西安交通大學

XI'AN JIAOTONG UNIVERSITY

项目设计报告

课程: 项目设计

姓名: 陈思成

班级: 微电子 002

时间: 2024 年 3 月

目录

1 介绍 **4**

1.1 目标检测 4

1.1.1 目标检测模型和算法 4

1.1.2 目标检测评价指标 5

1.1.3 YOLO 系列模型简介 6

1.2 量化 8

1.2.1 部署流程 8

1.2.2 背景介绍 9

1.2.3 量化方法 9

1.2.4 图融合 10

2 方法 **11**

2.1 环境准备 11

2.1.1 安装 CUDA 11

2.1.2 安装 cuDNN 11

2.1.3 安装 TensorRT 12

2.1.4 安装 Visual Studio 12

2.1.5 安装 PyCharm/VSCode 12

2.1.6 安装 Conda（可选） 12

2.1.7 安装 PyTorch 和 YOLOv8 13

2.1.8 测试 YOLOv8 14

2.2 模型训练 14

2.2.1 训练平台 15

2.2.2 训练数据集 15

2.2.3 训练参数 15

2.2.4 训练代码 16

2.3 模型量化 16

2.3.1 Ultralytics 量化 16

2.3.2 TensorRT 量化 18

2.3.3 PPQ 量化 19

3 结果 **20**
3.1 训练结果 20
3.2 量化结果 20

4 总结 **21**
4.1 训练前后对比 21
4.2 不同精度对比 21
4.3 不同格式对比 21
4.4 不同量化工具对比 22
4.5 PPQ 内部对比 22
4.6 个人经验 22

A 训练参数 **23**

B TensorRT 量化代码 **23**

C PPQ 量化代码 **27**

D PPQ 解除 22 层量化代码 **29**

E 自定义校准数据集代码 **30**

摘要

YOLOv8 是 Ultralytics 公司开发的目前最受欢迎的深度学习目标检测模型。由于所需算力相对较高，YOLOv8 不能直接在部分算力较低的设备上运行；此外，部分不支持高精度计算的设备也不能直接运行 YOLOv8。需要对 YOLOv8 进行针对部署平台的优化，以保证其能适配部署平台设备。量化是目前最主流的模型优化方式，通过计算图融合和精度缩放在小幅牺牲精度的前提下大幅提升模型在部署平台上的运行效率。本项目设计的课题是探索在量化 YOLOv8 的过程中不同量化操作对模型性能带来的影响，并对比量化前后模型的各项性能指标，从而确定量化对模型性能带来的增益。

1 介绍

1.1 目标检测

1.1.1 目标检测模型和算法

目标检测是深度学习中的一类任务。目标检测模型的目的是找出图像或视频中人们感兴趣的物体，同时检测出它们的位置和大小。

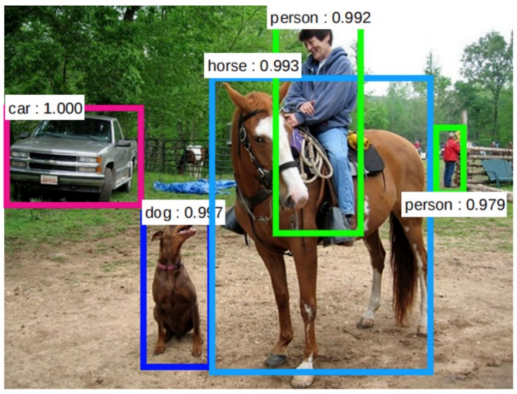


图 1: 目标检测可视化示例

近年来，深度学习目标检测模型经历了快速且多样化的发展。目前为止，该领域内主要有两大算法派系：

1. 以 R-CNN 为代表的二阶段算法。模型在一阶段生成预选框，每个预选框代表一个模型认为的感兴趣区域，进行目标检测；在二阶段对不同预选框内的区域抽取特征，进行目标分类。

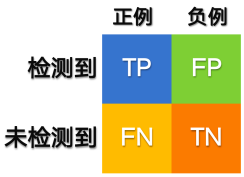
2. 以 YOLOv1 为代表的一阶段算法。没有生成预选框的一阶段，模型直接对整张图片抽取特征，同时完成目标检测和分类。

在目标检测模型发展之初，因为精度所限，二阶段算法首先被提出，在牺牲速度的前提下达成较高的精度。随后，YOLOv1 的问世带来了一阶段算法。它的名字（You Only Look Once）说明一阶段算法速度更快，相比二阶段算法更加符合目标检测任务所追求的时效性。

早期的一阶段算法虽然在速度上大幅领先二阶段算法，但在精度上不能让人满意，因此彼时两大算法都有很多支持者；随着深度学习的逐渐发展，各种新想法的提出使得一阶段算法的精度进步显著，与二阶段算法的差距在逐渐缩小，但在速度上的优势却依旧显著。因此，一阶段算法逐渐成为目标检测模型的主流算法。

1.1.2 目标检测评价指标

在目标检测中，检测结果可以被归为四个类别：模型检测到的正确样本 TP，模型未检测到的正确样本 FN，模型检测到的错误样本 FP，模型未检测到的错误样本 TN。四者关系如下：



目标检测的评价指标围绕两个主要指标展开：查准率（Precision）和查全率（Recall）。

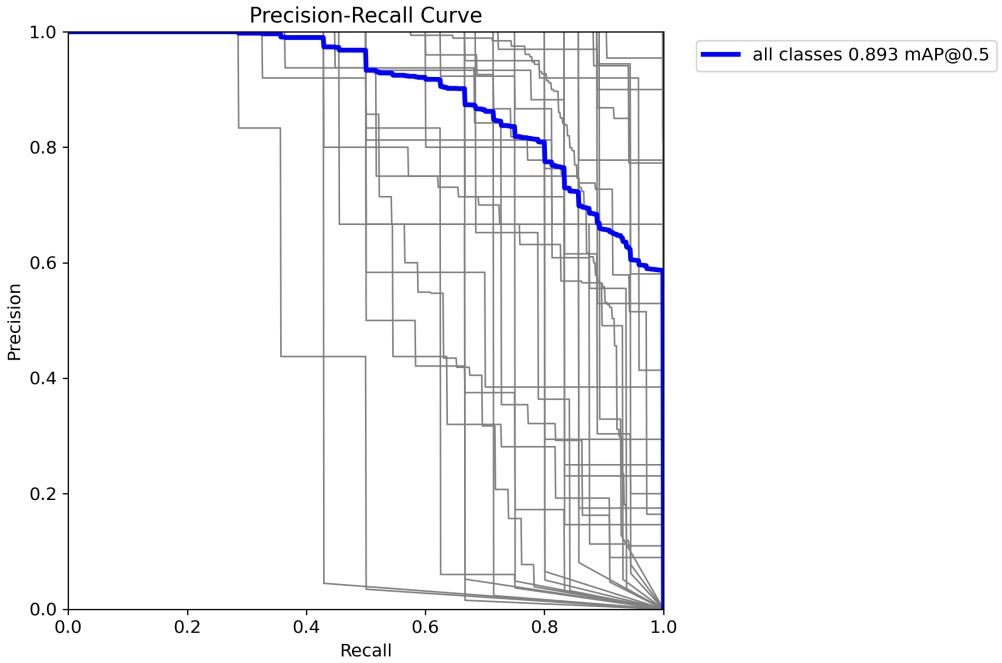
查准率指模型检测到的正确样本在模型检测到的总样本中的占比：

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

查全率指模型检测到的正确样本在总正确样本中的占比：

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

以这两项指标为坐标轴可以画出 PR 曲线，而 PR 曲线和坐标轴所围成的面积则代表平均准确率 AP（Average Precision）。在多目标检测模型中，所有目标 AP 的平均值 mAP 一般被认为可以较为全面的反应模型的性能。



