**《介绍》**

python学习笔记：<https://github.com/TWK2022/notebook>

查看简易目录：工具栏->视图->导航窗格

快速搜索：CTRL+F

**《环境》**

**显卡驱动**

下载地址：<https://www.nvidia.cn/Download/index.aspx?lang=cn>

显卡安装的版本要与显卡型号对应

nvidia-smi：查看显卡版本、支持的最大cuda版本(不是当前cuda版本)、GPU情况

**cuda**

下载地址：<https://developer.nvidia.com/cuda-toolkit-archive>

GPU驱动程序。cuda安装的版本要与显卡驱动对应(nvidia-smi查看，显卡驱动版本向下兼容https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html)。cuda可以安装多个版本并切换，安装和卸载最好搜索一下教程。windows安装可以放到非C盘并创建子文件，安装包下载后要解压，注意解压位置要单独创建不要有其他文件，因为cuda安装完后整个文件夹会全部自动删除

完整的cuda包括cunda驱动和cuda工具cuda toolkit，不需要cuda toolkit也能使用cuda，但无法使用nvcc -V。官网下载cuda会自动带上cuda toolkit

nvcc -V：查看cuda版本

**cudnn**

下载地址：https://developer.nvidia.com/rdp/cudnn-archive

神经网络加速库。cudnn安装的版本要与cuda对应。下载的cudnn压缩包中的bin、include、lib(linux版没有bin)是所需要的文件。windows复制放到cuda的development中的bin、include、lib中。linux同样放到对应的cuda文件下，如果使用时提示没找到dll，则使用[sudo ldconfig lib位置]添加到环境变量中。卸载时删除其中cudnn的文件即可

**conda**

conda是适用于任何语言的软件包、依赖项和环境管理系统。安装miniconda3或anaconda3会自带conda。安装之后要重启

conda -V：查看conda版本

conda info -e：查看所有环境

conda activate 环境名：进入环境

conda create -n 环境名称 python=3.X：创建环境

conda env remove -n 环境：删除环境

conda install python==3.X：更换python版本

conda clean -a：删除从未使用的库、索引缓存、tar包

**Miniconda3**

下载网址：https://mirrors.tuna.tsinghua.edu.cn/anaconda/miniconda/

比Anaconda3小很多，相当于简化版的Anaconda3

linux安装时使用'sh 安装包位置.sh'安装，之后一直按回车，再输入yes。安装后需要重启服务器才能使用conda

linux 子用户安装tmux、conda等库时，如果安装后显示识别不到，可以使用source ~/.bashrc命令运行.bashrc配置文件

**Anaconda3**

下载网址：https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/

python环境管理软件，通常python的库和包都安装在Anaconda3下，使用编程软件如pycharm时直接选用Anaconda3中的python.exe

**Anaconda3 Jupyter Notebook**

Jupyter打开后会在网页中编辑(pycharm中也可以编辑)。Jupyter的特点是运行过的函数会一直存在，因此可以把程序叉开来运行，适用于小的demo展示。CTRL+P导出为PDF

Anaconda3\Lib\site-packages:安装的库位置

Jupyter:后缀.ipynb

Pycharm:文件路径和Anaconda3一致，Python3路径使用Anaconda3的

Jupyter中使用history查看所有运行记录

Jupyter中cell>>All Output>>Clear只清楚记录

conda--version:查看anaconda版本

XX.\_\_version\_\_:查看版本

XX.\_\_path/file\_\_:查看文件位置(file位置更细)

**Anaconda Prompt**

anaconda版的终端命令框

python:查看python版本并进入python编译。exit()/CTRL+C:退出环境

pip/conda --version:查看版本

pip/conda list：查看所有安装的包

**Pycharm**

环境推荐使用anaconda3中的环境，在文件->设置->项目->python interpreter中选择anaconda3下的python.exe

pycharm使用久后可能会产生几个G的文件在用户/AppData/Local/JetBrains中

左下的terminal是pycharm的命令终端

简单的语法错误会显示红色波浪线，拼写函数时会智能的跳出存在的函数

代码左侧打上断点可调式，右键可拖动断，运行到断点处时并未执行程序

在pycharm中更改文件名时可以启用引用搜索，同步更改代码相关文件名

pycharm中VCS菜单->git->clone复制远程库地址可以克隆项目

pycharm每次启动时都会更新环境和库的路径信息

ctrl+altl：使整个页面的程序格式规范。Pycharm有规范的代码格式，格式不规范时会出现波浪线(但不影响运行)。可能与QQ的快捷键相冲突

ctrl+alt++/-：展开或折叠所选代码

ctrl+f：在当前文档中搜寻关键词，可按↑↓按钮快速寻找

ctrl+b：快速转到函数所在位置

terminal终端：等同于cmd。如果有时安装库后无法识别，需要重启pycharm

**vscode**

vscode的优点是比pycharm更小更快，对其他语言如c++支持更好。缺点是操作界面、画图界面等没有pycharm好用

环境配置：在插件中下载python插件，会自动搜索并选择anaconda中已有的环境。进入项目后，可以在右下角查看并更改环境

Chinese插件：中文翻译插件

Remote-SSH插件：ssh远程连接服务器插件。安装后需要配置一下服务器的信息

PyCharm Theme 插件：将vscode风格换成pycharm

JetBrains IDE Keymap插件：将vscode快捷键换成pycharm

**pip**

系统一般默认已安装pip，用于管理和维护python包。当程序中调用库时可能无法更换库。更换库版本时需要先卸载原版本或指定版本

pip安装包下载：<https://pypi.org/project/pip/>

解压后从命令终端进入文件夹使用python setup.py install，删除安装包

pip --version：查看pip版本

pip install 库(==版本号)：安装库。会自动寻找和系统对应的型号，不指定版本时安装最新版本。已安装该库后，指定版本再次安装可更换版本

(-i https://pypi.doubanio.com/simple：添加豆瓣源)

(-i https://pypi.tuna.tsinghua.edu.cn/simple：添加清华源)

pip install --U 库：更新库

pip uninstall 库：删除库(可以删除pip库)

如果直接pip安装失败，可以去官网下载要安装的版本的.whl文件到本地，然后使用pip install 文件位置.whl。pypi网站中可以搜到很多库：<https://pypi.org/>

windows上使用VPN时可能无法使用pip命令

pip install git+https://github.com/作者名/项目名.git：等同于git clone ...+cd 项目+python setup.py install --user。setup.py用于安装环境和包，其中包括一些不能直接pip install安装的特殊的包

pip install numpy pandas matplotlib tqdm pyyaml wandb opencv-python albumentations pycocotools scikit-learn onnxruntime-gpu onnx-simplifier -i https://pypi.tuna.tsinghua.edu.cn/simple

**cmake**

安装包：<https://cmake.org/download/>

windows安装：下载.msi安装包直接安装

cmake用来安装c++编写的库，需要有GCC等编译器环境，比如安装vs2022或MinGW

**c++**

python中的很多库都是有c++写的，需要c++环境安装和运行

c++标准：c++语言规范，国际标准化组织制定，c++标准和编译器都在不断更新

c++库安装：由于c++的编译器没有统一的标准，导致很多库不通用，没有像python的conda一样的库管理工具，要手动下载和管理c++的库

编译器：c++语言需要编译器转换为对应系统的可执行文件，比如windows上编译后为.exe文件。常见的编译器有GCC、Clang、MSVS等。常用的编译器为GCC，支持多种系统和最新c++标准。windows上可以用vs2022(自带环境)或下载MinGW开发工具(包含GCC编译器)来配置c++环境，linux上通常自带GCC编译环境

GCC：GCC编译器有gcc、g++两种，gcc针对c，g++针对c++

cmake：用来安装c++编写的库，需要有GCC等编译器环境

编译模式：编译器会将代码编译为可执行语言，有debug编译和release编译两种方式。前者包含调试信息，后者是经过大量优化的最终版本

静态库：.a或.lib。程序使用静态库时，会复制静态库的内容到系统上。优点是效率高，跨系统时不依赖环境；缺点是占用空间大，每次调试代码时要重新编译静态库

动态库：.so或.dll。动态库存在于程序外部，供程序调用

**《python基础》**

conda install python==3.X：更换python版本

**基础知识**

python文件运行：.py→python解释器→.pyc。再次运行时若源代码没变python解释器直接运行.pyc，若源代码改变先生产.pyc再运行

python效率：python文件也可以编译为exe文件，会提高运行效率，但依然是使用内嵌解释器的方式运行，相比c++编译为机器码的运行效率依然低。python文件还可以转换为c++或机器码，这样效率最高。但涉及深度学习等矩阵运算时，c++的eigen、numcpp和python的numpy库相差不大，单纯矩阵计算eigen>numcpp>numpy，动态矩阵计算(矩阵大小经常改变)numpy>numcpp>eigen

python中所有None共用一个内存地址，用if A is None来判断是否为None

python是进程安全的，全局变量在多进程中修改不影响

python中有一些自动优化的算法，比如B=A或A传入函数为B时，B只是引用A，只有对B做改变时，才会额外创造一份B的内存。但注意有的库中的数据会同时改变

python库有串行计算和并行计算。速度通常GPU并行>CPU并行>CPU串行

部分python库是由c++写的，只是可以通过python调用，库中的python文件的函数只是个占用的空壳，只有个pass

python的部分库需要c++环境，可以安装visual studio，安装时只需选C++桌面开发

下标从0开始，A[2]为A第三个元素，A[0:2]表示A[0]和A[1]，左闭右开

若列表list\_=[1,2,3]，可以直接用a,b,c=list\_进行对应取值，numpy数组等其他数据结构同理

不需要定义常量或变量，单双引号没有区别。如果字符串中有双引号，外面可为单引号

句子结尾不需加';'，函数结尾加':'

变量自带内置属性(不同类型的变量自带属性不一样)，在Pycharm中使用调试时可以看到，比如tensor.device：查看张量所在的设备

**导入库**

import 库：导入库。部分库在运行时会在用户目录下生成'.库名'的临时文件，甚至下载一些模型文件

import 库.函数 as A：之后A就代表[库.函数]

from 库 import 函数：调用库中的某函数

运行路径：导入库时会从sys.path路径和运行代码的目录为起点寻找函数。假设A目录下有文件a.py、B目录/b.py、B目录/c.py。运行from时会以其实路径为基础，比如在b.py中可以使用[from c import 函数]。但在a.py中调用b.py文件时，b.py中的[from c import 函数]会报错，要改为[from .c import 函数]，'.'代表相对路径。如果让代码同时可以直接运行、被上级调用，可以在\_\_init\_\_.py文件中用sys.path.append添加B目录路径

\_\_init\_\_.py文件：解决导入路径问题，将文件夹变为库。如果在B目录下有\_\_init\_\_.py文件，B目录外的代码调用B目录功能时会提前运行\_\_init\_\_.py文件。如果写入from . import b后，可以从外部使用import B，可以使用[B.b.b中函数]。要从外部import B的前提是B的根目录要在系统路径列表sys.path中，pycharm会自动将项目根目录加入，但命令行不会

\_\_all\_\_=['库名']：写在\_\_init\_\_.py中以声明哪些函数可用。不包含的函数在隐式导入[from B import \*]不可用，但显示导入[import B.函数]依然可用，因此使用\_\_all\_\_的意义更在于方便维护。不使用\_\_all\_\_时会声明所有非\_开头的私有函数

\_\_file\_\_：当前文件的绝对路径

有的库分CPU/GPU版，CPU版要小很多，但GPU版在CPU和GPU上都可以使用

**运行程序**

python 路径/文件名.py：运行python文件。会以执行命令所在的目录为根目录

python 路径/文件名.py 传入参数：部分库如argparse需要传入参数

python -O 路径/文件名.py：程序会忽略assert等测试用的语句

**日志**

日志有debug、info、warning、error、critical五种级别

debug：调试代码时使用

info： 记录一般事件信息

warning：记录告警信息

error：记录报错信息

critical：记录最严重的错误

**相关知识**

/[文件位置]：从最上级目录开始查找

./[文件位置]：从当前目录开始查找

../[文件位置]：从上级目录开始查找

[代码] # [注释内容]：为代码添加注释

'''[//内容段]'''：添加长注释

[CTRL+/]：将选中的代码段快速注释

**基本操作**

A,B,C=a,b,c：连续赋值

A,B=B,A：交换数值

a=a+1等同于a+=1。a-=1、a/=1等类似

del A：删除变量A

len(列表/数组)：得到的是第一个维度(0维)的长度

with A() as B:：相当于B=A()，只是with执行完后会自动关闭B

max/min(a,b,c)：求最大值/最小值。如果只有两个比较时，用if判断+赋值略快一点

next(可迭代对象)：获取可迭代对象的下一个值，每次使用之后都会累加。第1次使用会获取第1个值，第2次使用会获取第2个值

raise：结束程序并提示异常。raise str\_打印错误信息；raise e抛出异常信息

exit()：结束整个python进程。exit(str\_)：结束并打印信息str\_='a=3' // exec(str\_)：执行a=3。exec比eval功能更全面

**字符串str**

定义字符串后对其中的元素更改实际上是重新定义一个新的字符串。不需要定义常量或变量。单双引号没有区别，如果字符串中有双引号，外面可为单引号

**创建**

str(A)：将A转化为字符串。A可以是数字、列表、字典等

str2=str1：不同于列表，字符串可以直接赋值

str\_='A'\*3：结果为'AAA'

'12{}34{}56'.format(A,B)：将A、B加入字符串中

'12{A}34{B}56'.format(A=A,B=B)：将A、B加入字符串中

A.join(B)：在B的每个字符间插入A组成新字符串。A为分隔符如' '，B为列表

str\_.rjust(a,'A')：在str\_左边填充A达到a长度

**取值**

字符串取值后不能直接修改，比如str\_[i]=A会报错

str\_[i]：第i个字符

str\_[a:b]：按下标截取字符串

**操作**

str\_+='A'：添加元素到字符串中

str\_.split('A')：以str\_中A划分字符返回列表；'A'不填为去掉所有空格来划分(所有类型的空格)。如果'A'在首位或A连续出现时，每个A的旁边视为一个空字符。如果字符串中有连续空格且A为空格时，会划分为空字符

str\_.strip('A')：去除字符串两边的A字符。'A'为空时可去除空格和'\n'

str\_.insert(a,'A');：在下标为a的字符前插入字符A

str\_.replace('A','B',a)：将字符串中的A替换为B，a为替换前a个匹配的A，不指定a则全替换

str\_.upper():全转化为大写。str\_.isupper()：判断字符串是否全为大写

str\_.lower()：全转化为小写。str\_.islower()：判断字符串是否全为小写

str\_.isalpha()：判断字符是否全由大小写字母组成

str\_='function(...)' // eval(str\_)：将字符串变为表达式，执行function(...)

**元组tuple**

元组是长度固定、元素类型不需相同的特殊数组，元素不可变，元组定义后不可再对其中的元素修改。改变元素的值实际上是重新定义了一个新的元组。静态的元组比动态的列表初始化速度更快、占用空间更少，在确定是常量时，尽量用元组代替列表

tuple\_=(1,2,'A')：创建元组。可不加括号。如果是单独数字要加逗号区分如(1,)或1,

tuple1=(1,) // tuple2=(2,) // tuple3=tuple1+tuple2：元组相加，tuple3=(1,2)

tuple1=(1,) // tuple2=tuple1\*3：复制以创建元素，tuple2=(1,1,1)

**列表list**

列表是动态数组，内存中不连续存放，相比元组读写效率较低、占内存大。列表中存放的是对应数值的内存地址。列表中可以存放元组、字典、np数组、tensor张量等

创建N个元素的列表实际会分配N+1个元素的内存，第一个元素存储列表长度等信息。使用list\_.append()会按一定规律扩充列表的内存空间(之后使用append时如果内存够将不再扩充，直到不够时会重复扩充列表)。使用list\_.append()后内存地址不变，内存大小可能会改变

**创建**

list\_=[1,2,”A”]：创建列表

list\_=['A']\*3：结果为['A','A','A']，但此时所有元素共用内存，会同时改变

list(str\_)：把字符串拆分为单个字符并组成列表

list(dict\_)：将字典的键转化为数列表

list2=list1：只是对list1引用，列表内存地址一样，对列表更改时同步

list2=list1.copy()/list1[:]：浅层复制list1(list1只能为一级列表)。import copy // list2=copy.deepcopy(list1)：深层复制，完全复制一份新的list1

list1==list2：判断两个列表是否完全一致，返回True或False

list2=[\_ for \_ in list1]：同list1.copy()

list\_=[0 for \_in range(a)]：创建长度为a的全0列表

list\_=[[]for \_in range(a)]：创建二维列表

list\_=[[0] \* n for \_in range(a)]：创建二维列表，每个维度长度为n

**取值**

list\_[a]和list\_[a:b]：取值。如果list1是一级列表则，list2和list1不共用内存，如果是二级及以上时共用内存，会同时改变。

list\_[a:b][c:d]：相当于先执行list\_[a:b]再对其结果使用[c:d])，不能使用list\_[a:b,c:d]切片

list\_[::i]：间隔为i显示。list\_[::-1]：列表倒序

list\_[a:]：索引为a元素之后所有元素组成的列表，a超过边界时为空列表

list\_[-i]：倒数第i个元素。list\_[-i:]：最后i个元素组成的列表

**修改**

list\_[i]=a：改变列表元素的值

list\_[i:j]=[]：删除原[i:j]的值

list\_[i:j]=[a,b,c]：删除原[i:j]的值，再插入a,b,c

list\_+=[A,B]：添加元素A和B

list\_+=str\_：等同于list\_+=list(str\_)

list\_.append(A)：增加一个元素A

list\_.extend(A)：将A中每个元素分别增加进去

list\_.insert(i,A)：在列表的第i个下标处插入A

list\_.remove(A)：删除列表中元素A,如有相同元素，则只删除一个元素A

list\_.pop(a)：删除列表中a+1个元素。list\_.pop()：删除最后一项

del list\_[a]：删除列表中a+1个元素。等同于list\_.pop(a)

'A'.join(list\_)：list\_元素都为str时，将每个元素用A连接成一个str

**排序**

random.shuffle(list\_)：将元素打乱，会直接更改list\_，没有返回值

list\_.reverse()：倒序排列

list\_.sort()：按升序排列。sort函数会根据数据情况自动选择合适的排序方法

sorted(list\_,key=function,reverse=False)：排序。如果list\_是二维则排序根据一维的第一个数大小，key是根据function(list\_)的值排序,reverse=True倒序排列。sorted使用Timsort排序方法，排序稳定，当排序长度小于32/64时使用二分插入排序，大于时使用归并排序，同时使用了节约内存和加速合并的方法

sorted(list\_,key=lambda x:x[0],reverse=False)：假如列表为嵌套列表，则根据二级列表的第1个值排序。key=lambda x:(x[0],-x[1])：先按第1个值，再按第2个值的倒序

sorted(list1,key=lambda x:list2[list1.index(x)],reverse=False)：根据list2列表的值排序。list2[list1.index(x)]表示list2中对应list1的x索引的值

function1=lambda x:功能：匿名函数。使用：function1(a)时a传入到x中，并执行后面的功能

**其他**

enumerate(list\_,start=0)：将列表中元素扩充索引。enumerate([A,B])得到[(0,A),(1,B)]

list(map(int,list\_))：将列表中字符的转化为数字

list\_.count(A)：统计A在list\_中出现的次数

list\_=[function(\_) for \_ in list\_]：对list\_中每个元素进行操作

map(function,list\_)：将list\_中每个元素执行function，返回map格式。function可以为int、lambda等

**集合set**

集合set和字典dict都是基于哈希表建立的数据结构，set无序，dict有序

集合中元素不能重复,set\_={1,2,'A'}，集合中元素没有顺序之分

set\_=set('122A')：创建集合{'1','2','A'}。会打乱顺序

set\_=set([a,b,c,a])：创建集合{a,b,c}。会打乱顺序

set.pop()：去除并返回任意一个元素

set\_.add(A)增加元素A。set\_.update('AB')分别增加字符A和B。注意不可增加数字

set\_.remove(A),删除元素A,若A不存在会报错,set\_.discard(A),删除元素A，若A不存在不会报错,set\_.pop()删除第一个元素,set\_.clear()删除所有元素

set3=set1-set2求差集,|求并集,&求交集,^求非交集

set0.intersection(set1)：求两个元组的交集

set0.difference(set1)：求两个元组的差集

set0.union(set1)：求两个元组的并集

**字典dict**

字典是散列表(哈希表)，由键和值组成,字典的键不能重复,字典一般嵌套在列表里用列表操作，字典可以嵌套。python3.6之后的字典元素是有序的

**创建**

dict\_={}：创建空字典。dict\_={A:[A,1],B:2}：创建字典

dict\_={zip(list1,list2)}：创建字典，list1中元素与list2中元素一一对应

len(dict\_)：查看字典中的键数

**取值**

dict\_[A]：查询A键对应的值，若无A键则报错

dict\_[A][0]：查询A键中的第一个值

dict\_.get(A)：查询A键对应的值，若无A键则返回None

dict\_.keys()：所有关键字(特殊对象，可用list转化)。注意dict\_.keys()与字典是共用内存的，会同时改变，可用tuple/list转化一下

dict\_.values():所有值

dict\_.items()：将字典中的键和值全部列出。数据结构为dict\_items([(key1,value2),(key2,value2)...])，list(dict\_.items())将其变为列表

key,value=next(iter(dict\_.items()))：取出字典中的第一个键和值

**修改**

dict\_[A]=a：改变键A对应的值，没有A则创建一个键A并赋予值a

key in dict\_.keys()：判断字典中是否存在键，返回True或False

dict\_.pop(key)：移除字典中键为key的元素，返回值为该键对应的值

dict1.update(dict2)：添加字典dict2到dict1中，没有返回值

dict(sorted(dict\_.items(),key=lambda x:x[0]))：按键给字典排序

dict(sorted(dict\_.items(),key=lambda x:x[1]))：按值给字典排序

**数值变换**

abs(a)：求a的绝对值

round(a,0)：将a四舍五入，结果为浮点数。0为保留到个位，-1为保留到十位

**进制转换**

bin(a)：将a转换为二进制表达式。bin(5)=0b101，结果为字符串

hex(a)：将a转换为16进制表达式。hex(17)=0x11，结果为字符串

int(str\_)：将字符串转换为整型

int(str\_,a)：将其他进制转为10进制。比如16进制时用标准的0x1a或者1A都行

**ascii码**

ord('A')：字符转为ascii码。字符与ascii码对应关系：0-9:48-57，A-Z:65-90，a-z:97-122

chr('A')：ascii码转字符

**字节类型**

byte=str\_.encode('UTF-8')：将字符串转换为字节类型。若str\_='abc'，则str\_.encode('UTF-8')为b'abc'

str\_=byte.decode('UTF-8')：将字节类型转换为字符串

**输入输出**

input('提示信息')：输入一个字符串，按回车确认(回车不会输进去)

print(A,end=''):连续输出不换行

print(f'abc{A:.4f}def')：将待输出的值A插入输出信息中

print('abc{}def{3:.4f})'.format(A,B))：将A,B对应填入{}中。3:.4f表示保留3位整数和4位小数

**读写文件**

f=open(文件路径,读取模式,encoding='utf-8')：打开文件，如果文件不存在则会创建新文件。读取模式'r'读取为字符串，'rb'读取为字节类型(图片读取选用)，'w'清空重写(如果该文件不存在会创建一个空的文件)，'wb'写入字节类型数据，'a'接着最后一行写入数据；encoding='utf-8'为读取编码方式。结束要使用f.close

with open(文件路径/新文件路径,读取模式,encoding='utf-8') as f：等同于f=open()，结束之后会自动调用f.close()，如果忘记用close会占用系统资源

f.read(i)：读取i个字符内容

f.read()：读取整个文件。对于大文件不要使用

f.readline()：读取一行，包括\n。\n可加.strip()去除

[\_.strip() for \_ in f.readlines()]：将txt文件中所有行变为列表元素

f.readlines()：读取所有行到列表中，包括\n。如果最后一行什么都没有则不会读取，转行符号算作上一行的内容

f.write(str\_)：将字符串写入文件。如要转行可写入转行符'\n'；如果要写入列表，要先将列表序列化f.write(json.dumps(list))，或直接用json.dump(list,f)便捷写入。对于元素都为list的列表，使用f.write('\n'.join(label\_list)+'\n')快速写入

f.writelines(list)：将list中元素合并后一起写入。如果要把每个元素作为一行，要在每个元素后加上\n

f.truncate()：清空文件

**语句**

系统会先运行and/or前面的条件语句，如果此时已经可以得出判读结果，就不会再运行后面的条件语句了。False、None、0、[]、{}等为否；True、非零值、非空值等为是

**条件语句**

优先级not>and>or>大小比较

if [条件1] end(or) [条件2]:：条件语句。后面可以加elif、else

x=True if [条件] else False：满足条件时x=True，否则x=False

if 0<a<2<b<4:：大小比较语句可以合在一起

if A (not) in list\_:：判断A是否在list\_中

if a==b:：判断a和b的值是否相等

if a is b:：判断a和b是否共用内存

**循环语句**

for i in [0,1,2]:：遍历，i会依次为0,1,2

for (a,b,c) in [[0,1,2],[3,4,5]]:：多个值同时遍历

for i in range(10):：遍历0,1,..,9。等同于range(0,10)、range(0,10,1)。range(9,-1,-1)表示遍历9,8,...,1,0

for i,j in zip(A,B):：并行遍历

for index,value in enumerate(list\_):：同时遍历索引和值

while A<B:：while循环。ctrl+d强行结束循环

while True: // try:[内容] // except EOFError: // break：编程题不断读取输入

yield：用于循环时的取值。写在函数中替代return的位置，当for循环调用函数时才执行函数中的内容。比如用for i in result\_list时要先得到列表占用空间，如果先定义一个使用yield的函数，函数的内容是生成result\_list的一个值，就可以一边循环一边生成下一个要循环的值。如果定义了函数A，函数中先写入其他代码，再写入for循环返回yield，此时在外部使用for i in A()时，其他代码只在首轮执行一次，其他轮次只执行for中的

**异常检测**

try:... // except (错误类型,...) as e: // except:... // finally:...：异常检测。如果try中代码报错则会根据错误类型执行[except (错误类型,...) as e:]中内容，print(e)可以查看错误类型，raise e可以查看原始报错信息。如果找不到对应的[except (错误类型,...) as e:]，会执行except中内容。无论有没有报错最终都会执行finally中内容。[except (错误类型,...) as e:]和finally可不加

OSError：文件读取失败。包含FileNotFoundError文件不存在等错误

KeyError：字典的键不存在

ImportError:文件导入失败

ZeroDivisionError：除以零报错

**判断语句**

assert 条件,'提示'：检查条件是否错误，如果错误停止程序。使用[python -O 路径/文件名.py]运行程序时，会忽略assert等测试语句，因此不在生产版本的业务代码中使用

isinstance(A,type)：判断A的数据类型是否为type，返回布尔值。type为单个类型如int或集类型合如(int,str,list)，type可以为某个函数

hasattr(A,B)：判断A中是否有内置参数B，返回True或False。A.B存在返回True

**函数**

函数需先定义后调用，def需写在调用代码前面

def 函数名(参数1=默认值，参数2=默认值...):：定义一个函数。若参数没有赋值则使用默认值，如果没有return返回None。函数可以不传入参数：def 函数名():

函数参数的默认值不要使用可变对象，因此参数的默认值共用内存：比如def function(list\_=[]) // list\_.append('A')，不传入参数时会使用默认值[]，但这个默认值共用内存，当多次调用函数时，默认值就会改变为['A','A'...]

global n：让全局变量n可以在函数中使用，加在函数内部的开头。使用global n后使用n=a会同时改变外部变量n，如果先使用n=a再使用global n则会报错。实际上不使用global，但在函数内没有定义n的情况下也会使用外部变量n，只是此时使用n=a是定义函数内变量n。多层函数嵌套时global n指都的是全局变量n

nonlocal n：只能在函数内部的嵌套函数中使用。假如外层函数中定义了变量n，则嵌套函数中使用nonlocal n后可以使用n，此时n也会同步改变。注意如果外层函数中使用了global，则嵌套函数不能再使用nonlocal

**装饰函数**

def decorator(function): // def template(\*args,\*\*kwargs):... // return result // return template：定义装饰函数。与普通函数的区别是外层函数的参数为函数function，内层函数的参数为function的参数。装饰函数需要嵌套2层，第1层传入函数本身，第二层传入函数的参数，(\*args,\*\*kwargs)表示可以传入任意参数

@decorator // def function(...)：使用装饰函数。使用function(...)时，会将function本身作为参数传入到decorator得到template函数，再将function的参数传入到template中，原function函数的return忽略。装饰函数相当于模版，通常function为功能函数，template中包含一些通用的前后处理操作

**类**

类相当于把多个函数集成在一起，同一个类中的函数可以共用内部变量。通常为了方便函数管理会更多的使用类而不是直接用def函数。一个小功能的代码合为一个小类，一个大功能又将多个小类合为一个大类

类在程序运行前会提前初始化，可以写在调用代码后面

**类的定义**

类结构规范：\_\_init\_\_、@classmethod、@staticmethod、函数、\_函数(私有函数)

class 类名(继承类):：定义一个类。(继承类)可以不加。在类中可以添加内部变量、内置函数、自定义函数。类定义后即使不使用也会初始化，初始化会执行直接写入类中的内容，不会执行函数中的内容。因此通常不会在类中直接写入内容，而是写入\_\_init\_\_函数，\_\_init\_\_函数会在实例化A=类名(\_init\_\_参数)时执行，\_init\_\_参数不包括self，self就是实例化后的A本身

内部变量：在类中直接写入如value=0、function=XX等可以得到内部变量，类中可以通过self.value、self.function来使用内部变量。但通常会在内置函数\_\_init\_\_中使用self.value=0、self.function=XX来定义，因为\_\_init\_\_更灵活。内部变量还可以在实例化后使用A.value从外部添加

内置函数：定义类时自带的函数，形式如’\_\_A\_\_’，比如\_\_init\_\_、\_\_call\_\_等

自定义函数：相比单独的函数，可以通过self.XXX来使用类的内部变量和内部其他函数，因此要多传入一个参数：def A(self,...)。使用时需要先实例化类：A=类名(...)，再使用A.function(...)。代码规范中只在类内部使用的函数名称前加上’\_’如’\_A’，会在外部使用的函数不加’\_’

