General Question

1.

怎么防止DDOS攻击:（Distributed Denial Of Service）

一、备份网站

备份网站不一定是全功能的，如果能做到全静态浏览，就能满足需求。最低限度应该可以显示公告，告诉用户，网站出了问题，正在全力抢修。

二、HTTP 请求的拦截

HTTP 请求的特征一般有两种：IP 地址和 User Agent 字段。比如，恶意请求都是从某个 IP 段发出的，那么把这个 IP 段封掉就行了。或者，它们的 User Agent 字段有特征（包含某个特定的词语），那就把带有这个词语的请求拦截。HTTP 拦截有一个前提，就是请求必须有特征。但是，真正的 DDOS 攻击是没有特征的，它的请求看上去跟正常请求一样，而且来自不同的 IP 地址，所以没法拦截。

（1）专用硬件

Web 服务器的前面可以架设硬件防火墙，专门过滤请求。这种效果最好，但是价格也最贵

（2）本机防火墙

操作系统都带有软件防火墙，Linux 服务器一般使用 iptables。

（3）Web 服务器

Web 服务器也可以过滤请求。Web 服务器的拦截非常消耗性能，尤其是 Apache。稍微大一点的攻击，这种方法就没用了。

三、带宽扩容

对于网站来说，就是在短时间内急剧扩容，提供几倍或几十倍的带宽，顶住大流量的请求。这就是为什么云服务商可以提供防护产品，因为他们有大量冗余带宽，可以用来消化 DDOS 攻击。

四、CDN

网站内容存放在源服务器，CDN 上面是内容的缓存。用户就近访问CDN，提高速度。如果内容不在 CDN 上，CDN 再向源服务器发出请求。

http://www.ruanyifeng.com/blog/2018/06/ddos.html

2.

爬虫相关的问题：

一、技巧

（1）设置下载等待时间/下载频率

（2） 设置**cookies**

（3） 修改**User-Agent**（用户代理）

User-Agent是指包含浏览器信息、操作系统信息等的一个字符串，也称之为一种特殊的网络协议。服务器通过它判断当前访问对象是浏览器、邮件客户端还是网络爬虫。在request.headers里可以查看user-agent.

（4） 修改**IP**

（5） 分布式爬取

看一下此前的Python爬虫项目。

二、爬虫如果中间崩了，怎么确定从哪里再继续

核心思路是做好日志（log）。爬虫程序本身应该是**无状态**的，任何想要**持久化**的东西都应该放在**数据库里**，mongo、redis、mysql、postgres，或者直接使用文件系统、Python内部储存管理（使用两个`set`，一个管理已经爬取过的链接（set1），一个管理未爬取的链接（set2），每个网页上爬取到的新的链接都经过与set2做对比，若**不在set2内且不在set1内**则加入，然后set2取出时将其加入set1内 ）等等。

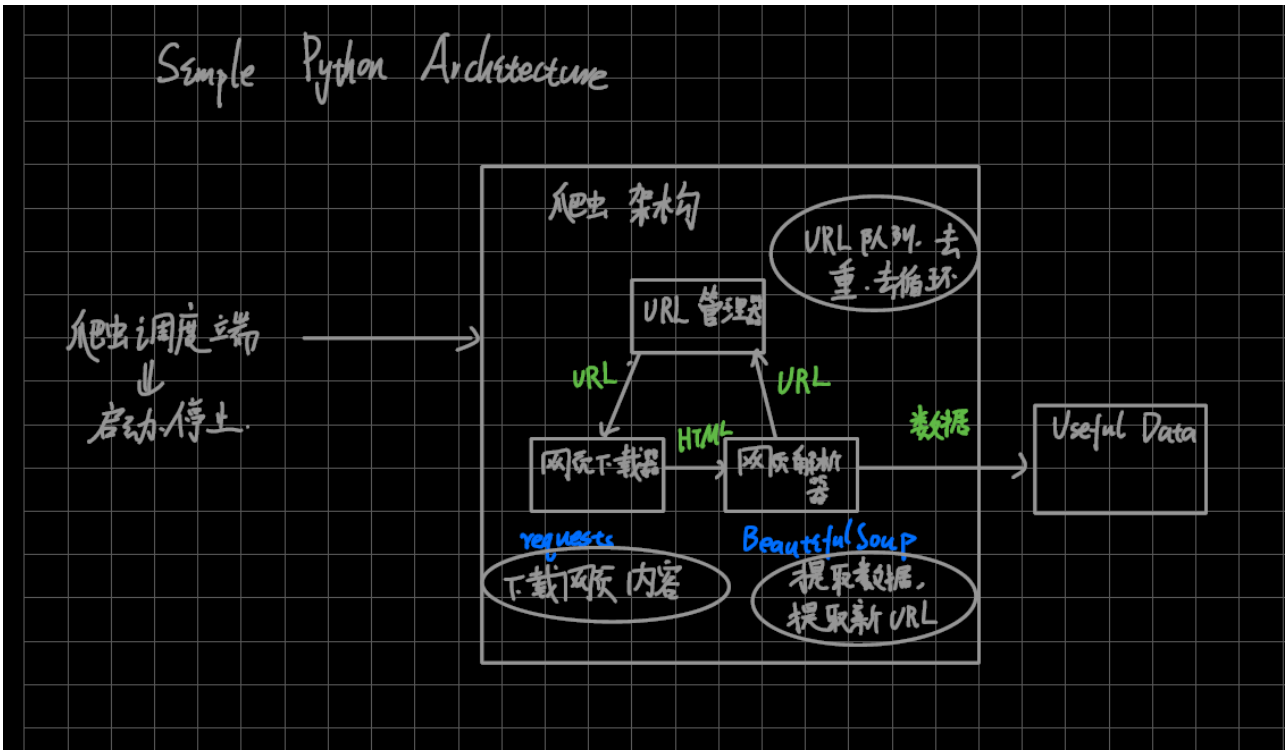
（1）

每次抓取之后立刻**更新数据库**或者更新文件，把已经抓取的链接标记.

然而实时保存意味着程序需要多费一部分功夫做这个事情，因此会产生很多细节问题，比如说连接数据库的频率太高等，会在实际问题中影响效率造成问题。

（2）

实现上可以用Python的`request`+ `beautifulsoup4`的包加上URL管理器的组合。具体结构如下：



https://www.zhihu.com/question/29754961

3.

B树，B+树：

每个节点的key值可以看作管理一棵子树（父节点存有右孩子的第一个元素索引），这个子树上的值都大于等于key值。左侧key的子树的值都小于该key值

共同点：

- 根节点： [1,m-1], 叶节点：[m/2,m-1]
- 插入：分裂，中间元素提到父节点。注意B树的非叶子节点可以储存数据而B+树不行，因此在叶节点的右子树要补上中间元素。B树插入为自顶向下，B+树为自底向上。
- 删除：叶子节点：向兄弟节点借/合并；非叶子节点：用后继元素覆盖要删除的key，然后在右子树上删除后继元素，回到叶子节点删除的情况。注意B+树可以直接向兄弟节点借，并修改父节点为借了的节点的key值。当B+树合并时，可以删除被合并节点相应的父节点，因为不再有用了。

B+树相对于B树有一些自己的优势，可以归结为下面几点。

- 单一节点存储的元素更多，使得**查询的IO次数**更少，所以也就使得它更适合做为数据库MySQL的底层数据结构了。
- 所有的查询都要查找到叶子节点，**查询性能是稳定**的，而B树，每个节点都可以查找到数据，所以不稳定。
- 所有的叶子节点形成了一个有序链表，更加便于查找。

https://segmentfault.com/a/1190000020416577

4.

MySQL相关

1.

数据库知识复习

2.

数据库题目复习

3.

具体知识点整理和个人理解，见我的Github项目：Database_technology

OS 相关

1.

Mutual lock的原理，机制，损耗

- 原理：Mutual lock和Semaphore、Monitors均是实现Process sychronization的一些常用方法。它们的原理即确保：
    a. Mutual Exclusion: 每次使用critical section 只有一个process, 晚到或者优先级低的process需要等待；
    b. Progress: 在当前process执行完成之后，等待的process会被唤醒（`signal()`）执行它的critical section
    c. Bounded waiting: 每个process不会无限制的等待下去 。
- 机制：这里以mutual lock 为例，使用`wait()`函数进入等待或者抢占互斥锁；使用`signal()`函数释放互斥锁。

deadlock:

- 死锁产生的4个必要条件 1、互斥 2、占有且等待 3、不可抢占 4、循环等待

starving:

- 死锁与starvation的一个重要区别，死锁占有资源，但是starvation不占有资源
- 解决：
    - First-in-First-out
    - 允许进程间优先级抢占
    - 确保进程处于以下两种状态：一次性拥有其需要的资源并执行；手头没有任何资源并在等待资源
    - Resource allocation graph, ensure the graph has no cycles
    - Banker's algorithm, ensure the system in safe state
    - Periodically call detect algorithm, if detects deadlock, select some safe process (low priority, low execution time, hold large resource, need a long time to complete...) to kill and preempt some resources.

2.

进程线程区别：

- process是资源管理最小单位，thread 是CPU调度最小单位
- process之间的通信开销大，可以用管道(pipeline)，rpc，文件等
- thread之间通信开销小，因为共享内存，线程之间共享内存的安全性基于COW

线程挂了会让内存也崩掉吗？如果突然关机了，内存中数据不就没了吗？--> 一步步引导，可以设定数据优先级，根据优先级设计存储策略

僵尸进程：

- 一个进程使用`fork()`创建子进程，如果子进程(init除外)在exit()之后，而父进程并没有调用`wait()`或`waitpid()`获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。这时用ps命令就能看到子进程的状态是“Z”。如果父进程能及时 处理，可能用ps命令就来不及看到子进程的僵尸状态，但这并不等于子进程不经过僵尸状态。 如果父进程在子进程结束之前退出，则子进程将由init接管。init将会以父进程的身份对僵尸状态的子进程进行处理。
- 危害：僵尸进程保留的那段信息不会释放，其进程号就会一直被占用。但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。
- 如何避免：如果要消灭系统中大量的僵死进程，只需要将其父进程杀死，此时所有的僵死进程就会变成**孤儿进程**，从而被init所收养，这样init就会释放所有的僵死进程所占有的资源，从而结束僵死进程。

守护进程：

Linux Daemon（守护进程）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。它不需要用户输入就能运行而且提供某种服务。。Linux系统的大多数服务器就是通过守护进程实现的。

一个守护进程的父进程是init进程，因为它真正的父进程在fork出子进程后就先于子进程exit退出了，所以它是一个由init继承的**孤儿进程**。守护进程是非交互式程序，没有控制终端，所以任何输出，无论是向标准输出设备stdout还是标准出错设备stderr的输出都需要特殊处理。

僵尸进程，孤儿进程，守护进程区别

3.

