一、安装 Pytorch 及 YOLO v5

docker 方式（推荐）

使用 docker 主要是因为与主机性能区别不大，且配置简单，只需要安装 GPU 驱动，不用考虑安装 Pytorch 指定的 CUDA,cuDNN 等（容器内部已有）

```
# 安装 docker
sudo apt-get remove docker docker-engine docker.io containerd runc
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg lsb-release
sudo mkdir -m 0755 -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
sudo apt-get install nvidia-container2
sudo systemctl restart docker
```

具体安装细节可以参考 docker 文档：https://docs.docker.com/engine/install/ubuntu/。 然后安装 `nvidia-container-toolkit`，依次运行以下命令：

```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID) \
      && curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg \
      && curl -s -L https://nvidia.github.io/libnvidia-container/$distribution/libnvidia-container.list | \
            sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' | \
            sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list
sudo apt-get update
sudo apt-get install -y nvidia-container-toolkit
```

安装完成后，拉取 pytorch 镜像以进行测试（这里我们使用 pytorch 版本

为 1.12.0a0+2c916ef）：

```
# $(pwd):/app 表示把当前 host 工作路径挂载到容器的/app 目录下
# --name env_pyt_1.12 表示容器的名称是 env_pyt_1
docker run --gpus all -it --name env_pyt_1.12 -v $(pwd):/app nvcr.io/nvidia/pytorch:22.03-py3
# 容器内部检查 pytorch 可用性
$ python
>>> import torch
>>> torch.__version__
>>> print(torch.cuda.is_available())
True
```

经过漫长的拉取后，我们便可以进入 docker 的命令行中进行操作。

1.2 安装 YOLO v5 所需依赖

- 安装

```
# 克隆地址
git clone https://github.com/ultralytics/yolov5.git
# 进入目录
cd yolov5
# 选择分支，这里使用了特定版本的 yolov5，主要是避免出现兼容问题
git checkout a80dd66efe0bc7fe3772f259260d5b7278aab42f
# 安装依赖（如果是 docker 环境，要进入容器环境后再安装）
pip3 install -r requirements.txt
```

- 下载预训练权重文件

下载地址：https://github.com/ultralytics/yolov5，附件位置：3.预训练模型/，将权重文件放到 weights 目录下：

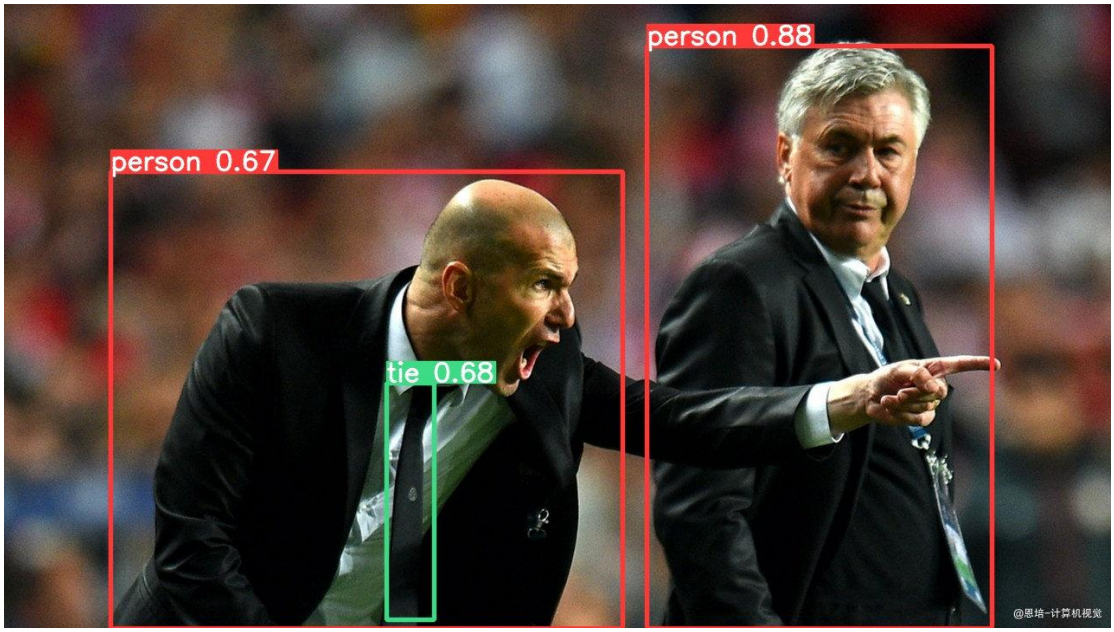
- 测试是否安装成功

```
python detect.py --source ./data/images/ --weights weights/yolov5s.pt --conf-thres 0.4
```

docker 容器内部可能报错：AttributeError: partially initialized module 'cv2' has no attribute

'_registerMatType' (most likely due to a circular import)，使用 `pip3 install "opencv-python-headless<4.3"`

