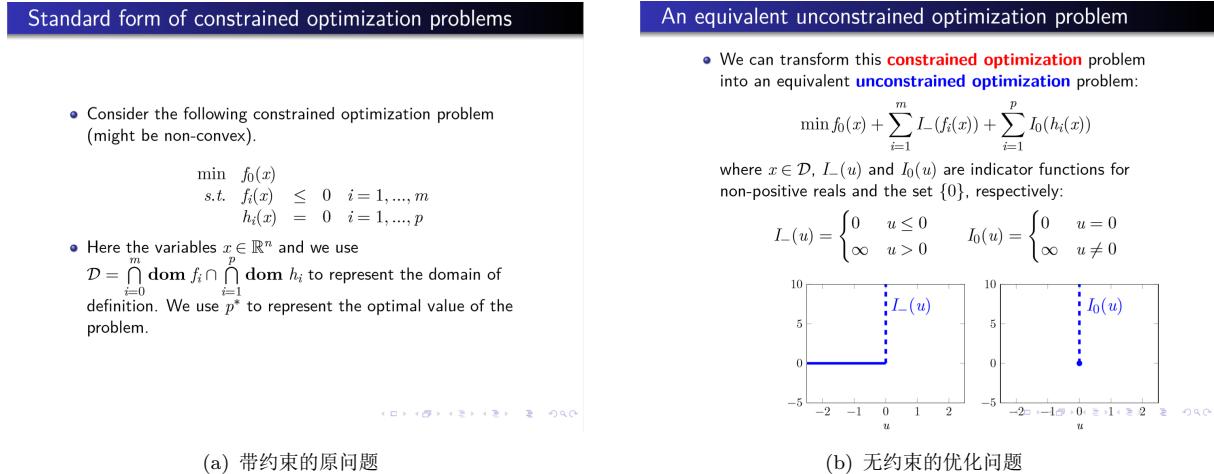


图 33: 约束问题转换为无约束问题

小值，实际上可以用该目标函数下界的最大值来估计，利用不同的下界估计函数，对应着不同的方法。如下图，用对数障碍函数做下界，对应着内点法；用平方项函数做下界，对应着罚函数法；用线性函数做下界，对应着 lagrange 乘子法。

关于线性函数估计下界，即 lagrange 乘子法，lagrange 乘子可以当作是乘子系数，也是对偶问题的变

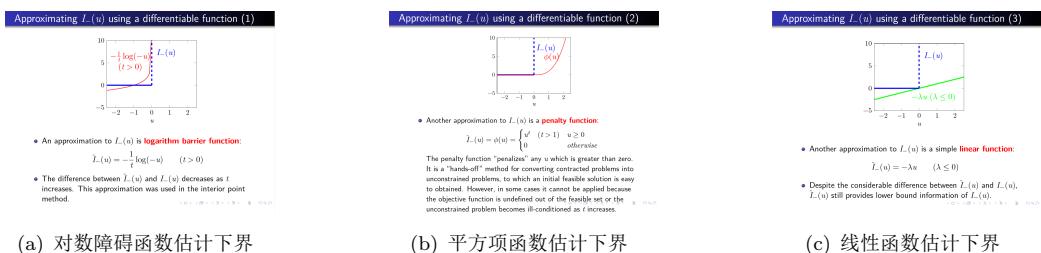


图 34: 不同的函数来估计下界

量。对于上述的问题，有  $f(x) \leq L(x, y) \leq g(y)$ 。

下面详细介绍 lagrange 乘子法：(以一个实例来推导对偶问题)

lagrange 乘子法利用线性函数来估计下界，由原函数得到 lagrange 函数，再求解该 lagrange 函数的上界，即可得到原函数的一个下界估计。再说明这个下界与原问题最优解之间的差距即可。

原问题：

$$\begin{aligned}
& \min x_1 + 2x_2 \\
\text{s.t.} & 3x_1 + 4x_2 \geq 5 \\
& 6x_1 + 7x_2 \geq 8 \\
& x_1, x_2 \geq 0
\end{aligned}$$

### 1. 对偶问题推导

(1) 写出 lagrange 函数

lagrange 函数为:  $L(x_1, x_2, \lambda_1, \lambda_2) = x_1 + 2x_2 - \lambda_1(3x_1 + 4x_2 - 5) - \lambda_2(6x_1 + 7x_2 - 8) = 5\lambda_1 + 8\lambda_2 + (1 - 3\lambda_1 - 6\lambda_2)x_1 + (2 - 4\lambda_1 - 7\lambda_2)x_2$ .

当  $\lambda_1 \geq 0, \lambda_2 \geq 0$ , 并且  $x_1, x_2$  是可行的, 则  $L(x_1, x_2, \lambda_1, \lambda_2)$  是原函数的一个下界。

(2) 求解 lagrange 函数的下界

lagrange 对偶函数

$$g(\lambda_1, \lambda_2) = \inf_{x_1, x_2 \in D} L(x_1, x_2, \lambda_1, \lambda_2) = \begin{cases} 5\lambda_1 + 8\lambda_2 & 3\lambda_1 + 6\lambda_2 - 1 \leq 0, 4\lambda_1 + 7\lambda_2 - 2 \leq 0 \\ -\infty & \text{otherwise} \end{cases} \quad (1)$$

(3) 写出对偶问题

$$\begin{aligned}
& \max 5\lambda_1 + 8\lambda_2 \\
\text{s.t.} & 3\lambda_1 + 6\lambda_2 \leq 1 \\
& 4\lambda_1 + 7\lambda_2 \leq 2 \\
& \lambda_1, \lambda_2 \geq 0
\end{aligned}$$

下面解释线性规划对偶问题中的记忆规则:

Q1: 为何原约束  $3x_1 + 4x_2 \geq 5$  时, 对偶问题中  $y_1 \geq 0$ ?

A1: 在计算 lagrange 函数时, 需要找到原目标函数的一个下界, 只有当  $\lambda_1$  和约束条件同号时才有  $f(x_1, x_2) \geq L(x_1, x_2, \lambda_1, \lambda_2)$ ;

Q2: 为何原约束  $x_1 \geq 0$  时, 对偶问题的约束  $3\lambda_1 + 6\lambda_2 \leq 1$ ?

A2: lagrange 函数  $L(x_1, x_2, \lambda_1, \lambda_2)$  中  $x_1$  的系数一定得是正数, 否则  $g(\lambda_1, \lambda_2) = -\infty$ ;  
等式条件的约束也可类似推导, 不过约束条件会有一点变化。

2.lagrange 函数的最优解 (KKT 条件) 与对偶问题的最优值 (slater 条件), 只介绍, 不证明。

弱对偶问题: 原问题的最小值与对偶问题的最大值 (所求的一个下界) 之间存在 duality gap

强对偶问题: 原问题的最小值  $p^*$  与对偶问题的最大值 (所求的一个下界)  $d^*$  之间不存在 duality gap,

即  $p^* = d^*$

slater 条件:

(1) 目标函数  $f(x)$  是凸函数;

(2) 存在一个  $x, s.t. f_i(x) < 0, i = 1, \dots, m$

当满足 slater 条件时, 该规划问题一定是强对偶问题 (线性规划或者二次规划均可), 由于线性规划一定满足 slater 条件, 所以线性规划是强对偶问题。

KKT 条件:

KKT 条件是关于 lagrange 函数最优解的条件。包括两条

(1) lagrange 条件:  $\nabla L(x^*, y^*, \lambda) = 0$ ;

(2) 互补松弛性:  $\lambda f_1(x^*, y^*) = 0$ .

## Lagrange Multipliers

$$\begin{aligned}
 & \text{Minimize } f(x) && L(x, \alpha) = f(x) - \alpha_1 a(x) - \alpha_2 b(x) - \alpha_3 c(x) \\
 & \text{subject to} && \begin{cases} \alpha_1 \geq 0 \\ \alpha_2 \leq 0 \\ \alpha_3 \text{ is unconstrained} \end{cases} \\
 & && \text{We can recover the primal problem by maximizing the Lagrangian with respect to the Lagrange multipliers} \\
 & && \max_{\alpha} L(x, \alpha) = \begin{cases} f(x), & \text{if } \begin{cases} a(x) \geq 0 \\ b(x) \leq 0 \\ c(x) = 0 \end{cases} \\ +\infty, & \text{otherwise} \end{cases} \\
 & && \text{So, the Primal problem can be changed into Dual problem} \\
 & && \min_x \max_{\alpha} L(x, \alpha) = \max_{\alpha} \min_x L(x, \alpha)
 \end{aligned}$$

## Karush-Kuhn-Tucker conditions

- For a local minimum

$$\begin{cases} \text{Stationarity} & \nabla f(x^*) - \alpha_1 \nabla a(x^*) - \alpha_2 \nabla b(x^*) - \alpha_3 \nabla c(x^*) = 0 \\ \text{Primal feasibility} & \begin{cases} a(x^*) \geq 0 \\ b(x^*) \leq 0 \\ c(x^*) = 0 \end{cases} \\ \text{Dual feasibility} & \begin{cases} \alpha_1 \geq 0 \\ \alpha_2 \leq 0 \\ \alpha_3 \text{ is unconstrained} \end{cases} \\ \text{Complementary slackness} & \begin{cases} \alpha_1 a(x^*) = 0 \\ \alpha_2 b(x^*) = 0 \\ \alpha_3 c(x^*) = 0 \end{cases} \end{cases}$$

46

(a) 二次规划的 lagrange 函数

47

(b) KKT 条件

图 35: 二次规划的 lagrange 函数及 KKT 条件

## 二、二次规划的对偶问题 (结合支持向量机):

Hard-Margin SVM 对偶问题:

简单推导如下:

(1) 目标函数

$$\begin{aligned}
 & \min \frac{1}{2} \|w\|_2^2 \\
 & \text{s.t. } y_i(w^T x_i + b) \geq 1, i = 1, \dots, n
 \end{aligned}$$

(2) 引入 lagrange 函数

$$L(w, b, \alpha) = \frac{1}{2} \|w\|_2^2 + \sum_{i=1}^n \alpha_i [1 - y_i(w^T x_i + b)]$$

(3) 可行域分析

若使得 lagrange 函数  $L(w, b, \alpha)$  是原目标函数的一个下界, 则当  $y_i(w^T x_i + b) \geq 1$  时, 必有  $\alpha_i \geq 0$ , 即新的目标函数变为:

$$\begin{aligned}
 & \max L(w, b, \alpha) = \frac{1}{2} \|w\|_2^2 + \sum_{i=1}^n \alpha_i [1 - y_i(w^T x_i + b)] \\
 & \text{s.t. } \alpha_i \geq 0
 \end{aligned}$$

(4) KKT 条件求解对偶问题

lagrange 函数对  $w, b$  求导, 并使得导数为 0, 得到:

$$\begin{aligned} w + \sum_{i=1}^n \alpha_i (-y_i) x_i &= 0, w = \sum_{i=1}^n \alpha_i y_i x_i \\ \sum_{i=1}^n \alpha_i y_i &= 0 \end{aligned}$$

(5) 对偶目标函数重述

将 (4) 中条件代入对偶问题中，得到：

$$\begin{aligned} \max W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j x_i^T x_j \\ \text{s.t. } \alpha_i &\geq 0, \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

(6) 决策面求解

(5) 中求解出  $\alpha$  之后，求出  $w$  与  $b$  即可得到模型：

$$f(x) = w^T x + b = \sum_{i=1}^n \alpha_i y_i x_i^T x + b \quad (2)$$

观察 KKT 条件中的可行域约束：

- (1)  $\alpha_i \leq 0$
- (2)  $y_i f(x_i) \leq 1$
- (3)  $\alpha_i (y_i f(x_i) - 1) = 0$

对于任意训练样本  $(x_i, y_i)$ ，若  $\alpha_i = 0$ ，则该样本不会在决策面公式中出现，不会对决策面有任何影响；若  $\alpha_i > 0$ ，则对该样本来说必有  $y_i f(x_i) = 1$ ，此时样本落在最大间隔边界上，是一个支持向量，这显示支持向量机一个重要性质：训练完成后，大部分的训练样本都不需要保留，最终模型仅和支持向量有关。

求出  $w$  之后，再确定偏移量  $b$ ，对任意的支持向量来说，都有  $y_i(w^T x_i + b) = 1$ ，通过该公式求  $b$ ，理论上可以选择任何一个支持向量来计算  $b$ ，或者使用更加鲁棒的计算方法，求所有支持向量的平均值。

(7) 核方法

从结果中可以看出样本  $x_i$  在目标函数中总是成对出现，即  $x_i^T x_j$ ，所以对于非线性分类问题，可以采用核方法进行升维，将问题转换为线性分类问题。简单推导：

令  $\phi(x)$  表示将  $x$  映射后的特征向量，于是在特征空间中划分超平面  $f(x) = w^T \phi(x) + b$ ，同样可以写出原问题的目标函数：

$$\begin{aligned} \min_{w,b} \frac{1}{2} \|w\|_2^2 \\ \text{s.t. } y_i(w^T \phi(x_i) + b) \geq 1, i = 1, \dots, n \end{aligned}$$

其对偶问题是：

$$\begin{aligned} \max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \phi(x_i)^T \phi(x_j) \\ \text{s.t. } \sum_{i=1}^n \alpha_i y_i = 0, \alpha_i \geq 0, i = 1, \dots, n \end{aligned}$$

引入核方法，重写对偶问题：

$$\begin{aligned} & \max_{\alpha} \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ & \text{s.t. } \sum_{i=1}^n \alpha_i y_i = 0, \alpha_i \geq 1, i = 1, \dots, n \end{aligned}$$

求解后即得：

$$f(x) = w^T x + b = \sum_{i=1}^n \alpha_i y_i k(x_i, x) + b \quad (3)$$

### Lagrange Transformation

$$\begin{aligned} & \text{Minimize } \frac{1}{2} \|w\|^2 \\ & \text{subject to } 1 - y_i(w^T x_i + b) \leq 0 \quad \text{for } i = 1, \dots, n \end{aligned}$$

- The Lagrangian is
 
$$\mathcal{L} = \frac{1}{2} w^T w + \sum_{i=1}^n \alpha_i (1 - y_i(w^T x_i + b))$$
Lagrangian multipliers

- Setting the gradient of  $\mathcal{L}$  w.r.t.  $w$  and  $b$  to zero, we have

$$\begin{aligned} w + \sum_{i=1}^n \alpha_i (-y_i) x_i &= 0 \quad \Rightarrow \quad w = \sum_{i=1}^n \alpha_i y_i x_i \\ \sum_{i=1}^n \alpha_i y_i &= 0 \quad \alpha_i \geq 0 \end{aligned}$$

(a) 1

### Dual Problem

- We can transform the problem to its dual

$$\begin{aligned} \max_{\alpha} W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j x_i^T x_j \\ \text{subject to } \alpha_i \geq 0, \sum_{i=1}^n \alpha_i y_i &= 0 \end{aligned}$$

Dot product of  $x$

$\alpha$ 's  $\rightarrow$  New variables  
(Lagrangian multipliers)

- This is a convex quadratic programming (QP) problem
  - Global maximum of  $\alpha$  can always be found
  - $\rightarrow$  well established tools for solving this optimization problem

- Note:  $w = \sum_{i=1}^n \alpha_i y_i x_i$

49

(b) 2

图 36: Hard-Margin SVM 对偶问题

Soft-Margin SVM 对偶问题：  
(类似 Hard-Margin 对偶问题推导过程)

## Question

决策树基本流程为：训练集建立决策树 (ID3, C4.5)，验证集用来剪枝，测试集测试精度。  
决策树有如下几个特点：

- (1) 决策树的分类过程是可解释的，深度学习不可解释；
- (2) 对离散变量 (离散特征，比如高收入、中收入和低收入特征) 有很好的解决机制，深度学习和支持向量机主要是解决连续的特征 (比如收入是 7000, 8000, 10000 等)；
- (3) 决策树容易过拟合，对噪声很敏感，泛化能力差。

先介绍基本概念：

设随机变量  $X$  的概率分布为  $P(X = x_i) = p_i, i = 1, 2, \dots, n$ ，则随机变量  $X$  的熵定义为  $H(X) = -\sum_{i=1}^n p_i \log p_i$ ，是随机变量  $X$  不确定性的度量。设有随机变量  $X, Y$ ，联合概率分布为  $P(X = x_i, Y = y_j) = p_{ij}, i = 1, 2, \dots, n; j = 1, 2, \dots, m$ ，条件熵  $H(Y|X)$  表示在已知随机变量  $X$  的条件下随机变量  $Y$  的

## Soft-Margin Case

- Lagrangian Problem

$$L = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i (y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^n \gamma_i \xi_i$$

$\alpha_i \geq 0 \quad \gamma_i \geq 0$

$$\frac{\partial L}{\partial w} = 0 \Rightarrow w = \sum_i \alpha_i y_i x_i$$

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_i \alpha_i y_i = 0$$

$$\frac{\partial L}{\partial \xi_i} = 0 \Rightarrow \alpha_i + \gamma_i = C \Rightarrow 0 \leq \alpha_i \leq C$$

## Dual Problem for Soft-Margin SVM

- The dual of the problem is

$$\max. \quad W(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j x_i^T x_j$$

subject to  $C \geq \alpha_i \geq 0 \quad \sum_{i=1}^n \alpha_i y_i = 0$

•  $w$  is also recovered as  $w = \sum_{j=1}^s \alpha_j y_j x_j$

• The only difference with the linear separable case is that there is an upper bound  $C$  on  $\alpha_i$

• Once again, a QP solver can be used to find  $\alpha_i$  efficiently!!!