OS知识点复习

4.

OS题目复习

5.

具体知识点整理和个人理解，见我的Github项目：Operating Systems

C++

1.

《Effective C++》

2.

《More Effective C++》

static, const, 引用计数相关...

3.

C++11新增的属性：

1. 自动类型推断`auto`: 编译器在一个变量声明的时候，能够根据变量赋的值推断该变量的数据类型。这样就有些逼近Python中定义变量的功能，无需提前声明定义的变量的数据类型。E.g.

```
auto i = 1;     //编译器自动推断i为int类型

vector<int> vec(6,10);
vector<int>::iterator iter = vec.iterator();
auto iterAuto = vec.iterator(); //相比较上一句方便很多
```

2. 匿名函数`Lambda`: 该功能函数实际上在其他面向对象语言中早就存在，例如Java，Python都定义了该功能。C++中Lambda表达式格式如下：

```
[capture](params)->ret { body};

auto func = [](int i){ return i+4};// 可以体会auto的好处了
cout<< func(10) << endl;            //输出为14
```

- capture: 可见域范围内lambda表达式代码内可见的参数
- params: lambda表达式内部变量定义
- ret: 返回类型，如果 lambda 代码块中包含了 `return` 语句，则该 lambda 表达式的返回类型由 `return`语句的返回类型确定。如果没有 `return`语句，则类似 void f(...) 函数。

3. 显示重写`override`和`final`:

   a. `override`: 编译器将去检查基类中有没有一个具有相同签名的虚函数，如果没有，编译器就会报错。用于防止重写基类某个函数的时候却意外地创建了另一个虚函数。

```
struct Base {
    virtual void some_func(float);
};
struct Derived : Base {
    virtual void some_func(int) override; // 病态的,不会重写基类的方法
};
```

   b. `final`: 1. 防止基类被继承；2. 防止基类函数被子类重写

```
struct Base1 final { };

struct Derived1 : Base1 { }; // 病态的, 因为类Base1被标记为final了

struct Base2 {
    virtual void f() final;
};

struct Derived2 : Base2 {
    void f(); // 病态的, 因为虚函数Base2::f 被标记为final了.
};
```

   需要注意的是，override和final都不是C++语言的关键字。他们是技术上的标识符，只有在它们被用在上面这些特定的上下文在才有特殊意义。用在其它地方他们仍然是有效标识符。

4. 空指针常量 `nullptr`: `NULL`通常在C语言中预处理宏定义为 (void*) 0 或者0，这样0就有int型常量和空指针的双重身份。为了避免这个歧义，C++11重新定义了一个新关键字`nullptr`，充当单独空指针常量。

5. `long long int`: C++03中，最大的整数类型是`long int`。它保证使用的位数至少与int一样。 这导致`long int`在一些实现是64位的，而在另一些实现上却是32位的。C++11增加了一个新的整数类型`long long int`来弥补这个缺陷。它保证至少与`long int`一样大,并且**不少于64位**。

6. 线程支持:

    - 新的标准库提供了一个线程类`(std::thread)`来运行一个新线程
    - `std::thread::join()`支持的线程连接操作可以让一个线程直到另一个线程执行完毕才停止
    - `std:thread::native_handle()`成员函数提供了对底层本地线程对象的可能且合理的**平台相关的操作**
    - 为支持线程同步,标准库增加了**互斥体**(`std::mutex, std::recursive_mutex`等)和**条件变量**(`std::condition_variable` 和 `std::condition_variable_any`)。这些都是通过RAII锁和加锁算法就可以简单使用的。
    - `futures`和`promises`，用于在线程间传递异步结果，并且提供了`std::packaged_task`来封装可以产生这种异步结果的函数调用。

7. 元组类型: 难用，几乎没有用过。

4.

list, set, map, multimap, hashmap, treemap, vector, queue, stack等数据结构

- map存储的是键值对，set一般存值，都是红黑树实现
- **map和set都不允许重复**，multimap才允许重复
- map的迭代器不允许修改key，可以修改value，而set的迭代器是const的

5.

malloc, new, delete

- `malloc` 和 `new`:

    a. **一般的**g++编译器实现的`new`的调用过程如下：**new operator->operator new->malloc**。即new 通过 调用malloc实现

    malloc分配的内存位于**堆**上，new分配的内存位于‘自由存储区’，自由存储区是C++中一个抽象的概念，有别于堆。（operator new可以被重载，所以通过operator new分配的内存未必都在堆上）

    b. new和delete会调用类的`constructor`和`destructor`，malloc则不会

c. malloc分配内存失败会返回`nullptr`，而new则会直接抛出异常。另外new还可以通过`set_new_handler`设置分配失败时执行的逻辑，malloc则没有提供这样的使用方式

d. malloc返回的指针类型是`void*`，需要手动强转为需要的类型，而new不需要如此，new的返回类型是**类型安全的**

- `free`和`delete`：delete时会先调用类的destructor，再调用free释放这个对象。
- malloc分配的时候要指定分配内存的大小，利用free释放的时候却不需要指定大小，那么free是怎么知道该释放多大内存呢？在实现上，malloc分配的内存大小会比传入的值稍大，**多余的内存**用来存储分配的内存大小等额外的关于这段内存的信息（用户是无权访问的，应该是由操作系统管理），然后在free的时候找到这个内存的大小，就可以释放对应的内存了。另外因为delete内部实现也是调用free，因此delete也不用传入内存大小。

6.

C++的三大特性（或者是C++对象的三大特性）

- 封装
- 继承

  纯虚函数与抽象类

  - 如果类中至少有一个函数被声明为纯虚函数，则这个类就是抽象类。纯虚函数是通过在声明中使用 "= 0" 来指定的，例如

```
class Box
{
   public:
      // 纯虚函数
      virtual double getVolume() = 0;
   private:
      double length;      // 长度
      double breadth;     // 宽度
      double height;      // 高度
};
```

  设计抽象类的目的，是为了给其他类提供一个可以继承的适当的基类。**抽象类不能被用于实例化对象**，它只能作为**接口**使用。如果试图实例化一个抽象类的对象，会导致编译错误。

- **抽象类可以有成员变量吗？**

  可以拥有，但是不推荐。因为抽象类只能作为接口使用，不能用于实例化对象，因此成员变量没有什么意义，在继承子类中定义成员变量较好。

- 多态

  多态的类型：

  静态多态指依靠函数重载和泛型编程的**编译期多态**，动态多态和**虚函数**有关，在运行期确定。

  静态绑定和动态绑定：

  对于一般的成员函数，它和（指针）对象的静态类型绑定，称为**静态绑定**，发生在编译期；而对于虚函数，它绑定的则是指针对象的动态类型，称为**动态绑定**，发生在运行期。注意绑定指的是函数和和指向函数的对象之间的关系。

  虚函数是如何工作的：

  编译器会为每个**有虚函数的类**创建一个虚函数表，该虚函数表将被该类的**所有对象**共享。 类的每个虚成员占据虚函数表中的一行。如果类中有N个虚函数，那么其虚函数表将有**N*4**字节的大小。虚函数表通过虚函数表指针指向。

  - 在单继承中，Child类先复制了Base类的虚函数表，此后覆盖Base类中的同名虚函数。在虚函数表中体现为对应位置被Child类中的新函数替换，而没有被覆盖的函数则没有发生变化。对于子类自己的虚函数，直接添加到虚函数表后面。
  - 在多继承中，有多少个**虚基类就有多少个虚函数表指针**。子类虚函数会覆盖**每一个**父类的**每一个**同名虚函数。当父类中没有的虚函数而子类有，填入**第一个虚函数表**中，且用父类指针是不能调用。父类中有的虚函数而子类没有，则不覆盖，仅子类和该父类指针能调用。

  https://blog.csdn.net/s_lisheng/article/details/75034966

  https://blog.csdn.net/qq_36359022/article/details/81870219

  虚函数表：

  虚函数表指针在对象的地址最前端，可以通过`(int*)&F`进行强制类型转换来获取。虚函数储存在虚函数表中，每个父类都保持有这样一张虚函数表，当动态类型绑定时，运行期间父类指针通过查找虚函数表确定应该调用哪个虚函数。虚函数表的第一个虚函数可以通过`(int*)*(int*)(&F)`来获取。这里经过解引`(int*)&F`获得虚函数表首地址储存的内容，然后再进行一次强制类型转换得到第一个虚函数的地址。虚函数表中的虚函数通过以下代码获取：

```
pFunc pfun;
pfun=(pFunc)*((int*)(*(int*)(&b)));
pfun();
pfun=(pFunc)*((int*)(*(int*)(&b))+1);
pfun();
pfun=(pFunc)*((int*)(*(int*)(&b))+2);
pfun();
```

https://starkschroedinger.github.io/2020/10/05/dynamic/

什么函数不适合做虚函数：

- 友元函数，它不是成员函数，更不能为虚。
- 全局函数
- 静态成员函数，这没有`this`指针做参数。
- 各种`构造函数以及操作符重载`一般不建议为虚函数，这也是编译期要求的。
- 想要`inline`的函数，由于inline是在编译器进行，而多态需要运行期确定，所以此时编译器忽略`inline`建议并给出嘲讽（warning）。

dynamic_cast：

dynamic_cast的作用是将一个基类对象指针**安全地**cast到子类指针，即对指针做**下行转换（downcast）**，如此一来，基类指针也能够访问被指向的子类对象的**非虚成员**了，这才是主要目的。`dynamic_cast`会根据指针的实际指向对象类型和基类的继承关系来做相应处理，如果转换失败，对指针将会返回一个`NULL`，对引用将会抛出一个异常，这就是**安全性**所在。使用的前提有两个：1. 基类存在虚函数，否则编译期报错；2.待转换的基类的指针指向子类对象，否则转换是不安全的。真正在项目中使用`dynamic_cast`的实用建议：不要用！(要用尽量也压到继承树的底部)。

基类指针如何动态得知道自己指向的动态对象？

使用dynamic_cast，当dynamic_cast成功转换为子类时，证明该基类指针指向的是该子类动态对象。

https://blog.csdn.net/Stephan14/article/details/48883325

7.

