模型轻量化的实现：

模型轻量化的目标是在保持模型性能的前提下，减少模型的参数量和计算量，使其更适合在资源受限的环境中运行，如移动设备和嵌入式系统。实现模型轻量化的方法主要包括：

模型剪枝（Pruning）：通过移除神经网络中不重要的连接（权重），可以是结构化的（如整个通道、卷积核的移除）或非结构化的（任意权重的移除）。

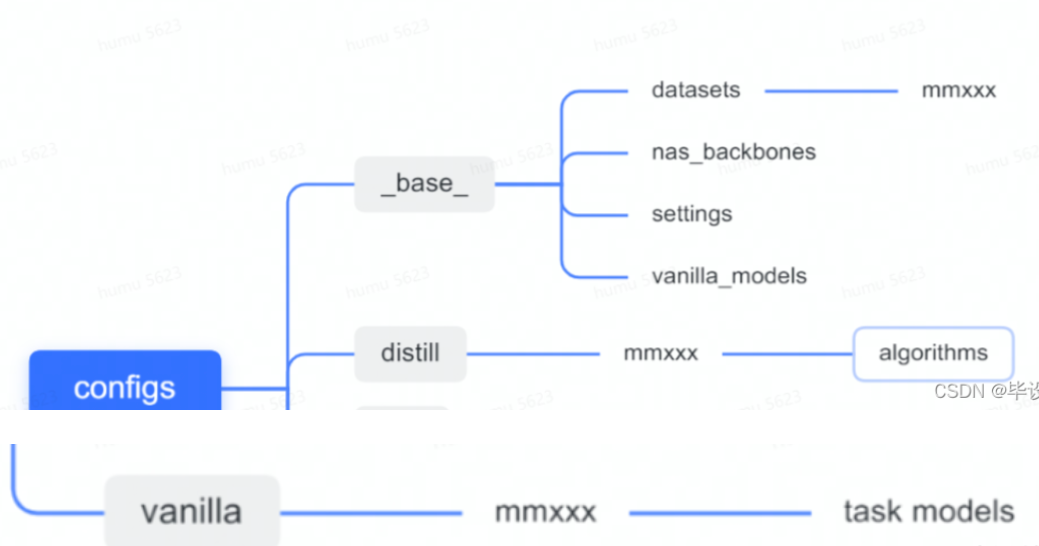
量化（Quantization）：将模型权重和激活的精度从浮点数降低到较低位宽的固定点或整数表示，减少模型大小，加速推理过程。