如果一切配置成功，则可以看到以下检测结果



二、YOLO v5 训练自定义数据

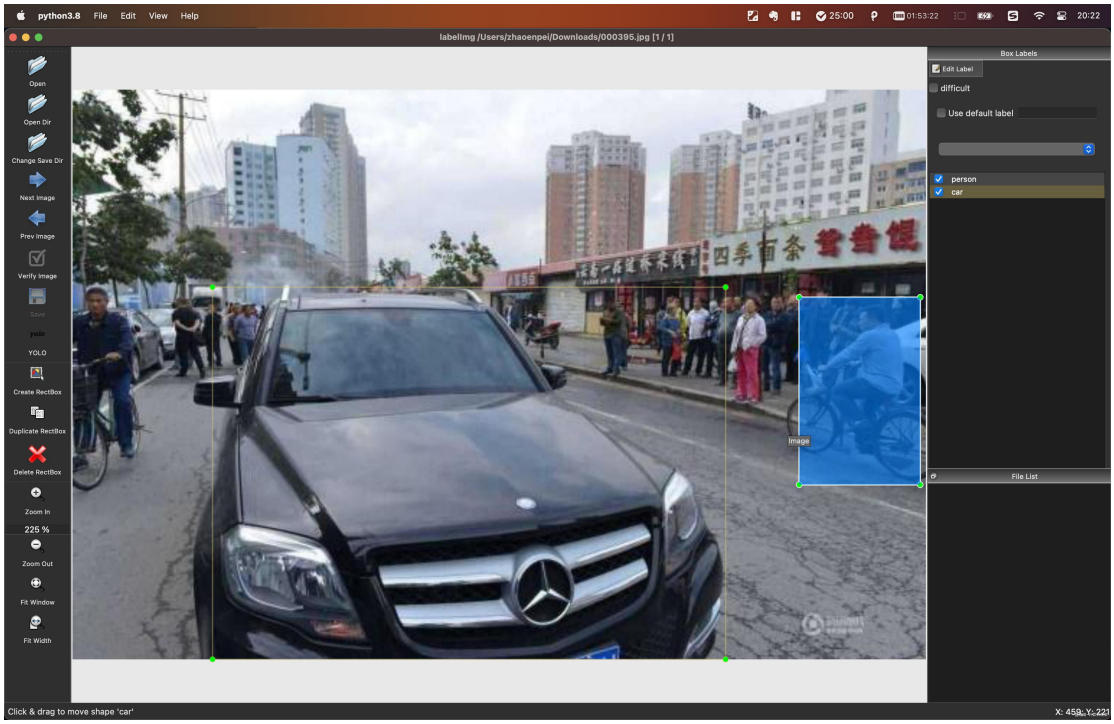
2.1 标注数据

2.1.1 安装 labelImg

需要在有界面的主机上安装，远程 ssh 无法使用窗口

```
# 建议使用 conda 虚拟环境
# 安装
pip install labelImg
# 启动
labelImg
```

2.1.2 标注



- 一张图片对应一个 txt 标注文件（如果图中无所要物体，则无需 txt 文件）；
- txt 每行一个物体（一张图中可以有多个标注）；
- 每行数据格式：类别 id、x_center y_center width height；
- **xywh** 必须归一化（0-1），其中 x_center、width 除以图片宽度，y_center、height 除以画面高度；
- 类别 id 必须从 0 开始计数。

2.2 准备数据集

2.2.1 组织目录结构

```
. 工作路径
├── datasets
│   └── person_data
│       ├── images
│       │   ├── train
│       │   │   └── demo_001.jpg
│       │   └── val
```

```
|       |       └─── demo_002.jpg
|       └─── labels
|           ├─── train
|           |   └─── demo_001.txt
|           └─── val
|               └─── demo_002.txt
└─── yolov5
```

要点：

- datasets 与 yolov5 同级目录；
- 图片 datasets/person_data/images/train/{文件名}.jpg 对应的标注文件在 datasets/person_data/labels/train/{文件名}.txt，YOLO 会根据这个映射关系自动寻找（images 换成 labels）；
- 训练集和验证集
  - images 文件夹下有 train 和 val 文件夹，分别放置训练集和验证集图片;
  - labels 文件夹有 train 和 val 文件夹，分别放置训练集和验证集标签(yolo 格式）；