mAP 目前有两大主流标准：mAP50 和 mAP50-95，后边的数字代表的是 IoU 阈值。IoU（Intersection of Union）指的是模型预测的边界框和人工标注的真实边界框的交集和并集的面积之比：

$$IoU = \frac{|Pred \cap GT|}{|Pred \cup GT|} \tag{3}$$

mAP50 表示以 IoU50% 为判定模型是否检测到目标的阈值，在该阈值下计算得到的 mAP；50-95 则是把 IoU 阈值从 50% 开始，以每 5% 为间隔，分别计算不同阈值下的 mAP，直到 IoU 阈值达到 95%。最后把不同 IoU 阈值下的 mAP 求平均，得到最终 mAP。

1.1.3 YOLO 系列模型简介

随着一阶段算法逐渐成为主流，其代表模型——YOLO 系列也成为了最为广泛使用的目标检测算法。自从 YOLOv3 起，YOLO 系列的原作者宣布

退出深度学习界，由 Ultralytics 公司接手版权。Ultralytics 公司在 YOLOv3 的基础上推出了 YOLOv3-SPP 以及 YOLOv5 的多个版本。同时，其他作者也踊跃开发，YOLOv4，YOLOv6，YOLOv7，YOLOX 等作品相继问世。

2023 年 1 月，Ultralytics 公司发布了 YOLOv8，YOLO 家族的最近作品。与前任 YOLOv5 相比，YOLOv8 吸取了 YOLOv7,YOLOX 等其他作者的设计思路，引入了部分新想法来提升模型的性能。

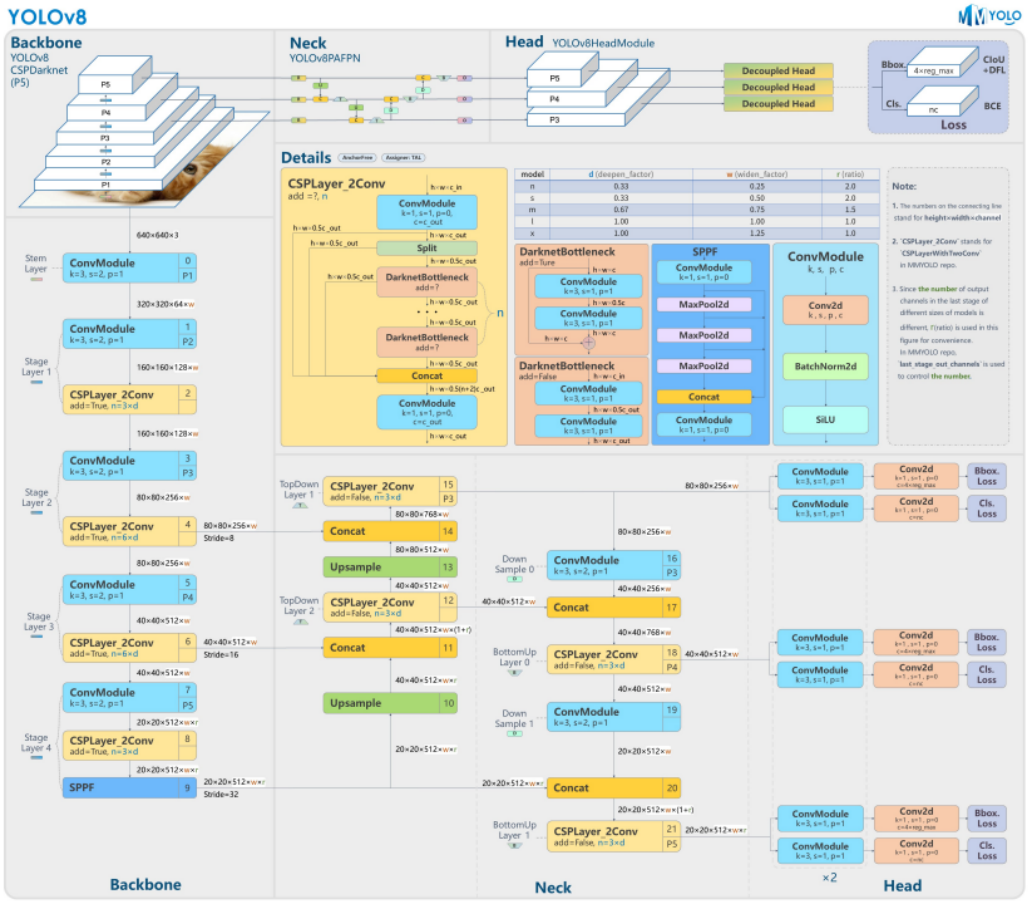


图 2: YOLOv8 结构图

YOLOv8 仍为全卷积网络，没有引入目前主流的 Transformer 结构，因为其所需算力大幅超越全卷积网络，而性能提升并不明显，不适合目标检测这类注重时效性的任务。因此，全卷积网络仍旧是最均衡的选择。

除了没有使用 Transformer，YOLOv8 采用了几乎所有的主流算法，如

解耦头，SPP 模块，FPN 模块以及 C2f 模块。因此，它的性能相当出众。但是 YOLOv8 最受欢迎的并不是它的优越性能，而是 Ultralytics 公司为它提供的丰富 API，满足绝大部分人的使用需求。YOLOv8 是现在最容易上手，功能最丰富的目标检测模型。

1.2 量化

1.2.1 部署流程

经过训练后的深度学习模型的各项指标已经满足需求，不再需要对其进行参数更新，而只需其进行推理，因此可以通过一系列不可逆的操作使其具备更好的泛化性和更高的运算速度，从而满足部署平台的需求，这一过程称为部署流程。