52

(a) 1

53

(b) 2

图 37: Soft-Margin SVM 对偶问题

不确定性, 定义为给定  $X$  条件下,  $Y$  的条件概率分布的熵对  $X$  的数学期望  $H(Y|X) = \sum_{i=1}^n p_i H(Y|X = x_i), p_i = P(X = x_i), i = 1, 2, \dots, n$ 。

### 一、建立决策树

建立决策树的过程最重要的是确定特征的选择顺序, 引入信息论中的信息增益 (ID3) 和信息增益率 (C4.5) 来确定特征的选择顺序。

#### 1.ID3:

特征  $A$  对训练数据集  $D$  的信息增益  $g(D, A)$ , 定义为集合的经验熵  $H(D)$  与特征  $A$  给定条件下  $D$  的经验条件熵  $H(D|A)$  之差, 即  $g(D|A) = H(D) - H(D|A)$ , 信息增益准则的特征选择方法是: 对训练数据集 (或者子集)  $D$ , 计算其每个特征的信息增益, 并比较他们的大小, 选择信息增益最大的特征。经验熵  $H(D)$  表示对数据集  $D$  进行分类的不确定性, 而经验条件熵  $H(D|A)$  表示在特征  $A$  给定的条件下对数据集  $D$  进行分类的不确定性。那么它们的差, 即信息增益, 表示由于特征  $A$  而使得对数据集  $D$  的分类的不确定性减少的程度, 显然信息增益大的特征具有更强的分类能力。

缺点: 信息增益偏向于有大量值的特征 (如学号、姓名, 每个值都是一类), 当特征是连续变量时, ID3 每次都会选择这些连续变量, 不好处理, 需要引入可以处理连续变量的方法;

#### 2.C4.5:

在信息增益的基础上除以一项分裂系数 (训练数据集  $D$  关于特征  $A$  的值的熵  $H_A(D) = -\sum_{i=1}^n \frac{|D_i|}{|D|} \log_2 \frac{|D_i|}{|D|}$ ,  $n$  是特征  $A$  取值的个数), 得到信息增益率。特征值越连续, 分裂系数越大, 对于连续变量有惩罚的作用。

### 3.CART(Classification and Regression Trees, 分类与回归树):

CART 规定决策树必须是二叉树, 对回归树用平方误差最小化准则, 对分类树用基尼指数 (Gini index) 最小化准则, 所以被称为分类与回归树。

#### (1) 回归树的生成

假设  $X$  和  $Y$  分别为输入和输出变量, 并且  $Y$  是连续变量, 给定训练数据集:  $D = (x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ 。一棵回归树对应着输入空间的一个划分以及在划分的单元上的输出值。假设已经将输入空间划分为  $M$  个单元  $R_1, R_2, \dots, R_M$ , 并且在每个单元  $R_m$  上有一个固定的输出值  $c_m$ , 于是回归树模型可以表示成  $f(x) = \sum_{m=1}^M c_m I(x \in R_m)$ 。

当输入空间确定时, 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, 对该误差求一阶导容易得到每个单元上的最优输出值。单元  $R_m$  上的  $c_m$  的最优值  $\hat{c}_m$  是  $R_m$  上的所有输

入实例  $x_i$  对应的输出  $y_i$  的均值，即  $\hat{c}_m = \text{ave}(y_i | x_i \in R_m)$ 。

问题是怎样对输入空间进行划分，这里采用启发式的方法，选择第  $j$  个变量  $x^{(j)}$ （输入样本的第  $j$  个特征），和它的取值  $s$  作为切分变量和切分点，并定义两个区域  $R_1(j, s) = x|x^{(j)} \leq s, R_2(j, s)x|x^{(j)} \geq s$ 。然后寻找最优切分变量  $j$  和最优切分点  $s$ ，具体地求解：

$$\min_{j,s} [\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2]$$

对于固定输入变量  $j$  可以找到最优切分点  $s$ ， $\hat{c}_1 = \text{ave}(y_i | x_i \in R_1(j, s))$ ,  $\hat{c}_2 = \text{ave}(y_i | x_i \in R_2(j, s))$ 。遍历所有输入变量，找到最优的切分变量  $j$ ，构成一个对  $(j, s)$ 。依次将输入空间划分为两个区域，接着对每个区域重复上述划分过程，直到满足条件为止。这样的回归树通常称为最小二乘回归树。

## (2) 分类树的生成

分类树用基尼系数选择最优特征，同时决定该特征的最优二值切分点。

基尼指数：分类问题中，假设有  $K$  个类，样本点属于第  $k$  类的概率为  $p_k$ ，则概率分布的基尼指数定义为：

$$Gini(p) = \sum_{k=1}^K p_k(1 - p_k) = 1 - \sum_{k=1}^K p_k^2$$

对于给定的样本集合  $D$ ，其基尼指数为  $Gini(D) = 1 - \sum_{k=1}^K (\frac{|C_k|}{D})^2$ ，这里  $C_k$  是  $D$  中属于第  $k$  类的样本子集， $K$  是类的个数。

如果样本集合根据特征  $A$  是否取某一可能值  $a$  被分割成  $D_1$  和  $D_2$  两部分，即  $D_1 = (x, y) \in D | A(x) = a, D_2 = D - D_1$ ，则在特征  $A$  的条件下，集合  $D$  的基尼指数定义为： $Gini(D, A) = \frac{|D_1|}{D} Gini(D_1) + \frac{|D_2|}{D} Gini(D_2)$ 。基尼指数  $Gini(D)$  表示集合  $D$  的不确定性，基尼指数  $Gini(D, A)$  表示经过  $A = a$  分割后集合  $D$  的不确定性。基尼指数越大，则样本集合的不确定性也越大，所以每次选择基尼指数最小对应的特征和切分点作为节点。

## 二、剪枝

剪枝就是减少规则（用验证集中的数据来验证各个规则，看是否可靠）。剪枝是减少“弱枝”，弱枝值得是在验证集上误分类率高的数枝。

特点：剪枝会增加训练数据集上的错误分类率，但精简的树会提高测试集上的预测能力。缺点是在剪枝的时候不知道剪到什么程度合适，需要引入新的决策模型（随机森林）

## 三、随机森林

随机森林是集成学习的一种方法，基于 Bagging。决策树的一个缺点就是容易过拟合，随机森林是由一群弱的决策树构成的一片森林。

为了解决过拟合的问题，随机森林在建立分类器的过程中有两次随机性（样本随机和特征随机）：

- (1) 从样本集中用 bagging 采样选出  $n$  个样本（样本随机），建立 CART；
- (2) 在树的每个节点上，从所有的特征中随机选择  $k$  个特征（特征随机），选择出一个最佳的分割属性作为节点（RI：从特征的子集当中选择最优；RC：将特征的子集线性组合形成新的特征）；
- (3) 重复以上两步  $m$  次，构建  $m$  颗 CART（不剪枝，建立的 CART 树由于随机样本和随机特征，具有很弱的分类能力）；
- (4) 这  $m$  颗树形成 Random Forest；
- (5) 建立好随机森林后，输入一个样本，每个决策树都要输出一个决策分类，对分类结果做投票得到最终的分类结果。

注意：

- (1) 在样本随机时，采用的是 0.632 自助法，每次只选择 63% 的样本数作为样本子集；
- (2) bagging 是采用重采样样本的方法，boosting 是采用重加权样本或分类器的方法。

## 四、提升树

提升树的基本单位是 CART，当解决二分类问题时，提升树只需将 AdaBoost 算法中的基本分类器限制为二类分类树即可，可以说这时的提升树算法是 AdaBoost 算法的特殊情况，这里主要叙述提升树中

的回归问题。

在 CART 中已经介绍回归树的一些概念，已知一个训练数据集  $T = (x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ ,  $x_i \in X$ ,  $X$  为输入空间。将输入空间  $X$  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  $\Theta = (R_1, c_1), (R_2, c_2), \dots, (R_J, c_J)$  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的个数。

提升树使用以下前向分步算法 (一共  $M$  步):

$$f_0(x) = 0; f_m(x) = f_{m-1} + T(x; \Theta_m), m = 1, 2, \dots, M; f_M(x) = \sum_{m=1}^M T(x; \Theta_m)$$

在前向分步算法的第  $m$  步, 给定当前模型  $f_{m-1}(x)$ , 需要求解  $\hat{\Theta}_m = \operatorname{argmin}_{\Theta_m} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m))$ , 得到  $\Theta_m$ , 即第  $m$  棵树的参数。

当采用平方误差损失函数时,  $L(y, f(x)) = (y - f(x))^2$ 。

## 五、GBDT(Gradient Boosting Decision Tree)

提升树模型使用加法模型和前向分布算法实现学习的优化过程。当损失函数是平方损失和指数损失函数时, 优化过程较简单, 但对一般的损失函数, 每一步的优化并不是那么容易。利用最速下降法的近似方法, 关键是利用损失函数的负梯度在当前模型的值  $-[\frac{\partial L(y, f(x_i))}{\partial f(x_i)}]_{f(x)=f_{m-1}(x)}$  作为回归问题提升树算法中的残差的近似值 (当损失函数是均方损失时, 梯度和残差相同; 对于一般的损失函数, 梯度是残差的近似值), 来拟合一个回归树。

## 六、GBDT 和随机森林的区别

相同点:

- (1) 都是由多棵树组成;
- (2) 最终的结果都是由多棵树一起决定;

不同点:

1. 组成随机森林的数可以是分类树, 也可以是回归树; 而 GBDT 只由回归树组成 (GBDT 用于分类时, 相当于把 AdaBoost 中的分类器换成二类分类树即可, 可以说是 AdaBoost 算法的特殊情况, 所以 GBDT 一般由回归树组成)。
2. 组成随机森林的数可以并行生成; 而 GBDT 只能是串行生成。
3. 对于最终结果, 随机森林采用多数投票的方式; 而 GBDT 则是将所有结果累加起来, 或者加权起来。
4. 随机森林对异常值不敏感; 而 GBDT 对异常值非常敏感。
5. 随机森林对训练集一视同仁; 而 GBDT 是基于权值的弱分类器的集成。

# Question

## 特征提取、特征降维以及特征选择

“模式识别”在早期的含义实际上特指特征工程, 即如何提取、降维以及选择数据中的特征, 然后再使用经典的机器学习方法进行分类或者回归。现在这两个过程被深度学习方法统一起来。

深度学习实际上是一种自动提取特征的方法, 卷积神经网络中的卷积-池化层会把物体 (比如要识别的猫, 狗) 弄到图像的中间位置, 相当于图像的配准过程, 所以深度学习也叫表示学习。深度学习降低了计算机视觉各个子方向的门槛, 转方向的时候 (比如从文字识别到虹膜识别) 不需要该领域特别多的基础知识, 只要会深度学习框架就能训练。

深度学习提取特征虽然强大, 但也有缺点:

- (1) 小样本学习: 样本个数很少, 不适合使用深度学习训练, 此时可以使用手工设计特征;
- (2) 连续性学习: 在给定的样本上和类别上可以训练的精度很高, 当增加一些类别和样本时, 深度学习需要重新训练来学习特征, 手工设计特征不需要。

## 一、特征提取

经典的特征提取方法有四种：

1.SIFT( Scale Invariant Feature Transform), 尺度不变性 (实际上具有平移、尺度以及旋转不变性)

(1)DOG 首先使用不同带宽的高斯函数进行滤波，相邻带宽的滤波结果作差，得到一系列的 DOG；

(2) 取极值点 (极大或者极小)；

(3) 计算极值点的梯度方向 (点 + 方向)；

(4) 计算极值点周围的点的方向，每个像素点变成一个 128 维的向量，但是极值点个数不一样。

2.Haar 特征

3.HOG 特征

4.LBP 特征 (Local Binary Pattern)

## 二、特征降维 (维数消减)

特征降维与特征选择不一样 (虽然都可以起到维数消减的作用)，特征降维是将  $n$  维的样本降到  $d$  ( $d < n$ ) 维空间，再降维过程中  $n$  维特征都用到了，而特征选择是直接选择其中的  $d$  维特征，其他维度的特征直接舍弃。

维数消减方法分为线性降维和非线性降维。

### 1. 线性降维

PCA(Principal Components Analysis, 主成分分析)

主成分分析是一种无监督降维方法，实际上是在寻找方差最大的维度作为主成分 (方差大的方向，说明在该方向数据较分散)，核心思想是将  $D$  维特征映射到  $K$  维 ( $K < D$ )，这  $K$  维形成主元且相互正交，是重构出来最能代表原始数据的正交特征，图示如下：

推导过程以及算法流程如下：

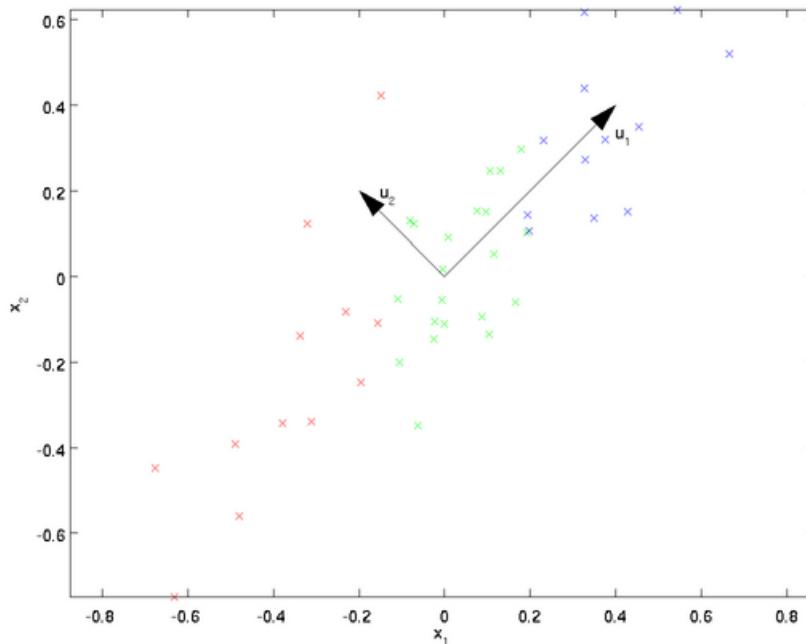


图 38: PCA 图解

(1) 给定  $N$  个样本  $x_1, x_2, x_3, \dots, x_N, x_i \in R^D$ ，现在将这些数据从  $D$  维降到  $K$  维 ( $K < D$ )，即需要找到  $K$  个主方向  $u_1, u_2, \dots, u_K \in R^D$  ( $u_i^T u_j = 0, i \neq j; u_i^T u_i = 1$ )，使得数据在这些方向映射之后整体方差最大。

(2) 不失一般性，只考虑  $u_1$ ，将所有样本投影到该方向上，得到  $u_1^T x_n, n = 1, \dots, N$ ，投影之后的均值为

$$u_1^T \bar{x} = u_1^T \frac{1}{N} \sum_{n=1}^N x_n;$$

(3) 计算该方向上的方差:

$$\frac{1}{N} \sum_{n=1}^N (u_1^T x_n - u_1^T \bar{x})^2 = \frac{1}{N} \sum_{n=1}^N (u_1^T x_n - u_1^T \bar{x})(u_1^T x_n - u_1^T \bar{x})^T = u_1^T S u_1$$

; 其中  $S = \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})(x_n - \bar{x})^T$ , 即未映射之前数据方差。

(4) 目标函数为  $\arg\max_{u_1} u_1^T S u_1$ , s.t.  $u_1^T u_1 = 1$ , 采用 lagrange 乘子法 (略去, LDA 部分会有介绍), 得到该函数最大值即为矩阵  $S = \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})(x_n - \bar{x})^T$  的最大特征值,  $u_1$  即矩阵  $S$  最大特征值对应的特征向量。

所以选取的  $K$  个主方向即为矩阵  $S$  的前  $K$  个特征值对应的特征向量, 再给定任意一个样本  $x$ , 和各个主方向做内积 ( $z_i = u_i^T x$ ) 就得到  $x$  在各个方向的分量  $z_i$ 。

**PCA: Finding Principal Components**

- Given:  $N$  examples  $x_1, \dots, x_N$ , each example  $x_n \in \mathbb{R}^D$
- Goal: Project the data from  $D$  dimensions to  $K$  dimensions ( $K < D$ )
- Want to capture the maximum possible variance in the projected data
- Let  $u_1, \dots, u_D$  be the principal components, assumed to be:
  - Orthogonal:  $u_i^T u_j = 0$  if  $i \neq j$ . Orthonormal:  $u_i^T u_i = 1$
- Each principal component is a vector of size  $D \times 1$
- We want only the first  $K$  principal components

(a) Step1

**PCA: Finding Principal Components**

- Projection of a data point  $x_n$  along  $u_1$ :  $u_1^T x_n$
- Projection of the mean  $\bar{x}$  along  $u_1$ :  $u_1^T \bar{x}$  (where  $\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n$ )
- Variance of the projected data (along projection direction  $u_1$ ):

$$\frac{1}{N} \sum_{n=1}^N \left\{ u_1^T x_n - u_1^T \bar{x} \right\}^2 = u_1^T S u_1$$

where  $S$  is the data covariance matrix defined as

$$S = \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})(x_n - \bar{x})^T$$

- Want to have  $u_1$  that maximizes the projected data variance  $u_1^T S u_1$ 
  - Subject to the constraint:  $u_1^T u_1 = 1$
  - We will introduce a Lagrange multiplier  $\lambda_1$  for this constraint

(b) Step2

**PCA: Finding Principal Components**

- Objective function:  $u_1^T S u_1 + \lambda_1(1 - u_1^T u_1)$
- Taking derivative w.r.t.  $u_1$  and setting it to zero gives:

$$S u_1 = \lambda_1 u_1$$

- This is the eigenvalue equation
  - $u_1$  must be an eigenvector of  $S$  (and  $\lambda_1$  the corresponding eigenvalue)
- But there are multiple eigenvectors of  $S$ . Which one is  $u_1$ ?
- Consider  $u_1^T S u_1 = u_1^T \lambda_1 u_1 = \lambda_1$  (using  $u_1^T u_1 = 1$ )
- We know that the projected data variance  $u_1^T S u_1 = \lambda_1$  is maximum
  - Thus  $\lambda_1$  should be the largest eigenvalue
  - Thus  $u_1$  is the first (top) eigenvector of  $S$  (with eigenvalue  $\lambda_1$ )
  $\Rightarrow$  the first principal component (direction of highest variance in the data)
- Subsequent PC's are given by the subsequent eigenvectors of  $S$

(c) Step3

**PCA: Finding Principal Components**

- Objective function:  $u_1^T S u_1 + \lambda_1(1 - u_1^T u_1)$
- Taking derivative w.r.t.  $u_1$  and setting it to zero gives:

$$S u_1 = \lambda_1 u_1$$

- This is the eigenvalue equation
  - $u_1$  must be an eigenvector of  $S$  (and  $\lambda_1$  the corresponding eigenvalue)
- But there are multiple eigenvectors of  $S$ . Which one is  $u_1$ ?
- Consider  $u_1^T S u_1 = u_1^T \lambda_1 u_1 = \lambda_1$  (using  $u_1^T u_1 = 1$ )
- We know that the projected data variance  $u_1^T S u_1 = \lambda_1$  is maximum
  - Thus  $\lambda_1$  should be the largest eigenvalue
  - Thus  $u_1$  is the first (top) eigenvector of  $S$  (with eigenvalue  $\lambda_1$ )
  $\Rightarrow$  the first principal component (direction of highest variance in the data)
- Subsequent PC's are given by the subsequent eigenvectors of  $S$