2.2.2 创建 dataset.yaml

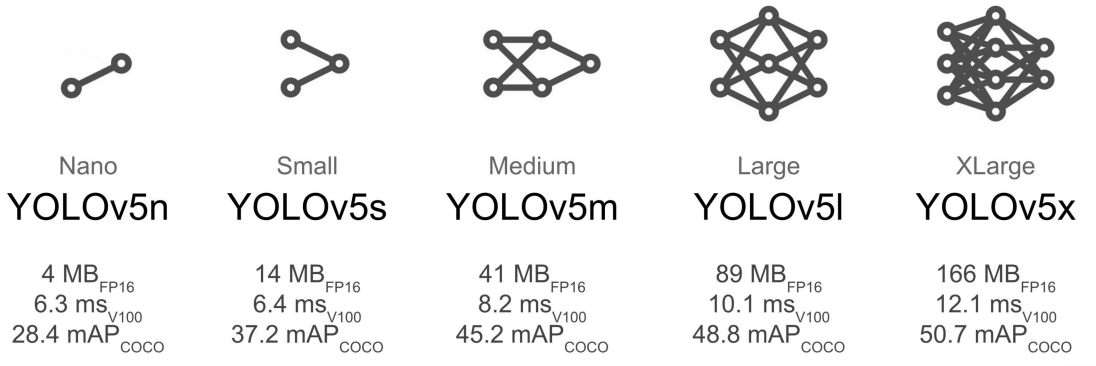
复制 yolov5/data/coco128.yaml 一份，比如为 coco_person.yaml

```
# Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]
path: ../datasets/person_data  # 数据所在目录
train: images/train  # 训练集图片所在位置（相对于 path）
val:  images/val # 验证集图片所在位置（相对于 path）
test:  # 测试集图片所在位置（相对于 path）（可选）
```

```
# 类别
nc: 5  # 类别数量
names: ['pedestrians','riders','partially-visible-person','ignore-regions','crowd'] # 类别标签名
```

2.3 选择合适的预训练模型

官方权重下载地址：https://github.com/ultralytics/yolov5



根据你的设备，选择合适的预训练模型，具体模型比对如下：

Model	size (pixels)	mAP^{val} 0.5:0.95	mAP^{val} 0.5	Speed CPU b1 (ms)	Speed V100 b1 (ms)	Speed V100 b32 (ms)	params (M)	FLOPs @640 (B)
YOLOv5n	640	28.0	45.7	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.4	56.8	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.4	64.1	224	8.2	1.7	21.2	49.0
YOLOv5l	640	49.0	67.3	430	10.1	2.7	46.5	109.1
YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7
YOLOv5n6	1280	36.0	54.4	153	8.1	2.1	3.2	4.6
YOLOv5s6	1280	44.8	63.7	385	8.2	3.6	12.6	16.8
YOLOv5m6	1280	51.3	69.3	887	11.1	6.8	35.7	50.0
YOLOv5l6	1280	53.7	71.3	1784	15.8	10.5	76.8	111.4
YOLOv5x6 + TTA	1280 1536	55.0 **55.8**	72.7 **72.7**	3136 -	26.2 -	19.4 -	140.7 -	209.8 -

复制 models 下对应模型的 yaml 文件，重命名，比如课程另存为 yolov5s_person.yaml，并修改其中：

```
# nc: 80  # 类别数量
```

```
nc: 5  # number of classes
```

2.4 训练

下载对应的预训练模型权重文件，可以放到 weights 目录下，设置本机最好性能的各个参数，即可开始训练，课程中训练了以下参数：

```
# yolov5s
python ./train.py --data ./data/coco_person.yaml --cfg ./models/yolov5s_person.yaml --weights ./weights/yolov5s.pt --batch-size 32 --epochs 120 --workers 0 --name s_120 --project yolo_person_s
```

