pyskl脱离了mmaction2，依赖于mmcv, mmdet和mmpose

第一步，新建pyskl环境

conda create -n pyskl python=3.8

第二步，添加pytorch==1.10.1 cuda=11.3

conda install pytorch==1.10.1 torchvision==0.11.2 torchaudio==0.10.1 cudatoolkit=11.3 -c pytorch -c conda-forge

第三步，pyskl中相关包安装 requirements.txt中修改为mmcv-full==1.5.0

cd pyskl

pip install -r requirements.txt

pip install -e .

第四步，安装mmpose和mmdet 切换到home/yl目录

git clone <https://github.com/open-mmlab/mmpose.git>

cd mmpose

pip install -r requirements.txt

pip install -v -e .

git clone <https://github.com/open-mmlab/mmdetection.git>

cd mmdetection

pip install -v -e .

pip install ftfy

pip install regex

pip install pandas

理解mmcv的四部分



我理解的Registry注册器

第一步：定义一个注册表CONVERTERS，这个名字可以diy，

CONVERTERS=Registry(‘converter’), 括号内的字符串不知道有没有什么要求，但是我知道可以是’dataset’ ‘pipeline’等，然后在这个文档里实现CONVERTERS的build函数

第二步：写具体类的实现

比如上图中Converter1类，然后在最前面加上@CONVERTERS.register\_module()

于是配置文件中的type = ‘Convert1’就对应了Converter1类

第三步：调用类并且实例化

先写配置文件type = ‘Converter1’ + 其余的参数

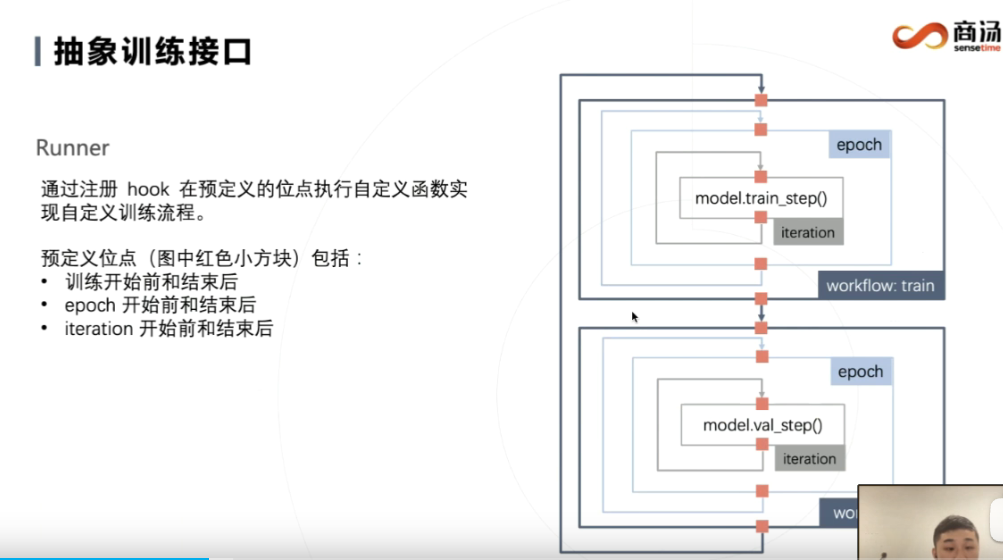
最后实例化为converter = build\_converter(converter\_cfg)