(d) 算法流程

图 39: 主成分分析推导过程以及算法流程

主成分分析可以用来压缩数据, 压缩过程为  $\hat{x} = \sum_{i=1}^K (x^T u_i) u_i = \sum_{i=1}^K z_i u_i$ , 著名的应用有 Eigen-faces。

当样本数  $N$  很小, 而特征维数  $D$  很大时 ( $N < D$ ), 传统 PCA 不再适用, 因为要求的特征值和特征向量的矩阵规模 ( $D * D$ ) 很大, 此时可以求解较小规模矩阵的特征向量和特征值 (对应关系如下图):

PCA 还有一种另一种优化形式如下:

观察其形式, PCA 与 AutoEncoder 非常类似。实际上:

(1) RBM 和 AutoEncoder 是非线性的 PCA, 都可以做到维数消减;

(2) RBM 和 AutoEncoder 可以连接很多层进行训练, PCA 不可以, PCA 是线性的, 多次嵌套还是相当于一次 PCA。

LDA (Linear Discriminant Analysis, 线性判别分析)

(1) 概念

LDA 是一种经典的降维方法, 和主成分分析 PCA 不考虑样本类别输出的无监督降维技术不同, LDA 是一种监督学习的降维技术, 数据集的每个样本有类别输出。

LDA 分类思想简单总结如下:

1. 多维空间中, 数据处理分类问题较为复杂, LDA 算法将多维空间中的数据投影到一条直线上, 将  $d$  维数据转化成 1 维数据进行处理;

- ## PCA for Very High Dimensional Data
- eigen-decomposition*
- In many cases,  $N < D$
  - Recall: PCA requires eigen-decomposition of  $D \times D$  covariance matrix  $\mathbf{S} = \frac{1}{N}\mathbf{X}\mathbf{X}^T$  (assuming centered data, and  $\mathbf{X}$  being  $D \times N$ )
  - Eigen-decomposition can be **expensive** if  $D$  is very large
  - **Fact:** If  $N < D$ , at most  $N - 1$  eigenvalues are non-zero
    - The remaining  $D - N + 1$  eigenvalues are zero
  - **Fact:**  $\mathbf{S} = \frac{1}{N}\mathbf{X}\mathbf{X}^T$  has the same  $N - 1$  non-zero eigenvalues as that of the  $N \times N$  matrix  $\frac{1}{N}\mathbf{X}^T\mathbf{X}$  (for which eigen-decomposition is cheaper if  $N < D$ )
  - The eigenvectors aren't exactly the same (but still related)
  - The relationship is  $\mathbf{u}_i = \frac{1}{(\bar{N}\lambda_i)^{1/2}}\mathbf{X}\mathbf{v}_i$
  - $\{\lambda_i, \mathbf{v}_i\}$  is an eigenvalue-eigenvector pair of the  $N \times N$  matrix  $\frac{1}{N}\mathbf{X}^T\mathbf{X}$ , and  $\mathbf{u}_i$  is the corresponding eigenvector of  $\mathbf{S} = \frac{1}{N}\mathbf{X}\mathbf{X}^T$  (that we want)

46

图 40: 样本数小于特征维数时的 PCA

## Optimality Property of PCA

Main theoretical result:

The matrix  $\mathbf{G}$  consisting of the first  $p$  eigenvectors of the covariance matrix  $\mathbf{S}$  solves the following min problem:

$$\min_{G \in \mathbb{R}^{d \times p}} \|X - G(G^T X)\|_F^2 \text{ subject to } G^T G = I_p$$

↓

$$\|X - \bar{X}\|_F^2 \quad \text{reconstruction error}$$

*PCA projection minimizes the reconstruction error among all linear projections of size  $p$ .*

50

图 41: PCA 的另一种优化形式

2. 对于训练数据，设法将多维数据投影到一条直线上，同类数据的投影点尽可能接近，异类数据点尽可能远离；

3. 对数据进行分类时，将其投影到同样的这条直线上，再根据投影点的位置来确定样本的类别；用一句话概括 LDA 思想，即“投影后类内方差最小，类间方差最大”。

假设红、蓝两类数据，这些数据特征均为二维，如下图所示。我们的目标是将这些数据投影到一维，让每一类相近的数据的投影点尽可能接近，不同类别数据尽可能远，即图中红色和蓝色数据中心之间的距离尽可能大。

左图和右图是两种不同的投影方式。左图思路：让不同类别的平均点距离最远的投影方式，右图思路：

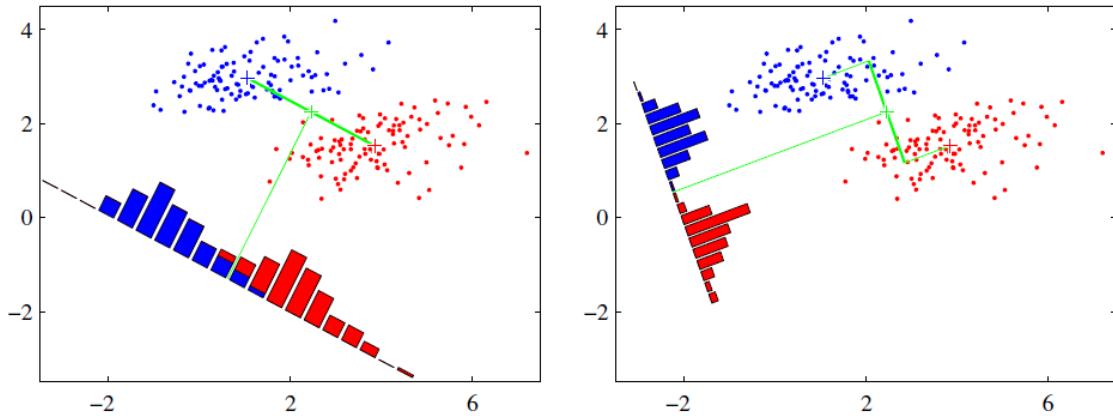


图 42: LDA 图解

让同类别的数据挨得最近的投影方式。从上图直观看出，右图红色数据和蓝色数据在各自的区域来说相对集中，根据数据分布直方图也可看出，所以右图的投影效果好于左图，左图中间直方图部分有明显交集。以上例子是基于数据是二分类的，分类后的投影是一条直线。如果原始数据是多维的，则投影后的分类面是一个低维的超平面（维数是  $N-1$ ,  $N$  是分类数）。

## (2) 二分类 LDA 推导

输入：数据集  $D = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \dots, (\mathbf{x}_m, \mathbf{y}_m)\}$ ，其中样本  $\mathbf{x}_i$  是  $n$  维向量， $\mathbf{y}_i \in \{0, 1\}$ ，降维后的目标维度  $d$ 。定义：

$N_j(j = 0, 1)$  为第  $j$  类样本个数；

$X_j(j = 0, 1)$  为第  $j$  类样本的集合；

$u_j(j = 0, 1)$  为第  $j$  类样本的均值向量；

$\sum_j(j = 0, 1)$  为第  $j$  类样本的协方差矩阵。

其中

$$u_j = \frac{1}{N_j} \sum_{\mathbf{x} \in X_j} \mathbf{x} (j = 0, 1), \quad \sum_j = E[(\mathbf{x} - u_j)(\mathbf{x} - u_j)^T] (j = 0, 1)$$

假设投影直线是向量  $\mathbf{w}$ ，对任意样本  $\mathbf{x}_i$ ，它在直线  $w$  上的投影为  $\mathbf{w}^T \mathbf{x}_i$ ，两个类别的中心点  $u'_0, u'_1$  在直线  $w$  的投影分别为  $\mathbf{w}^T u_0, \mathbf{w}^T u_1$ ；两个类别的协方差矩阵  $\Sigma'_0, \Sigma'_1$  在直线  $w$  的投影分别为  $\mathbf{w}^T \Sigma_0 \mathbf{w}, \mathbf{w}^T \Sigma_1 \mathbf{w}$ （如： $\Sigma'_0 = E[(\mathbf{w}^T \mathbf{x} - \mathbf{w}^T u_0)(\mathbf{w}^T \mathbf{x} - \mathbf{w}^T u_0)^T] = \mathbf{w}^T \Sigma_0 \mathbf{w}$ ）。

LDA 的目标是让两类别的数据中心间的距离  $\|\mathbf{w}^T u_0 - \mathbf{w}^T u_1\|_2^2$  尽量大，与此同时，希望同类样本投影点的协方差  $\mathbf{w}^T \sum_0 \mathbf{w}, \mathbf{w}^T \sum_1 \mathbf{w}$  尽量小，最小化  $\mathbf{w}^T \sum_0 \mathbf{w} - \mathbf{w}^T \sum_1 \mathbf{w}$ 。

定义类内散度矩阵：

$$S_w = \sum_0 + \sum_1 = \sum_{\mathbf{x} \in X_0} (\mathbf{x} - u_0)(\mathbf{x} - u_0)^T + \sum_{\mathbf{x} \in X_1} (\mathbf{x} - u_1)(\mathbf{x} - u_1)^T。$$

类间散度矩阵  $S_b = (u_0 - u_1)(u_0 - u_1)^T$  据上分析，优化目标为：

$$\arg \max_{\mathbf{w}} J(\mathbf{w}) = \frac{\|\mathbf{w}^T u_0 - \mathbf{w}^T u_1\|_2^2}{\mathbf{w}^T \sum_0 \mathbf{w} + \mathbf{w}^T \sum_1 \mathbf{w}} = \frac{\mathbf{w}^T (u_0 - u_1)(u_0 - u_1)^T \mathbf{w}}{\mathbf{w}^T (\sum_0 + \sum_1) \mathbf{w}} = \frac{\mathbf{w}^T S_b \mathbf{w}}{\mathbf{w}^T S_w \mathbf{w}}。$$

求解：注意到分子分母都是关于  $\mathbf{w}$  的二次项，所以比值和  $\mathbf{w}$  的长度无关，只和方向相关，不失一般性，

假设  $\mathbf{w}^T S_w \mathbf{w} = 1$ , 则优化目标可以简化成:

$$\arg \min_{\mathbf{w}} J(\mathbf{w}) = -\mathbf{w}^T S_b \mathbf{w}$$

$$s.t. \mathbf{w}^T S_w \mathbf{w} = 1$$

使用 lagrange 乘子法, 目标函数为:

$$F(\lambda) = -\mathbf{w}^T S_b \mathbf{w} + \lambda(\mathbf{w}^T S_w \mathbf{w} - 1)$$

该函数分别对  $\lambda$  以及  $\mathbf{w}$  求导, 并令结果为 0, 可以得到:

$$\lambda S_w \mathbf{w} = S_b \mathbf{w}$$

$$S_w^{-1} S_b \mathbf{b} = \lambda \mathbf{w}$$

即,  $\mathbf{w}$  是矩阵  $S_w^{-1} S_b$  的一个特征值, 对应的特征向量即为  $\mathbf{w}$ 。将此条件代入  $J(\mathbf{w})$ , 得到  $\arg \max_{\mathbf{w}} J(\mathbf{w}) = \lambda$ , 取矩阵  $S_w^{-1} S_b$  的最大特征值以及对应的特征向量即为最优解。

补充: 由于  $S_b \mathbf{w} = (u_0 - u_1)(u_0 - u_1)^T \mathbf{w}$  的方向与  $u_0 - u_1$  一致, 可以取  $S_b \mathbf{w} = \lambda(u_0 - u_1)$ ; 代入  $\lambda S_w \mathbf{w} = S_b \mathbf{w}$ , 得到  $\mathbf{w}$  的一个解:  $\mathbf{w} = S_w^{-1}(u_0 - u_1)$ 。

从以上推导过程来看, 最终投影的结果最少也得是一个值, 故降维最多降  $k - 1$  维,  $k$  是类别数。

CCA(Canonical Correlation Analysis, 典型相关性分析)

两组变量的相关性, 经济学当中使用较多, 不适合解决分类问题。

ICA(Independent Component Analysis, 独立成分分析)

ICA 用于去除冗余, 和 PCA 不同, ICA 追求的是输出的变量相互独立, 而非仅不相关, 因此, ICA 需要利用数据分布的高阶统计信息而非仅二阶信息。

## 2. 非线性降维

### Kernel PCA

在 PCA 的基础上引入核方法, PCA 是在给定的样本下算出样本的协方差矩阵, 然后计算该矩阵的较大特征值对应的特征向量作为主成分, 便于解决线性数据的降维问题。

Kernel PCA 主要解决非线性降维问题, 对于给定的非线性数据, 首先引入映射  $\phi(x)$  变为线性关系, 在空间  $\phi$  中进行 PCA 降维, 得到主成分  $v_i$ 。

对于给定的样本, 先映射到  $\phi$  空间进行投影计算各主成分  $v_i$  的系数, 由于  $\phi$  空间比较复杂, 主成分  $v_i$  不容易计算。所以引入核方法, 可以用核矩阵的主成分  $a_i$  来代替计算  $v_i$ 。

第四步里面要计算降维后的样本的各个分量  $z_i = \phi^T v_i$ , 先将样本变换到高维空间, 再和相应维度的主成分做内积, 但是  $\phi$  和  $v_i$  都不知道, 所以可以使用核矩阵的特征向量  $a_i$  和以及已知的核函数  $k(x_i, x_j)$  代替求解。推导过程如下:

(1) 给定  $N$  个样本  $x_1, x_2, \dots, x_n, x_i \in R^D$ , 将这些样本通过映射  $\phi$  映射到高维空间中, 生成线性数据样本  $\phi(x_1), \phi(x_2), \dots, \phi(x_n)$ ;

(2) 按照线性 PCA 的思路, 计算这些数据的协方差阵  $C = \frac{1}{N} \sum_{n=1}^N [\phi(x_n) - \bar{\phi}(x)][\phi(x_n) - \bar{\phi}(x)]^T$ , 假设映射后数据的均值为 0, 即  $\bar{\phi}(x) = \sum_{n=1}^N \phi(x_n) = 0$ , 则矩阵  $C = \frac{1}{N} \sum_{n=1}^N \phi(x_n) \phi(x_n)^T$ ;

(3) 设该矩阵的特征值和特征向量为  $Cv_i = \lambda_i v_i$ , 代入 (2) 中矩阵  $C$  的表达式, 得到:

$$\frac{1}{N} \sum_{n=1}^N \phi(x_n) [\phi(x_n)^T v_i] = \lambda_i v_i$$

通过上式, 可以将  $v_i$  重新记为  $v_i = \sum_{n=1}^N a_{in} \phi(x_n)$ ;

(4) 将  $v_i = \sum_{n=1}^N a_{in} \phi(x_n)$  重新代入  $\frac{1}{N} \sum_{n=1}^N \phi(x_n) [\phi(x_n)^T v_i] = \lambda_i v_i$ , 得到:

$$\frac{1}{N} \sum_{n=1}^N \phi(x_n) \phi(x_n)^T \sum_{m=1}^N a_{im} \phi(x_m) = \lambda_i \sum_{n=1}^N a_{in} \phi(x_n)$$

(5) 两边同乘  $\phi(x_l)$ , 得到:

$$\frac{1}{N} \sum_{n=1}^N \phi(x_l)^T \phi(x_n) \sum_{m=1}^N a_{im} \phi(x_n)^T \phi(x_m) = \lambda_i \sum_{n=1}^N a_{in} \phi(x_l)^T \phi(x_n)$$

假设  $\phi(x_n)^T \phi(x_m) = K(x_n, x_m)$ , 上式变为:

$$\frac{1}{N} \sum_{n=1}^N K(x_l, x_n) \sum_{m=1}^N a_{im} K(x_n, x_m) = \lambda_i \sum_{n=1}^N a_{in} K(x_l, x_n)$$

(6) 假设  $K = [k(x_n, x_m)]_{N \times N}$ ,  $a_i = [a_{in}]_{N \times 1}$ , 则上式变为:

$$K^2 a_i = \lambda_i N K a_i, \text{ 即 } K a_i = \lambda_i N a_i$$

通过此式可以计算矩阵  $K$  的特征值和特征向量, 得到  $a_i$ , 计算前  $K$  个最大特征值对应的特征向量  $a_1, a_2, \dots, a_K$ ;

(7) 任意给定一个样本  $x$ , 先将  $x$  映射到高维空间进行主方向的映射得到系数  $z_i$ , 即  $z_i = \phi(x)^T v_i$ , 将

(3) 式子  $v_i = \sum_{n=1}^N a_{in} \phi(x_n)$  代入, 即有:

$$z_i = \phi(x)^T v_i = \sum_{n=1}^N a_{in} k(x, x_n)$$

通过选取合适的核函数  $k(x, y)$  就可以避免在高维空间计算主方向  $v_i$ , 通过计算核矩阵  $K$  的特征向量即可将样本降维。

### 流形学习 (Manifold Learning)

(1)LLE(Locally Linear Embedding , 局部线性嵌入)

1) 在高维空间中, 用样本的邻居来重构该样本, 即求解一个 QP 问题;

2) 求解得系数之后, 利用该系数来拟合低维表示。

优点: 能保持近邻关系, 即保持流形。

算法与效果如下:

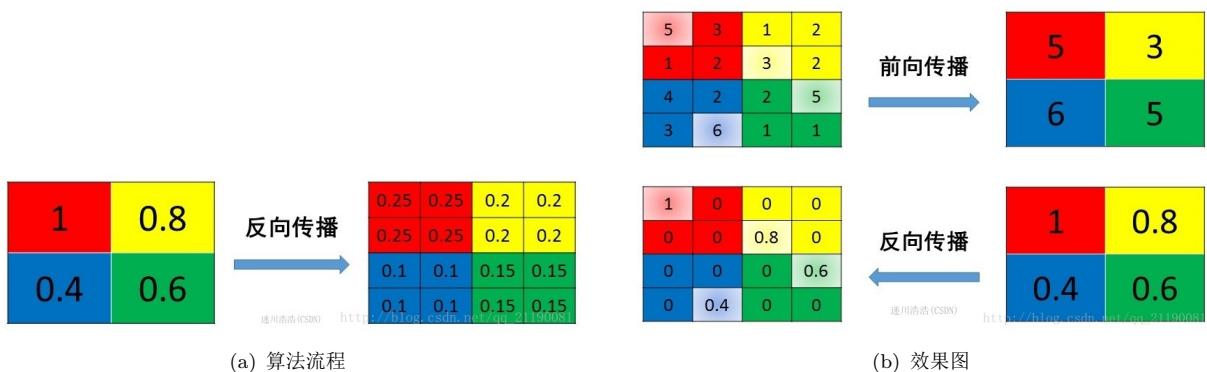


图 43: 算法流程以及效果图

(2)ISOMAP(Isometric Feature Mapping , 等距特征映射)

计算高维空间中两个点的测地距离, 降维的时候保持距离不变。

## 深度学习部分

## Question

### 手推 BP(Back Propagation) 算法。

先设出以下变量 (只考虑一个样本的三层全连接神经网络):

(1) 训练数据输入输出对:  $\{x = (x_1, x_2, \dots, x_i, \dots, x_d), t = (t_1, t_2, \dots, t_j, \dots, t_c)\}$ , 样本已经包括偏移量;

(2) 输出层节点的加权输入及输出:  $\{net = (net_1, net_2, \dots, net_j, \dots, net_c), z = (z_1, z_2, \dots, z_j, \dots, z_c)\}$ ;

(3) 隐藏层节点的加权输入及输出:  $\{neth = (neth_1, neth_2, \dots, neth_h, \dots, neth_{n_H}), y = (y_1, y_2, \dots, z_h, \dots, y_{n_H})\}$ ;

(4) 输入层节点  $i$  到隐藏层节点  $h$  的权重:  $W_{ih}$ ;

(5) 隐藏层节点  $h$  到输出层节点  $j$  的权重:  $W_{hj}$ ;

(6) 损失函数使用  $MSE, J(W) = \frac{1}{2} \sum_{j=1}^c (t_j - z_j)^2$ .