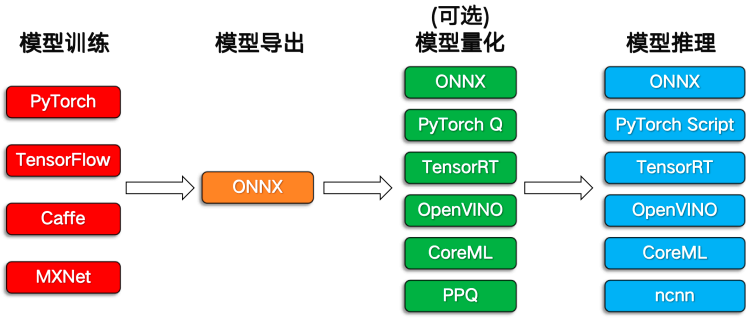


图 3: 模型部署流程

部署流程一般包含模型导出，模型优化（可选）和模型推理三步。模型导出指把模型从训练框架的格式（PyTorch，TensorFlow 等）导出到通用格式 ONNX；模型优化包含模型量化，模型剪枝，模型蒸馏等，其中量化是目前应用范围最广的方式；模型推理则是把模型从通用格式导出到各个部署平台的推理框架的格式（TensorRT,OpenVINO,ncnn 等）。

在早期，量化工具通常由训练框架或者推理框架提供，功能较少；随着量化的发展，量化工具逐渐规范化，在功能增加的同时也衍生出了专门的量化框架（PPQ 等）。相较于训练框架或推理框架提供的量化工具，量化框架更加专业全面。使用者可以知晓量化流程中每一步的详细信息，并可以

自行决定某一层网络或某一个算子的量化方式，甚至可以自行配置量化信息，拥有更高的自由度。此外，量化框架支持将量化后的模型导出到不同的推理框架，具备更好的泛化性。因此，量化框架是最好用的量化工具。

1.2.2 背景介绍

在深度学习模型部署的过程中，部分部署平台不支持模型的原生精度，或者在模型的原生精度下算力不满足运行需求，因此需要一种方法来把模型从原生的较高精度缩放到部署平台支持的较低精度，从而保证其在部署平台能够正常运行，这种方法被叫做量化。

量化的核心思想在于通过确定缩放因子和偏移量，将原本的高精度参数，如常见的 FP32 和 INT64 类型参数，缩放到相对较低的精度，如 FP16，INT8 和 INT4，并尽量保持模型本身的准确性。因此，量化的目标就是通过校准或训练来确定每一个参数所对应的缩放因子和偏移量。

$$xq = Clip(Round(\frac{x}{Scale} + Offset)) \tag{4}$$

根据各大部署平台给出的信息，量化通常能给模型带来三到十倍的性能增益。因此，量化不仅被用于解决部署平台精度不匹配的问题，也被用于节省计算资源或者提升推理速度.

1.2.3 量化方法

目前常见的量化方法有两种：后训练量化（PTQ，Post-Training Quantization）和量化感知训练（QAT，Quantization-Aware Training）。

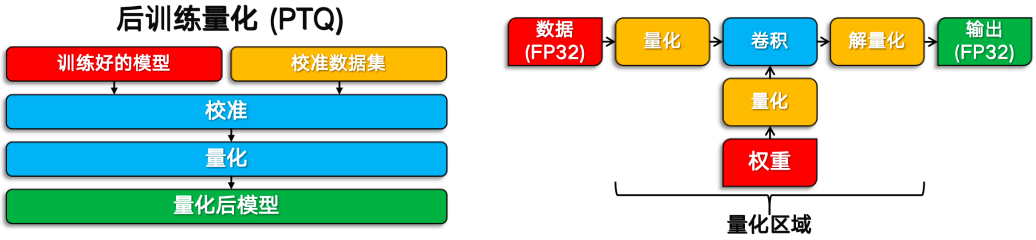


图 4: PTQ 流程

PTQ 是最常用的量化方式。其流程为将模型训练集的一个较小的子集（通常为 1000 张图像左右）作为校准数据集输入到需要量化的模型中，为量化过程中计算缩放因子和偏移量提供数据分布信息，从而完成二者的校准工作。

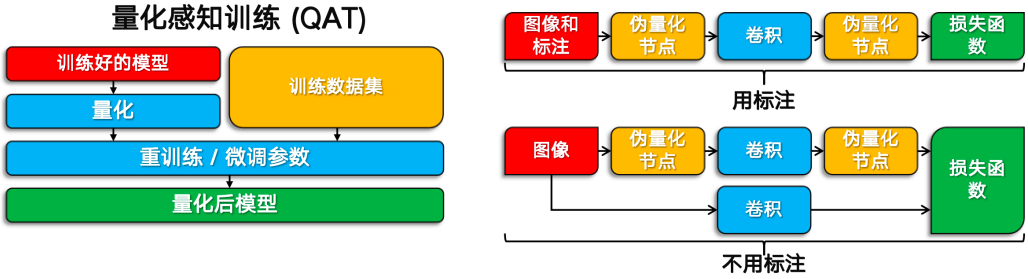


图 5: QAT 流程

QAT 是最近提出的量化方式。其流程为将伪量化节点插入到需要量化模型的每一个算子的输入输出，并将训练集分别输入插入伪量化节点模型和原模型，比较二者的输出结果，从而去调整伪量化节点中缩放因子和偏移量的值。

目前各大主流框架开始逐渐支持效果更好的 QAT，但量化过程更快的 PTQ 仍然是主流方式，故本次项目设计采用的是更加成熟的 PTQ。

1.2.4 图融合

随着量化技术的逐渐发展，图融合被引入到量化流程中，进一步提升了量化对模型的性能提升。

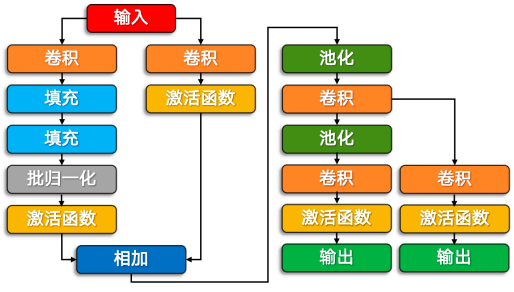
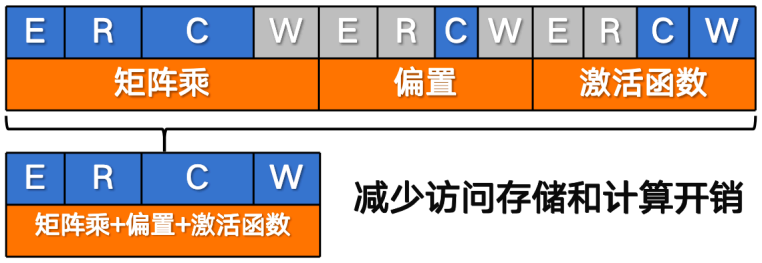


图 6: 计算图示例

图融合指的是对模型计算图中的算子（如卷积，批归一化，相加，拼接等）进行比较，将可以相融的算子进行融合。该操作通过大幅减少算子数量来简化模型计算图，从而降低模型的所需算力。这一流程通常发生在精度缩放，即狭义量化之前。