友元函数

- **既然私有成员不想让外界访问，为什么设计友元的方式去访问其私有成员？**

  C++ 是从结构化的C语言发展而来的，需要照顾结构化设计程序员的习惯，当他们就是想在类的成员函数外部直接访问对象的私有成员，那还是做一点妥协以满足他们的愿望为好。

- **如何访问其他类的私有成员？**

  a. 调用其他类的public函数访问。
  b. 友元函数、友元类中访问

8.

函数模板与函数重载

- 函数重载需要函数体不同（形式参数类型、个数、顺序至少有一个不同，函数返回值类型可以相同也可以不同）。重载的函数视为多个函数
- 模板只适用于函数体相同、函数的参数个数相同而类型不同的情况，如果参数的个数不同，则不能用函数模板。

9.

静态库和动态库的区别

- 动态库**避免代码重复**，可执行文件大小远远小于静态库：静态链接的可执行文件需要用到的代码从二进制文件中“拷贝”了一份，而动态库仅仅是复制了一些重定位和符号表信息。
- 动态库的**扩展性**要优于静态库：如果静态库中某个函数的实现变了，那么可执行文件必须重新编译，而对于动态链接生成的可执行文件，只需要更新动态库本身即可，不需要重新编译可执行文件。正因如此，使用动态库的程序方便升级和部署。
- 静态库的**移植性**要优于动态库：静态链接的可执行文件不需要依赖其他的内容即可运行，而动态链接的可执行文件必须依赖动态库的存在。

10.

单例模式

- 线程安全写法：
  - 一个私有构造函数 （确保只能单例类自己创建实例）
  - 一个私有静态变量 （确保只有一个实例）
  - 一个公有静态函数 （给使用者提供调用方法）

```
cpublic class Singleton {

    private static Singleton uniqueInstance = new
Singleton();

    private Singleton() {
    }

    public static Singleton getUniqueInstance() {
        return uniqueInstance;
    }

}
```

- 应用场景：
    a. **频繁实例化然后又销毁的对象**，使用单例模式可以提高性能。
    b. **经常使用的对象，但实例化时耗费时间或者资源多**，如数据库连接池，使用单例模式，可以提高性能，降低资源损坏。
    c. **使用线程池之类的控制资源时**，使用单例模式，可以方便资源之间的通信。

Python

1.

Python基础知识

2.

《Python编程，从入门到实践》

3.

list, set, dict, tuple

- list: 有序（可以索引、切片）、可重复、类型可以不同

- set: 无序（无法索引、切片）、不可重复（可以看成key的集合）、由list, dict + set() 得到、有并，交，差，对称差等集合操作
- dict: 无序、value可重复,key 一一对应、键值对
- tuple: （与list一致）有序、可以重复、类型可以不同，但不能修改

4.

Python中如何定义私有变量

- 默认情况下，Python中的成员函数和成员变量都是公开的(public),在python中没有类似public,private等关键词来修饰成员函数和成员变量。
  在python中定义私有变量只需要在变量名或函数名前加上 "__"两个下划线，那么这个函数或变量就是私有的了。
- 在内部，python使用一种 **name mangling 技术**，将 membername替换成 `_classname__membername`。例如：为了保证不能在class之外访问私有变量，Python会在类的内部自动的把我们定义的spam私有变量的名字替换成为 `_classname__spam`(注意，classname前面是一个下划线，spam前是两个下划线)，因此，用户在外部访问spam的时候就会
  提示找不到相应的变量。
- python中的私有变量和私有方法仍然是可以访问的；访问方法如下：
  - 私有变量: object.*classname__variablename*
  - 私有方法: object._classname__func()

机器学习

1.

随机森林(Random Forest)

- Bagging 和 Boosting:
- 随机森林介绍:

  采用Bagging思想，以决策树为弱分类器将他们集成起来，通过投票决定最终的分类。
- 为什么RF要随机选取一些特征:

RF的话，如果有一个**特征和标签特别强相关**。选择划分特征时，如果不随机的从所用特征中随机取一些特征的话，那么每一次那个强相关特征都会被选取。那么每个数都会是一样的。这就是随机森林随机选取一些特征的作用，让某些树，不选这个强相关特征。

- Boostrap aggregating: 即Bagging

2.

SVM (Support Vector Machine)

- 为什么要把原问题转换为对偶问题：因为原问题是凸二次规划问题，变量数量等于特征数量，当样本特征非常多时，求解难度很大。而若将原始问题转化为对偶问题，则求解更加高效。因为:
    a. 求对偶问题只用求解α系数，**而α系数只有支持向量才非0，其他全部为0**。因此α系数的个数为样本点的个数。（证明在KKT限制中）
    b. 可以方便地引入核函数，求解非线性SVM
- 如何求解SVM的对偶问题：SMO （Sequential Minimimal Optimization）为最常见的求解（即训练SVM模型）线性SVM模型的方法，也是最快速的求解方法之一。下面是SMO算法求解SVM的对偶问题：

  SVM对偶问题的优化目标：

$$\min_{\alpha} \sum_{m=1}^{N}\sum_{n=1}^{N}\alpha_m\alpha_n y_m y_n z_m z_n - \sum_{n=1}^{N}\alpha_n$$
$$s.t. \quad 0 \le \alpha_n \le C, for\ n = 1, 2, \dots N$$
$$\sum_{n=1}^{N} y_n\alpha_n = 0 \text{ (此限制在后续推导)}$$

  式中 C 是惩罚因子，优化目标的第二项是正则项/惩罚项。求解SVM的对偶问题即求解$\alpha_1, \alpha_2, \dots \alpha_N$. 在得到最优解之后，根据KKT:

$$\Delta_w L(w, \beta, \alpha) = w - \sum_{i=1}^{N}\alpha_i y_i x_i = 0$$
$$\Delta_\beta L(w, \beta, \alpha) = -\sum_{n=1}^{N} y_n\alpha_n = 0$$
$$\alpha_i g_i(w) = 0, i = 1, 2, \dots N$$
$$g_i(w) \le 0, , i = 1, 2, \dots N$$

  1.从第二个式子我们得知，目标函数需要满足限制: $\sum_{n=1}^{N} y_n\alpha_n = 0$.

  2.从第三四个式子我们可以知道，α_i**只有在$g(w) = 0$时（也就是支持向量）不为0，其他全部为0。**

  3.通过第一二个式子我们可以求出w, b:

$$w = \sum_{i=1}^{N} \alpha_i y_i z_i$$
$$b = y_n - w^T z_n$$

得到w,b之后，可以得到分类超平面

$$g(x_i) = \sum_{j}^{N} \alpha_j y_j x_i + b$$

按照SVM理论，如果g(x)是最优的分类超平面，则需要满足以下限制：

$$y_i g(x_i) = \begin{cases} \geq 1, & x_i | \alpha_i = 0; \\ 1, & x_i | 0 \leq \alpha_i \leq C; \\ \leq 1, & x_i | \alpha_i = C; \end{cases}$$

若$\alpha_1, \alpha_2, \ldots \alpha_N$满足了上述g(x)限制条件，则才是SVM的一组解。SMO算法求解时遵循以下原则：

- 每次优化时，必须**同时优化α的两个分量**，因为只优化一个分量的话，新的α就不再满足初始限制条件中的等式条件$\sum_{n=1}^{N} y_n \alpha_n = 0$。
- 每次优化的两个分量应当是违反**g(x)限制条件**比较多的。就是说，本来应当是大于等于1的，越是小于1违反g(x)限制条件就越多，这样一来，选择优化的两个分量时，就有了基本的标准。

3.

聚类算法

- K-means:
  - 思想：EM算法，目标为求得最小化平方误差对应的参数。
  - 优缺点：

    缺点：1. 易收敛到**局部最优**而非全局最优；2. **初始值**的选择对结果影响很大；3. 无法**自动确定K**值；4. **硬聚类**，无法给出每个点从属某个类的概率；5. 使用Euclidean Distance作为距离度量，对数据分布有先验的假设：**球状分布**，而现实中的数据很少有服从这个分布的，因此通常表现不够好。6. 易受**离群值**的影响

    优点：计算速度快:O(Nk) (N: #samples; k: #clusters)，$k << N$，因此通常可以看作线性时间复杂度O(N)
  - EM过程：

    E step: 将每个sample point分配给距离最近的cluster center;

    M step: 依据当前的分配，通过计算cluster points的均值作为新的聚类中心。

- 如何确定**K**：

    i. 拐点法：通过寻找Loss-K 折线图的拐点。此为一个经验的方法，不够自动化。

    ii. Gap statistic: Gap Statistic定义为$Gap(k) = E(logD_k) - logD_k$，其中$E(logD_k)$通过蒙特卡洛模拟产生：我们在样本所在的区域内按照均匀分布随机地产生和原始样本数一样多的随机样本，并对这个随机样本做K均值，得到一个D_k；重复多次就可以计算出$E(logD_k)$的近似值，它代表了随机样本损失的期望值。$logD_k$代表当前样本的损失值。二者相减得到的gap体现了聚类带来的效果，因此$gap(k)$越大表明聚类效果越好，此时k越佳。

- 如何改进：

    i. 核K均值算法：解决数据不服从球状分布假设

    引入核函数，将原始的数据空间变换到新的数据空间之后，在新的数据空间做K-means，此时数据满足球状分布假设。

    ii. K-means++：解决初始值的选择的影响

    根据规则依次选取初始化聚类中心 --> 随机选择第一个聚类中心，此后离当前已选择的聚类中心较远的点有较高的概率被选作聚类中心（通过概率分布的方法选择）。

    iii. ISODATA： 解决无法自动选取K值

    相比K-means需要给入更多的经验参数，思想为当一个聚类中点的数量小于给定的阈值时将它与最近的聚类中心合并；当一个聚类中心的方差大于给定的阈值时将它分裂为两个聚类中心；当聚类中心的个数等于0.5K或者2K时停止合并、分裂操作。

- 证明K-means收敛性：

    先证明EM算法的收敛性，再将K-means里面的步骤对应到EM算法。

- DBSCAN：

    密度聚类（Density-Based Spatial Clustering of Applications with Noise），利用传递闭包实现聚类。给定领域半径E和最少数目MinPts，核心对象定义为：一个样本点，半径为E的区域内（E领域）包含的样本数大于MinPts；**密度可达即核心对象的闭包**，即核心对象所在的E领域内存在另一个核心对象，则这两个核心对象直接密度可达，而通过直接密度可达相连的核心对象直接都是密度可达。因此DBSCAN的步骤即对每个点判断是否为核心对象，若不是则继续；若是则求它的密度可达，将密度可达内的样本点划分为一个类。

- GMM:

    Gaussian Mixture Model，本质思想为用高斯分布去拟合原始的数据，训练使用EM算法：

E-step：

$$r(z_{nk}) = \frac{\pi_k N(x_n|u_k, \Sigma_k)}{\sum_{i=1}^{K} \pi_i N(x_n|u_i, \Sigma_i)}, \quad n = 1, 2, \dots N; k = 1, 2, \dots K$$

where N is the number of samples and K is the number of clusters.