知识蒸馏（Knowledge Distillation）：大模型（教师模型）的知识迁移到小模型（学生模型）中，学生模型学习教师模型的输出作为软标签

模型的轻量化方法

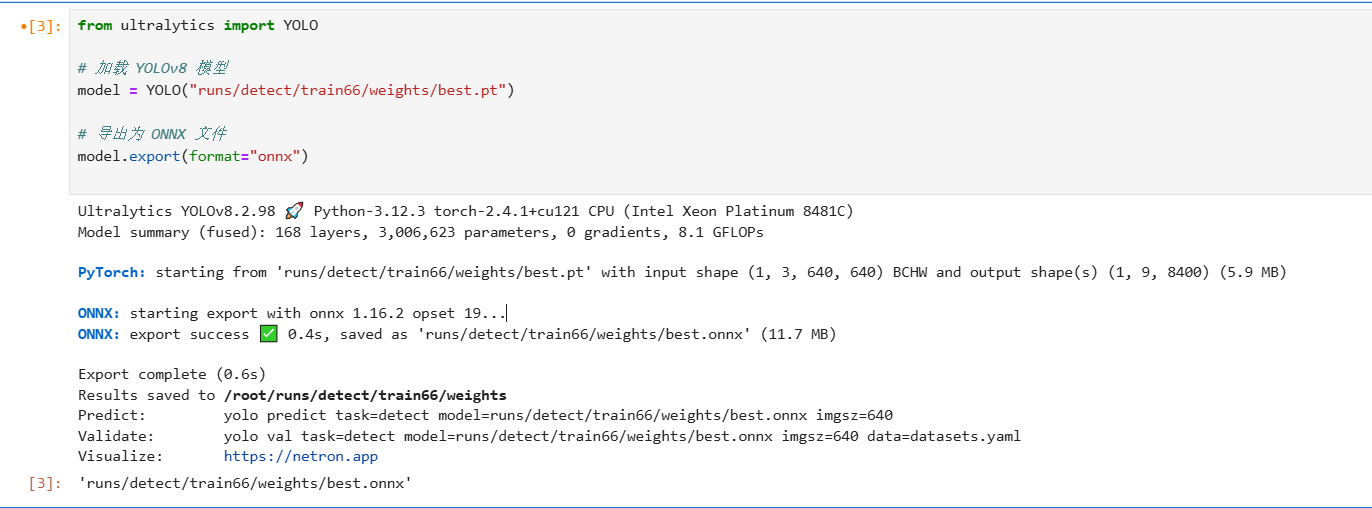
●考虑使用mmrazor进行知识蒸馏

Mmrazor中的config文件的配置：



背景介绍：[MMRazor](https://link.zhihu.com/?target=https://github.com/open-mmlab/mmrazor/tree/dev-1.x" \t "https://zhuanlan.zhihu.com/p/_blank) 是 OpenMMLab 生态中面向模型压缩的开源算法库，目前主要涵盖了网络结构搜索、知识蒸馏、剪枝三类算法，MMRazor 提供了基础的 backbone 用于构建常见的超网络，其大多由主流的 backbone 网络拓展而来，例如 [AttentiveMobileNetV3](https://link.zhihu.com/?target=https://github.com/open-mmlab/mmrazor/blob/dev-1.x/configs/_base_/nas_backbones/attentive_mobilenetv3_supernet.py" \t "https://zhuanlan.zhihu.com/p/_blank)、 [SearchableShuffleNetV2](https://link.zhihu.com/?target=https://github.com/open-mmlab/mmrazor/blob/dev-1.x/configs/_base_/nas_backbones/spos_shufflenet_supernet.py" \t "https://zhuanlan.zhihu.com/p/_blank) 等。在配置文件中通过指定 yaml 的模型配置文件进而可以加载超网络中的任意子网。

将网上的模型由pt格式向onnx格式进行转换



由于训练时使用的resnet文件，故mmrazor的配置文件如下：

base\_ = [

'mmcls::\_base\_/datasets/imagenet\_bs32.py',

'mmcls::\_base\_/schedules/imagenet\_bs256.py',

'mmcls::\_base\_/default\_runtime.py'

]