图融合配合精度缩放（狭义量化）组成了现在的主流量化流程。为了便于区别，后将狭义量化称为精度缩放，将图融合和精度缩放合称为量化。

2 方法

2.1 环境准备

2.1.1 安装 CUDA

CUDA 是 Compute Unified Device Architecture 的缩写，是为了让 NVIDIA 显卡可以完成通用计算任务的一种集成技术，可以为大吞吐量数据的工作提供很好的加速功能。如果有 NVIDIA 的显卡，请安装 CUDA 以使用显卡为 PyTorch 进行加速。

在CUDA 官网下载 CUDA，进行安装。

2.1.2 安装 cuDNN

cuDNN 全称 NVIDIA CUDA Deep Neural Network library，是一个用于深度神经网络的 GPU 加速库。cuDNN 包含了为神经网络中常见的计算任务提供高度优化的实现。包括前向卷积、反向卷积、注意力机制、矩阵乘法（matmul）、池化（pooling）和归一化（normalization）等。安装 cuDNN 可以加速 PyTorch。

在cuDNN 官网下载 cuDNN（注意不要下载 9.0），把解压出来的三个文件复制到 CUDA 的安装根目录（默认为 C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/vx.x）。

2.1.3 安装 TensorRT

TensorRT 是一款用于高性能深度学习推理的 SDK，包含深度学习推理优化器和运行时，可为推理应用程序提供低延迟和高吞吐量。在运行 engine 格式的模型权重时，需要安装 TensorRT。

在TensorRT 官网下载 TensorRT(需要 NVIDIA 账号，请自行注册)，根据官方指导进行安装。（第 4 步选 b，跳过第 6 步）

2.1.4 安装 Visual Studio

Visual Studio 是微软开发的 IDE，支持 C、F、VB、C/C++ 等多种语言的开发。我们需要其内置的 C++ 编译器编译部分 Python 包。

在Visual Studio 官网下载 Visual Studio 安装器，选择默认组件即可。

2.1.5 安装 PyCharm/VSCode

PyCharm 和 VSCode 是两种主流的集成开发环境（IDE），提供多种辅助功能，可以帮助我们更方便的管理项目。其中 PyCharm 是 Python 的专用 IDE，VSCode 是多种语言的泛用 IDE。二者选择其一即可。

PyCharm：在PyCharm 官网下载 PyCharm 教育版或者专业版（收费，若有远程控制需求可使用），进行安装。

VSCode：在VSCode 官网下载 VSCode，进行安装。

2.1.6 安装 Conda（可选）

在同时进行多个项目时，不同项目的环境依赖可能有冲突，最好给每个项目配置单独的环境来避免冲突。

Conda 是最主流的 Python 环境管理工具。Conda 可以提供简单快速的环境创建，环境切换，环境删除等常用功能，并提供 Conda 认证的 Python

包源，相比 Python 自带的 pypi 源更加规范。使用 Conda 进行环境管理和包管理可以大幅减少各类冲突问题。

本项目设计全程在AutoDL 云平台所租用的服务器远程进行，不需要在本地进行 Python 环境配置。同时，AutoDL 云平台所提供的服务器自带 Conda 环境，故跳过本步骤。若在本地进行项目，建议进行本步骤以实现更好的项目管理。

在Anaconda 官网下载 Anaconda，进行安装。

2.1.7 安装 PyTorch 和 YOLOv8

用 Anaconda Prompt 创建一个虚拟环境，并在该虚拟环境中安装 PyTorch 和 YOLOv8:

```
# 创建YOLOv8安装Python环境
conda create -n yolo python=3.11 -y
# 激活YOLOv8安装Python环境
conda activate yolo
# 安装PyTorch。注意老显卡可能不支持CUDA12，请自行查看自己显卡最高支持的CUDA版本
# 查看显卡最高支持CUDA版本：在命令行输入nvidia-smi，回车后右上角显示的CUDA Version即是
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl
    /cu121
# 安装YOLOv8。pip会自动把所需的所有依赖包都安装，因此只需要这一行指令即可
pip install ultralytics
# 安装PPQ
pip install ppq
# 安装Ninja
pip install ninja
# 安装onnx
pip install onnx
# 安装onnxruntime
pip install onnxruntime
# 安装onnxruntime-gpu
pip install onnxruntime-gpu
# 安装TensorRT。前边已经装过了，但以防万一
pip install tensorrt
```

后续可能会需要安装其他包，输入第四行代码后修改第九行代码为要导入的包名运行即可。

2.1.8 测试 YOLOv8

安装完成后，首先测试一下 YOLOv8 是否安装成功：

```python
from ultralytics import YOLO
from multiprocessing import freeze_support

if __name__ == '__main__':
    freeze_support()  # 一个运行bug，如果不加这句会报错
    model = YOLO('yolov8n.pt')
    results = model.val(data='coco128.yaml')
```

如果成功安装 YOLOv8，此时程序应该会在运行时显示如下界面。如果有独立显卡，请注意此处是否显示 CUDA。若为 CPU 则说明在安装 PyTorch 时没有正确安装 CUDA，请自行查看是否存在版本冲突并更换版本重新安装 PyTorch。

```
Ultralytics YOLOv8.1.16 🚀 Python-3.11.7 torch-2.2.0+cu121 CUDA:0 (NVIDIA GeForce RTX 4060 Ti, 8188MiB)
YOLOv8n summary (fused): 168 layers, 3151904 parameters, 0 gradients, 8.7 GFLOPs
```

程序运行结束时显示如下界面：

```
               Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 8/8
                 all        128        929      0.64      0.537      0.605      0.446
              person        128        254      0.796      0.677      0.765      0.538
             bicycle        128          6      0.514      0.333      0.315      0.264
                 car        128         46      0.813      0.217      0.273      0.168
          motorcycle        128          5      0.687      0.887      0.898      0.685
            airplane        128          6      0.819      0.833      0.927      0.675
                 bus        128          7      0.492      0.714      0.728      0.671
               train        128          3      0.534      0.667      0.706      0.604
               truck        128         12          1      0.332      0.473      0.297
                boat        128          6      0.226      0.167      0.316      0.135
       traffic light        128         14      0.734      0.201      0.202      0.139
Speed: 1.5ms preprocess, 10.4ms inference, 0.0ms loss, 1.7ms postprocess per image
Results saved to runs\detect\val7
```

至此，环境准备工作完成。

2.2 模型训练

通常，一个全新的模型会根据任务类型不同（分类，分割，目标检测等）在 ImageNet，COCO 2017 或者 PASCAL VOC 这类公认的数据量极大，标注详细且准确的大数据集上进行初次训练，也叫做预训练。

预训练通常耗费很多时间和资源，因为模型权重要从相对无序的状态慢慢收敛到有序的状态。此外，大训练集也大幅提升了算力需求。一般只有较大型的机构可以进行模型的预训练。

因为大数据集给予的通用知识，经过预训练的模型通常具备较强的泛化性，为满足下游任务的特定需求做好准备。下游任务一般是领域特定或者目标特定的任务，所提供的训练集相比预训练集更小而专。经过预训练的模型通常可以在降低所需算力和训练时间的同时在下游训练集更好收敛。预训练和下游训练分别满足了不同的需求，都是整体训练流程中不可或缺的步骤。

