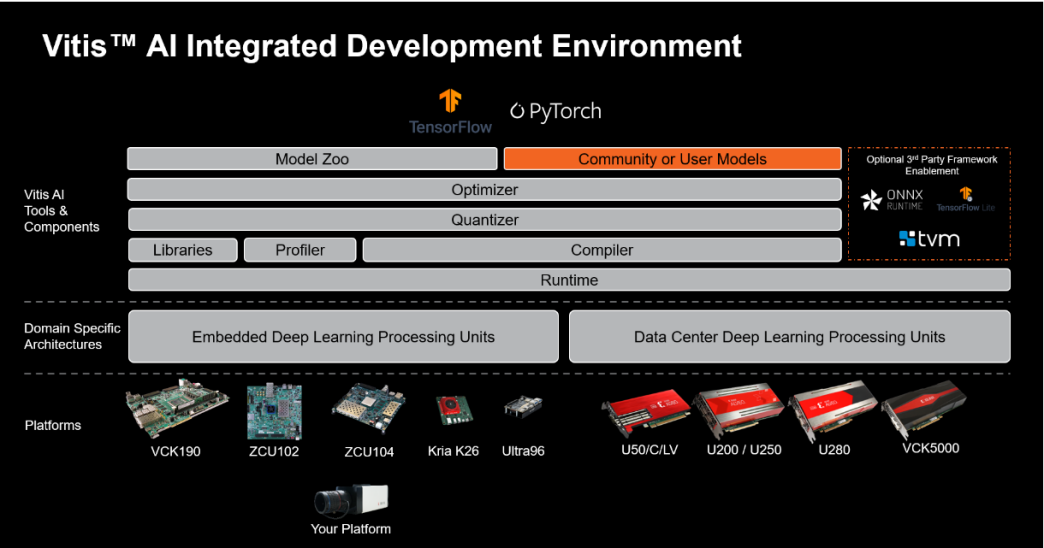
# 1、Vitis AI 工具概述

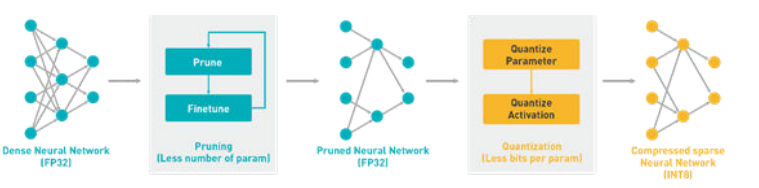
深度学习处理器 (DPU) 是一个专为深度神经网络而优化的可编程引擎。它由一组可参数化的 IP 核组成，这些 IP 核在硬 件上预实现，且无需布局布线。其设计主旨是为了帮助各种计算机视觉应用中广泛采用的深度学习推断算法实现计算工 作负载加速，适合的应用包括图像/视频分类、语义分段以及目标检测/追踪。DPU 随 Vitis AI 专用指令集一起发布，从 而促进深度学习网络的有效实现。 高效的张量层指令集旨在支持并加速各种常用的卷积神经网络，例如，VGG、ResNet、GoogLeNet、YOLO、SSD 和 MobileNet 等。DPU 可轻松缩放以适应从边缘到云端的各种赛灵思 Zynq UltraScale+ MPSoC、赛灵思 Kria KV260、 Versal 卡和 Alveo 开发板，满足多种多样应用的不同需求。 Vitis 流程中会生成配置文件 arch.json。arch.json 文件可供 Vitis AI 编译器用于模型编译。一旦 DPU 配置发生修 改，就必须生成新的 arch.json。并且必须使用新的 arch.json 文件重新生成模型。例如，在 Vitis 流程的 DPUCZDX8G 的 ZCU102 TRD 中，arch.json 文件位于 $TRD\_HOME/prj/Vitis/binary\_container\_1/ link/vivado/vpl/prj/prj.gen/sources\_1/bd/xilinx\_zcu102\_base/ip/ xilinx\_zcu102\_base\_DPUCZDX8G\_1\_0/arch.json。 Vitis AI 能够为赛灵思 Zynq® UltraScale+™ MPSoC、Kria KV260、Versal 卡等嵌入式器件以及 Alveo 卡（如 U50LV、 U200、U250 和 U55C）提供一系列不同的 DPU，从而在吞吐量、时延、可缩放性以及功耗方面实现独特的差异性和 灵活性。



