MMYOLO

发布 *0.6.0*

MMYOLO Authors

2023 年 08 月 24 日

开启 MMYOLO 之旅

您可以在页面右上角切换中英文文档。

CHAPTER 1

概述

1.1 MMYOLO 介绍

MMYOLO 是一个基于 PyTorch 和 MMDetection 的 YOLO 系列算法开源工具箱，它是 OpenMMLab 项目的一部分。MMYOLO 定位为 YOLO 系列热门开源库以及工业应用核心库，其愿景图如下所示：

目前支持的任务如下：

- 目标检测
- 旋转框目标检测

目前支持的 YOLO 系列算法如下：

- YOLOv5
- YOLOX
- RTMDet
- RTMDet-Rotated
- YOLOv6
- YOLOv7
- PPYOLOE
- YOLOv8

目前支持的数据集如下：

- COCO Dataset
- VOC Dataset
- CrowdHuman Dataset
- DOTA 1.0 Dataset

MMYOLO 支持在 Linux、Windows、macOS 上运行，支持 PyTorch 1.7 及其以上版本运行。它具有如下三个特性：

- **统一便捷的算法评测**

  MMYOLO 统一了各类 YOLO 算法模块的实现，并提供了统一的评测流程，用户可以公平便捷地进行对比分析。

- **丰富的入门和进阶文档**

  MMYOLO 提供了从入门到部署到进阶和算法解析等一系列文档，方便不同用户快速上手和扩展。

- **模块化设计**

  MMYOLO 将框架解耦成不同的模块组件，通过组合不同的模块和训练测试策略，用户可以便捷地构建自定义模型。

1.2 本文档使用指南

MMYOLO 中将文档结构分成 6 个部分，对应不同需求的用户。

- **开启 MMYOLO 之旅**。本部分是第一次使用 MMYOLO 用户的必读文档，请全文仔细阅读
- **推荐专题**。本部分是 MMYOLO 中提供的以主题形式的精华文档，包括了 MMYOLO 中大量的特性等。强烈推荐使用 MMYOLO 的所有用户阅读
- **常用功能**。本部分提供了训练测试过程中用户经常会用到的各类常用功能，用户可以在用到时候再次查阅
- **实用工具**。本部分是 tools 下使用工具的汇总文档，便于大家能够快速的愉快使用 MMYOLO 中提供的各类脚本
- **基础和进阶教程**。本部分涉及到 MMYOLO 中的一些基本概念和进阶教程等，适合想详细了解 MMYOLO 设计思想和结构设计的用户
- **其他**。其余部分包括模型仓库、说明和接口文档等等

不同需求的用户可以按需选择你心怡的内容阅读。如果你对本文档有异议或者更好的优化办法，欢迎给 MMYOLO 提 PR ~，请参考*如何给 MMYOLO 贡献代码*

CHAPTER 2

依赖

下表为 MMYOLO 和 MMEngine, MMCV, MMDetection 依赖库的版本要求，请安装正确的版本以避免安装问题。

本节中，我们将演示如何用 PyTorch 准备一个环境。

MMYOLO 支持在 Linux，Windows 和 macOS 上运行。它的基本环境依赖为：

- Python 3.7+
- PyTorch 1.7+
- CUDA 9.2+
- GCC 5.4+

注解： 如果你对 PyTorch 有经验并且已经安装了它，你可以直接跳转到下一小节。否则，你可以按照下述步骤进行准备

步骤 0. 从 官方网站 下载并安装 Miniconda。

步骤 1. 创建并激活一个 conda 环境。

```
conda create -n mmyolo python=3.8 -y
conda activate mmyolo
```

步骤 2. 基于 PyTorch 官方说明 安装 PyTorch。

在 GPU 平台上：

```
conda install pytorch torchvision -c pytorch
```

在 CPU 平台上:

```
conda install pytorch torchvision cpuonly -c pytorch
```

步骤 3. 验证 PyTorch 安装

```
python -c "import torch; print(torch.__version__); print(torch.cuda.is_available())"
```

如果是在 GPU 平台上，那么会打印版本信息和 True 字符，否则打印版本信息和 False 字符。

CHAPTER 3

安装和验证

3.1 最佳实践

步骤 0. 使用 MIM 安装 MMEngine、MMCV 和 MMDetection 。

```
pip install -U openmim
mim install "mmengine>=0.6.0"
mim install "mmcv>=2.0.0rc4,<2.1.0"
mim install "mmdet>=3.0.0,<4.0.0"
```

如果你当前已经处于 mmyolo 工程目录下，则可以采用如下简化写法

```
cd mmyolo
pip install -U openmim
mim install -r requirements/mminstall.txt
```

注意：

a. 在 MMCV-v2.x 中，`mmcv-full` 改名为 `mmcv`，如果你想安装不包含 CUDA 算子精简版，可以通过 `mim install mmcv-lite>=2.0.0rc1` 来安装。

b. 如果使用 `albumentations`，我们建议使用 `pip install -r requirements/albu.txt` 或者 `pip install -U albumentations --no-binary qudida,albumentations` 进行安装。如果简单地使用 `pip install albumentations==1.0.1` 进行安装，则会同时安装 `opencv-python-headless`（即便已经安装了 `opencv-python` 也会再次安装）。我们建议在安装 albumentations 后检查环境，以确保没有同

时安装 opencv-python 和 opencv-python-headless，因为同时安装可能会导致一些问题。更多细节请参考 官方文档 。

步骤 1. 安装 MMYOLO

方案 1. 如果你基于 MMYOLO 框架开发自己的任务，建议从源码安装

```
git clone https://github.com/open-mmlab/mmyolo.git
cd mmyolo
# Install albumentations
mim install -r requirements/albu.txt
# Install MMYOLO
mim install -v -e .
# "-v" 指详细说明，或更多的输出
# "-e" 表示在可编辑模式下安装项目，因此对代码所做的任何本地修改都会生效，从而无需重新安装。
```

方案 2. 如果你将 MMYOLO 作为依赖或第三方 Python 包，使用 MIM 安装

```
mim install "mmyolo"
```

3.2 验证安装

为了验证 MMYOLO 是否安装正确，我们提供了一些示例代码来执行模型推理。

步骤 1. 我们需要下载配置文件和模型权重文件。

```
mim download mmyolo --config yolov5_s-v61_syncbn_fast_8xb16-300e_coco --dest .
```

下载将需要几秒钟或更长时间，这取决于你的网络环境。完成后，你会在当前文件夹中发现两个文件 yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py 和 yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth。

步骤 2. 推理验证

方案 1. 如果你通过源码安装的 MMYOLO，那么直接运行以下命令进行验证：

```
python demo/image_demo.py demo/demo.jpg \
                          yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
                          yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth

# 可选参数
# --out-dir ./output * 检测结果输出到指定目录下，默认为./output，当--show 参数存在时，不保存检测
结果
# --device cuda:0    * 使用的计算资源，包括 cuda，cpu 等，默认为 cuda:0
```

(下页继续)

(续上页)

```
# --show              * 使用该参数表示在屏幕上显示检测结果，默认为 False
# --score-thr 0.3     * 置信度阈值，默认为 0.3
```

运行结束后，在 output 文件夹中可以看到检测结果图像，图像中包含有网络预测的检测框。

支持输入类型包括

- 单张图片，支持 jpg, jpeg, png, ppm, bmp, pgm, tif, tiff, webp。
- 文件目录，会遍历文件目录下所有图片文件，并输出对应结果。
- 网址，会自动从对应网址下载图片，并输出结果。

方案 2. 如果你通过 MIM 安装的 MMYOLO，那么可以打开你的 Python 解析器，复制并粘贴以下代码：

```
from mmdet.apis import init_detector, inference_detector

config_file = 'yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py'
checkpoint_file = 'yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.
↪pth'
model = init_detector(config_file, checkpoint_file, device='cpu')  # or device='cuda:0
↪'
inference_detector(model, 'demo/demo.jpg')
```

你将会看到一个包含 DetDataSample 的列表，预测结果在 pred_instance 里，包含有预测框、预测分数和预测类别。

3.3 通过 Docker 使用 MMYOLO

我们提供了一个 Dockerfile 来构建一个镜像。请确保你的 docker 版本 >=19.03。

温馨提示：国内用户建议取消掉 Dockerfile 里面 Optional 后两行的注释，可以获得火箭一般的下载提速：

```
# (Optional)
RUN sed -i 's/http:\/\/archive.ubuntu.com\/ubuntu\//http:\/\/mirrors.aliyun.com\/
↪ubuntu\//g' /etc/apt/sources.list && \
    pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

构建命令：

```
# build an image with PyTorch 1.9, CUDA 11.1
# If you prefer other versions, just modified the Dockerfile
docker build -t mmyolo docker/
```

用以下命令运行 Docker 镜像：

```
export DATA_DIR=/path/to/your/dataset
docker run --gpus all --shm-size=8g -it -v ${DATA_DIR}:/mmyolo/data mmyolo
```

其余自定义安装流程请查看自定义安装

3.4 排除故障

如果你在安装过程中遇到一些问题，你可以在 GitHub 上 打开一个问题。

CHAPTER 4

15 分钟上手 MMYOLO 目标检测

目标检测任务是指给定一张图片，网络预测出图片中所包括的所有物体类别和对应的边界框

以我们提供的猫 cat 小数据集为例，带大家 15 分钟轻松上手 MMYOLO 目标检测。整个流程包含如下步骤：

- 环境安装
- 数据集准备
- 配置准备
- 模型训练
- 模型测试
- *EasyDeploy* 模型部署

本文以 YOLOv5-s 为例，其余 YOLO 系列算法的猫 cat 小数据集 demo 配置请查看对应的算法配置文件夹下。

4.1 环境安装

假设你已经提前安装好了 Conda，接下来安装 PyTorch

```
conda create -n mmyolo python=3.8 -y
conda activate mmyolo
# 如果你有 GPU
conda install pytorch torchvision -c pytorch
```

(下页继续)

(续上页)

```
# 如果你是 CPU
# conda install pytorch torchvision cpuonly -c pytorch
```

安装 MMYOLO 和依赖库

```
git clone https://github.com/open-mmlab/mmyolo.git
cd mmyolo
pip install -U openmim
mim install -r requirements/mminstall.txt
# Install albumentations
mim install -r requirements/albu.txt
# Install MMYOLO
mim install -v -e .
# "-v" 指详细说明，或更多的输出
# "-e" 表示在可编辑模式下安装项目，因此对代码所做的任何本地修改都会生效，从而无需重新安装。
```

注解： 温馨提醒：由于本仓库采用的是 OpenMMLab 2.0，请最好新建一个 conda 虚拟环境，防止和 OpenMMLab 1.0 已经安装的仓库冲突。

详细环境配置操作请查看安装和验证

4.2 数据集准备

Cat 数据集是一个包括 144 张图片的单类别数据集（本 cat 数据集由 @RangeKing 提供原始图片，由 @PeterH0323 进行数据清洗），包括了训练所需的标注信息。样例图片如下所示：

你只需执行如下命令即可下载并且直接用起来

```
python tools/misc/download_dataset.py --dataset-name cat --save-dir ./data/cat --
↪unzip --delete
```

数据集组织格式如下所示：

data 位于 mmyolo 工程目录下，`data/cat/annotations` 中存放的是 COCO 格式的标注，`data/cat/images` 中存放的是所有图片

4.3 配置准备

以 YOLOv5 算法为例，考虑到用户显存和内存有限，我们需要修改一些默认训练参数来让大家愉快的跑起来，核心需要修改的参数如下

- YOLOv5 是 Anchor-Based 类算法，不同的数据集需要自适应计算合适的 Anchor
- 默认配置是 8 卡，每张卡 batch size 为 16，现将其改成单卡，每张卡 batch size 为 12
- 默认训练 epoch 是 300，将其改成 40 epoch
- 由于数据集太小，我们选择固定 backbone 网络权重
- 原则上 batch size 改变后，学习率也需要进行线性缩放，但是实测发现不需要

具体操作为在 `configs/yolov5` 文件夹下新建 `yolov5_s-v61_fast_1xb12-40e_cat.py` 配置文件(为了方便大家直接使用，我们已经提供了该配置)，并把以下内容复制配置文件中。

```
# 基于该配置进行继承并重写部分配置
_base_ = 'yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py'

data_root = './data/cat/' # 数据集根路径
class_name = ('cat', ) # 数据集类别名称
num_classes = len(class_name) # 数据集类别数
# metainfo 必须要传给后面的 dataloader 配置，否则无效
# palette 是可视化时候对应类别的显示颜色
# palette 长度必须大于或等于 classes 长度
metainfo = dict(classes=class_name, palette=[(20, 220, 60)])

# 基于 tools/analysis_tools/optimize_anchors.py 自适应计算的 anchor
anchors = [
    [(68, 69), (154, 91), (143, 162)],  # P3/8
    [(242, 160), (189, 287), (391, 207)],  # P4/16
    [(353, 337), (539, 341), (443, 432)]  # P5/32
]
# 最大训练 40 epoch
max_epochs = 40
# bs 为 12
train_batch_size_per_gpu = 12
# dataloader 加载进程数
train_num_workers = 4

# 加载 COCO 预训练权重
load_from = 'https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_
↪8xb16-300e_coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.
↪pth'  # noqa
```

(下页继续)

(续上页)

```
model = dict(
    # 固定整个 backbone 权重，不进行训练
    backbone=dict(frozen_stages=4),
    bbox_head=dict(
        head_module=dict(num_classes=num_classes),
        prior_generator=dict(base_sizes=anchors)
    ))

train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        data_root=data_root,
        metainfo=metainfo,
        # 数据集标注文件 json 路径
        ann_file='annotations/trainval.json',
        # 数据集前缀
        data_prefix=dict(img='images/')))

val_dataloader = dict(
    dataset=dict(
        metainfo=metainfo,
        data_root=data_root,
        ann_file='annotations/test.json',
        data_prefix=dict(img='images/')))

test_dataloader = val_dataloader

_base_.optim_wrapper.optimizer.batch_size_per_gpu = train_batch_size_per_gpu

val_evaluator = dict(ann_file=data_root + 'annotations/test.json')
test_evaluator = val_evaluator

default_hooks = dict(
    # 每隔 10 个 epoch 保存一次权重，并且最多保存 2 个权重
    # 模型评估时候自动保存最佳模型
    checkpoint=dict(interval=10, max_keep_ckpts=2, save_best='auto'),
    # warmup_mim_iter 参数非常关键，因为 cat 数据集非常小，默认的最小 warmup_mim_iter 是 1000，
导致训练过程学习率偏小
    param_scheduler=dict(max_epochs=max_epochs, warmup_mim_iter=10),
    # 日志打印间隔为 5
    logger=dict(type='LoggerHook', interval=5))
# 评估间隔为 10
train_cfg = dict(max_epochs=max_epochs, val_interval=10)
```

以上配置从 yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py 中继承，并根据 cat 数据的特点更新了 data_root、metainfo、train_dataloader、val_dataloader、num_classes 等配置。

4.4 模型训练

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py
```

运行以上训练命令 work_dirs/yolov5_s-v61_fast_1xb12-40e_cat 文件夹会被自动生成，权重文件以及此次的训练配置文件将会保存在此文件夹中。在 1660 低端显卡上，整个训练过程大概需要 8 分钟。

在 test.json 上性能如下所示：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.631
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.909
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.747
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.631
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.627
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.703
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.703
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.703
```

上述性能是通过 COCO API 打印，其中 -1 表示不存在对于尺度的物体。根据 COCO 定义的规则，Cat 数据集里面全部是大物体，不存在小和中等规模物体。

4.4.1 一些注意事项

在训练过程中会打印如下两个关键警告：

- You are using YOLOv5Head with num_classes == 1. The loss_cls will be 0. This is a normal phenomenon.
- The model and loaded state dict do not match exactly

这两个警告都不会对性能有任何影响。第一个警告是说明由于当前训练的类别数是 1，根据 YOLOv5 算法的社区，分类分支的 loss 始终是 0，这是正常现象。第二个警告是因为目前是采用微调模式进行训练，我们加载了 COCO 80 个类的预训练权重，这会导致最后的 Head 模块卷积通道数不对应，从而导致这部分权重无法加载，这也是正常现象。

4.4.2 中断后恢复训练

如果训练中途停止，可以在训练命令最后加上 `--resume`，程序会自动从 `work_dirs` 中加载最新的权重文件恢复训练。

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py --resume
```

4.4.3 节省显存策略

上述配置大概需要 3.0G 显存，如果你的显存不够，可以考虑开启混合精度训练

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py --amp
```

4.4.4 训练可视化

MMYOLO 目前支持本地、TensorBoard 以及 WandB 等多种后端可视化，默认是采用本地可视化方式，你可以切换为 WandB 等实时可视化训练过程中各类指标。

1 WandB 可视化使用

WandB 官网注册并在 https://wandb.ai/settings 获取到 WandB 的 API Keys。

```
pip install wandb
# 运行了 wandb login 后输入上文中获取到的 API Keys ，便登录成功。
wandb login
```

在 `configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py` 配置文件最后添加 WandB 配置