M-step:

$$argmax_\theta L(X) = argmax_\theta \sum_{i=1}^{N} log(P(x_i; Z|\theta))$$

$$= argmax_\theta \sum_{i=1}^{N} log(\sum_{z_i} P(x_i; z_i|\theta)) = argmax_\theta \sum_{i=1}^{N} \sum_{z_i} log(P(x_i; z_i|\theta))$$

$$= argmax_\theta \sum_{i=1}^{N} \sum_{z_i} log\frac{(P(x_i; z_i|\theta))Q(z_i)}{Q(z_i)} \geq argmax_\theta \sum_{i=1}^{N} \sum_{z_i} Q(z_i) log\frac{P(x_i; z_i|\theta)}{Q(z_i)}$$

From E-step we've known that $Q(z_i) = P(z_i|x_i; \theta)$，so we have:

$$argmax_\theta \sum_{i=1}^{N} \sum_{z_i} Q(z_i) log\frac{P(x_i; z_i|\theta)}{Q(z_i)} = argmax_\theta \sum_{i=1}^{N} \sum_{z_i} Q(z_i) log\frac{P(z_i|x_i; \theta)P(x_i|\theta)}{P(z_i|x_i; \theta)}$$

$$= argmax_\theta \sum_{i=1}^{N} \sum_{z_i} P(z_i|x_i; \theta) log(P(x_i|\theta))$$

Thus, we can get the update formula of parameters π, μ, Σ as below:

$$\pi_k = \frac{N_k}{N} = \frac{\sum_{i=1}^{N} r(z_{ik})}{N}$$

$$u_k = \frac{1}{N_k} \sum_{i=1}^{N} r(z_{ik}) x_i$$

$$\Sigma_k = \frac{1}{N} \sum_{i=1}^{N} r(z_{ik})(x_i - u_k)(x_i - u_k)^T, where\ k = 1, 2, \dots K$$

优点：1. GMM为**软聚类**，能给出样本点属于某个类的概率值，鲁棒性更好；2. GMM**没有对数据的先验假设**（理论上当K的数量足够大，使用全部的协方差类型的K个高斯分布可以拟合任意分布）。

缺点：1. 容易陷入局部最优值；2. 不能自动确定K值；3. 计算量相比K-means要大很多；4. 不适用于数据量少的情况，因为此时估计协方差很困难。

- Fuzzle-C means （FCM）
- 如何评估聚类算法的好坏：

a. 轮廓系数法：$s(p)=\frac{b(p)-a(p)}{\max\{a(p),b(p)\}}$，这里**p指的是给定点**，b(p)指的是p与另一个不同簇中的点之间的最小平均距离（如果有n个其他簇，则只计算和点p最接近的一簇中的点与该点的平均距离），a(p)指的是点p与同一簇中的其他点之间的平均距离。因此b(p)反应的是不同簇之间的紧凑程度而a(p)反应堆是同个簇中的紧凑程度。总体的思想为使得类内差异/距离小，类间差异/距离大。

b. R 方 （R Square）：$RS=\frac{\sum_{i=1}^{N}(x_i-\mu)^2-\sum_{i=1}^{N}\sum_{j=1}^{K}\gamma_{ij}(x_i-u_j)^2}{\sum_{i=1}^{N}(x_i-\mu)^2}$，其中$\sum_{i=1}^{N}(x_i-\mu)^2$表示原始数据集不进行聚类（视为一个类）的误差，而$\sum_{i=1}^{N}\sum_{j=1}^{K}\gamma_{ij}(x_i-u_j)^2$表示进行聚类之后的误差，二者相减体现聚类的效果。因此类似于Gap statistic ，R方越大表明聚类的效果越好。

c. 贝叶斯选择法（Bayesian model selection）

4.

KD-tree

- 思想：源于查找。经典的查找算法中输入的数据都是一维，因此可以构建二分查找树来提升查找的速度（线性查找: $O(n)$,二分查找树: $O(nlogn)$）。在输入数据是多维(k dimension)的情况下，构造KD-tree来加速查找。

  KD-tree构建：

  a. 建立根节点；
  b. 选取**方差最大的特征**作为分割特征 --> 希望尽可能的分散；
  c. 选择该特征的中位数作为分割点 --> 希望左右子树的规模尽可能相等；
  d. 将数据集中该特征小于中位数的传递给根节点的左儿子，大于中位数的传递给根节点的右儿子；
  e. 递归执行步骤2-4，直到所有数据都被建立到KD Tree的节点上为止。

- 为何能加速：

  KD-tree查找：

  a. 从根节点开始，根据目标在分割特征中是否小于或大于当前节点，向左或向右移动。
  b. 一旦算法到达叶节点，它就将节点点保存为“当前最佳”。
  c. 回溯，即从叶节点再返回到根节点
  d. 如果当前节点比当前最佳节点更接近，那么它就成为当前最好的。
  e. 如果目标距离当前节点的父节点的**超平面**的距离更接近，说明当前节点的兄弟节点所在的子树**有可能**包含更近的点。因此需要对这个兄弟节点递归执行1-4步。

  从KD-tree查找可以看出，平均时间复杂度为$O(nlogn)$，因此可以加速查找。

https://zhuanlan.zhihu.com/p/45346117

5.

避免过拟合的方法

- 获取更多的数据，这是最有效的方法。
- 损失函数添加正则化项（L1,L2）。例如LR加上L1正则化即LASSO Regression，加上L2正则化即Ridge Regression。L1正则化可以实现参数选择/参数稀疏（将系数压成0），而L2正则化不行（只能将系数压迫到接近0）: 因为从图像上看，L1正则是菱形，L2是椭圆形，参数线和L1正则相交往往取到顶点，此时参数系数被置为0，而参数线和L2正则相交往往取不到椭圆的切线，因此参数的系数只能趋近于0。
- 神经网络
    - Dropout:
        i. 应加在神经网络的顶部而非底部，即使是都加那么底部的dropout rate也要低于顶部的dropout rate: 底部应尽可能的保持输入信息的丰富性，而顶层加入Dropout 相当于引入随机的结构，使得一个model等同于多个相似但task-specific的特征略有不同的model叠加后取均值，减少了model 对某些特征学习不充分的现象（这里类似于集成学习的思想: 使用多个好而不同的model优化表现）。

        ii. 如何调整: 当过拟合现象明显减轻时减小dropout rate；过拟合现象仍然严重时继续增大dropout rate。

        iii. 卷积层dropout: 卷积层一般不使用dropout（全连接层用的多），因为卷积操作使得相邻的参数共享很多信息，即使一个卷积单元被dropout删除，它所包含的信息仍然可以从相邻的单元传递到下一层。因此此时dropout相当于加强model对噪声的鲁棒性，而达不到全连接层中的效果。

        卷积层使用的**spatial dropout**，它会随机的将整个特征图丢弃，而非仅仅丢弃单个单元。

        iv. 循环层dropout: RNN，LSTM，GRU等循环层的dropout**对不同的时间步采用相同的循环层dropout掩码**，使得model朝着正确的方向学习。

        v. 即使不出现过拟合现象（当前数据规模大于参数规模）也应当加入Dropout或者其他一些形式的噪声。它有助于处理异常值，避免网络中出现极端权重结果，打破数据中的偶然相关性，使得model的鲁棒性更强。
    - Batch Normalization:

i. 原理：简约白化（Whitening）+ 线性变换

- Whitening: 使得特征的均值μ为0，方差σ^2为1。白化通常有两种：PCA Whitening (特征的均值μ为0，方差σ^2为1) and ZCA Whitening (特征的均值μ为0，方差σ^2相等)。ZCA Whitening = PCA Whitening + Rotating，通过增加旋转操作，ZCA白化使得处理后的数据更加贴近原始的数据（保留了更多原始数据特征）。

  白化的**作用**为降低特征之间的关联性，PCA白化还可以起到降维的作用。

  白化的**缺点**为计算成本高，降低了网络的表达能力。--> 有部分底层参数信息被白化操作给丢弃了。

- 线性变换: 恢复数据的表达能力

ii. 作用:

i. 防止ICS（Inherent Covariate Shift）--> 降低模型对超参数的敏感度（ICS造成蝴蝶效应），使得调整超参数更容易，使得网路学习更加稳定。

起到相同效果的还有**权重初始化方法的选择**: e.g.使用Xavier（Glorot） initialization / He initialization/Lecun initialization 相比Gaussian initialization/Random initialization 更加稳定）和 **learning rate 的大小**: 小的learning rate 网络学习稳定，不容易diverge.

ii. 避免模型进入梯度饱和区（和relu作用相似） --> 加快模型的训练速度

ICS：内部协变量移位，即第i层的神经网络通过 $z^i = W^i X^i + b^i, A^i = g(z^i), X^{i+1} = A^i$将**参数的变化**传递给第i+1层，**影响第i+1层的数据分布**。因此第i+1层需要不停的去适应这样的变化，因为BP先作用于i+1层。

iii. 应加在神经网络的底部而非顶部，由于ICS通常在底部发生。BN layer是需要通过训练调整的，在训练时候得到μ, σ^2并将其用于测试集。--> 从这里可以看出BN Layer不能直接用于迁移学习，因为训练得到的μ, σ^2代表了数据变化的特征，和测试集上的数据有较大的差异。

白化是BN可以起到正则化的作用的原因：1. 白化降低了特征之间的关联性，同LR中一样，共线性会导致模型过拟合（实际有效的数据量少了，然而模型的规模是以原始的数据量为标准的，因此共线性的特征数据会造成过拟合）；2. 白化操作使得部分底层参数信息被丢弃，降低了网络的规模。PS：通常加了BN就不加Dropout

- Early Stopping: 训练轮数的调整避免过拟合

    - 降低网络复杂度（网络深度、宽度、共享权重...）：若不能增加数据的规模，那么为了使得参数规模与之相配（参数规模大于数据规模是引起正则化最本质的原因），可以降低参数的规模。
- 集成学习: 将多个模型集成在一起，降低单一模型过拟合的风险。E.g. Bagging

6.