继承类：加入继承类A后，定义的子类B可以直接获得父类A中直接写入的内部变量、内置函数、定义的函数，但当B中的函数和A中的重名时，则只使用B的函数。省去了冗余的代码

def \_\_init\_\_(self,value): // super().\_\_init\_\_() // self.value=value：内置初始化函数。\_\_init\_\_中可以使用self.value=value来定义内部变量。\_\_init\_\_函数会在实例化时执行，super().\_\_init\_\_()可不加，加入时会执行并获得继承类\_\_init\_\_中的内容。代码规范中所有的内部变量都应该在\_\_init\_\_中定义好，即使不确定值也要赋个值占个位置

def function(self,参数):：在类中添加自定义函数。self必带且不算做传入的参数

@staticmethod // def function(参数):：静态方法函数。由于加了@staticmethod，function虽然写在类中但不需要实例化(实例化后也可以)就可以使用：类名.function(...)。此时不需要self参数，但类内部依然可以用self.XX调用，代码规范中当自定义函数不使用self.XX时要加上@staticmethod

@classmethod // def function(cls,参数):：类方法函数。与@staticmethod类似，只是多了一个cls参数，cls等同于类名，可以在函数中用A=cls(...)实例化类。通常使用@classmethod函数是需要对传入类的参数进行前处理，处理后并在函数中实例化类

**类的使用**

A=类名(\_\_init\_\_参数)：初始化类

A.value：如果\_\_init\_\_中定义了self.value，可以在类外面使用和改变内部变量

A.function(参数)：使用在类中自定义的function函数

A.function(\*\*dict)：传入字典作为函数的参数。如果函数的变量名正好与字典的键一致，可直接将键的值传入对应的参数

**常用内置函数**

\_\_slots\_\_=('value',...)：只读内部变量限制，直接写在类中。只有写在\_\_slots\_\_申明中的变量才可以从外部定义或改变，比如number变量没有申明，使用A.number会报错。\_\_slots\_\_在继承类中时，不会被子类继承

def \_\_call\_\_(self,参数):：实例化后，直接调用类本身会执行\_\_call\_\_函数：A(参数)

def \_\_str\_\_(self):：实例化后，print类本身会执行\_\_str\_\_函数，print(A)

def \_\_add\_\_(self,other):：实例化后，两个类相加会执行\_\_add\_\_函数：A+B。其中self代表A，other代表B

**print结果**

类本身：<class '\_\_main\_\_.demo'>

实例化：<\_\_main\_\_.demo object at 0x0000024A9306A990>

type(类本身)：<class 'type'>

type(实例化)：<class '\_\_main\_\_.demo'>

**其他**

**时间与内存**

python采用分离式结构动态数组

python多线程适用于阻塞式IO场景(文件读写)，不适用于并行计算场景

id(A):查看对象A的内存地址

import sys // sys.getsizeof(A)：查看对象A的内存大小

不同的对象内存地址不一样：A=[1]，B=[1]，id(A)!=id(B)。但通常每个对象中具体的某个数值在系统中只有一份：A=[1]，B=[1]，id(A[0])=id(B[0])

创建任意对象时系统会随机分配一个合适的内存地址，系统中有一个计时器记录该对象被引用的次数，每次有新的对象引用该对象，计数器+1，用del释放一个引用时-1，当垃圾回收机制发现某对象引用次数为0时将其内存删除

合并大量字符串：使用list\_.append(str\_) + 'A'.join(list\_)的方式时间更短；使用result+=str\_的方式内存占用更少

元组与列表：在确定是常量时，用元组比用列表更节省空间

列表大量追加元素：使用list\_=[\_ for \_ in range(len\_list)] + list\_[i]=a的方式比使用list\_.append()时间更短、内存占用更少

及时关闭资源：比如文件读取等操作，使用完毕后要及时加上.close()，推荐直接用[with ... as ...:]的方式打开，会自动加上.close()

**信息安全**

eval和exec：不要对不可信的表达式直接使用eval和exec。可以使用import ast // ast.literal\_eval(...)替代。ast.literal\_eval对输入表达式有限制

os.system：不要对不可信的命令直接使用os.system

SQL注入：与eval和exec类似，比如在登录密码后面加入[or '1'='1']等额外语句，如果不对输入进行校验则存在安全风险

XML拼接：对于加入XML的外部数据要进行检验，如果数据中带有一些格式性的东西会破坏XML文档，可以使用defusedxml库处理数据

ReDos攻击：使用正则表达式要检验外部输入文本的长度、减少使用过于复杂的表达式、尽量不动态构建表达式(需要加黑白名单检验)

yaml加载：pyyaml库加载不可信yaml文件时，要加上Loader=yaml.SafeLoader检验yaml文件，尽量保证pyyaml库为最新版本

json序列化：对不可信数据进行序列化和反序列化时推荐使用json库

os.chmod：如果需要修改文件权限，要紧跟在打开或创建文件后，防止中间出现风险

日志记录：不可信数据要经过处理后再记录到日志中。不要记录敏感信息，不可信数据中可能存在很长的垃圾信息

**特殊情况**

lambda公式覆盖：当用循环for i in range(4): // append(lambda x:x+i)时，列表中的lambda公式虽然内存地址不同，但公式内容会被最后一次覆盖，最终列表元素为4个lambda x:x+3

函数参数的默认值共用内存：比如def function(list\_=[]) // list\_.append('A')，正常传入参数不会有影响，如果不传入参数就会使用默认值[]，但这个默认值共用内存，当多次调用函数时，默认值就会改变为['A','A'...]

**《python常用库》**

**import logging**

实时记录日志信息，有info、debug、warning、error、critical五个等级

logging.basicConfig(filename,level=logging.WARNING,encoding,format)：设置日志记录。filename为保存位置如'log.log'；level为记录日志等级，logging.WARNING只记录warning及以上的信息；需要显示中文时设置encoding='utf-8'；format为日志的格式如'%(asctime)s | %(levelname)s | %(message)s'

logging.info(text)：记录信息，等级为info

logging.debug(text)：记录信息，等级为debug

logging.warning(text)：记录信息，等级为warning

logging.error(text)：记录信息，等级为error

logging.critical(text)：记录信息，等级为critical

**import math**

math.pi：代表pi(3.1415...)，三角函数中的角度范围为0-2pi

math.sin(a)：sin函数，math.sin(math.pi/2)=1。math.cos()、math.tan()同理

**import copy**

list2=copy.deepcopy(list1)：深度复制，完全复制一份新的

**import time**

time.time()：当前系统时间

time.sleep(1)：让代码在此等待一秒

**import tqdm**

for i in tqdm.tqdm(range(n)):：显示循环的进度条

for i,j in enumerate(tqdm.tqdm(list\_)):：显示循环的进度条

tqdm\_show=tqdm.tqdm(iterable=None,total,mininterval=0.1)：定义显示方法。iterable为可迭代对象；或者使用total指定长度；mininterval为最小显示间隔

//tqdm\_show.set\_postfix({'当前loss':loss\_batch.item()})：实时添加显示信息

//tqdm\_show.update(1)：更新进度条，进度条总长度为设定的total，1为更新的进度

list(tqdm.tqdm(多进程,total=len(长度)))：对于多进程等情况如果不显示进度条要加total参数，加list是让进度条会显示加载进度

**import random**

random.seed(999)：设置随机种子为999。之后使用random函数得到的结果会一致

random.randint(a,b)：生产a到b区间的随机整数

random.choice([...])：随机选取列表中一个数

**import sys**

sys.path：存放系统路径的列表，系统导入库函数等会遍历列表中的路径寻找

sys.path.append(path)：添加路径到系统路径中

for line in sys.stdin：可以不断读取输入(输入为一行或多行，多行时处理较为复杂)

**import os**

os.listdir(文件夹路径)：读取文件夹中所有文件的相对路径(带格式)。在windows下读取的文件路径是自动排序好的，但linux中顺序是乱的。windows中读取路径可以用r'\'，linux不能识别

os.getcwd()：当前系统所在的路径

os.path.dirname(\_\_file\_\_)：文件所在目录的位置

os.path.dirname(path)：获取文件所在目录的路径

os.path.basename(path)：除去目录只取出文件名

path,extension=os.path.splitext(path)：将路径中.后的扩展名分开

os.path.realpath(path)：转换为标准格式的路径

os.chdir(path)：更改当前系统所在的路径为path。path不存在时会报错

os.makedirs(path)：创建目录

os.chdir(path)：改变当前目录到path。注意在函数中使用后也会改变全局的路径

os.remove(path)：删除位置为path的文件

os.rename(path,更改后名称)：更改文件名

os.path.abspath('')：当前文件所在目录。'../'上级目录

os.path.join(A,B,C)：将A,B,C路径拼接起来，会自动加/

os.path.exists(path)：判断路径是否存在返回布尔值

os.path.isdir(path)：判断路径是否为目录

os.path.isfile(path)：判断路径是否为文件

os.environ：获取系统当前的环境变量，返回一个字典，键为环境变量名

os.system('cd /usr/local')：将字符串转化为命令并执行。会等待命令执行结束

os.getuid()：获取当前用户的id

**import shutil**

shutil.move(path,new\_path)：移动文件

shutil.copyfile(path, new\_path)：复制文件

**import pathlib**

pathlib.Path类似os.path的路径操作，但操作类型是WindowsPath

pathlib.Path(str\_)：将路径转换为WindowsPath类型

pathlib.Path.cwd()：当前文件所在目录(需要用str()转换为字符串类型)

pathlib.Path.home()：用户的home根目录

pathlib.Path.exists(WindowsPath1)：判断路径是否存在返回布尔值

WindowsPath1 // WindowsPath2：相当于str1+'/'+str2

WindowsPath1.mkdir(parents=False)：创建路径文件。parents=True时会依次创建路径中间缺少的文件夹

文件名.rename(新文件名可带路径)：如果不带路径则直接更改文件名，如果带路径则是移动到目标路径下再更改文件名

**import subprocess**

subprocess在os.system()的基础上实现了更多的功能

popen=subprocess.Popen('命令',shell=False,stdout=None,stderr=None)：创建一个子进程并在服务器上运行命令，不会等待子进程完成。shell=True用shell语言运行，stdout=subprocess.PIPE可以得到子进程的返回结果，stderr=subprocess.PIPE可以得到子进程的报错信息

popen.stdout.read().decode()：得到子进程的输出(字符串)，会在此等待子进程完成。结果会加上/r/n，可以用.strip()去除

popen.stderr.read().decode()：得到子进程的错误信息(字符串)，会在此等待子进程完成。没有报错时为空字符串，报错结果会加上/r/n

popen.pid：子进程号

popen.poll()：判断子进程是否结束。已结束返回0，未结束返回None

popen.wait()：会在此等子进程结束

**import multiprocessing**

进程>线程>协程：进程是一个大任务，线程是进程中并行的分任务，协程是分任务中串行的小任务，但协程可以根据实际情况在做A任务的空隙完成一点B任务。进程相比线程占更多的资源，但各进程之间不会相互影响。线程相比进程占更少的资源，但各线程之间会相互影响，一个线程出问题会导致整个进程出问题

process=multiprocessing.Process(target=function, args=(x,))：创建一个进程。function为执行的函数，args传入函数的参数。当进程创建好后，传入的变量就被固定了

process.start()：执行进程。执行后主程序不会在此等待，而是执行后面的代码

process.join()：程序会在此等待该进程执行结束，然后才执行后面的代码

**import threading**

thread=threading.Thread(target=function,args=(),kwargs={})：创建一个线程。function为执行的函数；args按顺序传入函数的参数；kwargs可以按变量名传入参数。当线程创建好后，传入的变量就被固定了

thread.start()：执行进程。执行后主程序不会在此等待，而是执行后面的代码

thread.join()：程序会在此等待该进程执行结束，然后才执行后面的代码

**from concurrent.futures import ThreadPoolExecutor**

ThreadPoolExecutor多线程与threading.Thread功能一致，但可以自动分配线程数，将参数自动划分并传入线程

def A(x,y):...：创建函数

A\_partial=functools.partial(A,y=1)：固定A函数的y变量为1。使用A\_partial函数时就不能传入y参数了。使用多线程时对于固定的参数可以使用partial固定，固定参数要在传入参数的后面

with ThreadPoolExecutor(a) as executer:：创建多线程。a为线程个数，不指定a时会自动采用最大值

//executer.map(A,list\_)：返回map对象，先将变量列表中的元素平均分配到多个线程中，在各线程中会将元素依次作为变量传入A函数

**import numpy as np**

numpy是python中最基本的库之一，串行计算但速度也较快

numpy数据在内存中连续存放，CPU中计算，默认float64和int32

numpy中比较数值大小时使用=符号时，容易因为float64计算精度问题出错

numpy中切片取值时共用内存

numpy运算时如果数组最后的维度相同会扩展，比如A.shape=(2,3)，B.shape=(3,)，A与B之间可以进行运算。但如果A.shape=(3,2)则不会扩展

numpy数组很大的情况下，使用连续取值array[screen][index]=a时，赋值可能会失败

**取值：**

如果使用a=np.array([1,2,3])，b=a，此时a和b公用内存，使用b[0]=4时a[0]和b[0]会同时改变(如果是b=1则只是b重新赋值)，可以使用b=a.copy()新建内存。

可以使用array[a:b][c:d]或array[a:d,c:d]切片

若a=array([0,1,2,3])，a[0]会直接取出结果0；a[0:1]会保留原形状，得到(1,)形状的np数组，a[[0,3]]会取第0和第3个数组成新的数组array([0,3])

array[...,a]：取出最后一维第a+1个元素

array=np.delete(array,delete\_index,axis=0)：去除索引对应的元素，需要加axis

**属性：**

array.shape：查看形状。array.dtype：查看数据类型

array.size：查看元素数。array.ndim：查看维度

array.dtype：显示数组类型如float64。

array.item()：如果取出array对象中的值

array=array.astype(np.float32)：转换np数据类型

str(array.dtype)：将array转化为字符串

type(data)==np.ndarray：判断数据是否是numpy数组，用if data会报错

**创建和转换：**

array.tolist()：将array变为列表，不能用list(array)。浮点数可能会有精度误差

array.copy()：复制一个array

np.array([[1,2],[3,4]])：创建数组([[1,2],[3,4]])

np.arange(a,b,间距)：创建从a到b的数组，末尾不一定有b。如果a>b则间距为负数

np.linspace(a,b,c)：从a起始到b结束，创建c个间隔均匀的数组

np.zeros((a,b),dtype)：创建a行b列全0数组。dtype=bool创建全为False的数组

np.ones((a,b),dtype)：创建a行b列全1数组。dtype=bool全True数组

np.full((a,b),value,dtype)：创建a行b列全value数组

**随机操作：**

np.random.seed(种子号)：设置随机种子,相同种子的随机值一致

np.random.rand(a,b)：创建a行b列随机0-1均匀分布的数组

np.random.randint(a,b,(c,d))：创建形状为(c,d)值为a到b的随机整数

np.random.choice(array,size,replace)：从array中随机抽取元素组成形状为size的数组。replace=True/False表示抽取元素是否可以相同

np.random.normal(loc=0.0,scale=1.0,size=None)：创建正态分布数组。loc为均值；scale为标准差，scale越大效果越明显，可为0.05；size为生成数组的形状

np.random.shuffle(array)：打乱array中各元素的顺序，会直接改变array中的元素，没有返回值

**基本操作：**

array[array==a]=b：将数组中元素a替换为b

bool\_opposite=~bool：布尔数组取反

array2=array1[np.newaxis]：增加维度。[np.newaxis]在0维增加；[:,np.newaxis]在1维增加；[...,np.newaxis]加在最后

np.around(array,decimals=0)：四舍五入，结果为浮点数。decimals为保留小数点后位数，0为保留到个位，-1为保留到十位

np.append(array,[...],axis)：添加元素

np.where((array<a)&/|(array>b))：返回array中满足条件的元素下标。如果array的维度为a，则返回包含a个等长数组的tuple，每个数组代表元素在每个维度的下标。np.where会并行遍历array中所有元素，效率高，常用于数组元素的筛选

np.where((array<a)&/|(array>b),A,B)：返回与array相同形状的数组。满足条件时数组元素为A，否则为B

np.argmin/argsmax(array,axis=0)：取array最小值/最大值下标的索引

np.clip(input,min,max)：将input中的值压缩到min到max之间

np.sort(array,axis=0)：根据axis维度的值从小到大排序

np.argsort(array,axis=0)：根据axis维度的值从小到大排序，返回下标

np.round(a,decimals=0)：近似数，结果为浮点数。decimals为保留小数点后位数，0为保留到个位，-1为保留到十位

np.isnan(array)：判断array是否为nan值，返回同等大小的True或False数组

np.isnan(array).any()：判断array中是否存在nan值，返回一个True或False

np.ascontiguousarray(array)：将数组的内存变为连续存储

np.linalg.norm(vector)：求向量的模长

**数据转换：**

np.frombuffer(字节类型,dtype=None)：从字节类型解码为np.array(一行)。如果是图片需要指定dtype=np.uint8

array.tostring()：将np数组(一行)转换为字节类型

**维度变换：**

array.reshape(a,b)：改变数组形状，如果array(a,b).reshape(a,b,1)表示变成三维。reshape(c,-1)表示变成c行，列数自动分配

array.flatten()：将数组变成一维

np.append(array1,array2,axis=0)：将两个数组在axis维合并

np.stack([array1,array2],axis)：在axis维度上叠加数组，增加维度

np.vstack([array1,array2])：上下堆叠数组，不增加维度

np.hstack([array1,array2])：左右堆叠数组，不增加维度

np.concatenate((array1,array2),axis)：在当前的axis维度叠加数组，不增加维度。axis=0时相当于np.vstack，axis=1时相当于np.hstack

np.delete(array,(a,b,...),axis=0)：删除axis维上下标为a...的元素

array.repeat(a,axis=0)：在axis维度复制a倍，必须设置axis

array.transpose(0,2,1)：更改维度，0,2,1表示将原来的1,2维换位

np.stack([A,B]).reshape(-1)：实现A、B数组中元素交叉

np.flip(array,axis=0)：翻转。二维数组axis=0时为上下翻转、axis=1时为左右翻转

list\_=np.meshgrid(x,y,indexing='xy')：根据坐标轴生成网格坐标，list\_中为[x轴网格坐标,y轴网格坐标]。假设x=np.arange(20)、y=np.arange(10)，则list返回为[(10,20),(10,20)]，indexing='ij'时，结果为[(20,10),(20,10)]

**计算：**

np.diff(array)：求array差分

np.argmax(array)：求array中最大元素的索引

np.square/exp(array)：求每个元素的平方/e的平方，返回数组

np.maximum/minimum(A,B)：逐位比较A,B中的每个元素，保留最大/小值

np.max/min/sum/mean/var/std(array,axis)：求axis维度的最大值/最小值/和/均值/方差/标准差(不指定axias时求所有值的)。若array形状为(2,3,4)，axis=0，则结果形状为(3,4)，若axis=1，则结果形状为(2,4)

np.unique(array)： 将array展平后去除重复的元素并从小到大排序

np.intersect1d(array0,array1)：取两个1维数组的交集

np.union1d(array0,array1)：取两个1维数组的并集

np.setdiff1d(array0,array1)：取array0-array1的差集

**三角函数：**

np.pi：代表pi(3.1415...)，三角函数中的弧度范围为0-2pi

np.sin(array)：sin函数，np.sin(np.pi/2)=1。np.cos()、np.tan()同理

np.arcsin(array)：反sin函数。np.arccos()、np.arctan()同理

np.degrees(radian)：弧度转为角度

**矩阵运算：**

array.T：转置

np.linalg.inv(array)：求逆

np.dot(a,b)：求a和b的点积(矩阵相乘)

**函数拟合：**

array=np.polyfit(x,y,deg)：用线性拟合曲线。array为多项式的系数。deg=1,2...为拟合的多项式阶次

np.polyval(array,x)：使用拟合的函数

**np格式保存：**

np.savetxt(path+name,array)：保存1维/2维数据到TXT文件

np.savez(path+name,array)：保存多维数组到np专用npz文件中

np.load(path,array)：读取npz文件(numpy专用二进制文件)

**import pandas as pd**

Pandas基于numpy库编写，切片取值时共享内存，可以使用.copy()复制

Pandas包中数据结构：序列series、数据框dataframe

Pandas将Nan和None都处理为np.nan

**序列(Series)：**

序列由标签(索引)和值组成。可以用值的下标找值，也可以用值对应的标签找值，相当于列表和字典的结合

pd.Series(dict,dtype)：将字典转化为序列。dtype为数据类型如np.float32

pd.Series([0,1],index=['A','B'],dtype)：创建series结构,可以由list或numpy创建。index=自定义标签,不指定时自动生成标签0,1,...

series.index：查看所有标签

series.values：查看所有值

series.sort\_index；按标签升序排序。标签可为字符

series.sort\_values:按值升序排序

series.loc[A]：按值的标签找值。series.loc[A:B]：按值的标签切片，左闭右闭

series.iloc[a]：按值的下标找值。series.iloc[a:b]：按值的下标切片，左闭右开

series.loc[A]/series.iloc[a]=b：按值的标签/值的下标将值改b。如果没有A标签则会创建新的标签和值

series1.add(series2,fill\_value)：按照标签相加序列。fill\_value不指定时，两个序列不一致标签的结果为NaN。fill\_value=0表示先把相互缺失的标签扩充为0再相加

series2=series1.dropna()：去除NaN值。有返回值，不共用内存

series2=series1.fillna(a)：填充NaN为a。有返回值，不共用内存

**数据框(DataFrame)：**

数据框由索引、列名、内容组成。数据框的0维度对应内容长度，1维度对应列名

DataFrame相当于把相同的Series放在一起并加上列名

DataFrame中每列元素的数据类型一致，不同列之间可以不一样

pd.DataFrame(dict,index,dtype)：将字典转化为数据框,字典的键对应列名,值对应内容。不指定index时索引从0开始自动生成，index=None不加索引。当传入包含多个键一致的字典的列表时，会依次生成多行

pd.DataFrame(array/list1,columns=list2,index=list3,dtype)：将列表/数组转为数据框。若list1是一维，则list2中只有一个列名，list3等于array长度；若array是二维，则list2列名数应等于array列数，list3等于array行数

pd.read\_csv(r'文件位置.csv',header=0,index\_col=None,encoding='utf-8',dtype=np.float64)：导入csv。前面加r是为了把文件位置中的\转化为/，否则会报错。header=0时会将第一行读取作为表头，=None时不需要表头；index\_col=None时生成新的索引，=0时将一列作为索引；encoding为解码方式，csv文件在WPS打开后可能会变为'gbk'编码

pd.read\_excel(r'文件位置.xls(xlsx)',sheet\_name):导入表格。sheet\_name指定读入的工作表，'sheet1'读取'sheet1'为df，[A,B]读取A、B为字典{A:df1,B:df2}，None读取所有的工作表为字典

df.to\_csv(r'A.csv',index=True,header=True)：保存为scv文件。index=True时会把数据框的索引作为第1列保存到csv中；header=True时会把列名保存到csv中

df.to\_excel(r'A.xls/xlsx')：保存为xls/xlsx文件

df2=df1.copy():复制一份df1给df2

df.index：索引

df.columns：列名

df.columns=[A,B,...]：更改列名

df.values：将数据结构转为numpy数组

df[列名]:选择列(结果为Series)。df[列名]=[...]：更改/添加列

df[[A,B,...]]:选择多列(结果为DataFrame)

df.iloc[a:b]：选择下标为a:b的行(结果为数据框)

df.mean(axis=0)：求axis维度的平均值，返回Series

df.sum(axis=0)：求axis维度的和，返回Series

pd.concat([df1,df2],axis=0,join='outer',ignore\_index=False)：合并df1和df2。axis=0时按行合并，列名要一致；join='outer'取并集，'inner'取交集；ignore\_index=True时合并后索引重新排序；axis=1时按列合并,索引要一致

df.T：转置，索引和列名也会互换

df.isnull().values.any()：判断是否存在Nan，返回True或False

df.drop(columns=A)：删除名为A的列

df.drop(index=0)：删除行

df.fillna(A)：将所有空白(nan)的地方填充为A

df.dropna(axis=0,how='any')：删除所有含Nan的行。axis=1时为列操作；how='all'时只删除全部为Nan的行

df.sort\_index(axis=0)：按索引从小到大排序。axis=1时按列名

df.sort\_values(by='列名')：按某一列的值从小到大排序

df.rename(columns={'A': 'B'},inplace=True)：将列名A换为B

**时间处理：**

DatetimeIndex=pd.DatetimeIndex(list/array)：创建DatetimeIndex对象。list/array为时间，可以为类似2000/12/24或2000-12-24的形式。创建后精确为秒，但使用to\_csv保存时如果分和秒都为0，只会保存精度到日

str(DatetimeIndex[i])：取出时间如'2000-12-24 00:00:00'

pd.date\_range(start,end,periods,freq='D')：创建pd时间数组。start和end为开始和结束时间；periods为总长度，使用periods时不使用end；freq为间隔默认为1天，可以为'1h20min'等其他间隔

DatetimeIndex.second/minute/hour/day/month/year/dayofweek：取出DatetimeIndex中某一时间的值

df['2000-12']：当数据库索引为DatetimeIndex对象时，可以取出该月的所有天

df.resample('M')：将数据取样为月，得到一个resample对象。在其基础上操作时，索引会变为这个月的最后一天，df.resample('MS')将索引变为这个月的第一天

df.resample('M').mean()：计算每个月的平均值，索引变为每个月的最后一天

df.resample('M').first()：得到这个月的第一天的值，但索引依然为最后一天

**import datetime**

time\_now=datetime.datetime.now()：当前系统时间(年月日时分秒)。datetime数据

time\_now.date()：当前时间的年月日。date数据

time\_now.time()：当前时间的时分秒。time数据

time\_now.weekday()：获取当日是一周中的第几天(0-6)

time\_now.total\_seconds()：将当前时间转换为秒

time=datetime.datetime.combine(time\_now.date(),time\_now.time())：合并为datetime数据

time=datetime.datetime(2000,1,1,9,30,00)：创建datetime时间数据。支持加减运算

time=datetime.time(9,30,00)：创建time时间数据。不支持加减法

**import matplotlib.pyplot as plt**

plt.figure(figsize=(10,10))：设置画布大小，10对应1000的像素长，画图时多余的地方会填充空白，不加时figure会自动调整图大小。高清图可以为(20，20)

plt.imshow(image)：在image的基础上画图

plt.rcParams['font.sans-serif']=['SimHei']：设置可以显示中文。linux需要下载字体到matplotlib库位置下的mpl-data/fonts/tff，然后清除.cache中matplotlib缓存

plt.rcParams['axes.unicode\_minus']=False：使用字体时让坐标轴正常显示负号

plt.rcParams['font.size']=10：设置字体大小，默认为10

plt.xlabel('X')/plt.ylabel('Y')：添加x/y轴标签

plt.axis('off')：不显示坐标轴

plt.grid()：自动加上网格

plt.title('标题')：创建图像标题

plt.text(x,y,label)：在坐标位置添加标签

plt.savefig(保存的文件夹位置.jpg)：保存结果。要放到plt.show()前否则为空白

plt.show()：显示图片到窗口，同时清空画布

plt.close()：清空画布

**曲线图**

plt.plot(x,y,linestyle='-',color='blue',label='')：画曲线图。x轴自动取等间距。linestyle为线形，'-'是线，'o'是点，''是点加线；color还有black、green、cyan、yellow、orange、red等

plt.legend()：当plt.plot中label不显示时，加上此设置

**散点图**

plt.scatter(x,y,s=20,c='blue',cmap=None,alpha=None,marker='o')：画散点图。s为点的大小；c为颜色，c可以为列表或numpy数组，数据可以为'red'等具体颜色，也可以为数值代表渐变色；cmap只有c为列表时才生效，比如'RdYlBu'代表数值从小到大时颜色从红变黄变蓝，'RdYlBu\_r'反之；alpha为透明度0-1，三维散点图时默认远的点透明度低；marker为点的形状，比如'o'为圆，'+'为+号

plt.colorbar()：散点图设置c和cmap时可以显示颜色条

**三维散点图**

ax=plt.subplot(projection='3d')：创建三维画布

ax.set\_title(name)：设置标题

ax.scatter(x,y,z,s=20,c='blue')：画散点图。s为点的大小；c为颜色

ax.set\_xlabel('x')：设置x坐标名称

ax.set\_xlabel('y')：设置y坐标名称

ax.set\_xlabel('z')：设置z坐标名称

**合并图**

plt.subplot(2,2,1)：分割图。2,2表示分割为2行2列共4个区域，1表示在第一块区域画图，此设置后的操作都只在第1个区域，同理plt.subplot(2,2,2)表示画在第2个区域

**坐标轴间隔**

**from matplotlib.pyplot import MultipleLocator**

plt.gca().xaxis.set\_major\_locator(MultipleLocator(a))：将x坐标轴间隔设置为a。同理yaxis设置y坐标轴。a推荐设置为len(data)//20以内

plt.xlim(a,b)：设置x坐标轴范围为a-b。同理xlim设置y坐标轴

**图片显示**

**import matplotlib.image as mpimg**

plt.imshow(image)：将图片显示到画布中，注意此后在画布中画点和线时，改为使用以左上角为原点的(x,y)图片索引坐标

**import PIL**

image=PIL.Image.open(路径):读取图片为PIL.Image格式。当图片为rgba时，透明部分值为(0,0,0,0)

image=image.convert('RGB')：将RGBA图片转为RGB格式

image.size：图片尺寸(w,h)

image.mode：图片通道格式('RGBA'/'RGB')

image\_array=np.array(image)：将图片从PIL.Image格式变为np数组

image=PIL.Image.fromarray(image\_array)：将图片从np数组变为PIL.Image格式

image=PIL.Image.open(io.BytesIO(image\_byte))：字节类型图片转PIL.Image格式

image.save(路径):保存图片

image.format:看格式；image.size:看尺寸；image.mode:看色彩模式

plt.show(image对象/numpy数组):显示图片

image.convert(色彩模式):1是二值，L是灰度，P是8位彩色，RGB是24位彩色，RGBA是32位彩色，CMYK是CMYK彩色，YCbCr是YCbCr彩色，I是32位整形灰度，F是32位浮点灰度

np.array(pic1):将图片转化为数组

image.crop((x1,y1,x2,y2)):裁剪图像

image.resize((w,h))：改变图片大小

image1.paste(image2,(x,y))：将image2粘贴到image1上面，没有返回值。(x,y)为左上坐标

**import cv2**

(pip install opencv-python安装)