```
visualizer = dict(vis_backends = [dict(type='LocalVisBackend'), dict(type=
↪'WandbVisBackend')])
```

重新运行训练命令便可以在命令行中提示的网页链接中看到 loss、学习率和 coco/bbox_mAP 等数据可视化了。

```
python tools/train.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py
```

2 Tensorboard 可视化使用

安装 Tensorboard 依赖

```
pip install tensorboard
```

同上述在配置文件 `configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py` 配置的最后添加 tensorboard 配置

```
visualizer = dict(vis_backends=[dict(type='LocalVisBackend'), dict(type=
↪'TensorboardVisBackend')])
```

重新运行训练命令后，Tensorboard 文件会生成在可视化文件夹 `work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/{timestamp}/vis_data` 下，运行下面的命令便可以在网页链接使用 Tensorboard 查看 loss、学习率和 coco/bbox_mAP 等可视化数据了：

```
tensorboard --logdir=work_dirs/yolov5_s-v61_fast_1xb12-40e_cat
```

4.5 模型测试

```
python tools/test.py configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                     work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
                     --show-dir show_results
```

运行以上测试命令，你不仅可以得到**模型训练**部分所打印的 AP 性能，还可以将推理结果图片自动保存至 `work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/{timestamp}/show_results` 文件夹中。下面为其中一张结果图片，左图为实际标注，右图为模型推理结果。

如果你使用了 `WandbVisBackend` 或者 `TensorboardVisBackend`，则还可以在浏览器窗口可视化模型推理结果。

4.6 特征图相关可视化

MMYOLO 中提供了特征图相关可视化脚本，用于分析当前模型训练效果。详细使用流程请参考特征图可视化

由于 `test_pipeline` 直接可视化会存在偏差，故将需要 `configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py` 中 `test_pipeline`

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
```

(下页继续)

(续上页)

```
        backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

修改为如下配置：

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # 删除␣
↪YOLOv5KeepRatioResize, 将 LetterResize 修改成 mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))  # 删除 pad_param
]
```

我们选择 data/cat/images/IMG_20221020_112705.jpg 图片作为例子，可视化 YOLOv5 backbone 和 neck 层的输出特征图。

1. 可视化 YOLOv5 backbone 输出的 3 个通道

```
python demo/featmap_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                                configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                                work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.
↪pth \
                                --target-layers backbone \
                                --channel-reduction squeeze_mean
```

结果会保存到当前路径的 output 文件夹下。上图中绘制的 3 个输出特征图对应大中小输出特征图。由于本次训练的 backbone 实际上没有参与训练，从上图可以看到，大物体 cat 是在小特征图进行预测，这符合目标检测分层检测思想。

2. 可视化 YOLOv5 neck 输出的 3 个通道

```
python demo/featmap_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                                configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                                work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.
↪pth \
                                --target-layers neck \
                                --channel-reduction squeeze_mean
```

从上图可以看出，由于 neck 是参与训练的，并且由于我们重新设置了 anchor, 强行让 3 个输出特征图都拟合同一个尺度的物体，导致 neck 输出的 3 个图类似，破坏了 backbone 原先的预训练分布。同时也可以看出 40 epoch 训练上述数据集是不够的，特征图效果不佳。

3. Grad-Based CAM 可视化

基于上述特征图可视化效果，我们可以分析特征层 bbox 级别的 Grad CAM。

安装 `grad-cam` 依赖：

```
pip install "grad-cam"
```

(a) 查看 neck 输出的最小输出特征图的 Grad CAM

```
python demo/boxam_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                              configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                              work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.
↪pth \
                              --target-layer neck.out_layers[2]
```

(b) 查看 neck 输出的中等输出特征图的 Grad CAM

```
python demo/boxam_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                              configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                              work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.
↪pth \
                              --target-layer neck.out_layers[1]
```

(c) 查看 neck 输出的最大输出特征图的 Grad CAM

```
python demo/boxam_vis_demo.py data/cat/images/IMG_20221020_112705.jpg \
                              configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
                              work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.
↪pth \
                              --target-layer neck.out_layers[0]
```

4.7 EasyDeploy 模型部署

此处我们将通过 MMYOLO 的 EasyDeploy 来演示模型的转换部署和基本推理。

首先需要在当前 MMYOLO 的虚拟环境中按照 EasyDeploy 的 基本文档对照自己的设备安装好所需的各个库。

```
pip install onnx onnxruntime
pip install onnx-simplifier # 如果需要使用 simplify 功能需要安装
pip install tensorrt        # 如果有 GPU 环境并且需要输出 TensorRT 模型需要继续执行
```

完成安装后就可以用以下命令对已经训练好的针对 cat 数据集的模型一键转换部署，当前设备的 ONNX 版本为 1.13.0，TensorRT 版本为 8.5.3.1，故可保持 `--opset` 为 11，其余各项参数的具体含义和参数值需要对照使用的 config 文件进行调整。此处我们先导出 CPU 版本的 ONNX 模型，`--backend` 为 ONNXRUNTIME。

```
python projects/easydeploy/tools/export_onnx.py \
    configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
    --work-dir work_dirs/yolov5_s-v61_fast_1xb12-40e_cat \
    --img-size 640 640 \
    --batch 1 \
    --device cpu \
    --simplify \
    --opset 11 \
    --backend ONNXRUNTIME \
    --pre-topk 1000 \
    --keep-topk 100 \
    --iou-threshold 0.65 \
    --score-threshold 0.25
```

成功运行后就可以在 `work-dir` 下得到转换后的 ONNX 模型，默认使用 `end2end.onnx` 命名。

接下来我们使用此 `end2end.onnx` 模型来进行一个基本的图片推理:

```
python projects/easydeploy/tools/image-demo.py \
    data/cat/images/IMG_20210728_205117.jpg \
    configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/end2end.onnx \
    --device cpu
```

成功完成推理后会在默认的 MMYOLO 根目录下的 `output` 文件夹生成推理结果图，如果想直观看到结果而不需要保存，可以在上述命令结尾加上 `--show` ，为了方便展示，下图是生成结果的截取部分。

我们继续转换对应 TensorRT 的 engine 文件，因为 TensorRT 需要对应当前环境以及部署使用的版本进行，所以一定要确认导出参数，这里我们导出对应 TensorRT8 版本的文件，`--backend` 为 2。

```
python projects/easydeploy/tools/export.py \
    configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/epoch_40.pth \
    --work-dir work_dirs/yolov5_s-v61_fast_1xb12-40e_cat \
    --img-size 640 640 \
    --batch 1 \
    --device cuda:0 \
    --simplify \
    --opset 11 \
    --backend 2 \
    --pre-topk 1000 \
    --keep-topk 100 \
    --iou-threshold 0.65 \
    --score-threshold 0.25
```

成功执行后得到的 end2end.onnx 就是对应 TensorRT8 部署需要的 ONNX 文件，我们使用这个文件完成 TensorRT engine 的转换。

```
python projects/easydeploy/tools/build_engine.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/end2end.onnx \
    --img-size 640 640 \
    --device cuda:0
```

成功执行后会在 work-dir 下生成 end2end.engine 文件：

```
work_dirs/yolov5_s-v61_fast_1xb12-40e_cat
├── 202302XX_XXXXXX
│   ├── 202302XX_XXXXXX.log
│   └── vis_data
│       ├── 202302XX_XXXXXX.json
│       ├── config.py
│       └── scalars.json
├── best_coco
│   └── bbox_mAP_epoch_40.pth
├── end2end.engine
├── end2end.onnx
├── epoch_30.pth
├── epoch_40.pth
├── last_checkpoint
└── yolov5_s-v61_fast_1xb12-40e_cat.py
```

我们继续使用 image-demo.py 进行图片推理：

```
python projects/easydeploy/tools/image-demo.py \
```

(下页继续)

(续上页)

```
    data/cat/images/IMG_20210728_205312.jpg \
    configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py \
    work_dirs/yolov5_s-v61_fast_1xb12-40e_cat/end2end.engine \
    --device cuda:0
```

此处依旧选择在 output 下保存推理结果而非直接显示结果，同样为了方便展示，下图是生成结果的截取部分。

这样我们就完成了将训练完成的模型进行转换部署并且检查推理结果的工作。至此本教程结束。

以上完整内容可以查看 15_minutes_object_detection.ipynb。如果你在训练或者测试过程中碰到问题，请先查看常见错误排除步骤，如果依然无法解决欢迎提 issue。

CHAPTER 5

15 分钟上手 MMYOLO 旋转框目标检测

TODO

CHAPTER 6

15 分钟上手 MMYOLO 实例分割

实例分割是计算机视觉中的一个任务，旨在将图像中的每个对象都分割出来，并为每个对象分配一个唯一的标识符。与语义分割不同，实例分割不仅分割出图像中的不同类别，还将同一类别的不同实例分开。

以可供下载的气球 balloon 小数据集为例，带大家 15 分钟轻松上手 MMYOLO 实例分割。整个流程包含如下步骤：

- 环境安装
- 数据集准备
- 配置准备
- 模型训练
- 模型测试
- *EasyDeploy* 模型部署

本文以 YOLOv5-s 为例，其余 YOLO 系列算法的气球 balloon 小数据集 demo 配置请查看对应的算法配置文件夹下。

6.1 环境安装

假设你已经提前安装好了 Conda，接下来安装 PyTorch

```
conda create -n mmyolo python=3.8 -y
conda activate mmyolo
# 如果你有 GPU
conda install pytorch torchvision -c pytorch
# 如果你是 CPU
# conda install pytorch torchvision cpuonly -c pytorch
```

安装 MMYOLO 和依赖库

```
git clone https://github.com/open-mmlab/mmyolo.git
cd mmyolo
pip install -U openmim
mim install -r requirements/mminstall.txt
# Install albumentations
mim install -r requirements/albu.txt
# Install MMYOLO
mim install -v -e .
# "-v" 指详细说明，或更多的输出
# "-e" 表示在可编辑模式下安装项目，因此对代码所做的任何本地修改都会生效，从而无需重新安装。
```

注解： 温馨提醒：由于本仓库采用的是 OpenMMLab 2.0，请最好新建一个 conda 虚拟环境，防止和 OpenMMLab 1.0 已经安装的仓库冲突。

详细环境配置操作请查看安装和验证

6.2 数据集准备

Balloon 数据集是一个包括 74 张图片的单类别数据集，包括了训练所需的标注信息。样例图片如下所示：

你只需执行如下命令即可下载并且直接用起来

```
python tools/misc/download_dataset.py --dataset-name balloon --save-dir ./data/
↪balloon --unzip --delete
python ./tools/dataset_converters/balloon2coco.py
```

data 位于 mmyolo 工程目录下，`train.json`, `val.json` 中存放的是 COCO 格式的标注，`data/balloon/train`, `data/balloon/val` 中存放的是所有图片

6.3 配置准备

以 YOLOv5 算法为例，考虑到用户显存和内存有限，我们需要修改一些默认训练参数来让大家愉快的跑起来，核心需要修改的参数如下

- YOLOv5 是 Anchor-Based 类算法，不同的数据集需要自适应计算合适的 Anchor
- 默认配置是 8 卡，每张卡 batch size 为 16，现将其改成单卡，每张卡 batch size 为 4
- 原则上 batch size 改变后，学习率也需要进行线性缩放，但是实测发现不需要

具体操作为在 `configs/yolov5/ins_seg` 文件夹下新建 `yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon.py` 配置文件 (为了方便大家直接使用，我们已经提供了该配置)，并把以下内容复制配置文件中。

```
_base_ = './yolov5_ins_s-v61_syncbn_fast_8xb16-300e_coco_instance.py'  # noqa

data_root = 'data/balloon/'
# 训练集标注路径
train_ann_file = 'train.json'
train_data_prefix = 'train/'  # 训练集图片路径
# 测试集标注路径
val_ann_file = 'val.json'
val_data_prefix = 'val/'  # 验证集图片路径
metainfo = {
    'classes': ('balloon', ), # 数据集类别名称
    'palette': [
        (220, 20, 60),
    ]
}
num_classes = 1
# 批处理大小 batch size 设置为 4
train_batch_size_per_gpu = 4
# dataloader 加载进程数
train_num_workers = 2
log_interval = 1
######################
train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        data_root=data_root,
        metainfo=metainfo,
        data_prefix=dict(img=train_data_prefix),
        ann_file=train_ann_file))
val_dataloader = dict(
    dataset=dict(
```

(下页继续)

(续上页)

```
        data_root=data_root,
        metainfo=metainfo,
        data_prefix=dict(img=val_data_prefix),
        ann_file=val_ann_file))
test_dataloader = val_dataloader
val_evaluator = dict(ann_file=data_root + val_ann_file)
test_evaluator = val_evaluator
default_hooks = dict(logger=dict(interval=log_interval))
######################

model = dict(bbox_head=dict(head_module=dict(num_classes=num_classes)))
```

以上配置从 yolov5_ins_s-v61_syncbn_fast_8xb16-300e_coco_instance.py 中继承，并根据 balloon 数据的特点更新了 data_root、metainfo、train_dataloader、val_dataloader、num_classes 等配置。

6.4 模型训练

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py
```

运行以上训练命令 work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance 文件夹会被自动生成，权重文件以及此次的训练配置文件将会保存在此文件夹中。在 1660 低端显卡上，整个训练过程大概需要 30 分钟。

在 val.json 上性能如下所示：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.330
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.509
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.317
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.103
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.417
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.150
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.396
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.454
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.317
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.525
```

上述性能是通过 COCO API 打印，其中 -1 表示不存在对于尺度的物体。

6.4.1 一些注意事项

在训练过程中会打印如下关键警告：

- You are using `YOLOv5Head` with num_classes == 1. The loss_cls will be 0. This is a normal phenomenon.

这个警告都不会对性能有任何影响。第一个警告是说明由于当前训练的类别数是 1，根据 YOLOv5 算法的社区，分类分支的 loss 始终是 0，这是正常现象。

6.4.2 中断后恢复训练

如果训练中途停止，可以在训练命令最后加上 `--resume`，程序会自动从 `work_dirs` 中加载最新的权重文件恢复训练。

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py --resume
```

6.4.3 节省显存策略

上述配置大概需要 1.0G 显存，如果你的显存不够，可以考虑开启混合精度训练

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py --amp
```

6.4.4 训练可视化

MMYOLO 目前支持本地、TensorBoard 以及 WandB 等多种后端可视化，默认是采用本地可视化方式，你可以切换为 WandB 等实时可视化训练过程中各类指标。

1 WandB 可视化使用

WandB 官网注册并在 https://wandb.ai/settings 获取到 WandB 的 API Keys。

```
pip install wandb
# 运行了 wandb login 后输入上文中获取到的 API Keys ，便登录成功。
wandb login
```

在`configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.py` 配置文件最后添加 WandB 配置

```
visualizer = dict(vis_backends = [dict(type='LocalVisBackend'), dict(type=
↪'WandbVisBackend')])
```

重新运行训练命令便可以在命令行中提示的网页链接中看到 loss、学习率和 coco/bbox_mAP 等数据可视化了。

```
python tools/train.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py
```

2 Tensorboard 可视化使用

安装 Tensorboard 环境

```
pip install tensorboard
```

同上述在配置文件 `configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instan` `py` 配置的最后添加 `tensorboard` 配置

```
visualizer = dict(vis_backends=[dict(type='LocalVisBackend'), dict(type=
↪'TensorboardVisBackend')])
```

重新运行训练命令后，Tensorboard 文件会生成在可视化文件夹 `work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/{timestamp}/vis_data` 下，运行下面的命令便可以在网页链接使用 Tensorboard 查看 loss、学习率和 coco/bbox_mAP 等可视化数据了：

```
tensorboard --logdir=work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_
↪instance
```

6.5 模型测试

```
python tools/test.py configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_
↪balloon_instance.py \
                    work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_
↪instance/best_coco_bbox_mAP_epoch_300.pth \
                    --show-dir show_results
```

运行以上测试命令，你不仅可以得到**模型训练**部分所打印的 AP 性能，还可以将推理结果图片自动保存至 `work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/{timestamp}/show_results` 文件夹中。下面为其中一张结果图片，左图为实际标注，右图为模型推理结果。

如果你使用了 `WandbVisBackend` 或者 `TensorboardVisBackend`，则还可以在浏览器窗口可视化模型推理结果。

6.6 特征图相关可视化

MMYOLO 中提供了特征图相关可视化脚本，用于分析当前模型训练效果。详细使用流程请参考特征图可视化

由于 test_pipeline 直接可视化会存在偏差，故将需要 configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py 中 test_pipeline

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

修改为如下配置:

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # 删除␣
↪YOLOv5KeepRatioResize, 将 LetterResize 修改成 mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))  # 删除 pad_param
]
```

我们选择 data/balloon/train/3927754171_9011487133_b.jpg 图片作为例子，可视化 YOLOv5 backbone 和 neck 层的输出特征图。

```
python demo/featmap_vis_demo.py data/balloon/train/3927754171_9011487133_b.jpg \
    configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.
↪py \
```

(下页继续)

(续上页)