在进行模型量化之前，我们需要对模型在特定数据集上进行训练，以保证其能更好的适应我们的任务。我们采用 Ultralytics 公司预训练的 YOLOv8n 权重，在其基础上展开我们的下游训练任务。

2.2.1 训练平台

本次项目设计的训练平台采用 AutoDL 云平台所提供的服务器，硬件和软件参数如下：

GPU	CPU	RAM	CUDA
NVIDIA RTX4090	Intel Xeon Platinum 8352V	90GB	12.1
TensorRT	Python	PyTorch	Ultralytics
8.6.1	3.11.7	2.2.0	8.1.17

2.2.2 训练数据集

本次项目设计所使用的数据集为 COCO128，由 COCO 2017 的训练集中的前 128 张图像组成。根据官方介绍，YOLOv8 的检测模型由 COCO 2017 进行预训练，因此使用 COCO 128 作为本次项目设计的训练数据集一方面可以加速模型收敛，一方面可以减少训练时间。同时，COCO 128 数据集所包含的类别丰富的实例也可以较为全面的反应模型在实际应用场景中的性能表现。

2.2.3 训练参数

见附录 A。

2.2.4 训练代码

```
from ultralytics import YOLO
from multiprocessing import freeze_support

if __name__ == '__main__':
    freeze_support()
    model = YOLO('weights/yolov8n.pt')
    model.train(data='coco128.yaml',pochs=100)
```

2.3 模型量化

经过训练后的模型可以进行量化。量化已经成为主流的模型优化方式，Ultralytics 的导出 API 因此提供了极其便利的多种量化设置。此外，各大推理框架也提供了自己的量化 API。最后，之前提到的量化框架也提供了更加丰富的量化设置。

本次项目设计将以 NVIDIA RTX4090 为目标部署设备，分别使用 Ultralytics 提供的官方 API，NVIDIA 提供的 TensorRT API 和 PPQ 提供的 API，用三种方式分别进行量化，并比较各个量化方式之间的区别。

2.3.1 Ultralytics 量化

Ultralytics 公司为 YOLOv8 的导出 API提供了极其丰富便利的选项:

格式	format 论据	模型	元数据	论据
PyTorch	-	yolov8n.pt	✅	-
TorchScript	torchscript	yolov8n.torchscript	✅	imgsz, optimize
ONNX	onnx	yolov8n.onnx	✅	imgsz, half, dynamic, simplify, opset
OpenVINO	openvino	yolov8n_openvino_model/	✅	imgsz, half, int8
TensorRT	engine	yolov8n.engine	✅	imgsz, half, dynamic, simplify, workspace
CoreML	coreml	yolov8n.mlpackage	✅	imgsz, half, int8, nms
TF SavedModel	saved_model	yolov8n_saved_model/	✅	imgsz, keras, int8
TF GraphDef	pb	yolov8n.pb	❌	imgsz
TF 轻型	tflite	yolov8n.tflite	✅	imgsz, half, int8
TF 边缘TPU	edgetpu	yolov8n_edgetpu.tflite	✅	imgsz
TF.js	tfjs	yolov8n_web_model/	✅	imgsz, half, int8
PaddlePaddle	paddle	yolov8n_paddle_model/	✅	imgsz
ncnn	ncnn	yolov8n_ncnn_model/	✅	imgsz, half

图 7: 官方支持的导出格式

钥匙	默认值	说明
format	'torchscript'	指定导出格式。支持的值包括 'torchscript', 'onnx', 'coreml', 'engine' (TensorRT)、 'saved_model' (TensorFlow SavedModel) 等。
imgsz	640	定义导出图像的大小。对于正方形图像，接受一个整数，或者一个元组 (height, width) 用于非正方形图像。
keras	False	导出到TensorFlow SavedModel 时，将其设置为 True 利用 Keras 进行输出处理。
optimize	False	适用于TorchScript 导出，可优化移动部署。
half	False	为导出模型启用半精度（FP16）量化，在兼容硬件上可减小体积并提高推理速度。
int8	False	激活 INT8 量化，以牺牲精度为代价，进一步缩小模型大小，提高推理速度。适用于边缘设备。
dynamic	False	对于ONNX 和TensorRT 格式，可启用动态轴，允许在推理中使用不同的输入尺寸。
simplify	False	简化ONNX 和TensorRT 格式的模型结构，提高效率和兼容性。
opset	None	指定用于导出的ONNX opset 版本。如果未设置，则使用最新支持的版本。用于确保与旧版ONNX 解析器兼容。
workspace	4	以 GB 为单位定义TensorRT 导出的最大工作空间大小，影响优化过程和内存使用。
nms	False	导出到CoreML 时，为模型添加非最大值抑制 (NMS) 层，用于过滤重叠检测。

图 8: 官方导出 API 设置

可以看到，YOLOv8 可以一键式导出到各大主流推理框架，并且支持大部分常用的导出参数设定，基本覆盖了大部分的使用场景。但本次项目设计的主题是探索量化的详细过程，因此 Ultralytics 公司提供的 API 只作为探索量化的第一步，并为后续更高自由度的量化提供对比参照。

本次的目标部署平台是 NVIDIA RTX4090，使用如下的导出设置：

```
1 from ultralytics import YOLO
2
3 model = YOLO('weights/best.pt')
4 model.export( format='engine')
```

运行代码分别得到后缀为 ONNX 和后缀为 engine 的两个文件。ONNX 是通用格式，是连接训练框架和推理框架的桥梁，而 engine 则是 TensoRT 专用的格式。在所有导出流程中，训练框架格式的模型权重都会先导出到 ONNX，再导出到推理框架格式。ONNX 本身也可以直接运行在各大推理框架上，但效率低于专用格式。

拿到两个导出模型权重后，使用 Ultralytics 提供的验证 API 比较量化前后的性能差异和 ONNX 格式和 engine 格式的性能差异：

```python
from ultralytics import YOLO
from multiprocessing import freeze_support

if __name__ == '__main__':
    freeze_support()

    WEIGHTS = [
        'weights/best.pt',
        'weights/FP32.onnx',
        'weights/FP32.engine',
        'weights/FP16.engine',
        'weights/MinMax.engine',
        'weights/Entropy2.engine',
        'weights/PPQ_Default.onnx',
        'weights/PPQ_Remove22.onnx',
    ]
    for WEIGHT in WEIGHTS:
        model = YOLO(WEIGHT)
        results = model.val(data='Coco128.yaml',
                            imgsz=640,
                            batch=1,
                            )
```

所有运行结果对比放在结果部分的表格中。

2.3.2 TensorRT 量化

Ultralytics 公司提供的导出 API 虽然相对完善，但也不能很好的满足本次项目设计的探索需求。比如阅读源码会发现，其提供的导出 API 在导出为 engine 格式时只能导出为 FP16 格式，但查阅 TensoRT 官网发现，官方的量化器是支持 INT8 量化的，只是 Ultralytics 公司没有提供该量化配置。因此我们需要自己配置 TensorRT 的 INT8 量化流程。

