



搬运工：保罗史蒂芬乔治洪 ~ [paul.ht](#)

布局结构：

- Python
 - Pytorch
 - Tensorflow
 - Python
 - opencv2
 - MATLAB
 - Linux
 - Git
 - Mac
 - Markdown
 - LaTeX
 - Others
-

- Python
 - Pytorch
 - Tensor
 - item()
 - torchvision.transforms
 - torchvision.save_image
 - Relu
 - nn.ConvTranspose2d
 - torchvision
 - cv2的坑
 - 调节学习率
 - torch.nn.DataParallel()
 - torch.nn.DistributedDataParallel
 - num_workers
 - device
 - multinomial
 - load_lua -> torchfile.load

- hook
- torchsummary
- cProfile
- Tensorflow
 - tf.Session()
 - tensorflow 之 checkpoint
 - 选择GPU
 - tf.nn
 - 卷积探讨
 - 感受野
 - ResNet
 - (N,C,W,H)
 - 优化器
 - training accuracy
- Python
 - print
 - np.clip()
 - np.random.choice()
 - 排序
 - zip
 - eval()
 - f-string
 - glob
 - os.path.dirname(__file__)
 - argparse
 - bool型argparse 坑
 - class
 - __call__()
 - __dir__()
 - Python函数——传对象(call by object)
 - globals()
 - zfill
 - ravel() & flatten()
 - np.rollaxis ()
 - matplotlib

- plt.plot()
 - opencv2
 - resize
- MATLAB
 - MATLAB bsxfun
 - Python 与 MATLAB的一些函数区别（细节）
 - 复数域
- Linux
 - bash
 - adduser useradd
 - sudo
 - chown
 - ls
 - 软链接
 - ssh
 - 查看CPU GPU使用情况
 - 输出机制
 - export & echo
 - tar
 - scp
 - Ctrl类快捷键
 - 查看位置
 - Linux查看文件大小数量
 - Linux 查看硬盘分区内存
 - mkdir
 - 查看/杀死 进程
 - ps ax | grep python
 - dos2unix
 - 安装MATLAB
 - windows 远程连接 linux
 - rename
 - vim
 - ~/.vimrc
 - 我的~/.vimrc
 - vim自动补全

- vim Bundle
- pip
- conda
- Linux更改默认Python版本
- 查看网关
- slurm集群管理
- tmux
- \$PATH
- CUDA配置
- Git
 - push
 - clone
 - pull
 - diff
 - .gitignore
 - commit
- Mac
 - 常用快捷键
 - 新建文件
 - 隐藏文件
- Markdown
 - Markdown 超链接
 - Markdown 空格
 - Markdown 代码
 - Markdown 公式
 - Markdown 图片
 - Markdown 目录
- LaTeX
 - VSCode 编译器
 - 一些符号代码
- Others
 - paper writing
 - 插入图片
 - cmd 脚本
 - USB失效

Python

Pytorch

连接，返回tensor:

```
torch.cat(inputs, dimension=0)
```

分块：

```
torch.chunk()
```

Tensor

`torch.tensor` 会从data中的数据部分做拷贝（而不是直接引用），根据原始数据类型生成相应的`torch.LongTensor`、`torch.FloatTensor`和`torch.DoubleTensor`

而 `torch.Tensor()` 是python类，更明确地说，是默认张量类型 `torch.FloatTensor()` 的别名，`torch.Tensor([1,2])` 会调用Tensor类的构造函数 `__init__`，生成单精度浮点类型的张量。

会改变tensor的函数操作会用一个下划线后缀来标示。比如，`torch.FloatTensor.abs_()` 会在原地计算绝对值，并返回改变后的tensor，而 `tensor.FloatTensor.abs()` 将会在一个新的tensor中计算结果。

若将数据 `a` 由 `torch.DoubleTensor()` 转化为 `torch.FloatTensor()`，可记为 `a.float()` 或 `a.to(torch.float32)`。

Tensor

item()

`item()` 只针对仅含一个元素的张量，取出其值。若为多个元素的张量，可考虑 `tolist()`。

torchvision.transforms

- `torchvision.transforms.ToTensor()`
输入为 `PIL.Image` 类型 或 `numpy.array` 中的 `numpy.uint8` 类型时，才会对其归一化 (scale)，即除以255。
 - `transforms` 中的一些变换如 `Resize()`, `Crop()` 等输入必为 `Image` 型，`numpy.array` 会报错。
 - 注：`pytorch` 的图片接口为 `PIL.Image` 库，该库读取的图片类型为 `Image.Image`，而 `cv2` 读取的图片则与 `numpy` 库一致，即 `numpy.ndarray` 型。
-

t 代指数据：

cpu转gpu使用 `t.cuda()`

gpu转cpu使用 `t.cpu()`

tensor转numpy使用 `t.numpy()`

numpy转tensor使用 `torch.from_numpy()`

cpu,gpu转variable使用 `Variable(t)`

Variable转cpu, gpu使用 `v.data`

注意：`y = Variable(t.cuda())` 生成一个节点y，`y = Variable(t).cuda()`，生成两个计算图节点t和y

注：`torch 0.4.0`及其后续版本合并了Variable与Tensor，故Variable不再使用。

GPU上的Tensor不能直接转为numpy，需先转为CPU上的Tensor再转为numpy。

torchvision.save_image

```
mul(255).add_(0.5).clamp(0, 255).permute(1, 2, 0).to('cpu', torch.uint8)
```

对于uint8类型，torch与numpy均为**向下取整**，故先+0.5再clamp。

```
torch.Tensor: a.to(torch.uint8)
numpy.array: a.astype(np.uint8)
```

Relu

```
torch.nn.ReLU(inplace=False)
```

inplace为True，将会改变输入的数据，否则不会改变原输入，只会产生新的输出。默认为False。

nn.ConvTranspose2d

$$H_{out} = (H_{in} - 1) \times stride - 2p + dilation \times (k - 1) + output_padding + 1$$

反卷积即对应上采样过程，HW大小的计算对应卷积计算公式的逆过程。nn.ConvTranspose2d函数默认参数为：dilation=1, output_padding=0。

torchvision

toTensor() 中有归一化，且若是二维会扩成三维 `[:, :, None]`。

ImageFolder 类中 pil_loader 返回时会 `img.convert('RGB')`；classes 为 [name], class_to_idx 为 {name: idx}, samples 为 [(path, target)]。

cv2的坑

因cv2读取图片为BGR格式，若想将其转为RGB格式后转为tensor，

`img = img[:, :, -1]` 后使用 `torch.from_numpy(img)` 会报错。

解决方案一：

```
img = img[:, :, -1].copy()
```

解决方案二：

```
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

调节学习率

在Pytorch 1.1.0及以后的版本中，应先更新优化器optimizer，再更新学习率，代码框架可如下所示：

```
scheduler = ...  
for epoch in range(100):  
    train(...)  
    validate(...)  
    scheduler.step()
```

How to adjust Learning Rate

Prior to PyTorch 1.1.0, the learning rate scheduler was expected to be called before the optimizer's update; 1.1.0 changed this behavior in a BC-breaking way. If you use the learning rate scheduler (calling `scheduler.step()`) before the optimizer's update (calling `optimizer.step()`), this will skip the first value of the learning rate schedule. If you are unable to reproduce results after upgrading to PyTorch 1.1.0, please check if you are calling `scheduler.step()` at the wrong time.

torch.nn.DataParallel()

model, optimizer 等均可用DataParallel包裹，即表示用多块GPU训练。

(CUDA_VISIBLE_DEVICES=0,1,2 此种方式仍为单卡训练，只是占用多块卡的显存。)

以model为例，其一旦被DataParallel包裹之后，其对应的参数state_dict的keys前会多七个字符，即 `module.`。所以在读写checkpoint时需注意，单卡时不能有 `module.`，所以读取一个多卡训练的checkpoint，中间需加入 `.module`，即由 `model.state_dict()` 变为 `model.module.state_dict()`，其实就相当于把读取的参数字典的keys去掉了前七个字符 `module.`。

当然了，若果存储模型时就选择单卡型，即