```
    work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/best_coco_bbox_
↪mAP_epoch_300.pth \ --target-layers backbone \
    --channel-reduction squeeze_mean
```

结果会保存到当前路径的 output 文件夹下。上图中绘制的 3 个输出特征图对应大中小输出特征图。

2. 可视化 YOLOv5 neck 输出的 3 个通道

```
python demo/featmap_vis_demo.py data/balloon/train/3927754171_9011487133_b.jpg \
    configs/yolov5/ins_seg/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance.
↪py \
    work_dirs/yolov5_ins_s-v61_syncbn_fast_8xb16-300e_balloon_instance/best_coco_bbox_
↪mAP_epoch_300.pth \ --target-layers neck \
    --channel-reduction squeeze_mean
```

3. Grad-Based CAM 可视化

TODO

6.7 EasyDeploy 模型部署

TODO

至此本教程结束。

以上完整内容可以查看 15_minutes_instance_segmentation.ipynb。如果你在训练或者测试过程中碰到问题，请先查看常见错误排除步骤，如果依然无法解决欢迎提 issue。

CHAPTER 7

中文解读资源汇总

本文汇总了 MMYOLO 或相关的 OpenMMLab 解读的部分文章（更多文章和视频见 OpenMMLabCourse ），如果您有推荐的文章（不一定是 OpenMMLab 发布的文章，可以是自己写的文章），非常欢迎提 Pull Request 添加到这里。

7.1 MMYOLO 解读文章和资源

7.1.1 脚本命令速查表

你可以点击链接，下载高清版 PDF 文件。

7.1.2 文章

- 社区协作，简洁易用，快来开箱新一代 YOLO 系列开源库
- MMYOLO 社区倾情贡献，RTMDet 原理社区开发者解读来啦！
- MMYOLO 自定义数据集从标注到部署保姆级教程
- 满足一切需求的 MMYOLO 可视化：测试过程可视化
- MMYOLO 想你所想: 训练过程可视化
- YOLOv8 深度详解！一文看懂，快速上手
- 玩转 MMYOLO 基础类第一期：配置文件太复杂？继承用法看不懂？配置全解读来了

- 玩转 MMYOLO 工具类第一期：特征图可视化
- 玩转 MMYOLO 实用类第一期：源码阅读和调试「必备」技巧文档
- 玩转 MMYOLO 基础类第二期：工程文件结构简析
- 玩转 MMYOLO 实用类第二期：10 分钟换遍主干网络文档

7.1.3 视频

工具类

基础类

实用类

源码解读类

演示类

7.2 MMDetection 解读文章和资源

7.2.1 文章

- MMDetection 3.0：目标检测新基准与前沿
- 目标检测、实例分割、旋转框样样精通！详解高性能检测算法 RTMDet
- MMDetection 支持数据增强神器 Simple Copy Paste 全过程

7.2.2 知乎问答和资源

- 深度学习科研，如何高效进行代码和实验管理?
- 深度学习方面的科研工作中的实验代码有什么规范和写作技巧？如何妥善管理实验数据?
- COCO 数据集上 1x 模式下为什么不采用多尺度训练?
- MMDetection 中 SOTA 论文源码中将训练过程中 BN 层的 eval 打开?
- 基于 PyTorch 的 MMDetection 中训练的随机性来自何处?

7.3 MMEngine 解读文章和资源

- 从 MMCV 到 MMEngine，架构升级，体验升级！

7.4 MMCV 解读文章和资源

- MMCV 全新升级，新增超全数据变换功能，还有两大变化
- 手把手教你如何高效地在 MMCV 中贡献算子

7.5 PyTorch 解读文章和资源

- PyTorch1.11 亮点一览：TorchData、functorch、DDP 静态图
- PyTorch1.12 亮点一览：DataPipe + TorchArrow 新的数据加载与处理范式
- PyTorch 源码解读之 nn.Module：核心网络模块接口详解
- PyTorch 源码解读之 torch.autograd：梯度计算详解
- PyTorch 源码解读之 torch.utils.data：解析数据处理全流程
- PyTorch 源码解读之 torch.optim：优化算法接口详解
- PyTorch 源码解读之 DP & DDP：模型并行和分布式训练解析
- PyTorch 源码解读之 BN & SyncBN：BN 与多卡同步 BN 详解
- PyTorch 源码解读之 torch.cuda.amp: 自动混合精度详解
- PyTorch 源码解读之 cpp_extension：揭秘 C++/CUDA 算子实现和调用全流程
- PyTorch 源码解读之即时编译篇
- PyTorch 源码解读之分布式训练了解一下?
- PyTorch 源码解读之 torch.serialization & torch.hub

7.6 其他

- Type Hints 入门教程，让代码更加规范整洁

CHAPTER 8

如何给 MMYOLO 贡献代码

欢迎加入 MMYOLO 社区，我们致力于打造最前沿的计算机视觉基础库，我们欢迎任何类型的贡献，包括但不限于

修复错误

修复代码实现错误的步骤如下：

1. 如果提交的代码改动较大，建议先提交 issue，并正确描述 issue 的现象、原因和复现方式，讨论后确认修复方案。
2. 修复错误并补充相应的单元测试，提交拉取请求。

新增功能或组件

1. 如果新功能或模块涉及较大的代码改动，建议先提交 issue，确认功能的必要性。
2. 实现新增功能并添单元测试，提交拉取请求。

文档补充

修复文档可以直接提交拉取请求

添加文档或将文档翻译成其他语言步骤如下

1. 提交 issue，确认添加文档的必要性。
2. 添加文档，提交拉取请求。

8.1 准备工作

拉取请求工作的命令都是用 Git 去实现的，该章节详细描述 Git 配置以及与 GitHub 绑定

8.1.1 1. Git 配置

首先，确认电脑是否安装了 Git。Linux 系统和 macOS 系统，一般默认安装 Git，如未安装可在 Git-Downloads 下载。

```
# 在命令提示符（cmd）或终端下输入以下命令，查看 Git 版本
git --version
```

其次，检测自己 Git Config 是否配置

```
# 在命令提示符（cmd）或终端下输入以下命令，查看 Git Config 是否配置
git config --global --list
```

若 user.name 和 user.email 为空，则输入以下命令进行配置。

```
git config --global user.name " 这里换上你的用户名"
git config --global user.email " 这里换上你的邮箱"
```

最后，在 git bash 或者 终端中，输入以下命令生成密钥文件。生成成功后，会在用户目录下出现 .ssh 文件，其中 id_rsa.pub 是公钥文件。

```
# useremail 是 GitHub 的邮箱
ssh-keygen -t rsa -C "useremail"
```

8.1.2 2. GitHub 绑定

首先，用记事本打开 id_rsa.pub 公钥文件，并复制里面全部内容。

其次，登录 GitHub 账户找到下图位置进行设置。

点击 New SSH key 新增一个 SSH keys，将刚才复制的内容粘贴到下图所示的 Key 中，Title 可以写设备名称，最后确认即可。

最后，在 git bash 或者 终端中输入以下命令，验证 SSH 是否与 GitHub 账户匹配。如果匹配，输入 yes 就成功啦 ~

```
ssh -T git@github.com
```

8.2 拉取请求工作流

如果你对拉取请求不了解，没关系，接下来的内容将会从零开始，一步一步地指引你如何创建一个拉取请求。如果你想深入了解拉取请求的开发模式，可以参考 github 官方文档

8.2.1 1. 复刻仓库

当你第一次提交拉取请求时，先复刻 OpenMMLab 原代码库，点击 GitHub 页面右上角的 **Fork** 按钮，复刻后的代码库将会出现在你的 GitHub 个人主页下。

将代码克隆到本地

```
git clone git@github.com:{username}/mmyolo.git
```

进入项目并添加原代码库为上游代码库

```
cd mmyolo
git remote add upstream git@github.com:open-mmlab/mmyolo
```

检查 remote 是否添加成功，在终端输入 `git remote -v`

```
origin        git@github.com:{username}/mmyolo.git (fetch)
origin        git@github.com:{username}/mmyolo.git (push)
upstream        git@github.com:open-mmlab/mmyolo (fetch)
upstream        git@github.com:open-mmlab/mmyolo (push)
```

注解： 这里对 origin 和 upstream 进行一个简单的介绍，当我们使用 git clone 来克隆代码时，会默认创建一个 origin 的 remote，它指向我们克隆的代码库地址，而 upstream 则是我们自己添加的，用来指向原始代码库地址。当然如果你不喜欢他叫 upstream，也可以自己修改，比如叫 open-mmlab。我们通常向 origin 提交代码（即 fork 下来的远程仓库），然后向 upstream 提交一个 pull request。如果提交的代码和最新的代码发生冲突，再从 upstream 拉取最新的代码，和本地分支解决冲突，再提交到 origin。

8.2.2 2. 配置 pre-commit

在本地开发环境中，我们使用 pre-commit 来检查代码风格，以确保代码风格的统一。在提交代码，需要先安装 pre-commit（需要在 MMYOLO 目录下执行）：

```
pip install -U pre-commit
pre-commit install
```

检查 pre-commit 是否配置成功，并安装 `.pre-commit-config.yaml` 中的钩子：

```
pre-commit run --all-files
```

注解： 如果你是中国用户，由于网络原因，可能会出现安装失败的情况，这时可以使用国内源

pre-commit install -c .pre-commit-config-zh-cn.yaml

pre-commit run –all-files -c .pre-commit-config-zh-cn.yaml

如果安装过程被中断，可以重复执行 `pre-commit run ...` 继续安装。

如果提交的代码不符合代码风格规范，pre-commit 会发出警告，并自动修复部分错误。

如果我们想临时绕开 pre-commit 的检查提交一次代码，可以在 `git commit` 时加上 `--no-verify`（需要保证最后推送至远程仓库的代码能够通过 pre-commit 检查）。

```
git commit -m "xxx" --no-verify
```

8.2.3 3. 创建开发分支

安装完 pre-commit 之后，我们需要基于 dev 创建开发分支，建议的分支命名规则为 `username/pr_name`。

```
git checkout -b yhc/refactor_contributing_doc
```

在后续的开发中，如果本地仓库的 dev 分支落后于 upstream 的 dev 分支，我们需要先拉取 upstream 的代码进行同步，再执行上面的命令

```
git pull upstream dev
```

8.2.4 4. 提交代码并在本地通过单元测试

- MMYOLO 引入了 mypy 来做静态类型检查，以增加代码的鲁棒性。因此我们在提交代码时，需要补充 Type Hints。具体规则可以参考教程。
- 提交的代码同样需要通过单元测试

```
# 通过全量单元测试
pytest tests

# 我们需要保证提交的代码能够通过修改模块的单元测试，以 yolov5_coco dataset 为例
pytest tests/test_datasets/test_yolov5_coco.py
```

  如果你由于缺少依赖无法运行修改模块的单元测试，可以参考指引-单元测试

- 如果修改/添加了文档，参考指引确认文档渲染正常。

8.2.5 5. 推送代码到远程

代码通过单元测试和 pre-commit 检查后，将代码推送到远程仓库，如果是第一次推送，可以在 `git push` 后加上 `-u` 参数以关联远程分支

```
git push -u origin {branch_name}
```

这样下次就可以直接使用 `git push` 命令推送代码了，而无需指定分支和远程仓库。

8.2.6 6. 提交拉取请求（PR）

(1) 在 GitHub 的 Pull request 界面创建拉取请求

(2) 根据指引修改 PR 描述，以便于其他开发者更好地理解你的修改

注解： 注意在 PR branch 左侧的 base 需要修改为 dev 分支

描述规范详见拉取请求规范

注意事项

(a) PR 描述应该包含修改理由、修改内容以及修改后带来的影响，并关联相关 Issue（具体方式见文档）

(b) 如果是第一次为 OpenMMLab 做贡献，需要签署 CLA

(c) 检查提交的 PR 是否通过 CI（集成测试）

MMYOLO 会在 Linux 上，基于不同版本的 Python、PyTorch 对提交的代码进行单元测试，以保证代码的正确性，如果有任何一个没有通过，我们可点击上图中的 `Details` 来查看具体的测试信息，以便于我们修改代码。

(3) 如果 PR 通过了 CI，那么就可以等待其他开发者的 review，并根据 reviewer 的意见，修改代码，并重复 4-*5* 步骤，直到 reviewer 同意合入 PR。

所有 reviewer 同意合入 PR 后，我们会尽快将 PR 合并到 dev 分支。

8.2.7 7. 解决冲突

随着时间的推移，我们的代码库会不断更新，这时候，如果你的 PR 与 dev 分支存在冲突，你需要解决冲突，解决冲突的方式有两种：

```
git fetch --all --prune
git rebase upstream/dev
```

或者

```
git fetch --all --prune
git merge upstream/dev
```

如果你非常善于处理冲突，那么可以使用 rebase 的方式来解决冲突，因为这能够保证你的 commit log 的整洁。如果你不太熟悉 `rebase` 的使用，那么可以使用 `merge` 的方式来解决冲突。

8.3 指引

8.3.1 单元测试

在提交修复代码错误或新增特性的拉取请求时，我们应该尽可能的让单元测试覆盖所有提交的代码，计算单元测试覆盖率的方法如下

```
python -m coverage run -m pytest /path/to/test_file
python -m coverage html
# check file in htmlcov/index.html
```

8.3.2 文档渲染

在提交修复代码错误或新增特性的拉取请求时，可能会需要修改/新增模块的 docstring。我们需要确认渲染后的文档样式是正确的。本地生成渲染后的文档的方法如下