# 2量化模型

## 2.1 概述

推断是一个计算密集型流程，需要大量存储器带宽以满足边缘 (Edge) 应用的低时延和高吞吐量要求。 通过采用量化和通道剪枝技巧可解决这些问题，同时实现高性能和高能效，并最大程度减少精度劣化。量化可使用整数 计算单元并按较低的位数来呈现权重和激活，同时剪枝可以减少所需的运算总量。下图为剪枝和量化流程。

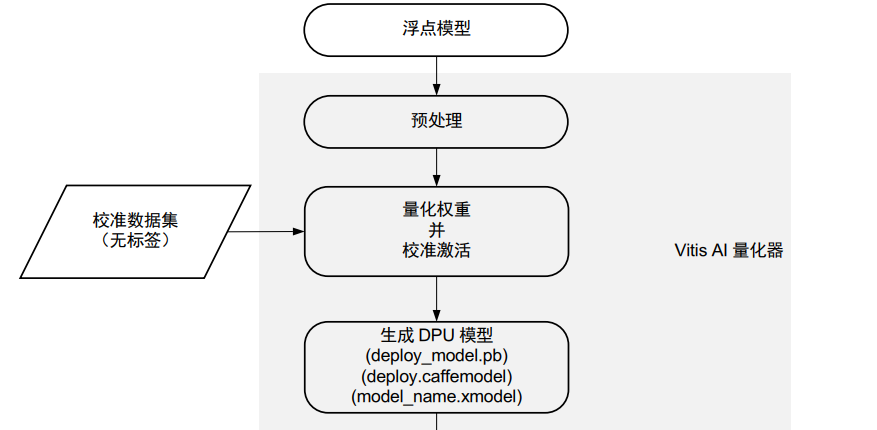


通常，进行神经网络训练时，使用的是 32 位浮点权重和激活值。通过将 32 位浮点权重和激活转换为 8 位整数 (INT8) 格式，Vitis AI 量化器即可降低计算复杂性，而不会损失预测精度。定点网络模型所需存储器带宽较少，因此相比浮点 模型，速度更快且能效更高。Vitis AI 量化器支持神经网络中存在公用层，包括但不限于卷积、池化、完全连接和批量 归一。 Vitis AI 量化器当前支持 TensorFlow（1.x 和 2.x）以及 PyTorch。量化器名称分别为 vai\_q\_tensorflow 和 vai\_q\_pytorch。

而我们本次使用的yolo模型为Pytorch框架，所以下面将详细讲述如何对Pytorch框架的神经网络进行量化。

## 2.2 Vitis AI 量化器流程

下图中提供了总体模型量化流程的详细信息。



Vitis AI 量化器使用浮点模型作为输入并执行预处理（折叠 batchnorm 并移除推断不需要的节点），然后将权重/偏差和激活量化为给定位宽。量化浮点模型前，有一个名为“检查器”的可选步骤。此步骤用于在量化模型前对其进行检查。检查器将输出分区信息，以指示将在相应器件 (DPU/CPU) 上运行的各项运算符。总体上，DPU 比 CPU 更快。

为采集激活统计数据并提升量化模型精度，Vitis AI量化器必须运行多次推断迭代，以对激活进行校准。因此，校准图像数据集输入是必需的。通常，量化器能够妥善处理 100 - 1000 张校准图像。由于无需反向传输，因此无标签数据集 足矣。校准后，量化模型被转换为 DPU 可部署模型，该模型针对 vai\_q\_tensorflow 名为 deploy\_model.pb，针对 vai\_q\_pytorch 名为 model\_name.xmodel，针对 vai\_q\_caffe 则名为 deploy.prototxt / deploy.caffemodel， 并且遵循 DPU 数据格式。随后，Vitis AI 编译器将对该模型进行编译，并将其部署到 DPU。

我们使用的框架为Pytorch，所以我们的目标是生成xmodel可执行文件。

## 2.3 Pytorch 量化

### 2.3.1 安装vai\_q\_pytorch

有两种方法可以安装：

1. 使用 Docker 容器安装。

Vitis AI 为量化工具（包括 vai\_q\_pytorch）提供了 Docker 容器。运行 GPU/CPU 容器后，激活 Conda 环境 vitis-aipytorch。