模型超参数的调整

- 深度（number of layers）: 加深网络在模型欠拟合的时候可以起到显著的效果，然而这有个前提：较为浅层的网络能有效地提取数据中的信息。若浅层网络因没有能有效提取数据的特征而准确率低下，那么加深网络也不会有效果。
- 宽度（number of neurons in one layer）: 过大的网络宽度使得模型训练缓慢（相同的增大规模下加大宽度比加大深度更耗时），并且使得残留的噪声难以消除（BP无法有效的达到各个神经元）；过小的宽度造成了瓶颈（bottleneck），限制了网络的表达能力。
- 为隐藏层选择适合的激活函数：sigmoid/tanh 函数容易造成梯度消失，使得model训练缓慢；而relu函数可以有效解决这个问题，然而relu容易受不良梯度的影响，造成神经元死亡（在训练的过程中，如果Loss还未达到很小就连续几个epoch不发生变化，很有可能是relu已经造成所有的神经元死亡）。selu/leaky relu/elu等函数能高级激活函数能有效避免神经元死亡的问题。
- batch size: 过大的batch size 使得梯度下降的随机性减弱，降低了模型的探索能力；过小的batch size随机性过高，模型训练不稳定，而且不能很好的利用GPU的并行能力，造成训练速度缓慢。
- learning rate: 过大的learning rate使得model训练不稳定，容易造成divergence；过小的learning rate 使得model探索能力变差，容易卡在局部最优值，最重要的是使得训练变得十分缓慢。确定合适的learning rate可以将梯度剪裁（gradient clipping，用于避免某些异常的梯度对模型更新造成的影响，不会造成梯度爆炸）关闭，此后在误差爆炸的附近选取最佳的learning rate值（e.g. $\alpha^* = 0.9\alpha_{max}$）。

7.

经典机器学习分类模型

- 决策树
    - 什么是决策树：自顶向下，利用树形分叉来达到分类目的的模型。非叶节点为特征的属性，叶节点为分类的结果。通过非叶节点属性值的调整来拟合训练的数据，通过剪枝来提升泛化的能力，使之能应用于测试集。最优决策树的选取是一个NP完全问题，通常使用启发式算法来选取。单一决策树的分类能力较弱（为经典的弱分类模型），可以通过集成学习构造随

机森林、梯度提升树来改进。决策树的生成包括以下三个过程：**特征选择 --> 决策树的构造 --> 剪枝**

- 决策树的类别（根据启发算法分类）：
    - ID3：

      定义信息增益g(D,A)，选取信息增益最大的特征作为非叶节点。特征选择过程如下：

      $$g(D,A)=H(D)-H(D|A)$$
      $$H(D)=-\sum_{i=1}^{N}\frac{|C_i|}{|D|}log(\frac{|C_i|}{|D|})$$
      $$H(D|A)=\sum_{i=1}^{M}\frac{|D_i|}{|D|}H(D_i)=\sum_{i=1}^{M}\frac{|D_i|}{|D|}\sum_{j=1}^{N}\frac{|C_{ij}|}{|D_i|}log(\frac{|C_{ij}|}{|D_i|})$$

      可以看到，H(D)为信息熵，C每个输出的类别，D为样本集合，N为样本集合的类别数，M为特征A的类别数。信息增益的本质即给定条件带来的确定性增加的程度。特征A的类别数越多 ==> 选取特征A为条件带来的确定性越大 ==> 信息熵H(D|A)越小 ==> 信息增益g(D,A)越大。因此ID3决策树会倾向于选择类别较多的特征。

    - C4.5：

      定义信息增益率$g_R(D,A)$，引入归一化分母，缓解了特征的类别数对模型选择的影响。定义过程如下:

      $$g_R(D,A)=\frac{g(D,A)}{H_A(D)}$$
      $$H_A(D)=\sum_{i=1}^{M}\frac{|D_i|}{|D|}log(\frac{|D_i|}{|D|})$$
      $$g(D,A)=H(D)-H(D|A)$$

    - CART：

      定义基尼指数$gini(D|A)$，选择基尼指数最小的特征及其最小的切分点作为二分类的非叶子节点，当基尼指数为0时完成决策树的生长。定义如下：

      $$gini(D|A)=\sum_{i=1}^{M}\frac{|D_i|}{|D|}gini(D_i)$$
      $$gini(D_i)=1-\sum_{j=1}^{N}(\frac{|C_{ij}|}{|D_i|})^2$$

- 不同启发算法决策树之间的对比：
    - 样本类型上：ID3只能处理离散类型，C4.5和CART能处理连续类型

    - 应用问题上：ID3,C4.5只能应用于分类问题，而CART能用于分类问题和回归问题。
    - 构造准则上：ID3寻找最大信息增益的特征，C4.5寻找最大信息增益率的特征，CART寻找最小基尼指数的特征以及切分点。
    - 实现细节上：ID3,C4.5可以多叉分支，特征在层之间不会复用；CART只能二分类，特征在层之间可以复用。
- 决策树剪枝：
    - 分类：
        - 预剪枝（Pre-pruning）: 在生成决策树的过程中依照某些判定条件进行剪枝。常见的判定条件有：1. 树达到一定深度；2. 树的叶子节点的样本数少于一定阈值；3. 每次分裂对测试集准确的的提升小于一定阈值。

          优点：简单，运算高效，适合于大规模的数据

          缺点：有欠拟合的风险；选取阈值需要一定的经验
        - 后剪枝（Post-pruning）: 在决策树生成完成之后，自底向上的进行条件检查，剪枝。常见的判定条件有：剪枝后测试集的准确度有所提升（大于一定阈值）时进行剪枝。著名的后剪枝方法：错误率降低剪枝（Reduced Error Pruning，REP）、悲观剪枝（Pessimistic Error Pruning，PEP）、**代价复杂度剪枝（Cost Complexity Pruning，CCP）**、最小误差剪枝（Minimum Error Pruning，MEP）、CVP（Critical Value Pruning）、OPP（Optimal Pruning）等。

          优点：相对预剪枝，可以生成泛化能力更强的决策树

          缺点：时间复杂度较高 --> 需要等待决策树生成结束，并且后剪枝算法的时间复杂度至少和叶子节点的数量呈线性关系
    - CCP (Cost Complexity Pruning):

      误差增加率α：

      $$\alpha = \frac{R(t) - R(T_t)}{L(T_t) - 1}$$

      其中$R(T_t)$为t时刻模型T_t在测试集上的错误率，$R(t)$表示t时刻非叶子节点剪枝后模型在测试集上的错误率，$L(T_t)$表示模型T_t的复杂度，为叶子节点的数量。

t时刻，在模型T_t上选取误差增加率α_i最小非叶子节点t_i将其剪枝，并依次执行n次，知道得到根节点或者满足停止条件，此时获得**序列**<$T_0, T_1, \dots T_n$>。再从序列中选择测试集准确率最高的决策树作为泛化性能最好的模型。

Pros:

代价复杂度剪枝获得的模型精度与REP差不多，但形成的**树复杂度小**

Cons:

生成子树序列的时间复杂度与原始决策树的非叶结点个数呈**二次关系**，导致算法相比REP、PEP、MEP等线性复杂度的后剪枝方法，时间复杂度大

- 剪枝作用：

  剪枝比树的生成过程更为关键。对于不同划分标准生成的过拟合决策树，在经过剪枝之
  后都能保留最重要的属性划分，因此最终的性能差距并不大

- SVM
- 逻辑回归

8.

Bias and Variance

- 泛化误差由bias+variance组成，其中bias代表了训练误差，variance代表了测试集误差和训练集误差相比的增量。bias的产生是由于欠拟合，也就是模型的表达能力不足（可能是模型的规模不够大，也可能是模型的建模方式和数据集不匹配（e.g. 线性模型匹配非线性数据集））；variance的产生是由于过拟合，也就是模型的参数量大于数据量，导致模型的泛化能力差。

9.

集成学习 (Ensemble Learning)

- What is ensemble learning:

  集成学习通过将多个学习器（弱学习器，因为通常集成学习用于分类问题，因此也叫弱分类器）进行集成从而达到比单个学习器更好的训练集上的拟合性能和测试集上的泛化性能。集成学习研究的**核心**在于找到”**好而不同**“的基分类器（个体分类器越准确，多样性越大，集成效果越好，此通过误差-分歧分解证明）。

- 集成学习分为几种？它们有何异同？

集成学习方法主要分为Boosting、Bagging (Boostrap Aggregating)、Stacking三种。

- Boosting的**基本思路**是将基分类器依次进行叠加，每一层在训练的时候，对前一层基分
  类器分错的样本，给予更高的权重（类似于人类通过错误进行学习），当残差为0或基学习器数目达到指定上限时停止。测试时，根据各层分类器的结果的**加权**得到最终结果，因此它是一个串行的方法，基分类器之间彼此强相关。

  Bagging的**基本思路**是一个集体决策的过程，每个个体都进行单独学习，学习的内容可以相同，也
  可以不同，也可以部分重叠。但由于个体之间存在差异性，最终做出的判断不会完全一致。在最终做决策时，每个个体单独作出判断，再通过投票的方式做出最后的集体决策。因此它是一个并行的方法，基分类器之间彼此独立（或弱相关）。