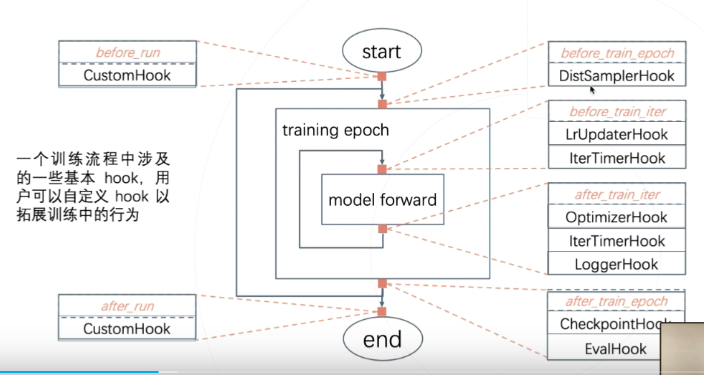
我的理解下面这些命令意思一致

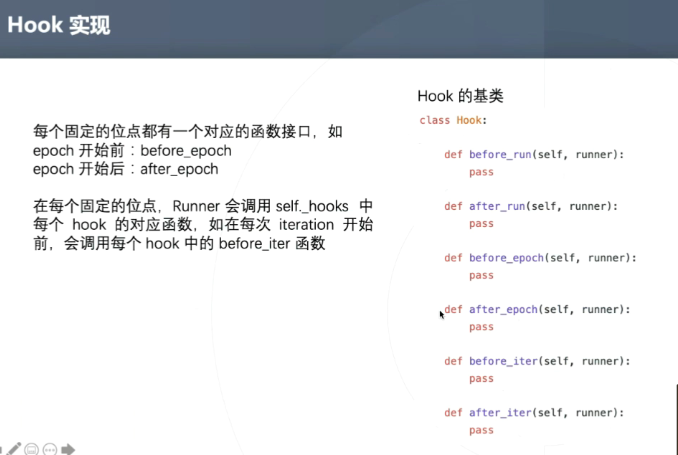
converter = CONVERTERS.build(converter\_cfg)

converter = build\_from\_cfg(converter\_cfg, CONVERTERS, default\_args)

converter = build\_converter(converter\_cfg)







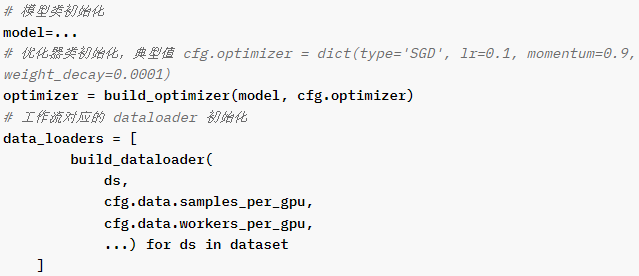
理解的执行器runner：有EpochBasedRunner和IterBasedRunner两种模式

EpochBasedRunner 是指以 epoch 为周期的工作流，例如设置 workflow = [(‘train’, 2), (‘val’, 1)] 表示循环迭代地训练 2 个 epoch，然后验证 1 个 epoch。

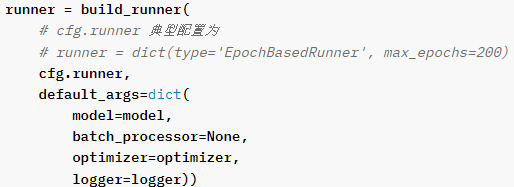
IterBasedRunner 是指以 iter 为周期的工作流，例如设置 workflow = [(‘train’, 2)， (‘val’, 1)] 表示循环迭代的训练 2 个 iter，然后验证 1 个 iter

一个训练任务的流程：

(1)dataloader, model 和优化器等类初始化



(2)runner类初始化



(3)注册默认训练所必须的hook和用户自定义hook



(4)开启训练流 runner.run(data\_loaders, cfg.workflow)

热力图形式的hmdb51数据集在pyskl上进行动作识别代码为例

理解各个模块如何进行调用，方便修改

(1)

train dataset初始化

cfg中data.train.type=’RepeatDataset’, 于是调用pyskl中dataset\_wrappers中的RepeatDataset类将原始的dataset复制times次，原始的data.train.dataset.type=’PoseDataset’，在pose\_dataset中的PoseDataset类进行数据集实例化

实例化过程中为什么dataset里有参数type=’PoseDataset’，因为他是用buid\_from\_cfg调用PoseDataset类，其中train\_pipeline是一个list，因为直接调用了Compose类，而Compose类中具体的transform实例化使用build\_from\_cfg调用，所以list中的每一个元素为dict，包含type=’pipeline类名’