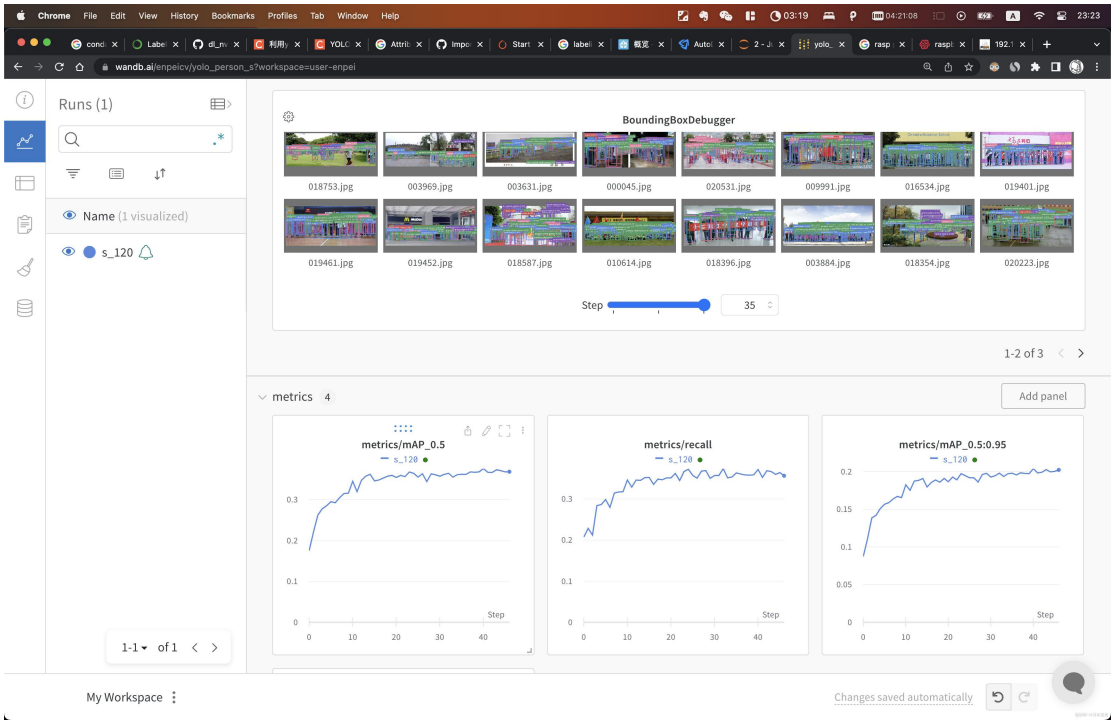
训练结果在 yolo_person_s/ 中可见，一般训练时间在几个小时以上。

2.5 可视化

2.5.1 wandb

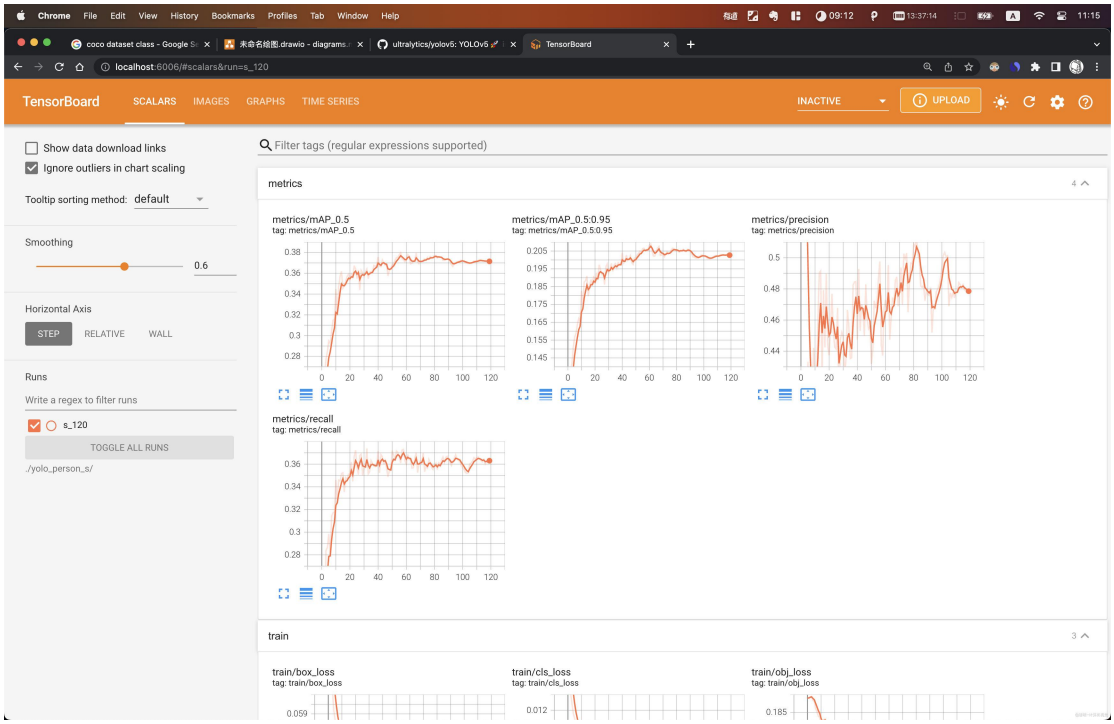
YOLO 官网推荐使用 https://wandb.ai/。

- 去官网注册账号；
- 获取 key 秘钥，地址：https://wandb.ai/authorize
- 使用 pip install wandb 安装包；
- 使用 wandb login 粘贴秘钥后登录；
- 打开网站即可查看训练进展。



2.5.2 Tensorboard

```
tensorboard --logdir=./yolo_person_s
```



2.6 测试评估模型

2.6.1 测试

```
# 如
python detect.py --source ./000057.jpg --weights ./yolo_person_s/s_120/weights/best.pt --conf-thres 0.3
# 或
python detect.py --source ./c3.mp4 --weights ./yolo_person_s/s_120/weights/best.pt --conf-thres 0.3
```

2.6.2 评估

```
# s 模型
# python val.py --data ./data/coco_person.yaml --weights ./yolo_person_s/s_120/weights/best.pt --batch-size 12
# 15.8 GFLOPs
                 Class      Images      Labels           P           R      mAP@.5 mAP@.5:.95:
                   all        1000       28027       0.451       0.372       0.375       0.209
           pedestrians        1000       17600       0.738       0.854       0.879       0.608
                riders        1000         185       0.546       0.492       0.522       0.256
 artially-visible-person      1000        9198       0.461       0.334       0.336       0.125
       ignore-regions         1000         391        0.36       0.132       0.116      0.0463
                 crowd        1000         653       0.152      0.0468      0.0244     0.00841
```