部分函数申请了专利要等保护器过后才能使用

cv2中读取图片、运算使用的是np数组组成的BGR通道(h,w,3)。cv2的HSV通道中H通道0-360简化为0-180。cv2中数据类型为np.uint8。np.array([-1], dtype=np.uint8)结果为255

cv2.\_\_version\_\_:查看版本；cv2.\_\_path\_\_:查看位置

opencv的读取RGB图片时，读取的顺序为BGR，转化的numpy数组类型为uint8，形状为(高,宽,通道)，内存要连续存储，对图像的操作通常会改变原图，最好复制内存

np.ascontiguousarray(image)：将数组的内存变为连续存储

cv2.imread(路径,flags):读取图片为BGR格式的numpy数组(英文路径)。flags：默认为cv2.IMREAD\_COLOR，当图片为png时忽略A通道；cv2.IMREAD\_GRAYSCALE/0读入灰度图；cv2.IMREAD\_UNCHANGED如果是png时包括A通道，如果是jpg还是3通道

image=cv2.imdecode(np.fromfile(image\_path,dtype=np.uint8),cv2.IMREAD\_COLOR)：读取中文路径的图片，imdecode读取后也是BGR通道

cv2.cvtColor(image,flag)：通道转换。flag：cv2.COLOR\_BGR2GRAY将BGR转为gray；cv2.COLOR\_BGR2RGB/cv2.COLOR\_RGB2BGR将BGR转为RGB；cv2.COLOR\_BGR2HSV将BGR转为HSV

capture=cv2.VideoCapture(路径):读取视频文件。路径为0时打开电脑摄像头，按S键保存图片

capture.isOpened():判断是否能打开，返回True或False

bool,image=capture.read():按顺序读取一帧(如果是摄像头则实时获取一帧)，返回bool值和图片

cv2.imwrite(路径,image):保存图片到指定位置，路径中包括名称。如果保存出的图片是黑色的，可能是因为image归一化的原因，此时需要image\*255

**显示图片：**

cv2.imshow(名称,image):将numpy数组显示成图片，显示通道为BGR

cv2.waitKey(0):0为一直显示(按任意键退出)，可改为任意的时长(毫秒)

cv2.destroyAllWindows():关闭所有窗口并释放内存(最好加上)

cv2.namedWindow(名称)：创建一个一直存在的窗口，然后使用cv2.imshow(名称要一致，否则会单独创建一个显示)和cv2.waitKey显示图片，最后再用cv2.destroyAllWindows()

**基本操作：**

image2=image1：引用image1 | image2=image1.copy()：复制image1

image[a:b,c:d]:选取图像的某个区域；image[:,:,0]:选取B通道

B,G,R=cv2.split(image1):取出每个颜色通道

image=cv2.merge((B,G,R)):通道合成

cv2.addWeighted(image,a,img2,b,c):融合图。a,b为融合比例，c为0

thresh,image2=cv2.threshold(image1,thresh,maxval,type):转化为二值图,返回阈值和图像两个值。thresh为阈值，maxval执行的操作一般为255，type：cv2.THRESH\_BINARY低于阈值设为0，高于阈值为maxval，cv2.THRESH\_BINARY\_INV与之相反

cv2.inRange(image,a,b):将image中a到b的值变为255，其余变为0，当image为BGR时，a和b也为BGR三通到。把图像直接进行inRange操作的效果比转化为灰度图再做好

cv2.resize(image,(w,h),interpolation):将图像拉伸为指定形状(w,h)(注意与一般的reshape是反着的)。interpolation：cv2.INTER\_LINEAR双线性插值(默认，速度快效果较好)，cv2.INTER\_NEAREST最近邻插值，cv2.INTER\_AREA使用像素区域关系重采样，cv2.INTER\_CUBIC为像素邻域的双3次插值，cv2.INTER\_LANCZOS4为像素邻域的Lancaos插值

cv2.copyMakeBorder(image,a,b,c,d,borderType):扩充边界。a,b,c,d为上下左右边界的扩充宽度，borderType为填充类型：cv2.BORDER\_REPLICATE复制边缘像素，cv2.BORDER\_REFLECT反射镜象，cv2.BORDER\_REFLECT\_101以边缘为轴反射镜像，cv2.BORDER\_CONSTANT,value=a填充常数a

cv2.flip(image,a)：翻转图像。a：1为水平翻转，0为垂直翻转，-1为水平垂直翻转

cv2.transpose(image)：转置。效果上为图片先左右翻转，再逆时针转90度

cv2.dnn.blobFromImage(image,size=(a,a),swapRB,mean,scalefactor=1)：将图片如(640,640,3)转为如(1,3,640,640)的形状并做预处理，数据类型也转为方便训练的float32。a为转换后图片的形状；mean=[r,g,b]为每个通道减去的均值，如果图片是BGR的，swapRB=True，不指定mean时不减均值；scalefacteor一般为1/255做归一化(减均值后)

cv2.dnn.blobFromImages([image,...])：处理多张图片时更快

cv2.pointPolygonTest(contour,(x,y),False) ：判断某点是否在多边形内并返回最短距离，在多边形内部返回正数，在外部返回负数，在边框上返回0.0。输入数据要为整数，contour为包含多组坐标的np.array，坐标顺序要按顺时针填入

**数据转换：**

cv2.imencode('.jpg',image)：将图片用'.jpg'编码方式转为一行np.array数组

cv2.imdecode(np.array,cv2.IMREAD\_COLOR)：将一行np数组转换为BGR图片数组

**画图(会改变原图)：**

画图时共用内存会直接改变原图，可以使用image.copy()新建内存

cv2.line(image,point1,point2,color)：在image中画直线。point1/2为直线起始/终点坐标；color单通道为a，BGR通道为(a,b,c)

cv2.rectangle(image,point1,point2,color=(0,255,0),thickness=2)：在image中画矩形框。point1、point2为矩形左上(x,y)、右下(x,y)坐标，以图片左上角为原点，必须为整数型；thickness为框的宽度，必须为整数，推荐min(w,h)//256

cv2.circle(image,(w,h),radius=3,color=(0,255,0),thickness)：画圆。radius为圆半径，推荐min(w,h)//256；thickness为边的粗细，thickness=-1时为实心圆

cv2.putText(image,'名称',(x1,y1),font,a,(0,255,0),b)：画标签注释。x1,y1为标签左下坐标；font为字体如cv2.FONT\_HERSHEY\_SIMPLEX；a为字大小，一般为1以内；(0,255,0)为颜色；b为字粗细，必须为整数一般为1或2

cv2.fillConvexPoly(image,points,(a,b,c))：在image上根据points中的坐标画实心图案，(a,b,c)为颜色。points要为np.array

cv2.fillPoly(image,[point],color)：画实心多边形。point为顺时针顶点

cv2.polylines(image,[point],isClosed,color)：画多边形边框。isClosed为True

**形态学运算：**

cv2.bitwise\_and/or/add/subtract(image1,image2):与/或/加/减运算。越界时值为边界值

kernel=np.ones((a,a),np.uint8)：全为1的卷积核

cv2.erode(image,kernel,iterations):腐蚀，凹的地方变钝，凸的地方变尖。iterations为迭代次数

cv2.dilate(image.kernel,iterations):膨胀，凹的地方变尖，凸的地方变钝

cv2.morphologyEx(image,op,kernel):利用腐蚀和膨胀运算。kernel同上；op：cv2.MORPH\_OPEN开运算，先腐蚀再膨胀消除黑点；cv2.MORPH\_CLOSE闭运算，先膨胀再腐蚀消除小孔；cv2.MORPH\_TOPHAT顶帽，原图-开运算，突出亮的部分；cv2.MORPH\_BLACKHAT黑帽，闭运算-原图，突出暗的部分

cv2.calcHist([image],[channels],mask,[histSize],[ranges]):计算图像直方图，返回存放含histSize个元素的一维数组。channels为图像的通道；mask：一般为None处理全图；histSize表示使用多少柱子一般为256；ranges是像素的范围一般为[0,256]

cv2.equalizeHist(image):将image(单通道)直方图均衡化，返回图像

clahe1=cv2.createCLAHE(clipLimit=a,tileGridSize=(b,b)):定义一个clahe模板。clipLimit可为2，tileGridSize=(b,b)为划分的区域数量，分别对各个区域进行直方图均衡化，然后把区域边界额外处理后合成

clahe1.apply(image):将clahe1模型应用到image中，会改变image

**卷积运算与边缘检测：**

cv2.filter2D(image,ddepth,kernel):对图像进行卷积。ddepth：-1表示输出与原图像深度相同

cv2.blur(image,(a,a)):均值滤波。(a,a)为卷积核，一般卷积核较大

cv2.GaussianBlur(image,(a,a),sigmaX,sigmaY):高斯滤波，比均值效果好。(a,a)为卷积核，sigmaX/Y为x和y轴上的高斯核标准偏差，偏差越大越模糊

cv2.medianBlur(image,a):中值滤波，适合去除噪音点。a为方框大小

cv2.Sobel(image,ddepth,dx,dy,ksize):用Sobel算子计算，会有负数。ddepth一般为cv2.CV\_64F；dx,dy为0/1，选用横或竖算子，同时为1时计算横和竖边缘，但效果不如分别计算再相加好，ksize=a为卷积核边框大小

cv2.convertScaleAbs(image):将image取绝对值并转换为8位数

(cv2.addWeighted(sobelx1,1,sobely1,1,)

cv2.Scharr(image,ddepth,dx,dy,ksize):用Scharr算子计算，会有负数

cv2.Laplacian(image,ddepth):用Laplacian算子计算，会有负数

cv2.pyrDown(image):高斯下采样。图像变为1/4(与高斯核卷积后去除偶数行和列)

cv2.pyrUp(image):高斯上采样，图像扩大4倍(先将图像每个像素点的右、右下、下三个方向填充0，然后使用高斯核卷积获得近似值)

image-cv2.pyrUp(cv2.pyrDown(image)):利用高斯下、上采样进行边缘提取。下采样后再上采样与原图相比略缩小

cv2.Canny(image,minval,maxval)：Canny边缘提取。minval和maxval(0-255)越小保留的细节越多

cv2.HoughLines()：霍夫变换。利用边缘提取和霍夫变换可以使倾斜图片摆正

**轮廓检测：**

contour,hierarchy=cv2.findContours(image,mode,methde)：检测二值图的物体轮廓。contour[0]形状为(点数,1,2)。mode：cv2.RETR\_EXTERNAL检测外轮廓；methode：cv2.CHAIN\_APPROX\_NONE存储轮廓上所有的点，其他还有cv2.CHAIN\_APPROX\_SIMPLE、cv2.CHAIN\_APPROX\_TC89\_L1、cv2.CHAIN\_APPROX\_TC89\_KCOS，四种方法的轮廓点数依次减少

**模板匹配：**

list,array=cv2.findContours(image,mode,method):轮廓提取。list为所有轮廓组成的列表，包括很多没用的轮廓和图像边界框；array为每条轮廓对应的属性，image应先用处理为二值图。mode：cv2.RETR\_EXTERNAL只检测外轮廓，cv2.RETR\_TREE检索所有轮廓并重构嵌套轮廓的层次；method：cv2.CHAIN\_APPROX\_SIMPLE对于直线轮廓只保留轮廓终点

cv2.drawContours(image,contours,q,(a,b,c),w):将轮廓绘制到image中，会改变image。列表contours为轮廓信息；q为-1时显示所有轮廓；(a,b,c)为BGR通道表示绘制轮廓的颜色；w表示绘制线条的粗细

cv2.contourArea(list[a]):计算轮廓list[a]的面积

cv2.arcLength(list[a],True/False):计算轮廓list[a]的周长，True时曲线是封闭的

x,y,w,h=cv2.boundingRect(list[a]):用矩形边框包含第a个轮廓

arrary1=cv2.matchTemplate(image,template1,method):模板匹配，返回各板块数组。method：cv2.TM\_SQDIFF平方差匹配法，值越接近0越好；cv2.TM\_CCORR相关匹配法，数值越大越好；cv2.TM\_CCOEFF 相关系数匹配法，1最好，-1最差；cv.TM\_SQDIFF\_NORMED归一化平方差匹配法；cv2.TM\_CCORR\_NORMED归一化相关匹配法；cv2.TM\_CCOEFF\_NORMED归一化相关系数匹配法

cv2.TM\_SQDIFF\_NORMED归一化平方差匹配，cv2.TM\_CCORR\_NORMED归一化相关匹配，cv2.TM\_CCOEFF\_NORMED归一化相关系数匹配

min\_val,max\_val,min\_loc,max\_loc=cv2.minMaxLoc(arrary1):得到arrary1中最小值、最大值和其位置，min/max\_loc中存放模板左上角的(列，行)

**获取关键点：**

sift=cv2.xfeatures2d.SIFT\_create()：调用sift特征提取

keypoints=sift.detect(image)：获取图像的关键点(建议用灰度图)

img2=cv2.drawKeypoints(image,keypoints,img2)：用彩色小圆圈绘制关键点，image为原图，img2为绘制关键点的图

**坐标变换：**

M=cv2.getAffineTransfrom(points1,points2)：得到Affine类型变换矩阵。Points1/2要至少3组对应的坐标

cv2.warpAffine(image,M)：将image进行仿射变换

M=cv2.getAffineTransfrom(points1,points2)：得到仿射变换矩阵。Points1/2要至少3组对应的坐标

cv2.warpAffine(image,M)：将image进行M仿射变换。可实现旋转，平移，缩放；变换后的平行线依旧平行

M=cv2.getPerspectiveTransform(points1,points2)：得到透视变换矩阵。Points1/2要至少4组对应的坐标

cv2.warpPerspective(image,M)：将image进行M透视变换。可保持直线不变形，但是平行线可能不再平行

**import albumentations**

图片处理与增强库，需要opencv库

<https://albumentations.ai/docs/getting_started/image_augmentation>

transform=albumentations.Compose([...])：定义转换的框架。将其他执行的函数依次放入框架中

//image=transform(image=image)['image']：执行框架

albumentations.LongestMaxSize(max\_size)：等比缩放，指定最大边长

albumentations.SmallestMaxSize(min\_size)：等比缩放，指定最小边长

albumentations.Normalize(max\_pixel\_value=255.0,mean=(0.485,0.456,0.406),std=(0.229,0.224,0.225))：归一化、减均值、除以方标准差。默认为BGR

albumentations.PadIfNeeded(min\_height=320,min\_width=320,border\_mode=cv2.BORDER\_CONSTANT,value=(126,126,126))：向图片四周填充value

albumentations.GaussianBlur(blur\_limit=(3,3),sigma\_limit=0,p=0.5)：高斯模糊。blur\_limit为卷积核大小；sigma\_limit为标准差；p为使用的概率

albumentations.GaussNoise(var\_limit=(10,50),p=0.5)：高斯噪声。var\_limit为噪声方差范围；p为使用概率

**import sklearn**

(pip install scikit-learn)

机器学习库

sklearn.metrics.explained\_variance\_score(A,B)：期望方差

sklearn.model\_selection.train\_test\_split()：划分数据集

**K近邻**

model=sklearn.neighbors.KNeighborsClassifier(n\_neighbors,weights,algorithm,leaf\_size,metric,p)：初始化。n\_neighbors是KNN的k值默认为5：k值较小时训练误差小但泛化误差大，容易受噪声影响，k值较大时反之，可以采用交叉验证法选取k值；weights是权重：uniform均等权重，distance距离近的点权重影响较；algorithm：auto自动选择，brute是线性扫描,训练集大时很耗时，kd\_tree构造kd数存储数据，维数小于20时效率高，ball\_tree克服了kd树高纬失效；leaf\_size默认30：构造kd树和ball树的大小，这个值会影响树构建的速度和存储大小；P默认为2：p=2是欧氏距离，P=1是曼哈顿距离，曼哈顿距离一般会比欧氏距离长,但计算速度快,不会有误差

model.fit(train\_x,train\_y):训练模型

model.predict(test\_x):用模型预测

model.score(test\_x,test\_y):求准确率

**K均值**

model=sklearn.cluster.KMeans(n\_clusters=8,max\_iter=300,n\_init=10)：初始化。n\_clusters是生成聚类数；max\_iter是执行一次算法最大迭代数；n\_init是用不同聚类中心运行的次数，最终用inertia指标选出最优结果

model.fit(train\_x)：拟合模型

model.labels\_：显示聚类结果

**PCA主成分分析法**

**from sklearn.decomposition import PCA**

利用PCA可以将数据降维度。比如对空间中均匀分布的3维数据降1维后会得到均匀的二维数据，但将均匀的二维数据逆变换后在空间会是一个斜面

model=PCA(n\_components,copy)：初始化。基于方差最大化。n\_components是要变成的维度,用'mle'表示采用mle算法自动决定变成的维度；copy默认为True,表示训练数据时使用备份,若为False会改变原始数据

model.fit(train\_x)：拟合模型

model.explained\_variance\_：降维后各主成分方差,越大证明越重要

model.explained\_variance\_ratio\_：降维后各主成分方差占比

model.transform(train\_x)：使用模型，得到降维后的数据

model.inverse\_transform(result)：将降维后的数据还原

**支持向量机分类**

model=sklearn.svm.SVC(C=1.0,kernel='rbf',cache\_size=200,max\_iter=-1,)：创建模型。C为惩罚参数：C越大准确度越高但泛化能力越弱；kernel为核函数：'linear'线性核函数，'poly'多项式核函数，'rbf'高斯核函数；'sigmoid'sigmoid核函数；cache\_size为指定训练需要的内存；max\_iter为最大迭代次数：-1表示无限制。

model.fit(train\_x)：训练模型

model.predict(test\_x):用模型预测

**线性回归(LR)**

**from sklearn.linear\_model import LinearRegression**

输入形状(batch,feature\_in)，输出形状(batch,feature\_out)，线性拟合

model=LinearRegression()：创建模型

model.fit(train\_input,train\_output)：拟合模型

model.predict(data)：使用模型

**全连接层(MLP)**

**from sklearn.neural\_network import MLPRegressor**

model=MLPRegressor(hidden\_layer\_sizes=(64,128,256,512),activation='relu',solver='adam',max\_iter=200,random\_state=1)：创建模型。hidden\_layer\_sizes为层数；max\_iter为最大训练次数；random\_state为随机种子

model.fit(train\_input,train\_output)：拟合模型

model.predict(data)：使用模型

**高斯过程回归(GPR)**

**from sklearn.gaussian\_process import GaussianProcessRegressor**

输入形状(batch,feature\_in)，输出形状(batch,feature\_out)，通常对于表格数据能完全拟合。高斯过程的内核之间可以相加，也可以直接加上数值来实现偏移

model=GaussianProcessRegressor(kernel=None,alpha=1e-10)：创建模型。kernel为协方差函数的核，None时默认平方指数核RBF(径向基函数)；alpha为拟合时在核矩阵对角线上增加的值，较大的值对应于观测中噪声水平的增加

model.fit(train\_input,train\_output)：拟合模型。train\_input形状为(batch,feature\_in)，train\_output形状为(batch,feature\_out)

model.predict(val\_input)：使用模型

kernel：高斯核。常用RBF+0.1\*DotProduct或Matern

**from sklearn.gaussian\_process.kernels import RBF**

kernel=RBF(length\_scale=1)：平方指数核(径向基函数核)。length\_scale为核的长度尺寸，调整几乎没有影响，如果显示警告无法聚合等，在不影响结果时可以调整此参数

**from sklearn.gaussian\_process.kernels import Matern**

kernel=Matern(length\_scale=1,nu=1.5)：Matern内核，这类矩阵核是径向基函数RBF的推广。length\_scale为核的长度尺寸；nu控制学习函数的平滑性，nu越小越不光滑，nu=0.5为绝对指数核，nu=1.5为1次可微函数，nu=2.5为二次可微函数，nu=np.inf等价于RBF核

**from sklearn.gaussian\_process.kernels import DotProduct**

kernel=DotProduct(sigma\_0=1.0)：点积核，降低拟合能力增强泛化能力。sigma\_0控制核的不均匀性，sigma\_0=0时内核是同质的。通常RBF+0.1\*DotProduct配合使用

**from sklearn.gaussian\_process.kernels import RationalQuadratic**

kernel=RationalQuadratic(length\_scale=1.0,alpha=1.0)：有理二次核。有理二次核可以看作是具有不同特征长度尺度的RBF核的尺度混合物(无穷和)。length\_scale为核的尺度；alpha为混合参数

**from sklearn.gaussian\_process.kernels import ExpSineSquared**

kernel=ExpSineSquared(length\_scale=1.0, periodicity=1.0)：sin平方核。允许对完全重复的函数建模。length\_scale为核的尺度；periodicity为核的周期性。注意GaussianProcessRegressor的alpha要改为4以上

**from sklearn.gaussian\_process.kernels import WhiteKernel**

kernel\_add=WhiteKernel(noise\_level=1.0)：白噪声核，模拟噪声，通常与其他核相加后使用。noise\_level为噪声的方差

**import base64**

二进制加密传输格式。之所以要转为base64格式而不直接用字节类型，是因为数据传输格式要为字符串，而字节类型无法直接转为字符串，同时字节类型转为base64内容也会变长

如果传输的数据是嵌套在字典中的，则还需要使用json.dumps()和json.loads()将字典变为字符串

data\_base64=base64.b64encode(data\_byte)：字节类型转为base64

data\_byte=base64.b64decode(data\_base64)：base64转为字节类型

data\_base64\_str=data\_base64.decode()：base64转为字符串

data\_base64=data\_base64\_str.encode()：字符串转为base64

**编码：**

1，image\_array=cv2.imencode('.jpg/.png',image)[1]：图片数组转为一行np数组

2，image\_byte=image\_array.tostring()：一行np数组转为字节类型

3，image\_base64=base64.b64encode(image\_byte)：字节类型转为base64

4，image\_base64\_str=image\_base64.decode()：base64转为字符串

5，image\_dict={'image':image\_str}：数据放到字典中

6，dict\_str=json.dumps(image\_dict)：字典转为字符串

**解码：**

1，image\_dict=json.loads(dict\_str)：字符串转为字典

2，image\_base64\_str=image\_dict['image']：从字典中取出数据

3，image\_base64=image\_base64\_str.decode()：字符串转为base64

4，image\_byte=base64.b64decode(image\_base64)：base64转为字节类型

或者with open(图片位置,'rb',encoding='utf-8') as f: // image\_byte=f.read()

5，image\_array=np.frombuffer(image\_byte,dtype=np.uint8)：字节类型转为np数组

6，image=cv2.imdecode(image\_array,cv2.IMREAD\_COLOR)：一行np数组转为BGR图片

**import re**

正则表达式测试网站：<https://c.runoob.com/front-end/854/>

regex=re.compile('正则表达式')：定义正则匹配函数

result\_list=regex.findall('字符串')：找到所有匹配的值返回到列表中

result=regex.search('字符串')：找到到第一个匹配的值返回特殊对象，没找到返回None。result.group()为匹配到的内容；result.group(1)为匹配到的第一个对象，比如a(b)c时返回b，如果不存在会报错；result.regs[0]得到元组(起始字符下标，结束字符下标+1)

result=regex.match('字符串')：只检测字符串的开头是否匹配，用法与search一致

result=re.sub(regex,'\*',text)：将匹配到的内容替换为一个\*号

(^| )(13|15|18)\d{9}($| )：电话号码正则表达式(会带空格)。优先级：()、其他、|；|表示或；^表示字符前面不能有其他字符；$表示字符后面不能有其他字符；(^| )表示除了空格不能有其他字符。如果前后的空格不加进去要使用(13|15|18)\d{9}($|(?= ))

1或[1]：表示该位置为1。普通字符加不加[]都可以，但特殊字符和表达式要加，比如[\*]

[0-9]或[\u0030-\u0039]：表示该位置为0-9中的数字。[0-9]+表示任意个数字

[A-Z]或[\u0041-\u005a]：A-Z

[a-z]或[\u0061-\u007a]：a-z

[a-zA-Z]或[\u0041-\u007a]：大小写字母

[^a-zA-Z]：除了大小写字母

[\u4e00-\u9fa5]：汉字

.：匹配任意个除\n外的字符，不能为0个

.{m-n}：匹配任意m-n个除\n外的字符但m不能为0

.+：匹配任意个除\n外的字符，不能为0个

.\*：匹配任意个除\n外的字符，可以为0个

\d：匹配1个数字

\D：匹配除数字以外的字符

\w：匹配大小写英文字符、数字或下划线的组合

\W：匹配除\w以外的字符

\s：匹配空白字符，包括空格、制表符、换行符、回车符等

\S：匹配除\s以外的字符

a?：匹配可能存在的1个a字符

a(.\*)b：匹配最长的a...b字符

a(.\*?)b：匹配最短的a...b字符，也可能就是ab

a(.+?)b：匹配最短的ax...b字符，中间至少有一个字符

\n：匹配换行符

**import json**

json无法编码numpy格式，要转为列表

json.loads('{"text": [0,1,2]}')：将字符串解析为字典或列表，会去除\n。注意字典里面是双引号，外面是单引号，因此要使用json.dumps(字典)得到字符串而不用str(字典)

json.dumps(字典)：将字典或列表变为为字符串。用于数据传输

json.load(json\_path)：读取json文件。如果保存时用过json.dumps，需要用json.loads

json.dump(字典,f,ensure\_ascii=True)：将字典(可以先经过json.dumps)保存为json

with open(path,'r',encoding='utf-8') as f: // load=json.load(f)：读取json

with open(path,'w',encoding='utf-8') as f: // json.dump(字典, f, ensure\_ascii=True)：转字典为json文件并保存。ensure\_ascii=True时会将中文转换为ascii码(占用更少空间，但不便于阅读)

标准COCO数据集格式为json，在数据转换为json格式前也需要变成对应的标准字典格式，COCO中annotations保存的坐标为xmin,ymin,w,h：

{'info','categories','images','annotations'}

info{'year','version','description','contributor','url','date\_created'}

categories[{'id','name'}]

images[{'id','file\_name','width','height'}]

annotations[{'id','category\_id','image\_id','box':[xmin,ymin,w,h],'area'}]

**import yaml**

(pip install pyyaml安装)

标注：# 内容

字典格式：名称: 内容

多级嵌套：1级名称:// 2级名称: 内容...

列表：1级名称://- 元素1//- 元素2...

注意当名称为'01'时可能识别为int也可能为str

with open('文件位置.yaml','r',encoding='utf-8')as f:

//yaml\_dict=yaml.load(f,Loader=yaml.SafeLoader)：加载yaml中内容为字典。等同于yaml.safe\_load(f)

with open('文件位置.yaml','w',encoding='utf-8') as f:

//yaml.dump(字典,f,allow\_unicode=False,sort\_keys=True)：保存。allow\_unicode=True显示中文；sort\_keys=True先给内容按键排序再写入

**import trimesh**

github：<https://github.com/mikedh/trimesh>

官方文档：<https://trimesh.org/>

trimesh的操作对象有网格trimesh和场景scene(用于简单渲染)，trimesh格式会把导入的模型合并再转为三角形网格，所有材质合并到一张纹理贴图上，scene格式会保持原样。模型由多个分开部分组成时，可以用scene格式加载和保存，对各部分可以单独转为trimesh格式进行修改，再合入scene。scene只支持窗口渲染查看，在后台渲染要使用pyrender库

trimesh在顶点着色时会将所有材质合到一张由颜色点组成的纹理贴图上，顶点的UV坐标会指向这些颜色点，导出后也会保留改变

trimesh中的数据使用时可以转换为numpy，但修改数据时可以直接=numpy数组

trimesh中对面mesh.faces修改时，其他面属性比如mesh.face\_normals会自动改变

trimesh中对顶点mesh.vertices改变值时，其他顶点属性比如mesh.vertex\_normals会自动改变。但对顶点添加和删除时，需要手动对mesh.visual.uv、mesh.faces等修改

pyrender渲染后端：Pyglet(默认)、OSMesa、EGL。Pyglet需要显示管理器(linux上没有)；OSMesa为cpu软件渲染器；EGL为GPU加速渲染器。OSMesa和EGL的安装比较麻烦

**创建网格**

mesh=trimesh.load(path,force=None)：加载3D文件。force设置为'mesh'或'scene'，设置为mesh时会合并为一个网格；设置为scene时可以单独取出每个网格

visual=trimesh.visual.TextureVisuals(uv,material=None,image=None)：创建纹理，至少需要material或image其中一个

mesh=trimesh.Trimesh(vertex,face,vertex\_normals=None,face\_normals=None,visual=None,process=True,validate=False)：创建Trimesh网格对象。face传入非三角形时，会让第1个点与其他所有顶点组成三角形。vertex\_normals和face\_normals没传入时会自动创建，面的顶点索引的逆时针为正面；visual为包含材质纹理信息的TextureVisuals对象，注意visual中包含顶点、面的信息时，process和validate会失效；process=True时会删除错误、合并顶点；validate=True时会更改有问题的面

mesh.copy()：复制全新的mesh。mesh传入函数后内存不变

**网格属性**

mesh.metadata['name']=name：网格命名

np.array(mesh.vertices)：顶点坐标(顶点数,3)

np.array(mesh.vertex\_normals)：顶点法线(顶点数,3)

np.array(mesh.faces)：面索引(面数,3)

np.array(mesh.triangles)：面的顶点坐标(面数,3,3)

mesh.edges：所有边的顶点索引(包括没有面的孤立边)。形状(边数,2)

np.array(mesh.face\_normals)：面法线(面数,3)

np.array(mesh.visual.uv)：顶点UV坐标

mesh.vertex\_neighbors：每个点相连的其他点索引(顶点数,n)

mesh.face\_adjacency：相邻面的索引。形状(边数,2)

mesh.face\_adjacency\_angles：相邻面的法线弧度夹角。形状为(边数,)。如果是平整的面弧度接近0，注意面的法线方向可能相反

mesh.face\_adjacency\_edges：相邻面的顶点索引。形状(边数,2)

mesh.volume：网格体积

mesh.center\_mass：网格中心坐标

mesh.visual.material.image：加载obj文件时的纹理贴图(PIL.Image的png)

mesh.visual.material.baseColorTexture：加载glb文件时的纹理贴图

**基本操作**

mesh.update\_faces(face\_mask)：删除面，自动更改面法线等，但不会改变顶点和UV等。face\_mask为bool组成的numpy数组

mesh.update\_vertices(vertex\_index)：删除顶点，自动更改UV等，原来在面中的顶点会替换为相近的顶点，面数不变。通常需要先使用mesh.update\_faces去除面

mesh.process()：删除错误顶点、合并顶点

mesh.split(only\_watertight=True)：分割为多个网格。only\_watertight=True时只保留封闭网格，剩余的丢弃

mesh.show()：显示渲染网格

mesh.export(save\_path)：保存网格文件

vertex\_color=trimesh.visual.uv\_to\_color(uv,image)：从纹理贴图上找到每个顶点的颜色(v,4)，最后一个通道为alpha，一般为255

visual=trimesh.visual.TextureVisuals(uv,material=None,image=None)：创建纹理，至少需要material或image其中一个

near\_vertex,distance,near\_face=mesh.nearest.on\_surface(point)：求point到模型表面的最近顶点坐标、最近面距离、最近面索引

location,ray\_index,face\_index=mesh.ray.intersects\_location(origin,direction)：射线检测(投影)。origin为射线原点；direction为射线方向；location为交点坐标；ray\_index为射线对应的原点索引；face\_index为投影到的网格面索引。Location、ray\_index和face\_index的结果对应，会有没射线到的点和一个点射线到多个面的情况