1. conda activate vitis-ai-pytorch

需要注意，运行该脚本前，必须检查 /opt/vitis\_ai/scripts/replace\_pytorch.sh脚本中的 Python、PyTorch 和 cuda-toolkit 版本，并根据需要对其版本进行编辑。选择 PyTorch 版本并编辑命令行时，需遵循 PyTorch 官方网站上的 指示信息进行操作。

1. 从源代码安装

1）在 .bashrc 中添加 CUDA\_HOME 环境变量。

export CUDA\_HOME=/usr/local/cuda

2）安装 PyTorch (1.2-1.10.2) 和 Torchvision。

1. pip install torch==1.7.1.0 torchvision==0.8.2

3）安装其它依赖项。

1. pip install -r requirements.txt

4）安装 vai\_q\_pytorch。

1. cd ./pytorch\_binding
2. python setup.py install (for user)
3. python setup.py develop (for developer)

5）验证安装。

1. python -c "import pytorch\_nndct"

### 2.3.2 量化前检查浮点模型

Vai\_q\_pytorch 提供了一个名为 inspector 的函数，可以诊断不同器件架构下的神经网络 (NN) 模型。检查器可以基于硬件约束来预测目标器件分配。生成的检查报告可用于对 NN 模型进行修改或最优化，从而显著降低部署难度并缩短部署时间。

1. 导入 vai\_q\_pytorch 模块

from pytorch\_nndct.apis import Inspector

1. 创建含目标名称或指纹（指纹指的是DPU相对应的型号）的检查器

inspector = Inspector("0x603000b16013831") *# by target fingerprint*

or

inspector = Inspector("DPUCAHX8L\_ISA0\_SP") *# by target name*

1. 检查浮点模型

input = torch.randn([batch\_size, 3, 224, 224])

inspector.inspect(model, input)

1. 这里以resnet18模型举例，运行以下命令行以检查模型：

python resnet18\_quant.py --quant\_mode float --inspect

1. 可以看到检查器成功运行，并且在输出目录“./quantize\_result”下会生成 3 个文件。

inspect\_{target}.txt: Target information and all the details of operations

in float model

inspect\_{target}.svg: If image\_format is not None. A visualization of

inspection result is generated

inspect\_{target}.gv: If image\_format is not None. Dot source code of

inspetion result is generated

### 2.3.3 运行vai\_q\_pytorch

1）首先在运行前，要准备三个文件：

|  |  |  |
| --- | --- | --- |
| 编号 | 名称 | 描述 |
| 1 | model.pth | 经预训练 PyTorch 模型，通常为 PTH 文件。 |
| 2 | model.py | 包含浮点模型定义的 Python 脚本。 |
| 3 | 校准数据集 | 训练数据集，包含 100 到 1000 张图像。 |

2）修改模型定义

要使 PyTorch 模型可量化，需要修改模型定义，以确保修改后的模型满足以下条件。

a. 要量化的模型应仅含前传方法。所有其它函数都应移出或者迁移至派生的类。这些函数通常作为预处理和后处理来工作。如果不将其移出，那么此 API 会在量化模块中将其移除，这样会在前传量化模块时导致异常行为。

b. 这样浮点模型应可通过 jit 追踪测试。将浮点模块设置为评估状态，然后使用 torch.jit.trace 函数来测试浮点模型。

我们使用的yolov5模型，需要将特征提取中前传函数等多于部分去掉，修改后代码如下：

1. z = []
2. for i in range(nl):
3. bs, \_, ny, nx, \_no= x[i].shape
4. *# x[i] = x[i].view(bs, na, no, ny, nx).permute(0, 1, 3, 4, 2).contiguous()*
5. if grid[i].shape[2:4] != x[i].shape[2:4]:
6. grid[i], anchor\_grid[i] = \_make\_grid(anchors,stride,nx, ny, i)