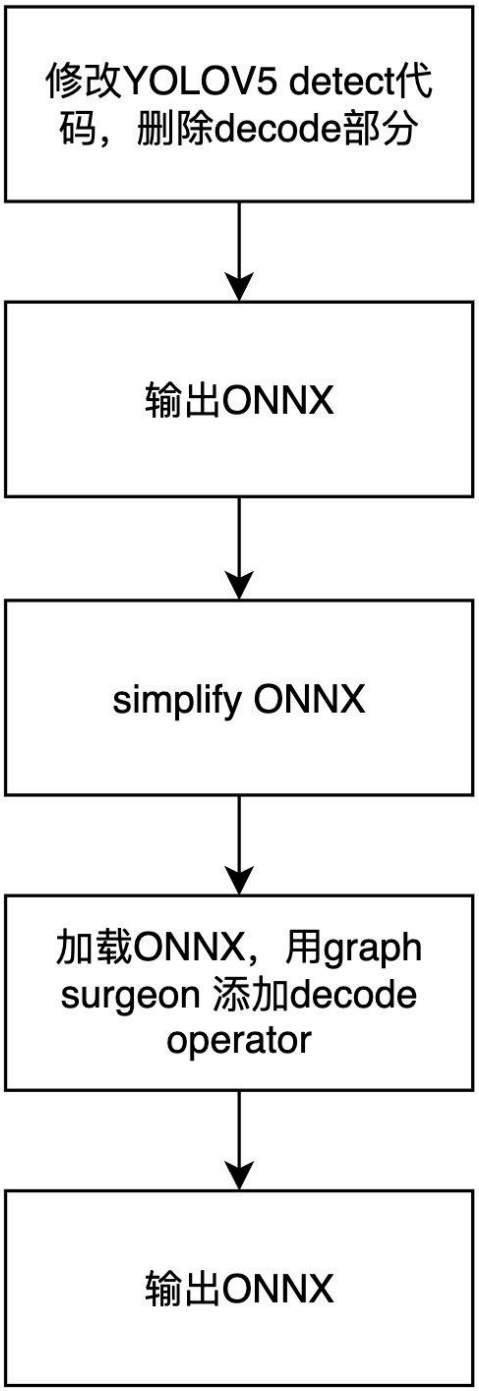
三、yolov5 模型导出 ONNX

在本项目中，我们将使用 tensort decode plugin 来代替原来 yolov5 代码中的 decode 操作，如果不替换，这部分运算将影响整体性能。

为了让 tensorrt 能够识别并加载我们额外添加的 plugin operator，我们需要修改 Yolov5 代码中导出 onnx 模型的部分。

onnx 是一种开放的模型格式，可以用来表示深度学习模型，它是由微软开发的，目前已经成为了深度学习模型的标准格式。可以简单理解为各种框架模型转换的一种桥梁。

流程如图所示：



@思瑞-计算机视觉

3.2 修改 yolov5 代码，输出 ONNX

首先根据上文自行根据 yolov5 的要求安装相关依赖，然后再执行下面命令安装导出 onnx

需要的依赖：

```
pip install seaborn
pip install onnx-graphsurgeon
pip install onnx-simplifier==0.3.10
apt update
apt install -y libgl1-mesa-glx
```

安装完成后，准备好训练好的模型文件，这里默认为 yolov5s.pt，然后执行：

```
python export.py --weights weights/yolov5s.pt --include onnx --simplify --dynamic
```

以生成对应的 onnx 文件。

3.3 具体修改细节

在 models/yolo.py 文件中 54 行，我们需要修改 class Detect 的 forward 方法，以删除其 box decode 运算，以直接输出网络结果。在后面的 tensorrt 部署中，我们将利用 decode plugin 来进行 decode 操作，并用 gpu 加速。修改内容如下：

```
-                bs, _, ny, nx = x[i].shape  # x(bs,255,20,20) to x(bs,3,20,20,85)
-                x[i] = x[i].view(bs, self.na, self.no, ny, nx).permute(0, 1, 3, 4, 2).contiguous()
-                if not self.training:  # inference
-                    if self.onnx_dynamic or self.grid[i].shape[2:4] != x[i].shape[2:4]:
-                        self.grid[i], self.anchor_grid[i] = self._make_grid(nx, ny, i)
-                    y = x[i].sigmoid()
-                    if self.inplace:
-                        y[..., 0:2] = (y[..., 0:2] * 2 + self.grid[i]) * self.stride[i]  # xy
-                        y[..., 2:4] = (y[..., 2:4] * 2) ** 2 * self.anchor_grid[i]  # wh
-                    else:  # for YOLOv5 on AWS Inferentia https://github.com/ultralytics/yolov5/pull/2953
-                        xy, wh, conf = y.split((2, 2, self.nc + 1), 4)  # y.tensor_split((2, 4, 5), 4)  # torch 1.8.0
-                        xy = (xy * 2 + self.grid[i]) * self.stride[i]  # xy
-                        wh = (wh * 2) ** 2 * self.anchor_grid[i]  # wh
-                        y = torch.cat((xy, wh, conf), 4)
-                    z.append(y.view(bs, -1, self.no))
```

```
-            return x if self.training else (torch.cat(z, 1),) if self.export else (torch.cat(z, 1), x)
+                y = x[i].sigmoid()
+                z.append(y)
+            return z
```