则:

(1) 隐藏层节点  $h$  的输入加权和:  $neth_h = \sum_{i=1}^d W_{ih} x_i$ ;

(2) 经过激励函数, 隐藏层节点  $h$  的输出:  $y_h = f_1(neth_h) = f_1(\sum_{i=1}^d W_{ih} x_i), f_1(x) = \frac{1}{1+e^{-x}}$ ;

- (3) 输出层节点  $i$  的输入加权和:  $net_j = \sum_{h=1}^{n_H} W_{hj} y_h = \sum_{h=1}^{n_H} W_{hj} f_1(\sum_{i=1}^d W_{ih} x_i)$ ;  
(4) 经过激励, 输出层节点  $j$  的输出:  $z_j = f(net_j) = f_2(\sum_{h=1}^{n_H} W_{hj} y_h) = f_2(\sum_{h=1}^{n_H} W_{hj} f_1(\sum_{i=1}^d W_{ih} x_i)), f_2(x_i) = \frac{e^{x_i}}{\sum_{i=1}^n e^{x_i}}$ .

1. 先确定隐藏层节点  $h$  到输出层节点  $j$  的连接权重调节量:

$$\begin{aligned}\Delta W_{hj} &= -\eta \frac{\partial J}{\partial W_{hj}} \\ &= -\eta \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial W_{hj}} \\ &= -\eta \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial W_{hj}} \\ &= \eta(t_j - z_j) f'_2(net_j) y_h \\ &= \eta \delta_j y_h\end{aligned}$$

其中  $\delta_j = -\frac{\partial J}{\partial net_j} = f'_2(net_j)(t_j - z_j) = (t_j - z_j) \frac{e^{net_j} (\sum_{k \neq j} e^{net_k})}{\sum_{j=1}^c e^{net_j}} = (t_j - z_j) f_2(net_j)(1 - f_2(net_j))$

2. 再确定输入层节点  $i$  到隐藏层节点  $h$  的连接权重调节量:

$$\begin{aligned}\Delta W_{ih} &= -\eta \frac{\partial J}{\partial W_{ih}} \\ &= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial W_{ih}} \\ &= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial W_{ih}} \\ &= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial y_h} \frac{y_h}{\partial W_{ih}} \\ &= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial y_h} \frac{\partial y_h}{\partial neth_h} \frac{\partial neth_h}{\partial W_{ih}} \\ &= \eta \sum_{j=1}^c (t_j - z_j) f'_2(net_j) W_{hj} f'_1(neth_h) x_i \\ &= \eta \delta_h x_i\end{aligned}$$

其中  $\delta_h = -\frac{\partial J}{\partial neth_h} = \sum_{j=1}^c (t_j - z_j) f'_2(net_j) W_{hj} f'_1(neth_h) = f'_1(neth_h) \sum_{j=1}^c W_{hj} \delta_j$

总结: 相邻层之间连接节点  $a$  到  $b$  的权重调节量由两部分决定, 一是边起始对应的节点的输出 (激励之后), 二是边指向对应的节点收集到的误差 (即损失函数对该点加权和的导数的相反数, 此导数是经过激励函数的导数放缩过的)。前一层所收集到的误差等于后一层收集到的误差的加权求和再缩放一个前一层激励函数的导数。

# Question

## 网络训练的常见问题

1. 初始化：网络的初始权重可正可负，通常从一个均匀分布中随机选择初始值， $-w_0 < w < w_0$ ；
2. 正则化：为了防止网络出现 *overfitting* 的一种有效方法是采用一些正则化技术，如利用矩阵的 2-范数修正损失函数， $E_{new}(w) = E(w) + kw^T w$ ，无论是哪种正则化方式，基本的思想都是希望通过限制权重的大小，使得模型不能任意拟合训练数据中的随机噪声。对比  $L_1$  正则化和  $L_2$  正则化：  
 (1)  $L_2$  正则化，给损失函数后面加一个正则化项  $L(W) = l(W) + \lambda \|W\|_2^2$ ，这里的正则化项可以看成是向量范数，即  $\|W\|_2^2 = \sum_{i=1}^n w_i^2$ ，则有：

$$\begin{aligned} w_i &= w_i - \eta \frac{\partial L}{\partial w_i} \\ &= w_i - \eta \left[ \frac{\partial l}{\partial w_i} + 2\lambda w_i \right] \\ &= (1 - 2\lambda\eta)w_i - \eta \frac{\partial l}{\partial w_i} \end{aligned}$$

深度学习训练中，参数  $\lambda$  会非常小，导致  $(1 - 2\lambda\eta)$  是一个小于 1 的系数，故  $L_2$  正则化迭代过程会使权重参数衰减较快。

- (2)  $L_1$  正则化，给损失函数后面加一个正则化项  $L(W) = l(W) + \lambda \|W\|_1$ ，这里的正则化项可以看成是向量范数，即  $\|W\|_1 = \sum_{i=1}^n |w_i|$ ，则有：

$$\begin{aligned} w_i &= w_i - \eta \frac{\partial L}{\partial w_i} \\ &= w_i - \eta \left[ \frac{\partial l}{\partial w_i} + \lambda sign(w_i) \right] \\ &= w_i - \lambda\eta sign(w_i) - \eta \frac{\partial l}{\partial w_i} \end{aligned}$$

符号函数或 1 或 -1，在 0 处为 0。 $\lambda\eta sign(w_i)$  是一个固定大小的系数，故  $L_1$  正则化迭代过程每次权重都是衰减一个相同的数。

为什么  $L_1$  正则化的解一般要比  $L_2$  正则化的解要稀疏？假设权重参数矩阵  $W$  只有两个参数  $w_1$  和  $w_2$ ，损失函数  $l(W)$  一般在第一象限，从下图中看出  $L_1$  正则化使得  $w_1$  和  $w_2$  形成一个斜正方形， $L_2$  正则化使得  $w_1$  和  $w_2$  形成一个圆形，该圆形包含  $L_1$  正则化的斜正方形，使得相同函数值的时候， $L_1$  正则化的解要比  $L_2$  更靠近原点，即解更加稀疏。一句话总结就是：L1 使权重稀疏，L2 使权重平滑，L1 会趋向于产生少量的特征，而其他的特征都是 0，而 L2 会选择更多的特征，这些特征都会接近于 0，实践中 L2 正则化通常优于 L1 正则化。

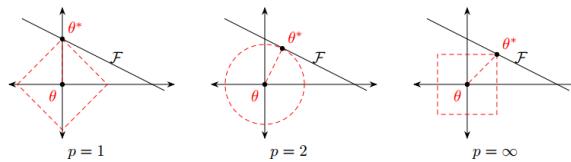


图 44: 不同正则化情况

3. 归一化：批量归一化 (batch normalization) 层能让较深的神经网络的训练变得更加容易，归一化分为标准化和仿射变换两步骤，处理后的任意一个特征在数据集中所有样本上的均值为 0、标准差为

1. 标准化处理输入数据使各个特征的分布相近：这往往更容易训练出有效的模型。通常来说，数据标准化预处理对于浅层模型就足够有效了。随着模型训练的进行，当每层中参数更新时，靠近输出层的输出较难出现剧烈变化。但对深层神经网络来说，即使输入数据已做标准化，训练中模型参数的更新依然很容易造成靠近输出层输出的剧烈变化。这种计算数值的不稳定性通常令我们难以训练出有效的深度模型。批量归一化的提出正是为了应对深度模型训练的挑战。在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。批量归一化和残差网络为训练和设计深度模型提供了两类重要思路。

对全连接层和卷积层做批量归一化的方法稍有不同，分别介绍这两种情况下的批量归一化。

(1) 对全连接层做批量归一化 我们先考虑如何对全连接层做批量归一化。通常，我们将批量归一化层置于全连接层中的仿射变换和激活函数之间。设全连接层的输入为  $u$ ，权重参数和偏差参数分别为  $W$  和  $b$ ，激活函数为  $\phi$ 。设批量归一化的运算符为 BN。那么，使用批量归一化的全连接层的输出为  $\phi(\text{BN}(x))$ ，其中批量归一化输入  $x$  由仿射变换  $x = Wu + b$  得到。考虑一个由  $m$  个样本组成的小批量，仿射变换的输出为一个新的小批量  $B = \{x^{(1)}, \dots, x^{(m)}\}$ 。它们正是批量归一化层的输入。对于小批量  $B$  中任意样本  $x^{(i)} \in R^d, 1 \leq i \leq m$ ，批量归一化层的输出同样是  $d$  维向量  $y^{(i)} = \text{BN}(x^{(i)})$ ，并由以下几步求得。首先，对小批量  $B$  求均值和方差：

$$\begin{aligned}\mu_B &\leftarrow \frac{1}{m} \sum_{i=1}^m x^{(i)}, \\ \sigma_B^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu_B)^2,\end{aligned}$$

其中的平方计算是按元素求平方。接下来，使用按元素开方和按元素除法对  $x^{(i)}$  标准化：

$$\hat{x}^{(i)} \leftarrow \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}},$$

这里  $\epsilon > 0$  是一个很小的常数，保证分母大于 0。在上面标准化的基础上，批量归一化层引入了两个可以学习的模型参数，拉伸 (scale) 参数  $\gamma$  和偏移 (shift) 参数  $\beta$ 。这两个参数和  $x^{(i)}$  形状相同，皆为  $d$  维向量。它们与  $x^{(i)}$  分别做按元素乘法 (符号  $\odot$ ) 和加法计算： $y^{(i)} \leftarrow \gamma \odot \hat{x}^{(i)} + \beta$ 。

至此，我们得到了  $x^{(i)}$  的批量归一化的输出  $y^{(i)}$ 。值得注意的是，可学习的拉伸和偏移参数保留了不对  $\hat{x}^{(i)}$  做批量归一化的可能：此时只需学出  $\gamma = \sqrt{\sigma_B^2 + \epsilon}$  和  $\beta = \mu_B$ 。我们可以对此这样理解：如果批量归一化无益，理论上，学出的模型可以不使用批量归一化。

(2) 对卷积层做批量归一化 对卷积层来说，批量归一化发生在卷积计算之后、应用激活函数之前。如果卷积计算输出多个通道，我们需要对这些通道的输出分别做批量归一化，且每个通道都拥有独立的拉伸和偏移参数，并均为标量。设小批量中有  $m$  个样本。在单个通道上，假设卷积计算输出的高和宽分别为  $p$  和  $q$ 。我们需要对该通道中  $m \times p \times q$  个元素同时做批量归一化。对这些元素做标准化计算时，我们使用相同的均值和方差，即该通道中  $m \times p \times q$  个元素的均值和方差。

(3) 预测归一化：使用批量归一化训练时，我们可以将批量大小设得大一点，从而使批量内样本的均值和方差的计算都较为准确。将训练好的模型用于预测时，我们希望模型对于任意输入都有确定的输出。因此，单个样本的输出不应取决于批量归一化所需要的随机小批量中的均值和方差。一种常用的方法是通过移动平均估算整个训练数据集的样本均值和方差，并在预测时使用它们得到确定的输出。可见，和 Dropout 层一样，批量归一化层在训练模式和预测模式下的计算结果也是不一样的，在 pytorch 中使用 `model.eval()` 来固定 BN 层的参数以及关闭 Dropout 层。

4. 学习率太小，则收敛太慢，太大则不稳定；

5. 网络“训不动”：网络训不动分为梯度消失和网络麻痹。从上一题 BP 算法的推导来看，误差反向传播的过程中，误差会乘以激励函数的导数，是慢慢缩小的，会导致梯度消失。当这个导数趋于 0 的时候，误差会趋于 0，导致网络麻痹。

6. 深度学习中的最优化迭代方法

(1) SGD，随机梯度下降，将最优化中的梯度下降法用于深度学习训练中，不一样的是随机二字，每次的梯度只是随机一部分样本的梯度；

(2) 带 momentum 的 SGD, 迭代公式如下:

$$\mathbf{v}_t \leftarrow \gamma \mathbf{v}_{t-1} + \eta_t \mathbf{g}_t,$$

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{v}_t,$$

从公式中可以看出, 每次的更新量不仅仅是步长和当前梯度的乘积, 还需要考虑上一次迭代时的步长和梯度的乘积 (动量) 的加权和。比如现有目标函数  $f(\mathbf{x}) = 0.01x_1^2 + 100x_2^2$ , 在使用梯度下降法时, 每次求梯度  $x_2$  方向更新的量非常大, 导致  $x_2$  方向震荡很快, 但  $x_1$  方向进步很小, 所以时更新时, 和上一次的更新量做一个加权和 (向量相加), 可以很好的缓解两个方向变化差距过大的情况, 如下图:

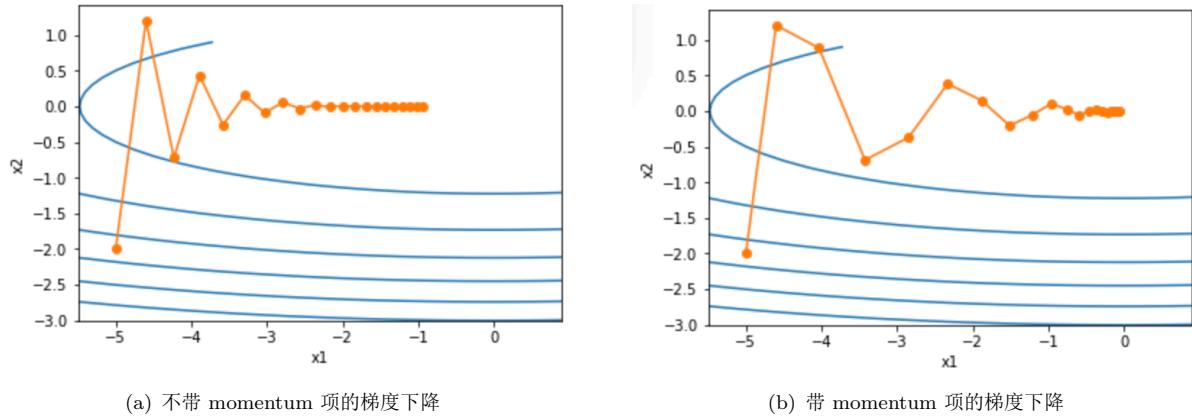


图 45: momentum

(3) AdaGrad:

AdaGrad 根据自变量在每个维度的梯度值的大小来调整各个维度上的学习率, 从而避免统一的学习率难以适应所有维度的问题, AdaGrad 算法会使用一个小批量随机梯度  $\mathbf{g}_t$  按元素平方的累加变量  $\mathbf{s}_t$ 。在时间步 0, AdaGrad 将  $\mathbf{s}_0$  中每个元素初始化为 0。在时间步  $t$ , 首先将小批量随机梯度  $\mathbf{g}_t$  按元素平方后累加到变量  $\mathbf{s}_t$ :

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t,$$

其中  $\odot$  是按元素相乘。接着, 我们将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下:

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t,$$

其中  $\eta$  是学习率,  $\epsilon$  是为了维持数值稳定性而添加的常数, 如  $10^{-6}$ 。这里开方、除法和乘法的运算都是按元素运算的。这些按元素运算使得目标函数自变量中每个元素都分别拥有自己的学习率。需要强调的是, 小批量随机梯度按元素平方的累加变量  $\mathbf{s}_t$  出现在学习率的分母项中。因此, 如果目标函数有关自变量中某个元素的偏导数一直都较大, 那么该元素的学习率将下降较快; 反之, 如果目标函数有关自变量中某个元素的偏导数一直都较小, 那么该元素的学习率将下降较慢。

(4) RMSProp:

在 momentum 里介绍过指数加权移动平均。不同于 AdaGrad 算法里状态变量  $\mathbf{s}_t$  是截至时间步  $t$  所有小批量随机梯度  $\mathbf{g}_t$  按元素平方和, RMSProp 算法将这些梯度按元素平方做指数加权移动平均。具体