**创建场景**

scene=trimesh.Scene(base\_frame='world')：创建场景。base\_frame为场景名称

scene.add\_geometry(mesh)：添加网格到场景中

scene.delete\_geometry(name)：删除网格

scene.geometry[name]：选择网格

scene.show()：显示渲染场景

image\_byte=scene.save\_image(resolution=None)：打开窗口渲染并返回字节类型图片。resolution=[w,h]设置图片大小

image\_array=np.frombuffer(image\_byte,dtype=np.uint8)：转为一行numpy数组

image=cv2.imdecode(image\_array,cv2.IMREAD\_COLOR)：转为图片

**import pyrender**

github：<https://github.com/mmatl/pyrender>

官方文档：<https://pyrender.readthedocs.io/en/latest/examples/index.html>

pyrender是一个支持后台渲染的库，通常配合trimesh库使用，注意pyrender的mesh和scene数据格式与trimesh的不一样，需要转换

mesh=pyrender.Mesh.from\_trimesh(trimesh\_mesh)：转换为pyrender格式的网格

light=pyrender.DirectionalLight(color=[1.0,1.0,1.0],intensity=1.0)：设置环境灯光，环境灯光不需要设置位置。intensity为光强度，设置为20时比较正常

camera=pyrender.OrthographicCamera(xmag=1.0,ymag=1.0)：设置相机类型

camera\_pose=[[a,0,0,0],[0,a,0,0],[0,0,a,z],[0,0,0,1]]：相机位置矩阵。修改a提高相机的视角广度，通常a=1；修改z提高摄像机位置的z轴高度

scene=pyrender.Scene()：创建场景

scene.add(mesh)：添加网格

scene.add(light)：添加灯光

scene.add(camera,pose=camera\_pose)：添加相机

pyrender.Viewer(scene)：在窗口中渲染场景。没有灯光时是黑色的，可以不设置相机

render=pyrender.OffscreenRenderer(viewport\_width,viewport\_height)：创建后台渲染器。需要设置渲染视角的图片大小

image,depth=render.render(scene)：得到渲染视角的图片np数组。

image=PIL.Image.fromarray(image)：转为PIL.Image图片

**import open3d**

**创建网格**

mesh=open3d.io.read\_triangle\_mesh(path,enable\_post\_processing=False)：加载网格。enable\_post\_processing=True时显示纹理

mesh=open3d.geometry.TriangleMesh()：创建空的open3d网格

mesh.vertices=open3d.utility.Vector3dVector(vertex)：添加顶点

mesh.triangles=open3d.utility.Vector3iVector(face)：添加面索引

mesh.vertex\_normals=open3d.utility.Vector3dVector(vertex\_normal)：添加顶点法线。使用mesh.compute\_vertex\_normals()自动计算顶点法线

mesh.triangle\_uvs=open3d.utility.Vector2dVector(uv)：添加顶点UV坐标

mesh.textures=[open3d.geometry.Image(image\_array)]：添加纹理图片

**网格属性**

vertex=np.array(mesh.vertices)：顶点坐标(v,3)

face=np.array(mesh.triangles)：面索引(f,3)

vertex\_normal=np.array(mesh.vertex\_normals)：顶点法线

face\_normal=np.array(mesh.triangle\_normals)：面法线(通常不需要)

uv=np.array(mesh.triangle\_uvs)：顶点UV坐标

vertex\_color=np.array(mesh.vertex\_colors)：顶点颜色(可能没有)

image=np.array(mesh.textures[0])：纹理图片(可能没有)

**基本操作**

open3d.visualization.draw\_geometries([mesh])：显示渲染网格

**import schedule**

定时任务的设置和执行库。基于python解释器运行，程序本身存在运行时间误差，不是高精度的定时器，误差在秒级以内。schedule的时间参考电脑本地的系统时间

while True://schedule.run\_pending()//time.sleep(1)：每隔1秒(实际会有误差)检查一次schedule中的定时任务，满足条件的任务会执行

schedule.every(a).seconds.do(function,param1=A)：每a秒执行一次function函数。param1为function传入的参数。此外还有minutes、hours、days、weeks等

schedule.every().second.do(function)：每1秒执行一次。同理minute、hour、day、week等

schedule.every(a).days.at('09:30').do(function)：每隔a天在当日的9:30执行一次。.weeks为每隔a周在该周的第一天执行

schedule.every().day.at('09:30').do(function)：每天9:30执行一次。.week为每周的第一天；

**《其他库》**

**import warnings**

warnings.filterwarnings('ignore')：忽略此条语句之后的警告信息

**import heapq**

优先队列通过最小堆实现，最小堆是一个完全二叉树，每个父节点的值都小于或等于子节点的值。循环查找列表的最值时，时间复杂度为O(n\*m)，使用优先队列时为0(log n)

heapq.heapify(list\_)：将list\_的数据结构转为最小堆，初次转换后按从小到大排序，没有返回值。转换后list\_依然为列表，但使用append等非heapq函数会破坏数据结构。如果为多层列表：[[0,1,2],[0,2,1]]，会依次根据子列表中的第1、2、3个值排序

heapq.heappop(list\_)：弹出并返回最小堆最上面的父节点(最小值)

heapq.heappush(list\_,A)：将元素插入到列表中，没有返回值。插入之后列表元素的顺序会打乱，不一定按照从小到大排序，但第1个元素一定是最小值

**import itertools**

list(itertools.permutations(list\_,a))：去重排列组合。list\_为所有元素的列表，a为选出其中的a个数进行排列，返回[(...),(...)...]

**import fitz**

(pip install pymupdf)

pdf、word等文档处理库

document=fitz.open(path)：读取文档

page=document.load\_page(n)：文档第n页

image = page.get\_pixmap(matrix=None)：渲染页面为图片。保存的图片会有压缩，matrix=fitz.Matrix(n,n)提高分辨率为n倍

image.save(save\_path)：保存图片

**import xml.etree.ElementTree as ET**

root=ET.parse(文件路径.xml).getroot()：定义文件对应的root工具

root.find/findall('标签')：读取xml文件中的/所有的'标签'

root.find('标签').text：将读取的标签以正常形式显示

for object in root.findall('object')：遍历xml文件中所有的'object'标签，并定义object工具(同root)，可以继续使用object.find('标签')

object[i][j].text：读取标签object中第i个子标签的第j个内容

**import psutil**

psutil.cpu\_times()：获取CPU情况

psutil.cpu\_count()：获取CPU逻辑个数

psutil.cpu\_count(logical=False)：获取CPU物理个数

psutil.virtual\_memory()：获取内存的使用情况

**import scipy**

scipy库基于numpy实现了更高级的数学计算功能

face=scipy.spatial.Delaunay(point\_2d).simplices：三角形剖分。得到三角形网格的顶点下标

**from scipy import ndimage**

可以处理插值、积分、优化、图像处理、常微分方程数值解的求解、信号处理等问题

ndimage.rotate(image,角度,reshape=False)：逆时针旋转图片，图像大小不变，超出原边框的地方被裁减，多出来的地方用黑色补充

**KD树**

kd\_tree=scipy.spatial.KDTree(point1)：构建KD树，可以是3维点

\_,near\_index=kd\_tree.query(point2)：point2中离point1中最近点的索引

**from mpl\_toolkits.mplot3d import Axes3D**

Axes3D.scatter(x,y,z):三维散点图

Axes3D.plot\_surface(x,y,z):三维表面图

Axes3D.plot\_wireframe(x,y,z):三维线框图

**import thop**

可以计算模型的FPS

in\_=torch.rand(1,3,640,640,dtype) ：创建输入张量

flops, params=profile(model, inputs=(in\_,))：计算FPS和参数量

**import psycopg2**

建立和操作数据库PostgreSQL

(pip install psycopg2-binary安装)

connect=psycopg2.connect：建立一个数据库connect

cursor=connect.cursor()：建立一个光标执行数据库操作

**import obs：华为云从网站上下载数据**

(pip install esdk-obs-python安装)

obs=obs.ObsClient(access\_key\_id,secret\_access\_key,server)：配置

obs.getObject(容器名称,图片服务器路径,保存位置,loadStreamInMemory=False)：从服务器下载图片

image=obs.getObject(容器名称,图片服务器路径,保存位置,loadStreamInMemory=True)：从服务器下载图片到本地缓存而不直接保存，此时会自动忽略保存位置

image=image.body.buffer：得到图片的字节类型数据

image=np.frombuffer(image,dtype=np.uint8)：将字节类型一行数组

image=cv2.imdecode(image,cv2.IMREAD\_COLOR)：将一行数组转为BGR图片

**blender**

语言设置：左上角[编辑] -> [偏好设置] -> [语言]

blender可以保存.blender文件，注意完整模型还包括同目录下的纹理等文件

blender中红色线为X轴、绿色为Y轴、蓝色为Z轴，Z轴向上、Y轴向前。模型导出时需要注意更改坐标轴方向、颜色等选项

**物体**

全选：点击物体按A键，可以选择物体的所有点和面

移动：G键+XYZ键或左上[移动图标]，再点击左下[移动]输入移动距离

旋转：R键+XYZ键或左上[旋转图标]，再点击左下[旋转]输入旋转角度

缩放：S键+XYZ键或左上[缩放图标]，再点击左下[缩放]输入缩放比例

游标：shift+鼠标右键或[游标图标]选中游标，shift+C键游标居中。选择物体后使用shift+S并选择游标可以让物体回到游标

隐藏：H键或右上[隐藏图标]，alt+H键取消隐藏

删除：X或delete键

隐藏侧边栏：N键

编辑：TAB键或左上角可以切换物体模式和编模式

**视角**

坐标轴：绿线是Y轴，红线是X轴，蓝线是Z轴

视角选择：tab上面的键或右上[视角图标]

视角移动：鼠标中键旋转视角、shift+鼠标中键平移视角、shift+C视角对齐物体

**基础**

编辑器窗口：所有编辑器窗口的左上角都可以选择编辑器的类型，鼠标移动到窗口的最边缘处，通过拖动来新建或合并窗口

视图着色方式：在布局右上角可以选择线框、实体、材质预览、渲染的视图着色方式。实体可能有颜色，但即使有颜色也不准确；材质预览是模型上色后的样子；渲染是加上环境光线等最终呈现出的样子

属性编辑器：默认页面的右下为属性编辑器，里面有很多常用的模块

修改器模块：属性编辑器中选择修改器模块，能够快速改变模型形状，比如把模型的边变圆滑

材质模块：属性编辑器中选择材质模块，可以新建和保存材质

世界环境模块：属性编辑器中选择世界环境模块，表(曲)面中可以选择世界颜色，通常选背景和白色

渲染模块：视图着色方式选择渲染后可以看到模型的最终效果，在右侧[属性编辑器]的[渲染]中可以进行一些设置，通常[渲染引擎]选择cycles

**UV编辑**

UV编辑器中可以编辑UV的展开方式，展开后的UV贴图是透明的框架。背景中的图片是纹理贴图，可以自定义，背景图片的尺寸会改变UV贴图的展开方式

UV贴图：进入属性编辑器中的数据模块中的[UV贴图]，可以新建多个UV贴图，但只能应用一个UV贴图。烘焙的时候，原来UV贴图右边的渲染要点亮，并选择要烘焙的新UV贴图

**着色**

着色编辑器中可以对每个材质添加节点以实现复杂的着色方案。选择一个材质后，会出现相连的节点流程图，初始有[原理化BSDF]和[材质输出]模块。1个物体可能有多个材质，但只能应用1个UV图，每个材质都有自己的节点流程图，如果要导入纹理贴图等需要对每个材质做相同的修改

创建纹理贴图：选择一个材质，添加[纹理]中的[图像纹理]，然后导入图片或新建空白图片。新建时可以设置白底、透明、32位浮点格式

烘焙：将材质按UV展开方式放到空白纹理贴图上。选择要烘焙的物体；世界环境模块背景设置为白色；选择要烘焙的UV贴图，原来的贴图开渲染；选择1个材质创建空白纹理贴图并选中，如果有多个材质则复制过去，烘焙结果会显示到空白纹理贴图上；进入渲染模块，[渲染引擎]推荐cycles，[采样]中修改视图采样为图片尺寸、渲染采样为64的倍数；在烘焙中点击[烘焙]

**编程模块**

blender中的scripting模块中可以使用代码进行操作，新建文本可以输入代码，代码中需要先import bpy。对物体的操作会转换成代码并显示到左下区域。文本的右上角可以使用文本换行功能。很多代码操作的前提是用鼠标选中目标物体

object\_list=bpy.context.selected\_objects：鼠标选择到的物体列表，object\_list会随鼠标选择的物体实时变动。len(...)查看物体数量

object=object\_list[i]：选择其中一个物体

object.name：物体名称

**《深度学习库》**

**labelimg**

(pip install labelimg)

可以标注图片并保存标签为xml格式，终端输入labelimg打开软件

功能open dir：打开存放图片的文件夹。每次都要重新新设置

功能change save dir：设定标注存放的地址。A、D快速切换图片

功能view>>Auto Save mode：自动保存标注 | CTRL+S保存图片

按住CTRL加鼠标滚轮放大缩小

Create RectBox画框 | Delete删除框 | 可以选择拖动每个框

**huggingface**

开源模型的存放网站

sudo apt-get install git-lfs：linux安装git-lfs。windows安装git时自带

git lfs install：启用lfs。不使用lfs无法下载大文件

git clone https://huggingface.co/作者名/项目名：克隆项目到本地

下载huggingface上单个文件时，可以使用git clone [文件连接]，[文件连接]通过鼠标右键目标文件获得

**import fiftyone**

目标检测数据集可视化工具

官方教程：https://docs.voxel51.com/index.html

可以选择本地数据(图片分类、目标检测)并后以浏览器的方式查看，会自动去除空标注的图片和找不到图片的标注，数据缓存为文件.fiftyone。需要pycocotools库

(pip install fiftyone)

(pip uninstall fiftyone fiftyone-brain fiftyone-db)

dataset=fiftyone.Dataset.from\_dir(name,labels\_path,data\_path,dataset\_type=fo.types.COCODetectionDataset)：查看数据。name为自定义名称，默认为当前的时间日期；labels\_path为coco标签位置(为空时也可正常运行)，如果设定了data\_path，coco标签可以为相对路径否则要为绝对路径

dataset.persistent=True：保存缓存，方便之后查看(可以设置为False并直接加载空标签，以快速启动fiftyone)

/ session=fiftyone.launch\_app(dataset)：启动fiftyone查看dataset

/ session.wait()：让网页等待，才能看到。如果网页关了要重新启动

如果在A服务器上启动程序，则网址为”A服务器公网地址:端口号”，端口号会在运行程序后给出，一般为5151

**import doccano**

文本分类标注工具

github：<https://github.com/doccano/doccano>

**import argparse**

python自带的命令行参数解析包

parser=argparse.ArgumentParser(description='提示')：创建参数解析器

parser.add\_argument('--参数名',default,type,choices=[],action,required,help)：添加参数。default为不传入参数时的默认值；type是命令行参数会被转成的类型如type=str；有choices时固定选用的参数；action='store\_true'时如果命令行传入参数如--A，则A的值为True，否则默认为False，'store\_false'与之相反；required=True时必须从命令行传入参数；help为提示信息

args=parser.parse\_args()：解析参数。args中包含所有定义的参数，args的类型为namespace

args,\_=parser.parse\_known\_args()：与parser.parse\_args()功能一样，但是可以避免参数冲突，比如用gunicorn启动代码时，它们的参数传入会有冲突

在终端使用'cd'命令进入文件所在目录，输入'python 文件名.py --参数名 值 --参数名 值...'传入参数并运行程序

python -m torch.distributed.launch --master\_port 9999 --nproc\_per\_node 2 文件名.py --device 0,1 --参数名 值...：启用pytorch的分布式训练。master\_port为随意一个未被占用的端口即可，nproc\_per\_node为使用GPU数，device为使用的GPU编号

**import wandb**

Weights & Biases是一个类似于tensorboard的在线模型训练可视化网站，wandb是Weights & Biases的辅助库，放到代码中可实时上传到网站观看。除此之外还会保存日志信息到当前目录wandb下。使用时wandb库时用VPN可能会影响

使用教程：<https://docs.wandb.ai/guides>

先注册Weights & Biases账号：<https://wandb.ai>

安装wandb库后在命令终端使用wandb login，复制Weights & Biases网站个人账户中的API key粘贴到命令终端指定位置就可以完成登录。Project Defaults中可以设置保存到哪个项目中(个人/小组的)

wandb\_run=wandb.init(project,name,config)：初始化，在网页中可看到。project为网站中项目的名称；name是单次训练的名称，一个项目中的所有单次训练可以合在一起观察；config是记录一些配置信息，定义了argparse后可以直接config=args

wandb\_run.log({'metric/val\_loss':val\_loss},...)：添加数据，'/'前面的'charts'为网页中的集合名。一般放在每轮循环中，在wandb网址上会自动生成图像。CPU和GPU的使用信息会自动获取。每一次使用log时所有图x轴都会+1，因此一轮中只能有一个log

wandb\_image=wandb.Image(image,boxes=None)：将图片变为可添加的对象。image为rgb通道的numpy数据

//boxes={"predictions":{"box\_data":[box1,box2,...],"class\_labels":wandb\_class\_name}}：记录所有的边框。用于给图片添加目标边框

//box1={"position":{"minX":0.1,"minY":0.2,"maxX":0.3,"maxY":0.4},"class\_id":0,"box\_caption":wandb\_class\_name[0],"score":{"score":b}}：每个边框的格式。position为相对坐标；class\_id和box\_caption用于给边框加标签；wandb\_class\_name={ 0:'name0',1:'name1',... }为类别；有score时可以根据数值筛选框

//使用wandb\_run.log可添加wandb\_image或列表[wandb\_image1,wandb\_image2,...]，添加多张图片会加上筛选功能和index筛选项，每一次wandb\_run.log上传都为一个step

**import tensorboard**

原本是tensorflow的配套训练可视化工具，但pytorch1.2版本以后也可以使用，需要安装pytorch

**from torch.utils.tensorboard import SummaryWriter**

tensorboard\_run=SummaryWriter('tensorboard')：初始化。'tensorboard'为保存路径

**import tensorflow**

(pip install tensorflow：CPU版本)

(pip install tensorflow-gpu：GPU版本，要有对应的cuda版本)

与cuda版本对应：<https://www.tensorflow.org/install/source#gpu>

默认动态图机制，tf.tensor引用方式和数组一样

tf.tensor张量：可以运行在GPU和TPU上，小数默认32位浮点，当运行在CPU时和numpy数组共用内存，运行在GPU时从内存中复制一份到GPU上，多维张量在内存中以一维数组存储

tf.config.experimental.list\_physical\_devices('CPU'/'GPU')：查看CPU/GPU

tf.config.experimental.set\_memory\_growth(GPUS[0],True)：允许运行时分配更多内存

tf.constant(value,dtype):创建tf.tensor常量张量，value可为数字、列表、numpy数组。dtype=tf.float/int...；张量显示时带<>

tensor.numpy():将tensor张量变为numpy数组

tf.shape(a):看形状；tf.size(a):看元素数；tf.rank(a):看维度

tf.cast(tensor,dtype=tf.float32)转换数据类型

tf.reshape(tensor,(m,n,k...)):转换tensor形状

tf.zeros(shape,dtype):全0张量 | tf.ones(shape,dtype):全1张量

tf.fill(shape,a):全a张量

tf.random.normal(shape,mean,stddev):根据正态分布随机初始化值。shape是初始化形状，mean是均值默认0；stddev是标准差默认1

tf.reshape(tensor,shape):改变形状

tf.expand\_dims(tensor,axis):增加维度

tf.gather(tensor,axis,indices):按索引值提取张量

tf.equal(A,B)分别比较两个矩阵各数是否相等,返回True/False矩阵

tf.matmul(A,B)将两个矩阵相乘

tf.add(A,B):逐元素相加。tf.subtract(A,B):逐元素相减

Variable1.assign\_sub(A):逐元素相减，更新梯度时使用

tf.multiply(A,B):逐元素相乘，tf.divide(A,B):逐元素相除

tf.pow(A,B):逐元素求幂，tf.square(A):逐元素求平方

tf.sqrt(A):逐元素开平方，tf.exp(A):逐元素计算e的A次方

tf.one\_hot(indices,depth):创建深度为depth的独热编码，独热编码使取值不具有偏序性

tf.where(array<a,b,c):array中元素小于a返回b，否则返回c

tf.square(A)：求A中每个元素的平方

tf.argmax(A,axis),axis=0:求每列最大值的索引(axis默认为零)

tf.reduce\_mean(A,axis)：axis=0求每列的均值(不指定求所有的均值)

tf.reduce\_sum(A,axis)：axis=0求每列的和(不指定求所有的和)

tf.reduce\_max(A,axis)：axis=0求每列的(不指定求所有的最大值)

with GradientTape as GT://loss=...//grads=GT.gradient(loss,[w,b]):求w,b的损失值

**tf.keras：Tensorflow的标准高阶API，keras也有单独用的库**

keras:开源人工神经网络库，安装tensorflow时自动安装kera

compose(A,B,C):依次执行A,B,C4层，前一个输出作为后一个输入

**方式1：**model=tf.keras.Sequential()

/ model.add(layer1)：添加层，但Sequential不能构建复杂的结构

或者：model=tf.keras.Sequential(layer1,layer2...)

tf.keras.Sequential(\*(layer1 for i in range(n)))：构建相同层

**方式2：**input\_shape=tf.keras.layers.Input(...)

/ x=tf.keras.layers...(...)(input\_shape)

/ x=tf.keras.layers...(...)(x)：不断叠加层，第一层必须为输入层。

/ model=tf.keras.Model(input\_shape,[x1,x2,...],name=''):组合

**方式3：**class model(tf.keras.layers.Layer) // def \_\_init\_\_(self,...) // self.layer1=layer1...：初始化A=model(\_\_init\_\_中参数)时会搭建好模型结构，因此\_\_init\_\_中要把所有用到的层按顺序传入self(tf.keras.layers.Layer为搭建模型的继承类库函数)

// def call(self,input1) // return x：调用A(input1)会直接执行call函数，此时才会确定模型内部的连接方式。对于复杂的网络定义多个类模块嵌套，类中可以使用Sequential函数

**基础网络：**

tf.keras.layers.Flatten()：将多维数据变成一维。输入(batch,...)，输出(batch,n)

tf.keras.layers.Dense(a,activation,name)：全连接层，输入(batch,in\_features)，输出(batch,out\_features)。activation有'relu''softmax''sigmoid'等；input\_shape=(b,)中b是输入一维数据个数

tf.keras.layers.Dropout(0.2,name)：丢弃层，输入/输出(batch,n)

**二维网络：**

tf.keras.layers.Input(shape=(a,a,b),name))：指定输入形状

tf.keras.layers.Conv2D(filters,kernel\_size,strides=1,padding='valid',activation=None,name)：二维卷积层。filters是卷积核数；kernel\_size=(n,n)/n是卷积核大小；strides为步长；padding是扩充边界方式，'same'为不够时扩充0(步长为1时使输入和输出一致，优先扩充右和下)、'valid'不够时舍弃；activation是激活函数(可不加):softmax,sigmoid,tanh,relu,softplus等经典激活函数

tf.keras.layers.BatchNormalization(name)：批归一化层

tf.keras.layers.Activation(...)：经典激活函数。mish函数可通过叠加softplus和tanh激活函数实现

tf.keras.layers.LeakyReLU(alpha=0.05)：LeakyReLU激活函数

tf.keras.layers.MaxPool2D(pool\_size,strides,padding,name)：二维最大池化层。pool\_size=(n,n)，strides为步长

tf.keras.layers.AveragePooling2D(pool\_size,strides,padding,name)：二维平均池化层

tf.keras.layers.concatenate([A,B],axis=3/-1,name)：合并分支

def inception1(x,...)//branch1=...(...)(x)//branch2=...(...)(x)//branch2=...(...)(branch2)//x=...concatenate(...)//return x

input1=...Input(...)//x=...(...)(input1)//x=inception1(x,...)：把分支加入网络

tf.keras.layers.add([A,B],name)：将两个完全一样的层参数相加

tf.keras.layers.UpSampling2D(size=a)：复制上采样层。扩大为a倍

tf.keras.layers.Conv2DTranspose(filters,kernel\_size,strides=1,padding='valid',name)：反卷积上采样层。stride决定放大的倍数；output\_padding为输出后每条边填充0

**二维损失：**

tf.keras.losses.categorical\_crossentropy(A,B):求A,B每行的交叉熵

**一维网络：**

tf.keras.layers.LSTM()：创建LSTM模块

tf.keras.layers.Conv1D(filters,kernel\_size,strides,padding,activation,name)：卷积层

**数据处理：**

**设置：**

tf.random.set\_seed(a)：设置全局随机种子a，使实验结果一样

**模型训练：**

model.summary():查看模型摘要。model.compile(loss,optimizer,[metrics1,metrics2]):配置模型；loss有均方差：'mse'，多分类交叉熵：'categorical\_crossentropy'(标签独热编码)，'sparse\_categorical\_crossentropy'(标签自然顺序码)，'binary\_crossentropy'(二分类交叉熵)；optimizer有'sgd''adagrad'，'adadelta'，'adam'；metrics有'accuracy'(标签数值，预测数值)'binary\_accuracy'(二分类，标签数值，预测概率)，'categorical\_accuracy'(多分类，标签独热编码，预测独热编码)，'sparse\_categorical\_accuracy'(多分类，标签数值，预测独热编码)

model.fit(train\_x,train\_y,batch\_size,epochs,shuffle,validation\_split,validation\_freq,verbose):训练模型，可反复运行。batch\_size是把总数据分成几份，一般取；epoch是训练次数；shuffle默认True是每次训练前打乱数据；validation\_split=0-1是分配测试集百分比默认为0；validation\_freq是训练几轮显示一次默认为1；verbose是显示方式：0为不显示，默认为1方式显示。

model.history.history:查看训练时记录的各指标。model.history.history['loss']得到损失函数历史列表

model.evaluate(test\_x,test\_y,,batch\_size,verbose):用测试集评估

**模型保存与加载：**

模型保存时要保存model，不用保存moedl.eval()

model.predict(x,batch\_size,verbose):使用模型

model.load\_weights(path.h5)：读取网络参数

model.save\_weight(path.h5)：保存网络参数

model.save(filepath,overwrite,include\_optimizer,save\_format):保存模型；filepath是保存路径默认当前路径；overwrite默认为Ture表示保存的文件名相同时覆盖原文件；include\_optimizer默认为True表示保存优化器当前状态(训练时间过长时，下次可以接上断点继续训练)；save\_format默认为None,None保存为HDF5格式，文件后缀.h5，'tf'保存为Tensorflow特有格式，会出现多个子文件

tf.saved\_model.load(model\_file):加载保存的模型文件。model\_file为一个特殊格式的文件夹

**import torch**

pytorch官网：<https://pytorch.org/get-started/previous-versions/>

(pip install torch：cpu版本，GPU版本详见官网，需要有对应的cuda)

torch中tensor是并行计算。默认float32和int64，CPU上支持float32，GPU支持float32和float16(部分GPU只支持float32，用float16不报错，但实际是转化为float32计算)。tensor中的运算要比np快

.cuda()/.cpu()/.to('cpu'/'cuda'/'cuda:0'/torch.device(...),non\_blocking=False)：将数据/模型放到GPU/CPU上计算。'cuda:0'代表放入的GPU设备号，单卡时cuda和cuda:0等效；non\_blocking为是否放到锁存上(主机内存够时放到锁存上会快一点，但第一次放入gpu锁存会比较慢，因此在预测时不启用)。np数组只能在CPU上运算，张量能在CPU/GPU上运算。对模型的操作都会共用一个模型，除非使用copy.deepcopy(model)得到新的模型，新模型的设备等信息完全一致，要注意显存消耗

torch中tensor1=tensor2是共用内存的，会共同改变

torch大部分函数的用法与numpy一致，区别在于numpy中的axis对应torch中的dim，numpy中的keepdims对应torch中的keepdim

**属性：**

tensor.shape：张量形状

tensor.device：张量所在的设备

**创建和转换：**

tensor1=tensor2.clone()：复制张量。torch中张量直接用=是共用内存的

tensor.to(torch.float32)：数据类型转换

tensor.type(torch.int32/float32)：转换数据类型为int32/float32

torch.tensor(array,dtype=torch.float32,device='cpu')：创建torch.tensor张量，value可为列表、numpy数组等，建议将对象转为np再转为张量更快速。device='cuda'为在GPU上创建tensor

torch.from\_numpy(array)：将numpy数组转化为torch.tensor张量

tensor.tolist()：转换为列表

tensor.repeat(a,b)：在0维复制a倍，在1维复制b倍，(a,b)个数要与tensor一致

tensor.numpy():将tensor张量变为numpy数组。当tensor放到模型中计算后，转为numpy需要阻断反向传播：tensor.detach().cpu().numpy()

torch.rand(a,b,dtype)：创建形状为(a,b)、0-1均匀分布的随机张量

torch.randn(a,b,dtype)：创建形状为(a,b)、0-1标准正态分布的随机张量

torch.randperm(a)：相当于torch.arrange再打乱顺序

torch.zeros\_like(tensor)：类似torch.zeros，但是创建张量的形状、dtype和device与tensor一致。torch.ones\_like、torch.full\_like同理