我们直接使用上一步中导出的 ONNX 格式模型权重，并根据 NVIDIA 官方的量化教程进行量化器的配置和量化流程的部署。

经实验发现，根据 NVIDIA 官方的量化教程得到的量化后模型权重并不能直接在 Ultralytics 的训练 API 中运行。经查阅代码发现，Ultralytics 公司在模型权重导出过程中在权重中嵌入了模型的元数据，而模型需要元

数据才能在 Ultralytics 的框架下正确运行。

经过多次尝试，发现直接获取模型的元数据难度较大，因此决定继承 Ultralytics 公司导出 API 中的 Exporter 类，并重写其中的 export engine 方法，从而实现使用 Ultralytics 公司的 API 导出 INT8 精度的 engine 格式权重。

需要注意的是，TensorRT 提供了四种不同的校准器。本次项目设计分别使用了四种校准器进行测试，并比较各种校准器的性能。

量化代码见附录 B。

2.3.3 PPQ 量化

TensorRT 提供的 API 已经为量化流程提供了相当高的自由度，但使用者对量化流程的控制还不够精确。本部分使用的 PPQ 量化框架可以提供更为详细的量化设置，并详细显示各类量化指标。

```
--------- Network Snapshot ---------
Num of Op:                    [239]
Num of Quantized Op:          [227]
Num of Variable:              [405]
Num of Quantized Var:         [390]
------- Quantization Snapshot ------
Num of Quant Config:          [710]
ACTIVATED:                    [164]
BAKED:                        [64]
OVERLAPPED:                   [334]
PASSIVE:                      [60]
PASSIVE_BAKED:                [64]
FP32:                         [24]
```

图 9: 量化信息

在 PPQ 中，我们可以自定义每一个算子的量化方式，也可以控制哪些算子和层需要量化，根据打印出的误差分析来灵活决定量化方式。PPQ 还提供了各大推理框架的量化规则，只需要在导出模型权重时选择目标部署平台所对应的量化规则，就可以实现针对部署平台的模型导出。

本次项目设计分别尝试了量化所有层和量化部分层，观察其对模型性能的影响。

量化层模型所有层的代码见附录 C。在量化过程中发现第 22 层的部分算子累计误差较大：

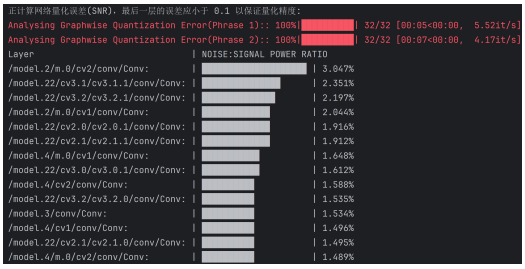


图 10: 累计误差信息部分展示

关闭 22 层量化并观察其对模型产生的影响。代码见附录 D。

3 结果

3.1 训练结果

指标	Precision(%)	Recall(%)	mAP50(%)	mAP50-95(%)
训练前	62.4	52.9	60.2	44.7
训练后	90.3	84.9	89.5	74.8

3.2 量化结果

格式	精度	推理时间 (ms)	查准率 (%)	查全率 (%)	mAP50(%)	mAP50-95(%)
pt	FP32	27.1	90.3	84.9	89.5	74.8
ONNX	FP32	62.0	89.0	82.9	89.2	74.7
engine	FP32	3.7	89.0	82.9	89.2	74.7
engine	FP16	1.9	89.0	82.9	89.1	74.6
engine-MinMax	INT8	1.7	85.1	81.7	87.8	70.2
engine-Entropy2	INT8	1.7	82.1	75.7	85.7	68.7
PPQ_CUDA_Default	INT8	15.3	86.6	82.2	88.2	73.6
PPQ_CUDA_DeQuant22	FP32/INT8	18.9	88.5	82.8	88.9	74.5

4 总结

4.1 训练前后对比

训练前后，YOLOv8n 在 COCO128 数据集上的性能表现显著提升。虽然使用的 YOLOv8n 模型权重已经在其母集 COCO 2017 上进行过预训练，但模型的参数量较少，很难有效拟合全部数据。在体量较小的 COCO128 上训练后，模型可以较好拟合其全部数据。

4.2 不同精度对比

相比于 FP32 精度 pt 格式的原始模型权重，量化为 FP16 精度 engine 格式的模型权重在几乎不损失精度的同时大幅提升运算速度。但量化为 INT8 精度 engine 格式的模型权重与 FP16 相比则在小幅提升运算速度的同时损失较多精度。因此，在 RTX4090 上，FP16 相较于 INT8 是更为合适的量化精度选择。根据 Ultralytics 公司官方提供的 TensorRT 量化 API 中没有提供 INT8 量化也可以认为大部分 NVIDIA 设备使用 FP16 量化的性价比高于 INT8 量化。但在其他平台上，INT8 量化的性价比可能优于 FP16 量化。

4.3 不同格式对比

相比于训练框架 PyTorch 的 pt 格式模型权重，engine 格式的模型权重性能表现提升较为明显，验证了部署流程的重要性。但作为中间格式的 ONNX 模型权重表现反而不如 pt 格式，查阅资料发现 pt 格式针对 NVIDIA 计算平台有优化，使其性能可以超越通用格式 ONNX。

不同格式之间转换时均存在少量的精度损失。查阅资料发现不同格式间转换本质是算子的实现方式之间的转换，这其中存在参数转换导致的误差。但从结果来看，格式转换导致的精度损失可以接受。

4.4 不同量化工具对比

比较令人意外的是，虽然 PPQ 提供了最为专业的量化选择，但其导出的量化后模型权重性能反而是三者中最弱的，而 TensorRT 和 Ultralytics 量化的模型性能则各有优劣。猜测可能是 TensorRT 作为 NVIDIA 官方的量化工具提供了更多和硬件相匹配的优化策略，而 PPQ 作为通用型量化框架不能做到这样的针对性优化。

但 PPQ 并非没有优势。一方面，它提供的丰富平台导出规则有利于开发人员对于没有 TensorRT 这类优秀官方量化工具的平台上进行量化，如 FPGA；另一方面，它提供的丰富量化配置利于开发人员深入了解量化流程，积累经验和知识。

4.5 PPQ 内部对比

在解除 22 层的量化后，模型在增加少量推理时间的基础上减少了可观的精度损失。推论得出，在部署平台算力允许的情况下，可以视情况决定量化部分精度损失较低的算子或层，保留量化损失精度较高的算子或层的精度。查阅资料发现，这项技术已经被广泛应用，叫做 AMP（自动混合精度），且 YOLOv8 默认开启此技术。

4.6 个人经验

本次项目设计是一个漫长而充实的过程。虽然最终没能通过自身的努力使得自己优化的模型性能超越官方，但在探索的过程中自己收获了很多知识。从一开始对量化的一无所知，到四处搜索资料慢慢构建起自己的知识体系，再到实操的过程中碰到各种困难，并一步步解决。每一步都不容易，但自己最终还是完成了。很感谢本次项目设计，让自己学会了很多新知识和技能。