stage\_ratio\_1 = 0.65

stage\_ratio\_2 = 0.6

stage\_ratio\_3 = 0.9

stage\_ratio\_4 = 0.7

# the config template of target\_pruning\_ratio can be got by

# python ./tools/pruning/get\_channel\_units.py {config\_file} --choice

target\_pruning\_ratio = {

'backbone.layer1.0.conv1\_(0, 64)\_64': stage\_ratio\_1,

'backbone.layer1.0.conv2\_(0, 64)\_64': stage\_ratio\_2,

'backbone.layer1.0.conv3\_(0, 256)\_256': stage\_ratio\_3,

'backbone.layer1.1.conv1\_(0, 64)\_64': stage\_ratio\_1,

'backbone.layer1.1.conv2\_(0, 64)\_64': stage\_ratio\_2,

'backbone.layer1.2.conv1\_(0, 64)\_64': stage\_ratio\_1,

'backbone.layer1.2.conv2\_(0, 64)\_64': stage\_ratio\_2,

# block 1 [0.65, 0.6] downsample=[0.9]

'backbone.layer2.0.conv1\_(0, 128)\_128': stage\_ratio\_1,

'backbone.layer2.0.conv2\_(0, 128)\_128': stage\_ratio\_2,

'backbone.layer2.0.conv3\_(0, 512)\_512': stage\_ratio\_3,

'backbone.layer2.1.conv1\_(0, 128)\_128': stage\_ratio\_1,

'backbone.layer2.1.conv2\_(0, 128)\_128': stage\_ratio\_2,

'backbone.layer2.2.conv1\_(0, 128)\_128': stage\_ratio\_1,

'backbone.layer2.2.conv2\_(0, 128)\_128': stage\_ratio\_2,

'backbone.layer2.3.conv1\_(0, 128)\_128': stage\_ratio\_1,

'backbone.layer2.3.conv2\_(0, 128)\_128': stage\_ratio\_2,

# block 2 [0.65, 0.6] downsample=[0.9]

'backbone.layer3.0.conv1\_(0, 256)\_256': stage\_ratio\_1,

'backbone.layer3.0.conv2\_(0, 256)\_256': stage\_ratio\_2,

'backbone.layer3.0.conv3\_(0, 1024)\_1024': stage\_ratio\_3,

'backbone.layer3.1.conv1\_(0, 256)\_256': stage\_ratio\_1,

'backbone.layer3.1.conv2\_(0, 256)\_256': stage\_ratio\_2,

'backbone.layer3.2.conv1\_(0, 256)\_256': stage\_ratio\_1,

'backbone.layer3.2.conv2\_(0, 256)\_256': stage\_ratio\_2,

'backbone.layer3.3.conv1\_(0, 256)\_256': stage\_ratio\_4,

'backbone.layer3.3.conv2\_(0, 256)\_256': stage\_ratio\_4,

'backbone.layer3.4.conv1\_(0, 256)\_256': stage\_ratio\_4,

'backbone.layer3.4.conv2\_(0, 256)\_256': stage\_ratio\_4,

'backbone.layer3.5.conv1\_(0, 256)\_256': stage\_ratio\_4,

'backbone.layer3.5.conv2\_(0, 256)\_256': stage\_ratio\_4,

# block 3 [0.65, 0.6]\*2+[0.7, 0.7]\*2 downsample=[0.9]

'backbone.layer4.0.conv1\_(0, 512)\_512': stage\_ratio\_4,

'backbone.layer4.0.conv2\_(0, 512)\_512': stage\_ratio\_4,

'backbone.layer4.0.conv3\_(0, 2048)\_2048': stage\_ratio\_3,

'backbone.layer4.1.conv1\_(0, 512)\_512': stage\_ratio\_4,

'backbone.layer4.1.conv2\_(0, 512)\_512': stage\_ratio\_4,

'backbone.layer4.2.conv1\_(0, 512)\_512': stage\_ratio\_4,

'backbone.layer4.2.conv2\_(0, 512)\_512': stage\_ratio\_4

# block 4 [0.7, 0.7] downsample=[0.9]

}

optim\_wrapper = dict(

optimizer=dict(type='SGD', lr=0.1, momentum=0.9, weight\_decay=0.0001))

param\_scheduler = dict(

type='MultiStepLR', by\_epoch=True, milestones=[30, 60, 90], gamma=0.1)

train\_cfg = dict(by\_epoch=True, max\_epochs=120, val\_interval=1)

data\_preprocessor = {'type': 'mmcls.ClsDataPreprocessor'}

# model settings

model = dict(

\_scope\_='mmrazor',

type='DCFF',

architecture=dict(

cfg\_path='mmcls::resnet/resnet50\_8xb32\_in1k.py', pretrained=False),

mutator\_cfg=dict(

type='DCFFChannelMutator',

channel\_unit\_cfg=dict(

type='DCFFChannelUnit', default\_args=dict(choice\_mode='ratio')),

parse\_cfg=dict(

type='ChannelAnalyzer',

demo\_input=(1, 3, 224, 224),

tracer\_type='BackwardTracer')),

data\_preprocessor=None,

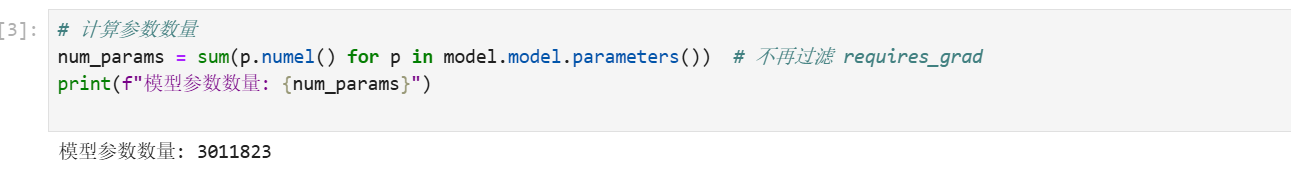
target\_pruning\_ratio=target\_pruning\_ratio,

step\_freq=1,

linear\_schedule=False)

val\_cfg = dict(\_delete\_=True, type='mmrazor.ItePruneValLoop')