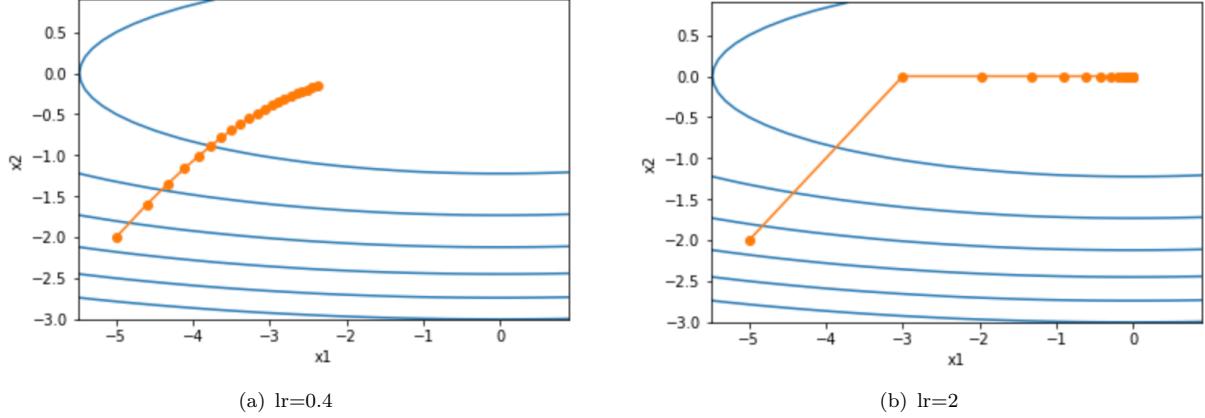


图 46: Adagrad

来说, 给定超参数  $0 \leq \gamma < 1$ , RMSProp 算法在时间步  $t > 0$  计算:

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t \odot \mathbf{g}_t$$

和 AdaGrad 算法一样, RMSProp 算法将目标函数自变量中每个元素的学习率通过按元素运算重新调整, 然后更新自变量:

$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t$ , 其中  $\eta$  是学习率,  $\epsilon$  是为了维持数值稳定性而添加的常数, 如  $10^{-6}$ 。因为 RMSProp 算法的状态变量  $\mathbf{s}_t$  是对平方项  $\mathbf{g}_t \odot \mathbf{g}_t$  的指数加权移动平均。

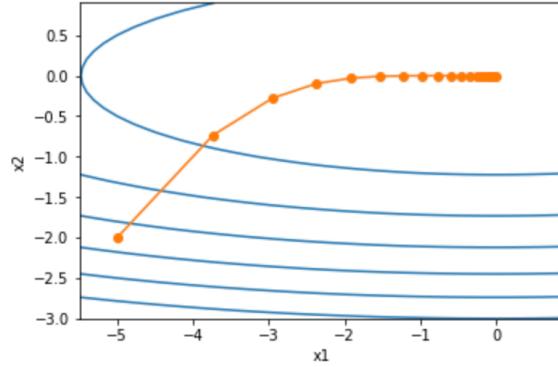


图 47: RMSProp

(5) Adam:

Adam 算法使用了动量变量  $\mathbf{v}_t$  和 RMSProp 算法中小批量随机梯度按元素平方的指数加权移动平均变量  $\mathbf{s}_t$ , 并在时间步 0 将它们中每个元素初始化为 0。给定超参数  $0 \leq \beta_1 < 1$ (算法作者建议设为 0.9), 时间步  $t$  的动量变量  $\mathbf{v}_t$  即小批量随机梯度  $\mathbf{g}_t$  的指数加权移动平均:

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t$$

和 RMSProp 算法中一样, 给定超参数  $0 \leq \beta_2 < 1$ (算法作者建议设为 0.999), 将小批量随机梯度按元素平方后的项  $\mathbf{g}_t \odot \mathbf{g}_t$  做指数加权移动平均得到  $\mathbf{s}_t$ :

$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t$$

由于我们将  $\mathbf{v}_0$  和  $\mathbf{s}_0$  中的元素都初始化为 0, 在时间步  $t$  我们得到:  $\mathbf{v}_t = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} \mathbf{g}_i$ 。将过去各时间步小批量随机梯度的权值相加, 得到  $(1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} = 1 - \beta_1^t$ 。需要注意的是, 当  $t$  较小时, 过去各时间步小批量随机梯度权值之和会较小。例如, 当  $\beta_1 = 0.9$  时,  $\mathbf{v}_1 = 0.1 \mathbf{g}_1$ 。为了消除这样

的影响，对于任意时间步  $t$ ，我们可以将  $\mathbf{v}_t$  再除以  $1 - \beta_1^t$ ，从而使过去各时间步小批量随机梯度权值之和为 1。这也叫作偏差修正。在 Adam 算法中，我们对变量  $\mathbf{v}_t$  和  $\mathbf{s}_t$  均作偏差修正：

$$\hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1 - \beta_1^t},$$

$$\hat{\mathbf{s}}_t \leftarrow \frac{\mathbf{s}_t}{1 - \beta_2^t}.$$

接下来，Adam 算法使用以上偏差修正后的变量  $\hat{\mathbf{v}}_t$  和  $\hat{\mathbf{s}}_t$ ，将模型参数中每个元素的学习率通过按元素运算重新调整：

$$\mathbf{g}'_t \leftarrow \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}},$$

其中  $\eta$  是学习率， $\epsilon$  是为了维持数值稳定性而添加的常数，如  $10^{-8}$ 。和 AdaGrad 算法、RMSProp 算法以及 AdaDelta 算法一样，目标函数自变量中每个元素都分别拥有自己的学习率。最后，使用  $\mathbf{g}'_t$  迭代自变量：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t.$$

## Question

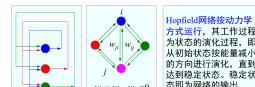
简单介绍 Hopfield 网络，BM, RBM, DBN, DBM。

在工业界研究 CNN 的时候，科学界主要在研究下面这些内容。

1. Hopfield 网络中各个节点地位等同，全连接起来都是输入输出节点，从初始状态开始运行到最终的

### 6.7.1 Hopfield 网络

- 网络结构



常见的两种形式

网络演化特点

第29页

(a) Hopfield

### 6.7.2 玻尔兹曼机

- Boltzman Machine, BM

是一种随机的 Hopfield 网络，是具有隐单元的反馈互联网络



$W_j = W_{ij}, V_0 = 0$

网络结构复杂、训练代价大、局部极小

第33页

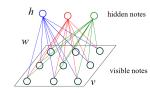
(b) BM

### 6.7.3 受限玻尔兹曼机

- Restricted BM, RBM

具有两层结构，层内节点不相连，信息可双向流动

包含可视节点层（与外界相连）和隐含层（状态层）



可见节点层 (与外界相连)

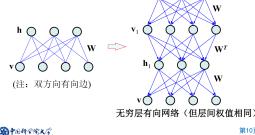
隐含层 (状态层)

第34页

(c) RBM

### RBM

- 对网络结构的等价解释

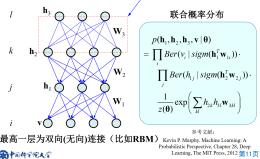


无穷层有向网络 (但层间权值相等)

第10页

(d) RBM 的等价形式

### 深度信念网络(DBN)



联合概率分布

$$p(\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_l, \mathbf{v})$$

$$= \prod_i Ber(h_i | sigmoid(w_i^\top \mathbf{v})) \cdot$$

$$\prod_i Ber(h_i | sigmoid(w_i^\top \mathbf{h}_{i-1})) \cdot$$

$$\prod_i \exp(-\sum_j h_i h_{i,j} w_{i,j}) / z(\mathbf{w})$$

最高一层为双向(无向)连接 (比如RBM)

Kevin P. Murphy. Machine Learning: A Probabilistic Perspective. Chapter 26. Deep Learning. The MIT Press, 2012. 第11页

(e) DBN

### 深度 Boltzman 机 (DBM)

- 联合概率分布

$$p(\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_l, \mathbf{v} | \mathbf{w})$$

$$= \frac{1}{z(\mathbf{w})} \exp \left( \sum_i v_i h_i w_{i,0} \right) \cdot$$

$$\exp \left( \sum_i h_i h_{i,1} w_{i,1} \right) \cdot$$

$$\exp \left( \sum_i h_i h_{i,2} w_{i,2} \right) \cdots$$

层与层均为双向连接 (比如RBM)

Kevin P. Murphy. Machine Learning: A Probabilistic Perspective. Chapter 26. Deep Learning. The MIT Press, 2012. 第12页

(f) DBM

图 48: Hopfield,BM,RBM,DBN,DBM

平衡状态：

2. BM(BoltzmanMachine) 与 Hopfield 网络不同的是，BM 对节点功能做了区分，其中的一部分神经元是输入输出，受外界条件影响，另一部分视为隐藏节点，是一个深层网络；

3. RBM(RestrictedBoltzmanMachine) 中，可视节点与隐藏节点相连(全连接)，层内不连接。训练的时候需要用到隐藏节点与可视节点的联合分布(较复杂！)，RBM 是一种很重要的特征表示方法，跟后续的自动编码器功能类似，是反馈神经网络的典型代表。

RBM 有一个等价的形式，如图中 (d) 所示，因为 RBM 是双向流动的，所以相当于一个无穷层有向网络，但层间权值相等。

4. DBN(deepbeliefnetwork) 是玻尔兹曼机 (双向)+一般的前向网络 (单向)。

## Question

简单介绍 CNN 网络, AutoEncoder, RNN, LSTM。

1. CNN 实际上是前向神经网络的特例, 是少量神经元的线性加权求和再激励, 所以其反向传播过程与全连接前向神经网络类似。池化操作是在每个通道上做池化, 添加了池化层的网络, 其反向传播过程与前向神经网络不一样, 当池化模板是  $2 * 2$  时, 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(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 ([https://blog.csdn.net/Jason\\_yz/article/details/80003271](https://blog.csdn.net/Jason_yz/article/details/80003271))。

1. 平均池化: mean pooling 的前向传播就是把一个模板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之和保持不变;

2. 最大池化: max pooling 也要满足梯度之和不变的原则, max pooling 的前向传播是把模板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。

两种池化方式反向传播的过程见下图:

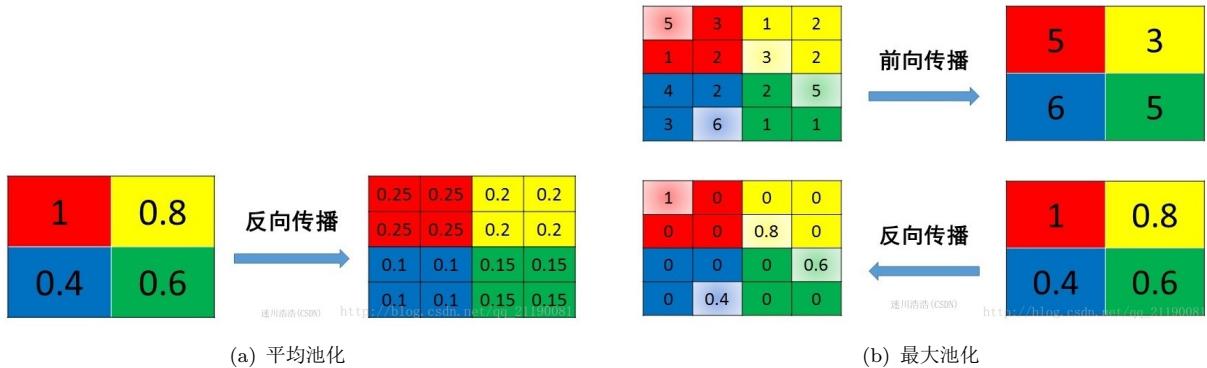


图 49: 池化层反向传播

卷积操作可以大幅度减少参数, 主要通过两个途径: (1) 局部连接: 原因有二, 一是视觉生理学相关研究普遍认为, 人对外界的认知是从局部到全局的。视觉皮层的神经元就是局部接受信息的, 即只响应某些特定区域的刺激; 二是图像空间相关性, 对图像而言, 局部邻域内的像素联系较紧密, 距离较远的像素相关性则较弱; (2) 权值共享。

2. AutoEncoder 是一种尽可能重构输入信号的神经网络, 让整个网络的输出与输入相等。在此网络中, 隐含层则可以理解为用于记录数据的特征, 像主成分分析中获得的主成分那样, 因此这是一种典型的表示学习方法。训练过程如下图, 训练完成之后, 可以得到一个初始值 (权重的初始化), 以此训练每个 encoder, 每个 encoder 之后都是原样本的一种特征表示。

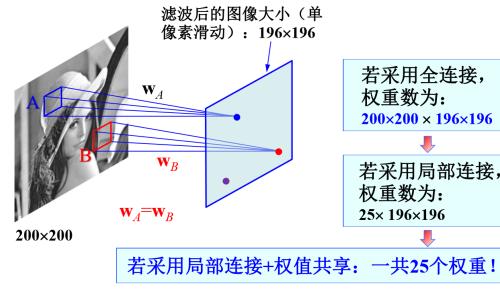
3. RNN: 将一般的前向神经网络做个压缩, 并延迟一步时间, 就得到 RNN 的基本结构, 按时间顺序全结点展开更容易看懂连接的方式。

LSTM 的数据流动如下:

考虑 RNN, LSTM 的不同之处: 由前一时刻的输出 (输出到下一时刻, 不是输出网络) $h_{t-1}$ , 和这一时刻的输入  $x_t$ , 如何得到这一时刻的输出 (输出到下一时刻, 不是输出网络) $h_t$ ?

1. RNN: 直接计算  $h_t = \tanh(b + Ux_t + Wh_{t-1})$ , 只需训练  $U, W$ ;

• 局部连接  
– 考虑 $5 \times 5$ 大小的滤波器且权值共享

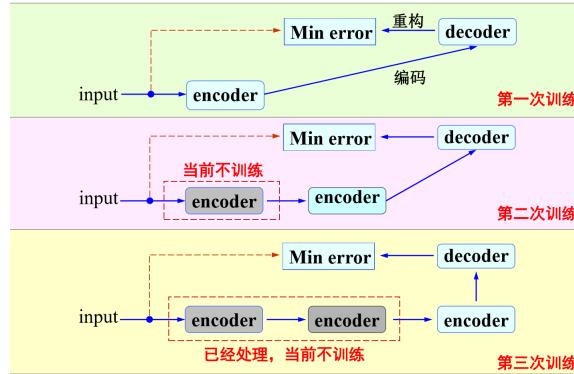


中国科学院大学  
University of Chinese Academy of Sciences

第40页

图 50: 参数情况

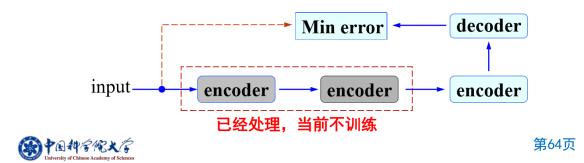
• 网络训练: (逐层静态进行)



(a) 1

• 网络训练:

- 对每个样本, 将第一层输出的 code 当成第二层的输入信号, **再次利用一个新的三层前向神经网络来最小化重构误差**, 得到第二层的权重参数 (获得第二个编码器); 同时得到样本在该层的code, 即原始号的第二个表达。
- 在训练当前层时, 其它层固定不动。完成当前编码和解码任务。前一次“编码”和“解码”均不考虑。
- 因此, 这一过程实质上是一个**静态的堆叠(stack)**过程
- 每次训练可以用BP算法对一个**三层前向网络**进行训练

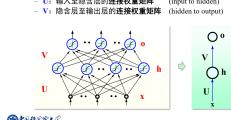


(b) 2

图 51: AutoEncoder 训练过程

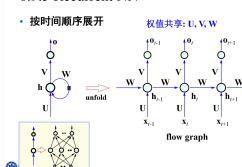
6.9.5 Recurrent NN

- 前向神经网络
  - 输入层:  $x$  (向量)
  - 隐藏层输出:  $h$  (向量)
  - 输出层:  $y$  (向量)
  - U: 输入至隐含层的连接权重矩阵 (input to hidden)
  - V: 隐含层至输出层的连接权重矩阵 (hidden to output)



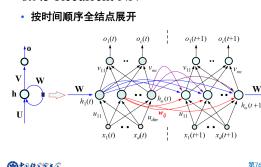
(a) 压缩前向神经网络

6.9.5 Recurrent NN



(b) 一步延迟等价形式

6.9.5 Recurrent NN



(c) 时间顺序节点全展开

图 52: RNN

## 6.9.6 LSTM

### • 结构描述——采用矩阵形式

遗忘门、输入门、输出门:

$$f_t = \text{sigmod}(\mathbf{b}_f + \mathbf{U}_f \mathbf{x}_t + \mathbf{W}_f \mathbf{h}_{t-1})$$

$$i_t = \text{sigmod}(\mathbf{b}_i + \mathbf{U}_i \mathbf{x}_t + \mathbf{W}_i \mathbf{h}_{t-1})$$

$$o_t = \text{sigmod}(\mathbf{b}_o + \mathbf{U}_o \mathbf{x}_t + \mathbf{W}_o \mathbf{h}_{t-1})$$

候选记忆（新贡献部分）：

$$c_t = \tanh(\mathbf{b} + \mathbf{U} \mathbf{x}_t + \mathbf{W} \mathbf{h}_{t-1})$$

Cell产生的新记忆:

$$s_t = f_t \otimes s_{t-1} + i_t \otimes c_t$$

Cell的输出:

$$h_t = o_t \otimes \tanh(s_t)$$

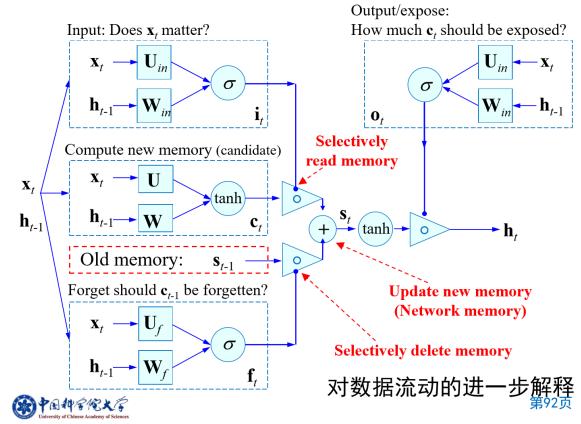
网络的输出:

$$z_t = \text{softmax}(\mathbf{V} \mathbf{h}_t + \mathbf{c})$$



第91页

(a) 1



(b) 2

图 53: LSTM

2. **LSTM:** 先计算  $c_t = \tanh(b + Ux_t + Wh_{t-1})$  作为候选记忆,  $s_{t-1}$  是上一时刻的记忆;

计算三个权重  $(0|1)f_t, i_t, o_t$ , 表示多大程度上, 公式如图;

计算这一时刻的记忆  $s_t$ , 公式如图;

最后计算这一时刻的输出  $h_t$ , 公式如图。需要训练  $U, W, U_f, W_f, U_{in}, W_{in}, U_o, W_o$ 。

## Part IV

## 面试经历

一、陌陌计算机视觉实习生岗位 (2019.11):

技术一面问题:

1.tensorflow 有什么特点? 手写 tensorflow 代码, 搭建卷积神经网络。

2. 写出神经网络中的 Batch Normalization 的公式, 在 tensorflow 中有哪些方式? (哪些参数)。

3. 神经网络的 BP 算法, 误差传到池化层的时候该怎么传? (均值池化和最大池化)

4. 介绍 ResNet 和 GoogleNet 的核心思想?