A 训练参数

task: detect	val: true	workspace: 4	resume: false
mode: train	split: val	nms: false	amp: true
model: yolov8n.pt	save_json: false	lr0: 0.01	fraction: 1.0
data: coco128.yaml	save_hybrid: false	lrf: 0.01	profile: false
epochs: 100	conf: null	momentum: 0.937	freeze: null
time: null	iou: 0.7	weight_decay: 0.0005	multi_scale: false
patience: 50	max_det: 300	warmup_epochs: 3.0	overlap_mask: true
batch: 16	half: false	warmup_momentum: 0.8	mask_ratio: 4
imgsz: 640	dnn: false	warmup_bias_lr: 0.1	dropout: 0.0
save: true	plots: true	box: 7.5	show_boxes: true
save_period: -1	source: null	cls: 0.5	line_width: null
cache: false	vid_stride: 1	dfl: 1.5	format: torchscript
device: null	stream_buffer: false	pose: 12.0	keras: false
workers: 8	visualize: false	kobj: 1.0	optimize: false
project: null	augment: false	label_smoothing: 0.0	int8: false
name: train	agnostic_nms: false	nbs: 64	dynamic: false
exist_ok: false	classes: null	hsv_h: 0.015	simplify: false
pretrained: true	retina_masks: false	hsv_s: 0.7	opset: null
optimizer: auto	embed: null	hsv_v: 0.4	mosaic: 1.0
verbose: true	show: false	degrees: 0.0	mixup: 0.0
seed: 0	save_frames: false	translate: 0.1	copy_paste: 0.0
deterministic: true	save_txt: false	scale: 0.5	auto_augment: randaugment
single_cls: false	save_conf: false	shear: 0.0	erasing: 0.4
rect: false	save_crop: false	perspective: 0.0	crop_fraction: 1.0
cos_lr: false	show_labels: true	flipud: 0.0	cfg: null
close_mosaic: 10	show_conf: true	fliplr: 0.5	tracker: botsort.yaml

B TensorRT 量化代码

```python
import os
import onnx
import glob
import pycuda.autoinit  # noqa
import pycuda.driver as cuda  # noqa
import tensorrt as trt
import numpy as np
from PIL import Image
import torchvision.transforms as transforms
import json
import torch
from ultralytics.engine.exporter import Exporter, try_export
from ultralytics.engine.model import Model
from ultralytics.utils import DEFAULT_CFG

PT_PATH = 'weights/best.pt'  # 导入模型权重路径
```

```
17 ENGINE_PATH = 'weights/MinMax.engine'  # 导出模型权重路径
18 CALIBRATION_PATH = 'datasets/coco128/images/train2017'  # 校准数据集路径
19
20
21 class EntropyCalibrator(trt.IInt8MinMaxCalibrator):
22     """
23     根据TensorRT官方教程配置INT8校准器，需要自定义数据集导入和复写几个方法。
24     这里给出的是IInt8MinMaxCalibrator的配置,其他三种校准器的配置步骤大抵相同。
25     四种校准器:
26     IInt8LegacyCalibrator
27     IInt8EntropyCalibrator
28     IInt8EntropyCalibrator2
29     IInt8MinMaxCalibrator
30     注：经测试YOLOv8仅支持IInt8EntropyCalibrator2和IInt8MinMaxCalibrator
31     参考：https://docs.nvidia.com/deeplearning/tensorrt/api/python_api/infer/Int8/
           MinMaxCalibrator.html
32     """
33
34     def __init__(self, data_path):
35         trt.IInt8MinMaxCalibrator.__init__(self)  # 继承官方校准器类的初始化方法
36         self.cache_file = 'Net.cache'  # 量化过程中缓存校准数据存放路径
37         self.batch_size = 1  # 校准数据集每次输入图像批次大小
38         self.Channel = 3  # 校准数据集图像通道数，RGB图像为三通道
39         self.Height = 640  # 校准数据集图像长度
40         self.Width = 640  # 校准数据集图像宽度
41         self.imgs = glob.glob(os.path.join(data_path, '*'))
42         self.batch_idx = 0
43         self.transform = transforms.Compose([
44             transforms.Resize([self.Height, self.Width]),
45             transforms.ToTensor(),
46         ])
47         self.max_batch_idx =  len(self.imgs) // self.batch_size
48         self.data_size = trt.volume([self.batch_size, self.Channel, self.Height, self.
               Width]) * trt.float32.itemsize
49         self.device_input = cuda.mem_alloc(self.data_size)
50
51     # 定义加载校准数据集的迭代器
52     def next_batch(self):
53         while self.batch_idx < self.max_batch_idx:
54             batch_files = self.imgs[self.batch_idx * self.batch_size: (self.batch_idx
                   + 1) * self.batch_size]
55             batch_imgs = np.zeros((self.batch_size, self.Channel, self.Height, self.
```

```
                Width),
56                                  dtype=np.float32)
57            for i, f in  enumerate(batch_files):
58                img = Image. open(f).convert('RGB')
59                img = self.transform(img).numpy()
60                assert (img.nbytes == self.data_size / self.batch_size), 'not valid
                      img!' + f
61                batch_imgs[i] = img
62            self.batch_idx += 1
63            print("batch:[{}/{}]". format(self.batch_idx, self.max_batch_idx))
64            return np.ascontiguousarray(batch_imgs)
65
66    # 下边定义的方法均遵从官方指导
67
68    def get_batch_size(self):
69        return self.batch_size
70
71    def get_batch(self, names):
72        try:
73            batch_imgs = self.next_batch()
74            cuda.memcpy_htod(self.device_input, batch_imgs)
75            return [ int(self.device_input)]
76        except:
77            return None
78
79    def read_calibration_cache(self):
80        if os.path.exists(self.cache_file):
81            with  open(self.cache_file, "rb") as f:
82                return f.read()
83
84    def write_calibration_cache(self, cache):
85        with  open(self.cache_file, "wb") as f:
86            f.write(cache)
87
88
89 class INT8_Exporter(Exporter):
90    """
91    继承Ultralytics官方的导出器，并复写export_engine方法
92    实现自定义INT8量化方法与原导出流程的融合
93    """
94
95    def __init__(self, cfg=DEFAULT_CFG, overrides=None, _callbacks=None, engine_path=
```

```
        None, calibration_path=None, ):
96          super().__init__(cfg, overrides, _callbacks)
97          self.engine_path = engine_path  # 增加导出模型权重路径
98          self.calibration_path = calibration_path  # 增加校准数据集路径
99
100     # 除了把校准器从原来的FP16校准器更换为自定义的INT8校准器和引入校准数据集外，其他代码均未改动
101     @try_export
102     def export_engine(self, prefix="TensorRT:"):
103         f_onnx, _ = self.export_onnx()
104         self.args.simplify = True
105
106         logger = trt.Logger(trt.Logger.INFO)
107         if self.args.verbose:
108             logger.min_severity = trt.Logger.Severity.VERBOSE
109
110         builder = trt.Builder(logger)
111         config = builder.create_builder_config()
112         config.max_workspace_size = self.args.workspace * 1 << 30
113         # config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 1 << 20)  # 1 MiB
114         config.set_flag(trt.BuilderFlag.INT8)
115         calibrator = EntropyCalibrator(data_path=self.calibration_path)
116         config.int8_calibrator = calibrator
117
118         flag = 1 <<  int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH)
119         network = builder.create_network(flag)
120         parser = trt.OnnxParser(network, logger)
121         onnx_model = onnx.load(f_onnx)
122         parser.parse(onnx_model.SerializeToString())
123
124         del self.model
125         torch.cuda.empty_cache()
126
127         # Write file
128         with builder.build_engine(network, config) as engine,
                open(self.engine_path, "wb") as t:
129             # Metadata
130             meta = json.dumps(self.metadata)
131             t.write( len(meta).to_bytes(4, byteorder="little", signed=True))
132             t.write(meta.encode())
133             # Model
134             t.write(engine.serialize())
```

```
135
136         return self.engine_path, None
137
138
139 class TRT_YOLO(Model):
140     # 复写模型中的导出方法使之前自定义的导出器生效
141     def export(self, engine_path, data_path, **kwargs):
142         self._check_is_pytorch_model()
143         custom = {"imgsz": self.model.args["imgsz"], "batch": 1, "data": None, "
                verbose": False}  # method defaults
144         args = {**self.overrides, **custom, **kwargs, "mode": "export"}  # highest
                priority args on the right
145         return INT8_Exporter(
146             engine_path=engine_path,
147             calibration_path=data_path,
148             overrides=args,
149             _callbacks=self.callbacks,
150         )(model=self.model)
151
152
153 if __name__ == '__main__':
154     # 使用自定义的导出设置
155     model = TRT_YOLO(PT_PATH)
156     model.export(
157          format='engine',
158         engine_path=ENGINE_PATH,
159         data_path=CALIBRATION_PATH)
```