可以看到这里删除了主要的运算部分，将模型输出直接作为 list 返回。修改后，onnx 的输出将被修改为三个原始网络输出，我们需要在输出后添加 decode plugin 的算子。首先我们先导出 onnx，再利用 nvidia 的 graph surgeon 来修改 onnx。首先我们修改 onnx export 部分代码：

GraphSurgeon 是 nvidia 提供的工具，可以方便的用于修改、添加或者删除 onnx 网络图中的节点，并生成新的 onnx。参考链接：

https://github.com/NVIDIA/TensorRT/tree/master/tools/onnx-graphsurgeon。

```
torch.onnx.export(model,im,f,verbose=False,opset_version=opset,
            training=torch.onnx.TrainingMode.TRAINING if train else torch.onnx.TrainingMode.EVAL,
            do_constant_folding=not train,input_names=['images'],output_names=['p3', 'p4', 'p5'],
            dynamic_axes={
                    'images': {0: 'batch',2: 'height',3: 'width'},  # shape(1,3,640,640)
                    'p3': {0: 'batch',2: 'height',3: 'width'},  # shape(1,25200,4)
                    'p4': {0: 'batch',2: 'height',3: 'width'},
                    'p5': {0: 'batch',2: 'height',3: 'width'}
            } if dynamic else None)
```

将 onnx 的输出改为 3 个原始网络输出。输出完成后，我们再加载 onnx，并 simplify：

```
model_onnx = onnx.load(f)
model_onnx = onnx.load(f)  # load onnx model
onnx.checker.check_model(model_onnx)  # check onnx model
# Simplify
if simplify:
    # try:
    check_requirements(('onnx-simplifier',))
    import onnxsim
```

```
    LOGGER.info(f'{prefix} simplifying with onnx-simplifier {onnxsim.__version__}...')
    model_onnx, check = onnxsim.simplify(model_onnx,
            dynamic_input_shape=dynamic,
            input_shapes={'images': list(im.shape)} if dynamic else None)
    assert check, 'assert check failed'
    onnx.save(model_onnx, f)
```

然后我们再将 onnx 加载回来，用 nvidia surgeon 进行修改：

```
import onnx_graphsurgeon as onnx_gs
import numpy as np
yolo_graph = onnx_gs.import_onnx(model_onnx)
```

首先我们获取原始的 onnx 输出 p3,p4,p5：

```
p3 = yolo_graph.outputs[0]
p4 = yolo_graph.outputs[1]
p5 = yolo_graph.outputs[2]
```

然后我们定义新的 onnx 输出，由于 decode plugin 中，有 4 个输出，所以我们将定义 4 个新的输出。其名字需要和下面的代码保持一致，这是 decode_plugin 中预先定义好的。

```
decode_out_0 = onnx_gs.Variable(
  "DecodeNumDetection",
  dtype=np.int32
)
decode_out_1 = onnx_gs.Variable(
  "DecodeDetectionBoxes",
  dtype=np.float32
)
decode_out_2 = onnx_gs.Variable(
  "DecodeDetectionScores",
  dtype=np.float32
)
decode_out_3 = onnx_gs.Variable(
  "DecodeDetectionClasses",
```

```
    dtype=np.int32
)
```

然后我们需要再添加一些 decode 参数，定义如下：

```
decode_attrs = dict()
decode_attrs["max_stride"] = int(max(model.stride))
decode_attrs["num_classes"] = model.model[-1].nc
decode_attrs["anchors"] = [float(v) for v in [10,13, 16,30, 33,23, 30,61, 62,45, 59,119, 116,90, 156,198, 373,326]]
decode_attrs["prenms_score_threshold"] = 0.25
```

在定义好了相关参数后，我们创建一个 onnx node，用作 decode plugin。由于我们的 tensorrt plugin 的名称为 YoloLayer_TRT,因此这里我们需要保持 op 的名字与我们的 plugin 名称一致。通过如下代码，我们创建了一个 node：

```
decode_plugin = onnx_gs.Node(
        op="YoloLayer_TRT",
        name="YoloLayer",
        inputs=[p3, p4, p5],
        outputs=[decode_out_0, decode_out_1, decode_out_2, decode_out_3],
        attrs=decode_attrs
    )
```