**基本运算：**

tensor[mask]：mask为布尔张量，可以筛选出mask中为True的值

torch.concat/cat([tensor1,...],dim=0)：不增加维度的方式合并张量

torch.stack([tensor1,tensor2...],dim=0)：增加维度的方式合并张量

torch.chunk(tensor,2,dim=0)：拆分张量。torch.concat的逆过程

torch.clamp(input,min,max)：将input中的值压缩到min到max之间

tensor.sigmoid()：进行sigmoid函数运算

torch.argsort(tensor,axis=0,descending)：根据axis维度的值排序，返回下标

torch.norm(tensor,dim=0)：求tensor在dim维度的模长

torch.sign(tensor)：小于0的值变为-1，大于0的值变为1，0不变

torch.abs(tensor)：取绝对值。tensor.abs()同理

torch.pow(tensor,a)：次方运算。a为次方数

torch.square(tensor)：平方运算

torch.sqrt(tensor)：开方运算

torch.rsqrt(tensor)：对tensor开根号后再求倒数

np.max/min/sum/mean/var/std(array,axis,keepdims=False)：求axis维度的最大值/最小值/和/均值/方差/标准差(不指定axias时求所有值的)。若array形状为(2,3,4)，axis=0，则结果形状为(3,4)，keepdims=True时为(1,3,4)

torch.nonzero(tensor==1).squeeze(1)：找到符合条件的元素下标

value,index=torch.sort(tensor,dim=0,descending=False)：根据dim维度的值排序，返回元组(排序后的值,对应的索引)。descending=True时为降序排序

torch.matmul(tensor1,tensor2)：矩阵相乘。当tensor1为3维时将0维当成batch；torch.matmul((batch,a,m),(batch,m,b))结果为(batch,a,b)；torch.matmul((batch,n,a,m),(batch,n,m,b))结果为(batch,n,a,b)

torch.ne(tensor1,tensor2)：逐元素比较是否相等，返回同样大小的布尔张量。tensor2可以只为一个数字

torch.nn.utils.rnn.pad\_sequence(label\_list,batch\_first=True,padding\_value=0)：填充、合并标签、转为张量。batch\_first=True时批量才会在0维，默认False；padding\_value为在短的标签后面填充的值，默认0

torch.fft.rfft(x,dim=0)：傅里叶变换。返回元素长度为dim//2+1

torch.fft.rfftfreq(a)：计算采样频率。a为整数，返回列表长度为a//2+1

boll\_tensor=torch.tensor((tensor<a)&/|(tensor>b))：返回布尔张量

torch.nonzero(boll\_tensor).squeeze()：返回元素为真的张量索引

**维度转换：**

tensor.reshape(a,b)：变换张量形状。为(-1,)时变一维。与reshape类似的是view函数，通常reshape和view的效果一致，不需要额外内存，但某些需要额外内存的情况下view不能使用，而reshape依然可以使用

tensor.permute(0,2,1)：更改维度，0,2,1表示将原来的1,2维换位

tensor.unsqueeze(a)：在a维添加一个维度

tensor.squeeze(a)：当a维度大小为1时去掉

tensor.T：转置

**torch.nn：搭建神经网络**

对于网络需要输入形状时，输入形状后系统会把batch理解为批量，然后并行计算。网络搭建时最好不用if判断选择网络结构，后续转为onnx模型时容易出错

**方式1：**model=torch.nn.Sequential()

//model.add\_module('name',layer1)：添加层。但不能构建复杂结构

或者：model=torch.nn.Sequential(layer1,layer2...)：名字自动生成

torch.nn.Sequential(\*(layer1 for i in range(n)))：构建相同层

**方式2：**class model(torch.nn.Module) // def \_\_init\_\_(self,...) // self.layer1=layer1...：初始化A=model(\_\_init\_\_中参数)时会搭建好模型结构，因此\_\_init\_\_中要把所有用到的层按顺序传入self(torch.nn.Module为搭建模型的继承类库函数)

//def forward(self,x) // return x：调用A(x)会直接执行forward函数，此时才会确定模型内部的连接方式。对于复杂的网络定义多个类模块嵌套，类中可以使用Sequential函数。forward中用if可能导致onnx导出失败

self中定义的参数分为会更新、不更新和会保存、不保存，会保存是指使用torch.save等保存模型时会记录的信息，有的参数只是训练时用就不需要保存。self.linear这样网络层会更新会保存，使用self.name=value定义的参数不更新不保存，使用self.register\_buffer(name,value)定义的参数不更新会保存

self.register\_buffer(name,value)：定义不更新但会保存的参数，使用self.name调用定义的参数。register\_buffer是继承类中自带的函数

self.training：判断模型是否在训练。默认情况下为True(对应model.train)，使用model.eval()后变为False。training为继承类中自带的参数

**基础网络：**

torch.nn.Parameter(tensor)：将张量转换为可训练的参数

torch.nn.Sigmoid/Tanh/ReLU/LeakyReLu/SiLU/...(inplace=False)：添加激活函数.inplace=True时，节约内存但会修改原来的数值，比如：x1=function(x)，执行完后x的值也被修改了

torch.nn.Flatten()：将多维数据变成一维。输入(batch,...)，输出(batch,n)

torch.nn.Linear(in\_features,out\_features,bias=True)：全连接层。输入(batch,in\_features)时，输出(batch,out\_features)。输入(batch,n,in\_features)时，输出(batch,n,out\_features)，此时各变量n之间没有交互，但共用一个linear权重

torch.nn.Dropout(0.2)：丢弃层，输入/输出(batch,n)

torch.nn.Softmax(dim)：将输出结果归一化。dim是进行归一化的维度。如果是单类别不能用softmax而要用sigmoid。softmax函数之后再加网络输出层可能会出错

torch.nn.Upsample(scale\_factor=a)：上采样层。a为扩大多少倍

torch.nn.LayerNorm(normalized\_shape)：归一化。通常在自然语言处理和时序预测中使用。输入为(batch,dim...,feature)，当normalized\_shape=(dim...,feature)时对batch中每个样本分别进行归一化，当normalized\_shape=feature时对batch中每个样本的每个最后维度分别进行归一化

**一维网络：**

torch.nn.Conv1d(in\_,out\_,kernel\_size,stride=1,padding=0,dilation=1,bias=True)：卷积层。输入(batch,channel,n)，in\_/ou\_为输入/输出维度，对应channel；stride和padding对应n，stride为卷积步长(卷积后四周会缩短stride//2)；padding=a在序列首尾填充a个0；dilation为卷积时卷积核各点的距离(形成空洞卷积)；bias=True在输出添加可学习偏差

output,(h\_n,c\_n)=torch.nn.LSTM(input\_size,hidden\_size,num\_layers,dropout=,bidirectional=False)：创建LSTM模块。input\_size为输入的长度；hidden\_size为输出的长度；num\_layers为层数数；dropout为在除最后一层外都加丢弃层；bidirectional为是否双向传播。LSTM加在卷积后效果很差

torch.nn.BatchNorm1d(in\_,track\_running\_stats=True)：批归一化层。输入(batch,in\_,feature)或(batch,n)，会对1维进行归一化。in\_为输入的维度；track\_running\_stats=True时会根据所有训练数据进行归一化，False时只会根据当前输入数据，如果训练数据差别比较大时建议设为False

torch.nn.MaxPool1d(kernel\_size,stride,padding)：最大池化层

torch.nn.Embedding(num\_embeddings,embedding\_dim)：将词索引向量编码成词向量，输入(batch,n)，输出(batch,n,embedding\_dim)。num\_embeddings为词表总数，输入词索引的值要<词表总数，词索引向量(batch,n)可以是任意长度；embedding\_dim为编码后单个词向量的长度

transformer=torch.nn.Transformer(d\_model=512,batch\_first=True,num\_encoder\_layers=6,num\_decoder\_layers=6,nhead=8,dim\_feedforward=2048,activation='relu',dropout=0.1,custom\_encoder,custom\_decoder)：定义transformer结构，包含编码器和解码器。注意要单独加入词向量编码层、位置编码层、词向量解码层。d\_model为输入词向量的长度embedding\_dim；batch\_first=True要手动设置，默认False，代表输入的0维度为批量；num\_encoder\_layers和num\_decoder\_layers为编解码层中的编解码模块堆叠数，默认6；nhead为编解码模块中多头注意力层MultiheadAttention中的并行注意力层数，默认8；dim\_feedforward为编解码模块中注意力层后面的全连接层长度，默认2048；activation为全连接层的激活函数，默认relu，建议用gelu；dropout为丢弃概率，默认0.1；custom\_encoder和custom\_decoder可以指定自定义的编码器和解码器

transformer(src,tgt,tgt\_mask,src\_key\_padding\_mask,tgt\_key\_padding\_mask)：使用网络。src和tgt为编码器和解码器的输入，形如(batch,n,size)，输出也为(batch,n,size)，预测时只取(batch,-1,size)作为输出，训练时有掩码矩阵因此可以批量计算损失；tgt\_mask为解码器的掩码矩阵；src\_key\_padding\_mask和tgt\_key\_padding\_mask为编解码器的掩码向量。输出为(batch,n-1,channel)。通常tgt=label[:,-1]，tgt\_lable=label[:,1:]

memory=transformer.encoder(src,src\_key\_padding\_mask)：单独使用编码器

output=transformer.decoder(tgt,memory,tgt\_mask,tgt\_key\_padding\_mask)：单独使用解码器。memory不变的情况下单独使用可以避免重复计算

torch.nn.Transformer.generate\_square\_subsequent\_mask(a)：创建a\*a的掩码矩阵。掩码矩阵左下边和对角线为0，右上边为-torch.inf，之所以用-inf是因为经过softmax时会得到0

torch.nn.MultiheadAttention(embed\_dim,num\_heads,batch\_first=False,dropout=0., bias=True)：多头注意力机制。d\_model为输入词向量的长度embedding\_dim；nhead为并行的注意力层数；batch\_first=True要手动设置，默认False，代表输入的0维度为批量

**二维网络：**

torch.nn.Conv2d(in\_,out\_,kernel\_size,stride=1,padding=0,dilation=1,bias=True)：卷积层。输入(batch,channel,m,n)，in\_/out\_为输入/输出维度，对应channel；stride和padding对应m、n，stride为卷积步长(卷积后四周会缩短stride//2)；padding=a在图片周围填充a个0；dilation为卷积时卷积核各点的距离(形成空洞卷积)；bias=True在输出添加可学习偏差

torch.nn.BatchNorm2d(in\_,eps=1e-05,momentum=0.1,track\_running\_stats=True)：批归一化层。输入(batch,in\_,feature1,feature2)，会对1维进行归一化。in\_为输入的维度；eps为防止分母为零的小值，推荐0.001；momentum为原均值和标准差的占比，推荐0.03，new=(1-momentum)\*new+momentum\*last。track\_running\_stats=True时会根据所有训练数据进行归一化，False时只会根据当前输入数据，如果训练数据差别比较大时建议设为False

torch.nn.MaxPool2d(kernel\_size,stride,padding,dilation=1)：最大池化层。stride默认为kernel\_size的值，dilation>1为空洞卷积

torch.nn.ConvTranspose2d(in\_,out1,kernel\_size,stride=1,padding=0,output\_padding=0)：反卷积上采样层。stride决定放大的倍数；output\_padding为输出后每条边填充0

torch.nn.AdaptiveAvgPool2d(n)：将尺寸进行平均变为1。如n=1时输入(batch,16,20,20)，输出(batch,16,1,1)

**损失：**

通常时间序列预测任务用mse损失；分类任务中，单标签二分类、多标签多类别使用BCEWithLogitsLoss，多标签单类别、多模态的特征损失中使用CrossEntropyLoss。交叉熵损失相比mse的梯度有利于收敛，只追求分布(归一化后数值)相同

torch.nn.L1Loss()(a,b)：平均误差。预测值和真实值绝对误差和的平均

torch.nn.MSELoss()(a,b)：均方误差。预测值和真实值误差平方和的平均

torch.nn.BCELoss()(pred,true)：交叉熵损失(Binary Cross Entropy)。pred,true为0-1的值，独热编码。输入的weight为权重，要和标签形状一致；reduction：'mean'返回损失的平均值

torch.nn.BCEWithLogitsLoss()(pred,true)：先进行sigmoid归一化再进行交叉熵，推荐使用。如果已经使用过sigmoid要使用BCELoss，两个sigmoid容易导致不稳定

torch.nn.CrossEntropyLoss()(pred,true)：先进行softmax归一化再进行交叉熵损失。单标签时不能使用，因为不能进行softmax。输入为pred=(batch,n)，true=(batch,n)，dtype=torch.float32；当true为独热编码序号(如词索引向量)时，true可以直接传入([a,b,...])，dtype=torch.int64，不用先变为独热编码

**模型网络层操作：**

for module in model.modules():：遍历模型所有的网络结构。比如网络为CBS，会依次遍历“CSB总结构、Conv2d、BatchNorm2d、SiLU”四个模块

if isinstance(module,torch.nn.BatchNorm2d):：判断当前模块是否为BatchNorm2d。其他网络层同理

Conv2d层参数：module.weight.data

BatchNorm2d层参数：module.weight.data、module.bias.data、module.running\_mean、module.running\_var。BatchNorm2d层module.weight.data参数大小可作为模型剪枝的依据，剪枝模型赋值时后面要加.clone()

Linear层参数：module.weight.data

**数据处理：**

使用Dataset-Dataloader方法，训练时一边训练一边取数据、预处理、合成批量，CPU数据处理和GPU模型推理并行，节约数据处理时间

一般在数据合并时启用pin\_memory=True，for循环取出数据后再放到GPU上。也可以在collate\_fn中将数据统一放到GPU上，但使用多个num\_workers时会比较麻烦

class torch\_dataset(torch.utils.data.Dataset) // def \_\_init\_\_(self,参数) // def \_\_len\_\_(self) // return len(训练数据长度) // def \_\_getitem\_\_ (self,index) // ... // return image,label：通过继承定义数据集类。\_\_len\_\_为数据集的长度。\_\_getitem\_\_中写入数据读取和处理的函数，返回结果，index为系统参数，会自动在\_\_len\_\_长度内选取一个索引。在\_\_getitem\_\_中只对单个的数据处理，实际调用时DataLoader函数会用多个进程在模型训练时提前并行处理多个数据，最终合成一个批量

dataset=torch\_dataset(参数)：定义数据集的处理方式

dataloader=torch.utils.data.DataLoader(dataset,batch\_size,shuffle,drop\_last,num\_workers,pin\_memory=False,sampler=None,collate\_fn)：根据Dataset批量划分训练数据。batch\_size为批量大小；shuffle=True每次取数据前打乱数据；drop\_last=True时如果最后一批数据不满足批量时舍弃，False时把剩余数据当作一个批量；num\_workers为有几个进程处理数据，0为所有数据加载到主进程，可以开多个以保证数据预处理的效率，预测时一般只用一个；pin\_memory=True把数据放到CPU锁存上，之后再放到GPU上的时间会加快(此时\_\_getitem\_\_或collate\_fn中不能先把数据放到GPU上)；sampler为取数据的方式，一般只在分布式训练时设置sampler=torch.utils.data.distributed.DistributedSampler(dataset)，同时设置shuffle为False，否则各GPU会运行相同的dataloader，设置sampler会让各个GPU取的index错开；collate\_fn为合成数据的方式，默认将每个\_\_getitem\_\_返回的结果合在一起并转为张量(增加在0维上)

//自定义collate\_fn：def collate\_fn(getitem\_list): // return train\_batch,true\_batch。当执行完batch个\_\_getitem\_\_后，每个\_\_getitem\_\_返回的值会合成元组，所有\_\_getitem\_\_返回的值会合成列表作为getitem\_list传入函数。可以定义在torch\_dataset类中：def collate\_fn(self,getitem\_list)，使用方式为collate\_fn=dataset.collate\_fn

//使用多个num\_workers时，相当于把dataloader分为num\_workers份，每个线程中在每轮训练时都会重新进行self的初始化

for item,(train\_batch,true\_batch...) in enumerate(dataloader)：取出准备好的数据(自动转为张量),每次for循环都是新的一轮。加enumerate时前面一定要带()区分item

**训练设置；**

os.environ["CUDA\_VISIBLE\_DEVICES"]='1,2,3'：放在import os后、import torch前，指定哪些GPU可以使用，指定后GPU的本地编号为0、1、2

torch.cuda.is\_available()：判断cuda是否可用返回True或False

torch.manual\_seed(种子号)：为CPU设置随机种子

torch.cuda.manual\_seed(种子号)：为当前GPU设置随机种子

torch.cuda.manual\_seed\_all(种子号)：为所有GPU设置随机种子

torch.backends.cudnn.deterministic=True：固定每次返回的卷积算法

torch.backends.cudnn.enabled =True：cuDNN使用非确定性算法

torch.backends.cudnn.benchmark=True：在训练前cuDNN会先搜寻每个卷积层最合适实现它的卷积算法，加速运行；但对于复杂变化的输入数据，可能会有过长的搜寻时间，对于训练比较快的网络建议设为False

**模型文件：**

.pt/.pth/.pkl：pytorch模型权重文件，3者只是文件后缀不同，是同一种文件。需要注意权重文件通常是一个字典，只包含网络权重参数和一些辅助信息，要加载完整的模型还需要模型结构的代码，代码可以是本地自定义或导入模型库中的，但代码中的网络和名称需要与保存模型时完全一致才能合在一起

.safetensors：pytorch模型权重文件，额外包含用于验证数据完整性的元素，在huggingface中的很多大模型都使用.safetensors格式传输

.bin：二进制格式权重文件，需要处理后才能在pytorch中使用，在huggingface中的很多大模型会使用.bin格式传输

**加载和保存模型：**

分布式训练时会在模型外面套一层结构，保存模型时使用model=model.module取出

model\_dict={'model':model,...}：将模型和其他各类信息放到模型字典里

torch.save(model\_dict,save\_path)：保存模型搭建信息(不是真正的模型)、模型参数。torch.save本质是让目标序列化，可以保存包含模型、学习率等信息的字典，此时torch.load加载后就是一个字典

torch.load(model\_path,map\_location='cpu'/'gpu',weight\_only=False)：加载模型，需要目录下有保存模型时的代码。map\_location等同于.to(device)；如果保存的模型为一个嵌套的字典等，weight\_only要为False

torch.save(model.state\_dict(),save\_path)：保存模型参数

model.load\_state\_dict(torch.load('state\_dict.pt'))：加载模型参数

**模型状态：**

model.train()：启用训练模型。会开启dopout；影响BatchNorm计算方式，相同输入在train和eval下会有所不同

model.eval()：启用推理模式。会关闭dropout；影响BatchNorm计算方式

with torch.no\_grad():：取消梯度计算，不存储梯度参数减少内存，加快推理速度。使用这种方式时，即使只在局部使用也会阻断整个模型的梯度计算，无法用于冻结训练

for param in model.parameters() // param.requires\_grad\_(False)：长期关闭所有参数层的梯度，可以选择性关闭参数层实现冻结训练

BatchNorm：train模式会根据输入数据动态归一化，eval模式会使用固定的值

训练数据太少时归一化也会过拟合，train和eval的区别会很大，会出现train很准但eval很差。训练批量减少为1可以改善归一化的问题

**模型训练：**

torch.optim.SGD(model.parameters(),lr)：SGD优化器。model.parameters()为模型的网络参数；lr为学习率一般为0.01

torch.optim.Adam(model.parameters(),lr,betas=(0.9,0.999),eps=1e-08,weight\_decay=0)：Adam优化器。model.parameters()为模型参数；lr为学习率一般为0.001；betas为计算梯度以及梯度平方的运行平均值的系数；eps为防止分母为0；weight\_decay为L2正则化的衰减权重，不填则不使用L2正则化，L2正则化会加在损失函数中使用，但不显示在损失结果中。调参时weight\_decay可以从0.0001开始逐渐增加，直到找到最佳值。yolov7中betas=(0.937,0.999)，weight\_decay=0.0005。大语言模型中weight\_decay=0.0001

model.train().float()：加在训练前，启用批归一化层和丢弃层

optimizer=torch.optim.Adam(model.parameters(),lr)：优化器

1，pred=model(data)：将数据传入到模型中进行运算，会自动求导

2，loss\_value=loss(pred,true)：根据输出计算损失。train\_loss.item()：损失的值

3，loss\_value.backward()：将损失反向传播，计算当前梯度

4，optimizer.step()：根据梯度更新参数

5，optimizer.zero\_grad()：将梯度设为0。每次计算的梯度会积累，需要梯度清零

**学习率保存和加载：继续训练模型**

optimizer=torch.optim.Adam(model.parameters(),lr)：优化器

model\_dict['optimizer\_state\_dict']=optimizer.state\_dict()：记录优化器参数

optimizer.load\_state\_dict(model\_dict['optimizer\_state\_dict'])：加载优化器参数，但此时的优化器optimizer要和之前的优化器保持一致

**混合float16精度训练：**

使用混合精度时，很小的数值会被截断为0，如果网络中有log(0)会导致输出全为nan，使用sigmoid等归一化需要注意。使用混合精度训练时，要使用BCEWithLogitsLoss，不能使用BCELoss。在windows上使用混合精度可能会出现nan，但在linux上正常

amp=torch.cuda.amp.GradScaler()：自动使用混合精度训练装置

with torch.cuda.amp.autocast():：使用混合精度训练

amp.scale(loss\_value).backward()：替代原来的loss\_value.backward()

amp.step(optimizer)：替代原来的optimizer.step()

amp.update()：装置本身也需要更新

optimizer.zero\_grad()：梯度清零

**指数移动平均(EMA)：给予近期数据更高权重的平均方法**

指数移动平均(EMA)：将原模型复制一份作为ema模型，在训练中平滑的记录原模型参数，减少极端值的影响，关闭梯度，不影响模型训练，只影响最终结果。训练中每次原模型更新参数后，ema模型会根据原模型更新自己的参数。分布式时只在主机上创建和更新ema模型

ema=model\_ema(model)：model\_ema为自定义的类

ema.update\_total=0：新模型训练时设置为0，每次迭代后使用ema.update(model)时会自动+1，继续训练时赋值为之前保存的update\_total

ema.update(model)：每次原模型更新参数后，调整ema模型参数

model=ema.ema.eval()：验证和保存模型时，使用ema模型

**分布式训练：只有多卡下能使用**

分布式训练时，会自动把代码复制到各GPU上运行(模型也会创建多份)，只有local\_rank有区别，0为主设备，因此日志、模型验证和保存等只在0时才运行。不同GPU上的代码独立运行，但在特定地方会有改变：在loss\_batch.backward()时会自动汇总所有模型的梯度并计算平均值，再返回给各GPU更新自己的模型；在dataloader中要传入DistributedSampler，会自动把传入的总批量拆分给各GPU，但验证时只在主设备上运行，总批量要除以设备数

torch.nn.DataParallel：单机多卡数据并行训练(训练效率无法达到最高)

torch.nn.parallel.DistributedDataParallel：多机多卡数据并行训练(推荐)

torch.cuda.device\_count()：统计使用的GPU数

1，local\_rank=int(os.environ["LOCAL\_RANK"])：当前GPU在服务器的GPU排号

int(os.environ["RANK"])：当前GPU在所有服务器的GPU排号

2，torch.distributed.init\_process\_group(backend)：分布式训练初始化。backend为GPU上模型同步更新参数时通信使用的后端，一般为'ncll'

3，dataloader中设置sampler=torch.utils.data.distributed.DistributedSampler(dataset)，同时shuffle=False

3，model=model.to('cuda')：将模型放到当前设备上，n为GPU排号。或者torch.cuda.set\_device(local\_rank)

4，torch.nn.parallel.DistributedDataParallel(model,device\_ids=[local\_rank], output\_device=local\_rank,find\_unused\_parameters=False))：对模型进行封装。放在ema前。device\_ids是指定设备的id，output\_device是指定设备的位置，通常都是local\_rank；find\_unused\_parameters=True时会找出模型中没有梯度的参数，在更新参数时会加快速度，如果网络中有很多没有梯度的参数时应开启，否则反而会略微降低速度

5，torch.distributed.barrier()：每轮训练结束后同步设备，比如主进程的GPU要进行验证需要等待。训练时的更新参数会自动同步

6，在命令行传入参数来使用分布式多线程：python -m torch.distributed.launch --master\_port 9999 --nproc\_per\_node 2 文件名.py。master\_port为各GPU之前的通信端口(不被占用就行)，nproc\_per\_node为使用的GPU数。使用args时可以留有参数--local\_rank，执行命令后会自动分配每个GPU的local\_rank给每个GPU的代码

**清理显存空间：**

del ...：删除一些无用但还在显存上的变量

torch.cuda.empty\_cache()：释放一些没用的显存占用，有较明显的效果

**模型测试：**

torchvision.ops.nms(boxes,scores,iou\_threshold)：官方非极大值抑制，返回索引index组成的张量,使用boxes[tensor]取出结果。boxes为(x\_min,y\_min,x\_max\_y\_max)。torchvision为pytorch辅助库

model.eval()：训练的时候归一化层是动态的，如果有丢弃层也是启用的。使用model.eval()可以固定归一化层参数，关闭丢弃层。但也会导致模型精度下降，训练时的批量越小，影响越明显

1，model.eval().float()/.half()：转为测试时的精度(需要GPU且支持float16)

2，with torch.no\_grad()：测试模型时关闭反向求导

3，pred=model(data).detach().cpu()：阻断反向传播并放到cpu上

4，val\_loss=loss(pred,true)

**模型部署：**

详见onnx和tensorrt库

**import torchvision：pytorch辅助库**

安装要与torch版本对应

**import timm：集成了很多pytorch的图像模型**

github：<https://github.com/rwightman/pytorch-image-models>

运行后下载的模型文件位置在本地用户中

timm.list\_models(pretrained=False)：查看所有模型名称，返回列表。pretrained=True时只查看可传入训练参数的模型，大部分都可传入

timm.list\_models('\*resnet\*')：搜索名称中带有resnet的所有模型，返回列表，没有搜到为空列表

timm.create\_model('模型名',pretrained=False,features\_only=False,in\_chans,num\_classes,exportable)：创建模型。pretrained=True时会下载模型预训练权重参数到本地，可以加快训练速度；features\_only为True时只提取特征；in\_chans指定输入维度，num\_classes指定输出类别数(features\_only=False才生效)，模型输出没有经过归一化；exportable=True方便导出onnx

**import torch\_geometric**

API文档：<https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html>

dataset=torch\_geometric.datasets.数据集名()：下载并加载官方数据集。torch\_geometric.datasets库中收录了所有开源的数据集

dataset=torch\_geometric.datasets.KarateClub()：加载俱乐部成员数据集。dataset.data里面包含数据(x=(34, 34),edge\_index=(2,156),y=(34,), train\_mask=(34,))，x为特征矩阵，edge\_index为邻接关系，y为分类标签，train\_mask为需要计算损失的掩码

data=torch\_geometric.data.Data(x,edge\_index)：创建标准格式的数据

dataloader=torch\_geometric.loader.DataLoader(data,batch\_size,shuffle=True)：简便的使用dataloader

**网络：**

torch\_geometric.nn.GCNConv(in\_channels,out\_channels,bias=True)：定义图卷积层。in\_channels为输入通道数；out\_channels为输出通道数；bias=True在输出添加可学习偏差。权重矩阵为(in\_channels,out\_channels)

torch\_geometric.nn.GCNConv(...)(x,edge\_index)：使用网络。x为特征矩阵(n,channel)；edge\_index为邻接关系(2,n)，形如([[0,0,1...],[1,2,2...]])，前后两列不能互换位置，比如0认识1但1不一定认识0，0与0本身默认有关系不用列出

torch\_geometric.nn.SAGEConv(in\_channels,out\_channels，aggr='mean'，bias=True)：定义图采样聚合层。in\_channels为输入通道数；out\_channels为输出通道数；aggr为聚合方式默认为'mean'，还有'max'、'lstm'；bias=True在输出添加可学习偏差

torch\_geometric.nn.SAGEConv(...)(x,edge\_index)：使用网络。x为特征矩阵(n,channel)；edge\_index为adj\_t格式数据

torch\_geometric.nn.TopKPooling()

**import pytorch3d**

github：<https://github.com/facebookresearch/pytorch3d>

pip install "git+https://github.com/facebookresearch/pytorch3d.git"

pytorch3d是Facebook Research开源的基于PyTorch框架的3D计算机视觉任务的库

pytorch3d使用obj文件格式，需要obj文件和mtl材质文件(可没有)

pytorch3d中的面都为三角形，非三角形的面会被拆分为三角形

pytorch3d中的基本操作单位为mesh(批量网格)，类似于将多个张量合到一起得到(batch,...)，mesh批量网格的属性为pytorch3d.structures.meshes.Meshes。mesh网格可视化：可以先采样为点数据，再用plt函数画三维点图；或者创建视角得到某一视角的渲染2D图

pytorch3d中有3种纹理对象：纹理顶点TexturesVertex、纹理图集TexturesAtlas、顶点的纹理图片坐标TexturesUV

**from pytorch3d.io import ...**

vertex,face\_information,aux=load\_obj(obj\_path,create\_texture\_atlas=False,texture\_atlas\_size=4,device='cpu')：加载obj模型。vertex(v,3)为顶点坐标；face\_information中包含：面的3个顶点索引verts\_idx(f,3)、法线索引normals\_idx(f,3)、纹理索引textures\_idx(f,3)、材质索引materials\_idx(f,)，顶点索引必须有，其他索引不存在时张量的值全为-1；aux为渲染信息(可能有)，aux.verts\_uvs为顶点uv；create\_texture\_atlas=True时会创建aux.texture\_atlas纹理图集；texture\_atlas\_size为纹理图集尺寸

mesh=load\_objs\_as\_meshes(obj\_path\_list,create\_texture\_atlas=False,device=None)：加载多个obj模型为mesh。device指定设备

save\_obj(save\_path,vertex,face)：保存为obj文件

**from pytorch3d.renderer import ...**

texture\_vertex=TexturesVertex([])：创建批量纹理顶点

texture\_atlas=TexturesAtlas([aux.texture\_atlas])：创建批量纹理图集

texture\_uv=TexturesUV([])：创建批量顶点的纹理图片坐标

**from pytorch3d.ops import ...**

point=sample\_points\_from\_meshes(mesh,num\_samples=10000)：通过均匀采样将mesh转换成点数据，结果为(batch,num\_samples,3)。num\_samples为采样出的点数

**from pytorch3d.structures import ...**

mesh=Meshes(verts=[vertex],faces=[face],texture=None)：建立mesh。texture为纹理信息，可传入TexturesAtlas([aux.texture\_atlas])。通常会先对vertex进行减均值+除以方差的归一化

texture=mesh.textures：mesh中的批量纹理对象

vertex=mesh.verts\_list()[index]：mesh中的第index批量的顶点坐标

face\_verts\_idx=mesh.faces\_list()[index]：mesh中的第index批量的面索引

mesh=join\_meshes\_as\_batch(mesh\_list,include\_textures=True)：将多个mesh合并成批量mesh。mesh\_list为mesh对象列表；include\_textures=True时，如果mesh对象有纹理，则纹理类型要一致，False时忽略纹理

mesh=join\_meshes\_as\_scene(mesh\_list,include\_textures=True)：将批量mesh合并成单个mesh。mesh\_list为mesh对象列表；include\_textures=True时，如果mesh对象有纹理，则纹理类型要一致，False时忽略纹理

**import OCC**

github：<https://github.com/tpaviot/pythonocc-core>

教程：<https://blog.csdn.net/weixin_42755384/article/details/87893697>

安装包：<https://anaconda.org/conda-forge/pythonocc-core/files>

(conda install --use-local 安装包)

pythonocc是opencascade的python封装版本

**import deepspeed**

github：<https://github.com/microsoft/DeepSpeed>

2023年微软发布DeepSpeed训练框架，解决了小显存无法训练大语言模型的问题，并提高大语言模型的训练效率。需要先安装pytorch。windows可能不支持部分功能

**import transformers**

集成了很多pytorch的自然语言处理模型

github：<https://github.com/huggingface/transformers>

模型：https://huggingface.co/models

使用transformers库中的模型时，会自动从huggingface网站下载模型到本地用户/.cache/huggingface/hub中，也可以手动下载后加载。transformers库中的模型为hf格式