C PPQ 量化代码

```
1  from ultralytics import YOLO
2  from ppq import *
3  from ppq.api import *
4  import os
5  from utils import quantization_data_loader
6
7  # 先把模型导出为ONNX格式。注意PPQ最高支持op版本到13，此处需要加入opset参数
8  # model = YOLO('weights/best.pt')
9  # model.export(format='onnx',
10 #              opset=13,
```

```
#             )

WEIGHT_PATH = 'weights/best.onnx'
DATASET_PATH = 'datasets/coco128/images/train2017'
EXPORT_NAME = 'PPQ_Default.onnx'
TARGET_PLATFORM = TargetPlatform.PPL_CUDA_INT8
NETWORK_INPUTSHAPE = [1, 3, 640, 640]
CALIBRATION_BATCHSIZE = 16
EXECUTING_DEVICE = 'cuda'
REQUIRE_ANALYSE = True
TRAINING_YOUR_NETWORK = True

graph = load_onnx_graph(onnx_import_file=WEIGHT_PATH)
QS = QuantizationSettingFactory.default_setting()

if TRAINING_YOUR_NETWORK:
    QS.lsq_optimization = True
    QS.lsq_optimization_setting.steps = 500
    QS.lsq_optimization_setting.collecting_device = 'cuda'

dataloader = quantization_data_loader(DATASET_PATH, num_samples=1024)

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,
    model=graph,
    calib_dataloader=dataloader,
    calib_steps=32,
    input_shape=NETWORK_INPUTSHAPE,
    inputs=None,
    collate_fn=lambda x: x.to(EXECUTING_DEVICE),
    platform=TARGET_PLATFORM,
    device=EXECUTING_DEVICE,
    do_quantize=True)

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))

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(WEIGHT_PATH.rsplit('/', 1)[0], EXPORT_NAME),
)
```

D PPQ 解除 22 层量化代码

```
from ultralytics import YOLO
from ppq import *
from ppq.api import *
import os
from utils import quantization_data_loader
from ppq import QuantableOperation

WEIGHT_PATH = 'weights/best.onnx'
DATASET_PATH = 'datasets/coco128/images/train2017'
EXPORT_NAME = 'PPQ_Remove22.onnx'
TARGET_PLATFORM = TargetPlatform.PPL_CUDA_INT8
NETWORK_INPUTSHAPE = [1, 3, 640, 640]
CALIBRATION_BATCHSIZE = 16
EXECUTING_DEVICE = 'cuda'
TRAINING_YOUR_NETWORK = True

graph = load_onnx_graph(onnx_import_file=WEIGHT_PATH)
QS = QuantizationSettingFactory.default_setting()

if TRAINING_YOUR_NETWORK:
    QS.lsq_optimization = True
    QS.lsq_optimization_setting.steps = 500
    QS.lsq_optimization_setting.collecting_device = 'cuda'

```

```
dataloader = quantization_data_loader(DATASET_PATH, num_samples=1024)

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,
    model=graph,
    calib_dataloader=dataloader,
    calib_steps=32,
    input_shape=NETWORK_INPUTSHAPE,
    inputs=None,
    collate_fn=lambda x: x.to(EXECUTING_DEVICE),
    platform=TARGET_PLATFORM,
    device=EXECUTING_DEVICE,
    do_quantize=True)

# 解除第22层的量化
for op in quantized.operations.values():
    if  isinstance(op, QuantableOperation) and 'model.22' in op.name:
        op.dequantize()

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(WEIGHT_PATH.rsplit('/', 1)[0], EXPORT_NAME),
)
```

E 自定义校准数据集代码

```
import os
import random
import cv2
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms


class Quantization_Dataset(Dataset):
    def __init__(self, root_dir, num_samples=1024, image_size=(640, 640)):
        self.root_dir = root_dir
        self.trans = transforms.ToTensor()
```

```
        self.num_samples = num_samples
        self.image_size = image_size
        self.image_paths = []

        # 遍历文件夹，获取所有图片路径
        for dirpath, _, filenames in os.walk(root_dir):
            for filename in filenames:
                if filename.endswith(('.png', '.jpg', '.jpeg', '.PNG', '.JPG', '.JPEG'
                    )):
                    self.image_paths.append(os.path.join(dirpath, filename))

        # 随机抽取指定数量的图片
        self.image_paths = random.sample(self.image_paths,  min(
            len(self.image_paths), num_samples))

    def __len__(self):
        return  len(self.image_paths)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        img = cv2.imread(img_path)
        img = cv2.resize(img, self.image_size, interpolation=cv2.INTER_LINEAR)
        img = self.trans(img)

        return img


def quantization_data_loader(data_dir, num_samples=4096, image_size=(640, 640)):
    return DataLoader(Quantization_Dataset(root_dir=data_dir,
                                           num_samples=num_samples,
                                           image_size=image_size),
                      )
```