```
pip install -r requirements/docs.txt
cd docs/zh_cn/
# or docs/en
make html
# check file in ./docs/zh_cn/_build/html/index.html
```

8.4 代码风格

8.4.1 Python

PEP8 作为 OpenMMLab 算法库首选的代码规范，我们使用以下工具检查和格式化代码

- flake8：Python 官方发布的代码规范检查工具，是多个检查工具的封装
- isort：自动调整模块导入顺序的工具
- yapf：Google 发布的代码规范检查工具
- codespell：检查单词拼写是否有误

- mdformat：检查 markdown 文件的工具
- docformatter：格式化 docstring 的工具

yapf 和 isort 的配置可以在 setup.cfg 找到

通过配置 pre-commit hook ，我们可以在提交代码时自动检查和格式化 `flake8`、`yapf`、`isort`、`trailing whitespaces`、`markdown files`，修复 `end-of-files`、`double-quoted-strings`、`python-encoding-pragma`、`mixed-line-ending`，调整 `requirments.txt` 的包顺序。pre-commit 钩子的配置可以在 .pre-commit-config 找到。

pre-commit 具体的安装使用方式见拉取请求。

更具体的规范请参考*OpenMMLab 代码规范*。

8.4.2 C++ and CUDA

C++ 和 CUDA 的代码规范遵从 Google C++ Style Guide

8.5 拉取请求规范

1. 使用 pre-commit hook，尽量减少代码风格相关问题
2. 一个拉取请求对应一个短期分支
3. 粒度要细，一个拉取请求只做一件事情，避免超大的拉取请求
   - Bad：实现 Faster R-CNN
   - Acceptable：给 Faster R-CNN 添加一个 box head
   - Good：给 box head 增加一个参数来支持自定义的 conv 层数
4. 每次 Commit 时需要提供清晰且有意义 commit 信息
5. 提供清晰且有意义的拉取请求描述
   - 标题写明白任务名称，一般格式:[Prefix] Short description of the pull request (Suffix)
   - prefix：新增功能 [Feature], 修 bug [Fix], 文档相关 [Docs], 开发中 [WIP] (暂时不会被 review)
   - 描述里介绍拉取请求的主要修改内容，结果，以及对其他部分的影响, 参考拉取请求模板
   - 关联相关的议题 (issue) 和其他拉取请求
6. 如果引入了其他三方库，或借鉴了三方库的代码，请确认他们的许可证和 mmyolo 兼容，并在借鉴的代码上补充 `This code is inspired from http://`

CHAPTER 9

训练和测试技巧

MMYOLO 中已经支持了大部分 YOLO 系列目标检测相关算法。不同算法可能涉及到一些实用技巧。本章节将基于所实现的目标检测算法，详细描述 MMYOLO 中已经支持的常用的训练和测试技巧。

9.1 训练技巧

9.1.1 提升检测性能

1 开启多尺度训练

在目标检测领域，多尺度训练是一个非常常用的技巧，但是在 YOLO 中大部分模型的训练输入都是单尺度的 640x640，原因有两个方面：

1. 单尺度训练速度快。当训练 epoch 在 300 或者 500 的时候训练效率是用户非常关注的，多尺度训练会比较慢
2. 训练 pipeline 中隐含了多尺度增强，等价于应用了多尺度训练，典型的如 `Mosaic`、`RandomAffine` 和 `Resize` 等，故没有必要再次引入模型输入的多尺度训练

在 COCO 数据集上进行了简单实验，如果直接在 YOLOv5 的 DataLoader 输出后再次引入多尺度训练增强实际性能提升非常小，但是这不代表用户自定义数据集微调模式下没有明显增益。如果想在 MMYOLO 中对 YOLO 系列算法开启多尺度训练，可以参考多尺度训练文档

2 使用 Mask 标注优化目标检测性能

在数据集标注完备例如同时存在边界框和实例分割标注但任务只需要其中部分标注情况下，可以借助完备的数据标注训练单一任务从而提升性能。在目标检测中同样可以借鉴实例分割标注来提升目标检测性能。以下是 YOLOv8 额外引入实例分割标注优化目标检测结果。性能增益如下所示：

从上述曲线图可以看出，不同尺度模型都有了不同程度性能提升。需要注意的是 `Mask Refine` 仅仅的是作用在数据增强阶段，对模型其他训练部分不需要任何改动，且不会影响训练速度。具体如下所示：

上述的 Mask 表示实例分割标注发挥关键作用的数据增强变换，将该技巧应用到其他 YOLO 系列中均有不同程度涨点。

3 训练后期关闭强增强提升检测性能

该策略是在 YOLOX 算法中第一次被提出可以极大的提升检测性能。论文中指出虽然 Mosaic+MixUp 可以极大的提升目标检测性能，但是它生成的训练图片远远脱离自然图片的真实分布，并且 Mosaic 大量的裁剪操作会带来很多不准确的标注框，所以 YOLOX 提出在最后 15 个 epoch 关掉强增强，转而使用较弱的增强，从而为了让检测器避开不准确标注框的影响，在自然图片的数据分布下完成最终的收敛。

该策略已经被应用到了大部分 YOLO 算法中，以 YOLOv8 为例其数据增强 pipeline 如下所示：

不过在何时关闭强增强是一个超参，如果关闭太早则可能没有充分发挥 Mosaic 等强增强效果，如果关闭太晚则由于之前已经过拟合，此时再关闭则没有任何增益。在 YOLOv8 实验中可以观察到该现象

从上表可以看出：

- 大模型在 COCO 数据集训练 500 epoch 会过拟合，在过拟合情况下再关闭 Mosaic 等强增强效果没有效果
- 使用 Mask 标注可以缓解过拟合，并且提升性能

4 加入纯背景图片抑制误报率

对于非开放世界数据集目标检测而言，训练和测试都是在固定类别上进行，一旦应用到没有训练过的类别图片上有可能会产生误报，一个常见的缓解策略是加入一定比例的纯背景图片。在大部分 YOLO 系列中都是默认开启了加入纯背景图片抑制误报率功能，用户只需要设置 `train_dataloader.dataset.filter_cfg.filter_empty_gt` 为 False 即可，表示将纯背景图片不过滤掉加入训练。

5 试试 AdamW 也许效果显著

YOLOv5，YOLOv6，YOLOv7 和 YOLOv8 等都是采用了 SGD 优化器，该参数器对参数的设置比较严格，而 AdamW 则正好相反，其对学习率设置等没有那么敏感。因此如果用户在自定义数据集微调可以尝试选择 AdamW 优化器。我们在 YOLOX 中进行了简单尝试，发现在 tiny、s 和 m 尺度模型上将其优化器替换为 AdamW 均有一定程度涨点。

具体见 configs/yolox/README.md。

6 考虑 ignore 场景避免不确定性标注

以 CrowdHuman 为例，其是一个拥挤行人检测数据集，下面是一张典型图片：

图片来自 detectron2 issue。黄色打叉的区域表示 `iscrowd` 标注。原因有两个方面：

- 这个区域不是真的人，例如海报上的人
- 该区域过于拥挤，很难标注

在该场景下，你不能简单的将这类标注删掉，因为你一旦删掉就表示当做背景区域来训练了，但是其和背景是不一样的，首先海报上的人和真人很像，并且拥挤区域确实有人只是不好标注。如果你简单的将其当做背景训练，那么会造成漏报。最合适的做法应该是把拥挤区域当做忽略区域即该区域的任何输出都直接忽略，不计算任何 Loss，不强迫模型拟合。

MMYOLO 在 YOLOv5 上简单快速的验证了 `iscrowd` 标注的作用，性能如下所示：

`ignore_iof_thr` 为 -1 表示不考虑忽略标签，可以看出性能有一定程度的提升，具体见 CrowdHuman 结果。如果你的自定义数据集上也有上述情况，则建议你考虑 ignore 场景避免不确定性标注。

7 使用知识蒸馏

知识蒸馏是一个被广泛使用的技巧，可以将大模型性能转移到小模型上从而提升小模型检测性能。目前 MMYOLO 和 MMRazor 已支持了该功能，并在 RTMDet 上进行了初步验证。

星号即为采用了大模型蒸馏的结果，详情见 Distill RTMDet。

8 更大的模型用更强的增强参数

如果你基于默认配置修改了模型或者替换了骨干网络，那么建议你基于此刻模型大小来缩放数据增强参数。一般来说更大的模型需要使用更强的增强参数，否则可能无法发挥大模型的效果，反之如果小模型应用了较强的增强则可能会欠拟合。以 RTMDet 为例，我们可以观察其不同模型大小的数据增强参数

其中 `random_resize_ratio_range` 表示 `RandomResize` 的随机缩放范围，`mosaic_max_cached_images/mixup_max_cached_images` 表示 `Mosaic/MixUp` 增强时候缓存的图片个数，可以用于调整增强的强度。YOLO 系列模型都是遵循同一套参数设置原则。

9.1.2 加快训练速度

1 单尺度训练开启 cudnn_benchmark

YOLO 系列算法中大部分网络输入图片大小都是固定的即单尺度，此时可以开启 `cudnn_benchmark` 来加快训练速度。该参数主要针对 PyTorch 的 cuDNN 底层库进行设置, 设置这个标志可以让内置的 cuDNN 自动寻找最适合当前配置的高效算法来优化运行效率。如果是多尺度模式开启该标志则会不断的寻找最优算法，反而会拖慢训练速度。

在 MMYOLO 中开启 `cudnn_benchmark`，只需要在配置中设置 `env_cfg = dict(cudnn_benchmark=True)`

2 使用带缓存的 Mosaic 和 MixUp

如果你的数据增强中应用了 Mosaic 和 MixUp，并且经过排查训练瓶颈来自图片的随机读取，那么建议将常规的 Mosaic 和 MixUp 替换为 RTMDet 中提出的带缓存的版本。

Mosaic 和 MixUp 涉及到多张图片的混合，它们的耗时会是普通数据增强的 K 倍 (K 为混入图片的数量)。如在 YOLOv5 中每次做 Mosaic 时，4 张图片的信息都需要从硬盘中重新加载。而带缓存的 Mosaic 和 MixUp 只需要重新载入当前的一张图片，其余参与混合增强的图片则从缓存队列中获取，通过牺牲一定内存空间的方式大幅提升了效率。

如图所示，cache 队列中预先储存了 N 张已加载的图像与标签数据，每一个训练 step 中只需加载一张新的图片及其标签数据并更新到 cache 队列中 (cache 队列中的图像可重复，如图中出现两次 img3)，同时如果 cache 队列长度超过预设长度，则随机 pop 一张图，当需要进行混合数据增强时，只需要从 cache 中随机选择需要的图像进行拼接等处理，而不需要全部从硬盘中加载，节省了图像加载的时间。

9.1.3 减少超参

YOLOv5 中通过实践提供了一些减少超参数的方法，下面详细说明。

1 Loss 权重自适应，少 1 个超参

一般来说，对于不同的任务或者不同的类别，可能需要针对性的设置超参，而这通常比较难。YOLOv5 中根据实践提出了一些根据类别数和检测输出层个数来自适应缩放 Loss 权重的方法，如下所示：

```
# scaled based on number of detection layers
loss_cls=dict(
    type='mmdet.CrossEntropyLoss',
    use_sigmoid=True,
    reduction='mean',
    loss_weight=loss_cls_weight *
    (num_classes / 80 * 3 / num_det_layers)),
```

(下页继续)

(续上页)

```
loss_bbox=dict(
    type='IoULoss',
    iou_mode='ciou',
    bbox_format='xywh',
    eps=1e-7,
    reduction='mean',
    loss_weight=loss_bbox_weight * (3 / num_det_layer
    return_iou=True),
loss_obj=dict(
    type='mmdet.CrossEntropyLoss',
    use_sigmoid=True,
    reduction='mean',
    loss_weight=loss_obj_weight *
    ((img_scale[0] / 640)**2 * 3 / num_det_layers)),
```

loss_cls 可以根据自定义类别数和检测层数对 loss_weight 进行自适应缩放，loss_bbox 可以根据检测层数进行自适应计算，而 loss_obj 可以根据输入图片大小和检测层数进行自适应缩放。这种策略可以让用户不用去设置 Loss 权重超参。需要说明的是：这个只是经验规则，并不是说是最佳设置组合，只是作为一个参考。

2 Weight Decay 和 Loss 输出值基于 Batch Size 自适应，少 2 个超参

一般来说，在不同的 Batch Size 上进行训练，需要遵循学习率自动缩放规则。但是在各个数据集上验证表明 YOLOv5 实际上在改变 Batch Size 时候不缩放学习率也可以取得不错的效果，甚至有时候你缩放了效果还更差。原因就在于代码中存在 Weight Decay 和 Loss 输出值基于 Batch Size 自适应的技巧。在 YOLOv5 中会基于当前训练的总 Batch Size 来缩放 Weight Decay 和 Loss 输出值。对应代码为：

```
# https://github.com/open-mmlab/mmyolo/blob/dev/mmyolo/engine/optimizers/yolov5_optim_
↪constructor.py#L86
if 'batch_size_per_gpu' in optimizer_cfg:
    batch_size_per_gpu = optimizer_cfg.pop('batch_size_per_gpu')
    # No scaling if total_batch_size is less than
    # base_total_batch_size, otherwise linear scaling.
    total_batch_size = get_world_size() * batch_size_per_gpu
    accumulate = max(
        round(self.base_total_batch_size / total_batch_size), 1)
    scale_factor = total_batch_size * \
        accumulate / self.base_total_batch_size
    if scale_factor != 1:
        weight_decay *= scale_factor
        print_log(f'Scaled weight_decay to {weight_decay}', 'current')
```

```
# https://github.com/open-mmlab/mmyolo/blob/dev/mmyolo/models/dense_heads/yolov5_head.
↪py#L635
_, world_size = get_dist_info()
return dict(
    loss_cls=loss_cls * batch_size * world_size,
    loss_obj=loss_obj * batch_size * world_size,
    loss_bbox=loss_box * batch_size * world_size)
```

在不同的 Batch Size 下 Loss 的权重是不一样大的，Batch Size 越大，Loss 就越大，梯度就越大，我个人猜测这可以等价于 Batch Size 增大时候，学习率线性增加的场合。实际上从 YOLOv5 的 YOLOv5 Study: mAP vs Batch-Size 中可以发现确实是希望用户在修改 Batch Size 时不需要修改其他参数也可以相近的性能。上述两个策略是一个非常不错的训练技巧。

9.1.4 减少训练显存

如何减少训练显存是一个经常谈论的问题，所涉及的技术也非常多。MMYOLO 的训练执行器来自 MMEngine，因此如何减少训练显存可以查阅 MMEngine 的文档。MMEngine 目前支持梯度累加、梯度检查点和大模型训练技术，详情见 节省显存。

9.2 测试技巧

9.2.1 推理速度和测试精度的平衡

在模型性能测试时候，我们一般是要求 mAP 越高越好，但是在实际应用或者推理时候我们希望在保证低误报率和漏报率情况下模型推理越快越好，或者说测试只关注 mAP 而忽略了后处理和评估速度，而实际落地应用时候会追求速度和精度的平衡。在 YOLO 系列中可以通过控制某些参数实现速度和精度平衡，下面以 YOLOv5 为例对其进行详细描述。

1 推理时避免一个检测框输出多个类别

YOLOv5 在训练分类分支时候采用的是 BCE Loss 即 `use_sigmoid=True`。假设物体类别数是 4，那么分类分支输出的类别数是 4 而不是 5，并且由于使用的是 sigmoid 而非 softmax 预测模式，很可能在某个位置预测出多个满足过滤阈值的检测框，也就是会出现一个预测 bbox 对应多个预测 label 的情况。如下图所示

一般在计算 mAP 时候过滤阈值为 0.001，由于 sigmoid 非竞争性预测模式会导致一个框对应多个 label。这种计算方式可以提高 mAP 计算时候的召回率，但是实际落地应用会不方便。

一个常用的办法就是提高过滤阈值，但是如果你不需要出现较多漏报，此时推荐你修改 `multi_label` 参数为 False，其位于配置的 `mode.test_cfg.multi_label` 中，默认值是 True 表示允许一个检测框对应多个 label。

2 简化 test pipeline

注意到 YOLOv5 的 test pipeline 为如下：

```
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

其使用了两个不同功能的 Resize，目的依然是提高评估时候的 mAP 值。在实际落地应用时候你可以简化该 pipeline，如下所示：

```
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='LetterResize',
        scale=_base_.img_scale,
        allow_scale_up=True,
        use_mini_pad=True),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

实际上 YOLOv5 算法在实际应用时候是采用简化的 pipeline，并将 multi_label 设为 False, score_thr 提高为 0.25，iou_threshold 降低为 0.45。在 YOLOv5 配置中我们提供了一套 detect 落地时候的配置参数，具体见 yolov5_s-v61_syncbn-detect_8xb16-300e_coco.py。

3 Batch Shape 策略加快测试速度

Batch Shape 是 YOLOv5 中提出的可以加快推理的一个测试技巧，其思路是不再强制要求整个测试过程图片都是 640x640，而是可以变尺度测试，只需要保证当前 batch 内的 shape 是一样的就行。这种方式可以减少额外的图片像素填充，从而实现加速推理过程。Batch Shape 的具体实现可以参考 链接。MMYOLO 中几乎所有算法在测试时候都是默认开启了 Batch Shape 策略。如果用户想关闭该功能，可以设置 `val_dataloader.dataset.batch_shapes_cfg=None`。

在实际落地场景下，因为动态 shape 没有固定 shape 快且高效，所以一般会不采用这个策略。

9.2.2 TTA 提升测试精度

TTA 测试时增强是一个万能的涨点技巧，在打比赛时候非常有用。MMYOLO 已经支持了 TTA，只需要在测试时候输入 `--tta` 即可开启。详情见 TTA 说明。

CHAPTER 10

MMYOLO 模型设计相关说明

10.1 YOLO 系列模型基类

下图为 RangeKing@GitHub 提供，非常感谢！

YOLO 系列算法大部分采用了统一的算法搭建结构，典型的如 Darknet + PAFPN。为了让用户快速理解 YOLO 系列算法架构，我们特意设计了如上图中的 BaseBackbone + BaseYOLONeck 结构。

抽象 BaseBackbone 的好处包括：

1. 子类不需要关心 forward 过程，只要类似建造者模式一样构建模型即可。
2. 可以通过配置实现定制插件功能，用户可以很方便的插入一些类似注意力模块。
3. 所有子类自动支持 frozen 某些 stage 和 frozen bn 功能。

抽象 BaseYOLONeck 也有同样好处。

10.1.1 BaseBackbone

- 如图 1 所示，对于 P5 而言，BaseBackbone 为包含 1 个 stem 层 + 4 个 stage 层的类似 ResNet 的基础结构。
- 如图 2 所示，对于 P6 而言，BaseBackbone 为包含 1 个 stem 层 + 5 个 stage 层的结构。

不同算法的主干网络继承 BaseBackbone，用户可以通过实现内部的 `build_xx` 方法，使用自定义的基础模块来构建每一层的内部结构。

10.1.2 BaseYOLONeck

与 BaseBackbone 的设计类似，我们为 MMYOLO 系列的 Neck 层进行了重构，主要分为 Reduce 层，UpSample 层，TopDown 层，DownSample 层，BottomUP 层以及输出卷积层，每一层结构都可以通过继承重写 build_xx 方法来实现自定义的内部结构。

10.1.3 BaseDenseHead

MMYOLO 系列沿用 MMDetection 中设计的 BaseDenseHead 作为其 Head 结构的基类，但是进一步拆分了 HeadModule. 以 YOLOv5 为例，其 HeadModule 中的 forward 实现代替了原有的 forward 实现。

10.2 HeadModule 说明

如上图所示，虚线部分为 MMDetection 中的实现，实线部分为 MMYOLO 中的实现。MMYOLO 版本与原实现相比具备具有以下优势：

1. MMDetection 中将 bbox_head 拆分为 assigner + box coder + sampler 三个大的组件，但由于 3 个组件之间的传递为了通用性，需要封装额外的对象来处理，统一之后用户可以不用进行拆分。不刻意强求划分三大组件的好处为：不再需要对内部数据进行数据封装，简化了代码逻辑，减轻了社区使用难度和算法复现难度。
2. 速度更快，用户在自定义实现算法时候，可以不依赖于原有框架，对部分代码进行深度优化。

总的来说，在 MMYOLO 中只需要做到将 model + loss_by_feat 部分解耦，用户就可以通过修改配置实现任意模型配合任意的 loss_by_feat 计算过程。例如将 YOLOv5 模型应用 YOLOX 的 loss_by_feat 等。

以 MMDetection 中 YOLOX 配置为例，其 Head 模块配置写法为：