```
torch.save({'model': model.module.state_dict()}, save_path)
```

则读取时就不需进行去module的操作。

同理，读取单卡checkpoint进行多卡训练时，按单卡代码定义好model（注意此时从CPU转到GPU上，即末尾加个 `.cuda()` 或 `.to(device)`，device需定义一下，可为 `'cuda'`），optimizer等，最后加一个

```
model = torch.nn.DataParallel(model)
```

即可，此句代码相当于在model的state_dict的keys前加了个 `module.`。

总结: `torch.nn.DataParallel()` 相当于在checkpoint的state_dict()的keys前加上了 `module.`，意味着对应多卡；单卡的state_dict()则无 `module`，`model.module.state_dict()` 则为去掉了 `module.` 的state_dict()。

参考：[Missing key\(s\) in state_dict: Unexpected key\(s\) in state_dict:](#)

torch.nn.DistributedDataParallel

这才是真正的分布式训练，且为官方推荐，即使是在单节点多卡上使用时。简单说明一下分布式的常用参数：

```
--dist_url 通信地址，rank为0的进程在这个地址对应的机器上运行。可取为127.0.0.1:8888，8888为任意可用端口，在同一节点上若运行了
--world_size 进程数量，一般为节点数x每个节点上的GPU数，一个GPU对应一个进程。
--rank 进程的编号，从0开始。因为多进程的重复输出等现象，建议输出日志、存储模型等时调加一句if语句就可以了，如当前rank为0才输出
```

当用 `python -m torch.distributed.launch *.py` 训练时，`*.py` 中需要一个args参数为 `local_rank`。

笔者在单节点多卡上进行分布式训练时，习惯用 `CUDA_VISIBLE_DEVICES=2,3 python *.py` 的方式

来运行，此时程序会找到编号为2, 3的两块GPU，且给它们的rank即为0和1。

在argparse里有一个 `local_rank` 参数：

```
parser.add_argument("--local_rank", type=int, default=-1, help="number of cpu threads to use during batch generation")
```

通过 `torch.distributed.launch` 来启动训练，`torch.distributed.launch` 会给模型分配一个 `args.local_rank` 的参数，所以在训练代码中要解析这个参数，也可以通过 `torch.distributed.get_rank()` 获取进程id。

关于ImageNet等的分布式训练，可参考Github代码: [distributed-pytorch](#)

num_workers

`torch.utils.data.DataLoader` 常以batch的方式读取数据，其参数 `num_workers` 表示所用核数（并行读取），`num_workers=0` 表示只用一个主进程读取。

在笔者用风格迁移做数据增强的实验中，风格迁移本身需调用一个VGG网络，而分类采用的ResNet50网络。继承了 `torch.utils.Dataset` 类，在里面加入了style_transform操作，从而相当于在 `DataLoader` 阶段数据就需要放到cuda里。

采取 `torch.nn.DataParallel` 单节点多卡训练时，会报错 `RuntimeError: CUDA error: initialization error`，解决办法一只要把 `num_workers` 设为0就解决了；但考虑到 `num_workers=4` 等可能会提升速度，笔者还是尝试着去解决，在主函数的 `if __name__ == '__main__':` 后加入

```
import multiprocessing as mp
mp.set_start_method('spawn')
```

便解决了，意为用多进程读取数据。（注意此时这两行代码必须得写在 `if __name__ == '__main__':` 后，不然也可能报错。）

而同样的代码改为 `torch.nn.DistributedDataParallel` 版本的分布式单节点多卡训练后，`num_workers=0` 好像也报错了，猜测是分布式本身就是多进程，与只有一个主进程读取数据矛盾。然后改为 `num_workers=4` 报了个风格迁移的网络的模型参数weight和输入weight不在一张卡上，稍微修改了下`vgg.cuda()`和`decoder.cuda()`在代码中的位置，用了

个 `.to(torch.cuda.current_device())`，便解决了，因为分布式下每个进程会使用一块GPU，此时load数据时保证风格迁移网络和数据输入都在这张卡上，就没有问题了。

device

`torch.cuda.current_device()` 返回的是一个整数，如0 1，而获取某张量当前的device，用`str()`包裹后即为'cuda:0'的形式。

```
a = torch.tensor([2, 3])
device = torch.cuda.current_device() # 0 int
b = a.to(device) # a: tensor([2, 3]) b: tensor([2, 3], device='cuda:0')
b.device # device(type='cuda', index=0)
str(b.device) # 'cuda:0'
```

multinomial

```
torch.multinomial(input, num_samples, replacement=False, out=None) → LongTensor
```

按权重张量input的概率采样num_samples次。

参考：[torch.multinomial\(\)理解](#)

load_lua -> torchfile.load

pytorch由0.4版本升级为1.0+版本后，一些函数会发生变化。

对于训好的老式参数模型，读取函数由 `load_lua` 变为 `torchfile.load`。

在一次实际操作中，记读取的模型为 `vgg`，则其第一层的权重调用方式由 `vgg.get(0).weight` 变为 `vgg.modules[0].weight`。

hook

获取中间层feature map时经常利用hook函数，如 `register_forward_hook(fn)`，而其参数 `fn` 为一函数，定义形式为 `fn(module, input, output)`，其中的 `input` 和 `output` 均为 tuple 型，里面元素为 Tensor。

torchsummary

可借助 `torchsummary` 库查看网络结构，包括各层feature map的大小维度。

```
from torchsummary import summary
summary(net, (3, 32, 32), device='gpu')
```

cProfile

借助cProfile库观测代码各部分消耗时间。

如下第一行所示，在原有python运行脚本汇总加入 `-m` 和 `-o` 参数即可，其中 `-o` 表示输出文件名。之后在对自定义命名的 `time.profile` 运行第二行即

可，`p.sort_stats('time').print_stats(50)` 表示按运行时间降序，只显示前五十项。

```
python -m cProfile -o time.profile tmp.py
python -c "import pstats; p=pstats.Stats('time.profile'); p.sort_stats('time').print_stats(50)" > t
```

多进程之间共享全局变量

在做ISBI数据集的实验时，valid时会开个子进程来调用路径为 `script_path` 的java脚本，其会计算指标值。而想在主进程中获取指标值时，便可借助 `subprocess` 库了。

```
import subprocess
# subprocess.call([Fiji_path, script_path, original_label, proposal_label, tmp_file]) # 不需要子进程返回信息时运行此行
return_info = subprocess.Popen([Fiji_path, script_path, original_label, proposal_label, tmp_file], shell=False, stdout=subprocess.PIPE)
while return_info.poll() is None:
    line = return_info.stdout.readline()
    line = line.strip().decode('utf-8')
    ...
```

当然了，如果使用子进程只是为了获取一个值的话有些大材小用。比如在此实验的java脚本中把指标值保存在一个文件里，然后python代码里再读取该文件即可。

此外查到一个python多进程共享变量Value：

```
multiprocessing.Value(typecode_or_type, *args[, lock])
```

比如 `Value("i", 0)`，i表示整型，两个字节。"d" 表示float？

再比如 `import ctypes; Value(ctypes.c_float, 1.0)`。

参考：[Python中多进程间通信（multiprocessing.Manager）](#)

Tensorflow

tf.Session()

`sess.run()` 返回的不是张量类型，为numpy类型，如 `np.ndarray`，`np.float32` 等。

tensorflow由`Session.run()`或`eval()`返回的任何张量都是numpy数组类型。

```
with tf.Session().as_default() as sess:
    code A
code B
```

加了 `as_default()` 后会话结束仍可输出 `run()`，`eval()` 的值，即在code B对应的代码块仍可调用这些函数。

tensorflow 之 checkpoint