3）将 vai\_q\_pytorch API 添加至浮点脚本

如果在量化前，已有经过训练的浮点模型和 Python 脚本用于对模型精度/mAP 进行求值，那么量化器 API 会将浮点模块替换为量化模块。常规求值函数鼓励量化模块前传。如果 quant\_mode 标志设为“calib”，量化校准可在求值过 程中判定张量的量化步骤。校准后，将 quant\_mode 设置为“test”以对量化模型进行求值。

a.导入 vai\_q\_pytorch 模块。

from pytorch\_nndct.apis import torch\_quantizer, dump\_xmodel

b. 以需要输入的量化模型来生成量化器，并获取转换后的模型。

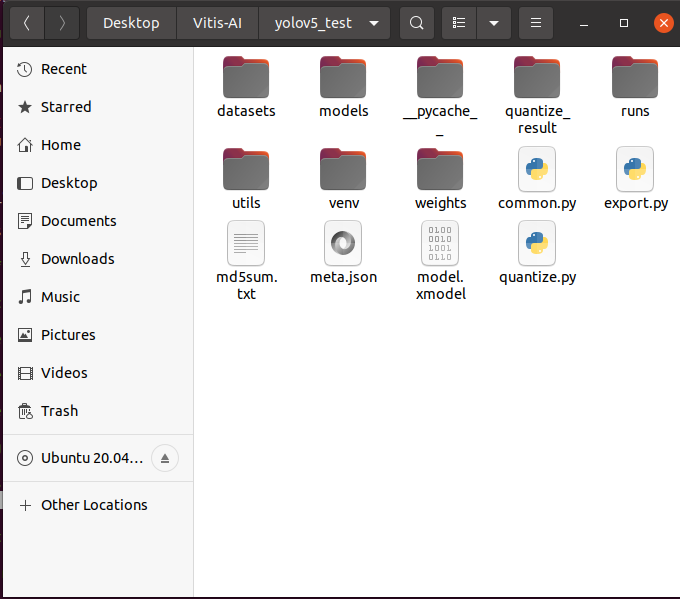
1. input = torch.randn([batch\_size, 3, 224, 224])
2. quantizer = torch\_quantizer(quant\_mode, model, (input))
3. quant\_model = quantizer.quant\_model

c. 使用转换后的模型前传神经网络。

1. acc1\_gen, acc5\_gen, loss\_gen = evaluate(quant\_model, val\_loader, loss\_fn)
2. d. if quant\_mode == 'calib':
3. quantizer.export\_quant\_config()
4. if deploy:
5. quantizer.export\_xmodel()
6. quantizer.export\_onnx\_model()