计算模型的大致参量：



使用PPQ对于模型进行量化：

PPQ的简介：PPQ（Post-training Quantization）是一个用于深度学习模型量化的工具，旨在提高模型在嵌入式设备和边缘计算平台上的推理速度和效率。它主要通过在模型训练完成后，减少模型的参数精度（如从浮点数到整数）来减小模型的存储和计算开销。

主要特点：

易用性：PPQ 提供简化的 API，使用户能够轻松集成到现有工作流中。

兼容性：支持多种深度学习框架（如 PyTorch、TensorFlow 等），方便用户应用于不同的项目。

灵活性：可以针对不同的硬件平台进行优化，满足特定应用的需求。

性能优化：通过量化，减少模型的计算量和内存占用，提高推理速度。

使用ppq对于模型进行量化的代码

from ppq import \*

from ppq.api import \*

import os

# modify configuration below:

WORKING\_DIRECTORY = 'working' # choose your working directory

TARGET\_PLATFORM = TargetPlatform.PPL\_CUDA\_INT8 # choose your target platform

MODEL\_TYPE = NetworkFramework.ONNX # or NetworkFramework.CAFFE

INPUT\_LAYOUT = 'chw' # input data layout, chw or hwc

NETWORK\_INPUTSHAPE = [1, 3, 640, 640] # input shape of your network

CALIBRATION\_BATCHSIZE = 16 # batchsize of calibration dataset

EXECUTING\_DEVICE = 'cuda' # 'cuda' or 'cpu'.

REQUIRE\_ANALYSE = False

TRAINING\_YOUR\_NETWORK = True # 是否需要 Finetuning 一下你的网络

# -------------------------------------------------------------------

# 加载你的模型文件，PPQ 将会把 onnx 或者 caffe 模型文件解析成自己的格式

# 如果你正使用 pytorch, tensorflow 等框架，你可以先将模型导出成 onnx

# 使用 torch.onnx.export 即可，如果你在导出 torch 模型时发生错误，欢迎与我们联系。

# -------------------------------------------------------------------

graph = None

if MODEL\_TYPE == NetworkFramework.ONNX:

graph = load\_onnx\_graph(onnx\_import\_file = os.path.join(WORKING\_DIRECTORY, 'model.onnx'))

if MODEL\_TYPE == NetworkFramework.CAFFE:

graph = load\_caffe\_graph(

caffemodel\_path = os.path.join(WORKING\_DIRECTORY, 'model.caffemodel'),

prototxt\_path = os.path.join(WORKING\_DIRECTORY, 'model.prototxt'))

assert graph is not None, 'Graph Loading Error, Check your input again.'

# -------------------------------------------------------------------

# SETTING 对象用于控制 PPQ 的量化逻辑，主要描述了图融合逻辑、调度方案、量化细节策略等

# 当你的网络量化误差过高时，你需要修改 SETTING 对象中的属性来进行特定的优化

# -------------------------------------------------------------------

QS = QuantizationSettingFactory.default\_setting()

# -------------------------------------------------------------------

# 下面向你展示了如何使用 finetuning 过程提升量化精度

# 在 PPQ 中我们提供了十余种算法用来帮助你恢复精度

# 开启他们的方式都是 QS.xxxx = True

# 按需使用，不要全部打开，容易起飞

# -------------------------------------------------------------------

if TRAINING\_YOUR\_NETWORK:

QS.lsq\_optimization = True # 启动网络再训练过程，降低量化误差

QS.lsq\_optimization\_setting.steps = 500 # 再训练步数，影响训练时间，500 步大概几分钟

QS.lsq\_optimization\_setting.collecting\_device = 'cuda' # 缓存数据放在那，cuda 就是放在gpu，如果显存超了你就换成 'cpu'

# -------------------------------------------------------------------

# 你可以把量化很糟糕的算子送回 FP32

# 当然你要先确认你的硬件支持 fp32 的执行

# 你可以使用 layerwise\_error\_analyse 来找出那些算子量化的很糟糕

# -------------------------------------------------------------------

QS.dispatching\_table.append(operation='OP NAME', platform=TargetPlatform.FP32)

print('正准备量化你的网络，检查下列设置:')

print(f'WORKING DIRECTORY : {WORKING\_DIRECTORY}')

print(f'TARGET PLATFORM : {TARGET\_PLATFORM.name}')

print(f'NETWORK INPUTSHAPE : {NETWORK\_INPUTSHAPE}')

print(f'CALIBRATION BATCHSIZE: {CALIBRATION\_BATCHSIZE}')