```
tf.train.get_checkpoint_state(checkpoint_dir,latest_filename=None)
```

该函数返回的是checkpoint文件CheckpointState proto类型的内容，其中有model_checkpoint_path和all_model_checkpoint_paths两个属性。其中model_checkpoint_path保存了最新的tensorflow模型文件的文件名，all_model_checkpoint_paths则有未被删除的所有tensorflow模型文件的文件名。

参考：<https://blog.csdn.net/change4foreve/article/details/80268522>

选择GPU

1. `tf.device('/gpu:2')`

虽然指定了第2块GPU来训练，但是其它几个GPU也还是被占用，只是模型训练的时候，是在第2块GPU上进行。

2. `os.environ['CUDA_VISIBLE_DEVICES']='2'`

在训练模型的时候，使用了第2块GPU，并且其它几块GPU也没有被占用，这种就相当于在我们运行程序的时候，将除第2块以外的GPU全部屏蔽了，只有第2块GPU对当前运行的程序是可见的。同样，如果要指定2，3块GPU来训练，则上面的代码中的 '2' 改成 '2, 3' 即可。

3. `CUDA_VISIBLE_DEVICES=2 python train.py`

在终端中运行命令时选择GPU。

tf.nn

1. 如果只是想快速了解一下大概，不建议使用 `tf.nn.conv2d` 类似的函数，可以使用 `tf.layers` 和 `tf.contrib.layers` 高级函数。
2. 当有了一定的基础后，如果想在领域进行深入学习，建议使用 `tf.nn.conv2d` 搭建神经网络，此时会帮助你深入理解网络中参数的具体功能与作用，而且对于loss函数需要进行正则化的时候很便于修改，能很清晰地知

道修改的地方。而如果采用 `tf.layers` 和 `tf.contrib.layers` 高级函数，由于函数内部有正则项，不利于深入理解。而且如果编写者想自定义loss，此时比较困难，如果读者想共享参数，计算loss函数中的正则项时，应该只计算一次，如果采用高级函数可能不清楚到底如何计算的。

卷积探讨

输出大小m与输入大小n的关系，其中p表示补丁padding大小，k表示卷积核kernel大小，s表示滑动步长stride：

$$m = \frac{n + 2 * p - k}{s} + 1$$

那么当不能整除时，各大框架如何处理呢？

先温故一下两大框架的2维卷积函数：

```
tf.nn.conv2d(input, filter, strides, padding)
```

其中input大小为[batch, height, width, channel_in]，filter大小为 [height_kernel, width_kernel, channel_in, channel_out], strides为 [1, stride, stride, 1], padding有'SAME'和'VALID'两个选项。

```
torch.nn.Conv2d(in_c, out_c, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)
```

其中参数一看就明白（论torch的可读性），就不赘述了。

Tensorflow卷积池化均**向上取整**，简单粗暴。其有'SAME'和'VALID'两种补丁模式：前者超过原图边界处用0填充，当kernel为奇数时，padding可能只补一边；后者确保不超过边界，可能会丢失一些信息。

要保持图片尺寸不变，看一个特例，即常用stride=1，且kernel为奇数，此时只需 $k-2p=1$ 即可。

PyTorch则**向下取整**。

以Resnet经典一层（[1x1, 3x3, 1x1] + [1x1] shortcut）为例，padding为0输出大小为：

$$m = \lfloor \frac{n-1}{s} + 1 \rfloor$$

当s=1时，大小不变；当s=2时，若输入n为偶数则m为其一半，为奇数则m相当于对n/2向上取整。

当padding只补一边时很有意思，caffe补左上，Tensorflow补右下，Pytorch补一圈（仍要保证大小不变时有待探究）。

感受野

$$RF_n = RF_{n-1} + (kernel_size - 1) * stride$$

RF(n): 当前层感受野

RF(n-1): 上一层感受野

kernel_size: 当前层卷积核大小

stride: 之前所有层stride的乘积

特殊: 二维stride=1时，共l层，每层kernel大小均为k（即k × k），则最后一层像素点感受野为

$$(k - 1) * l + 1$$

ResNet

Bottleneck每个block出去channel 为 planes * expansion, 如 512 * 4。

(N,C,W,H)

tensorflow默认为NHWC，其访存局部性更好；而torch的NCHW为GPU推荐方式。

优化器

据说SGD比ADAM稳定。

training accuracy

某些情况下全体数据集上的training accuracy显示为100%时不一定为100%，如在采用BatchNormalization模块时， μ 和 σ 随着batch变化而变化，而训练集上的准确度是以batch为单位来测量的。

Python

print

不换行输出，`\r`表示会回到行头：

```
print(*, end='\r')
```

np.clip()

上下界截取。

np.random.choice()

`random.choice()` 函数每次只能选择一个，而 `np.random.choice()` 则可选择多个，但需注意一个默认的参数 `replace=True`，表示选取的元素可能重复。

如下代码表示从 `a` 中不重复地选取三个元素，`a` 可以是 `list`、`np.array` 等类型，其中每一个元素被选到的比例均记录在参数 `p` 中。

```
np.random.choice(a, 3, replace=False, p=[*])
```

排序

```
a.sort()    a也变  返回 None  
sorted(a)   a不变  返回 变后结果
```

zip

python3中 `zip()` 返回iterator, 没有 `.sort()` 属性, 可 `list(zip())` 再调用 `sort()` 或者 直接 `sorted(zip())`。

```
zip(*a) 解压
```

eval()

`eval()` 函数十分强大, 官方demo解释为: 将字符串str当成有效的表达式来求值并返回计算结果。 `exec()` 功能类似, 执行python语句。

f-string

```
f"string"
```

类似`str.format{}`接受的格式字符串。

注: Python3.6才开始有。

参考: [python3.6 新特性: f-string PEP 498: Formatted string literals](#)

glob

`glob.glob()` 函数，里面的通配符匹配，在Windows下是不区分大小写的，而在Linux下是区分大小写的。

故比如在Windows中读取图片时，*.jpg和*.JPG若都放在形参里，则会读取两次，注意。

另需注意 `glob.glob()` 搜索时需有分隔符，即 \ 或 /，不然搜索结果为空。

推荐写法：

```
import glob
import os
imgs = glob.glob(os.path.join(img_path, '*.jpg'))
for img in imgs:
    print(os.path.split(img)[-1])
```

总结：`glob.glob` 的参数是一个只含有方括号、问号、正斜线的正则表达式，同时也是shell命令。

os.path.dirname(__file__)

在脚本test.py里写入 `print(os.path.dirname(__file__))`。

- 当脚本是以完整路径被运行的，那么将输出该脚本所在的完整路径，比如：

```
python d:/pythonSrc/test/test.py
输出：d:/pythonSrc/test
```

- 当脚本是以相对路径被运行的，那么将输出空目录，比如：

```
python test.py
输出：空字符串
```

argparse

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('-n', '--name')
args = parser.parse_args()
```

当 `-n` 和 `--name` 同时存在时，默认采用后者。命令行中输入二者均可（如 `python test.py -n ht`），代码中调用时用全称即 `args.name`。

bool型argparse 坑

在使用argparse时发现无法传递bool型变量，无论命令行输入True还是False，解析出来之后都是True。因为输入首先均作为str类型处理。

出错版本：代码 `test.py` 中

```
parser.add_argument('--trained', type=bool, default=False)
```

解决办法1：

注册自定义回调函数：

```
def str2bool(v):
    if v.lower() in ('yes', 'true', 't', 'y', '1'):
        return True
    elif v.lower() in ('no', 'false', 'f', 'n', '0'):
        return False
    else:
        raise argparse.ArgumentTypeError('Unsupported value encountered.')
```

从而将type由bool改为str2bool即可：

```
parser.add_argument('--trained', type=str2bool, default=False)
```

解决办法2：

将bool型变为str型：

```
parser.add_argument('--trained', type=str, default='False')
```

在主函数中对应判断稍加修改：

```
if args.trained == 'False':  
    code1  
elif args.trained == 'True':  
    code2
```

解决办法3：(推荐!!!)

argparse有参数action, 取值有两种：

```
action='store_true'  
action='store_false'
```

此时运行代码的命令行只需要参数名 `--trained` 即可，其后不需输入具体参数。

`action='store_true'` 意味着 `trained` 取值默认为 `False`。即 `python test.py` 时 `trained` 取值为 `False`；而 `python test.py --trained` 时 `trained` 取值变为 `True`。