```
bbox_head=dict(
    type='YOLOXHead',
    num_classes=80,
    in_channels=128,
    feat_channels=128,
    stacked_convs=2,
    strides=(8, 16, 32),
    use_depthwise=False,
    norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
    act_cfg=dict(type='Swish'),
    ...
    loss_obj=dict(
        type='CrossEntropyLoss',
        use_sigmoid=True,
        reduction='sum',
```

(下页继续)

(续上页)

```
        loss_weight=1.0),
    loss_l1=dict(type='L1Loss', reduction='sum', loss_weight=1.0)),
train_cfg=dict(assigner=dict(type='SimOTAAssigner', center_radius=2.5)),
```

在 MMYOLO 中抽取 `head_module` 后，新的配置写法为：

```
bbox_head=dict(
    type='YOLOXHead',
    head_module=dict(
        type='YOLOXHeadModule',
        num_classes=80,
        in_channels=256,
        feat_channels=256,
        widen_factor=widen_factor,
        stacked_convs=2,
        featmap_strides=(8, 16, 32),
        use_depthwise=False,
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='SiLU', inplace=True),
    ),
    ...
    loss_obj=dict(
        type='mmdet.CrossEntropyLoss',
        use_sigmoid=True,
        reduction='sum',
        loss_weight=1.0),
    loss_bbox_aux=dict(type='mmdet.L1Loss', reduction='sum', loss_weight=1.0)),
train_cfg=dict(
    assigner=dict(
        type='mmdet.SimOTAAssigner',
        center_radius=2.5,
        iou_calculator=dict(type='mmdet.BboxOverlaps2D'))),
```

CHAPTER 11

算法原理和实现全解析

11.1 YOLOv5 原理和实现全解析

11.1.1 0 简介

以上结构图由 RangeKing@github 绘制。

YOLOv5 是一个面向实时工业应用而开源的目标检测算法，受到了广泛关注。我们认为让 YOLOv5 爆火的原因不单纯在于 YOLOv5 算法本身的优异性，更多的在于开源库的实用和鲁棒性。简单来说 YOLOv5 开源库的主要特点为：

1. **友好和完善的部署支持**
2. **算法训练速度极快**，在 300 epoch 情况下训练时长和大部分 one-stage 算法如 RetinaNet、ATSS 和 two-stage 算法如 Faster R-CNN 在 12 epoch 的训练时间接近
3. 框架进行了**非常多的 corner case 优化**，功能和文档也比较丰富

如图 1 和 2 所示，YOLOv5 的 P5 和 P6 版本主要差异在于网络结构和图片输入分辨率。其他区别，如 anchors 个数和 loss 权重可详见配置文件。本文将从 YOLOv5 算法本身原理讲起，然后重点分析 MMYOLO 中的实现。关于 YOLOv5 的使用指南和速度等对比请阅读本文的后续内容。

提示： 没有特殊说明情况下，本文默认描述的是 P5 模型。

希望本文能够成为你入门和掌握 YOLOv5 的核心文档。由于 YOLOv5 本身也在不断迭代更新，我们也会不断的更新本文档。请注意阅读最新版本。

MMYOLO 实现配置：https://github.com/open-mmlab/mmyolo/blob/main/configs/yolov5/

YOLOv5 官方开源库地址：https://github.com/ultralytics/yolov5

11.1.2 1 v6.1 算法原理和 MMYOLO 实现解析

YOLOv5 官方 release 地址：https://github.com/ultralytics/yolov5/releases/tag/v6.1

性能如上表所示。YOLOv5 有 P5 和 P6 两个不同训练输入尺度的模型，P6 即为 1280x1280 输入的大模型，通常用的是 P5 常规模型，输入尺寸是 640x640 。本文解读的也是 P5 模型结构。

通常来说，目标检测算法都可以分成数据增强、模型结构、loss 计算等组件，YOLOv5 也一样，如下所示：

下面将从原理和结合 MMYOLO 的具体实现方面进行简要分析。

1.1 数据增强模块

YOLOv5 目标检测算法中使用的数据增强比较多，包括：

- **Mosaic 马赛克**
- **RandomAffine 随机仿射变换**
- **MixUp**
- **图像模糊等采用 Albu 库实现的变换**
- **HSV 颜色空间增强**
- **随机水平翻转**

其中 Mosaic 数据增强概率为 1，表示一定会触发，而对于 small 和 nano 两个版本的模型不使用 MixUp，其他的 l/m/x 系列模型则采用了 0.1 的概率触发 MixUp。小模型能力有限，一般不会采用 MixUp 等强数据增强策略。

其核心的 Mosaic + RandomAffine + MixUp 过程简要绘制如下：

下面对其进行简要分析。

1.1.1 Mosaic 马赛克

Mosaic 属于混合类数据增强，因为它在运行时候需要 4 张图片拼接，变相的相当于增加了训练的 batch size。其运行过程简要概况为：

1. 随机生成拼接后 4 张图的交接中心点坐标，此时就相当于确定了 4 张拼接图片的交接点
2. 随机选出另外 3 张图片的索引以及读取对应的标注
3. 对每张图片采用保持宽高比的 resize 操作将其缩放到指定大小

4. 按照上下左右规则，计算每张图片在待输出图片中应该放置的位置，因为图片可能出界故还需要计算裁剪坐标
5. 利用裁剪坐标将缩放后的图片裁剪，然后贴到前面计算出的位置，其余位置全部补 114 像素值
6. 对每张图片的标注也进行相应处理

注意：由于拼接了 4 张图，所以输出图片面积会扩大 4 倍，从 640x640 变成 1280x1280，因此要想恢复为 640x640，必须要再接一个 **RandomAffine 随机仿射变换，否则图片面积就一直是扩大 4 倍的**。

1.1.2 RandomAffine 随机仿射变换

随机仿射变换有两个目的：

1. 对图片进行随机几何仿射变换
2. 将 Mosaic 输出的扩大 4 倍的图片还原为 640x640 尺寸

随机仿射变换包括平移、旋转、缩放、错切等几何增强操作，同时由于 Mosaic 和 RandomAffine 属于比较强的增强操作，会引入较大噪声，因此需要对增强后的标注进行处理，过滤规则为：

1. 增强后的 gt bbox 宽高要大于 wh_thr
2. 增强后的 gt bbox 面积和增强前的 gt bbox 面积比要大于 ar_thr，防止增强太严重
3. 最大宽高比要小于 area_thr，防止宽高比改变太多

由于旋转后标注框会变大导致不准确，因此目标检测里面很少会使用旋转数据增强。

1.1.3 MixUp

MixUp 和 Mosaic 类似也属于混合图片类增强方法。随机选出另外一张图后将两图再随机混合。具体实现方法有多种，常见的做法是要么将 label 直接拼接起来，要么将 label 也采用 alpha 方法混合。原作者的做法非常简单，对 label 即直接拼接，而图片通过分布采样混合。

需要特别注意的是：**YOLOv5 实现的 MixUp 中，随机出来的另一张图也需要经过 Mosaic 马赛克 + RandomAffine 随机仿射变换的增强后才能混合。这个和其他开源库实现可能不太一样**。

1.1.4 图像模糊和其他数据增强策略

剩下的数据增强包括

- **图像模糊等采用 Albu 库实现的变换**
- **HSV 颜色空间增强**
- **随机水平翻转**

MMDetection 开源库中已经对 Albu 第三方数据增强库进行了封装，使用户可以简单的通过配置即可使用 Albu 库中提供的任何数据增强功能。而 HSV 颜色空间增强和随机水平翻转都是属于比较常规的数据增强，不需要特殊介绍。

1.1.5 MMYOLO 实现解析

常规的单图数据增强例如随机翻转等比较容易实现，而 Mosaic 类的混合数据增强则不太容易。在 MMDetection 复现的 YOLOX 算法中提出了 MultiImageMixDataset 数据集包装器的概念，其实现过程如下：

对于 Mosaic 等混合类数据增强策略，会需要额外实现一个 `get_indexes` 方法来获取其他图片索引，然后用得到的 4 张图片信息就可以进行 Mosaic 增强了。以 MMDetection 中实现的 YOLOX 为例，其配置文件写法如下所示：

```
train_pipeline = [
    dict(type='Mosaic', img_scale=img_scale, pad_val=114.0),
    dict(
        type='RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
    dict(
        type='MixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0),
    ...
]

train_dataset = dict(
    # use MultiImageMixDataset wrapper to support mosaic and mixup
    type='MultiImageMixDataset',
    dataset=dict(
        type='CocoDataset',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations', with_bbox=True)
        ]),
    pipeline=train_pipeline)
```

MultiImageMixDataset 数据集包装器传入一个包括 Mosaic 和 RandAffine 等数据增强，而 CocoDataset 中也需要传入一个包括图片和标注加载的 pipeline。通过这种方式就可以快速的实现混合类数据增强。

但是上述实现有一个缺点：**对于不熟悉 MMDetection 的用户来说，其经常会忘记 Mosaic 必须要和 MultiImageMixDataset 配合使用，否则会报错，而且这样会加大复杂度和理解难度**。

为了解决这个问题，在 MMYOLO 中我们进一步进行了简化。直接让 pipeline 能够获取到 dataset 对象，此时就可以将 Mosaic 等混合类数据增强的实现和使用变成和随机翻转一样。此时在 MMYOLO 中 YOLOX 的配

置写法变成如下所示：

```
pre_transform = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True)
]

train_pipeline = [
    *pre_transform,
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='mmdet.RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
    dict(
        type='YOLOXMixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0,
        pre_transform=pre_transform),
    ...
]
```

这样就不再需要 MultiImageMixDataset 了，使用和理解上会更加简单。

回到 YOLOv5 配置上，因为 YOLOv5 实现的 MixUp 中，随机选出来的另一张图也需要经过 Mosaic 马赛克+RandomAffine 随机仿射变换增强后才能混合，故 YOLOv5-m 数据增强配置如下所示：

```
pre_transform = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True)
]

mosaic_transform= [
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='YOLOv5RandomAffine',
```

(下页继续)

(续上页)

```
        max_rotate_degree=0.0,
        max_shear_degree=0.0,
        scaling_ratio_range=(0.1, 1.9),  # scale = 0.9
        border=(-img_scale[0] // 2, -img_scale[1] // 2),
        border_val=(114, 114, 114))
]

train_pipeline = [
    *pre_transform,
    *mosaic_transform,
    dict(
        type='YOLOv5MixUp',
        prob=0.1,
        pre_transform=[
            *pre_transform,
            *mosaic_transform
        ]),
    ...
]
```

1.2 网络结构

本小结由 RangeKing@github 撰写，非常感谢!!!

YOLOv5 网络结构是标准的 `CSPDarknet` + `PAFPN` + 非解耦 `Head`。

YOLOv5 网络结构大小由 `deepen_factor` 和 `widen_factor` 两个参数决定。其中 `deepen_factor` 控制网络结构深度，即 `CSPLayer` 中 `DarknetBottleneck` 模块堆叠的数量；`widen_factor` 控制网络结构宽度，即模块输出特征图的通道数。以 YOLOv5-l 为例，其 `deepen_factor = widen_factor = 1.0`。P5 和 P6 的模型整体结构分别如图 1 和图 2 所示。

图的上半部分为模型总览；下半部分为具体网络结构，其中的模块均标有序号，方便用户与 YOLOv5 官方仓库的配置文件对应；中间部分为各子模块的具体构成。

如果想使用 netron 可视化网络结构图细节，可以直接在 netron 中将 MMDeploy 导出的 ONNX 文件格式文件打开。

提示： 1.2 小节涉及的特征维度（shape）都为 (B, C, H, W)。

1.2.1 Backbone

在 MMYOLO 中 CSPDarknet 继承自 BaseBackbone，整体结构和 ResNet 类似。P5 模型共 5 层结构，包含 1 个 Stem Layer 和 4 个 Stage Layer：

- Stem Layer 是 1 个 6x6 kernel 的 ConvModule，相较于 v6.1 版本之前的 Focus 模块更加高效。
- 除了最后一个 Stage Layer，其他均由 1 个 ConvModule 和 1 个 CSPLayer 组成。如上图 Details 部分所示。其中 ConvModule 为 3x3 的 Conv2d + BatchNorm + SiLU 激活函数。CSPLayer 即 YOLOv5 官方仓库中的 C3 模块，由 3 个 ConvModule + n 个 DarknetBottleneck(带残差连接) 组成。
- 最后一个 Stage Layer 在最后增加了 SPPF 模块。SPPF 模块是将输入串行通过多个 5x5 大小的 MaxPool2d 层，与 SPP 模块效果相同，但速度更快。
- P5 模型会在 Stage Layer 2-4 之后分别输出一个特征图进入 Neck 结构。以 640x640 输入图片为例，其输出特征为 (B,256,80,80)、(B,512,40,40) 和 (B,1024,20,20)，对应的 stride 分别为 8/16/32。
- P6 模型会在 Stage Layer 2-5 之后分别输出一个特征图进入 Neck 结构。以 1280x1280 输入图片为例，其输出特征为 (B,256,160,160)、(B,512,80,80)、(B,768,40,40) 和 (B,1024,20,20)，对应的 stride 分别为 8/16/32/64。

1.2.2 Neck

YOLOv5 官方仓库的配置文件中并没有 Neck 部分，为方便用户与其他目标检测网络结构相对应，我们将官方仓库的 Head 拆分成 PAFPN 和 Head 两部分。

基于 BaseYOLONeck 结构，YOLOv5 Neck 也是遵循同一套构建流程，对于不存在的模块，我们采用 nn.Identity 代替。

Neck 模块输出的特征图和 Backbone 完全一致。即 P5 模型为 (B,256,80,80)、(B,512,40,40) 和 (B,1024,20,20)；P6 模型为 (B,256,160,160)、(B,512,80,80)、(B,768,40,40) 和 (B,1024,20,20)。

1.2.3 Head

YOLOv5 Head 结构和 YOLOv3 完全一样，为非解耦 Head。Head 模块只包括 3 个不共享权重的卷积，用于将输入特征图进行变换而已。

前面的 PAFPN 依然是输出 3 个不同尺度的特征图，shape 为 (B,256,80,80)、(B,512,40,40) 和 (B,1024,20,20)。由于 YOLOv5 是非解耦输出，即分类和 bbox 检测等都是在同一个卷积的不同通道中完成。以 COCO 80 类为例：

- P5 模型在输入为 640x640 分辨率情况下，其 Head 模块输出的 shape 分别为 (B, 3x(4+1+80),80,80),(B, 3x(4+1+80),40,40) 和 (B, 3x(4+1+80),20,20)。
- P6 模型在输入为 1280x1280 分辨率情况下，其 Head 模块输出的 shape 分别为 (B, 3x(4+1+80),160,160),(B, 3x(4+1+80),80,80),(B, 3x(4+1+80),40,40) 和 (B, 3x(4+1+80),20,20)。其

中 3 表示 3 个 anchor，4 表示 bbox 预测分支，1 表示 obj 预测分支，80 表示 COCO 数据集类别预测分支。

1.3 正负样本匹配策略

正负样本匹配策略的核心是确定预测特征图的所有位置中哪些位置应该是正样本，哪些是负样本，甚至有些是忽略样本。匹配策略是目标检测算法的核心，一个好的匹配策略可以显著提升算法性能。

YOLOV5 的匹配策略简单总结为：**采用了 anchor 和 gt_bbox 的 shape 匹配度作为划分规则，同时引入跨邻域网格策略来增加正样本**。其主要包括如下两个核心步骤：

1. 对于任何一个输出层，抛弃了常用的基于 Max IoU 匹配的规则，而是直接采用 shape 规则匹配，也就是该 GT Bbox 和当前层的 Anchor 计算宽高比，如果宽高比例大于设定阈值，则说明该 GT Bbox 和 Anchor 匹配度不够，将该 GT Bbox 暂时丢掉，在该层预测中该 GT Bbox 对应的网格内的预测位置认为是负样本
2. 对于剩下的 GT Bbox(也就是匹配上的 GT Bbox)，计算其落在哪个网格内，同时利用四舍五入规则，找出最近的两个网格，将这三个网格都认为是负责预测该 GT Bbox 的，可以粗略估计正样本数相比之前的 YOLO 系列，至少增加了三倍

下面会对每个部分进行详细说明，部分描述和图示直接或间接参考自官方 Repo。

1.3.1 Anchor 设置

YOLOv5 是 Anchor-based 的目标检测算法，其 Anchor size 的获取方式与 YOLOv3 类似，也是使用聚类获得，其不同之处在于聚类使用的标准不再是基于 IoU 的，而是使用形状上的宽高比作为聚类准则 (即 shape-match)。

在用户更换了数据集后，可以使用 MMYOLO 里带有的 Anchor 分析工具，对自己的数据集进行分析，确定合适的 Anchor size。

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} --algorithm v5-k-means
 --input-shape ${INPUT_SHAPE [WIDTH HEIGHT]} --output-dir ${OUTPUT_DIR}
```

然后在 config 文件 里修改默认 Anchor size:

```
anchors = [[(10, 13), (16, 30), (33, 23)], [(30, 61), (62, 45), (59, 119)],
           [(116, 90), (156, 198), (373, 326)]]
```