4）运行量化并获取结果

首先来看一下整体的文件目录结构。其中quantize.py即为量化脚本，数据集和label存放在datasets中，运行结果会生成在runs目录下。

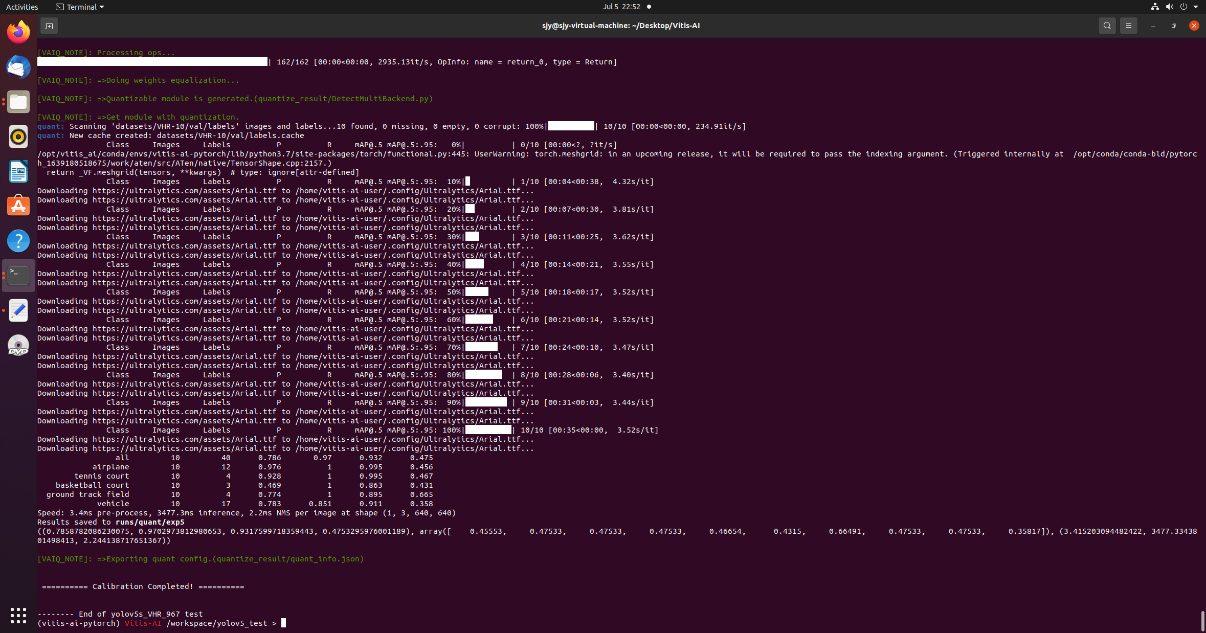


a.运行含“--quant\_mode calib”的命令以量化模型。

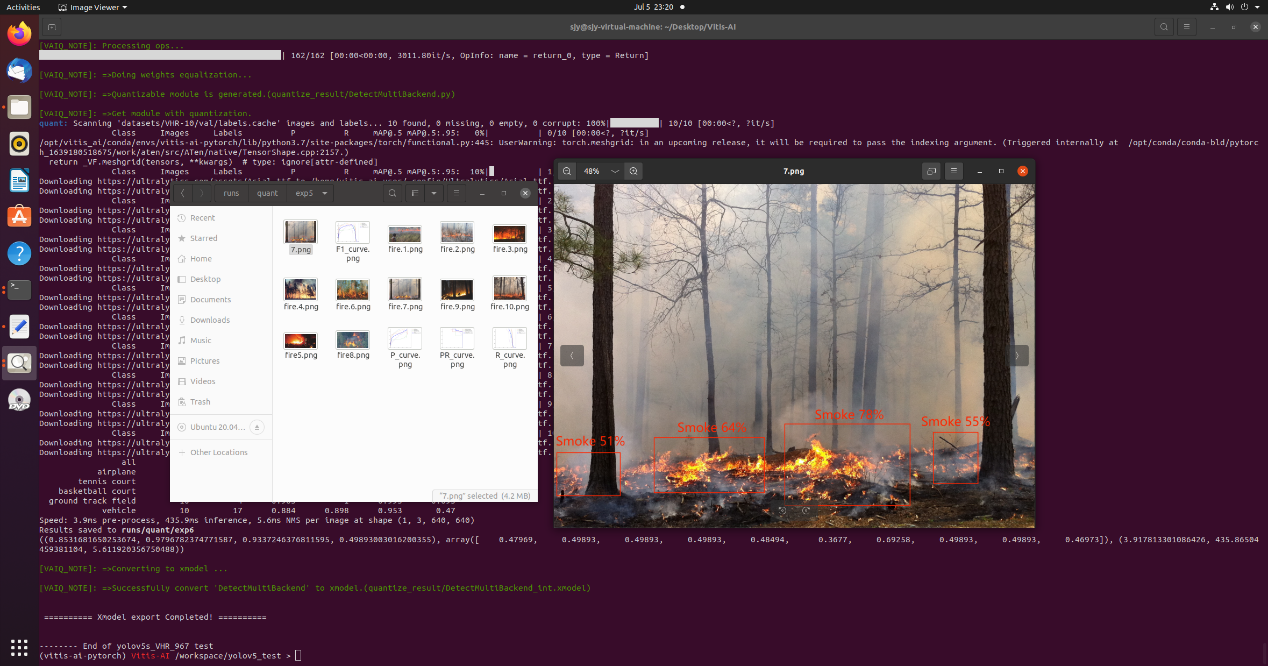
python resnet18\_quant.py --quant\_mode calib --subset\_len



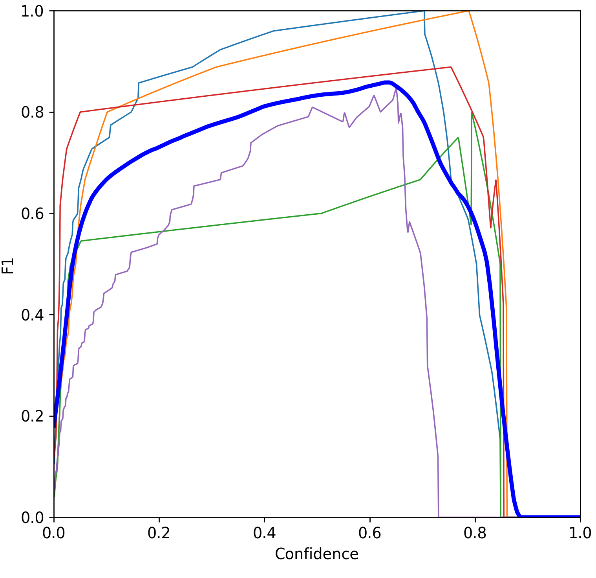
此时量化开始，并且成功加载数据集和模型。并且量化后，开始对数据集中的模型进行预测。