若 `argparse()` 中添加 `default`，则 `trained` 默认取值与 `default` 值相同。

`action='store_true'` 分析同理。

参考：<https://stackoverflow.com/questions/15008758/parsing-boolean-values-with-argparse>

class

class里面有多个类的属性时，如多个全连接层`fc1`, `fc2`, `fc3`：

1. 采用`setattr` 和 `getattr`

```
setattr('fc%i', i)
```

2. 直接`self`里构造一个list

__call__()

所有的函数都是可调用对象。

一个类实例也可以变成一个可调用对象，只需要实现一个特殊方法 `__call__()`

`__dir__()`

取类的属性，如 `a.__dir__()`，`a` 表示一个类的对象。

Python函数——传对象(call by object)

结论：Python不允许程序员选择采用传值还是传引用。Python参数传递采用的肯定是“传对象引用”的方式。这种方式相当于传值和传引用的一种综合。如果函数收到的是一个**可变对象**（比如字典或者列表）的引用，就能修改对象的原始值——相当于通过“**传引用**”来传递对象。如果函数收到的是一个**不可变对象**（比如数字、字符或者元组）的引用，就不能直接修改原始对象——相当于通过“**传值**”来传递对象。

测试：numpy数组也是“传引用”，不想传引用时可采用：

```
import copy
def temp(A):
    x = copy.copy(A)
    .....
```

注意对于list等涉及两个维度时可能需采用 `copy.deepcopy()`。

举例：

```
a = [[1, 2], [3, 4], 5]
print(a)

b = a.copy()
print(b)

b[0][0] = 9999
print(b)
print(a)

c = a.copy()
print(c)
c[2] = 8888
print(c)
print(a)
```

输出为：

```
[[1, 2], [3, 4], 5]
[[1, 2], [3, 4], 5]
[[9999, 2], [3, 4], 5]
[[9999, 2], [3, 4], 5]
[[9999, 2], [3, 4], 5]
[[9999, 2], [3, 4], 8888]
[[9999, 2], [3, 4], 5]
```

globals()

该函数会以字典类型返回当前位置的全部全局变量。

zfill

```
str.zfill(n)
```

字符串前面补0 至n位(str指代字符串)。

ravel() & flatten()

`a.ravel()` 和 `a.flatten()` 效果一样。

但前者是产生视图，令 `b=a.ravel()`，`b`变`a`也变，`flatten` 则不变。

但

```
b = a.ravel()
b is a      输出: False
c = a.flatten()
c is a      输出: False
```

np.rollaxis ()

改变维度顺序。

```
np.rollaxis(a, n1, n2)
```

`a`是一个数组，将第`n2`个维度移到`n1`维度前。

常见于图片张量中，比如`C x H x W` 通过`np.rollaxis(a, 2, 1)`变为`C x W x H`。

测试样例：

```
import numpy as np
a = np.arange(24).reshape(2, 3, 4)

print(a, '\n')

print(np.rollaxis(a, 2), '\n')

print(np.rollaxis(a, 2, 1), '\n')

print(np.rollaxis(a, 1, 0), '\n')

print(np.rollaxis(a, 1, 2), '\n')
```

输出：


```
[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]]]
```

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]]
```

```
[[[ 0  4  8]
   [12 16 20]]]
```

```
[[ 1  5  9]
 [13 17 21]]]
```

```
[[ 2  6 10]
 [14 18 22]]]
```

```
[[ 3  7 11]
 [15 19 23]]]
```

```
[[[ 0  4  8]
   [ 1  5  9]
   [ 2  6 10]
   [ 3  7 11]]]
```

```
[[12 16 20]
 [13 17 21]
 [14 18 22]
 [15 19 23]]]
```

```
[[[ 0  1  2  3]
   [12 13 14 15]]]
```

```
[[ 4  5  6  7]
 [16 17 18 19]]]
```

```
[[ 8  9 10 11]
 [20 21 22 23]]]
```

```
[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]]]
```

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]]
```

matplotlib

matplotlib在终端中不能显示图（通过ssh等连接Linux服务器）

```
import matplotlib as mpl
mpl.use('Agg')
import matplotlib.pyplot as plt
```

注：前两句必须在第三句前面。

matplotlib经常用在python shell中用作交互式编程，也有将其作为类似wxpython和pygtk这样的图形化界面使用，也有将其用在网络应用服务器中动态提供图片。因此为了能够包含所有的这些需求，matplotlib提供了指向不同输出的后端，相对应的前端就是用户编写使用的代码接口。后端包含两类，一类是user interface backends（用于pygtk, wxpython, tkinter, qt4, or macosx）这类的交互式后端），另一类则是hardcopy backends（主要用于PNG, SVG, PDF, PS这类图片渲染并保存）。

Agg是一个非交互式后端，这意味着它不会显示在屏幕上，只保存到文件。

plt.plot()

```
from matplotlib import pyplot as plt
```

- **保存图片**

`plt.savefig()` 函数第一个参数为保存路径，如*.png，但png等格式图片清晰度有损，而存为*.svg为无损格式，svg格式可通过浏览器打开。另一个参数为 `dpi`，其值越大图片的分辨率越高，`dpi=500` 在一定程度上已经很清晰了。

- **调节坐标轴刻度**

基础些的为 `xticks, yticks`，`xticks(position, label, rotation)` 表示在position位置标注label，这两个一般为List型，rotation控制标注的旋转。

而进阶一点则可借助 `MultipleLocator`，其后的参数表示刻度间距，而对应的在 `xlim, ylim` 取值错开一点可使第一个标注点不在原点。

如下例所示：

```
from matplotlib.pyplot import MultipleLocator
...
plt.xticks(range(2, 21, 2), list(range(2, 21, 2)))
ax = plt.gca()
y_major_locator = MultipleLocator(2)
ax.yaxis.set_major_locator(y_major_locator)
plt.ylim(57.5, 76.5)
...
```

Plot_tick

opencv2

resize

先宽再高！

resize 为原来的 width 和 height 也还是有一点时间消耗的。

cv2读取图片默认为BGR模式，且 `imshow`，`imwrite` 也都对应BGR模式！

BGR->RGB:

```
img_rgb = img[:, :, ::-1]
```

或

```
b, g, r = cv2.split(img)
img_rgb = cv2.merge([r, g, b])
```

或

```
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

注意Image包(`from PIL import Image`)读取图片时会取该图片的实际通道数, 而.png等类型图片会有4个通道, 即除R、G、B三个外还有一个透明度A, 以R为例, 此时计算公式为: $R = \alpha * R + (1 - \alpha) * A$, $\alpha = 1$ 时即对应RGB。故若Image读取此类图只想得到RGB三通道, 需

```
Image.open(img_path).convert('RGB')
```

而cv2的 `imread` 函数读取方式参数有

`cv2.IMREAD_COLOR`, `cv2.IMREAD_UNCHANGED`, `cv2.IMREAD_GRAYSCALE` 等, 默认即为 `cv2.IMREAD_COLOR`, 会自动得到三通道图。

像素取值为0~255时, `cv2.imshow()` 函数的参数矩阵元素必须为整数, 一般取 `np.uint8` 型。但 `cv2.imwrite()` 存图时参数可以不为整数, 相反取整之后可能存在一定的色彩失真 (若肉眼能观察到)。

MATLAB

MATLAB bsxfun

对两个矩阵A和B之间的每一个元素进行指定的计算 (函数fun指定) ; 并且具有自动扩维的作用。

例如 :