**文本编码器**

文本编码器：将文本放入固定模版中，再编码为词索引。通常需要tokenizer\_config.json和vocab.txt等配置文件。不同模型的配置文件不同，不同编码器的内置属性会有区别

tokenizer=transformers.AutoTokenizer.from\_pretrained(官方模型名称/本地模型文件夹位置)：加载模型的文本预处理

tokenizer.add\_tokens([词语])：添加编码组合。比如默认每个字编码单独编码，加入词后会将整个词一起编码

special\_tokens：特殊字符。比如开始符、结束符、其他自定义符号。bos\_token：开始符；eos\_token：结束符；pad\_token：填充符

text\_dict=tokenizer([text1,text2...],return\_tensors='pt',padding=False,max\_length,truncation,add\_special\_tokens=True)：对文书进行预处理。返回字典{'input\_ids':[...],'token\_type\_ids':[...],'attention\_mask':[...]}；input\_ids为字符串编码后的词索引向量；attention\_mask长度等于input\_ids，有值对应的地方为1，后面填充的地方为0。return\_tensors='pt'使返回字典的列表合成torch的张量；padding=True在后面填充0直到与列表中的最长序列相等，padding='max\_length'填充为max\_length；truncation=True将太长的序列截断为max\_length长度；add\_special\_tokens=True会加上开始、结束符。注意不同模型的tokenizer会有变化，开始、结束、填充符等也可能有变化

tokenizer.encode(text,add\_special\_tokens=True)：对文本进行预处理，等同于tokenizer(...)['input\_ids']，很多参数也与tokenizer一致

tokenizer.decode(dict['input\_ids'][i],skip\_special\_tokens=False)：解码为句子(一次只能解码一句)。skip\_special\_tokens=True忽略特殊符号

tokenizer.save\_pretrained('文件夹位置')：保存文本编码器。通常会得到tokenizer\_config.json和vocab.txt等文件

**加载模型**

model=transformers.AutoModel.from\_pretrained(官方模型名称/本地模型文件夹位置).to(device)：加载官方模型/本地模型位置

model=transformers.AutoModelForSequenceClassification.from\_pretrained(官方模型名称/本地模型文件夹位置).to(device)：加载模型

model=transformers.AutoModelForTokenClassification.from\_pretrained(官方模型名称/本地模型文件夹位置).to(device)：加载序列标注模型

model=transformers.AutoModelForQuestionAnswering.from\_pretrained(官方模型名称/本地模型文件夹位置).to(device)：加载问答模型

model=transformers.AutoModelForSeq2SeqLM.from\_pretrained(官方模型名称/本地模型文件夹位置).to(device)：加载翻译模型

model=transformers.pipeline('任务类型',model=model,tokenizer=tokenizer,device,...)：将编解码与模型组合到一起，直接传入句子，每个模型会有自己的自定义参数。

pred=model(input\_batch)：使用模型预测

model.save\_pretrained('文件夹位置')：保存模型，会得到config.json和pytorch\_model.bin文件。用这种方式保存的模型需要用.from\_pretrained的方式加载

torch.save(train\_dict,'last.pt')：保存训练信息。训练时可能要保存训练轮次、学习率参数、ema参数等以便继续训练。使用torch.load加载

**训练**

参考文档：<https://zhuanlan.zhihu.com/p/363670628>

transformers.Trainer(args,tokenizer,model,train\_dataset,eval\_dataset,data\_collator)：简洁化的训练方式。args中要定义很多的训练设置；train\_dataset和eval\_dataset为数据；data\_collator类似torch.utils.data.DataLoader中的collate\_fn

**大语言模型**

提示词模版：[开始符][任务]\n[内容][结束符]\n。不同模型的提示词模版不同，但通常有系统、用户、回答(可能包括思维链)共3个模版。加入历史对话时，只需将历史中用户、回答的内容堆叠在一起。输入模型时，最后一个回答只需要提示内容，比如[...]assistant\n，模型会回答剩下的内容

模型训练：input\_ids在后面填充pad\_token\_id，attention\_mask在后面填充0，label在后面填充忽略符(-100)

transformers.AutoModelForCausalLM.from\_pretrained(model\_path,trust\_remote\_code,torch\_dtype,device\_map,attn\_implementation)：加载大语言模型。trust\_remote\_code要设为True；attn\_implementation启用flash加速库，安装时默认使用v1版本，'flash\_attention\_2'使用v2版本；4位量化模型加载使用load\_in\_4bit=True和bnb\_4bit\_compute\_dtype=torch.bfloat16。适用于大部分模型

prompt=template.format(system=system,input=input\_)：将内容放入提示模版中。system为系统提示，input为用户输入

prompt=prompt+template\_history.format(...)+template\_think.format(...)：多轮对话时追加历史信息，有思维链时可以加入思维链信息

tokenizer.apply\_chat\_template(message,...)：将固定格式的历史消息处理为prompt

input\_ids=self.tokenizer.encode(prompt,add\_special\_tokens=False,return\_tensors='pt').to(self.device)：编码为词索引向量

pred=self.model.generate(input\_ids=input\_ids,generation\_config)：模型推理

result=self.tokenizer.decode(pred[0][len(input\_ids[0]):],skip\_special\_tokens=True)：模型输出

**流式输出**

self.stream=transformers.TextIteratorStreamer(self.tokenizer)：流式输出器

kwargs={'input\_ids': input\_ids, 'generation\_config': self.generation\_config, 'streamer': self.stream}：输入参数

thread=threading.Thread(target=self.model.generate, kwargs=kwargs)：定义输入

thread.start()：执行推理

for str\_ in self.stream: // yield str\_：不断获取模型输出。第一个str\_为模型的输入，剩下的str\_会多个字连在一起输出，可能输出空字符

**clip(2021)**

github：<https://github.com/openai/CLIP>

github：<https://github.com/TWK2022/clip>

(pip install ftfy regex tqdm)

(pip install git+https://github.com/openai/CLIP.git)

2021年openai发布的基于对比学习的多模态模型(pytorch)，用于将文字描述和图片匹配。由一个图片编码模型和一个文本编码模型组成，一张图片经过图片模型得到的特征向量和这张图片的描述经过文本模型得到的特征向量会相近，通过计算余弦相似度可以通过文本找图片、通过图片找文本

注意所有模型的输入数据和输出数据都为float32，即使在GPU上也是

model,image\_deal=clip.load(模型名称/模型位置,device)：加载模型和图片预处理方式。模型名称：ViT-B/16(335M)、ViT-B/32(338M)、ViT-L/14(890M)；device=cuda/cpu

with torch.no\_grad():：模型验证时设置

**图片模型：**

image=image\_deal(PIL.Image.open("image/01.jpg"))：读取图片并预处理为tensor(3,224,224)。要用PIL库读取图片才能使用image\_deal

image\_bacth=torch.stack([image0,image1,...],dim=0)：合成批量

image\_feature=model.encode\_image(image\_batch)：预测图片得到特征向量(batch,n)，n与模型型号有关

**英文文本模型：**

english\_text\_batch=clip.tokenize(["sentence1", "sentence2",...])：英文文本处理tensor(batch,77)

english\_text\_feature=model.encode\_text(english\_text\_batch)：预测英文文本得到特征向量(batch,n)，n与模型型号有关

**图片和文本匹配：**

score=(image\_feature@english\_text\_feature.t()).softmax(dim=1)：得到各图片与文本对应的匹配分数

**中文文本模型：**

中文文本模型(890M)：<https://huggingface.co/IDEA-CCNL/Taiyi-CLIP-Roberta-large-326M-Chinese>

图片模型公用model.encode\_image(image\_batch)，文本模型分开

chinese\_tokenizer=transformers.BertTokenizer.from\_pretrained("IDEA-CCNL/Taiyi-CLIP-Roberta-large-326M-Chinese"/本地下载文件夹)：中文文本预处理方式

chinese\_data=self.chinese\_tokenizer(text,max\_length=77,padding='max\_length',truncation=True,return\_tensors='pt')：中文文本处理

input\_ids=chinese\_data['input\_ids']

attention\_mask=chinese\_data['attention\_mask']

chinese\_model=transformers.BertForSequenceClassification.from\_pretrained("IDEA-CCNL/Taiyi-CLIP-Roberta-large-326M-Chinese")：中文语言特征提取模型

chinese\_text\_feature=chinese\_encode(chinese\_text,input\_ids=input\_ids,attention\_mask=attention\_mask).logits：预测中文文本得到特征向量

score=(image\_feature@chinese\_text\_feature.t()).softmax(dim=1)：得到各图片与文本对应的匹配分数

**nllb(2022)**

github：<https://github.com/facebookresearch/fairseq/tree/nllb/examples/nllb/modeling>

facebook大型语言翻译模型，能够在200种语言之间翻译

tokenizer=transformers.AutoTokenizer.from\_pretrained(r"D:\model\nllb-200-distilled-600M")

model=transformers.AutoModelForSeq2SeqLM.from\_pretrained(r"D:\model\nllb-200-distilled-600M")

model=transformers.pipeline("translation",tokenizer=tokenizer,model=model,truncation, max\_length,src\_lang,tgt\_lang)：加载nllb模型。src\_lang为输入语言，tgt\_lang为输出语言，zho\_Hans为中文简体，zho\_Hant为中文繁体，eng\_Latn为英文

**import flash\_attn**

(pip install flash-attn --no-build-isolation)

github：https://github.com/Dao-AILab/flash-attention

斯坦福团队发布的transformer推理加速库(训练和推理)，有v1(2022)、v2(2023)、v3(2025)多个版本，高版本仅支持高版本cuda和部分GPU。加载大模型时选用相关参数来启用

**model.generate**

generation\_config文档：<https://blog.csdn.net/muyao987/article/details/125917234>

语言生成模型控制输出的集成函数，预测时取代model(...)。大模型预测时不是简单的使用argmax，而是可以用很多复杂的选取方式，这些方式都集成在model.generate中

model.generate(input\_ids=input\_ids,attention\_mask=attention\_mask,generation\_config=generation\_config)：使用模型并得到处理后的最终结果

generation\_config=transformers.GenerationConfig(eos\_token\_id,pad\_token\_id,max\_new\_tokens=None,do\_sample=False,temperature=1,num\_beams=1,top\_k=50,top\_p=1,repetition\_penalty=1)：控制模型生成内容的配置。不同的参数设置会使用不同的搜索算法，默认贪婪搜索一次。do\_sample=True启用随机概率采样，优先级最高也最常用；通常只需调节temperature

generation\_config变化：eos\_token\_id=2,pad\_token\_id=32000

eos\_token\_id：指定模型结束符(有的模型tokenizer没有eos\_token\_id元素)

pad\_token\_id：指定模型填充符，与eos\_token\_id同时使用

max\_new\_tokens：最大生成字词数

do\_sample：根据概率来随机采样选词，默认False为不使用

temperature：影响do\_sample的效果，默认1。值越小则softmax后的数值差距越小

num\_beams：beam search搜索深度，默认1为不使用。使用时可能会出现重复的现象

top\_k：筛选结果中top\_k个概率最高的词，然后再进行采样，默认50

top\_p：筛选结果中概率之和刚好大于top\_p的几个词，默认1。与temperature效果类似，值越小则输出越稳定，值越大则输出越随机，不建议和temperature同时调整

repetition\_penalty：防止模型输出重复的惩罚权重，默认1为不惩罚

**llama2(2023)**

github：<https://github.com/ymcui/Chinese-LLaMA-Alpaca-2>

2023年Meta发布LLaMa2模型，有1.3B、7B、13B三个型号。词表长度：55296

tokenizer变化：开始符为1和<s>，结束符为2和</s>，填充符为32000和<pad>。保存的文件为tokenizer\_config.json、tokenizer.model、special\_tokens\_map.json

tokenizer=transformers.LlamaTokenizer.from\_pretrained(model\_path)：文本预处理

model=transformers.LlamaForCausalLM.from\_pretrained(model\_path,device\_map='auto',low\_cpu\_mem\_usage=True,torch\_dtype=torch.float16,load\_in\_4bit=False,load\_in\_8bit=False)：加载模型。device\_map指定模型使用设备，'auto'为自动分配；low\_cpu\_mem\_usage常为True；load\_in\_4bit和load\_in\_8bit根据情况定

generation\_config变化：eos\_token\_id=2,pad\_token\_id=0

model=model.eval()|.to(device)|.half/float()：模型状态设置

len(tokenizer)：文本预处理的词表长度

model.vocab\_size：模型词表长度

with torch.no\_grad():：模型验证设置

system='You are a helpful assistant. 你是一个乐于助人的助手。'：默认系统提示。用户输入的提示加在system后

template='<s>[INST] <<SYS>>\n{system}\n<</SYS>>\n\n{input} [/INST]'：提示模版

prompt=template.format(system=system,input=input\_)：在输入中加入提示模版。system为系统提示，input为输入问题

template\_history=' {output\_add}</s><s>[INST] {input\_add} [/INST]'：多轮对话时追加的历史信息模版(3个空格)。output\_add为上一轮的回答，input\_add为下一轮的输入

prompt=prompt+template\_history.format(output\_add=output\_add,input\_add=input\_add)：多轮对话时追加历史信息

input\_ids=tokenizer.encode(prompt,add\_special\_tokens=False,return\_tensors='pt').to(device)：模型输入

pred=model.generate(input\_ids=input\_ids,generation\_config=generation\_config)：使用模型预测

result=tokenizer.decode(pred[0])：解码。此时result内容为：输入+回答+结束符

**模型训练：**

模型输入长度通常为1024个词索引。input\_ids、attention\_mask、label长度一致。最终合成批量时：input\_ids在后面填充pad\_token\_id，attention\_mask在后面填充0，label在后面填充-100

prompt\_encode=tokenizer.encode(prompt,add\_special\_tokens=False)：提问数据

output\_encode=tokenizer.encode(output,add\_special\_tokens=False)：回答数据

input\_ids=torch.tensor(prompt\_encode+output\_encode+[eos\_token\_id],dtype=torch.int64)：模型输入

attention\_mask=torch.full\_like(input\_ids,1)：掩码

label=torch.full\_like(input\_ids,-100)：不需要的地方变为-100

label[len(prompt\_encode):]=input\_ids[len(prompt\_encode):]

input\_ids=input\_ids[:max\_length]

attention\_mask=attention\_mask[:max\_length]

label=label[:max\_length]

pred\_batch=model(input\_ids=input\_ids\_batch,attention\_mask=attention\_mask\_batch,labels=label\_batch)：模型推理

loss\_batch=pred\_batch.loss：当传入labels时模型内部会自动计算损失

**chatglm3(2023)**

github：<https://github.com/THUDM/ChatGLM3>

2023年智谱AI和清华大学实验室联合发布chatglm3模型，有6B、6B-32K、6B-128K三个型号。词表长度：64798

tokenizer变化：add\_special\_tokens=True时会在前面加入64790,64792([gMASK],sop)。用户问题标志为64795和<|user|>，模型回答标志为64796和<|assistant|>

提示词注入：提示词中通常有开始符和结束符等，如果用户在问题中输入这些内容，可能会干扰模型的理解。如果用户输入一些规则，可能让模型回答违规内容

**baichuan2(2023)**

github：<https://github.com/baichuan-inc/Baichuan2>

2023年百川智能发布baichuan2模型，有7B、13B两个型号。词表长度：125696

tokenizer变化：'{system}'为系统提示词标志，'<reserved\_106>{input}'为用户输入标志，'<reserved\_107>{assistant}</s>'为模型回答标志，模型回答的结束标志为</s>

tokenizer=transformers.AutoTokenizer.from\_pretrained(model\_path,trust\_remote\_code=True,use\_fast=False)：文本预处理

model=transformers.AutoModelForCausalLM.from\_pretrained(model\_path,device\_map='auto',trust\_remote\_code=True,torch\_dtype=torch.float16)：加载模型。device\_map指定模型使用设备，'auto'为自动分配

model=model.eval()|.to(device)|.half/float()：模型状态设置

with torch.no\_grad():：模型验证设置

system=''：默认系统提示。用户输入的提示加在system后

template='{system}<reserved\_106>{input}<reserved\_107>'：提示模版

prompt=template.format(system=system,input=input\_)：在输入中加入提示模版。system为系统提示，input为输入问题

template\_history='{output\_add}<reserved\_106>{input\_add}<reserved\_107>'：多轮对话时追加的历史信息模版。output\_add为上一轮的回答，input\_add为下一轮的输入

prompt=prompt+template\_history.format(output\_add=output\_add,input\_add=input\_add)：多轮对话时要追加历史信息

input\_ids=tokenizer.encode(prompt,return\_tensors='pt')：模型输入

pred=model.generate(input\_ids=input\_ids,generation\_config=generation\_config)：使用模型预测

result=tokenizer.decode(pred[0])：解码。此时result内容为：输入+回答+结束符

**模型训练：**

模型输入长度通常为1024个词索引。input\_ids、attention\_mask、label长度一致。最终合成批量时：input\_ids在后面填充pad\_token\_id，attention\_mask在后面填充0，label在后面填充-100

prompt\_encode=tokenizer.encode(prompt,add\_special\_tokens=False)：提问数据

output\_encode=tokenizer.encode(output,add\_special\_tokens=False)：回答数据

input\_ids=torch.tensor(prompt\_encode+output\_encode+[eos\_token\_id],dtype=torch.int64)：模型输入

attention\_mask=torch.full\_like(input\_ids,1)：掩码

label=torch.full\_like(input\_ids,-100)：不需要的地方变为-100

label[len(prompt\_encode):]=input\_ids[len(prompt\_encode):]

index=torch.nonzero(input\_ids==195).squeeze(1)：对应<reserved\_106>

label[index]=eos\_token\_id：变为eos\_token\_id，采用这种做法的好处是template\_history中的{output\_add}后不需要加</s>，节省一个字符

input\_ids=input\_ids[:max\_length]

attention\_mask=attention\_mask[:max\_length]

label=label[:max\_length]

pred\_batch=model(input\_ids=input\_ids\_batch,attention\_mask=attention\_mask\_batch,labels=label\_batch)：模型推理

loss\_batch=pred\_batch.loss：当传入labels时模型内部会自动计算损失

**qwen(2023)**

github：<https://github.com/QwenLM/Qwen>

2023年阿里发布qwen模型，有1.8B、7B、14B、72B四个型号。词表长度：151851

tokenizer变化：'<|im\_start|>system\n{system}<|im\_end|>\n'为系统提示词标志，同理将system改为user为用户输入标志，改为assistant为模型回答标志，模型回答的结束标志为<|im\_end|>\n

generation\_config变化：eos\_token\_id=151643或151645，pad\_token\_id=151643

tokenizer=transformers.AutoTokenizer.from\_pretrained(model\_path,trust\_remote\_code=True,use\_fast=False)：文本预处理

model=transformers.AutoModelForCausalLM.from\_pretrained(model\_path,device\_map='auto',trust\_remote\_code=True,torch\_dtype=torch.float16)：加载模型。device\_map指定模型使用设备，'auto'为自动分配

model=model.eval()|.to(device)|.half/float()：模型状态设置

with torch.no\_grad():：模型验证设置

system='You are a helpful assistant.\n'：系统提示

template='<|im\_start|>system\n{system}<|im\_end|>\n<|im\_start|>user\n{input}<|im\_end|>\n<|im\_start|>assistant\n'：提示模版

prompt=template.format(system=system,input=input\_)：在输入中加入提示模版。system为系统提示，input为输入问题

template\_history='{output\_add}<|im\_end|>\n<|im\_start|>user\n{input}<|im\_end|>\n<|im\_start|>assistant\n'：多轮对话时追加的历史信息模版。output\_add为上一轮的回答，input\_add为下一轮的输入

prompt=prompt+template\_history.format(output\_add=output\_add,input\_add=input\_add)：多轮对话时要追加历史信息

input\_ids=tokenizer.encode(prompt,return\_tensors='pt')：模型输入

pred=model.generate(input\_ids=input\_ids,generation\_config=generation\_config)：使用模型预测

result=tokenizer.decode(pred[0])：解码。此时result内容为：输入+回答+结束符

**模型训练：**

模型输入长度通常为1024个词索引。input\_ids、attention\_mask、label长度一致。最终合成批量时：input\_ids在后面填充pad\_token\_id，attention\_mask在后面填充0，label在后面填充-100

prompt\_encode=tokenizer.encode(prompt,add\_special\_tokens=False)：提问数据

output\_encode=tokenizer.encode(output,add\_special\_tokens=False)：回答数据

input\_ids=torch.tensor(prompt\_encode+output\_encode+[151645]+[198],dtype=torch.int64)：模型输入

attention\_mask=torch.full\_like(input\_ids,1)：掩码

label=torch.full\_like(input\_ids,-100)：不需要的地方变为-100

label[len(prompt\_encode):]=input\_ids[len(prompt\_encode):]

index=torch.nonzero(input\_ids==im\_start\_id).squeeze(1)：对应<|im\_start|>

label[index]=im\_start\_id：变为im\_start\_id

index=torch.nonzero(input\_ids==im\_end\_id).squeeze(1)：对应<|im\_end|>

label[index]=im\_end\_id：变为im\_end\_id

index+=1：<|im\_end|>后的\n对应的地方

label[index]=198：变为198

input\_ids=input\_ids[:max\_length]

attention\_mask=attention\_mask[:max\_length]

label=label[:max\_length]

pred\_batch=model(input\_ids=input\_ids\_batch,attention\_mask=attention\_mask\_batch,labels=label\_batch)：模型推理

loss\_batch=pred\_batch.loss：当传入labels时模型内部会自动计算损失

**aiXcoder(2024)**

github：<https://github.com/aixcoder-plugin/aiXcoder-7B>

2024年硅心科技发布aiXcoder-7B模型，侧重于代码生成。词表长度：49152

**llama3(2024)**

github：<https://github.com/LlamaFamily/Llama-Chinese>

github：<https://github.com/ymcui/Chinese-LLaMA-Alpaca-3>

2024年Meta发布LLaMa3模型，有8B、70B两个型号。此后中文团队训练出中文模型版本。llama3的提示模版与llama2的不同。词表长度：128256

tokenizer变化：'<|start\_header\_id|>system<|end\_header\_id|>\n\n{system}<|eot\_id|>'为系统提示词标志，同理将system改为user为用户输入标志，'<|start\_header\_id|>system<|end\_header\_id|>\n\n{assistant}<|end\_of\_text|>'为模型回答标志，模型回答的结束标志为<|end\_of\_text|>

tokenizer=transformers.AutoTokenizer.from\_pretrained(model\_path,trust\_remote\_code=True,use\_fast=False)：文本预处理

model=transformers.AutoModelForCausalLM.from\_pretrained(model\_path,device\_map='auto',trust\_remote\_code=True,torch\_dtype=torch.float16)：加载模型。device\_map指定模型使用设备，'auto'为自动分配

model=model.eval()|.to(device)|.half/float()：模型状态设置

with torch.no\_grad():：模型验证设置

system='You are a helpful assistant. 你是一个乐于助人的助手。'：默认系统提示。用户输入的提示加在system后

template='<|start\_header\_id|>system<|end\_header\_id|>\n\n{system}<|eot\_id|><|start\_header\_id|>user<|end\_header\_id|>\n\n{input}<|eot\_id|><|start\_header\_id|>assistant<|end\_header\_id|>\n\n'：提示模版

prompt=template.format(system=system,input=input\_)：在输入中加入提示模版。system为系统提示，input为输入问题

input\_ids=tokenizer.encode(prompt,return\_tensors='pt')：模型输入

pred=model.generate(input\_ids=input\_ids,generation\_config=generation\_config)：使用模型预测

result=tokenizer.decode(pred[0])：解码。此时result内容为：输入+回答+结束符

**qwen2(2024)**

github：<https://github.com/QwenLM/Qwen2>

2024年阿里发布qwen2模型，有0.5B、1.5B、7B、14B、72B五个型号。词表长度：151646

除了词表长度151646与qwen的151851不一致外，其余的特殊字符、提示词模版、训练方法等完全一致。qwen2相比qwen在网络结构方面将解码模块的堆叠数减少，增加了mlp网络的中间隐藏层数。72B能处理128k长度输入，安全测试得分与GPT4差不多

tokenizer变化：'<|im\_start|>system\n{system}<|im\_end|>\n'为系统提示词标志，同理将system改为user为用户输入标志，改为assistant为模型回答标志，模型回答的结束标志为<|im\_end|>\n。填充符为<|endoftext|>对应151643，<|im\_start|>对应151644，<|im\_end|>对应151645

generation\_config变化：eos\_token\_id=[151645,151643]，pad\_token\_id=151643

**qwen3(2025)**

github：<https://github.com/QwenLM/Qwen3>

2025年阿里发布qwen3模型，有0.6B、1.7B、4B、8B、14B、32B、30B-A3B、235B-A22B共8个型号。词表长度：151936 qwen3模型加入了思维链，提高了模型的推理能力，通过enable\_thinking参数控制，训练时如果要调整思维链，需要加入思维链的标签

提示词模版：'<|im\_start|>system\n{system}<|im\_end|>\n<|im\_start|>user\n{user}<|im\_end|>\n<|im\_start|>assistant\n<think>\n{think}\n\n{assistant}<|im\_end|>\n'

不启用思维链：模型输入加上'<think>\n\n<think>\n\n'

多轮对话：拼接历史中的user和assistant对话模版

bos\_token\_id：<|im\_start|>，151644

eos\_token\_id：<|im\_end|>，151645

pad\_token\_id：<|endoftext|>，151643

**import peft**

官方文档：<https://huggingface.co/docs/peft/index>

大模型微调库。只需要微调模型的少量参数就能取得很好效果

lora：低阶适应微调技术。冻结模型原来的权重参数，加入额外的可训练参数，在训练时不需要计算原模型的梯度，缩短训练时间。具体是在某些结构旁加入一个并行的分支网络，最终网络输出结果为：原网络输出+分支网络输出

基本原理：若原模型某层结构为y=w\*x，权重w维度为a\*b，输入x维度为b\*c，输出y维度为a\*c。加入lora后y=w\*x+A\*B\*x，A维度为a\*r，B维度为r\*c。原模型的b非常大，所以参数很多，而lora的r相比b很小，因此参数很少。除此之外还有lora\_alpha、q、v等调节参数

peft模型、lora模型、分支模型是同一个意思

peft\_config=peft.LoraConfig(r,lora\_alpha,lora\_dropout,inference\_mode=False,task\_type=peft.TaskType.CAUSAL\_LM,target\_modules)：分支网络配置，不同模型的配置差别很大。r为基本原理中的r，控制分支网络的大小，太大容易过拟合；lora\_alpha控制网络对模型的影响程度，越大微调影响越小，设置为r/2比较温和；lora\_droput防止过拟合；inference\_mode=False；task\_type=peft.TaskType.CAUSAL\_LM；target\_modules为要调节的结构名称

llama2默认参数：(r=64,lora\_alpha=128,lora\_dropout=0.05,inference\_mode=False,task\_type=peft.TaskType.CAUSAL\_LM,target\_modules=['q\_proj','v\_proj','k\_proj','o\_proj','gate\_proj','down\_proj','up\_proj'])

baichuan2默认参数：(r=1,lora\_alpha=32,lora\_dropout=0.1,inference\_mode=False,task\_type=peft.TaskType.CAUSAL\_LM,target\_modules=['W\_pack'])

qwen默认参数：(r=64,lora\_alpha=16,lora\_dropout=0.05,inference\_mode=False,task\_type=peft.TaskType.CAUSAL\_LM,target\_modules=['c\_attn','c\_proj','w1','w2'])

model=peft.get\_peft\_model(model,peft\_config).float()：创建新的分支网络加入到模型中。分支网络的初始权重为0，因此刚加入时模型的输出不会改变。连续使用函数并不会加入多个分支模型，只会加一次。使用.float()以确保float32精度

model.print\_trainable\_parameters()：输出当前可训练参数和总参数对比，无返回值

torch.save+torch.load：保存和加载模型。采用这种方式可以加载并继续训练模型，但占内存比较大

model.save\_pretrained(save\_peft\_dir)：保存分支模型

model=peft.PeftModel.from\_pretrained(model,perf\_model\_path)：加载原模型和分支模型并组合(并不是真正的合并)。此时不可训练

model=peft.AutoPeftModelForCausalLM.from\_pretrained(perf\_model\_path,trust\_remote\_code=True)：加载原模型和分支模型并组合(需要原模型位置不变)。此时不可训练。如果peft=0.8，加载模型同时会尝试加载tokenizer，但peft内部未相应设置trust\_remote\_code=True，因此会导致报错

model=model.merge\_and\_unload()：合并基础模型和peft模型。此时使用model.save\_pretrained()将保存合并后的完整模型

**import diffusers**

Hugging Face推出的用于推理和训练diffusion扩散模型的库

**StableDiffusion(2022)**

github：<https://github.com/Stability-AI/StableDiffusion>

Stable Diffusion主要由clip的文本编码+unet去噪生成+vae变分自编码3个模型组成，实际应用中还会加入图片编码模型、控制模型、各类风格lora模型等，是一类模型的集成。在Hugging Face的开源模型中，通常会提供合在一起的完整模型和拆分开的各模块模型，只是加载方式不同。可以在diffusers库中灵活的加载和替换各模块

扩散(diffusion)：图片的加噪或去噪

文本编码模型(500MB以内)：输入的文本先经过tokenizer编码，再输入到文本编码模型中得到特征向量，然后将特征向量输入unet模型

unet去噪生成模型(5G以内)：图片生成的核心结构，在unet去噪生成过程中，需要很多轮次的迭代，将上1轮的结果输入到下一轮中，迭代次数越多细节越丰富

scheduler采样器：scheduler会对上1轮unet的结果进行处理，再作为下一轮unet的输入，不同的scheduler配置会影响模型生成的结果和时长

vae变分自编码模型(500MB以内)：对最后一轮unet生成的结果进行处理和放大，得到最终的图片

**通用部分**

传入模型的图片可以使用PIL.Image、[PIL.Image]、torch.tensor格式

model.scheduler=diffusers.EulerAncestralDiscreteScheduler.from\_config(model.scheduler.config)：自定义采样器

model.enable\_model\_cpu\_offload()：智能GPU加载。完整模型由很多模块串联组成，智能GPU加载只将需要的部分放到GPU上、推理过的部分放回CPU中，增加CPU和GPU之间的少量转移时间，显著节约显存。注意使用后就不要在用model.to('cuda')

model.enable\_xformers\_memory\_efficient\_attention()：加速注意力层计算速度，需要安装xformers库

**文生图**

model=diffusers.StableDiffusionPipeline.from\_pretrained(模型名称/文件夹位置,torch\_dtype)：加载模型。文件夹结构需要与Hugging Face网站中的保持一致。当本地不存在模型时会自动从Hugging Face网站中下载

model=diffusers.StableDiffusionPipeline.from\_single\_file(safetensors文件,torch\_dtype)：加载模型。这种方式只需要一个完整的safetensors文件

image=model(prompt=None,negative\_prompt=None,ip\_adapter\_image=None,height,width,num\_inference\_steps=50,guidance\_scale=7.5).images[0]：输入文本生成图像。prompt为真相提示词(字符串)；negative\_prompt为负向提示词；ip\_adapter\_image可以传入提示图像(PIL.Image的RGB)，但需要加载额外模型，参考后面的IP-Adapter；height和width指定生成图片的大小，一般为512或1024；num\_inference\_steps为unet网络重复生成次数，次数越多细节越丰富但时间越长；guidance\_scale略微影响图像的质量，大于7.5时质量更好但多样性降低，反之同理