# -------------------------------------------------------------------

# load\_calibration\_dataset 函数针对的是单输入模型，输入数据必须是图像数据 layout : [n c h w]

# 如果你的模型具有更复杂的输入格式，你可以重写下面的 load\_calibration\_dataset 函数

# 请注意，任何可遍历对象都可以作为 ppq 的数据集作为输入

# 比如下面这个 dataloader = [torch.zeros(size=[1,3,224,224]) for \_ in range(32)]

# 当前这个函数的数据将从 WORKING\_DIRECTORY/data 文件夹中进行数据加载

#

# 如果你的数据不在这里

# 你同样需要自己写一个 load\_calibration\_dataset 函数

# -------------------------------------------------------------------

dataloader = load\_calibration\_dataset(

directory = WORKING\_DIRECTORY,

input\_shape = NETWORK\_INPUTSHAPE,

batchsize = CALIBRATION\_BATCHSIZE,

input\_format = INPUT\_LAYOUT)

# ENABLE CUDA KERNEL 会加速量化效率 3x ~ 10x，但是你如果没有装相应编译环境的话是编译不了的

# 你可以尝试安装编译环境，或者在不启动 CUDA KERNEL 的情况下完成量化：移除 with ENABLE\_CUDA\_KERNEL(): 即可

with ENABLE\_CUDA\_KERNEL():

print('网络正量化中，根据你的量化配置，这将需要一段时间:')

quantized = quantize\_native\_model(

setting=QS, # setting 对象用来控制标准量化逻辑

model=graph,

calib\_dataloader=dataloader,

calib\_steps=32,

input\_shape=NETWORK\_INPUTSHAPE, # 如果你的网络只有一个输入，使用这个参数传参

inputs=None, # 如果你的网络有多个输入，使用这个参数传参，就是 input\_shape=None, inputs=[torch.zeros(1,3,224,224), torch.zeros(1,3,224,224)]

collate\_fn=lambda x: x.to(EXECUTING\_DEVICE), # collate\_fn 跟 torch dataloader 的 collate fn 是一样的，用于数据预处理，

# 你当然也可以用 torch dataloader 的那个，然后设置这个为 None

platform=TARGET\_PLATFORM,

device=EXECUTING\_DEVICE,

do\_quantize=True)

# -------------------------------------------------------------------

# 如果你需要执行量化后的神经网络并得到结果，则需要创建一个 executor

# 这个 executor 的行为和 torch.Module 是类似的，你可以利用这个东西来获取执行结果

# 请注意，必须在 export 之前执行此操作。

# -------------------------------------------------------------------

executor = TorchExecutor(graph=quantized, device=EXECUTING\_DEVICE)

# output = executor.forward(input)

# -------------------------------------------------------------------

# PPQ 计算量化误差时，使用信噪比的倒数作为指标，即噪声能量 / 信号能量

# 量化误差 0.1 表示在整体信号中，量化噪声的能量约为 10%

# 你应当注意，在 graphwise\_error\_analyse 分析中，我们衡量的是累计误差

# 网络的最后一层往往都具有较大的累计误差，这些误差是其前面的所有层所共同造成的

# 你需要使用 layerwise\_error\_analyse 逐层分析误差的来源

# -------------------------------------------------------------------

print('正计算网络量化误差(SNR)，最后一层的误差应小于 0.1 以保证量化精度:')

reports = graphwise\_error\_analyse(

graph=quantized, running\_device=EXECUTING\_DEVICE, steps=32,

dataloader=dataloader, collate\_fn=lambda x: x.to(EXECUTING\_DEVICE))

for op, snr in reports.items():

if snr > 0.1: ppq\_warning(f'层 {op} 的累计量化误差显著，请考虑进行优化')

if REQUIRE\_ANALYSE:

print('正计算逐层量化误差(SNR)，每一层的独立量化误差应小于 0.1 以保证量化精度:')

layerwise\_error\_analyse(graph=quantized, running\_device=EXECUTING\_DEVICE,

interested\_outputs=None,

dataloader=dataloader, collate\_fn=lambda x: x.to(EXECUTING\_DEVICE))