```
J=bsxfun(@minus,im,A);  
J=bsxfun(@rdivide,J,F4);  
J=bsxfun(@plus,J,A);
```

参考 : [MATLAB bxfun](#)

Python 与 MATLAB的一些函数区别（细节）

Python中的matrix类型对象加个 **.A** 可变为np.array型对象。

MATLAB **A\b** 即求解 $Ax=b$ 时的 x , 当 A 不是方阵时仍可求解, 所以比 $\text{inv}(A)*b$ 要强大些 ; Python中 $A\b$ 不是方阵时则不行, 可采用最小二乘思想求解, 如求 $A' A x = A' b$ 。

spdiags函数 Python中位于scipy.sparse中, 当形参为`spdiags(A, d, m, n)`时, Python和MATLAB中的 A 为转置关系。

svd分解 Python中位于np.linalg中, **U, S, V = svd(A)**

如 A 为 3×5 矩阵, 则MATLAB中结果为 : $U: 3 \times 3$ $S: 3 \times 5$ $V: 5 \times 5$, S 为只有主对角元素非零的矩阵
Python中 U V 一致, S 为一向量。两种 S 可通过diag命令互相转化。

对于svd的其他参数, MATLAB的`svd(A, 'econ')`对应Python中的`svd(A, full_matrices=False)`, 注意此时前者返回的 U S V 维度分别为 3×3 3×3 5×3 , 后者则为 3×3 1×3 3×5 , 即除了 S 不同外, V 又为转置关系。

复数域

MATLAB一个复数矩阵(向量也为矩阵)记为 A , 则 **A'** 表示 A 的共轭再转置, 而 **$A.'$** 才表示 A 的转置, `conj(A)` 表示 A 的共轭。

Python中则 **$A.T$** 表示 A 的转置, `A.conjugate()` 或者 `np.conjugate(A)` 表示 A 的共轭。

所以, 当翻译MATLAB代码为Python时, 若为复数域上的矩阵, MATLAB中出现 A' 时, Python务必对应为 **$A.conjugate().T$** 。

另python中有个 `np.vdot()` 函数, `np.vdot(a, b)`中两个形参都必须为向量 ($1 \times n, n \times 1$ 矩阵也可), 但 a 、 b 不管是行还是列向量表示都不影响。`np.vdot(a, b)`表示 a 先取共轭再与 b 做内积 (即点乘求和), 故返回值为一个数值。(而`np.dot(a, b)`时, 若 a 为 $1 \times n$, b 为 $n \times 1$, 返回值为 1×1 矩阵, 即`[[value]]`。)

Linux

bash

推荐bash，比sh更强大。

shell开头写

```
#!/bin/bash
```

adduser useradd

adduser 较为便捷

另useradd也可，如

```
useradd -d /home/hongt -s /bin/bash -m hongt
```

sudo

以下命令可以查看系统上拥有sudo权限的所有用户：

```
getent group sudo
```

添加sudo用户一种方式为在 `/etc/sudoers` 里添加一行：

```
[username] ALL=(ALL:ALL) ALL
```

另一种方式：

```
sudo usermod -aG sudo [username]
```

chown

对于linux系统中的文件，有些无法编辑，除了用chmod更改其读写权限外，也可利用chown来更改该文件所属的用户。对于文件夹加入 -R 参数表示递归即可。

```
sudo chown -R [username] [dirname]
```

ls

ls隐藏pyc文件，可写在 ~/.bashrc 中：

```
alias ls='ls -I*.pyc'
```

注：第一个 ls 可任意命名。

软链接

```
ln -s 原链接路径 软链接路径
```

在链接路径有多层嵌套时，建议采用绝对路径避免出错。

ssh

图形界面：

```
ssh -X user_name@user_IP
```

查看CPU GPU使用情况

查看GPU

```
nvidia-smi
```

动态查看GPU，时间间隔为0.5

```
watch -n 0.5 nvidia-smi
```

查看CPU（按 `q` 退出）

```
top
```

查看显卡cuda版本

```
cat /usr/local/cuda/version.txt
```

输出机制

stdout和stderr两种模式，对应编号分别为1和2。

`[cmd] >[filename]` 表示将输出直接写入filename，但若报错仍打印在终端屏幕。

要使错误也写入文件，不打印在终端，末尾加个 `2>&1` 即可，即

```
[cmd] >[filename] 2>&1
```

若既想把输出打印在终端，又写入文件，则借助 `tee`：

```
[cmd] |tee [file]
```

注意 `tee` 不能写入错误，且发现运行python命令时调到 `schedule` 库时，并未在终端打印信息。此时加个 `-u` 即可解决！默认输出会优先输出stderr，因其不需缓存，而 `python -u` 则意味着完全按照程序顺序输出。

（将python执行脚本输出到屏幕结果直接重定向到日志文件的情况下，使用-u参数，这样将标准输出的结果不经缓存直接输出到日志文件。）

· 上述写入文件命令均为覆盖性写入，若想在尾部追加写入，则对应修改为：把 `>` 改为 `>>`；`tee` 后加入 `-a`。即：


```
[cmd] >>[filename] 2>&1  
[cmd] |tee -a [file]
```

推荐eg.:

```
python -u train.py |tee train.log
```

注：

命令行结尾有 `&` 相当于并行，即在一个终端窗口里各命令可以同时运行，运行了一行命令后可以继续输入。

而没有 `&` 相当于串行，按顺序执行命令，前一命令运行结束后才会运行下一条命令，但前提是前一命令能正常运行，不会报错。

export & echo

linux环境下每次新打开一个窗口都会预执行 `~/.bashrc`。

`export` 给变量赋值， `echo` 查看变量值。

但注意变量生效区域：

case1:

tmux环境下，左边窗口export, 右边窗口echo为空。

case2:

在当前窗口中新建一个脚本`ht.sh`，`vim ht.sh`，在其中export, 关掉脚本并 `bash ht.sh`后，`echo`也为空。

若export写进`~/.bashrc`, 则一直生效。

case3:

在当前窗口export, 然后`vim ht.sh`，在里面echo有值。

发散：**配置CUDA环境**

应在主函数开头便写好 `export CUDA_HOME=...` 等；

若想一直生效，写进 `~/.bashrc` 即可。

tar

- c: create, 建立压缩档案
- x: 解压
- z, -j: 分别表示以gzip和bzip2格式压缩解压
- v: 显示解压或压缩过程
- C: 指定目录, 需提前创建
- t: 查看内容
- r: 向压缩归档文件末尾追加文件
- u: 更新原压缩包中的文件
- f: (必需参数)使用档案名字, 该参数是最后一个参数, 后面只能接档案名。

解压:

```
tar -xvf file.tar //解压 tar包

tar -xzvf file.tar.gz //解压tar.gz

tar -xjvf file.tar.bz2 //解压 tar.bz2

tar -xZvf file.tar.Z //解压tar.Z

unrar e file.rar //解压rar

unzip file.zip //解压zip方式1

tar -zcvf *.zip // 解压zip方式2
```

压缩:

```
tar -cvf jpg.tar *.jpg //将目录里所有jpg文件打包成jpg.tar

tar -czf jpg.tar.gz *.jpg //将目录里所有jpg文件打包成jpg.tar后, 并且将其用gzip压缩, 生成一个gzip压缩过的

tar -cjf jpg.tar.bz2 *.jpg //将目录里所有jpg文件打包成jpg.tar后, 并且将其用bzip2压缩, 生成一个bzip2压缩

tar -cZf jpg.tar.Z *.jpg //将目录里所有jpg文件打包成jpg.tar后, 并且将其用compress压缩, 生成一个umcompress

rar a jpg.rar *.jpg //rar格式的压缩, 需要先下载rar for linux

zip jpg.zip *.jpg //zip格式的压缩, 需要先下载zip for linux
```

注: 若压缩时想排除一些文件或文件夹, 可借助 `--exclude` 参数, 排除多个文件(夹)时则使用多

次 `--exclude`，另排除的文件夹最后不要加 `/`。例如：

```
tar -czvf ht.tar.gz images --exclude=1.png --exclude=images/monkey
```

scp

`-P` 指定端口，`-v` 输出日志信息。

文件数过多等情况下，可能未完全复制，先设置超时时间为无穷大即可。

```
set timeout=-1
```

But failed!

建议先打包(压缩)再复制。

Ctrl类快捷键

`Ctrl+A`：光标移到行首。

`Ctrl+E`：光标移到行尾。

`Ctrl+S`：冻结窗口。注意：这不是保存的命令。当屏幕输出过快时，用户可冻结窗口来查看瞬时的输出。

`Ctrl+Q`：取消冻结窗口。

查看位置

查看安装应用的位置，如Python, ls (package代，下同)等：