1.3.2 Bbox 编解码过程

在 Anchor-based 算法中，预测框通常会基于 Anchor 进行变换，然后预测变换量，这对应 GT Bbox 编码过程，而在预测后需要进行 Pred Bbox 解码，还原为真实尺度的 Bbox，这对应 Pred Bbox 解码过程。

在 YOLOv3 中，回归公式为：

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= a_w \cdot e^{t_w} \\
b_h &= a_h \cdot e^{t_h}
\end{aligned}
$$

公式中，

$$
\begin{aligned}
a_w &\text{□□}Anchor\text{□□□} \\
c_x &\text{□□}Grid\text{□□□□□} \\
\sigma &\text{□□}Sigmoid\text{□□□}
\end{aligned}
$$

而在 YOLOv5 中，回归公式为：

$$
\begin{aligned}
b_x &= (2 \cdot \sigma(t_x) - 0.5) + c_x \\
b_y &= (2 \cdot \sigma(t_y) - 0.5) + c_y \\
b_w &= a_w \cdot (2 \cdot \sigma(t_w))^2 \\
b_h &= a_h \cdot (2 \cdot \sigma(t_h))^2
\end{aligned}
$$

改进之处主要有以下两点：

- 中心点坐标范围从 (0, 1) 调整至 (-0.5, 1.5)
- 宽高范围从

$$(0\text{□} + \infty)$$

调整至

$$(0\text{□}4a_{wh})$$

这个改进具有以下好处：

- **新的中心点设置能更好的预测到 0 和 1**。这有助于更精准回归出 box 坐标。
- 宽高回归公式中 exp(x) 是无界的，这会导致**梯度失去控制**，造成训练不稳定。YOLOv5 中改进后的宽高回归公式优化了此问题。

1.3.3 匹配策略

在 MMYOLO 设计中，无论网络是 Anchor-based 还是 Anchor-free，**我们统一使用 prior 称呼 Anchor**。

正样本匹配包含以下两步：

(1) “比例”比较

将 GT Bbox 的 WH 与 Prior 的 WH 进行“比例”比较。

比较流程：

$$
\begin{aligned}
r_w &= w\_gt / w\_pt \\
r_h &= h\_gt / h\_pt \\
r_w^{max} &= max(r_w, 1/r_w) \\
r_h^{max} &= max(r_h, 1/r_h) \\
r^{max} &= max(r_w^{max}, r_h^{max}) \\
if\ \ r_{max} &< prior\_match\_thr : match!
\end{aligned}
$$

此处我们用一个 GT Bbox 与 P3 特征图的 Prior 进行匹配的案例进行讲解和图示：

prior1 匹配失败的原因是

$$
h\_gt\ /\ h\_prior\ =\ 4.8\ >\ prior\_match\_thr
$$

(2) 为步骤 1 中 match 的 GT 分配对应的正样本

依然沿用上面的例子：

GT Bbox (cx, cy, w, h) 值为 (26, 37, 36, 24)，

Prior WH 值为 [(15, 5), (24, 16), (16, 24)]，在 P3 特征图上，stride 为 8。通过计算，prior2 和 prior3 能够 match。

计算过程如下：

(2.1) 将 GT Bbox 的中心点坐标对应到 P3 的 grid 上

$$
\begin{aligned}
GT_x^{center_grid} &= 26/8 = 3.25 \\
GT_y^{center_grid} &= 37/8 = 4.625
\end{aligned}
$$

(2.2) 将 GT Bbox 中心点所在的 grid 分成四个象限，**由于中心点落在了左下角的象限当中，那么会将物体的左、下两个 grid 也认为是正样本**

下图展示中心点落到不同位置时的正样本分配情况：

那么 YOLOv5 的 Assign 方式具体带来了哪些改进?

- 一个 GT Bbox 能够匹配多个 Prior
- 一个 GT Bbox 和一个 Prior 匹配时，能分配 1-3 个正样本
- 以上策略能**适度缓解目标检测中常见的正负样本不均衡问题**。

而 YOLOv5 中的回归方式，和 Assign 方式是相互呼应的：

1. 中心点回归方式：
2. WH 回归方式：

1.4 Loss 设计

YOLOv5 中总共包含 3 个 Loss，分别为：

- Classes loss：使用的是 BCE loss
- Objectness loss：使用的是 BCE loss
- Location loss：使用的是 CIoU loss

三个 loss 按照一定比例汇总：

$$Loss = \lambda_1 L_{cls} + \lambda_2 L_{obj} + \lambda_3 L_{loc}$$

P3、P4、P5 层对应的 Objectness loss 按照不同权重进行相加，默认的设置是

```
obj_level_weights=[4., 1., 0.4]
```

$$L_{obj} = 4.0 \cdot L_{obj}^{small} + 1.0 \cdot L_{obj}^{medium} + 0.4 \cdot L_{obj}^{large}$$

在复现中我们发现 YOLOv5 中使用的 CIoU 与目前最新官方 CIoU 存在一定的差距，差距体现在 alpha 参数的计算。

官方版本：

参考资料：https://github.com/Zzh-tju/CIoU/blob/master/layers/modules/multibox_loss.py#L53-L55

```
alpha = (ious > 0.5).float() * v / (1 - ious + v)
```

YOLOv5 版本：

```
alpha = v / (v - ious + (1 + eps))
```

这是一个有趣的细节，后续需要测试不同 alpha 计算方式情况下带来的精度差距。

1.5 优化策略和训练过程

YOLOv5 对每个优化器的参数组进行非常精细的控制，简单来说包括如下部分。

1.5.1 优化器分组

将优化参数分成 Conv/Bias/BN 三组，在 WarmUp 阶段，不同组采用不同的 lr 以及 momentum 更新曲线。同时在 WarmUp 阶段采用的是 iter-based 更新策略，而在非 WarmUp 阶段则变成 epoch-based 更新策略，可谓是 trick 十足。

MMYOLO 中是采用 YOLOv5OptimizerConstructor 优化器构造器实现优化器参数分组。优化器构造器的作用就是对一些特殊的参数组初始化过程进行精细化控制，因此可以很好的满足需求。

而不同的参数组采用不同的调度曲线功能则是通过 YOLOv5ParamSchedulerHook 实现。而不同的参数组采用不同的调度曲线功能则是通过 YOLOv5ParamSchedulerHook 实现。

1.5.2 weight decay 参数自适应

作者针对不同的 batch size 采用了不同的 weight decay 策略，具体来说为：

1. 当训练 batch size <= 64 时，weight decay 不变
2. 当训练 batch size > 64 时，weight decay 会根据总 batch size 进行线性缩放

MMYOLO 也是通过 YOLOv5OptimizerConstructor 实现。

1.5.3 梯度累加

为了最大化不同 batch size 情况下的性能，作者设置总 batch size 小于 64 时候会自动开启梯度累加功能。

训练过程和大部分 YOLO 类似，包括如下策略：

1. 没有使用预训练权重
2. 没有采用多尺度训练策略，同时可以开启 cudnn.benchmark 进一步加速训练
3. 使用了 EMA 策略平滑模型
4. 默认采用 AMP 自动混合精度训练

需要特意说明的是：YOLOv5 官方对于 small 模型是采用单卡 v100 训练，bs 为 128，而 m/l/x 等是采用不同数目的多卡实现的，这种训练策略不太规范，**为此在 MMYOLO 中全部采用了 8 卡，每卡 16 bs 的设置，同时为了避免性能差异，训练时候开启了 SyncBN**。

1.6 推理和后处理过程

YOLOv5 后处理过程和 YOLOv3 非常类似，实际上 YOLO 系列的后处理逻辑都是类似的。

1.6.1 核心控制参数

1. **multi_label**

对于多类别预测来说需要考虑是否是多标签任务，也就是同一个预测位置会预测的多个类别概率，和是否当作单类处理。因为 YOLOv5 采用 sigmoid 预测模式，在考虑多标签情况下可能会出现一个物体检测出两个不同类别的框，这有助于评估指标 mAP，但是不利于实际应用。因此在需要算评估指标时候 multi_label 是 True，而推理或者实际应用时候是 False

2. **score_thr 和 nms_thr**

score_thr 阈值用于过滤类别分值，低于分值的检测框当做背景处理，nms_thr 是 nms 时阈值。同样的，在计算评估指标 mAP 阶段可以将 score_thr 设置的非常低，这通常能够提高召回率，从而提升 mAP，但是对于实际应用来说没有意义，且会导致推理过程极慢。为此在测试和推理阶段会设置不同的阈值

3. **nms_pre 和 max_per_img**

nms_pre 表示 nms 前的最大保留检测框数目，这通常是为了防止 nms 运行时候输入框过多导致速度过慢问题，默认值是 30000。max_per_img 表示最终保留的最大检测框数目，通常设置为 300。

以 COCO 80 类为例，假设输入图片大小为 640x640

其推理和后处理过程为：

(1) 维度变换

YOLOv5 输出特征图尺度为 80x80、40x40 和 20x20 的三个特征图，每个位置共 3 个 anchor，因此输出特征图通道为 3x(5+80)=255。YOLOv5 是非解耦输出头，而其他大部分算法都是解耦输出头，为了统一后处理逻辑，我们提前将其进行解耦，分成了类别预测分支、bbox 预测分支和 obj 预测分支。

将三个不同尺度的类别预测分支、bbox 预测分支和 obj 预测分支进行拼接，并进行维度变换。为了后续方便处理，会将原先的通道维度置换到最后，类别预测分支、bbox 预测分支和 obj 预测分支的 shape 分别为 (b, 3x80x80+3x40x40+3x20x20, 80)=(b,25200,80)，(b,25200,4)，(b,25200,1)。

(2) 解码还原到原图尺度

分类预测分支和 obj 分支需要进行 sigmoid 计算，而 bbox 预测分支需要进行解码，还原为真实的原图解码后 xyxy 格式

(3) 第一次阈值过滤

遍历 batch 中的每张图，然后用 score_thr 对类别预测分值进行阈值过滤，去掉低于 score_thr 的预测结果

(4) 第二次阈值过滤

将 obj 预测分值和过滤后的类别预测分值相乘，然后依然采用 score_thr 进行阈值过滤。在这过程中还需要考虑 **multi_label 和 nms_pre，确保过滤后的检测框数目不会多于 nms_pre**。

(5) 还原到原图尺度和 nms

基于前处理过程，将剩下的检测框还原到网络输出前的原图尺度，然后进行 nms 即可。最终输出的检测框不能多于 **max_per_img**。

1.6.2 batch shape 策略

为了加速验证集的推理过程，作者提出了 batch shape 策略，其核心原则是：**确保在 batch 推理过程中同一个 batch 内的图片 pad 像素最少，不要求整个验证过程中所有 batch 的图片尺度一样**。

其大概流程是：将整个测试或者验证数据的宽高比进行排序，然后依据 batch 设置将排序后的图片组成一个 batch，同时计算这个 batch 内最佳的 batch shape，防止 pad 像素过多。最佳 batch shape 计算原则为在保持宽高比的情况下进行 pad，不追求正方形图片输出。

```
        image_shapes = []
        for data_info in data_list:
            image_shapes.append((data_info['width'], data_info['height']))

        image_shapes = np.array(image_shapes, dtype=np.float64)

        n = len(image_shapes)  # number of images
        batch_index = np.floor(np.arange(n) / self.batch_size).astype(
            np.int64)  # batch index
        number_of_batches = batch_index[-1] + 1  # number of batches

        aspect_ratio = image_shapes[:, 1] / image_shapes[:, 0]  # aspect ratio
        irect = aspect_ratio.argsort()

        data_list = [data_list[i] for i in irect]

        aspect_ratio = aspect_ratio[irect]
        # Set training image shapes
        shapes = [[1, 1]] * number_of_batches
        for i in range(number_of_batches):
            aspect_ratio_index = aspect_ratio[batch_index == i]
            min_index, max_index = aspect_ratio_index.min(
            ), aspect_ratio_index.max()
            if max_index < 1:
                shapes[i] = [max_index, 1]
            elif min_index > 1:
                shapes[i] = [1, 1 / min_index]

        batch_shapes = np.ceil(
            np.array(shapes) * self.img_size / self.size_divisor +
            self.pad).astype(np.int64) * self.size_divisor
```

(下页继续)

(续上页)

```
        for i, data_info in enumerate(data_list):
            data_info['batch_shape'] = batch_shapes[batch_index[i]]
```

11.1.3 2 总结

本文对 YOLOv5 原理和在 MMYOLO 实现进行了详细解析，希望能帮助用户理解算法实现过程。同时请注意：由于 YOLOv5 本身也在不断更新，本开源库也会不断迭代，请及时阅读和同步最新版本。

11.2 YOLOv6 原理和实现全解析

11.2.1 0 简介

以上结构图由 wzr-skn@github 绘制。

YOLOv6 提出了一系列适用于各种工业场景的模型，包括 N/T/S/M/L，考虑到模型的大小，其架构有所不同，以获得更好的精度-速度权衡。本算法专注于检测的精度和推理效率，并在网络结构、训练策略等算法层面进行了多项改进和优化。

简单来说 YOLOv6 开源库的主要特点为：

1. 统一设计了更高效的 Backbone 和 Neck：受到硬件感知神经网络设计思想的启发，基于 RepVGG style 设计了可重参数化、更高效的骨干网络 EfficientRep Backbone 和 Rep-PAN Neck。
2. 相比于 YOLOX 的 Decoupled Head，进一步优化设计了简洁有效的 Efficient Decoupled Head，在维持精度的同时，降低了一般解耦头带来的额外延时开销。
3. 在训练策略上，采用 Anchor-free 的策略，同时辅以 SimOTA 标签分配策略以及 SIoU 边界框回归损失来进一步提高检测精度。

本文将从 YOLOv6 算法本身原理讲起，然后重点分析 MMYOLO 中的实现。关于 YOLOv6 的使用指南和速度等对比请阅读本文的后续内容。

希望本文能够成为你入门和掌握 YOLOv6 的核心文档。由于 YOLOv6 本身也在不断迭代更新，我们也会不断的更新本文档。请注意阅读最新版本。

MMYOLO 实现配置：https://github.com/open-mmlab/mmyolo/blob/main/configs/yolov6/

YOLOv6 官方开源库地址：https://github.com/meituan/YOLOv6

11.2.2 1 YOLOv6 2.0 算法原理和 MMYOLO 实现解析

YOLOv6 2.0 官方 release 地址：https://github.com/meituan/YOLOv6/releases/tag/0.2.0

YOLOv6 和 YOLOv5 一样也可以分成数据增强、模型结构、loss 计算等组件，如下所示：

下面将从原理和结合 MMYOLO 的具体实现方面进行简要分析。

1.1 数据增强模块

YOLOv6 目标检测算法中使用的数据增强与 YOLOv5 基本一致，唯独不一样的是没有使用 Albu 的数据增强方式：

- **Mosaic 马赛克**
- **RandomAffine 随机仿射变换**
- **MixUp**
- **~~ 图像模糊等采用 Albu 库实现的变换 ~~**
- **HSV 颜色空间增强**
- **随机水平翻转**

关于每一个增强的详细解释，详情请看*YOLOv5 数据增强模块*

另外，YOLOv6 参考了 YOLOX 的数据增强方式，分为 2 种增强方法组，一开始和 YOLOv5 一致，但是在最后 15 个 epoch 的时候将 `Mosaic` 使用 `YOLOv5KeepRatioResize` + `LetterResize` 替代了，个人感觉是为了拟合真实情况。

1.2 网络结构

YOLOv6 N/T/S 模型的网络结构由 `EfficientRep` + `Rep-PAN` + `Efficient decoupled Head` 构成，M/L 模型的网络结构则由 `CSPBep` + `CSPRepPAFPN` + `Efficient decoupled Head` 构成。其中，Backbone 和 Neck 部分的结构与 YOLOv5 较为相似，但不同的是其采用了重参数化结构 `RepVGG Block` 替换掉了原本的 `ConvModule`，在此基础上，将 `CSPLayer` 改进为了多个 `RepVGG` 堆叠的 `RepStageBlock`（N/T/S 模型）或 `BepC3StageBlock`（M/L 模型）；Head 部分则参考了 FCOS 和 YOLOX 的检测头，将回归与分类分支解耦成两个分支进行预测。YOLOv6-S 和 YOLOv6-L 整体结构分别如图 1 和图 2 所示。

1.2.1 Backbone

已有研究表明，多分支的网络结构通常比单分支网络性能更加优异，例如 YOLOv5 的 CSPDarknet，但是这种结构会导致并行度降低进而增加推理延时；相反，类似于 VGG 的单分支网络则具有并行度高、内存占用小的优点，因此推理效率更高。而 RepVGG 则同时具备上述两种结构的优点，在训练时可解耦成多分支拓扑结构提升模型精度，实际部署时可等效融合为单个 3×3 卷积提升推理速度，RepVGG 示意图如下。因此，YOLOv6 基于 RepVGG 重参数化结构设计了高效的骨干网络 EfficientRep 和 CSPBep，其可以充分利用硬件算力，提升模型表征能力的同时降低推理延时。

在 N/T/S 模型中，YOLOv6 使用了 EfficientRep 作为骨干网络，其包含 1 个 Stem Layer 和 4 个 Stage Layer，具体细节如下：