# -------------------------------------------------------------------

# 使用 export\_ppq\_graph 函数来导出量化后的模型

# PPQ 会根据你所选择的导出平台来修改模型格式

# -------------------------------------------------------------------

print('网络量化结束，正在生成目标文件:')

export\_ppq\_graph(

graph=quantized, platform=TARGET\_PLATFORM,

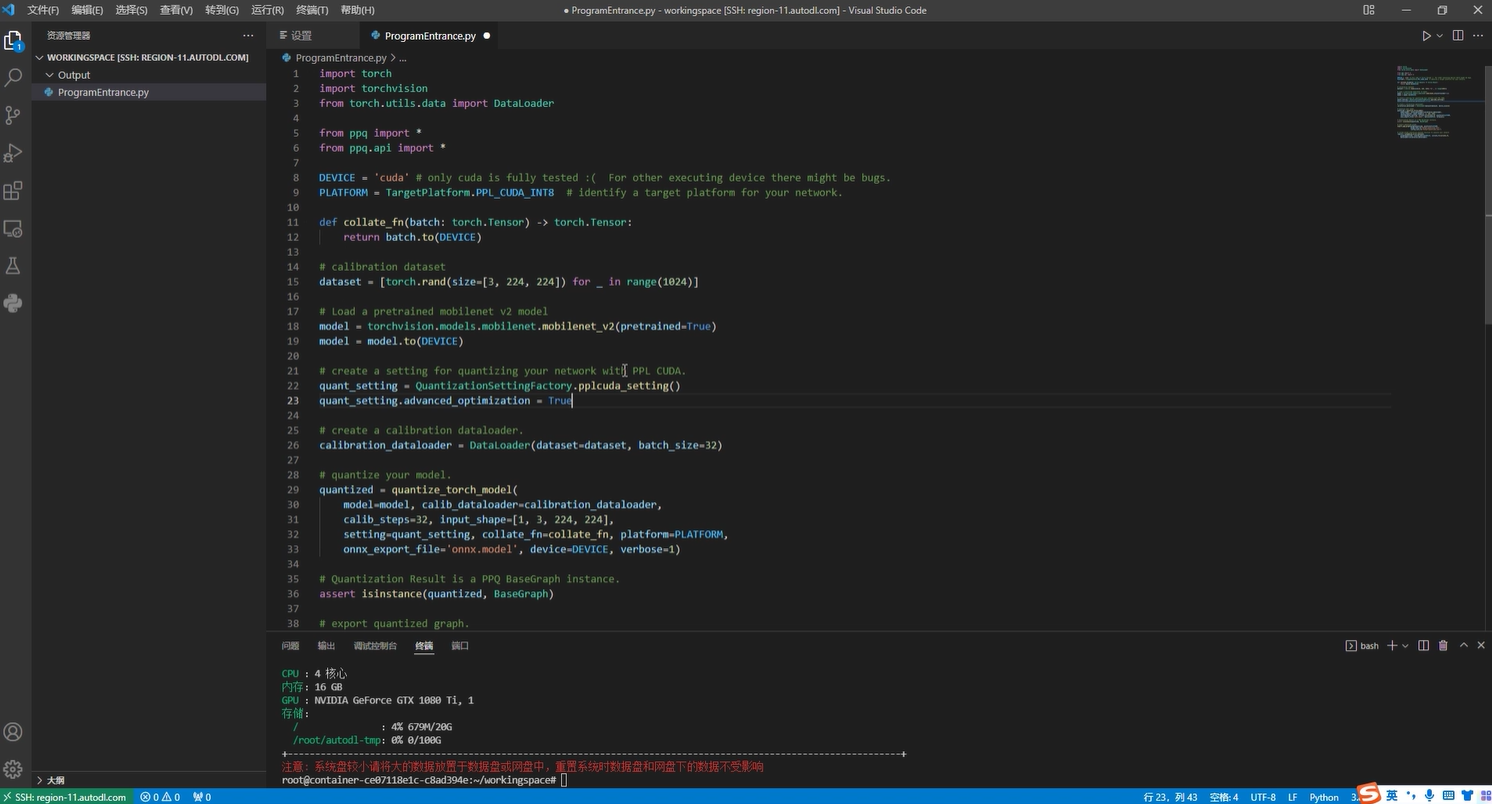
graph\_save\_to = os.path.join(WORKING\_DIRECTORY, 'quantized.onnx'),

config\_save\_to = os.path.join(WORKING\_DIRECTORY, 'quant\_cfg.json'))