然后我们将这个 node 添加了网络中：

```
yolo_graph.nodes.append(decode_plugin)
    yolo_graph.outputs = decode_plugin.outputs
    yolo_graph.cleanup().toposort()
    model_onnx = onnx_gs.export_onnx(yolo_graph)
```

最后添加一些 meta 信息后，我们导出最终的 onnx 文件，这个文件可以用于后续的 tensorrt 部署和推理。

```
d = {'stride': int(max(model.stride)), 'names': model.names}
    for k, v in d.items():
        meta = model_onnx.metadata_props.add()
        meta.key, meta.value = k, str(v)
```

```
    onnx.save(model_onnx, f)
    LOGGER.info(f'{prefix} export success, saved as {f} ({file_size(f):.1f} MB)')
    return f
```

四、TensorRT 部署

使用 TensorRT docker 容器：

```
docker run --gpus all -it --name env_trt -v $(pwd):/app nvcr.io/nvidia/tensorrt:22.08-py3
```

4.1 模型构建

代码位置：1.代码/tensorrt_cpp/build.cu

1. **创建 builder**。这里我们使用了 std::unique_ptr 只能指针包装我们的 builder，实现自动管理指针生命周期。

```
// ============ 1. 创建 builder ============
auto builder =
std::unique_ptr<nvinfer1::IBuilder>(nvinfer1::createInferBuilder(sample::gLogger.getTRTLog
ger()));
if (!builder)
{
  std::cerr << "Failed to create builder" << std::endl;
  return -1;
}
```

2. **创建网络。**这里指定了 explicitBatch

```
// ========== 2. 创建 network：builder--->network ==========
// 显性 batch
const auto explicitBatch =
 1U << static_cast<uint32_t>(nvinfer1::NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);
// 调用 builder 的 createNetworkV2 方法创建 network
auto network =
```

```
 std::unique_ptr<nvinfer1::INetworkDefinition>(builder->createNetworkV2(explicitBatch));
if (!network)
{
   std::cout << "Failed to create network" << std::endl;
   return -1;
}
```

3. **创建 config。**用于模型构建的参数配置

```
// ========== 3. 创建 config 配置：builder--->config ==========
auto config = std::unique_ptr<nvinfer1::IBuilderConfig>(builder->createBuilderConfig());
if (!config)
{
   std::cout << "Failed to create config" << std::endl;
   return -1;
}
```

4. **创建 onnx 解析器，**并进行解析

```
// 创建 onnxparser，用于解析 onnx 文件
auto parser = std::unique_ptr<nvonnxparser::IParser>(nvonnxparser::createParser(*network,
sample::gLogger.getTRTLogger()));
// 调用 onnxparser 的 parseFromFile 方法解析 onnx 文件
auto parsed =
parser->parseFromFile(onnx_file_path,static_cast<int>(sample::gLogger.getReportableSeveri
ty()));
if (!parsed)
{
   std::cout << "Failed to parse onnx file" << std::endl;
   return -1;
}
```

5. **配置网络构建参数。**这里由于我们导出 onnx 时并没有指定输入图像的 batch，height，width。因此在构建时，我们需要告诉 tensorrt 我们最终运行时，输入图像的范围，batch size 的范围。这样 tensorrt 才能对应为我们进行模型构建与优化。

这里我们将输入指定到了 1,3,640,640，这样 tensorrt 就会为这个尺寸的输入搜索最优算子并构建网络。其中设置 pofile 参数，即是我们用来指定输入大小搜索范围的

```
auto input = network->getInput(0);
auto profile = builder->createOptimizationProfile();
profile->setDimensions(input->getName(), nvinfer1::OptProfileSelector::kMIN, nvinfer1::Dims4{1, 3, 640, 640});
profile->setDimensions(input->getName(), nvinfer1::OptProfileSelector::kOPT, nvinfer1::Dims4{1, 3, 640, 640});
profile->setDimensions(input->getName(), nvinfer1::OptProfileSelector::kMAX, nvinfer1::Dims4{1, 3, 640, 640});
// 使用 addOptimizationProfile 方法添加 profile，用于设置输入的动态尺寸
config->addOptimizationProfile(profile);
// 设置精度，不设置是 FP32，设置为 FP16，设置为 INT8 需要额外设置 calibrator
config->setFlag(nvinfer1::BuilderFlag::kFP16);
builder->setMaxBatchSize(1);
// 设置最大工作空间（最新版本的 TensorRT 已经废弃了 setWorkspaceSize）
config->setMemoryPoolLimit(nvinfer1::MemoryPoolType::kWORKSPACE, 1 << 30);
// 7. 创建流，用于设置 profile
auto profileStream = samplesCommon::makeCudaStream();
if (!profileStream) { return -1; }
config->setProfileStream(*profileStream);
```