5. 图像边缘检测有哪些传统方法? 如果用深度学习工具来分割图像该怎么设计机制? (比如 Unet 网络)

6.Unet 网络模型先下采样, 再上采样, 在下采样的过程是准确的, 但是上采样的过程会使得图像变模糊, Unet 中有什么机制可以改善这一效果? (同级校对)

7. 数据结构: 二叉树的中序遍历 (递归与非递归), 输出数组滑窗中的最大值。如 [3,5,-1,3,2,-4,5], 窗口大小为 3, 输出 [5,5,3,3,5], 写完  $O(n)$  复杂度的之后要求优化 (使用队列等数据结构, 剑指 offer64 题)

8. 在 SLIC 超像素分割算法 (KMeans 聚类算法) 中有什么方法可以自适应地调整 k 值?

9. 介绍图像的双线性插值。

答案参考 (后续整理):

1.tensorflow 是静态图机制, Pytorch 是动态图机制。

静态图是指在图构建完成后, 在模型运行时无法进行修改。这里的“图”即为模型的意思, 一般一个图就是一个模型。这个图建好之后, 运行前需要 freeze, 然后进行一些优化, 例如图融合等, 最后启动 session 根据之前构建的静态图进行计算, 计算过程无法对静态图进行更改。

动态图和静态图对应, 在模型运行过程中可以对图进行修改。熟悉 PyTorch 的朋友应该了解, 因为

PyTorch 采用的就是动态图机制。不过一般情况，模型运行过程中也不需要对其进行修改。  
TensorFlow 静态图带来的一个弊端就是难 Debug。因为静态图在 freeze 后运行前会进行图优化，一些 operation 会被融合，所以有的 operation 会消失，无法在计算阶段提供断点和单步调试功能（图构建阶段能单步调试，但是只显示 tensor，用处不大）。当然，断点和单步调试功能也会影响程序运行的效率。而 PyTorch 采用了动态图，自然有 Debug 方面的优势，提供了单步调试的功能，可以在计算过程查看所有 tensor 的值，非常直观方便。（<https://blog.csdn.net/guanxs/article/details/90523905>）

2. Batch Normalization 方法为了使各层有适当的广度，强制性地调整激活值的分布，有以下优点：  
(1) 可以使学习快速进行（可以增大学习率）；如果每层的 scale 不一致，实际上每层需要的学习率是不一样的，同一层不同维度的 scale 往往也需要不同大小的学习率，通常需要使用最小的那个学习率才能保证损失函数有效下降，Batch Normalization 将每层、每维的 scale 保持一致，那么我们就可以直接使用较高的学习率进行优化。（<https://blog.csdn.net/u014365862/article/details/77159778>）  
(2) 不那么依赖初始值；  
(3) 抑制过拟合（降低 Dropout）；

Batch Norm 是使得数据分布的均值为 0，方差为 1 的正规化。写成公式如下：

$$\begin{aligned}\mu_B &= \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_B^2 &= \sum_{i=1}^m \frac{1}{m} (x_i - \mu_B)^2 \\ \hat{x}_i &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}\end{aligned}$$

正规化之后进行缩放和平移，写成公式如下：

$$y_i = \gamma \hat{x}_i + \beta$$

一开始  $\gamma = 1$ ,  $\beta = 0$ , 然后通过学习训练调整合适的值。

在 tensorflow 中：`tf.nn.batch_normalization(x, mean, var, beta, gamma, eps)`, 参数的意义跟上面公式一样。

3. 第三部分第四题讨论过这个问题。

4. ResNet 称为残差卷积神经网络。

ResNet 有的一个核心思想就是跳跃连接，这样做有两个非常重要的好处：(1)BP 算法中梯度从后往前传，每多传一层就会放缩一次误差，跳跃连接可以缓解梯度消失的情况；(2)这种跳跃连接使得目标函数变成一个关于参数的凸函数，传统的神经网络关于参数是非凸的，这样会很难求解全局最小解。

5.6. 介绍 Unet 网络模型：

7. 二叉树的中序遍历（递归和非递归）：

递归形式中，先序和后序只需更改核心三行的顺序即可；

非递归形式：递归本质上是在调用栈

```

1 #include<iostream>
2 #include<vector>
3 #include<stack>
4
5 using namespace std;
6
7 struct TreeNode{
8     int val;
9     TreeNode* left;
10    TreeNode* right;
11 };
12
13 //递归遍历
14 vector<int> midOrder(TreeNode *T)
15 {
16     vector<int> v;
17     if(T==NULL)
18     {
19         return v;
20     }
21     midOrder(T->left);
22     v.push_back(T->val); //此处改变顺序就得到先序和后序
23     midOrder(T->right);
24     return v;
25 }
26
27 //非递归遍历，中序
28 vector<int> Midorder(TreeNode *T)
29 {
30     vector<int> v;
31     stack<TreeNode*> s;
32     while(T||!s.empty())
33     {
34         while(T!=NULL)//如果当前的T不为空，则一直要到最左边
35         {
36             s.push(T);
37             T = T->left;
38         }
39         if (!s.empty())
40         {
41             T = s.top();
42             s.pop();
43             v.push_back(T->val);
44             T = T->right;
45         }
46     }
47     return v;
48 }
49
50 //非递归遍历，先序
51 vector<int> Midorder(TreeNode *T)
52 {
53     vector<int> v;
54     stack<TreeNode*> s;
55     while(T||!s.empty())
56     {
57         while(T!=NULL)//先输出当前的节点值，如果当前的T不为空，则一直要到最左边
58         {
59             v.push_back(T->val);
60             s.push(T);
61             T = T->left;
62         }
63         if (!s.empty())
64         {

```

```

65 | T = s.top();
66 | s.pop();
67 | T = T->right;
68 | }
69 |
70 | return v;
71 }

73 //非递归遍历，后序，只需将先序遍历的左右交换，同时将打印部分改为压入另一个栈，最后打印这个栈中的元素即
74 //得到后序
75 vector<int> Midorder(TreeNode *T)
76 {
77     vector<int> v;
78     stack<TreeNode*> s;
79     stack<TreeNode*> ss;
80     while(T || !s.empty())
81     {
82         while(T != NULL) //先输出当前的节点值，如果当前的T不为空，则一直要到最左边
83         {
84             ss.push(T); //按照根-右-左进栈
85             s.push(T);
86             T = T->right;
87         }
88         if (!s.empty())
89         {
90             T = s.top();
91             s.pop();
92             T = T->left;
93         }
94         while (!ss.empty())
95         {
96             T = ss.top();
97             ss.pop();
98             v.push_back(T->val);
99         }
100     return v;
101 }
```

8. 在网上看到一种方法“手腕法”， $SSE = \sum_{i=1}^k \sum_{p \in C_i} d(p, p_i)$ ，其中， $C_i$  是第  $i$  个簇， $p$  是  $C_i$  中的样本点， $m_i$  是  $C_i$  的质心 ( $C_i$  中所有样本的均值)， $SSE$  是所有样本的聚类误差，代表了聚类效果的好坏。手肘法的核心思想是：随着聚类数  $k$  的增大，样本划分会更加精细，每个簇的聚合程度会逐渐提高，那么误差平方和  $SSE$  自然会逐渐变小。并且，当  $k$  小于真实聚类数时，由于  $k$  的增大会大幅增加每个簇的聚合程度，故  $SSE$  的下降幅度会很大，而当  $k$  到达真实聚类数时，再增加  $k$  所得到的聚合程度回报会迅速变小，所以  $SSE$  的下降幅度会骤减，然后随着  $k$  值的继续增大而趋于平缓，也就是说  $SSE$  和  $k$  的关系图是一个手肘的形状，而这个肘部对应的  $k$  值就是数据的真实聚类数。当然，这也是该方法被称为手肘法的原因。<https://blog.csdn.net/xyisv/article/details/82430107>

#### 9. 图像的双线性插值：

双线性插值本质上是在两个方向分别进行单线性插值。[https://blog.csdn.net/qq\\_37577735/article/details/80041586](https://blog.csdn.net/qq_37577735/article/details/80041586))。

已知数据  $(x_1, y_1)$  与  $(x_2, y_2)$ , 要计算区间  $[x_1, x_2]$  内某一位置  $x$  在直线上的  $y$  值, 则有:

$$\begin{aligned} \frac{y - y_1}{x - x_1} &= \frac{y_2 - y}{x_2 - x} \\ (x_2 - x)(y - y_1) &= (y_2 - y)(x - x_1) \\ (x_2 - x)y - (x_2 - x)y_1 &= (x - x_1)y_2 - (x - x_1)y \\ (x_2 - x_1)y &= (x_2 - x)y_1 + (x - x_1)y_2 \\ y &= \frac{x_2 - x}{x_2 - x_1}y_1 + \frac{x - x_1}{x_2 - x_1}y_2 \end{aligned}$$

上面的公式仔细看就是用  $x$  和  $x_1, x_2$  的距离作为一个权重, 用于  $y_1$  和  $y_2$  的加权, 双线性本质上是在图像的两个方向上做线性插值, 核心思想是在两个方向上分别进行一次线性插值。假设想得到未知函数  $f$  在点  $P = (x, y)$  的值, 已知最近的四个点的坐标为  $Q_{11}(x_1, y_1), Q_{12}(x_1, y_2), Q_{21}(x_2, y_1), Q_{22}(x_2, y_2)$ , 如下图: 首先进行  $x$  方向的插值, 得到:

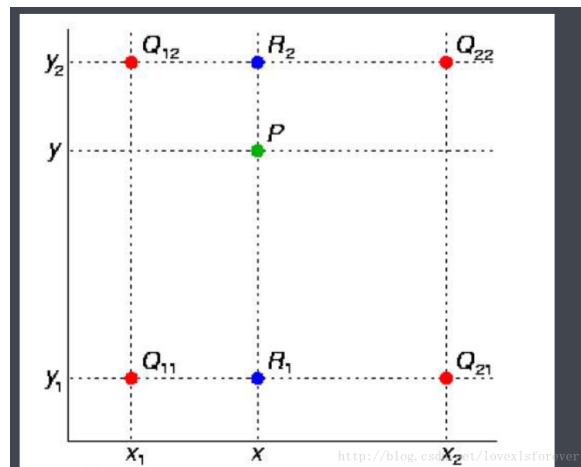


图 54: 双线性插值示意图

$$\begin{aligned} f(R_1) &= \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \\ f(R_2) &= \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \end{aligned}$$

然后在  $y$  方向进行线性插值, 得到:

$$f(P) = \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2)$$

由于图像的双线性插值只会用相邻的 4 个点, 因此上述公式的分母都是 1, 下面给出一个计算的代码:

```

1 import cv2
2 import numpy as np
3 import time
4 import os
5
6 def resize(src, new_size):
7     dst_w, dst_h = new_size # 目标图像宽高

```

```

src_h, src_w = src.shape[:2] # 源图像宽高
9 if src_h == dst_h and src_w == dst_w:
10    return src.copy()
11 scale_x = float(src_w) / dst_w # x缩放比例
12 scale_y = float(src_h) / dst_h # y缩放比例
13
# 遍历目标图像，插值
14 dst = np.zeros((dst_h, dst_w, 3), dtype=np.uint8)
15 for n in range(3): # 对channel循环
16    for dst_y in range(dst_h): # 对height循环
17        for dst_x in range(dst_w): # 对width循环
18            # 目标在源上的坐标
19            src_x = (dst_x + 0.5) * scale_x - 0.5
20            src_y = (dst_y + 0.5) * scale_y - 0.5
21            # 计算在源图上四个近邻点的位置
22            src_x_0 = int(np.floor(src_x))
23            src_y_0 = int(np.floor(src_y))
24            src_x_1 = min(src_x_0 + 1, src_w - 1)
25            src_y_1 = min(src_y_0 + 1, src_h - 1)
26
# 双线性插值
27 value0 = (src_x_1 - src_x) * src[src_y_0, src_x_0, n] + (src_x - src_x_0) * src[src_y_0, src_x_1, n]
28 value1 = (src_x_1 - src_x) * src[src_y_1, src_x_0, n] + (src_x - src_x_0) * src[src_y_1, src_x_1, n]
29 dst[dst_y, dst_x, n] = int((src_y_1 - src_y) * value0 + (src_y - src_y_0) * value1)
30
31 return dst
32
33 if __name__ == '__main__':
34     img_in = cv2.imread('./test/3.jpg')
35     img_out = cv2.resize(img_in, (600, 600))
36     cv2.imwrite('./.jpg', img_out)
37
38 time_start = time.time()
39 for img in os.listdir('./test'):
40     img_in = cv2.imread(os.path.join('./test', img))
41     img_out = cv2.resize(img_in, (600, 600))
42     cv2.imwrite('./test_preprocess_chazhi', img_out)
43
44 time_end = time.time()
45 time_c = time_end - time_start
46 print('time_c:{} s'.format(time_c))

```

代码中最重要的是目标图像和原图像坐标的对齐，这一步是为了插值出来的目标图像和原图像有个几何中心的对齐，使得插值的结果更加均匀地分布在原图像上。假设原图尺寸为  $(m_1, n_1)$ ，插值图像尺寸为  $(m_2, n_2)$ ，则  $x, y$  方向上的放缩因子分别为  $scale_x = \frac{m_1}{m_2}$ ,  $scale_y = \frac{n_1}{n_2}$ 。如果以原点  $(0, 0)$  对齐：要插值的目标图像上的像素坐标为  $(dst_x, dst_y)$ ，对应到原图上像素坐标为  $(src_x, src_y) = (dst_x * scale_x, dst_y * scale_y)$ ，以这个坐标周围四个点来进行插值会导致插值使用的像素偏向于原图像的右下方或左上方，而不是均匀分布整个图像。如  $(3, 3)$  的图像插值为  $(5, 5)$ ， $x, y$  方向上的比例因子都是  $\frac{3}{5}$ ，在插值中心元素  $(2, 2)$  时，对应的原图像上是坐标  $(2 * \frac{3}{5}, 2 * \frac{3}{5}) = (1.667, 1.667)$ ，使用这个坐标周边四个像素值来插值会导致插值的图像像素集中在原图像的右下方；同理，如果插值的图像比原图小的话，会导致像素集中在左上方。所以这里采用了一个对齐公式：

$$src_x = dst_x * scale_x + \frac{1}{2}(scale_x - 1)$$

$$src_y = dst_y * scale_y + \frac{1}{2}(scale_y - 1)$$

用这个对齐公式去计算在原图上的像素位置，然后使用这个像素周围四个像素的位置来插值新的图像。此外再注意插值公式的分母都为 1，即可理解上面这段代码。

技术二面问题：

1. 介绍线性子空间； 2. 介绍共轭梯度算法，联系梯度下降和牛顿法；
3. 卷积神经网络中一般都有哪些层？简要介绍 BN 层；
4. 介绍极大似然估计方法流程；
5. 操作系统（或计算机组成原理）中内存的 LRU 算法；
6. 学过哪些深度学习模型和机器学习算法；
7. 手写代码，将一个数组逆时针旋转 90°，要求不能开内存；（通过中间变量来赋值，解答见 leetcode 第 48 题）

## 二、腾讯微信应用开发岗位 (2021.03.15)

技术一面：主要针对项目来问，没有问基础算法问题。

1. 在海量视频数据中，重复的较多（重复指内容重复，但会有些许不同，比如 logo、分辨率等），去重的技术路线是怎样的？

我回答在图片去重领域可以用预训练好的深度学习网络（比如 ImageNet 上训好的 ResNet-50），将所有图片通过该网络得到每个图片的向量特征，然后计算向量间的均方损失设置阈值去重，面试官说差不多的思路，可以看下高维向量搜索，类似于以图搜图；

2. 在二分类或者多标签分类中，如何解决某个类别样本过少的问题？

我当时只回答了一个加大样本少的类别的损失函数权重，面试官提示使用 focal loss，focal loss 简要介绍：

基本交叉熵损失函数为：

$$L = -y \log y' - (1 - y) \log(1 - y') = \begin{cases} -\log y' & y = 1 \\ -\log(1 - y') & y = 0 \end{cases} \quad (4)$$

(1)focal loss 首先在原有的基础上加了一个  $\gamma$  因子，其中  $\gamma > 0$  减少易分类样本的损失，使得模型更关注于困难的、错分的样本。

$$L_{fl} = \begin{cases} -(1 - y')^\gamma \log y' & y = 1 \\ -y'^\gamma \log(1 - y') & y = 0 \end{cases} \quad (5)$$

例如  $\gamma$  为 2，对于正类样本而言，预测结果为 0.95 肯定是简单样本，所以  $(1 - 0.95)^2$  就会很小，这时损失函数值就变得更小；而预测概率为 0.3 的样本其损失相对很大。对于负类样本而言同样，预测 0.1 的结果应当远比预测 0.7 的样本损失值要小得多。对于预测概率为 0.5 时，损失只减少了 0.25 倍，所以更加关注于这种难以区分的样本。这样减少了简单样本的影响，大量预测概率很小的样本叠加起来后的效应才可能比较有效。

(2)focal loss 随后加入平衡因子  $\alpha$ ，用来平衡正负样本本身的比例不均。

$$L_{fl} = \begin{cases} -\alpha(1 - y')^\gamma \log y' & y = 1 \\ -(1 - \alpha)y'^\gamma \log(1 - y') & y = 0 \end{cases} \quad (6)$$

比如  $\alpha$  取 0.25，即正样本要比负样本占比小，可能是由于负样本比较少，让负样本的损失函数比例增大。

## 三、腾讯新闻应用研究岗位 (2021.03.16)

技术一面：聊了四十分钟项目，然后一道算法题。

给定一个 0-1 二维矩阵，1 代表 water，0 代表 land，现在给该二维矩阵赋高度值，有以下准则：

- (1) 所有高度值非负；
- (2)water 的地方高度值为 0；
- (3) 相邻的单元格高度差不超过 1。

You are given an integer matrix `isWater` of size  $n \times n$ , that represents a map of **land** and **water** cells.