```
which package
```

查看Python中安装的库的位置，如numpy,torch等，可通过pip:

```
pip show package
```

Linux查看文件大小数量

当前文件夹总大小

```
du -sh
```

当前文件夹下各文件大小

```
du -sh *
```

当前文件夹下 文件+子目录 个数：

```
ls -l |wc -l
```

当前文件夹下 文件 数目

```
ls -l |grep ^-|wc -l
```

可在 `~/.bashrc` 中 `alias` 该命令， 比如别称为`cal_num`，输入简洁。

Linux 查看硬盘分区内存

```
df -hl  
lsblk
```

`lsblk` 命令用于列出所有可用块设备的信息， 而且还能显示他们之间的依赖关系， 但是它不会列出RAM盘的信。

mkdir

直接make dir时， 超过两层会报错， 而加入 `-p` 参数后则多层均可以。

```
mkdir -p [dir-name]
```

查看/杀死 进程

查看进程：

```
ps -ef |grep [process-name]
```

杀死进程：

```
kill -9 [process-number]
```

ps ax | grep python

查看其他人在运行的代码。

在服务器上某目录下输入：

```
python -m http.server
```

再在自己电脑浏览器中输入 服务器IP:8000 即可访问该目录。

dos2unix

因格式原因，有时候文件从windows复制到linux系统后执行会报错，比如代码文件中的回车空格等问题。先执行一句 `dos2unix(filename)` 即可。

安装MATLAB

刚安装好时只能在MATLAB的安装路径内运行，即在 `...../bin` 内输入 `./matlab`。要使在终端的任意路径下都可运行MATLAB，在 `/usr/local/bin` 里绑定一个软链接即可，即

```
cd /usr/local/bin
sudo ln -s ...../bin/matlab matlab
```

此时在终端任意路径下输入 `matlab` 都可运行了。

windows 远程连接 linux

linux上安装xrdp

注意第一次连接到某个用户前该用户需在linux上运行：

```
echo "xfce4-session" > ~/.xsession
```

然后（此步似乎也可以跳过）

```
sudo service xrdp restart
```

windows上按“win+R”后输入**mstsc**，输入对应的IP、用户名、密码即可。

参考：[Windows10远程桌面Ubuntu16.04](#)

[Linux和Windows间的远程桌面访问](#)

rename

将文件*中的from重命名为to:

```
rename 's/from/to/' *
```

vim

三种模式名字：命令行模式，插入模式，末行模式

命令行模式中输入 `:u` 或 `:undo` 表示撤销。（注：命令行模式下输入 `:` 即进入了末行模式。）

查找函数定义处等: 命令行模式下输入 `;jd`

搜索：命令行模式下输入 `/usr` （usr为待搜索字符串，回车即到该字符串处）

`n` 查看下一个匹配；`N` 上一个

命令行模式下：

```
v 按字符复制
V 按行复制
Ctrl+V 按块复制
```

使用 `v` 命令进入visual模式后：

```
d 剪切
y 复制
p 粘贴
^ 选中当前行，光标位置到行首（或者使用键盘的HOME键）
$ 选中当前行，光标位置到行尾（或者使用键盘的END键）
```

使用 `Ctrl+V` 进入块模式后，可以进行多列的同时修改（比如多行注释，python中为插入#），修改方法是：

选中多列，按键 `Shift+i` 进行块模式下的插入, 输入字符之后，按键 `ESC`，完成多行的插入。

设置自动缩进后，整段复制时下一句会比上一句多个Tab,解决办法：

复制前在末行模式输入：`set paste`

取消则：`set nopaste`

替换文字:

在末行模式输入：`{作用范围}s/{目标}/{替换}/{替换标志}`

例如：`%s/foo/bar/g` 会在全局范围(%)查找foo并替换为bar，所有出现都会被替换（g）。

~/.vimrc

变量名	缩写	含义
tabstop=X	ts	编辑时一个TAB字符占多少个空格的位置。
(no)expandtab	(no)et	是否将输入的TAB自动展开成空格。开启后要输入TAB，需要Ctrl-V<TAB>
(no)smartindent	si	基于autoindent的一些改进(原版本为autoindent)
shiftwidth=X	sw	使用每层缩进的空格数。
softtabstop=X	sts	方便在开启了et后使用退格（backspace）键，每次退格将删除X个空格

在Vim中还可以进行自动缩进，主要有**cindent**、**smartindent**和**autoindent**三种。

cindent Vim可以很好的识别出C和Java等结构化程序设计语言，并且能用C语言的缩进格式来处理程序的缩进结构。可以使用以下命令，启用cindent缩进结构：

```
:set cindent
```

smartindent 在这种缩进模式中，每一行都和前一行有相同的缩进量，同时这种缩进形式能正确的识别出花括号，当遇到右花括号 `}`，则取消缩进形式。此外还增加了识别C语言关键字的功能。如果一行是以`#`开头的，那么这种格式将会被特殊对待而不采用缩进格式。可以使用以下命令，启用smartindent缩进结构：

```
:set smartindent
```

autoindent 在这种缩进形式中，新增加的行和前一行使用相同的缩进形式。可以使用以下命令，启用autoindent缩进形式。

```
:set autoindent
```

参考：[VIM学习笔记 缩进 \(Indent\)](#)

我的~/.vimrc

```
set ts=4
set expandtab
set autoindent
```


使vim在三种模式下光标不一样：

```
if exists('$TMUX')
    let &t_SI = "\<Esc>Ptmux;\<Esc>\<Esc>]50;CursorShape=1\x7\<Esc>\\\"
    let &t_EI = "\<Esc>Ptmux;\<Esc>\<Esc>]50;CursorShape=0\x7\<Esc>\\\"
else
    let &t_SI = "\<Esc>]50;CursorShape=1\x7\"
    let &t_EI = "\<Esc>]50;CursorShape=0\x7\"
endif
```

vim 自动补全

候选框挥之不去，很烦！

```
Ctrl+Y
```

表示退出下拉窗口，并接受当前选项。

其他：

```
Ctrl + P: 向前切换成员
Ctrl + N: 向后切换成员
Ctrl + E: 退出下拉窗口，并退回原来录入的文字
```

参考：[vim中自动补全的快捷键](#)

vim Bundle

插件管理：

- 第一步：安装vundle

```
git clone https://github.com/VundleVim/Vundle.vim.git ~/.vim/bundle/Vundle.vim
```

- YouCompleteMe安装完成时，即在vim末行运行 `:PluginInstall` 后，需进入其文件夹下运行：

```
./install.py
```

注：此时需要已经安装cmake；YCM文件夹里的第三方库里有谷歌的成分，笔者有一段时间下载不了。(发挥聪明才智，把以前编译好的YCM文件夹直接拷过去就好了)

· 清除不要的插件，在 .vimrc 中注释掉对应行后，在末行模式运行：

```
BundleClean
```

pip

从清华镜像源更新gpu版本tensorflow

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple/ --upgrade tensorflow-gpu
```

清华镜像网站设为默认网站

```
pip install pip -U  
pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

conda

```
conda create -n python3 python=3.5  (python3为自定义name, 下同; 3.5为指定python版本)
conda activate python3
conda deactivate
```

```
conda install -n python3 package  (package代指库, 如tensorflow)
conda remove -n python3 package  (全删则package改为--all)
```

```
conda env list  查看所有配置环境, 等价于 conda info --envs
conda list -n python3  查看名为python3环境下所有库
conda search package
```

```
conda install -c spyder-ide spyder=3.0.0  指定包的来源(以spyder为例)
(加一个-c表示从http://anaconda.org下载资源包)
```

例:

```
conda install -c cjj3779 tensorflow-gpu
conda install -c conda-forge opencv
conda install --channel https://conda.anaconda.org/menpo opencv3
```