可以看到第一步calib运行完成，可以去runs文件夹中查看运行结果。

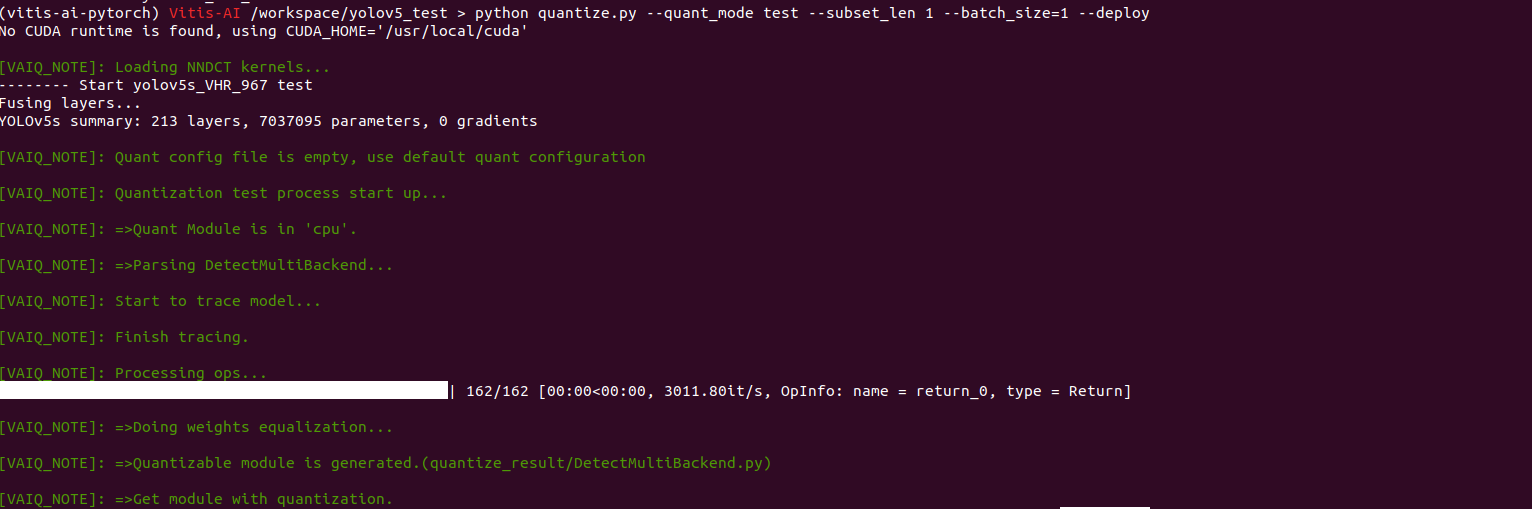


其中不仅生成了量化后模型对数据集的预测，还有量化后模型的各项指标参数：F1、P、R、PR。下图为生成的F1参数曲线。

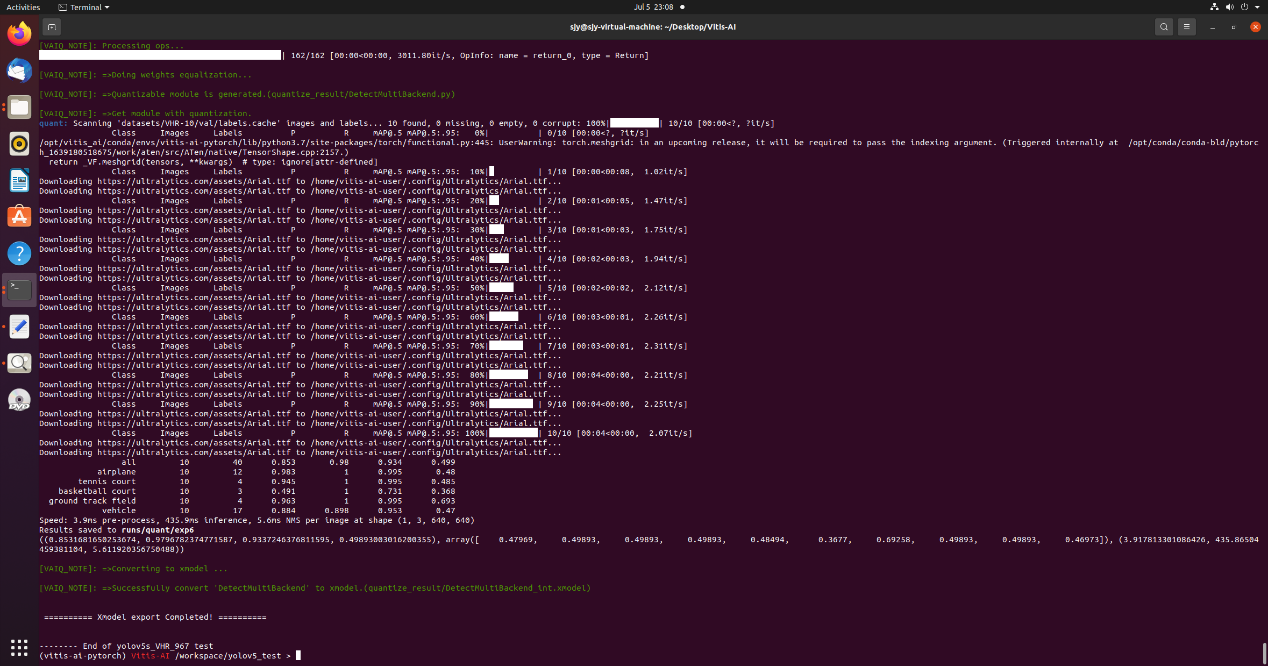


b. 要生成 xmodel 进行编译（以及 onnx 格式量化模型），批次大小应为 1。设置 subset\_len=1 可避免冗余迭代，并运行以下命令：

1. python resnet18\_quant.py *--quant\_mode test --subset\_len 1 --batch\_size=1 --deploy*



可以看到开始正确运行，下面是运行完成结果。同样，结果也会生成在runs文件夹下。

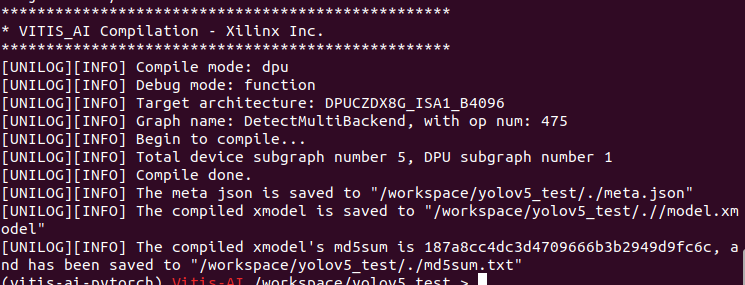


## 2.4 生成可执行文件Xmodel

运行以下命令即可，需要注意的是要选择自己板卡所对应的DPU型号。

1. vai\_c\_xir -x ./quantize\_result/DetectMultiBackend\_int.xmodel -a /opt/vitis\_ai/compiler/arch/DPUCZDX8G/ZCU104/arch.json -o ./ -n model

运行结束，可以看到生成了我们所需要的Xmodel可执行文件。



## 2.5 部署和运行

Vitis AI 提供了 C++ DpuRunner 类，其中包含以下接口：

1）提交输入张量以供执行，提交输出张量则用于存储结果。主机指针是使用 TensorBuffer 对象来传递的。该函数会 返回作业 ID 以及函数调用的状态。

1. std::pair<uint32\_t, int> execute\_async(
2. const std::vector<TensorBuffer\*>& input,
3. const std::vector<TensorBuffer\*>& output);

2）由 execute\_async 返回的作业 ID 将被传递给 wait() 以供阻塞，直至作业完成且结果就绪为止。

1. int wait(int jobid, int timeout);

3)查询可知其期望的张量格式。返回 DpuRunner::TensorFormat::NCHW 或 DpuRunner::TensorFormat::NHWC

1. TensorFormat get\_tensor\_format()

4）查询 DpuRunner 可知其针对加载的 Vitis AI 模型所期望的输入和输出张量的形状和名称。

1. std::vector<Tensor\*> get\_input\_tensors()
2. std::vector<Tensor\*> get\_output\_tensors()

要创建 DpuRunner 对象，调用以下函数

1. create\_runner(const xir::Subgraph\* subgraph, const std::string& mode =
2. "")