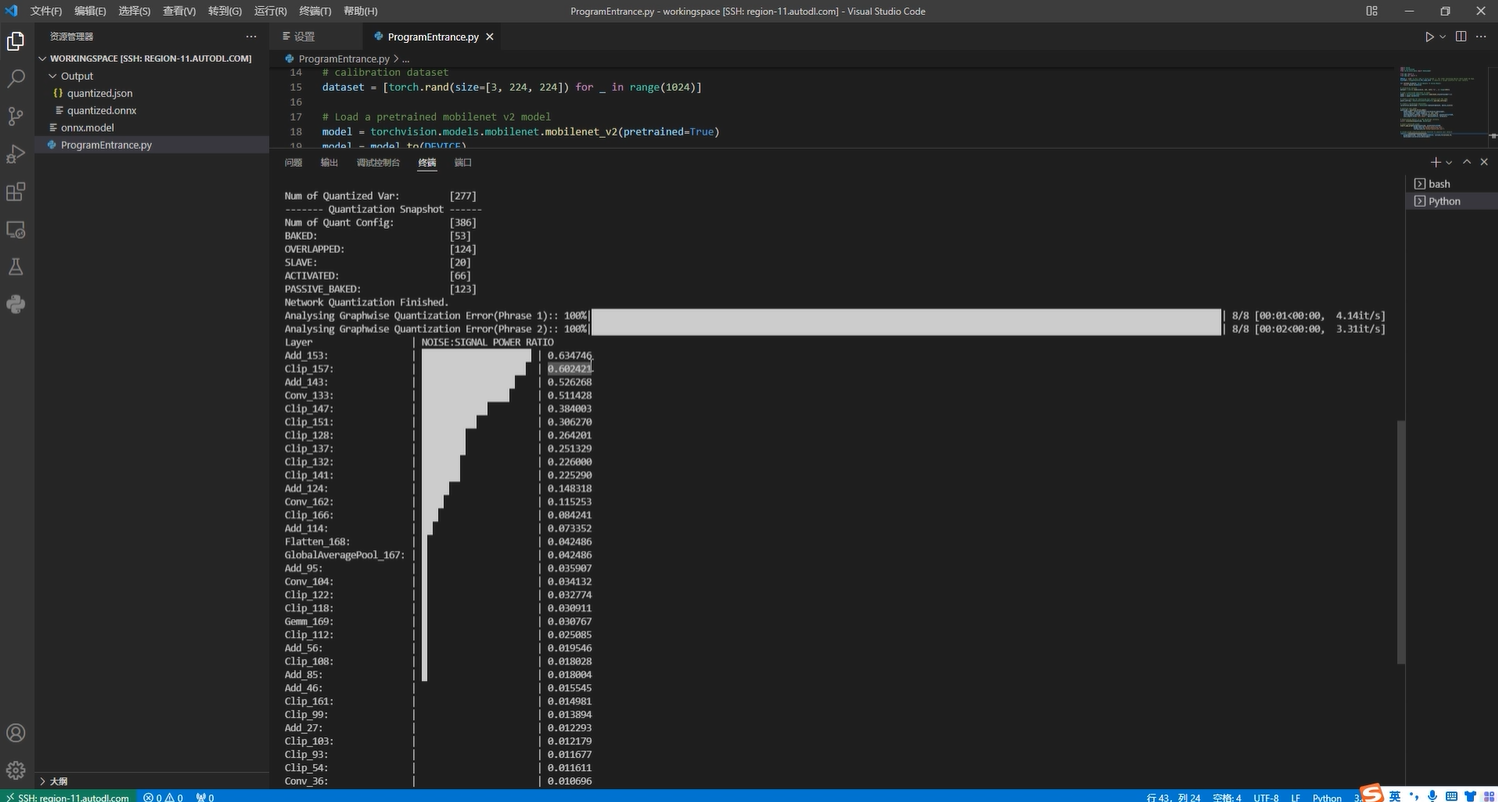
其中使用特定的参数来输入模型的有关信息



●用vscode远程连接到云服务器并启动代码



●启动代码进行量化：



从output文件夹中导出新的onnx文件