```
conda install numpy=1.12.1
```

```
conda info tensorflow-gpu=1.2.1
```

`-c` 与 `--channel` 为同一内容的两种表达形式

如安装bottleneck时在anaconda官网搜到一个源头为pandas/bottleneck, 则

`--channel https://anaconda.org/pandas` 等价于 `-c pandas`

添加清华镜像网站：

```
conda config --add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
```

搜索时显示通道地址：

```
conda config --set show_channel_urls yes
```

恢复默认源：

```
conda config --remove-key channels
```

显示库源信息：

```
conda config --show channels
```

显示所有config：

```
conda config --show
```

没有直接重命名，所以间接方式：

```
conda create --name [newname] --clone [oldname]
```

一次实际配置时记录：

安装Pytorch：

```
conda install pytorch=0.4.1 cuda80 -c pytorch
```

```
conda install torchvision=0.2.1 cuda80 -c pytorch
```

安装opencv：

```
conda install opencv -c anaconda
```

(笔者所装为3.4.2版本，hdf5库版本不匹配，所以需先卸载hdf5：

```
conda remove hdf5)
```

pip与conda关系：

pip是python自带的，而conda是安装anaconda或者miniconda提供的，俗称的蟒蛇软件商给的，conda可以用来安装管理python，pip当然不能管理python，pip是python下的，所以用pip来装python不可能，conda却可以装python。

有的人不用conda去管理python环境，他们自己安装自己要的python各个版本，然后通过修改全局变量来实验使用哪个版本。（全局变量就是比如你在某路径中输入python，要使可以运行在其他路径下的python.exe，那么这个python.exe就必须为全局变量。）

通过conda安装的工具包比如tensorflow只会出现在conda list中，不会出现在pip list中，倒过来也一样。

记录一次pytorch指定版本安装过程：

安装 pytorch 1.1.0和 torchvision 0.3.0，服务器上cuda版本为10.1。

参考官网推荐方式：

```
conda install pytorch torchvision cuda-toolkit=10.1 -c pytorch
```

先安装pytorch，采用：

```
conda install pytorch=1.1.0 -c pytorch
```

出现的问题是下载pytorch一直显示网络不行，检验是否添加了清华镜像网站作为channel，已经添加了，后来得知得去掉 `-c pytorch`，因为这意味着不会从清华镜像网站下载。但笔者采取的方式是找到pytorch库的链接，先手动下载到本地，再利用

```
conda install --use-local [pkg]
```

安装，其中 `[pkg]` 表示pytorch库的本地绝对路径，笔者放在 `~/anaconda3/pkg/` 中。

此时在python环境中测试 `import pytorch`，但是报错，信息

为 `libmkl_intel_lp64.so: cannot open shared object file: No such file or directory`，参考[在导入pytorch时libmkl_intel_lp64.so找不到](#)，

搜索到 `libmkl_intel_lp64.so` 对应的的路径后，添加到环境变量 `LD_LIBRARY_PATH` 中即可解决。

解决了pytorch后，再来处理torchvision，采用 `-c pytorch` 会遇到同样的网络问题。去掉后即 `conda install torchvision=0.3.0` 下载源不变，故仍有网络问题。同样先下载到本地再安装，`import torchvision` 测试时会显示没有 PIL 库，查询知 PIL 库不支持python3，改为安装 pillow 库即可，即 `conda install pillow`。

另若去掉版本号，即 `conda install torchvision` 则会自动选择当前最新的pytorch=1.3.0和torchvision=0.4.0，不符合笔者要求，故虽省事，还是得放弃。

另值得一提的是虽然官方上笔者装的版本都对应cuda10.0，服务器上是cuda10.1，但用起来也没问题。另NVIDIA官方cuda10.1有三个版本，10.1表示10.1.105，10.1 update1表示10.1.168，10.1 update2表示10.1.243。

Linux更改默认Python版本

```
ls /usr/bin/python*
alias python='/usr/bin/python3.5'
```

查看网关

```
route
```

slurm集群管理

`srun` , `sbatch` 和 `salloc` 为三大提交任务命令。`salloc` 为交互式, 任务结束后不一定及时释放资源, 对于按时长收费的集群请慎重; 个人喜欢用 `srun`, 其与 `2>&1 |tee [log name]` 配合, 可以在写入文件的同时输出到屏幕上, 甚是舒服; `sbatch` 虽然有 `-o` 命令表示输出文件, 但使用该命令后不能输出到屏幕上。

`srun` 参数众多, 如下列出其单字母简称和对应的全称:

```
srun
-J, --job-name=[job name]
-p, --partiton=[node partition]
--gres=[资源, 如 gpu:2 表示申请两块gpu]
-c, --cpus-per-task=[*]
-n, ntasks-per-node=[*]
-t, --time=[run time]
-q, --qos=[priority level, low/normal/high]
-o, --output=[output file]
[task command]
```

以北大未名一号为例,

```
srun --job-name=STL-train --gres=gpu:2 --qos low --time 120:00:00 python -u train.py 2>&1 |tee train.log
```

查询节点资源:

```
sinfo
```

查询用户任务:

```
squeue -u [username]
```

取消任务:

```
scancel [JOB_ID]
```

tmux

注：集群上不同节点tmux已建session可能不同。

重命名session:

```
tmux rename-session -t [old-name] [new-name]
```

\$PATH

配置环境变量，需加入某个应用时，将相应bin文件的路径添加到 ~/.bashrc 文件中。如：

```
export PATH="$PATH:/home/hongt/anaconda3/bin"
```

其中 `$PATH` 即为已有的环境变量；务必使用双引号！

CUDA配置

安装好CUDA后在 ~/.bashrc 中将其路径加入PATH，再source更新一下即可，`nvcc -V` 便可看到所装版本。

```
export PATH=/usr/local/cuda-11.1/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda-11.1/lib64:$LD_LIBRARY_PATH

source ~/.bashrc
nvcc -V
```

Git

push

git 把本地分支dev推到远程，运行

```
git push origin dev
```

即可，此时远程对应分支名字相同，也为dev。

而如果想把本地分支推到远程新的分支上（名字也不相同），如命名为dev_new，只需：

```
git push origin dev:dev_new
```

clone

- git clone默认会把远程仓库整个给clone下来，但只会在本地默认创建一个master分支。如果远程还有其他的分支，此时用 `git branch -a` 查看所有分支，能看到远程的所有的分支，如remotes/origin/dev，可以使用checkout命令来把远程分支取到本地，并自动建立tracking：

```
git checkout -b dev origin/dev
```

- git clone时也可 -b 参数只克隆指定分支，如只clone 远程的dev分支：

```
git clone -b dev [origin-url]
```

- 而把远程分支拉取到本地时默认名字也相同，若想把远程dev分支拉到本地且命名为dev_new，只需

```
git checkout -b dev_new origin/dev
```

参考：[git clone 某个分支或者所有分支](#)

pull

pull 相当于先 fetch 再 merge。如 `git pull` 等效于先执行 `git fetch origin [当前分支名]`，再执行 `git merge FETCH_HEAD`。

pull 虽然一步到位，但个人多数情况下偏向于后者，一来先 fetch 再借助 diff 命令可以看到远程与本地仓库的区别；二来当拉取时本地和远程有冲突时，也先有个准备。

git fetch 有四种用法：

- `git fetch`
这将更新git remote 中所有的远程repo 所包含分支的最新commit-id, 将其记录到.git/FETCH_HEAD文件中。
- `git fetch [remote_repo]`
这将更新名称为remote_repo 的远程repo上的所有branch的最新commit-id，将其记录。
- `git fetch [remote_repo] [remote_branch_name]`
这这将更新名称为remote_repo 的远程repo上的分支：remote_branch_name。
- `git fetch [remote_repo] [remote_branch_name]:[local_branch_name]`
这这将更新名称为remote_repo 的远程repo上的分支：remote_branch_name，并在本地创建local_branch_name 本地分支保存远程分支的所有数据。

举例说明：将远程origin仓库的xx分支合并到本地的yy分支，常见做法为：