- Stem Layer 中采用 stride=2 的 RepVGGBlock 替换了 stride=2 的 6×6 ConvModule。
- Stage Layer 结构与 YOLOv5 基本相似，将每个 Stage layer 的 1 个 ConvModule 和 1 个 CSPLayer 分别替换为 1 个 RepVGGBlock 和 1 个 RepStageBlock，如上图 Details 部分所示。其中，第一个 RepVGGBlock 会做下采样和 Channel 维度变换，而每个 RepStageBlock 则由 n 个 RepVGGBlock 组成。此外，仍然在第 4 个 Stage Layer 最后增加 SPPF 模块后输出。

在 M/L 模型中，由于模型容量进一步增大，直接使用多个 RepVGGBlock 堆叠的 RepStageBlock 结构计算量和参数量呈现指数增长。因此，为了权衡计算负担和模型精度，在 M/L 模型中使用了 CSPBep 骨干网络，其采用 BepC3StageBlock 替换了小模型中的 RepStageBlock 。如下图所示，BepC3StageBlock 由 3 个 1×1 的 ConvModule 和多个子块（每个子块由两个 RepVGGBlock 残差连接）组成。

1.2.2 Neck

Neck 部分结构仍然在 YOLOv5 基础上进行了模块的改动，同样采用 RepStageBlock 或 BepC3StageBlock 对原本的 CSPLayer 进行了替换，需要注意的是，Neck 中 Down Sample 部分仍然使用了 stride=2 的 3×3 ConvModule，而不是像 Backbone 一样替换为 RepVGGBlock。

1.2.3 Head

不同于传统的 YOLO 系列检测头，YOLOv6 参考了 FCOS 和 YOLOX 中的做法，将分类和回归分支解耦成两个分支进行预测并且去掉了 obj 分支。同时，采用了 hybrid-channel 策略构建了更高效的解耦检测头，将中间 3×3 的 ConvModule 减少为 1 个，在维持精度的同时进一步减少了模型耗费，降低了推理延时。此外，需要说明的是，YOLOv6 在 Backobone 和 Neck 部分使用的激活函数是 ReLU，而在 Head 部分则使用的是 SiLU。

由于 YOLOv6 是解耦输出，分类和 bbox 检测通过不同卷积完成。以 COCO 80 类为例：

- P5 模型在输入为 640x640 分辨率情况下，其 Head 模块输出的 shape 分别为 (B,4,80,80)，(B,80,80,80)，(B,4,40,40)，(B,80,40,40)，(B,4,20,20)，(B,80,20,20)。

1.3 正负样本匹配策略

YOLOv6 采用的标签匹配策略与 TOOD 相同，前 4 个 epoch 采用 ATSSAssigner 作为标签匹配策略的 warm-up，后续使用 TaskAlignedAssigner 算法选择正负样本，基于官方开源代码，MMYOLO 中也对两个 assigner 算法进行了优化，改进为 Batch 维度进行计算，能够一定程度的加快速度。下面会对每个部分进行详细说明。

1.3.1 Anchor 设置

YOLOv6 采用与 YOLOX 一样的 Anchor-free 无锚范式，省略了聚类和繁琐的 Anchor 超参设定，泛化能力强，解码逻辑简单。在训练的过程中会根据 feature size 去自动生成先验框。

使用 mmdet.MlvlPointGenerator 生成 anchor points。

```
prior_generator: ConfigType = dict(
                    type='mmdet.MlvlPointGenerator',
                    offset=0.5, # 网格中心点
                    strides=[8, 16, 32]) ,

# 调用生成多层 anchor points: list[torch.Tensor]
# 每一层都是 (featrue_h*feature_w,4), 4 表示 (x,y,stride_h,stride_w)
self.mlvl_priors = self.prior_generator.grid_priors(
                self.featmap_sizes,
                with_stride=True)
```

1.3.2 Bbox 编解码过程

YOLOv6 的 BBox Coder 采用的是 DistancePointBBoxCoder。

网络 bbox 预测的值为 (top, bottom, left, right)，解码器将 anchor point 通过四个距离解码到坐标 (x1,y1,x2,y2)。

MMYOLO 中解码的核心源码：

```
def decode(points: torch.Tensor, pred_bboxes: torch.Tensor, stride: torch.Tensor) ->␣
↪torch.Tensor:
    """
        将预测值解码转化 bbox 的 xyxy
        points (Tensor): 生成的 anchor point [x, y], Shape (B, N, 2) or (N, 2).
        pred_bboxes (Tensor): 预测距离四边的距离。(left, top, right, bottom). Shape (B, N,
↪ 4) or (N, 4)
        stride (Tensor): 特征图下采样倍率.
    """
    # 首先将预测值转化为原图尺度
```

(下页继续)

(续上页)

```
    distance = pred_bboxes * stride[None, :, None]
    # 根据点以及到四条边距离转为 bbox 的 x1y1x2y2
    x1 = points[..., 0] - distance[..., 0]
    y1 = points[..., 1] - distance[..., 1]
    x2 = points[..., 0] + distance[..., 2]
    y2 = points[..., 1] + distance[..., 3]

    bboxes = torch.stack([x1, y1, x2, y2], -1)

    return bboxes
```

1.3.3 匹配策略

- 0 <= epoch < 4，使用 `BatchATSSAssigner`
- epoch >= 4，使用 `BatchTaskAlignedAssigner`

ATSSAssigner

ATSSAssigner 是 ATSS 中提出的标签匹配策略。ATSS 的匹配策略简单总结为：**通过中心点距离先验对样本进行初筛，然后自适应生成 IoU 阈值筛选正样本**。 YOLOv6 的实现种主要包括如下三个核心步骤：

1. 因为 YOLOv6 是 Anchor-free，所以首先将 `anchor point` 转化为大小为 `5*strdie` 的 `anchor`。
2. 对于每一个 `GT`，在 `FPN` 的每一个特征层上，计算与该层所有 `anchor` 中心点距离 (位置先验)，然后优先选取距离 `topK` 近的样本，作为 **初筛样本**。
3. 对于每一个 `GT`，计算其 **初筛样本**的 `IoU` 的均值 `mean` 与标准差 `std`，将 `mean + std` 作为该 `GT` 的正样本的 **自适应 IoU 阈值**，大于该 **自适应阈值**且中心点在 `GT` 内部的 `anchor` 才作为正样本，使得样本能够被 `assign` 到合适的 `FPN` 特征层上。

下图中，(a) 所示中等大小物体被 assign 到 FPN 的中层，(b) 所示偏大的物体被 assign 到 FPN 中检测大物体和偏大物体的两个层。

```
# 1. 首先将 anchor points 转化为 anchors
# priors 为 (point_x,point_y,stride_w,stride_h), shape 为 (N,4)
cell_half_size = priors[:, 2:] * 2.5
priors_gen = torch.zeros_like(priors)
priors_gen[:, :2] = priors[:, :2] - cell_half_size
priors_gen[:, 2:] = priors[:, :2] + cell_half_size
priors = priors_gen
# 2. 计算 anchors 与 GT 的 IoU
overlaps = self.iou_calculator(gt_bboxes.reshape([-1, 4]), priors)
# 3. 计算 anchor 与 GT 的中心距离
```

(下页继续)

(续上页)

```
distances, priors_points = bbox_center_distance(
        gt_bboxes.reshape([-1, 4]), priors)
# 4. 根据中心点距离，在 FPN 的每一层选取 TopK 临近的样本作为初筛样本
is_in_candidate, candidate_idxs = self.select_topk_candidates(
        distances, num_level_priors, pad_bbox_flag)
# 5. 对于每一个 GT 计算其对应初筛样本的均值与标准差的和，作为该 GT 的样本阈值
overlaps_thr_per_gt, iou_candidates = self.threshold_calculator(
        is_in_candidate, candidate_idxs, overlaps, num_priors, batch_size,
        num_gt)
# 6. 筛选大于阈值的样本作为正样本
is_pos = torch.where(
        iou_candidates > overlaps_thr_per_gt.repeat([1, 1, num_priors]),
        is_in_candidate, torch.zeros_like(is_in_candidate))
# 6. 保证样本中心点在 GT 内部且不超图像边界
pos_mask = is_pos * is_in_gts * pad_bbox_flag
```

TaskAlignedAssigner

TaskAlignedAssigner 是 TOOD 中提出的一种动态样本匹配策略。由于 ATSSAssigner 是属于静态标签匹配策略，其选取正样本的策略主要根据 anchor 的位置进行挑选，并不会随着网络的优化而选取到更好的样本。在目标检测中，分类和回归的任务最终作用于同一个目标，所以 TaskAlignedAssigner 认为样本的选取应该更加关注到对分类以及回归都友好的样本点。

TaskAlignedAssigner 的匹配策略简单总结为：**根据分类与回归的分数加权的分数选择正样本**。

1. 对于每一个 GT，对所有的 预测框基于 **GT 类别对应分类分数**与 **预测框与 GT 的 IoU** 的加权得到一个关联分类以及回归的对齐分数 alignment_metrics。
2. 对于每一个 GT，直接基于 alignment_metrics 对齐分数选取 topK 大的作为正样本。

因为在网络初期参数随机，分类分数和 预测框与 GT 的 IoU 都不准确，所以需要经过前 4 个 epoch 的 ATSSAssigner 的 warm-up。经过预热之后的 TaskAlignedAssigner 标签匹配策略就不使用中心距离的先验，而是直接对每一个 GT 选取 alignment_metrics 中 topK 大的样本作为正样本。

```
# 1. 基于分类分数与回归的 IoU 计算对齐分数 alignment_metrics
alignment_metrics = bbox_scores.pow(self.alpha) * overlaps.pow(
            self.beta)
# 2. 保证中心点在 GT 内部的 mask
is_in_gts = select_candidates_in_gts(priors, gt_bboxes)
# 3. 选取 TopK 大的对齐分数的样本
topk_metric = self.select_topk_candidates(
            alignment_metrics * is_in_gts,
            topk_mask=pad_bbox_flag.repeat([1, 1, self.topk]).bool())
```

1.4 Loss 设计

参与 Loss 计算的共有两个值：loss_cls 和 loss_bbox，其各自使用的 Loss 方法如下：

- Classes loss：使用的是 `mmdet.VarifocalLoss`
- BBox loss：l/m/s 使用的是 `GIoULoss`, t/n 用的是 `SIoULoss`

权重比例是：`loss_cls:loss_bbox=1 : 2.5`

分类损失函数 VarifocalLoss

Varifocal Loss (VFL) 是 VarifocalNet: An IoU-aware Dense Object Detector 中的损失函数。

`VFL` 是在 `GFL` 的基础上做的改进，`GFL` 详情请看*GFL 详解*

在上述标签匹配策略中提到过选择样本应该优先考虑分类回归都友好的样本点，这是由于目标检测包含的分类与回归两个子任务都是作用于同一个物体。与 `GFL` 思想相同，都是将 **预测框与 `GT` 的 `IoU` 软化作为分类的标签**，使得分类分数关联回归质量，使其在后处理 NMS 阶段有**分类回归一致性很强的分值排序策略**，以达到选取优秀预测框的目的。

Varifocal Loss 原本的公式：

$$VFL(p,q)=\begin{cases}-q(qlog(p)+(1-q)log(1-p)), & q>0\\ -\alpha p^{\gamma}log(1-p), & q=0\end{cases}$$

其中 q 是预测 `bboxes` 与 `GT` 的 `IoU`，使用软标签的形式作为分类的标签。$p\in[0,1]$ 表示分类分数。

1. 对于负样本，即当 $q=0$ 时，标准交叉熵部分为 $-\log(p)$，负样本权重使用 αp^{γ} 作为 `focal weight` 使样本聚焦与困难样本上，这与 `Focal Loss` 基本一致。
2. 对于正样本，即当 $q>0$ 时，首先计算标准二值交叉熵部分 $-(qlog(p)+(1-q)log(1-p))$，但是针对正样本的权重设置，`Varifocal Loss` 中并没有采用类似 αp^{γ} 的方式降权，而是认为在网络的学习过程中正样本相对于负样本的学习信号来说更为重要，所以使用了分类的标签 q，即 `IoU` 作为 `focal weight`, 使得聚焦到具有高质量的样本上。

但是 YOLOv6 中的 Varifocal Loss 公式采用 `TOOD` 中的 `Task ALignment Learning (TAL)`, 将预测的 `IoU` 根据之前标签匹配策略中的分类对齐度 `alignment_metrics` 进行了归一化, 得到归一化 $\hat{t}$。具体实现方式为：

对于每一个 `Gt`，找到所有样本中与 `Gt` 最大的 `IoU`，具有最大 `alignment_metrics` 的样本位置的 $\hat{t}=max(Iou)$

$$\hat{t}=AlignmentMetrics/max(AlignmentMetrics)*max(IoU)$$

最终 YOLOv6 分类损失损失函数为：

$$VFL(p,\hat{t})=\begin{cases}-\hat{t}(\hat{t}log(p)+(1-\hat{t})log(1-p)), & \hat{t}>0\\ -\alpha p^{\gamma}log(1-p), & \hat{t}=0\end{cases}$$

MMDetection 实现源码的核心部分：

```
def varifocal_loss(pred, target, alpha=0.75, gamma=2.0, iou_weighted=True):
    """
        pred (torch.Tensor): 预测的分类分数，形状为 (B,N,C) ，N 表示 anchor 数量， C 表示类别
数
        target (torch.Tensor): 经过对齐度归一化后的 IoU 分数，形状为 (B,N,C)，数值范围为 0~1
        alpha (float, optional): 调节正负样本之间的平衡因子，默认 0.75.
        gamma (float, optional): 负样本 focal 权重因子， 默认 2.0.
        iou_weighted (bool, optional): 正样本是否用 IoU 加权
    """
    pred_sigmoid = pred.sigmoid()
    target = target.type_as(pred)
    if iou_weighted:
        # 计算权重，正样本 (target > 0) 中权重为 target,
        # 负样本权重为 alpha*pred_simogid^2
        focal_weight = target * (target > 0.0).float() + \
            alpha * (pred_sigmoid - target).abs().pow(gamma) * \
            (target <= 0.0).float()
    else:
        focal_weight = (target > 0.0).float() + \
            alpha * (pred_sigmoid - target).abs().pow(gamma) * \
            (target <= 0.0).float()
    # 计算二值交叉熵后乘以权重
    loss = F.binary_cross_entropy_with_logits(
        pred, target, reduction='none') * focal_weight
    loss = weight_reduce_loss(loss, weight, reduction, avg_factor)
    return loss
```

回归损失函数 GIoU Loss / SIoU Loss

在 YOLOv6 中，针对不同大小的模型采用了不同的回归损失函数，其中 l/m/s 使用的是 `GIoULoss`, t/n 用的是 `SIoULoss`。

其中 `GIoULoss` 详情请看*GIoU 详解*。

SIou Loss

SIoU 损失函数是 SIoU Loss: More Powerful Learning for Bounding Box Regression 中提出的度量预测框与 `GT` 的匹配度的指标，由于之前的 `GIoU`, `CIoU`, `DIoU` 都没有考虑预测框向 `GT` 框回归的角度，然而角度也确实是回归中一个重要的影响因素，因此提出了全新的 `SIoU`。

SIoU 损失主要由四个度量方面组成：

- IoU 成本
- 角度成本

- 距离成本
- 形状成本

如下图所示，**角度成本**就是指图中预测框 B 向 B^{GT} 的回归过程中，尽可能去使得优化过程中的不确定性因素减少，比如现将图中的角度 α 或者 β 变为 0，再去沿着 x 轴或者 y 轴去回归边界。

MMYOLO 实现源码的核心部分：

```
def bbox_overlaps(bboxes1, bboxes2, mode='siou', is_aligned=False, eps=1e-6):
    # 两个 box 的顶点 x1,y1,x2,y2
    bbox1_x1, bbox1_y1 = pred[:, 0], pred[:, 1]
    bbox1_x2, bbox1_y2 = pred[:, 2], pred[:, 3]
    bbox2_x1, bbox2_y1 = target[:, 0], target[:, 1]
    bbox2_x2, bbox2_y2 = target[:, 2], target[:, 3]
    # 交集
    overlap = (torch.min(bbox1_x2, bbox2_x2) -
               torch.max(bbox1_x1, bbox2_x1)).clamp(0) * \
              (torch.min(bbox1_y2, bbox2_y2) -
               torch.max(bbox1_y1, bbox2_y1)).clamp(0)
    # 并集
    w1, h1 = bbox1_x2 - bbox1_x1, bbox1_y2 - bbox1_y1
    w2, h2 = bbox2_x2 - bbox2_x1, bbox2_y2 - bbox2_y1
    union = (w1 * h1) + (w2 * h2) - overlap + eps
    # IoU = 交集/并集
    ious = overlap / union
    # 最小外界矩的宽高
    enclose_x1y1 = torch.min(pred[:, :2], target[:, :2])
    enclose_x2y2 = torch.max(pred[:, 2:], target[:, 2:])
    enclose_wh = (enclose_x2y2 - enclose_x1y1).clamp(min=0)
    enclose_w = enclose_wh[:, 0]  # enclose_w
    enclose_h = enclose_wh[:, 1]  # enclose_h
    elif iou_mode == 'siou':
        # 1. 计算 σ （两个 box 中心点距离）:
        # sigma_cw, sigma_ch: 上图中 cw,ch
        sigma_cw = (bbox2_x1 + bbox2_x2) / 2 - (bbox1_x1 + bbox1_x2) / 2 + eps
        sigma_ch = (bbox2_y1 + bbox2_y2) / 2 - (bbox1_y1 + bbox1_y2) / 2 + eps
        sigma = torch.pow(sigma_cw**2 + sigma_ch**2, 0.5)

        # 2. 在 α 和 β 中选择一个小的角度（小于 π/4）去优化
        sin_alpha = torch.abs(sigma_ch) / sigma
        sin_beta = torch.abs(sigma_cw) / sigma
        sin_alpha = torch.where(sin_alpha <= math.sin(math.pi / 4), sin_alpha,
                                sin_beta)
```