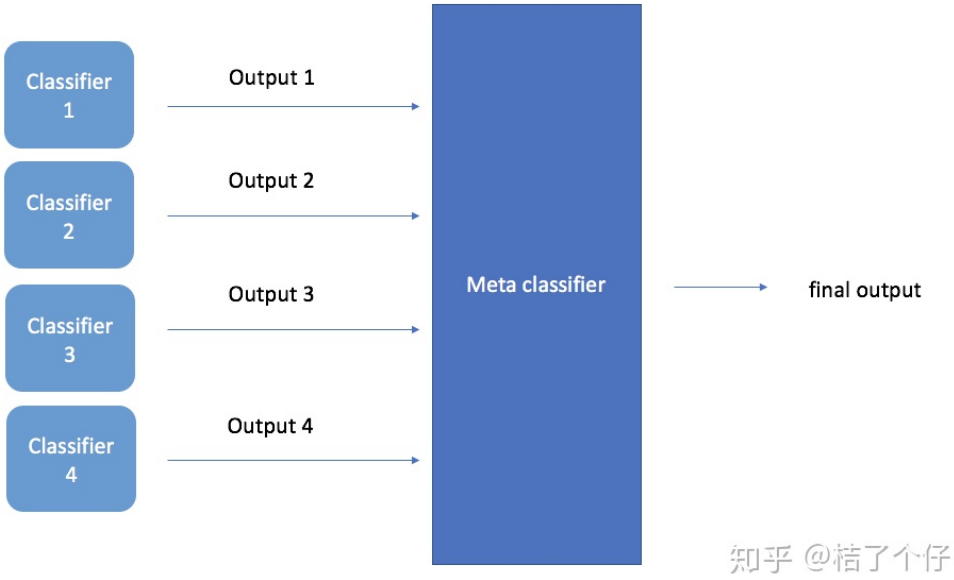
- Boosting的**主要好处**在于降低模型的bias，由于每层分类器都是加强对前一层分类器的 错误/残差 学习，因此可以有效降低基分类器的bias。。但Boosting的过程并不会显著降低方差。这是因为Boosting的训练过程使得各弱分类器之间是强相关的，缺乏独立性，所以并不会对降低方差有作用。

  Bagging的**主要好处**在于降低模型的variance，由于基分类器之间彼此独立（相关性很小），因此可以假设它们出错的概率也彼此独立，通过**Hoeffding不等式**我们知道集成模型的错误率随着基分类器数目的增加呈指数下降。通过相互独立变量的方差计算公式我们得知:

  $$x_1, x_2, \dots x_n \text{相互独立，则} \frac{\sum_{i=1}^{n} x_i}{n} \text{这个随机变量的方差为：}$$
  $$\alpha\sigma^2 + \frac{(1-\alpha)\sigma^2}{n} = 0 * \sigma^2 + \frac{(1-0)\sigma^2}{n} = \frac{\sigma^2}{n}$$

  因此Bagging能有效降低集成模型的方差和错误率。

- Stacking:



stacking和bagging的区别在于：

i. stacking的每个classifier是不同的模型，而bagging每个classifier其实是同一种模型

ii. stacking每个classifier都用了**全部**训练数据，bagging每个bag用**部分**数据训练

Note:

Bagging, Boosting, Stacking 三者区别

- 集成学习的基本步骤：

a. 构造**误差相互独立的**基分类器：

i. 基分类器的选择：

基分类器通常选择**不稳定**（对样本的差异性敏感，并且拥有较高的随机性）的分类器，如决策树和神经网络。E.g.: 随机森林选择ID3作为它的基分类器，GBDT选择CART作为它的基分类器。集成模型喜欢选择决策树（神经网络同理）作为基分类器的原因如下：

- 决策树模型简单，它的模型复杂度的调整（即表达能力和泛化能力的调整）只需要调整树的高度。
- 决策树模型容易引入随机性（例如一种经典的引入随机性的方法为在决策树**节点分裂**的时候，随机地选择一个特征子集从中找出最优分裂属性），从而容易构建不同的基分类器。
- 决策树可以较为方便地将**样本的权重**整合到训练过程中，而不需要使用
过采样的方法来调整样本权重。

而线性分类模型、逻辑回归、KNN、SVM、NB（朴素贝叶斯）等**稳定的学习器**本身方差就小，不宜作为基分类器。甚至可能因为Bagging的采样，而导致他们在训练中更难收敛，从而增大了集成分类器的偏差。

ii. 多样性：

当模型的多样性不足（相关性太高）时，要采取一些方法来构造多样化的基分类器。因为通过**误差-分歧分解（error-ambiguity decomposition**）我们知道，个体学习器的准确性越高、多样性（此处用ambiguity 表示）越大，则集成效果越好。以下为**衡量基分类器多样性**的几种指标：

- 不合度量 (disagreement measure): $dis_{ij} = \frac{b+c}{m}$, 其中 b+c 是基分类器i，j预测不同的样本数，m是两个基分类器预测的样本总数。不和度量越大，表明两个基分类器差异越大。
- 相关系数 (correlation coefficient): $\rho_{ij} = \frac{ad-bc}{\sqrt{(a+b)(a+c)(b+d)(c+d)}}$，若基分类器i,j相互独立，则$\rho_{ij} = 0$；正相关为正，负相关为负。
- Q统计量 (Q-statistic): $Q_{ij} = \frac{ad-bc}{ad+bc}$

以下为**增强基分类器多样性**的几种方法：

- 数据样本扰动（适合于不稳定的基分类器）：可以通过自助采样（Bagging中使用）或者序列采样（Boosting中使用）
- 输入属性扰动（适合于稳定的基分类器、冗余属性多的数据）：随机子空间算法（Random Subspace Algorithm）从初始属性集中随机选取若干属性子集作为基分类器训练的数据。
- 输出表示扰动：例如Flipping output，将部分分类0，1互换
- 算法参数扰动（适用于神经网络基分类器）：改变超参数

b. 训练基分类器：

训练基分类器的方法主要有三种：

- boosting: 每一层在训练的时候，对前一层基分类器分错的样本，给予更高的权重。例如Adaboost就是通过此方法训练。
- bagging: 对训练样本**多次采样**，并分别训练出多个**不同模型**。多次采样的目的即使得训练数据不同，从而导致模型尽可能的相互独立，这是Bagging应用的前提。例如 随机森林就是通过此方法训练。

- gradient boosting: 首先计算出当前模型在所有样本上的负梯度，然后以该值为目标训练一个新的弱分类器进行拟合并计算出该弱分类器的权重，最终实现对模型的更新。例如GBDT就是通过此方法训练。

  Gradient Boosting和Gradient Descent的区别：

  在梯度下降中，模型是以参数化形式表示，从而模型的更新等价于参数的更新。而在梯度提升中，模型并不需要进行参数化表示，而是直接定义在函数空间F中（前者是定义在参数空间W中），从而**大大扩展了可以使用的模型种类**。

c. 集成基分类器的**结果**：合并基分类器的结果的方法主要有三种：stacking、voting、averaging。averaging（e.g. 简单平均、加权平均（Boosting默认集成方法））用于Boosting，voting（majority voting(绝对多数投票)、plurality voting (相对多数投票)、weighted voting (加权投票) ）用于Bagging。stacking为学习法中的一种，通过Boosting的思想，将训练得到的多个基分类器，用初级分类器的输出作为次级分类器的输入，以此类推，最终一级分类器的结果即合并的结果。

- GBDT
  - 什么是GBDT：Gradient Boost Decision Tree，是Boosting模型中的一种，采用Gradient Boost方法进行训练--> 在每一轮的迭代中，首先获取当前基分类器在**所有样本上的负梯度**，依据负梯度构建一个新的基分类器并训练，进入下一轮。
  - Pros and Cons:

    Pros: 1. 训练速度快；2. 在分布稠密的数据集上模型表现优异；3. 采用决策树作为基分类器，不需要对原始的数据做归一化处理，并且具有较好的解释性（决策树能自动提取高级的特征）和鲁棒性。

    Cons: 1. 在分布稀疏的数据集上表现差；2. Boosting算法，需要依次得到基分类器，无法并行。（在决策树内部采用一些局部并行的手段提高训练速度。）

- XGBoost
  - 什么是XGBoost: GBDT的工程化实现，加入了算法的优化和实现上的优化。
  - 与GBDT的区别：

    i. **显式地加入了正则项**

    ii. **同时使用一阶和二阶导数**：GBDT在模型训练时只使用了cost function的一阶导数信息，XGBoost对代价函数进行二阶泰勒展开。

    iii. **支持多种类型的基分类器**：比如线性分类器。

    iv. **支持对数据进行采样**：传统的GBDT在每轮迭代时使用全部的数据。

v. **自动学习出缺失值的处理策略**：传统的GBDT没有设计对缺失值进行处理。

深度学习

I. NLP

总体框架：

> 理解词语 --> 理解句子 --> 注意力机制 --> 预训练语言模型

1.

NLP领域常见的预训练语言模型有哪些？说说他们的特征和区别？

- Word2vec: 一种word embedding模型，原理为将word映射到一个高维的空间中，使得此空间中的word向量直接的关系符合我们真实世界word的语义关系：(1). 相似的word的向量的余弦相似度大于不相似的word向量；(2). 使用频率较高的word的向量在空间中接近原点，使用频率低的，专业的word的向量在空间中远离原点。用一句话概述：**挑一个要预测的词，来学习这个词前后文中词语和预测词的关系**。主要有两种实现方式：
    - CBOW：Continuous Bag of Words, 通过一个固定大小的窗口滑动选取document中的连续片段，使用窗口中的周围词预测中心词。网络结构由：(1). Input layer, 由words的one-hot编码形式作为输入；(2). Projection layer, 将周围词映射后**求和取平均**，得到中心词的embedding向量；(3). Output layer, 使用softmax() 函数将中心词的embedding向量还原回原始的vocabulary的维度，通过选出概率最大的词作为中心词的预测。
    - Skip-Gram: Skip-Gram使用窗口的中心词预测周围词。 Skip-Gram 相比 CBOW **最大的不同**就是剔除掉了中间的那个 SUM 求和的过程。因为将词向量求和的这个过程不太符合直观的逻辑：不知道这加出来的到底代表着是一个句向量还是一个另词向量，求和是一种粗暴的类型转换。第二个不同就是对于每个窗口选取的单词，CBOW使用k个周围词对中心词调整一次；而Skip-Gram使用中心词对k个周围词进行预测，因此调整了k次。所以Skip-Gram更适用于样本数据量较少的情况，预测的词向量相对CBOW要准确。