```
git fetch origin xx
git checkout yy
git merge FETCH_HEAD
```

此时的MERGE_HEAD即指向origin/xx，若想合并其他远程分支当然也可，如想合并远程的dev分支到本地当前分支：`git merge origin/dev`。

参考：[git fetch](#)、[git pull](#)、[git merge 的理解](#)

diff

常用于比较本地分支和对应远程分支的区别。可先更新下本地的远程分支：

```
git fetch origin
```

- git log 输出本地与远程的差集（显示远程有而本地没有的commit信息），如比较master分支的区别：

```
git log master..origin/master
```

- git diff 统计文件的改动，一般形式为：

```
git diff [local_branch] [remote]/[remote_branch]
```

其中还有 `--stat` `--cached` `--color` 等参数来控制输出的详尽程度，`--stat` 表示摘要，而 `--cached` 表示已缓存的改动。笔者一般会先加入 `--stat` 看一下diff的概括，若有需要再去掉 `--stat` 看详细区别，如：

```
git --stat diff dev origin/dev  
git diff dev origin/dev
```

参考：[git比较本地仓库和远程仓库的差异](#)

.gitignore

有时候在项目开发过程中，突然心血来潮想把某些目录或文件加入忽略规则，按照上述方法定义后发现并未生效，原因是.gitignore只能忽略那些原来没有被track的文件，如果某些文件已经被纳入了版本管理中，则修改.gitignore是无效的。那么解决方法就是先把本地缓存删除（改变成未track状态），然后再提交。命令如下：

```
git rm -r --cached .  
git add .  
git commit -m "update .gitignore"
```

文件.gitignore的格式规范：

- #为注释
- 可以使用shell所使用的正则表达式来进行模式匹配
- 匹配模式最后跟"/"说明要忽略的是目录
- 使用！取反（例如目录中包含 test.a，并且gitignore文件中包含 *.oa]，如果在文件

中加入 `!test.a` 表明忽略除 `test.a` 文件以外的后缀名为 `.a` 或者 `.o` 的文件)

配置语法：

- 以斜杠“/”开头表示目录；
- 以星号“*”通配多个字符；
- 以问号“?”通配单个字符
- 以方括号“[]”包含单个字符的匹配列表；
- 以叹号“!”表示不忽略(跟踪)匹配到的文件或目录。

例子：

```
*.a # 忽略所有 .a 结尾的文件
!lib.a # 但 lib.a 除外
/TODO # 仅仅忽略项目根目录下的 TODO 文件，不包括 subdir/TODO
build/ # 忽略 build/ 目录下的所有文件
doc/*.txt # 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt
```

此外，git 对于 `.gitignore` 配置文件是按行从上到下进行规则匹配的，意味着如果前面的规则匹配的范围更大，则后面的规则将不会生效。

commit

如果commit注释写错了，只是想改一下注释，只需要：

```
git commit --amend
```

此时会进入默认vim编辑器，修改注释完毕后保存就好了。

Mac

常用快捷键

`command + shift + 4` 部分截屏

`control + space` 切换输入法

control + command + A QQ截屏

command + C 复制 command + option(alt) + V 剪切粘贴

新建文件

在终端输入：

```
touch filename
```

隐藏文件

查看隐藏文件：

```
ls -la
```

在访达中可见隐藏文件和.开头文件：

```
defaults write com.apple.Finder AppleShowAllFiles YES  
killall Finder
```

Markdown

Markdown 超链接

```
[name](url "hint")
```

在url后面输入 空格加双引号下的提示文字 则鼠标放在该超链接时即会显示该提示文字。

Markdown 空格

shift+space 可切换空格大小（全半角之区别？）（之后再按几个空格就有几个空格，若未这样做按多少个空格都之显示一个空格）。按两个空格即表示换行。

此法对MarkdownPad 2 编辑器有效，而VSCode无效？

Markdown 代码

``code`` 表示行内代码。

```
```  
code
```
```

表示行间代码。

Markdown 公式

在HTML Head编辑器导入相应文件。

基本与Latex语法一致，但\在Markdown中为转义字符，故行内公式 `$ A $` 变为 `\\(A \\)`。
（A代指公式）

用visual studio code编辑则 `$ $` 或 `$$ $$` 即可。

用VSCODE预览公式时没问题，但导出为pdf，html等时公式仍显示源码。解决办法有二，其一为在文档开头写入：

```
<script type="text/javascript" src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-MML_HTMLorMML">  
<script type="text/x-mathjax-config">  
    MathJax.Hub.Config({ tex2jax: {inlineMath: [['$', '$']}, messageStyle: "none" });  
</script>
```

其二为在VSCODE中安装 `Markdown+Math` 插件。

Markdown 图片

插入GitHub上的图片预览不能显示时，把图片链接地址中的 `blob` 换成 `raw` 即可！

Markdown 目录

用VSCODE编译，将光标置入待插入位置，按 `Ctrl+Shift+P`，在弹出框里输入 `ctoc` 即可。

LaTeX

VSCode 编译器

`Ctrl + Alt + B` 一次编译

`Ctrl + Alt + R` 选择recipe，此时才能显示目录、摘要等。

`Ctrl + Alt + J` 正向查找，即选中LaTeX代码后按此快捷键可定位到PDF中的对应文本。

参考：[使用VSCode编写LaTeX](#)

[LaTeX技巧932：如何配置Visual Studio Code作为LaTeX编辑器\[新版更新\]](#)

一些符号代码

`\pm` ±

`\equiv` ≡

`\approx` ≈

`\leq` ≤

`\leqslant` ≲

`\raggedright` 两端对齐

Others

paper writing

插入图片

一般插入eps或PDF格式图片，而将jpg、png等格式图片转化为eps格式可借助 `bmeps` 命令，在安装了Tex后已具有 `bmeps` 模块。在终端中采取如下命令即可（注意写对图片路径，Windows系统下在指定文件夹中按住Shift键再右键选择‘在此处打开powershell窗口’即可）：

```
bmeps -c *.png *.eps
```

cmd 脚本

Windows中的执行脚本后缀名为 `.bat`，不同于Linux系统中的 `.sh`。笔者经常需要借助cmd窗口ssh远程登录服务器，故写个脚本更为便捷。如新建个login.bat，在里面写入下行命令即可：

```
start cmd /c "ssh [username]@[IP-address]"
```

其中 `cmd /k` 表示cmd后面的命令执行完后不关闭窗口；而 `cmd /c` 表示执行完cmd命令后关闭窗口。

参考：[写一个打开cmd窗口并执行cmd命令的Windows脚本（.bat文件）](#)

USB失效

笔者有一天打开电脑（Windows系统）突然发现所有的USB接口都失效了，导致外接鼠标键盘都失效，插入U盘也读取不了。网上搜了很多教程都不管用，一类方法为电脑在节电模式下会禁用外接USB口等，所以需关闭这些设置，还有卸载USB驱动再重装等（可借助驱动精灵等软件），笔者试了均无果。

开机进入BIOS模式，在 `System Configuration` 里有一栏是 `USB Configuration`，勾选上 `Enable USB Boot Support` 和 `Enable External USB Ports`；另在 `Power Management` 里有一栏是 `Enable USB Wake Support`，勾选上 `ON`。此时开机后便一切恢复岁月静好了！

此次USB失效莫名其妙（系统更新的锅？），按照上述方法搞好后，过了两天又失效了，于是又重复搞了一次，后来就再也没有抽风过。

另很多电脑F1~12键都有另一种对应的功能，如F3除了传统的功能外还可增大音量，默认的是传统功能时要增大音量就得按住 `fn` 键再按F3了。在BIOS模式的 `POST Behavior` 里有一栏 `Fn lock` 的 `Lock Mode`，其有 `Lock Mode Standard` 和 `Lock Mode Secondary`，选择一个即可。笔者习惯选择后者，即F3默认为增大音量。

server config(~2020.12)

weiming1 :

python3.6.2

torch 0.4.0

torchvision 0.2.1

cv2 3.1.0

PKU163:

Python3.5.2

torch 0.4.1

torchvision 0.2.1

cv2 3.3.0

tensorflow 1.4.0