**文+图生图**

github：<https://github.com/tencent-ailab/IP-Adapter>

IP-Adapter(2023)是一个主要针对人像的图片编码插件(100MB以内)，需要配合Stable Diffusion模型、clip图片编码模型(2.5G)使用，可以将提示图片信息融入到文生图模型中，实现文+图生图的功能

model=文生图模型

model.image\_encoder=transformers.CLIPVisionModelWithProjection.from\_pretrained(文件夹位置,torch\_dtype)：加载clip图片编码模型

model.load\_ip\_adapter('IP-Adapter/models',subfolder='models',weight\_name='ip-adapter-plus\_sd15.safetensors')：加载图片编码插件模型。IP-Adapter为下载到本地的文件夹，文件夹结构需要与Hugging Face网站中的保持一致；subfolder为模型所在子目录，weight\_name为具体的模型，有很多版本

**文+图+控制生图**

生成控制模型，可以控制图片的生成区域，从而实现风格替换或细节完善等效果。生成控制模型是插入在unet模型中的神经网络，可以多个生成控制模型一起使用

边缘控制模型：<https://huggingface.co/lllyasviel/sd-controlnet-canny>

controlnet=diffusers.ControlNetModel.from\_pretrained(模型名称/文件夹位置,torch\_dtype)：加载控制模型。

model=diffusers.StableDiffusionControlNetPipeline.from\_pretrained(模型名称/文件夹位置,controlnet=controlnet,torch\_dtype)：加载带控制器的Stable Diffusion模型，替代StableDiffusionPipeline。controlnet可以传入包含多个控制模型的列表

image=model(...,image=None,controlnet\_conditioning\_scale=1.0)：使用模型。相比StableDiffusionPipeline会多出几个参数：image为传入的控制图片，sd-controlnet-canny中为经过cv2.Canny得到的轮廓图片，多个控制模型时传入列表；controlnet\_conditioning\_scale为控制模型的叠加权重，多个控制模型时传入列表

**图生图**

可以生成一张图片的相关图片，在3D建模中可以用于生成物体的多视角图

model=diffusers.StableDiffusionImageVariationPipeline(模型名称/文件夹位置)：加载图生图模型

image\_list=model(image,width,height,num\_inference\_steps=50,guidance\_scale=7.5,num\_images\_per\_prompt=1).images：使用模型。num\_images\_per\_prompt为生成图片数

**import paddle**

安装：<https://www.paddlepaddle.org.cn/install/quick?docurl=/documentation/docs/zh/install/pip/linux-pip.html>

github：<https://github.com/PaddlePaddle/Paddle>

API:https://www.paddlepaddle.org.cn/documentation/docs/zh/api/index\_cn.html

paddle模型保存分为静态图和动态图。静态图先定义好网络再计算，速度快；动态图边计算边构建网络，计算速度较慢但便于调试。一般模型导出和转换时才用静态图

动态图模型文件：.pdmodel、.pdparams

静态图模型文件：.pdmodel、.pdiparams、.pdiparams.info

**基本操作：大部分和numpy一致**

paddle.to\_tensor(A)：转换为张量

**paddle.nn：搭建神经网络(和pytorch大体一致)**

对于网络需要输入形状时，输入形状后系统会把batch理解为批量，然后并行计算。网络搭建时最好不用if判断选择网络结构，后续转为onnx模型时容易出错

**方式1：**model=paddle.nn.Sequential()

//model.add\_sublayer('name',layer1)：添加层。但不能构建复杂结构

或者：model=paddle.nn.Sequential(layer1,layer2...)：名字自动生成

paddle.nn.Sequential(\*(layer1 for i in range(n)))：构建相同层

**方式2：**class model(paddle.nn.Layer) // def \_\_init\_\_(self,...) // self.layer1=layer1...：初始化A=model(\_\_init\_\_中参数)时会搭建好模型结构，因此\_\_init\_\_中要把所有用到的层按顺序传入self(paddle.nn.Layer为搭建模型的继承类库函数)

//def forward(self,x) // return x：调用A(x)会直接执行forward函数，此时才会确定模型内部的连接方式。对于复杂的网络定义多个类模块嵌套，类中可以使用Sequential函数。forward中用if可能导致onnx导出失败

**基础网络：**

paddle.nn.Sigmoid/Tanh/ReLU/LeakyReLu/...()：添加激活函数

paddle.nn.Flatten()：将多维数据变成一维。输入(batch,...)，输出(batch,n)

paddle.nn.Linear(in\_features,out\_features)：全连接层。输入(batch,in\_features)，输出(batch,out\_features)。多变量预测时可(batch,n,in\_features)，输出(batch,n,out\_features)，但此时各dim之间是没有关系的

paddle.nn.Dropout(0.2)：丢弃层，输入/输出(batch,n)

**一维网络：**

paddle.nn.Conv1D(in\_,out\_,kernel\_size,stride=1,padding=0,dilation=1)：卷积层。输入(batch,dim,n)，in\_/out\_为输入/输出维度，对应dim；stride和padding对应n，stride为卷积步长(卷积后四周会缩短stride//2)；padding=a在序列首尾填充a个0；dilation为卷积时卷积核各点的距离(形成空洞卷积)

**二维网络：**

paddle.nn.Conv2D(in\_,out\_,kernel\_size,stride=1,padding=0,dilation=1)：卷积层。输入(batch,dim,m,n)，in\_/out\_为输入/输出维度，对应dim；stride和padding对应m、n，stride为卷积步长(卷积后四周会缩短stride//2)；padding=a在图片周围填充a个0；dilation为卷积时卷积核各点的距离(形成空洞卷积)

paddle.nn.AdaptiveAvgPool1D(output\_size)：平均池化。output\_size=1时，输入(batch,dim,n)，输出(batch,dim,1)

**数据处理：**

使用Dataset-Dataloader方法，训练时一边训练一边取数据、预处理、合成批量，CPU数据处理和GPU模型推理并行，节约数据处理时间

paddle.io.Dataset：类似torch.utils.data.Dataset

paddle.io.DataLoader：类似torch.utils.data.DataLoader

**模型保存与加载：**

**推理框架设置：**

config=paddle.inference.Config('model.pdmodel','model.pdiparams')：创建配置器

config.enable\_use\_gpu(200,0)：初始化200M显存，使用的gpu\_id为0

config.switch\_ir\_optim(True)：使用GPU时开启IR优化

config.disable\_gpu()：使用cpu

config.set\_cpu\_math\_library\_num\_threads(a)：设置cpu线程数为a

config.disable\_glog\_info()：推理时不会输出log信息

config.enable\_memory\_optim()：可以分享内存

config.switch\_use\_feed\_fetch\_ops(False)

model=paddle.inference.create\_predictor(config)：创建推理器

input\_name=model.get\_input\_names()：获取要输入的信息，返回[...]。一般列表中有'image'，不同的模型会有不同的信息

input\_tensor=model.get\_input\_handle('image')：input\_tensor为输入接口。其他需要输入的东西同理

**推理框架使用：**

input\_tensor.copy\_from\_cpu(image)：输入图片(RGB)，形状类似np.array(batch,3,640,640)，数据类型一般为np.float32。其他需要输入的东西同理

model.run()：模型执行推理

output\_name=model.get\_output\_names()：获取输出的信息

output\_tensor=model.get\_output\_handle(output\_name[0])：获取输出结果

pred=output\_tensor.copy\_to\_cpu()：转换为numpy数组

**pphuman(2022)**

github：<https://github.com/PaddlePaddle/PaddleDetection/blob/release/2.6/deploy/pipeline/docs/tutorials/pphuman_mot.md>

行人检测模型。下载模型后使用推理框架即可运行。模型有'image'、'scale\_factor'两个参数。image(RGB)为直接resize为(640,640,3)再转为np.array(batch,3,640,640)，scale\_factor为(h,w)的缩放的比例(零点几)，形状为np.array([[h,w]])。结果为所有的box(np.float32)

input\_tensor=model.get\_input\_handle('image')

input\_tensor=model.get\_input\_handle('scale\_factor')

input\_tensor.copy\_from\_cpu(image)

input\_tensor.copy\_from\_cpu(scale\_factor)

**reid(2022)**

github：<https://github.com/PaddlePaddle/PaddleDetection/blob/release/2.6/deploy/pipeline/docs/tutorials/pphuman_mtmct.md#使用方法>

行人重识别模型。下载模型后使用推理框架即可运行。模型有'x'一个参数，x代表裁剪后的图片(RGB),直接resize为(256,128,3)，再除以255、减均值和除以方差(mean=[0.485, 0.456, 0.406]，std=[0.229, 0.224, 0.225])，然后转为np.array(batch,3,256,128)。结果feature(np.float32)形状为(1,256)。

input\_tensor=model.get\_input\_handle('x')

input\_tensor.copy\_from\_cpu(image)

feature=feature/np.linalg.norm(feature, axis=1)：需要归一化再进行相似度匹配

**import paddle2onnx**

paddle2onnx --model\_dir 模型文件夹 --model\_filename .pdmodel --params\_filename .pdiparams --save\_file best.onnx --opset\_version 15 --export\_fp16\_model True：将paddle静态图模型转换为onnx模型

**import paddleslim：飞桨模型量化库**

**import paddlenlp**

github：<https://github.com/PaddlePaddle/PaddleNLP>

API：<https://paddlenlp.readthedocs.io/zh/latest/>

使用paddle模型，需要先安装paddlepaddle库

(pip install fast-tokenizer-python：安装tokenizer加速处理库)

tokenizer=paddlenlp.transformers.AutoTokenizer.from\_pretrained(官方模型名称/本地文件夹位置,use\_fast=True)：加载模型的预处理方式。use\_fast=True时需要下载fast-tokenizer-python库才会生效，可以加快文本处理速度

tokenizer.add\_tokens([词语])：添加编码组合。比如默认每个字编码单独编码，加入词后会将整个词一起编码

dict=tokenizer([句子1,句子2...],max\_length,padding=False,truncation=False)：对数据进行预处理，返回字典{'input\_ids':[...],'token\_type\_ids':[...]}，input\_ids中为字符串编码后的词索引向量，会在前后加上开始、结束编码(1和2)；token\_type\_ids为等长的列表，表示当前词是第几句话的，对于同一句话都为0。padding=True在后面填充0直到与最长序列相等，padding='max\_length'填充为max\_length；truncation=True截断为max\_length长度

input\_ids=paddle.to\_tensor(dict['input\_ids'])：转为张量

tokenizer.decode(input\_ids[i])：解码为句子(一次只能解码一句)，会带上[CLS]、[SEP]标识

tokenizer.add\_tokens([多个字符,...])：正常情况下会将一个字符编码为一个数字，使用后会将添加的多个字符一起编码

model=paddlenlp.transformers.AutoModel.from\_pretrained(官方模型名称/本地模型文件夹位置)：加载模型

model=paddlenlp.transformers.AutoModelForSequenceClassification.from\_pretrained(官方模型名称/本地模型文件夹位置,num\_classes)：加载文本分类模型。num\_classes为句子的类别数(只有加载未训练的官方模型才有用)

model=paddlenlp.transformers.AutoModelForTokenClassification.from\_pretrained(官方模型名称/本地模型文件夹位置)：加载序列标注模型。num\_classes为句子中每个字符的类别数

model=paddlenlp.transformers.AutoModelForQuestionAnswering.from\_pretrained(官方模型名称/本地模型文件夹位置)：加载问答模型

pred=model(input\_ids=input\_ids,token\_type\_ids=token\_type\_ids)：使用模型

**模型保存与加载：**

tokenizer.save\_pretrained(保存文件夹位置)：保存数据处理方式，文件夹下会有special\_tokens\_map.json、tokenizer\_config.json、vocab.txt文件

tokenizer=paddlenlp.transformers.AutoTokenizer.from\_pretrained(本地文件夹位置)：加载模型对应的tokenizer

model.save\_pretrained(保存文件夹位置)：保存模型参数，文件夹下会有config.json、model\_state.pdparams文件。分布式用model=model.\_layers取出模型

model=AutoModelForSequenceClassification.from\_pretrained(本地文件夹位置)：加载保存的分类模型，如果是其他模型AutoModelForSequenceClassification要换成对应的

**Taskflow(2021)**

文本纠错模型

model=paddlenlp.Taskflow('text\_correction')：加载模型。会下载模型(820M)到本地.paddlenlp中

pred=model('句子')：使用模型对句子纠错。pred形状为：[{'score':原句,'target':修改后句子,'errors':[{'position':错字下标,'correction':{'错字':改正后的字}}]}]

**ernie-3.0(2022)**

github：<https://github.com/PaddlePaddle/PaddleNLP/tree/develop/model_zoo>

百度提出的超大语言模型，在多个数据集上均优于以往模型

paddlenlp.transformers.AutoModel.from\_pretrained('ernie-3.0-medium-zh')：加载ernie模型

**import fastdeploy**

飞桨模型部署库

github：<https://github.com/PaddlePaddle/FastDeploy>

函数：<https://baidu-paddle.github.io/fastdeploy-api/python/html/>

(pip install fastdeploy-python/fastdeploy-gpu-python -f <https://www.paddlepaddle.org.cn/whl/fastdeploy.html)>

(cpu的只支持CPU推理，gpu的支持CPU和GPU，tensorrt只能在GPU上)

支持padding和pytorch模型，需要先安装cuda和cudnn，不需要安装tensorrt

飞桨框架的模型分为模型结构和参数两部分

runtime\_option=fastdeploy.RuntimeOption()：后端推理配置

runtime\_option.use\_cpu()/.use\_gpu()：使用CPU/GPU

runtime\_option.use\_ort\_backend()：Paddle/ONNX模型用ort框架推理

runtime\_option.use\_trt\_backend()：Paddle/ONNX用trt框架GPU推理

runtime\_option.set\_trt\_input\_shape('输入名称',形状如:[1, 3, 640, 640])：设置输入形状，有的模型还需要设置动态形状范围

runtime\_option.enable\_trt\_fp16()：trt推理时使用float16(默认32)

runtime\_option.set\_trt\_cache\_file(缓存位置.trt)：使用trt推理时会先转换为trt模型，如果存在缓存位置则直接加载，不存在则下次转换后会保存

**yolov7(2022)**

目标检测模型

runtime\_option.set\_trt\_input\_shape("images", [1, 3, 640, 640])

fastdeploy.vision.detection.YOLOv7(model,params\_file=None,runtime\_option=None,model\_format=fastdeploy.ModelFormat.ONNX)：构建YOLOV7推理模型。model可为.onnx/.pdmodel；params\_file当模型为onnx时无需设置；runtime\_option为后端推理配置；model\_format为模型格式，paddle为fastdeploy.ModelFormat.PADDLE

model.predict(image,conf\_threshold=0.25,nms\_iou\_threshold=0.5)：模型预测。image为WHC或BGR格式；conf\_threshold、nms\_iou\_threshold为阈值，如果导出为onnx模型时已经设置，因此只有此处阈值大于之前的才生效

**ppocrv3(2022)**

图片文字识别模型，由目标检测、文字方向检测、文字识别三个模型组成

github：<https://github.com/PaddlePaddle/FastDeploy/tree/develop/examples/vision/ocr/PP-OCR/cpu-gpu/python>

det\_option.set\_trt\_input\_shape("x",[1,3,64,64],[1,3,640,640],[1,3,960,960])：推荐输入范围设置

det\_model=fastdeploy.vision.ocr.DBDetector(pdmodel,pdiparams,runtime\_option=det\_option)：构建目标检测模型

cls\_option.set\_trt\_input\_shape("x",[1,3,48,10],[cls\_batch\_size,3,48,320],[cls\_batch\_size,3,48,1024])：推荐输入范围设置

cls\_model=fastdeploy.vision.ocr.Classifier(pdmodel,pdiparams,runtime\_option=cls\_option)：构建文字方向检测模型

rec\_option.set\_trt\_input\_shape("x",[1,3,48,10],[rec\_batch\_size,3,48,320],[rec\_batch\_size,3,48,2304])：推荐输入范围设置

rec\_model=fastdeploy.vision.ocr.Recognizer(pdmodel,pdiparams,rec\_label,runtime\_option=rec\_option)：构建文字识别模型。rec\_label为文字的标签，根据需要可以修改

model=fastdeploy.vision.ocr.PPOCRv3(det\_model=det\_model,cls\_model=cls\_model,rec\_model=rec\_model)：将三个模型串联组合起来

pred=model.predict(image)：使用模型。检测结果按照边框Y坐标轴从上到下排序，因此在一些证件识别时，可以先用霍夫转换对正，再略微顺时针旋转一点

text\_list=pred.text：得到所有文本，结果可能为标点符号、空字符、特殊字符，根据rec\_label中的标签

**import modelscope**

阿里魔搭社区，包含许多开箱即用的模型、简洁的API接口

网址：<https://modelscope.cn/models?page=1>

ocr图片文字检测:

<https://www.modelscope.cn/models/iic/cv_convnextTiny_ocr-recognition-general_damo>

识别效果一般，难以识别多行文字，速度很快，onnx模型在cpu上每次约0.04s

**import openai**

登录openai和使用API时要全程使用VPN，选新加坡，可用google账号登录

网址：https://platform.openai.com/docs/models

提示：https://platform.openai.com/examples

剩余额度：https://platform.openai.com/usage

限制规则：https://platform.openai.com/account/limits

计费规则：https://openai.com/pricing

2023年3月1日起，发送的API数据会被官方保留30天，但不使用数据改进模型。计费按照每1000个词索引计费，约等于750个单词或500个汉字。temperature越小则回答越简短。假设高质量的中文回答长度在700-900，假设输入价格为0.001美元/千词索引，输出价格0.002美元/千词索引，1美元可以单轮对话约600次

API接口与网页版的GPT有区别，通常网页版的效果更好、回答更长。使用API时要适当调整系统提示，比如[你是一个乐于助人的知识专家，你的回答要尽量丰富和全面，回答中的小标题要带\*\*号]

对话模型的输出中小标题可能会带上“\*\*”号，这是markdown格式的加粗，因为训练时的数据中就带有\*\*号。通常回答中的换行处为\n\n，偶尔会出现\n \n或\n等，除此之外多个空格、小标题格式等都需要注意，因此模型的输出还需进行一定的处理

API=openai.OpenAI(api\_key='API密钥')：初始化API接口

response=API.chat.completions.create(model,max\_tokens,temperature=1,messages)：使用API，以post方式发送信息。model为使用的模型(详见网址)；max\_tokens为最大生成词数，超过部分会直接截断，不建议设置；temperature较高时输出更加随机，较低时输出更加稳定且简短；messages为发送的信息[{'role':'system','content':'提示词'}, {'role':'user','content':'问题1'}, {'role':'assistant','content':'上轮回答'}, {'role':'user','content':'问题2'}...]

result=response.choices[0].message.content：得到结果

**import rembg**

去除图片背景的库

image=PIL.Image.open(image\_path)：读取图片

image=rembg.remove(image,session=None)：去除图片背景。session可以加载自定义模型。image可以为RGB或RGBA格式，返回结果为RGBA格式

**import prophet**

github：<https://github.com/facebook/prophet>

facebook基于加法模型的时间序列预测框架，将序列分为趋势、周期、季节、随机值，适合具有季节性影响和多个季节历史数据的时间序列

model=prophet.Prophet()：创建模型

model.fit(df)：用模型学习数据。df中只能有两列，时间列ds和数据y

model.changepoint\_prior\_scale=a：设置拟合参数，越大拟合性越强，默认0.05

future=model.make\_future\_dataframe(periods)：设置预测长度，periods为长度

df\_pred=model.predict(future)：预测未来数据。df\_pred中有时间列ds、预测值yhat、预测下限yhat\_lower、预测上限yhat\_upper、趋势trend、趋势下限trend\_lower、趋势上限trend\_upper、周趋势weekly、年趋势yearly等

figure=model.plot\_components(df\_pred)：画出trend、weekly、yearly的图，可以使用figure.savefig(path)保存或figure.show()显示

**import llama\_cpp**

github：<https://github.com/ggml-org/llama.cpp>

2024年使用c++实现的替代pytorch进行高效推理的库，提供将模型量化为float16、int8、int4的方法，针对不同平台进行了优化。需要先将模型转为gguf格式，gguf是一种包含了模型本身和所有配置文件的二进制格式。llama.cpp需要c++环境运行，安装时用cmake构建，安装后可以从python中的llama\_cpp\_python库调用

安装：git clone -> cd llama.cpp -> cmake -B build -DLLAMA\_CURL=OFF -> cmake --build build --config Release。安装完后，会生成main文件

模型转换为gguf：python llama.cpp/convert\_hf\_to\_gguf.py --model qwen3\_0.6b --outfile qwen3\_0.6b.gguf --outtype f16

llama.cpp/build/bin/Release/llama-cli.exe -m 模型.gguf：在终端运行模型。可以多轮对话，会自动处理历史输入

代码调用

model=llama\_cpp.Llama(model\_path,n\_gpu\_layers=0,n\_ctx=512)：加载模型。n\_gpu\_layers=0使用cpu，大于1将前n\_gpu\_layers层放到GPU上，-1时全部放到GPU上；n\_ctx为上下文窗口(输入和输出)最大token长度

pred=model.create\_completion(text,max\_tokens,temperature,stream)：模型推理。启用stream后，可以使用for循环不断获取流式输出

**import ollama**

github：<https://github.com/ollama/ollama>

下载：<https://ollama.com/download>

大模型部署程序，相当于把llama.cpp和flask进行了集成，使用gguf模型推理

windows开机启动：任务管理器->ollama。默认开机启动

ollama serve：启动ollama服务，不会加载模型，代码中调用模型时才会加载，调用不同模型时会替换加载。调用地址为http://localhost:11434/api/generate。

ollama create 模型名 -f 模型名.mf：配置mf文件并导入ollama中

ollama list：查看所有加载的模型

ollama run 模型名：在终端运行模型。可以多轮对话，会自动处理历史输入

mf配置文件

mf配置文件中不能加注释

FROM 模型绝对路径：模型位置

PARAMETER num\_ctx 4096：上下文窗口(输入和输出)最大token长度

PARAMETER stop "<|im\_end|>"：模型结束符

PARAMETER stop "<|endoftext|>" ：模型填充符或停止符

PARAMETER temperature 0.6：随机生成温度

TEMPLATE """

<|im\_start|>system

{{ .System }}<|im\_end|>

<|im\_start|>user

{{ .Prompt }}<|im\_end|>

<|im\_start|>assistant

"""：输入提示词模版

代码调用

response=ollama.generate(model,prompt,options)：模型推理。model为ollama中模型名；prompt为用户输入；options为控制参数字典{'temperature':...}

response=ollama.generate(model,messages,options)：模型推理。messages格式为{'role':'user','content':...,'role':'assistant','content':...}。相比generate可以更方便的传入历史数据

**import llama\_index**

官方文档：<https://docs.llamaindex.ai/en/stable/>

官方文档：https://docs.llamaindex.ai/en/stable/

llama\_index是2023年推出的数据检索库，专注于高性能的检索和rag构建，包含文本编码模型使用(需额外下载)、对数据库的高效检索和管理，注意它与llama2模型没有关系。llama\_index是一个大的框架，里面集成了很多其他的库，会略为杂乱当知识库庞大的时候，暴力搜索的性能会下降，llama\_index有多种优化措施。

FAISS(Facebook AI Similarity Search)：Facebook开源的向量相似性搜索库，支持近似最近邻搜索(ANN)，支持多种索引类型：IVF、HNSW、PQ

HNSW：基于图结构的ANN算法o(log n)，通过多层导航图加速搜索，需要额外内存占用，可以直接构筑索引，适合动态增删数据。适用于中等大小的知识库

IVF：先对特征使用聚类(如k-means)，搜索时只查询聚类子集。有两个变种：IVFFlat存储原始特征；IVFPQ乘积量化压缩特征，会降低精度。适用于庞大的知识库

元数据过滤：通过关键词查找需要的集合，只在需要的集合中查找数据

llama\_index数据结构为：[document文本块集合(逻辑管理)] --> [node文本块(实际单位)] --> [编码后以node为单位存储的特征数据库(又叫索引index)]。引入document集合的作用是方便管理，同文档中的文本块可以共用属性，可用于元数据过滤

准备阶段：[加载数据(划分文档、链接其他数据库)] --> [加载编码模型] --> [将文本块编码为特征] --> [存储特征]

使用阶段：[加载编码模型] --> [编码用户问题] --> [搜索向量库] --> [将文本块与用户问题合并] -- > [输入大语言模型]

0.10版本：数据库的最小存储单位是node，但最小操作单位是document。不建议直接对node进行增加和删除，而是对document进行操作

**from llama\_index.core import SimpleDirectoryReader**

document\_list=SimpleDirectoryReader(input\_dir).load\_data()：加载数据。input\_dir为txt文件目录

**from llama\_index.core import Document**

document=Document(text,id\_)：加载数据为document文本

document.id\_：更改id

document.metadata：字典metadata中包含更多详细信息

**from llama\_index.core.schema import TextNode**

node=TextNode(text,id\_)：加载数据为node文本块。没有document归属

node.id\_：更改id

node.metadata：字典metadata为空

**from llama\_index.core.node\_parser import SentenceSplitter**

split=SentenceSplitter(separator,chunk\_size,chunk\_overlap)：文本分割器

node\_list=split.get\_nodes\_from\_documents(document\_list)：划分文本块(node对象)。node会继承它原属于的document的metadata，同时会给出id\_和ref\_doc\_id，ref\_doc\_id的值为node归属的document的id\_

**from llama\_index.embeddings.huggingface import HuggingFaceEmbedding**

(pip install llama-index-llms-huggingface llama-index-embeddings-instructor)

model=HuggingFaceEmbedding(model\_name,device)：加载文本块编码模型。text2vec-base-chinese模型下载：<https://huggingface.co/shibing624/text2vec-base-chinese>

feature=model.get\_text\_embedding(text)：使用模型将文本编码为特征向量

**from llama\_index.embeddings.huggingface\_optimum import OptimumEmbedding**

(pip install transformers optimum)

(pip install llama-index-embeddings-huggingface-optimum)

OptimumEmbedding.create\_and\_save\_optimum\_model(model\_name\_or\_path,output\_path)：转换为onnx模型

model=OptimumEmbedding(folder\_name,device)：加载onnx模型

**from llama\_index.core import VectorStoreIndex**

feature\_database.docstore.docs中可以看到所有的node文本块。node中的ref\_doc\_id是它的document归属。feature\_database.ref\_doc\_info中可以看到所有document集合

feature\_database=VectorStoreIndex.from\_documents(document\_list,embed\_model=model)：从document\_list制作特征数据库

feature\_database=VectorStoreIndex(node\_list,embed\_model=model)：从node\_list制作特征数据库。注意node可能没有document归属

feature\_database.insert(document)：添加新的文本块集合。document的id\_作为ref\_doc\_id，document中node的id\_随机

feature\_database.insert\_nodes(new\_node\_list)：添加新的文本块。如果node所属document集合和node的id\_重复时不会添加或覆盖

feature\_database.delete\_ref\_doc(ref\_doc\_id)：删除文本块集合

**from llama\_index.core import StorageContext, load\_index\_from\_storage**

feature\_database.storage\_context.persist(persist\_dir)：保存数据库

storage=StorageContext.from\_defaults(persist\_dir)：加载数据库为storage

feature\_database=load\_index\_from\_storage(storage,embed\_model=model)：将storage转化为数据库

**from llama\_index.core.retrievers import VectorIndexRetriever**

retriever=feature\_database.as\_retriever()：配置搜索器。返回最相似的2个

result\_list=retriever.retrieve(input\_text)：搜索知识库。input\_text为要搜索的内容。result\_list[0].score为相似度，result\_list[0].text为内容

**import langchain**

官方文档：<https://js.langchain.com.cn/docs/>

langchain是2022年10月开源的一个项目，后来成为一家初创公司。langchain是一个大的框架，里面集成了很多其他的库。可以调用ollama、llama\_index等库

langchain用于智能助手的构建，包括多个模型、知识库、用户历史等整个系统的管理

**《模型部署库》**

**import netron**

netron可以看onnx模型结构，netron有还有专门的软件，可以直接打开.onnx文件查看网络结构

netron.start(onnx)：在跳出的网页中查看onnx模型的网络结构

**import onnx**

onnx是模型加速推理库，将tensorflow/pytorch模型转换为onnx可以与代码分离(pt模型加载需要代码)、减少大小、提高速度。后续还可以转为tensorrt。onnx模型可以用netron软件查看。onnx模型转出后输入输出的形状就固定了，只能将torch代码、基础结构(如列表)转换为onnx，数据处理等通常要分离，尽量不要在模型forward中用if判断，容易导致onnx模型运行出错。转出的onnx模型数据类型和pt模型一致，建议都为float16(cpu运行即使不支持也不会报错)

torch.onnx.export(model\_pt,输入,保存位置,opset\_version,input\_names,output\_names,dynamic\_axes)：将pytorch模型转为onnx。输入如torch.randn(1,3,640,640).to(device)，用于确定输入形状和执行一次推理以确定模型连接方式；opset\_version为opset版本，尽量用高版本，可不填用默认；input\_names为输入的名称如['input']；output\_names为输出的名称如['output']，目标检测多个输出如['num\_dets','det\_boxes','det\_scores','det\_classes']；dynamic\_axes={'input':{0:'batch\_size'},'output':{0:'batch\_size'}}指定输入输出是否为动态，0为动态、1为固定，固定的参数在初始化onnx模型后无法改变

**import onnxruntime**

(pip install onnxruntime：安装cpu版本)

(pip install onnxruntime-gpu：安装gpu版本，cpu和gpu都可以用)

onnx模型推理库。在电脑支持的情况下，CPU/GPU上都可以使用float32/float16推理。输入数据为np.array

session=onnxruntime.InferenceSession(onnx位置,providers)：加载模型和框架。providers=['CPUExecutionProvider']/['CUDAExecutionProvider']使用CPU/GPU推理

input\_name=session.get\_inputs()[0].name：获取网络输入名称。[0].name为第1个输入的名称

output\_name=session.get\_outputs()[0].name：获取网络输出名称

pred=session.run([output\_name],{input\_name: np数据})：推理

**import onnxsim**

(pip install onnx-simplifier)

压缩简化onnx模型

model=onnx.load(onnx位置)：加载onnx模型

onnx.checker.check\_model(model)：检测onnx模型，不报错就是没问题

model\_simplify,check=onnxsim.simplify(model)：压缩简化onnx模型

onnx.save(model\_simplify,保存位置)：保存模型

pip install onnxscript

torch.onnx.dynamo\_export()：pytorch2.0版本后的新导出方法

**import tensorrt**

下载：https://developer.nvidia.com/nvidia-tensorrt-8x-download

github：<https://github.com/NVIDIA/TensorRT>

使用函数：<https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-840-ea/api/c_api/index.html>

(安装要从NVIDIA官方下载对应cuda、cudnn版本的压缩包)

需要先安装cuda和cudnn，下载tensorrt版本要对应，只需要压缩包中的bin、include、lib文件，然后添加lib文件路径到系统路径中(windows为：系统->高级系统设置->环境变量->系统变量->Path中加入。linux为：sudo ldconfig lib位置)，然后找到对应版本的whl文件使用pip install .whl。bin中是官方提供的onnx转trt程序

tensorrt是英伟达官方针对自己的硬件设备面向AI工作者推出的一种部署方案，可以改善网络的延迟、吞吐量以及效率。对于trt模型，批量预测相对单张预测只有在小模型上有一点优势，大模型上差不多，甚至单张预测更快，反而单张预测占显存更少

**onnx转trt：使用bin中的trtexec**

linux需要添加环境变量export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:lib位置。命令终端输入trtexec：会看到使用的提示信息

trtexec --onnx=best.onnx --saveEngine=best --useCudaGraph ...