这里第一个要注意的点是softmax的分母要求归一化，因此需要遍历所有的output units，耗时+计算量大。解决方法为使用Negative Sampling。`nce_loss`（noise-contrastive estimation）能够大大加速softmax求loss的方式，它不关心所有词汇loss， 而是**抽样选取几个词汇**用计算和反向传递loss。

第二个要注意的点就是Word2vec不能处理一词多义的情况。它是一种静态的embedding，没有考虑上下文信息。

第三个要注意的点就是实际的实现过程中包含的优化的细节:

a. Training Samples: CBOW的training sample为（周围词，中心词），Skip-Gram的training sample为 k（*中心词，周围词），k代表2* skip_window的大小。以Skip-Gram为例，有许多的高频词（e.g. the,a...）都是"无效的"，因此选取这些词作为training sample的中心词对模型预测精确度的提高没有多少帮助。这里采用高频词采样技术：

$$p(w_i) = \frac{1e^{-3}}{f(w_i)}(\sqrt{\frac{f(w_i)}{1e^{-3}}} + 1)$$

where $f(w_i)$ is the frequency of word i and $p(w_i)$ is the probalility that word i is chosen as the center word.

b. Negative Samping:

$$p(w_i) = \frac{f(w_i)^{0.75}}{\sum_{j=0}^{n} f(w_j)^{0.75}}$$

可以看到若一个单词的频率越高，则这个单词对应的神经元单元被选中作为`nce_loss`中计算和反向传播Loss的概率就越高。

- ELMo: Word2vec + 2*(Bi-LSTM layer)
- GPT （Open AI）: Word2vec + several Transformer decoder layers + fine-tuning，

  使用迁移学习的方法，在与训练的Word2vec上加入几层预训练的Transformer decoder layers ，并使用Input documents进行fine-tuning。
- BERT （Google AI）: Word2vec + several Transformer encoder layers + fine-tuning，
- XLnet:

2.

理解词语方面的文本模型有哪些？

- Bag of Word：词袋法。使用文章中每个词的频率大小来近似代表这篇文章。没有考虑词性、上下文关系和一词多义的情况。

- n-Gram: 词袋法的改进，若使用单个词的频率表示不能很好的体现一篇文章的特征，那么就用词组来表示。增加了一些多样性和一定的上下文关系。
- 主题模型: 通常采用LDA等概率图模型，将原始的 `文档-词语矩阵` 分解为 `文档-主题 + 主题-词语` 矩阵，可以从中提取额外的信息，如一篇文章中有多少主题，主题相对于的词汇是什么样的等信息。
- Word embedding 模型: 应用最广泛也最精确的文本模型，Skip-Gram 和 CBOW是其中的代表。著名的还有ELMOT等模型。

3.

Transformer:

- 《Attention is all you need》这篇文章提出，是一种多重注意力机制。传统的seq2seq的注意力机制是在embedding features之上加的注意力，相当于一个人对文章（Input sentences）通读了一遍之后再着重去关注一些点。而Transformer的注意力机制是在Input data 上直接加注意力，通过多次的局部注意力来加深模型的理解，相当于请多个人（多头注意力）看多次（多层的注意力）这篇文章。Multi-head attention的引入是因为采用多次不同的注意力可以得到一个多样化、全面的理解，“三个臭皮匠顶个诸葛亮“。多层注意力的引入是因为在本次理解的基础上再做进一步的理解，可能提取到不同的关注点，就像人对一样事物多看几遍每一遍都有不同的理解。

  为什么传统的seq2seq的注意力机制不采用多层的注意力呢？因为采用LSTM等RNN结构实现的注意力在多层叠加后变得十分的缓慢，而Transformer采用CNN类的结构，可以通过GPU提高并行度，因此可以提升计算的速度。

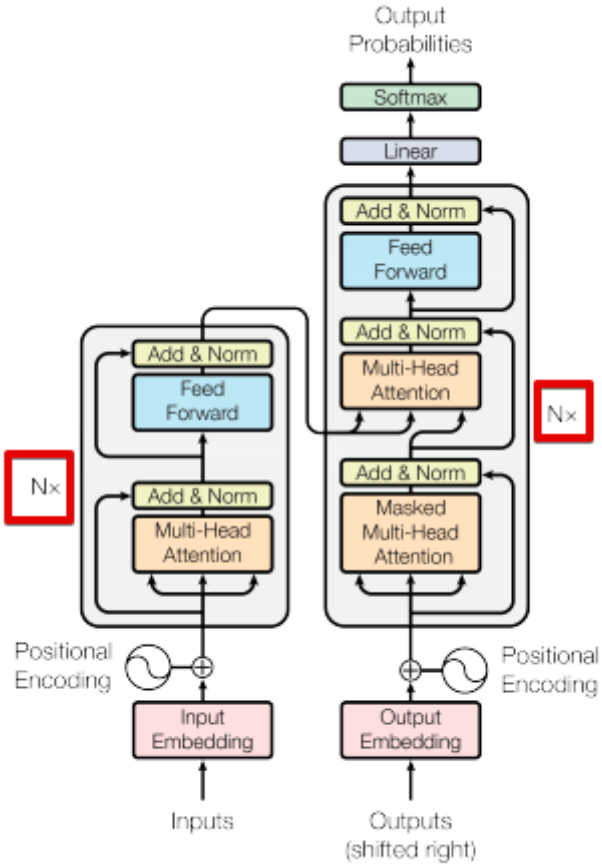
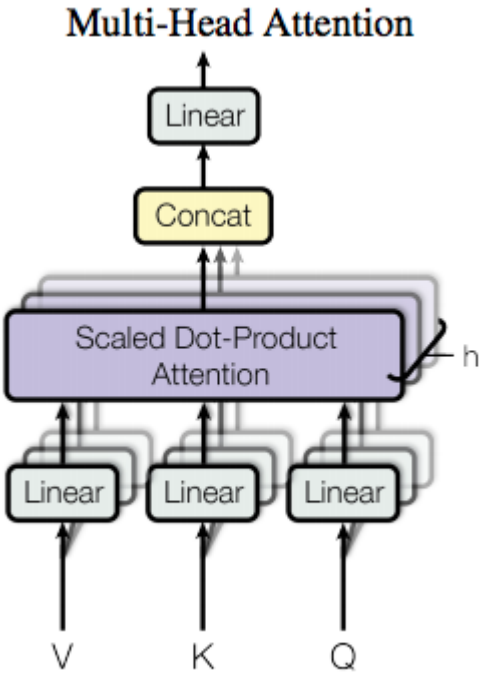
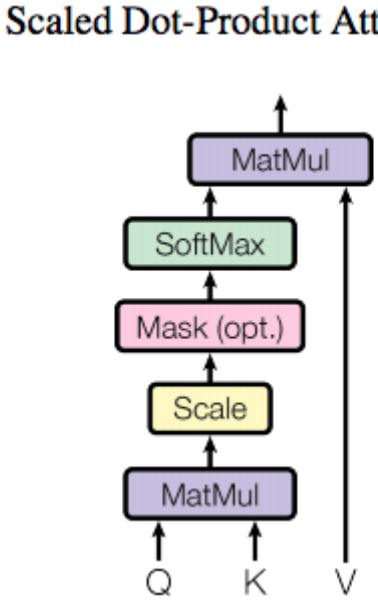


Figure 1: The Transformer - model architecture.



- 单次的注意力机制采用Q,K,V三个值来实现。Q即query，相当于用户内心想要关注的点（目标）；K即key，相当于局部的特征，我拿着目标和数据的特征进行匹配，找到匹配度最高的部分，然后用V（value）表示我对这个部分的关注程度。



- 模型的架构仍是采用Seq2seq的架构，transformer的encoder 对输入进行了self-attention（也就是它的attention没有涉及到外部的信息，相比encoder，decoder的attention所用的K,V就是来源于encoder）之后，得到了V,K并将其传递给decoder。decoder 使用attention机制对输入进行多层的注意力，此后得到输出。这里encoder和decoder的输入经过Input embedding layer之后都加入了`positional embedding`的信息，因为传统的RNN-based model能提取前后文关系的信息，而Transformer是CNN-based，因此需要从架构上加入positional embedding的信息。此后的BERT,GPT等大的模型能自动的学习时许的信息，不用显示的加入Positional embedding 信息。

II. GAN

1.

GAN的基本思想和训练过程

GAN的基本思想：

生成器(generator)希望生成的样本与真是数据集采样得到的样本尽可能一致，使得判别器(discriminator)分辨不出来；判别器尽量想分辨出来自于真是数据集采样得到的正样本和来自于生成器产生的负样本。在对抗的过程中，生成器与判别器相互促进，最终达到一个纳什均衡点。

GAN的本质：

缩小生成器产生的样本分布p_g与真实的数据分布p_r之间的JS divergence（Jensen Shannon divergence）。判别器做的就是衡量两个分布之间的JS divergence；生成器做的就是缩小两个分布之间的JS divergence。JS divergence定义如下：

$$JS(P_r||P_g) = \frac{1}{2}[KL(P_r|\frac{P_r+P_g}{2}) + KL(P_g|\frac{P_r+P_g}{2})]$$

Note: 具体证明过程见GAN的值函数推导

GAN的训练过程：

生成器和判别器交替优化。优化生成器：固定判别器D，将生成器产生的样本输入判别器得到预测的结果，将其与样本标签计算loss，通过误差的反向传播更新生成器网络。优化判别器：本质上为一个二分类问题，先固定生成器G，将生成器随机产生的负样本与从真是数据集采样得到的正样本一起输入判别器，由于我们知道正负样本的标签，通过计算二分类的误差并将其反向传播更新判别器网络。

2.

GAN的提出原因

GAN的提出原因是为了解决概率生成模型的估计问题。我们知道概率生成模型通过定义概率分布表达式$P(X|Z,\theta)$（引入隐变量求解为通常的做法），对此做最大似然估计，最大化估计的概率分布与真实的数据分布之间的相似性。当随机变量很多时，概率生成模型需要**复杂的概率推断来得到概率分布函数的估计**。而GAN并不直接对概率分布P（X）进行建模，而是通过制造样本来间接的体现概率分布。

GAN如何避免复杂的概率推断计算：当随机变量X、Z之间满足映射关系$X = f(Z)$，它们的概率分布之间也满足一定的关系：$p_x = Jp_z$，其中J为Jaccobi matrix (即$\frac{df(Z)}{dX}$)。因此若我们知道生成样本的概率分布p_z和映射函数$f(.)$（通常用神经网络表示），我们能通过上述式子间接的表达概率分布。

3.