6. 将模型序列化，并进行保存

```
// ========== 4. 创建 engine：builder--->engine(*nework, *config) ==========
// 使用 buildSerializedNetwork 方法创建 engine，可直接返回序列化的 engine（原来的 buildEngineWithConfig 方法已经废弃，需要先创建 engine，再序列化）
auto plan = std::unique_ptr<nvinfer1::IHostMemory>(builder->buildSerializedNetwork(*network, *config));
if (!plan)
{
```

```
    std::cout << "Failed to create engine" << std::endl;
    return -1;
}
// ========== 5. 序列化保存 engine ==========
std::ofstream engine_file("./model/yolov5.engine", std::ios::binary);
assert(engine_file.is_open() && "Failed to open engine file");
engine_file.write((char *)plan->data(), plan->size());
engine_file.close();
```

4.2 模型的推理

代码位置：1.代码/tensorrt_cpp/build.cu

1. 创建运行时

```
// ========== 1. 创建推理运行时 runtime ==========
auto runtime =
std::unique_ptr<nvinfer1::IRuntime>(nvinfer1::createInferRuntime(sample::gLogger.getTRTL
ogger()));
if (!runtime)
{
    std::cout << "runtime create failed" << std::endl;
    return -1;
}
```

2. 反序列化模型得到推理 Engine

```
// ========= 2. 反序列化生成 engine ==========
// 加载模型文件
auto plan = load_engine_file(engine_file);
// 反序列化生成 engine
auto mEngine =
  std::shared_ptr<nvinfer1::ICudaEngine>(runtime->deserializeCudaEngine(plan.data(),
plan.size()));
```

3. 创建执行上下文

```
// ======== 3. 创建执行上下文 context =========
auto context =
std::unique_ptr<nvinfer1::IExecutionContext>(mEngine->createExecutionContext());
```

4. 创建输入输出缓冲区管理器，这里我们使用的是 tensorrt sample code 中的 buffer 管理器，以方便我们进行内存的分配和 cpu gpu 之间的内存拷贝。

```
samplesCommon::BufferManager buffers(mEngine);
```

5. 我们读取视频文件，并逐帧读取图像，送入模型中，进行推理

```
auto cap = cv::VideoCapture(input_video_path);
while(cap.isOpened()) {
  cv::Mat frame;
  cap >> frame;
  if (frame.empty()) break;
  ...
```

6. 首先对输入图像进行预处理，这里我们使用 preprocess.cu 中的代码，其中实现了对输入图像处理的 gpu 加速（后续再进行讲解）。

```
// 输入预处理
process_input(frame, (float *)buffers.getDeviceBuffer(kInputTensorName));
```

7. 预处理完成后，我们调用推理 api executeV2，进行模型推理，并将模型输出拷贝到 cpu

```
context->executeV2(buffers.getDeviceBindings().data());
buffers.copyOutputToHost();
```

8. 最后我们从 buffer manager 中获取模型输出，并执行 nms，得到最后的检测框

```
// 获取模型输出
int32_t *num_det = (int32_t *)buffers.getHostBuffer(kOutNumDet); // 检测到的目标个数
int32_t *cls = (int32_t *)buffers.getHostBuffer(kOutDetCls);        // 检测到的目标类别
float *conf = (float *)buffers.getHostBuffer(kOutDetScores);        // 检测到的目标置信度
float *bbox = (float *)buffers.getHostBuffer(kOutDetBBoxes);        // 检测到的目标框
// 后处理
std::vector<Detection> bboxs;
```

```
yolo_nms(bboxs, num_det, cls, conf, bbox, kConfThresh, kNmsThresh);
```

9. 我们依次将检测框画到图像上，再打印对应的 fps 和推理时间。并显示图像

```
// 结束时间
auto end = std::chrono::high_resolution_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
auto time_str = std::to_string(elapsed) + "ms";
auto fps_str = std::to_string(1000 / elapsed) + "fps";
// 遍历检测结果
for (size_t j = 0; j < bboxs.size(); j++)
{
    cv::Rect r = get_rect(frame, bboxs[j].bbox);
    cv::rectangle(frame, r, cv::Scalar(0x27, 0xC1, 0x36), 2);
    cv::putText(frame, std::to_string((int)bboxs[j].class_id), cv::Point(r.x, r.y - 10), cv::FONT_HERSHEY_PLAIN, 1.2, cv::Scalar(0x27, 0xC1, 0x36), 2);
}
cv::putText(frame, time_str, cv::Point(50, 50), cv::FONT_HERSHEY_PLAIN, 1.2, cv::Scalar(0xFF, 0xFF, 0xFF), 2);
cv::putText(frame, fps_str, cv::Point(50, 100), cv::FONT_HERSHEY_PLAIN, 1.2, cv::Scalar(0xFF, 0xFF, 0xFF), 2);
cv::imshow("frame", frame);
```