转换过程中有很多提示信息，可以解决大多数问题。转换后会进行速度测试。不指定输入形状时默认为单批量预测(推荐)

# --onnx=onnx模型位置

# --saveEngine=trt模型保存位置

# --noTF32：禁用float32精度

# --fp16：启用float16精度(通常使用fp16)

# --int8：启用int8精度(精度下降较为严重)

# --best：开启所有精度(有的模型是混合精度的)

# --device=0：使用的GPU号码，默认为0

# --useCudaGraph：尝试使用cuda图

**使用tensorrt推理：**



input为输入、h\_input为锁存、d\_input为显存；output同理

使用cuda延迟加载能够提升性能，但需要较高的cuda版本

**import pycuda.driver as cuda：**python的cuda接口，cuda由c和少量c++组成，python要通过pycuda接口来使用cuda计算

**import pycuda.autoinit**：导入后会自动进行一些数据初始化和内存管理

logger=tensorrt.Logger(tensorrt.Logger.WARNING)：创建日志记录器

with open(model\_path, "rb") as f,tensorrt.Runtime(logger) as runtime: // model=runtime.deserialize\_cuda\_engine(f.read())：读取模型

model.num\_bindings：模型的接口总数(输入口和输出口)

for binding in model:：binding为模型接口的字符串名称

//model.get\_tensor\_mode(binding)：判断接口是否为输出返回bool值

//tuple(model.get\_tensor\_shape(binding))：获取接口的形状

//tensorrt.nptype(model.get\_tensor\_dtype(binding))：获取接口的数据类型并转为np的字符串格式

tensorrt.volume(形状:model.get\_tensor\_shape(binding))：计算该形状的一维长度。如输入(1,3,640,640)得到

h\_input=np.zeros(长度:(tensorrt.volume(model.get\_tensor\_shape(binding))),dtype=np.float32)：获取输入的形状(一维)。h\_output同理

h\_input.nbytes：获取h\_input的字节数。float32为，16为2

d\_input=cuda.mem\_alloc(h\_input.nbytes)：分配显存空间。d\_output同

stream = cuda.Stream()：创建cuda流

bindings=[int(d\_input), int(d\_output)]：绑定显存输入输出

context = model.create\_execution\_context()：创建模型推理器

cuda.memcpy\_htod\_async(d\_input, h\_input, stream)：将输入数据从CPU锁存复制到GPU显存。h\_input为输入的数据(一维)

context.execute\_async\_v2(bindings=bindings,stream\_handle=stream.handle)：执行推理

cuda.memcpy\_dtoh\_async(h\_output, d\_output, stream)：将输出数据从GPU显存复制到CPU锁存。h\_output为推理输出的结果(一维)

stream.synchronize()：同步线程

**import gradio**

官方API：<https://www.gradio.app/guides/quickstart>

gradio可以生成可视化的页面用于展示。gradio.Interface快速配置，gradio.Blocks()高级配置

gradio\_app=gradio.Interface(fn,inputs=[...],outputs=[...],examples=[[...],...])：快速配置。fn为运行的程序，fn的参数与inputs中的内容对应，return中的内容与output对应；inputs中指定传入的内容，如['number','text','image']表示可以在页面传入数值(包括小数)、字符串、一张图片(RGB)，此时fn中也要有对应的参数；outputs为输出显示，与inputs同理；examples=[[0,'内容',image],...]会显示到页面中以供选择，内容要与输入对应。inputs和outputs可传入自定义模块来实现更多功能

gradio\_app.launch(server\_port=7860,share=False)：启动服务，启动后默认可在http://127.0.0.1:7860访问。share=False时只能在本地访问，True时可在外部访问，但只有24小时的免费，超过的要在gradio官方购买云服务

**高级配置：**

with gradio.Blocks(theme,title,css) as gradio\_app: // 自定义模块：高级配置。theme为主题颜色；css可自定义页面。页面默认按照自定义模块的写入顺序从上到下排序渲染。配置好后使用gradio\_app.launch启动服务。为了更灵活的调整页面，部分模块可以在外部提前定义好如text=gradio.Textbox，然后在gradio.Blocks()中使用text.render()渲染，与直接写入的效果一致，注意部分模块不支持提前定义要写入gradio.Blocks()内部

theme=gradio.themes.Base(primary\_hue,secondary\_hue,neutral\_hue)：自定义主题。primary\_hue为用户点击框等颜色默认orange；secondary\_hue为选择按钮的颜色默认blue；neutral\_hue为文本等颜色默认gray。推荐primary\_hue和neutral\_hue同色。颜色有暗灰slate、亮灰zinc、琥珀黄amber、翡翠绿emerald、天蓝sky、紫罗兰紫violet、粉紫fuchsia、玫瑰红rose

**自定义模块：**

部分自定义模块也可以在gradio.Interface中使用，主要用于gradio.Blocks。部分模块可以提前在gradio.Blocks外面定义，使用.render()渲染，但部分模块不可以

text=gradio.Textbox(placeholder,label,lines=1)：定义文本输入/输出框。placeholder为框中提示；label为框上方的提示，默认值为fn参数名；lines为框的高度。不同的输入和输出要分开定义

number=gradio.Number(value,label,precision=None)：定义数字输入/输出框。value为框中提示；label为框上方的提示，默认值为fn参数名；precision=0时输入数字的类型会转为整数。不同的输入和输出要分开定义

option=gradio.components.Radio(choices,value,label)：定义输入单选项。choices传入选项列表如['A',1,1.0]；value为默认选项，传入列表中的元素；label为提示

option=gradio.CheckboxGroup(choices,value,label)：定义输入多选项

gradio.Markdown(value)：用Markdown格式写入一段话

gradio.Examples(examples,inputs,label)：添加示例。examples=['A', 'B']；inputs是提前定义好的模块如gradio.Textbox

gradio.ClearButton(components,value)：清除模块的历史记录。components传入模块列表，value为按钮名称

button=gradio.Button(value)：自定义按钮。value为按钮名称

button.click(fn,inputs,outputs)：点击按钮后的操作，需要写入gradio.Blocks内部

button.click(...).then(fn,inputs,outputs)：执行玩click后继续执行then中任务

with gradio.Tab(label):：创建一个标签页模块，可以在页面中选择。比如将图片功能写入一个Tab，文本功能写入一个Tab。label为标签页名称

with gradio.Accordion(label,open=True):：创建一个折叠页模块，可以隐藏其中的内容。label为折叠页名称；open为默认是否展开

with gradio.Column():：里面的内容在页面中垂直排列

with gradio.Row():：里面的内容在页面中水平排列。默认所有模块等宽度，要调整宽度可在各模块中加入参数scale，如果A的scale=2，B的scale=1，则A是B的两倍宽。Row中可以嵌套Column，因此Column中也有scale参数

**对话机器人：**

chatbot=gradio.Chatbot(value=[],height=400,bubble\_full\_width=True)：定义对话框，刷新网页后重置。height为对话框高度；bubble\_full\_width=False时聊天消息长度会自动调整。inputs=['text','chatbot']，outputs=['chatbot']，返回使用yield代替return，函数中的参数首次使用value的值，后续使用上次yield的值，数据格式为[[用户输入,模型输出]...]，当内容为None时会输出省略号

流式输出：需要将函数中的return改为循环+yield，每次循环将一个字加入到输出中，使用流式输出时需要在gradio\_app.launch前设置gradio\_app.queue()

**import flask**

flask是一个用Python编写的Web应用程序框架，可以将python程序包装成一个可以随时调用的服务，存放到服务器上

app=flask.Flask(\_\_name\_\_)：创建一个服务框架

@app.route('/name/',methods) // def flask\_app():...：初始化服务并定义每当调用服务时会执行的功能。'/name/'的'/'不能少，methods=['POST']表示只可以使用post请求

request=flask.request.get\_data()：接收请求，写在def flask\_app()中。如果传输的数据是字典转为的字符串，需要使用request\_dict=json.load(request)将字符串转为字典。如果字典中有图片数据，图片的编解码参考base64

return result：返回结果，写在def flask\_app()中。如果返回的结果是字典，如果忘记用json.dumps()则函数会自动使用

app.run(host='127.0.0.1',port=5000,debug=False)：启动服务。host为监听主机名，设为'0.0.0.0'时使服务可以通过外网访问(此时内网也可)；port为监听端口，可以设为2000-9999；debug=True时提供调试信息。运行后可通过http://host:port/name/访问服务。app.run的方式启动是单线程的

如果后续要使用gunicorn多线程等启动flask，为了避免传入参数时args冲突，需要用args,\_=parser.parse\_known\_args()替代常用的args=parser.parse\_args()

**浏览器访问：**

每搜索一次http://服务器IP:port/name/就会执行一次函数def flask\_app()中的内容

**post请求(import requests)：**

response=requests.post(url=,data)：post访问服务器。response.status\_code为200时传输正常

result=response.json()：将字符串转为字典

**gunicorn**

(pip install gunicorn)

gunicorn是一个 python的WSGI接口协议下的HTTP 服务器，可以创建多个进程，只能安装在linux中

gunicorn [参数] flask\_app:app：用gunicorn启动flask。通常会配置一个gunicorn\_config.py文件，使用[gunicorn -c gunicorn\_config.py flask\_start:app]启动

**参数：**

bind='0.0.0.0:9999'：设置端口，外部访问端口也会从http://host:port/name/变为http://bind/name/

workers=3：设置进程数。推荐核数\*2+1发挥最佳性能

worker\_connections=2000：客户端最大连接数，默认1000

worker\_class='sync'：设置工作模型。有sync(同步)(默认)、eventlet(协程异步)、gevent(协程异步)、tornado、gthread(线程)。sync根据请求先来后到处理；eventlet需要安装库：pip install eventlet；gevent需要安装库：pip install gevent；tornado需要安装库：pip install tornado；gthread需要指定threads参数

threads=1：设置线程数。指定threads参数时工作模式自动变成gthread(线程)模式

timeout=60：启动程序时的超时时间(s)

reload=True：当代码有修改时会自动重启，适用于开发环境，默认False

accesslog='gunicorn\_log/access.log'：设置日志的记录地址。需要提前创建gunicorn\_log文件夹

errorlog='gunicorn\_log/error.log'：设置错误信息的记录地址。需要提前创建gunicorn\_log文件夹

loglevel='info'：设置日志的记录水平。有debug、info(默认)、warning、error、critical，按照记录信息的详细程度排序

**docker**

基于go语言的开源容器引擎，可以将代码和环境放到一个可移植的容器中，方便部署到其他的linux机器上。docker容器轻量化，内存需求极低

镜像：包含程序运行环境的压缩包。包含操作系统、cuda版本、python版本、库等，但不包含cuda驱动等底层东西。一台新的机器只需要cuda驱动等满足条件，就可以加载镜像来创建程序运行环境

容器：使用镜像后创建的程序运行环境。即使在同一个服务器上，各容器之间也是互不干扰的，每个容器都有自己独立的空间

仓库：存放镜像的地方。仓库的管理方法和git很相似

docker镜像保存在本地的var文件中

sudo apt-get install docker：linux中下载docker

systemctl restart docker：重启docker

docker version：查看docker版本

docker pull 镜像源：下载镜像

docker images：查看已有镜像

docker image rm 镜像id：删除docker images中的镜像

docker save -o 保存名称.tar 镜像名/id：保存镜像为.tar文件。id可以不是完整的

docker load -i .tar包：加载镜像。加载后的镜像没有repository和tag

docker tag 镜像id repository:tag：给镜像打标签，id可以不是完整的。repository要按照镜像仓库的格式，tag一般为版本号

docker push repository:tag：上传镜像至镜像仓库。需要事先连接镜像仓库

**Dockerfile：**

Dockerfile：直接创建镜像的配置文件，构建镜像时会运行文件中的命令

FROM 其他镜像：加载其他镜像，以其他镜像为模板。此行必须要有且作为第一个

WORKDIR 路径：进入某个路径中，之后的操作也会在这个路径下执行

RUN 命令1 && 命令2：构建镜像时会执行的命令。RUN指令会产生缓存以便下次创建，使用--no-cache不保存缓存。如果一行太长写不下时，加入’ \’换行。可以把命令放到一个RUN后，也可以拆分为多个，多个时会略微增加大小

CMD 命令：构建容器后、启动容器时会执行的命令

**《数据库与爬虫》**

**基础知识**

**客户端/浏览器：**

客户端会向服务器发送http请求。服务器收到请求后,会解析请求，根据请求信息生成html文档。然后服务器向客户端返回一个http响应，并将html文档作为响应的一部分。客户端解析服务器的http响应，再解析响应中的html文档显示在屏幕上

浏览器网页中右键可以选择查看该网页的html代码，html代码通过浏览器的解析和渲染变成网页页面。浏览器的内核会影响代码解析和渲染的速度和质量

**网页Web：**

html：构建页面的结构，写入内容

css：网页内容的外观和位置

Javascript：网页的动态效果

**http协议：**客户端和服务器通信的数据传输协议

get请求：从服务器获取数据，请求字符串限制1024个字节，比post请求更加高效

post请求：可以向服务器发送数据

**数据库：**

数据库给服务器提供数据，服务器反馈数据显示到网站上

DB(database)：数据库。存储数据

DBMS：数据库管理系统(mysql)。存储数据和管理数据

SQL：绝大多数数据库支持的查询语言

**网址url：https://www.baidu.com/s?ie=...**

http/https：协议(https更加安全)

www.baidu.com：域名

/s：路径

ie=...：参数

**html：**

url、href、src：url为网页链接；url+href为网页内置链接；url+src为一些可替换元素的链接，比如头像等

**爬虫：**

网页源代码：服务器发送到浏览器的代码。静态内容

检查元素中的代码：源代码+动态渲染的内容，最终显示的html。动态内容

静态内容：网页的静态内容基本不变，被请求时会一次性返回内容，不需要连接后台数据库，比如网页中不变的框架和文字。requests发起请求时会获得网页的静态数据

动态内容：网页的动态内容需要服务器调取数据库加载，在加载静态内容后进行，数据内容也是经常变动的，比如淘宝界面的商品。requests发起请求时不会获得网页的动态数据，需要找到动态内容链接再次发起请求

AJAX(Asynchronouse JavaScript And XML)：异步JavaScript和XML。动态网页通过ajax来实现，可以在不重新加载整个网页的情况下，对网页的某部分进行更新

机器人协议：使用robots协议后，爬虫访问站点时会先检查根目录下是否存在robots.txt，该文件可以指定爬虫的抓取范围，如果存在会按照文件中的规则抓取

**爬虫框架：**selenium、scrapy、DrissionPage等，selenium可以模拟用户在浏览器中的操作，等网页渲染后再爬取数据，比较稳定但速度较慢，针对不同浏览器版本要下载不同的驱动；scrapy可以渲染JavaScript从而获取网页的动态数据，异步、协程、速度快、功能齐全；DrissionPage相当于selenium的优化升级版

**Mysql**

数据库。安装需搜索安装教程，安装时注意更改安装位置和数据存放位置，默认安装在C:\Program Files\MySQL，每个模块的安装位置都要单独选择，选不了时用ALT+TAB刷新。卸载时搜索卸载教程。windows右键此电脑、管理、服务、mysql属性中可以关闭开机启动

MySQL Workbench：可以远程连接服务器中的数据库

MySQL Server：数据库，如果本机要作为数据库才下载

windows启动和关闭：可以在任务管理器的服务中启动和关闭mysql

service mysql start：linux启动mysql

service mysql restart：linux重启mysql

service mysql stop：linux关闭mysql

**Navicat**

mysql的可视化管理软件。在navicat中可以连接到mysql，注意连接时可以点击高级选项更改设置位置。Navicat中可以创建新的数据库，设置数据类型等

**import pymysql**

建立python程序与mysql数据库的连接

connect=pymysql.connect(host,port,user,password, database)：建立连接

cursor=connect.cursor()：执行操作的函数

cursor.execute(sql语句)：执行sql操作

cursor.clone()：关闭函数

**import requests**

互联网上的数据传输格式遵循http协议

headers={'User-Agent':...,...}：可以在浏览器中找到本机的headers以伪装成正常的客户端。右键网页、检查、Network、按F5键、任意文件的headers，headers中的信息并非都要填。User-Agent为客户端浏览器类别

**get请求：**

response=requests.get(url,headers=None,timeout=None,verify=True):get访问服务器。不加headers时，请求方的User-Agent为requests库，而很多服务器会检查headers才会返回全部信息；timeout为请求超时时间；verify=True时只能访问有安全证书的网站。response.status\_code为200时传输正常，response.text为字符串格式的网页内容，response.content为字节类型的网页内容，response.request.headers为请求时的headers

headers={'User-Agent':...,...}：可以在浏览器中找到本机的headers以伪装成正常的客户端。右键网页、检查、Networ`k、按F5键、任意文件的headers，headers中的信息并非都要填。response.request.headers查看请求时完整的headers。User-Agent为客户端浏览器类别

**post请求：**

response=requests.post(url,data)：post访问服务器。response.status\_code为200时传输正常

**from lxml import etree**

html=etree.HTML(html\_str)：将字符串解析为html网页的xpath格式

search\_list=html.xpath('/html/head/title/text()')：用路径的方式搜索html中head下title中的文本内容返回到列表中。title/@属性名：返回title中的属性内容；title[@]/text()：只搜索带有限制的title文本内容，限制形式如class="..."；

search\_list=html.xpath('//title/text()')：直接搜索html中的所有的title中的文本内容返回到列表中

**import scrapy**

scrapy框架可以渲染JavaScript从而获取网页的动态数据，异步、协程

scrapy startproject scrapy\_start：创建scrapy项目文件

cd scrapy\_start：进入代码文件。不用再进入子目录scrapy\_start

scrapy genspider example example.com：创建example.py示例文件，会生成在scrapy\_start/spiders中。example.py中可以写入自定义的内容和操作

scrapy genspider 文件名 allowed\_domains：新建文件。 文件类中的name具有唯一性，最好与文件名保持一致；allowed\_domains=["allowed\_domains"]为允许爬取的域名，通常可以删除；start\_urls=["http://allowed\_domains/"]为爬取的url

def parse(self,response):：执行后返回的响应。有response.status\_code等内容

scrapy crawl 文件名：开始爬取网页。文件名不加.py

ROBOTSTXT\_OBEY = True：在settings.py中可以开启或关闭机器人协议

LOG\_FILE = 'scrapy\_log.log'：在settings中加入，可以将信息写入日志

from scrapy.cmdline import execute：

execute(['scrape','crawl','example'])：等同于执行[scrapy crawl 文件名]命令

**xpath：**

scrapy中集成了etree.HTML的xpath用法，但写法有一些变动

response.xpath('//title/text()').getall()：等同于html.xpath('//title/text()')

**import DrissionPage**

使用文档：<https://g1879.gitee.io/drissionpagedocs>

github：<https://github.com/g1879/DrissionPage>

DrissionPage使用 9222 端口，会自动搜索到本地的google浏览器

**基本操作：**

page.title：网页的标题

page.html：网页的html

page\_search=page.s\_ele()：page的静态副本，仅搜索用法一致。page会根据网页动态内容改变，创建静态副本后内容不变，搜索速度加快

element=page.ele('xpath://a[@class="title"]')：查找第一个满足的元素

element\_list=page.eles('xpath://a[@class="title"]')：查找所有满足元素

element.link：链接

element.text：文本内容

page.download(url, r'save.jpg')：下载图片

**操控浏览器：**

操控浏览器等同于用户使用浏览器，共用页面和缓存

page=DrissionPage.ChromiumPage()：接管google浏览器并获取当前页面信息，如果没打开浏览器会自动打开到首页。接管之后不要再用鼠标操作浏览器。page中的内容是动态的，会随着网页动态内容的加载和函数操作而改变

page.get('https://gitee.com/login')：转跳到网页。示例网页需要账号密码登录

page.wait.load\_start(timeout=0.8)：让程序等待所有的动态内容加载完，受网络影响。实际中全部加载完会非常慢，只需0.8左右就可以加载出完整的html，剩下的时间大多浪费在浏览器显示图片等上面

page.ele('#user\_login').input('账号')：定位到账号输入口，并输入账号

page.ele('#user\_password').input('密码')：定位到密码输入口，并输入密码

page.ele('@value=登 录').click().left()：找到登录按钮，并点击

.click().left()/.right()：模拟鼠标左/右键点击

page.quit()：关闭浏览器

page.tab\_id：当前标签页的id

page.process\_id：浏览器进程id

page.refresh()：刷新当前页面

page.back(1)：按浏览器历史回退1个网页

page.forward()：按浏览器历史前进1个网页

page.scroll.to\_bottom()：滚动页面到底部

page.scroll.to\_top()：滚动页面到顶部

image\_byte=page.get\_screenshot(as\_bytes='jpg',left\_top=(0,0),right\_bottom=(1400,700))：页面截图，as\_bytes='jpg'时返回.jpg格式的字节类型图片；不指定left\_top和right\_bottom时截全图

page.tab\_id：当前标签页的id

page.tabs：所有标签页id组成的列表

page.to\_tab(id)：切换控制的标签页

page.new\_tab(url)：新建标签页。此时控制对象仍为原标签页。page.wait.load\_start(timeout=0.9)

tab=page.get\_tab(page.latest\_tab)：新建一个tab对象控制刚创建的页面，tab对象的功能类似page，但权限比page低

page.close\_tabs()：关闭当前标签页

page.close\_other\_tabs()：关闭除当前标签页外的所有标签页

**访问网页：**

page=DrissionPage.SessionPage()：创建页面对象

page.get('http://www.baidu.com')：访问网页

page.response：等同于requests库的response

**测试：**

element\_list=page.eles('xpath://a[@class="title"]')：查找所有商品

element\_list[0].link：第一个笔记的链接

element\_list=page.eles('xpath://a[@class="author"]')：查找所有商品

element\_list[0].link：第一个用户的链接

element=page.ele('xpath://meta[@name="description"]')：笔记的内容

element.html：取出字符串内容，还需要进一步处理

**笔记：**

page.title：笔记标题

regex\_keyword=re.compile('keywords="(.\*?)"')：关键词正则表达式

keyword\_list=re.findall(self.regex\_keyword, element.html)[0].split(', ')：关键词

regex\_content=re.compile('content="(.\*?)#')：内容正则表达式

content\_list=re.findall(self.regex\_content, element.html)[0].split()：内容

**《金融》**

**import tushare**

官网：<https://tushare.pro/>

需要先在官网注册账号，获取密钥

tushare.set\_token('...')：设置密钥

pro=tushare.pro\_api()：初始化接口

df=pro.daily(ts\_code='000001.SZ',start\_date='20240101',end\_date='20240201')：获取股票数据。时间格式为20240101，不能为2024-01-01

**import WindPy**

需要先在wind金融终端购买账号才能使用其API，每日和每周的数据导出额度都有限定。先在wind金融终端内部使用'修复python接口'选项，并查看选用的python环境是否正确；然后使用命令'python wind安装位置/installWindPy.py wind安装位置'安装WindPy

wind金融终端中的代码生成器可以自动生成代码

WindPy.w.start()：启动WindPy接口

WindPy.w.isconnected()：判断WindPy接口是否启动成功

wind\_data=WindPy.w.wsd(股票号,变量名,起始日期,结束日期,options)：从wind金融终端获取股票数据。股票如'600004.SH'；变量名可为列表，如开盘价(open)、技术指标(ADTM)等；日期可以为字符、pandas时间数据；options可为"Currency=CNY;PriceAdj=F"，可以不填，Currency=CNY转换为人民币显示，PriceAdj=F为前复权，PriceAdj=B为后复权，不填则不复权，如果变量中有技术指标还需在options中设定一些周期参数，对于有多条线的指标会有\_IO参数用于选择是哪条线，如果此时如果要获取全部的线只能多次用函数。有如果结束日期超过了上限，会得到Nan

wind\_data.Data：取出获取到的数据，格式为二维列表，每行为每个变量的数据

wind\_data.Fields：取出变量名

wind\_data.Times：取出时间序列列表

**import finta**

github：<https://github.com/peerchemist/finta>

finta是计算股票技术指标的库，支持80多种指标的计算

df\_count=pd.read\_csv(...)：用于计算指标的数据，必须有['open', 'high', 'low', 'close', 'volume']，也可以自己加入其他变量

finta.TA.SMA(df\_count,a,column='close')：计算a日均线。返回DataFrame

finta.TA.EMA(df\_count,a,column='close')：计算a日指数均线

finta.TA.DEMA(df\_count,a,column='close')：计算a日双指数均线

**import ctypes**

ctypes.windll.kernel32.SetThreadExecutionState(0x80000000 | 0x00000001 | 0x00000002)：防止息屏。程序结束后会失效

ctypes.windll.kernel32.SetThreadExecutionState(0x80000000)：恢复息屏

**import pyperclip**

python自带的剪切板操作库

pyperclip.copy(text)：将text内容复制到剪切板。支持中文

text=pyperclip.paste()：获取剪切板中的内容

**import pyautogui**

使用文档：<https://pyautogui.readthedocs.io/en/latest/>

针对windows的GUI自动化库。可以模拟鼠标和键盘进行操作

图形用户界面GUI：采用图形方式显示的计算机操作用户界面

坐标轴：x轴为平面从左到右，对应w；y轴为平面从上到下，对应h

**鼠标操作**

w,h=pyautogui.size()：获取屏幕分辨率的宽和高

x,y=pyautogui.position()：获取当前鼠标位置

pyautogui.onScreen(x,y)：判断坐标是否在屏幕坐标系中，返回True或False

pyautogui.moveTo(x,y,duration=0)：将鼠标移动到(x,y)。duration为移动速度(秒)

pyautogui.move(x,y,duration=0)：将鼠标在原位置上移动(x,y)距离，x、y可为负数

pyautogui.dragTo(x,y,duration=0,button='left')：按住鼠标并拖动至(x,y)然后放开。button='left'时使用鼠标左键

pyautogui.drag(x,y,duration=0,button='left')：按住鼠标并拖动(x,y)距离

pyautogui.click(button='left',clicks=1,interval=0)：点击鼠标。clicks为点击次数，interval为点击间隔

pyautogui.mouseDown(button='left')：按住鼠标

pyautogui.mouseUp(button='left')：放开鼠标

pyautogui.scroll(a)：鼠标中键滚动。a通常为几百，正数为向上滚动

**键盘操作**

pyautogui.write(message,interval=0)：写入message文本内容

pyautogui.keyDown(key)：按住键盘中的key键

pyautogui.keyUp(key)：松开键盘中的key键

pyautogui.press(key,presses=1,interval=0)：按下键盘中的key键。key可以传入列表为依次按下；presses为按下次数，会先按完列表中的再重复列表

pyautogui.hotkey('A','B',interval=0)：依次按下'A'、'B'键再松开

**图片匹配**

image=pyautogui.screenshot(path,region=(x1,y1,w,h))：截取当前屏幕。path填入时会保存到本地；region不填时截取整个屏幕。image为PIL图片

x,y=pyautogui.locateCenterOnScreen(image/path,confidence=1)：在当前屏幕中匹配图片并返回中心点坐标。可以传入上一步的image或图片位置；confidence为置信度，建议0.8-0.85。没有匹配到会返回异常pyautogui.ImageNotFoundException。图片匹配的依据是像素，当匹配图片和目标图片分辨率不一致时基本无法匹配

**import pywinauto**

github：<https://github.com/pywinauto/pywinauto>

跨平台的GUI自动化库。可以对软件应用进行操作，软件可以是单进程或多进程。pywinauto的原理是先将软件解析为层级嵌套的窗口，然后通过解析出的窗口信息进行操作。软件更新可能导致解析出的信息变化。可以使用工具查看窗口信息(各工具解析结果会有差别)：https://github.com/blackrosezy/gui-inspect-tool

UISpy.exe：可以查看所有软件的窗口和控件信息，要在程序运行后启动软件

ViewWizard.exe：可以点击查看已启动窗口的各部位信息

Inspect.exe：在UI Automation中可以查看窗口中的控件信息

app=pywinauto.application.Application(backend='uia').start(exe\_path)：启动软件，exe\_path为启动程序的位置。很多软件启动后需要等待一会time.sleep(1)

app=pywinauto.application.Application(backend='uia').connect(process=a)：连接到已有程序，a为进程号(PID)

app.process：查看进程号

app.windows()：查看当前程序的主窗口名称。名称可以有多个，它们是等效的

window=app[窗口名称]：选择窗口

window.print\_control\_identifiers()：打印窗口中的所有子窗口信息。列表中为等效的窗口名称，child\_window中的信息也可以用来选择窗口

window1=window.child\_window(best\_match=窗口名称)：根据名称选择窗口

window1=window.child\_window(...)：根据详细信息选择窗口。'...'为打印出的信息

window.texts()：获取文本信息，返回列表

image=window.capture\_as\_image()：根据窗口的坐标位置截图。注意只是根据坐标位置截取桌面，需要截取的窗口要在最上方

image.save(path)：保存图片

**窗口类型**

Dialog：对话框

Pane：窗格

StatusBar：状态栏

Button：按钮。使用window.click()点击按钮

**import itchat**

github：<https://itchat.readthedocs.io/zh/latest/#itchat>

微信聊天机器人

itchat.auto\_login(hotReload=True)：登录微信。hotReload在一定时间登录不用扫码

itchat.logout()：退出登录

itchat.send(text,toUserName='filehelper')：给文件传输助手发消息