- If `isWater[i][j] == 0`, cell  $(i, j)$  is a **land** cell.
- If `isWater[i][j] == 1`, cell  $(i, j)$  is a **water** cell.

You must assign each cell a height in a way that follows these rules:

- The height of each cell must be non-negative.
- If the cell is a **water** cell, its height must be  $0$ .
- Any two adjacent cells must have an absolute height difference of **at most**  $1$ . A cell is adjacent to another cell if the former is directly north, east, south or west of the latter (i.e., their sides are touching).

Find an assignment of heights such that the maximum height in the matrix is **maximized**.

Return an integer matrix `height` of size  $n \times n$  where `height[i][j]` is cell  $(i, j)$ 's height. If there are multiple solutions, return **any** of them.

**Example 1:**

1	0
2	1

Input: `isWater = [[0,1],[0,0]]`

Output: `[[1,0],[2,1]]`

Explanation: The image shows the assigned heights of each cell.

The blue cell is the water cell, and the green cells are the land cells.

**Example 2:**

1	1	0
0	1	1
1	2	2

Input: `isWater = [[0,0,1],[1,0,0],[0,0,0]]`

Output: `[[1,1,0],[0,1,1],[1,2,2]]`

Explanation: A height of  $2$  is the maximum possible height of any assignment.

Any height assignment that has a maximum height of  $2$  while still meeting the rules will also be accepted.

(a) 1

(b) 2

图 55: 算法题

通过给该矩阵赋值，返回高度值最高的矩阵。

思路：

- (1) 遍历每个元素，将 0 的地方进队列，同时在矩阵中将 1 的地方设置为 0；
- (2) 遍历队列，进行上下左右四个元素广度优先遍历，如果该值没有被访问过，则进队列，同时给该位置赋值；

当时写的代码如下：

```

1 #include<iostream>
2 #include<vector>
3 #include<queue>
4
5 using namespace std;
6
7 class Solution
8 {
9 public:
10     vector<vector<int>> func(vector<vector<int>> matrix)
11     {
12         if(matrix==nullptr)
13             return nullptr;
14         int m = matrix.size();
15         int n = matrix[0].size();
16         //vector<vector<int>> flag(m,n); //有问题
17         vector<vector<int>> flag(m, vector<int>(n, 0)); //定义m*n二维数组
18         for(int i=0;i<m; i++)
19         {
20             for(int j=0;j<n; j++)
21                 if(matrix[i][j]==1)
22                     matrix[i][j] = 0;
23                 else
24                     q.push({i,j});
25         }
26         //int height = 0;
27         while(!q.empty())
28         {
29             vector<int> tmp = q.front();

```

```

31     q.pop();
32     int i = tmp[0];
33     int j = tmp[1];
34     int height = matrix[i][j] + 1;
35     flag[i][j] = 1;
36     if ((i > 0) && (j - 1 > 0) && flag[i][j - 1] == 0)
37     {
38         q.push({i, j - 1});
39         matrix[i][j - 1] = height;
40     }
41     if ((i > 0) && (j + 1 < n) && flag[i][j + 1] == 0)
42     {
43         q.push({i, j + 1});
44         matrix[i][j + 1] = height;
45     }
46     if ((i - 1 < m) && (j > 0) && flag[i - 1][j] == 0)
47     {
48         q.push({i - 1, j});
49         matrix[i - 1][j] = height;
50     }
51     if ((i + 1) < m && (j < n) && flag[i + 1][j] == 0)
52     {
53         q.push({i + 1, j});
54         matrix[i + 1][j] = height;
55     }
56     return matrix;
57 }
58 private:
59     queue<vector<int>> q;
60 };

```

#### 四、腾讯 AILab 游戏中心应用研究岗位 (2021.03.17)

技术一面：面试官是做 3D 姿态估计的，聊了半个多小时项目，然后让用 numpy 实现一个带 mask 的 softmax，主要考察两点：

- (1) 如何解决 softmax 函数中 e 的指数过大产生溢出的问题？当  $x$  的值非常大时，如  $x = [1000, 2000, 3000]$ ，此时  $e^{2000}$  会溢出。
- (2) 带 mask 的 softmax 和一般的 softmax 不一样，比如输入为  $x$ ，相应的 mask 有 0 或 1,0 的作用是要屏蔽该位置处  $x$  的值，只有 mask 中 1 对应的索引位置才参加 softmax 的计算，如  $x = [1, 1, 1, 0]$ ,  $mask = [1, 1, 0, 0]$ ，则经过 softmax 之后输出为  $[1/2, 1/2, 0, 0]$ ，需要注意直接用  $x * mask$  再求 softmax，是不对的，因为  $e^0 = 1$ ，这样 0 的位置也参与运算了。

```

#####
1 #####numpy版本#####
2 import numpy as np
3
4 x = np.array([[1, 1, 0, 0], [1, 1, 4, 1], [1, 0, 2000, 1]]) # 2000是为了测试防溢出机制问题
5 mask = np.array([[1, 1, 0, 0], [1, 0, 1, 0], [0, 0, 1, 1]])
6 # x_trans = np.transpose(x) # x的转置，这里使用x.reshape(x.shape[1], x.shape[0])是不行的
7 # print(x_trans)
8
9
10 # softmax的一般写法
11 x_exp = np.exp(x) # 逐个元素计算
12 x_exp = x_exp*(mask==1) # 加入mask信息
13 # x_sum = np.sum(x_exp, axis=0) # 按列加
14 x_sum = np.sum(x_exp, axis=1) # 按行加
15 # print(x_sum.reshape(x.shape[0], 1))
16 output = x_exp/x_sum.reshape(x.shape[0], 1)
17 print(output)

```

```

18
20 x_max = np.max(x, axis=1) # 按行取最大值
x_max = x - x_max.reshape(x.shape[0], 1) # 增加防止溢出机制
22
24 x_exp = np.exp(x_max) # 逐个元素计算
x_exp = x_exp*(mask==1) # 加入mask信息
x_sum = np.sum(x_exp, axis=1) # 按行加
# print(x_sum.reshape(x.shape[0], 1))
output = x_exp/x_sum.reshape(x.shape[0], 1)
28 print(output)

30 ######Tensor版本的#####
32 import torch

34 x = torch.Tensor([[1,1,0,0],[1,1,4,1],[1,0,2000,1]])
mask = torch.Tensor([[1,1,0,0],[1,0,1,0],[0,0,1,1]])
36 # x_trans = x.t() # x的转置, 这里使用x.view(x.size(1),x.size(0))是不行的
# print(x_trans)

38 x_max = torch.max(x, dim=1)[0] # 第0个元素是dim=1上的最大值, 第1个元素是最大值对应的索引
40 x_max = x - x_max.view(x.size(0), 1) # 或者x = x - x_max.reshape(x.size(0), 1)

42 x_exp = torch.exp(x_max)
# print(type(mask==1))
44 x_exp = x_exp*(mask==1).float() # 这里需要转换成float()
x_sum = torch.sum(x_exp, dim=1)
46 # print(x_sum.view(x.size(0), 1))
output = x_exp/x_sum.view(x.size(0), 1)
48 print(output)

50 ######torch自带softmax函数版本#####
52 import torch.nn as nn

54 x = torch.Tensor([[1,1,0,0],[1,1,4,1],[1,0,2000,1]])
softmax = nn.Softmax(dim=1)
56 # print(type(softmax)) # <class 'torch.nn.modules.activation.Softmax'>
output = softmax(x) # output = nn.Softmax(x, dim=1)这样直接计算是不对的, 需要先实例化一个类对象
58 print(output)

```

## 五、腾讯 PCG 事业群应用研究岗位 (2021.03.22)

技术一面：聊了二十多分钟项目，然后问了一些数学问题和机器学习问题，以及几道算法问题：

### 1. 机器学习基础问题

#### (1) 过拟合是什么意思？有哪些解决方法？

过拟合就是模型在验证集上的性能和训练集上的性能相差较大，具体解决方法有：(1) 使用轻量级网络，面对小样本的时候，使用重网络会导致过拟合；(2) 一系列数据增强操作，如旋转，平移等，为了增强模型鲁棒性和泛化性；(3) 使用 BN 层，让网络参数浮动幅度小一点；(4) 使用 Dropout；(5) 正则化，给损失函数添加一个正则项，是参数的 1 范数、2 范数或者 p 范数。

#### (2) 如何解决样本不均衡问题？

直观的方法是增加小样本类别的样本数以及增加少类别的损失损失权重，具体实施方法参考 focal loss。

### 2. 数学问题

一个实数轴上有  $n$  个点， $x_1, x_2, \dots, x_n$ ，现在在数轴上再取一个点  $x$ ，现在定义两个点之间的距离分别为均方距离和绝对值，求  $x$  距离所有点距离之和最小值点。

(1) 当距离定义为均方损失时，最优化目标函数为  $f(x) = \sum_{i=1}^n (x_i - x)^2$ ，该函数是一个凸函数，局部

极值就是全局极值,  $f(x)$  对  $x$  求一阶导并令结果为 0, 直接得出  $x = \frac{1}{n} \sum_{i=1}^n x_i$ ;

(2) 当距离定义为绝对值时, 最优化目标为  $f(x) = \sum_{i=1}^n |x_i - x|$ , 是不可导的函数, 可以从简单情况推导: 当  $n = 2$  时,  $x$  在  $[x_1, x_2]$  区间中任何一个地方, 目标函数的值都是  $|x_1 - x_2|$ , 当  $n = 3$  时, 假设  $x_1 < x_2 < x_3$ , 可以想得到, 当  $x = x_2$  时, 目标函数的值为  $x_1 - x_3$ , 其余无论在哪个位置都会多出  $x$  和  $x_2$  之间的绝对值距离; 依次类推, 当  $n$  为奇数时, 最优点  $x$  就是中间的点, 当  $n$  为偶数时, 最中间两个点之间的区间任何一个值都可以。

### 3. 编程问题

(1) 列举一些排序算法, 并说出算法复杂度;

(2) 简要介绍归并排序和快速排序的流程;

(3) 如何求解一个数组的中位数?

第一种方法是先排序, 然后直接求  $\frac{n}{2}$  位置的点; 优化一下就是求一个数组第  $n/2$  大的数, 是快速排序的一个应用。

(4) 岛屿问题, 求解岛屿上连通域的个数。

## 六、蚂蚁金服

技术一面 (2021.03.29): 聊了四十分钟项目, 然后针对项目中的问题来提问 (比如介绍下 MobileNetV1 中的深度可分离卷积), 最后一道算法题:

```
// 题目: 求以下两个序列的最长公共子序列(LCS)长度
1 //s1 = '13235412467'
2 //s2 = '2313715626'
3
4 #include<iostream>
5 #include<string>
6 #include<vector>
7
8 using namespace std;
9
10 class CommonSequence
11 {
12 public:
13     int func(string str1, string str2)
14     {
15         int m = str1.size();
16         int n = str2.size();
17         vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
18         for(int i=1;i<m+1;i++)
19         {
20             for(int j=1;j<n+1;j++)
21             {
22                 if(str1[i]==str2[j])
23                     dp[i][j] = dp[i-1][j-1] + 1;
24                 else
25                     dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
26             }
27         }
28         return dp[m][n];
29     }
30 };
31
32
33 int main()
34 {
35     string s1 = "13235412467";
36     string s2 = "2313715626";
37     CommonSequence cs;
```

```

40     int length = cs.func(s1, s2);
41     cout<<length<<endl;
42     return 0;
}

```

技术二面 (2021.04.04)

主要针对项目问一些 open 的问题，没有算法题。

七、腾讯北京部门 (2021.03.30)

技术一面：先聊了一会儿项目，针对项目问了一些简单的问题，随后问机器学习等基础问题。

1. 机器学习问题：

- (1) 介绍下 GBDT 的基本原理；(面试官是做 nlp 的，所以会问些 nlp 的东西，这个我没答出来)
- (2) 逻辑回归和线性回归的差别；(主要想问分类和回归的差别，在于损失函数不一样，一个用均方损失，一个用交叉熵)

2. 深度学习问题：

- (1) 介绍下 ResNet 和 inceptionv3 的区别，并说明两个网络为什么要这样设计；(inceptionv3 不太记得了，只说了 Inceptionv1 和 ResNet 的区别)
- (2) 卷积分解有什么好处？比如  $5 \times 5$  拆分成两个  $3 \times 3$ 。(1. 拆分卷积可以减少参数， $5 \times 5 = 25 > 18 = 3 \times 3 + 3 \times 3$ ; 2. 多层卷积可以使用多个非线性激活函数，提高模型表达能力；3. 经过实验，两个  $3 \times 3$  卷积效果和一个  $5 \times 5$  卷积效果差不多)
- (3)  $1 \times 1$  卷积的作用？( $1 \times 1$  卷积可以更改通道的数量，达到降维的目的，同时可以减少参数)
- (4) 简单介绍可变形卷积和空洞卷积？有什么好处？

传统卷积对  $3 \times 3$  或  $5 \times 5$  等方格范围进行加权求和，可变形卷积计算加权和的范围不固定，网格是可变形的，因为每个网格点都可以通过一个可学习的偏移量移动。卷积作用于这些移动的网格点上，因此称为可变形卷积，具体图示如下：

如图所示，有一个分支路专门用来学习偏移量，然后将学习到的偏移量用到主支路的传统卷积上，形

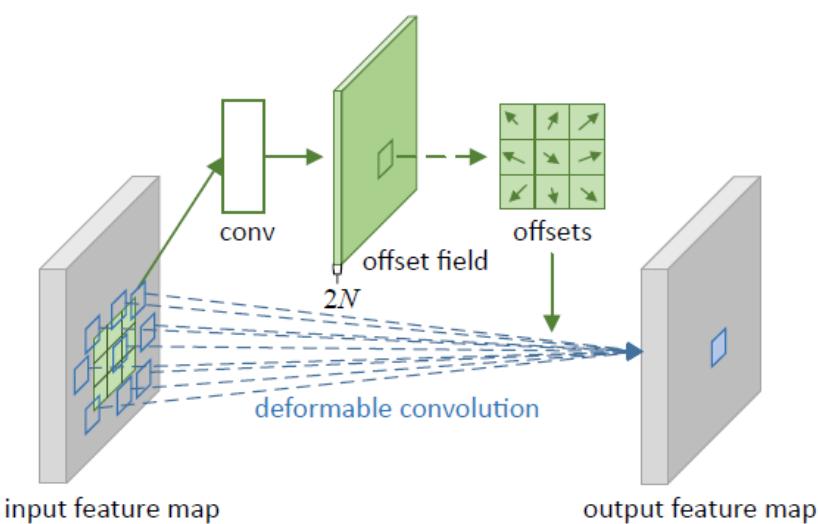


图 56: 可变形卷积

成可变形卷积，作用在于可以使模型自己去学习要卷积的区域，类似于 self-attention。原文章是《Deformable Convolutional Network》。

空洞卷积是在传统卷积的基础上，给卷积核注入空洞，pytorch 框架中的卷积函数 nn.Conv2d() 里面有一个 dilation 参数就是卷积核的间隔数量，传统卷积 dilation=1。空洞卷积好处在于不增加参数量的前提下扩大感受野，捕捉更大的上下文信息。

个人认为空洞卷积是可变形卷积的一种简单实现方式，是手工设计的一种扩大感受野方式的卷积，可变形卷积让网络自己学习格点的偏移量。

(5) 列举下有哪些损失函数？(回答了均方损失，交叉熵损失，BCEloss，最后问知不知道 KL 散度)

(以下内容参考：<https://zhuanlan.zhihu.com/p/74075915>)

简单介绍 KL 散度：

目前分类损失函数为何多用交叉熵，而不是 KL 散度。

首先损失函数的功能是通过样本来计算模型分布与目标分布间的差异，在分布差异计算中，KL 散度是最合适的。但在实际中，某一事件的标签是已知不变的（例如我们设置猫的 label 为 1，那么所有关于猫的样本都要标记为 1），即目标分布的熵为常数。而根据下面 KL 公式可以看到，KL 散度 - 目标分布熵 = 交叉熵。所以我们不用计算 KL 散度，只需要计算交叉熵就可以得到模型分布与目标分布的损失值。从上面介绍，知道了模型分布与目标分布差异可用交叉熵代替 KL 散度的条件是目标分布为常数。如果目标分布是有变化的（如同为猫的样本，不同的样本，其值也会有差异），那么就不能使用交叉熵，例如蒸馏模型的损失函数就是 KL 散度，因为蒸馏模型的目标分布也是一个模型，该模型针对同类别的不同样本，会给出不同的预测值（如两张猫的图片 a 和 b，目标模型对 a 预测为猫的值是 0.6，对 b 预测为猫的值是 0.8）。

1) 信息量

假设  $X$  是一个离散型随机变量，概率分布函数为  $p(x) = P(X = x)$ ，则定义事件  $X = x_0$  的信息量为  $I(x_0) = -\log(p(x_0))$ 。

2) 熵

当一个事件发生的概率为  $p(x)$ ，则它的信息量是  $-\log(p(x))$ ，计算信息量的期望： $H(X) = -\sum_{i=1}^n p(x_i)\log(p(x_i))$ 。

3) 相对熵

相对熵又称为 KL 散度，如果对于同一个随机变量有两个单独的概率分布  $p(x)$  和  $q(x)$ ，可以使用 KL 散度 (Kullback-Leibler (KL) divergence) 来衡量这两个分布的差异。在机器学习中， $P$  往往用来表示样本的真实分布， $Q$  用来表示模型所预测的分布，那么 KL 散度可以计算两个分布的差异，也就是 Loss 损失值：

$$D_{KL}(p||q) = \sum_{i=1}^n p(x_i)\log(\frac{p(x_i)}{q(x_i)})$$

4) 交叉熵

将 KL 散度公式展开：

$$D_{KL}(p||q) = \sum_{i=1}^n p(x_i)\log(\frac{p(x_i)}{q(x_i)}) = \sum_{i=1}^n p(x_i)\log(p(x_i)) - \sum_{i=1}^n p(x_i)\log(q(x_i)) = -H(p(x)) + [-\sum_{i=1}^n p(x_i)\log(q(x_i))]$$