PoseDataset类继承于BaseDataset类，abstractmethod方法load\_annotation必须在子类中进行实现

类中staticmethod装饰器的静态方法不需要传入self，同时该方法不能使用类变量和实例变量，不需要实例化类也可以调用该方法。在类内调用该方法时可以通过self.静态方法调用。

model初始化

cfg.model.type为Recognizer3D，于是build\_model函数中build\_recognizer中为RECOGNIZERS.build(cfg)，就调用了pyskl/models/recognizers/recognizer3d.py中的Recognizer3D类进行初始化模型。

Recognizer3D类的init过程中有backbone的初始化以及cls\_head的初始化

**backbone的初始化**采用BACKBONES.build(cfg),配置文件中cfg.model.backbone.type = ’ResNet3DSlowOnly’，于是调用pyskl/models/cnns/resnet3d\_slowonly.py中的ResNet3DSlowOnly类进行初始化。ResNet3dSlowOnly继承于ResNet3d，根据参数构建模型

一个batch的输入x\_shape [N, C, T, H, W]

self.conv1:

padding\_layer [N, C=17, T=48, H=48, W=48]

conv(

conv\_cfg = ‘Conv3d’, in\_channels=17, base\_channels = 32,

kernel\_size = conv1\_kernel = (1, 7, 7) resnet3dslowonly中的参数

stride = conv1\_stride = (1, 1, 1) cfg中的参数

) [N, C=32, T=32, H=48, W=48]

norm(Norm3d)

activate(ReLU)

self.maxpool

maxpool3d(

kernel\_size = (1, 3, 3),

stride = (1, 1, 1) cfg中的参数

padding = (0, 1, 1)

)

self.res\_layers[存储每一个res\_layer的layer\_name], res\_layer通过add\_module()添加到模块中，res\_layer通过make\_res\_layer()方法进行实例化3/4/6个res\_block, 也就是代码中的self.block变量的值，

调用make\_res\_layer()函数前确定每一个res\_layer的参数, expansion值均为4, 所有层的norm\_cfg为BN3d, act\_cfg = ReLU

res\_layer\_1:

self.inplanes = self.base\_channel = 32 该res\_layer的输入数据通道数

planes = self.base\_channel \* 2 \*\* 0 = 32 该res\_layer的输出通道数/expansion

num\_blocks = 3

stride = (temporal\_stride, spatial\_stride) = (1, 2)

inflate = 0

res\_layer\_2:

self.inplanes = 上层planes\* expansion = 128 输入通道数(上层的输出通道数)

planes = self.base\_channel \*2 \*\*1 = 64

num\_blocks = 4

stride = (temporal\_stride, spatial\_stride) = (1, 2)

inflate = 1

res\_layer\_3:

self.inplanes = 上层planes\* expansion = 256

planes = self.base\_channel \*2 \*\*2 = 256

num\_blocks = 6

stride = (temporal\_stride, spatial\_stride) = (2, 2)

inflate = 1

以res\_layer\_1为例，每一个block的内容

block1:

self.conv1 Conv3d

in\_ch = res\_layer1.inplanes = 32

out\_ch = res\_layer1.planes = 32

kernel\_size = (1,1,1) 因为inflate = 0

stride = (1,1,1) 定值

padding = 0 因为inflate = 0

然后接BN3d和ReLU

self.conv2 Conv3d

in\_ch = res\_layer1.planes = 32

out\_ch = res\_layer1.planes = 32

kernel\_size = (1, 3, 3) 因为inflate = 0

stride = (1, 2, 2) 每一个res\_layer，在block1的conv2中实现时空维度减小

padding = (0, 1, 1) 因为inflate = 0

然后接BN3d和ReLU

self.conv3 Conv3d

in\_ch = res\_layer1.planes = 32

out\_ch = res\_layer1.planes \* expansion = 128

kernel\_size = (1,1,1) 定值

只有BN3d没有ReLU

self.downsample Conv3d

in\_ch = res\_layer1.inplanes = 32

out\_ch = res\_layer1.planes \* expansion = 128

kernel\_size = (1,1,1) 定值

stride = (1, 2, 2) 与block保持一直

包含BN3d, 没有ReLU

前向过程输入x [B, C=32, T=32, H=56, W=56]