GAN的值函数

判别器要解决的为一个二分类问题，因此使用Cross-Entropy Loss并加以转换得到值函数:

$$L(D) = -\int [p(r|x)log(D(x)) + p(g|x)log(1-D(x))]p(x)dx$$
$$p(x) = p(r)p(x|r) + p(g)p(x|g)$$
$$L(D) = -\frac{1}{2}(E_{x\in p(x|r)}log(D(x)) + E_{x\in p(x|g)}log(1-D(x)))$$
$$V(G,D) = E_{x\in p(x|r)}log(D(x)) + E_{x\in p(x|g)}log(1-D(x))$$

GAN的优化目标:

$$\min_G \max_D \{V(G,D)\}$$

- 当给定生成器G^*时，求解GAN的优化目标得：

  $$D_G^* = \frac{p_r}{p_r + p_g}$$
  $$\min_G V(G,D) = \min_G \{-log4 + 2 * JS(p_r||p_g)\}$$

  因此优化生成器的本质就是最小化$JS(p_r||p_g)$。

- 当给定判别器D^*时，求解GAN的优化目标得：

  $$arg\min_G V(G,D^*) = arg\min_G \{KL(p_g|\frac{p_r + p_g^{'}}{2}) - KL(p_g|p_g^{'})\}$$

  因此优化生成器的过程也可以解释为尽量让现在的生成数据分布p_g远离前一步的数据分布$p_g^{'}$（第二项）的同时靠近分布$\frac{p_r+p_g^{'}}{2}$。

4.

GAN在实际训练的过程中遇到的问题

通常我们自己做的GAN的结果同论文相差甚远，其中一个主要的原因是由于GAN的自身对抗机制，生成器和判别器其中一方的微小变化都可能引发蝴蝶效应，被二者的迭代过程不断放大。另外有两个原因也会引起GAN网络的不良表现：

- **优化饱和**：早期阶段生成器的参数被随机初始化，此时生成器很差，生成的样本很容易被判别器分辨，因此此时的loss很小，反向梯度传播更新的幅度很小（甚至没有），达不到训练的目的（生成器和判别器都依据loss进行优化）。

  解决方案：

  将$E_{x\in p(x|g)}[log(1-D(G(z;\theta)))]$变成$E_{x\in p(x|g)}[log(D(G(z;\theta)))]$，这样在计算$\Delta log(D(G(z;\theta))) = (1-D(G(z;\theta))\Delta_o(G(z;\theta)) = \Delta_o(G(z;\theta)) \neq 0$，梯度不为0，可以正常更新

- **坍缩模式**：训练生成器基于JS距离，而高维空间中不是每点都能表达一个样本，空间大部分是多余的，**真实数据蜷缩在低维子空间的流形上**（即高维曲面）。因为维度低，真实的数据所占空间体积几乎为零。由于高维空间中绝大部分地方见不到真实数据（概率上测度为0），因此JS距离不变，对生成器的梯度为零。而训练神经网络是基于梯度下降的，梯度为0使得网络无法更新，因此训练也无法正常进行。

  解决方案：

  使用Wasserstein距离（也称Earth Mover Distance）。当生成器分布随参数θ变化而连续变化时，生成器分布与真实分布的Wasserstein距离也随θ变化而**连续变化，并且几乎处处可导**，而JS距离不保证随θ变化而连续变化。

5.

GANs

GAN一经提出就热度不减，研究人员基于GAN的思想开发出几种著名的模型，其中包括WGAN（Wasserstein GAN）,DCGAN（Deep Convolution GAN）, IRGAN, SeqGAN, ALI等。下面是这些GANs模型的介绍：

- WGAN：引入Wasserstein distance 代替JS divergence，保证了当模型的参数变化时度量距离连续变化且几乎处处可导。它的好处就是**不管真实分布藏在哪个低维子空间里，生成器都能感知它在哪**，因为生成器只要将自身分布稍做变化，就会改变它到真实分布的推土机距离；而JS距离是不敏感的，无论生成器怎么变化，JS距离都是一个常数。因此使用Wasserstein distance 保证了模型能锁定低维子空间中的真实数据分布。

  不过Wasserstein distance涉及最优化计算，十分繁琐，因此WGAN利用它的对偶式来解决。

  $$\min_G \max_D \{E_{x\in p_r} f(x) - E_{x\in p_g} f(x)\}$$

  对比原先的目标函数：

  $$\min_G \max_D V(G,D) = \min_G \max_D \{E_{x\in p(x|r)} log(D(x)) + E_{x\in p(x|g)} log(1-D(x))\}$$

  可以看出区别在于：1. WGAN中没有了$log(.)$，2. $f(.)$为1-Lipschitz 而 $D(.)$为sigmoid。sigmoid函数有天然的值界限$y \in$(0,1)，1-Lipschitz函数限制的是导数的界， WGAN通过限制每个权重矩阵的大小来约束网络输出对输入的梯度（梯度主要由线性变换的权重矩阵所贡献）。

  WGAN使用Critic 对输入的样本进行评分（原始使用判别器对输入样本进行二分类），越像真实样本分数越高，否则越低。通过Critic的输出与样本标签计算的loss反向传播更新生成器和Critic。这里生成器的作用是缩小Wasserstein distance而Critic的作用是计算生成器分布与真实分布的Wasserstein距离。

- DCGAN：将CNN加入到GAN中，用于图像生成。在DeepLearning中，图像生成有两种方法：VAE和DCGAN。其中VAE产生的Laten Space 具有良好的空间结构和空间连续性，适合用概念向量表示空间中的一个变化方向，通过操纵概念向量对图像生成进行编辑，然而它生成的图像比较模糊。而DCGAN适合用于生成逼真的图像，然而它没有良好的空间结构和空间连续性，并且GAN的生成存在大量的陷阱和大量的技巧（这些技巧大多数看上去像艺术而并非科学），难训练到收敛。下面给出DCGAN中使用的一些技巧：
    - 使用`tanh()`作为生成器的最后一层激活函数（而非sigmoid）
    - 使用`Gaussian Distribution`对latent space进行采样，得到生成器的输入
    - 在生成器、判别器中使用`drop out layer/batch norm layer`。引入随机性使得模型更加鲁棒，并且GAN是一个动态训练的过程，常常会因为各种方式卡住，引入随机性有助于破坏这种现象。生成模型越深，越需要Batchnorm层，否则训练不充分，极易出现模型坍塌问题。
    - 判别器中使用`Leaky Relu`代替 relu，生成器和判别器中使用`步进卷积`代替Max pooling layer和。 最大限度的留住位置信息，而Max pooling layer 是一个降采样的过程，通过丢失细节信息来保留高级语义（即分类相关信息），这对生成图像来说是一个糟糕的性质。
    - 当使用`步进的Conv2D layer`和 `Conv2DTranspose layer`时，使用的卷积核大小要能够被步进大小整除（如步进为2，则卷积核大小则为4，6等）。 避免棋盘状伪影。
    - 生成器中使用`relu()`作为激活函数避免梯度饱和。
- ALI：ALI模型的目标是将VAE和GAN联系起来，同时具备速度快（VAE优点）、质量好（GAN优点），而且能有效推断。ALI使用了VAE的实现架构和GAN的训练方法。实现框架提供了推断方法，训练方法提供了高质量的生成过程。与VAE的实现架构相比，ALI虽然也有Encoder和Decoder，但它们却是独立工作的，这与VAE有着巨大的差别，它们各自生成联合分布，交由判别器判定是否相同分布，期间映射和采样都是独立进行的。而ALI的价值函数（Value Function）是直接从GAN中继承过来的。

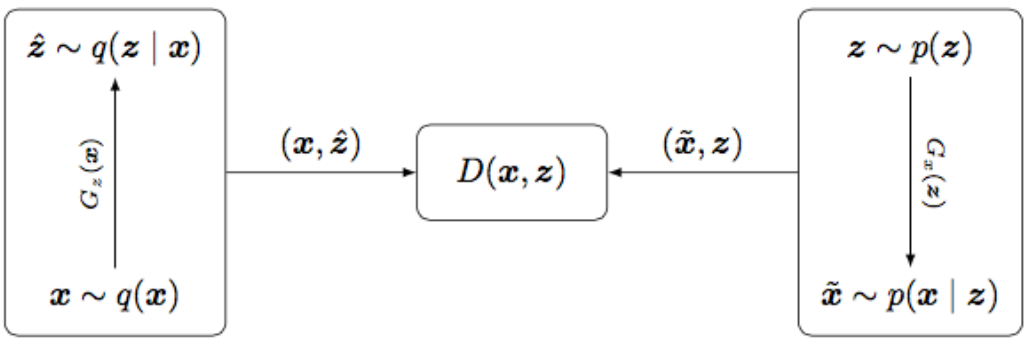


Figure 1: The adversarially learned inference (ALI) game.

Note: https://blog.csdn.net/StreamRock/article/details/81905648

- BigGAN： https://blog.csdn.net/c9Yv2cf9I06K2A9E/article/details/83026880?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-1.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-1.control
- VQ-VAE： https://blog.csdn.net/yH0VLDe8VG8ep9VGe/article/details/90989328?utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-14.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromMachineLearnPai2%7Edefault-14.control

III. Deep Learning Basics

1.

Hinge Loss: (铰链损失)

$$L(y,t) = max\{0, 1 - ty\}$$

其中t代表ground truth, y代表模型预测分类。若模型预测正确，则t,y同号，当|y|>1时Hinge loss =0；当模型预测错误时，t,y异号，此时Hinge loss 与|y|呈线性关系。在机器学习中，**铰链损失**是一个用于训练分类器的损失函数。铰链损失被用于“最大间格分类”，因此非常适合用于SVM。

2，

Spectral norm: (光谱规范)

The spectral norm is the maximum singular value σ_0 of a matrix. Intuitively, you can think of it as the maximum 'scale', by which the matrix can 'stretch' a vector. This is specifically important since **noise will be amplified by this value**. 因此为了降低噪声的影响，提高模型的稳定性，通常对变换矩阵W做光谱规范（spectral norm）。