等式的前一部分是概率分布  $p(x)$  的熵，等式的后一部分就是交叉熵，所以有：KL 散度 - 目标分布熵 = 交叉熵。

5) 相对熵的性质

首先相对熵关于  $p, q$  是不对称的，其次相对熵非负：

$$\text{欲证 } D_{KL}(p||q) = \sum_{i=1}^n p(x_i)\log(\frac{p(x_i)}{q(x_i)}) \geq 0$$

$$\text{即证 } \sum_{i=1}^n p(x_i)\log(\frac{q(x_i)}{p(x_i)}) \leq 0$$

又  $\ln(x) \leq x - 1$ ，当且仅当  $x = 1$  等号成立，故有：

$$\sum_{i=1}^n p(x_i)\log(\frac{q(x_i)}{p(x_i)}) \leq \sum_{i=1}^n p(x_i)(\frac{q(x_i)}{p(x_i)} - 1) = \sum_{i=1}^n [p(x_i) - q(x_i)] = \sum_{i=1}^n p(x_i) - \sum_{i=1}^n q(x_i) = 1 - 1 = 0$$

当且仅当  $\frac{q(x_i)}{p(x_i)} = 1$  时，等式成立。

简单介绍 JS 散度：

JS 散度基于 KL 散度，解决了 KL 散度关于两个分布不对称的问题，定义如下：

$$JS(p||q) = \frac{1}{2}KL(p||\frac{p+q}{2}) + \frac{1}{2}KL(q||\frac{p+q}{2})$$

### 3. 算法题：

给定一个 n 个数的数组，再给一个 m，每次从数组里面抽出第 m 个数，求最后剩下的数。比如 1,2,3,4,5,6,7,m=3，从 1 开始数，数三个数剔除，把 3 剔除，随后接着数三个数剔除，把 6 剔除，一次类推，求最后剩余的那个数的索引。

假设 n 个数的索引为 0, 1, 2, ..., n - 1，最后留的那个数的索引为  $f(n, m)$ 。第一次数到 m-1 时去除该数，则数组变为 0, 1, 2, ..., m - 2, m, ..., n - 1，此时从 m 开始数，相当于求解一个子问题  $f(n - 1, m)$ ，而该问题的序列为 m, m + 1, ..., n - 1, 0, 1, ..., m - 1。容易得到两个数组索引的关系： $f(m, n) = f(n - 1, m) + m - n$ 。由于  $f(n, m)$  问题对应的数组长度为 n，所以可以简化成  $f(m, n) = [f(n - 1, m) + m] \% n$ ，动态规划问题，代码如下：

```
1 class Solution {
2     public:
3         int LastRemaining_Solution(unsigned int n, unsigned int m)
4     {
5         if (n==0)
6             return -1;
7         if (n==1)
8             return 0;
9         else
10            return (LastRemaining_Solution(n-1,m)+m)%n;
11     }
12 }
```

## 技术二面 (2021.04.09)

主要针对项目问一些 open 的问题，没有算法题。

### 八、字节跳动 Data-EDU(2021.04.08)

1. 首先聊项目，在项目中提问一些问题；

如何解决长尾分布？(说了进行尾部类别的数据增强、重采样，头部类别的欠采样和 focal loss，还有其他方法，参考链接 <https://zhuanlan.zhihu.com/p/158638078>)

2. 深度学习基础问题：

(1) 简单介绍下 ResNet-50 的结构，并说明为什么 ResNet 系列网络为什么在第三个 stage 设计多个 block？

个人理解：主要从显存角度考虑，前两个 stage 对应的 feature map spatial 大，如果增加前两个 stage block 数量的话会增加较多显存；而最后一个 stage 虽然 feature map spatial 小，但是 channel 太多，增加该 stage block 数量同样会增大显存；所以 ResNet50 的 block 数量依次为 3,4,6,3，中间两个 block 会相对多一点，第三个比第二个多两个 block。

(2) 简要介绍 MobileNetV2 的结构，深度可分离卷积的卷积参数计算，相较于传统卷积参数减少了多少？MobileNetV2 为什么要设计成逆残差结构？比如中间通道数乘 6 倍。

假设输入的特征是  $C * H * W$ ，输出特征是  $C' * H' * W'$ ，则卷积的尺寸为  $3 * 3$ ，则：

正常卷积过程的参数量为  $C' * C * 3 * 3$ ( $C'$  个卷积核，每个卷积核是  $C$  通道，每个通道为  $3 * 3$  大小)。深度可分离卷积分为两个过程 depthwise 和 pointwise 卷积，depthwise 阶段每个卷积核只和输入特征的一个通道做卷积，假设输出有  $C'$  个通道，输入分为  $G$  组，则有  $\frac{C'}{G}$  个卷积，每个卷积核的尺寸为  $3 * 3 * C$ ，通道又分为  $G$  组，会和输入特征  $G$  组中每一组做卷积，参数为  $\frac{C'}{G} * C * 3 * 3 = C' * 3 * 3$ ；pointwise 阶段是一个正常卷积过程，不过将卷积核的空间尺寸设置为  $1 * 1$ ，所以参数量为  $C_1 * C_2 * 1 * 1$ (其中  $C_1$  是输入特征通道数， $C_2$  是输出特征通道数。)

个人理解：由于 ResNet 没有使用 depthwise conv，在进入 pointwise conv 之前的特征通道数是比较的，所以残差模块中使用了 0.25 倍的降维。而 MobileNet v2 由于有 depthwise conv，通道数相对较少，并且通道之间没有融合，所以残差中使用了 6 倍的升维来增加通道数。

### 3. 算法题：

给定一个长度为  $n$  的全 0 数组 vector $0,0,0,0,0,0,0$ ，第一次遍历每个元素都变化，变成  $1,1,1,1,1,1,1$ ；第二次遍历每两个元素变化一次，变成  $1,0,1,0,1,0,1$ ；第三次遍历每三个元素变化一次，变为  $1,0,0,0,1,1,1$ ；以此类推，问  $n$  次之后，数组中还有多少个 1？

(参考 leetcode319 题灯泡开关)

技术二面 (2021.04.12)

1. 问项目，同时提一些问题：如介绍一下 HRNet 以及姿态估计目前有哪些可以创新的方向；

2. 机器学习和深度学习基础问题

(1) 模型过拟合如何解决？

(2) 简要介绍 BN 层如何实现？

(3) 介绍交叉熵函数？

3. 数学问题

(1) 如何求一个浮点数的平方根 (二分法迭代)；

(2) 一个数组，将数组中的数分为两部分，如何分才使得两部分均值相差最大？

首先回答了可以排序，一边放大数一边放小数；然后思考一会具体技术路线是进行枚举，对  $n$  个数进行枚举划分，如两部分分别分为  $\{1, n - 1\}, \{2, n - 2\}, \dots, \{n - 1, 1\}$ ，随后计算哪种划分方式均值差最大；最后面试官问如何证明这种技术路线是最优的，我回答可以模拟证明：如两堆数分别为  $x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m$ ，假设有  $x_1 \leq x_2 \leq \dots \leq x_n \leq y_1 \leq y_2 \leq \dots \leq y_m$ ，现在添加些许扰动，某个  $x_i$  和  $y_j$  互换位置，可以证明互换位置之后差会比之前的大。

4. 算法题

```

代码考核 在线演示 设备信息
任务 题1(第1轮) 题2
面试官提出的问题将出现在这里。
C++(clang++11) 重置
1 #include <iostream>
2 #include<vector>
3 using namespace std;
4
5 //dp[i][j] 表示从第一列第一个节点到第i行第j列的最短路径的长度
6 //dp[1][j] = min(dp[0][j-1]+abs(v[0][j-1]-v[1][j]),
7 //dp[1][j-1]+abs(v[1][j-1]-v[1][j]),
8 //dp[2][j-1]+abs(v[2][j-1]-v[1][j]));
9
10 //计算第一列第一个节点到第i行第j列的节点(i,j)的最短路径长度
11 class solution{
12 public:
13     int func(vector<vector<int>>& v)
14     {
15         int n = v[0].size(); //列数
16         vector<vector<int>> dp(3, vector<int>(n-1, 0));
17         //初始化第一列
18         dp[0][0] = abs(v[0][0]-v[0][1]);
19         dp[1][0] = abs(v[0][0]-v[1][1]);
20         dp[2][0] = abs(v[0][0]-v[2][1]);
21         for(int i=0;i<n-1;i++)
22             for(int j=0;j<n-1;j++)
23                 dp[1][j] = min3(dp[0][j-1]+abs(v[0][j]-v[1][j+1]),
24                               dp[1][j-1]+abs(v[1][j]-v[1][j+1]),
25                               dp[2][j-1]+abs(v[2][j]-v[1][j+1]));
26         return min3(dp[0][n-2], dp[1][n-2], dp[2][n-2]);
27     }
28 private:
29     int min3(int a, int b, int c)
30     {
31         return min(min(a,b),c);
32     }

```

图 57: 算法题

```

#include <iostream>
#include<vector>
using namespace std;
4

```

```

6 //dp[i][j] 表示从第一列第一个节点到第i行第j列的最短路径的长度
7 //dp[i][j] = min{dp[0][j-1]+abs(v[0][j-1]-v[i][j]),
8 //dp[1][j-1]+abs(v[1][j-1]-v[i][j]),
9 //dp[2][j-1]+abs(v[2][j-1]-v[i][j])}

10 //计算第一列第一个节点到第i行第j列的节点(i,j)的最短路径长度
11 class solution{
12 public:
13     int func(vector<vector<int>>& v)
14     {
15         int n = v[0].size(); //列数
16         vector<vector<int>> dp(3, vector<int>(n-1,0));
17         //初始化第一列
18         dp[0][0] = abs(v[0][0] - v[0][1]);
19         dp[1][0] = abs(v[0][0] - v[1][1]);
20         dp[2][0] = abs(v[0][0] - v[2][1]);
21         for (int i=0;i<3;i++)
22             for (int j=0;j<n-1;j++)
23                 dp[i][j] = min3(dp[0][j-1]+abs(v[0][j]-v[i][j+1]),
24                                 dp[1][j-1]+abs(v[1][j]-v[i][j+1]),
25                                 dp[2][j-1]+abs(v[2][j]-v[i][j+1]));
26         return min3(dp[0][n-2],dp[1][n-2],dp[2][n-2]);
27     }
28 private:
29     int min3(int a, int b, int c)
30     {
31         return min(min(a,b),c);
32     }
33 };
34
35 int main()
36 {
37 //int a;
38 //cin >> a;
39 vector<vector<int>> v;
40 solution s;
41 int result1 = s.func(v);
42 int result2 = s.func(v);
43 int result3 = s.func(v);
44 //cout << "Hello World!" << endl;
45 return min(min(result1,result2),result3);
46 }

```

## 八、快手 MMU(2021.04.26)

一面完之后紧接着二面，两个面试都是先介绍项目然后一道算法题：

技术一面：

先介绍项目，然后针对项目提问：

(1) 灵长类动物姿态估计中选用简单的 SimpleBaseline 提升效果很高，换用更复杂的 HRNet 效果如何？为什么不用更好的 baseline 模型？(我回答 (1)HRNet 提升幅度没有 SimpleBaseline 高；(2) 投稿文章主要立足于特定场景问题的解决，提出的数据增强和模型改进对该场景有提升即可，并不需要去刷指标)

(2) Stack Hourglass Network 和 SimpleBaseline 都是先下采样后上采样，具体有什么区别？为何 SimpleBaseline 结构简单却性能好？(我回答 (1)Stack Hourglass Network 是一个多阶段的网络，每个阶段里面是先下采样后上采样，但是采用次数没有单阶段的 SimpleBaseline 多，并且 Hourglass 还有中间监督等；(2)Hourgalss 性能比不过 SimpleBaseline，个人觉得最关键的原因在于 feature map 分辨率如何恢复的问题，Hourgalss 的上采样过程只是简单的插值过程，而 SimpleBaseline 通过三层反卷积来恢复分辨率，这个过程通过让网络学习卷积核的参数达到分辨率恢复的目的，比简单的上采样要好，这也是为何后续 HRNet 要保持高分辨率的原因)

(3) 为何 GhostNet 在 ImageNet 上表现得比 MobileNetV2 好，但在真实业务场景 MobileNetV2 好？(我回答 (1)ImageNet 数据集比较干净，每一类的特征区别较大，用一个较少参数的网络可能可以实现较好的性能；(2)MobileNetV2 参数比 GhostNet 多，卷积操作比 GhostNet 中的模块复杂，对待真实业务场景中很多的困难样本，用参数更多，操作更复杂的模型可能更合适)

算法题：

leetcode322

技术二面：

先介绍项目，针对项目提问：

(1) 灵长类动物姿态估计中，改进的模型有没有在 COCO 上去实验，效果如何？(我回答还没有去做这个实验，因为文章立足于特定场景问题的解决，在小数据集上表现好即可。在 COCO 可以推测效果有提升，因为改进的网络结构模型参数增加了)

算法题：

leetcode189

## 九、微软亚洲研究院 (2021.06.01)

只有一面，项目聊完之后一道算法题

算法题：字符串相乘，leetcode43

## 十、阿里健康 (一面 2021.06.01，二面 2021.06.08，三面 2021.06.11)

(1) 介绍对偶问题的概念以及强对偶和弱对偶问题；

(2) 介绍支持向量机，为什么支持向量机需要写出对偶问题？

原问题的不是一个凸问题，需要转换成二次规划来解，二次规划是个凸问题，有全局最优值。

算法题是搜索旋转排序数组，leetcode33

二面较难，问的很多很细致的基础问题：

### 1. 基础题

(1) 二分法时间复杂度  $O(\log N)$ ，快排时间复杂度  $O(N \log N)$  和空间复杂度 ( $O(\log N)$ )，归并排序？(快排是递归地调用，每一次 partition 将数组分为小于 pivot 和大于 pivot 的两部分，随后递归地调用两部分的数组，在比较均匀的情况下，相当于是一个二叉树的深度优先遍历，每次递归调用函数时，需要记录该节点的地址信息，树的高度为  $\log_2 N$ ，故空间复杂度为  $O(\log N)$ )

(2) 介绍特征值和特征向量的概念，并介绍其在 PCA，LDA 中的应用以及奇异值分解。(需要介绍 PCA,LDA 的求解过程，使用了 lagrange 乘子法)

(3) 介绍 boosting 和 bagging 的区别？

adabost 是 boosting 的一种，随机森林是 bagging 的一种，所以 boosting 和 bagging 的类比类似于 adabost 和随机森林的对比：

样本选择上：Bagging 采取 Bootstrapping 的是随机有放回的取样，Boosting 的每一轮训练的样本是固定的，改变的是每个样的权重；

样本权重上：Bagging 采取的是均匀取样，且每个样本的权重相同，Boosting 根据错误率调整样本权重，错误率越大的样本权重会变大；

预测函数上：Bagging 所有的预测函数权值相同，Boosting 中误差越小的预测函数其权值越大；

并行计算：Bagging 的各个预测函数可以并行生成，Boosting 的各个预测函数必须按照顺序迭代生成。

(4) 随机森林中的子决策树是如何生成的？每次取随机的特征和随机的样本来生成子决策树，随机取的比例是多少？为什么是这个比例？

随机森林是 Bagging 方法的一种，随机取特征和样本生成子决策数。样本采样使用 0.632 自助法，假设样本数为  $m$ ，样本每一次被采样的概率是  $\frac{1}{m}$ ，没被采样到的概率为  $1 - \frac{1}{m}$ ，连续采样  $m$  次都没有被采样到的概率为  $(1 - \frac{1}{m})^m$ ，当  $m$  趋近无穷大时，极限值等于  $\frac{1}{e} = 0.368$ ，所以每次取样取 0.632 的样本。

(5) 机器学习中的经验风险和结构风险分别是什么？

经验风险：

机器学习模型关于训练数据集的平均损失称为经验风险，度量平均意义上模型预测效果的好坏，即  $R_{emp}(f) = \frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i))$ 。

结构风险：

在经验风险的基础上模型复杂度的正则项就是结构风险，即  $R_{str}(f) = \frac{1}{N} \sum_{i=1}^N L(y_i, f(x_i)) + \lambda J(f)$ ， $J(f)$  是模型的复杂度，模型  $f$  越复杂， $J(f)$  值就越大。 $\lambda$  是正则项的系数， $\lambda \geq 0$ 。需要权衡经验风险和模型复杂度，使得求解的目标是整体达到最小值，正则项  $J(f)$  的存在迫使模型不要太复杂，起到惩罚的作用。

(6) 一个圆周上随机取两个点，这两个点连线穿过圆心的概率？一个圆周上随机取三个点，问这三个点构成锐角三角形的概率？

第一个问题：首先固定一个点，在圆周上取另外一个点构成直径的概率为 0；

第二个问题：固定两个点，这两个点穿过圆心的概率为 0，所以随机取两个点应该不穿过圆心，可以求出两个点与圆心连线构成的角度的期望，角度符合  $[0, \pi]$  的均匀分布，期望值为  $\frac{\pi}{2}$ ，假设两个点  $A, B$  与圆心  $O$  连线角度为  $\theta$ ，延长  $AO, BO$  分别与圆周交于  $A', B'$ ，则当第三个点位于劣弧  $A'B'$  上时，三个点才构成锐角三角形，劣弧  $A'B'$  对应的角度为  $\theta$ ，所以概率为  $\frac{\theta}{2\pi}$ 。这个概率值是一个随机变量，服从  $[0, 1/2]$  上的均匀分布，所以概率的期望是  $\frac{1}{4}$ 。

## 2. 算法题

```
// 题目1：实现shuffle 函数，使得数组被均匀打乱，即每个位置出现每个数的概率是一样的
1 // 假设已知random函数可以生成0~1之间的随机数
2 // 思路，用n乘以0~1之间的随机数再向下取整，就可以均匀得到0,1,2,...,n-1中任意一个数，且这n个数取值的概率
3 // 一样都为1/n
4 void shuffle(vector<int>& v)
{
5     int n = v.size();
6     //n次随机
7     for(int i=n;i>=1;i--)
8     {
9         int index = int(i*random());
10        swap(v[index], v[i-1]);
11    }
12}
13

16 // 题目2：写一个函数，从一个有序数组里查找一个元素，找到则返回其下标，找不到返回-1，不用递归
17 int find(const vector<int>& v, int target)
{
18     if (!v)
19         return -1;
20     int i = 0;
21     int j = v.size() - 1;
22     while (i < j)
23     {
24         int mid = (i+j)/2;
25         if (v[mid]==target)
26             return mid;
27         else if (v[mid]<target)
28         {
29             i = mid+1;
30         }
31         else
32             j = mid;
33     }
34     return v[i]==target?i:-1;
35 }
```

---

三面 leader 面，