一路通过self.conv1, self.conv2, self.conv3 得到out [B, C=128, T=32, H=28, W=28]

一路通过self.downsample残差链接 得到out [B, C=128, T=32, H=28, W=28]

两路相加再通过ReLU得到最终输出

此时，其余block的inplanes均为res\_layer1.planes \* expansion，其余block与block1的相似和不同之处：

仍根据该res\_layer的inflate值确定conv1的卷积核尺寸

stride固定为(1, 1) 不进行时空的维度变换操作

由于数据维度不变换，不进行downsample维度变换操作，直接与该block输入进行残差连接

cls\_head的初始化通过HEADS.build(cfg)进行初始化，在cfg中cls\_head.type=’I3DHead'，调用pyskl.models.heads.simple\_head.py中的I3DHead类进行实例化

输入x [B, C=512, T=32, H=7, W=7]

pool = nn.AdaptiveAvgPool3d(1)

pool(x) [B, C=512, 1, 1, 1]

reshape [B, C=512]

dropout(0.5)

fc\_cls [B, C=51]

使用CLIP思想需要进行的修改

新的model类，包含原有的ResnetSlowOnly和新添加的text\_encoder

新的训练集和测试集前向过程

新的loss计算

tokenizer相关

方案：

新的Recognizer继承于BaseRecognizer

backbone需要包含posec3d的backbone ResnetSlowOnly于text\_encoder的backbone

新的cls\_head应该包含原有的I3Dhead和text\_encoder的cls\_head

在新的recognizer中forward\_train和forward\_test写对应的前向过程

步骤一：构建新的recognizer类RecognizerCLIP pyskl.models.recognizers.recognizerclip.py

在pyskl.models.recognizers.\_\_init\_\_.py中添加

from .recognizerclip import RecognizerCLIP

\_\_all\_\_中添加 ’RecognizerCLIP’

步骤二：构建新的backbone类BkCLIP pyskl.models.cnns.clipnet.py

在pyskl.models.cnns.\_\_init\_\_.py中添加

from .clipnet import BkCLIP

\_\_all\_\_中添加 ’BkCLIP’

步骤三：构建新的backbone类TextCLIP pyskl.models.cnns.clipnet.py

在pyskl.models.cnns.\_\_init\_\_.py中添加

from .clipnet import TextCLIP

\_\_all\_\_中添加 ’TextCLIP’

我想加载某一模块的预训练参数

在RecognizerCLIP这一recognizer类中有函数init\_weights，里面有self.backbone.init\_weights()和self.cls\_head.init\_weights()

而我的backbone类BkCLIP包含video\_encoder类和text\_encoder类，所以在BkCLIP中的init\_weights函数应该包含两种类的init\_weights()

在具体的video\_encoder与text\_encoder类中的init\_weights()中根据有无传入pretrain的路径选择何种初始化权重的方式

在pyskl.utils.text\_prompt.py中添加text\_prompt函数和tokenize类，用于构建文本tokenize，在pyskl.utils.\_\_init\_\_.py中添加from .text\_prompt import \*

在text\_prompt.py的同一层文件夹添加文件bpe\_simple\_vocab\_16e6.txt.gz

BkCLIP的前向过程输入就是img 和text tokenize, text\_prompt函数放置位置需要重新思考。输出为两者分别经过各自encoder的输出，之后的L2正则化等内容都未进行，放在cls\_head中，对应的一些参数也要放在cls\_head中

步骤四：构建新的cls\_head类CLIPHead pyskl.models.heads.clip\_head.py

在pyskl.models.heads.\_\_init\_\_.py中添加

from .clip\_head import CLIPHead

步骤五：构建新的losses类KLLoss

在pyskl.models.losses.\_\_init\_\_.py中添加

from .klloss import KLLoss

排查所有类的初始化函数是否添加了super().\_\_init\_\_()相关内容

RecognizerCLIP BKCLIP TextCLIP CLIPHead

如何能提升模型的性能

video\_encoder和text\_encoder采用不同的初始化学习率

改变学习率更新策略

dropout值改变

不利用text\_prompt，直接使用label的语义信息