(下页继续)

(续上页)

```
        # 角度损失 = 1 - 2 * ( sin^2 ( arcsin(x) - (π / 4) ) ) = cos(2α-π/2) = sin(2α)
        # 这里就是角度损失，当 α=0 或者 α=90° 时损失为 0，当 α=45° 损失为 1
        angle_cost = torch.cos(torch.arcsin(sin_alpha) * 2 - math.pi / 2)

        # 3. 这里将角度损失与距离损失进行融合
        # Distance cost = Σ_(t=x,y) (1 - e ^ (- γ ρ_t))
        rho_x = (sigma_cw / enclose_w)**2  # ρ_x:x 轴中心点距离距离损失
        rho_y = (sigma_ch / enclose_h)**2  # ρ_y:y 轴中心点距离距离损失
        gamma = 2 - angle_cost  # γ
        # 当 α=0，angle_cost=0，gamma=2，dis_cost_x =  1 - e ^ (-2 p_x)，因为 ρ_x>0，主要优化距离
        # 当 α=45°，angle_cost=1，gamma=1，dis_cost_x =  1 - e ^ (-1* p_x)，因为 ρ_x<1，主要优化角度
        distance_cost = (1 - torch.exp(-1 * gamma * rho_x)) + (
            1 - torch.exp(-1 * gamma * rho_y))

        # 4. 形状损失 就是两个 box 之间的宽高比
        # Shape cost = Ω = Σ_(t=w,h) ( ( 1 - ( e ^ (-ω_t) ) ) ^ θ )
        omiga_w = torch.abs(w1 - w2) / torch.max(w1, w2)  # ω_w
        omiga_h = torch.abs(h1 - h2) / torch.max(h1, h2)  # ω_h
        shape_cost = torch.pow(1 - torch.exp(-1 * omiga_w),
                               siou_theta) + torch.pow(
                                   1 - torch.exp(-1 * omiga_h), siou_theta)

        # 5. 综合 IoU、角度、距离以及形状信息
        # SIoU = IoU - ( (Distance Cost + Shape Cost) / 2 )
        ious = ious - ((distance_cost + shape_cost) * 0.5)

    return ious.clamp(min=-1.0, max=1.0)

@weighted_loss
def siou_loss(pred, target, eps=1e-7):
    sious = bbox_overlaps(pred, target, mode='siou', is_aligned=True, eps=eps)
    loss = 1 - sious
    return loss
```

Object Loss

在 YOLOv6 中，由于额外的置信度预测头可能与 `Aligned Head` 有所冲突，经实验验证在不同大小的模型上也都有掉点，所以最后选择弃用 `Objectness` 分支。

1.5 优化策略和训练过程

1.5.1 优化器分组

与 YOLOv5 一致，详情请看*YOLOv5 优化器分组*

1.5.2 weight decay 参数自适应

与 YOLOv5 一致，详情请看*YOLOv5 weight decay 参数自适应*

1.6 推理和后处理过程

YOLOv6 后处理过程和 YOLOv5 高度类似，实际上 YOLO 系列的后处理逻辑都是类似的。详情请看*YOLOv5 推理和后处理过程*

11.2.3 2 总结

本文对 YOLOv6 原理和在 MMYOLO 实现进行了详细解析，希望能帮助用户理解算法实现过程。同时请注意：由于 YOLOv6 本身也在不断更新，本开源库也会不断迭代，请及时阅读和同步最新版本。

11.3 RTMDet 原理和实现全解析

11.3.1 0 简介

高性能，低延时的单阶段目标检测器

以上结构图由 RangeKing@github 绘制。

最近一段时间，开源界涌现出了大量的高精度目标检测项目，其中最突出的就是 YOLO 系列，OpenMMLab 也在与社区的合作下推出了 MMYOLO。在调研了当前 YOLO 系列的诸多改进模型后，MMDetection 核心开发者针对这些设计以及训练方式进行了经验性的总结，并进行了优化，推出了高精度、低延时的单阶段目标检测器 RTMDet, **R**eal-**t**ime **M**odels for Object **Det**ection (**R**elease **t**o **M**anufacture)

RTMDet 由 tiny/s/m/l/x 一系列不同大小的模型组成，为不同的应用场景提供了不同的选择。其中，RTMDet-x 在 52.6 mAP 的精度下达到了 300+ FPS 的推理速度。

注解： 注：推理速度和精度测试（不包含 NMS）是在 1 块 NVIDIA 3090 GPU 上的 `TensorRT 8.4.3, cuDNN 8.2.0, FP16, batch size=1` 条件里测试的。

而最轻量的模型 RTMDet-tiny，在仅有 4M 参数量的情况下也能够达到 40.9 mAP，且推理速度 < 1 ms。

上图中的精度是和 300 epoch 训练下的公平对比，为不使用蒸馏的结果。

- 官方开源地址：https://github.com/open-mmlab/mmdetection/blob/3.x/configs/rtmdet/README.md
- MMYOLO 开源地址：https://github.com/open-mmlab/mmyolo/blob/main/configs/rtmdet/README.md

11.3.2 1 v1.0 算法原理和 MMYOLO 实现解析

1.1 数据增强模块

RTMDet 采用了多种数据增强的方式来增加模型的性能，主要包括单图数据增强:

- **RandomResize 随机尺度变换**
- **RandomCrop 随机裁剪**
- **HSVRandomAug 颜色空间增强**
- **RandomFlip 随机水平翻转**

以及混合类数据增强：

- **Mosaic 马赛克**
- **MixUp 图像混合**

数据增强流程如下：

其中 RandomResize 超参在大模型 M,L,X 和小模型 S, Tiny 上是不一样的，大模型由于参数较多，可以使用 large scale jitter 策略即参数为 (0.1,2.0)，而小模型采用 stand scale jitter 策略即 (0.5, 2.0) 策略。MMDetection 开源库中已经对单图数据增强进行了封装，用户通过简单的修改配置即可使用库中提供的任何数据增强功能，且都是属于比较常规的数据增强，不需要特殊介绍。下面将具体介绍混合类数据增强的具体实现。

与 YOLOv5 不同的是，YOLOv5 认为在 S 和 Nano 模型上使用 MixUp 是过剩的，小模型不需要这么强的数据增强。而 RTMDet 在 S 和 Tiny 上也使用了 MixUp，这是因为 RTMDet 在最后 20 epoch 会切换为正常的 aug，并通过训练证明这个操作是有效的。并且 RTMDet 为混合类数据增强引入了 Cache 方案，有效地减少了图像处理的时间, 和引入了可调超参 `max_cached_images` ，当使用较小的 cache 时，其效果类似 `repeated augmentation`。具体介绍如下：

1.1.1 为图像混合数据增强引入 Cache

Mosaic&MixUp 涉及到多张图片的混合，它们的耗时会是普通数据增强的 K 倍 (K 为混入图片的数量)。如在 YOLOv5 中，每次做 Mosaic 时，4 张图片的信息都需要从硬盘中重新加载。而 RTMDet 只需要重新载入当前的一张图片，其余参与混合增强的图片则从缓存队列中获取，通过牺牲一定内存空间的方式大幅提升了效率。另外通过调整 cache 的大小以及 pop 的方式，也可以调整增强的强度。

如图所示，cache 队列中预先储存了 N 张已加载的图像与标签数据，每一个训练 step 中只需加载一张新的图片及其标签数据并更新到 cache 队列中 (cache 队列中的图像可重复，如图中出现两次 img3)，同时如果 cache 队列长度超过预设长度，则随机 pop 一张图（为了 Tiny 模型训练更稳定，在 Tiny 模型中不采用随机 pop 的方式, 而是移除最先加入的图片），当需要进行混合数据增强时，只需要从 cache 中随机选择需要的图像进行拼接等处理，而不需要全部从硬盘中加载，节省了图像加载的时间。

注解： cache 队列的最大长度 N 为可调整参数，根据经验性的原则，当为每一张需要混合的图片提供十个缓存时，可以认为提供了足够的随机性，而 Mosaic 增强是四张图混合，因此 cache 数量默认 N=40，同理 MixUp 的 cache 数量默认为 20，tiny 模型需要更稳定的训练条件，因此其 cache 数量也为其余规格模型的一半（MixUp 为 10，Mosaic 为 20）

在具体实现中，MMYOLO 设计了 `BaseMiximageTransform` 类来支持多张图像混合数据增强：

```
if self.use_cached:
    # Be careful: deep copying can be very time-consuming
    # if results includes dataset.
    dataset = results.pop('dataset', None)
    self.results_cache.append(copy.deepcopy(results))  # 将当前加载的图片数据缓存到 cache␣
↪中
    if len(self.results_cache) > self.max_cached_images:
        if self.random_pop: # 除了 tiny 模型，self.random_pop=True
            index = random.randint(0, len(self.results_cache) - 1)
        else:
            index = 0
        self.results_cache.pop(index)

    if len(self.results_cache) <= 4:
        return results
else:
    assert 'dataset' in results
    # Be careful: deep copying can be very time-consuming
    # if results includes dataset.
    dataset = results.pop('dataset', None)
```

1.1.2 Mosaic

Mosaic 是将 4 张图拼接为 1 张大图，相当于变相的增加了 batch size，具体步骤为：

1. 根据索引随机从自定义数据集中再采样 3 个图像，可能重复

```
def get_indexes(self, dataset: Union[BaseDataset, list]) -> list:
    """Call function to collect indexes.

    Args:
        dataset (:obj:`Dataset` or list): The dataset or cached list.

    Returns:
        list: indexes.
    """
    indexes = [random.randint(0, len(dataset)) for _ in range(3)]
    return indexes
```

2. 随机选出 4 幅图像相交的中点。

```
# mosaic center x, y
center_x = int(
    random.uniform(*self.center_ratio_range) * self.img_scale[1])
center_y = int(
    random.uniform(*self.center_ratio_range) * self.img_scale[0])
center_position = (center_x, center_y)
```

3. 根据采样的 index 读取图片并拼接, 拼接前会先进行 `keep-ratio` 的 resize 图片 (即为最大边一定是 640)。

```
# keep_ratio resize
scale_ratio_i = min(self.img_scale[0] / h_i,
                    self.img_scale[1] / w_i)
img_i = mmcv.imresize(
    img_i, (int(w_i * scale_ratio_i), int(h_i * scale_ratio_i)))
```

4. 拼接后，把 bbox 和 label 全部拼接起来，然后对 bbox 进行裁剪但是不过滤 (可能出现一些无效框)

```
mosaic_bboxes.clip_([2 * self.img_scale[0], 2 * self.img_scale[1]])
```

更多的关于 Mosaic 原理的详情可以参考*YOLOv5 原理和实现全解析* 中的 Mosaic 原理分析。

1.1.3 MixUp

RTMDet 的 MixUp 实现方式与 YOLOX 中一样，只不过增加了类似上文中提到的 cache 功能。

更多的关于 MixUp 原理的详情也可以参考*YOLOv5 原理和实现全解析* 中的 MixUp 原理分析。

1.1.4 强弱两阶段训练

Mosaic+MixUp 失真度比较高，持续用太强的数据增强对模型并不一定有益。YOLOX 中率先使用了强弱两阶段的训练方式，但由于引入了旋转，错切导致 box 标注产生误差，需要在第二阶段引入额外的 L1 loss 来纠正回归分支的性能。

为了使数据增强的方式更为通用，RTMDet 在前 280 epoch 使用不带旋转的 Mosaic+MixUp, 且通过混入 8 张图片来提升强度以及正样本数。后 20 epoch 使用比较小的学习率在比较弱的增强下进行微调，同时在 EMA 的作用下将参数缓慢更新至模型，能够得到比较大的提升。

1.2 模型结构

RTMDet 模型整体结构和 YOLOX 几乎一致，由 `CSPNeXt` + `CSPNeXtPAFPN` + 共享卷积权重但分别计算 BN 的 `SepBNHead` 构成。内部核心模块也是 `CSPLayer`，但对其中的 `Basic Block` 进行了改进，提出了 `CSPNeXt Block`。

1.2.1 Backbone

`CSPNeXt` 整体以 `CSPDarknet` 为基础，共 5 层结构，包含 1 个 `Stem Layer` 和 4 个 `Stage Layer`:

- `Stem Layer` 是 3 层 3x3 kernel 的 `ConvModule` ，不同于之前的 `Focus` 模块或者 1 层 6x6 kernel 的 `ConvModule` 。
- `Stage Layer` 总体结构与已有模型类似，前 3 个 `Stage Layer` 由 1 个 `ConvModule` 和 1 个 `CSPLayer` 组成。第 4 个 `Stage Layer` 在 `ConvModule` 和 `CSPLayer` 中间增加了 `SPPF` 模块 (MMDetection 版本为 `SPP` 模块)。
- 如模型图 Details 部分所示，`CSPLayer` 由 3 个 `ConvModule` + n 个 `CSPNeXt Block`(带残差连接) + 1 个 `Channel Attention` 模块组成。`ConvModule` 为 1 层 3x3 `Conv2d` + `BatchNorm` + `SiLU` 激活函数。`Channel Attention` 模块为 1 层 `AdaptiveAvgPool2d` + 1 层 1x1 `Conv2d` + `Hardsigmoid` 激活函数。`CSPNeXt Block` 模块在下节详细讲述。
- 如果想阅读 Backbone - `CSPNeXt` 的源码，可以 **点此** 跳转。

1.2.2 CSPNeXt Block

Darknet（图 a）使用 1x1 与 3x3 卷积的 `Basic Block`。YOLOv6 、YOLOv7 、PPYOLO-E（图 b & c）使用了重参数化 Block。但重参数化的训练代价高，且不易量化，需要其他方式来弥补量化误差。RTMDet 则借鉴了最近比较热门的 ConvNeXt 、RepLKNet 的做法，为 `Basic Block` 加入了大 kernel 的 `depth-wise` 卷积（图 d），并将其命名为 `CSPNeXt Block`。

关于不同 kernel 大小的实验结果，如下表所示。

如果想阅读 `Basic Block` 和 `CSPNeXt Block` 源码，可以**点此**跳转。

1.2.3 调整检测器不同 stage 间的 block 数

由于 `CSPNeXt Block` 内使用了 `depth-wise` 卷积，单个 block 内的层数增多。如果保持原有的 stage 内的 block 数，则会导致模型的推理速度大幅降低。

RTMDet 重新调整了不同 stage 间的 block 数，并调整了通道的超参，在保证了精度的情况下提升了推理速度。

关于不同 block 数的实验结果，如下表所示。

最后不同大小模型的 block 数设置，可以参见源码 。

1.2.4 Neck

Neck 模型结构和 YOLOX 几乎一样，只不过内部的 block 进行了替换。

1.2.5 Backbone 与 Neck 之间的参数量和计算量的均衡

EfficientDet 、NASFPN 等工作在改进 Neck 时往往聚焦于如何修改特征融合的方式。但引入过多的连接会增加检测器的延时，并增加内存开销。

所以 RTMDet 选择不引入额外的连接，而是改变 Backbone 与 Neck 间参数量的配比。该配比是通过手动调整 Backbone 和 Neck 的 `expand_ratio` 参数来实现的，其数值在 Backbone 和 Neck 中都为 0.5。`expand_ratio` 实际上是改变 `CSPLayer` 中各层通道数的参数（具体可见模型图 `CSPLayer` 部分）。如果想进行不同配比的实验，可以通过调整配置文件中的 backbone {expand_ratio} 和 neck {expand_ratio} 参数完成。

实验发现，当 Neck 在整个模型中的参数量占比更高时，延时更低，且对精度的影响很小。作者在直播答疑时回复，RTMDet 在 Neck 这一部分的实验参考了 GiraffeDet 的做法，但没有像 GiraffeDet 一样引入额外连接（详细可参见 RTMDet 发布视频 31 分 40 秒左右的内容）。

关于不同参数量配比的实验结果，如下表所示。

如果想阅读 Neck - `CSPNeXtPAFPN` 的源码，可以**点此** 跳转。

1.2.6 Head

传统的 YOLO 系列都使用同一 Head 进行分类和回归。YOLOX 则将分类和回归分支解耦，PPYOLO-E 和 YOLOv6 则引入了 TOOD 中的结构。它们在不同特征层级之间都使用独立的 Head，因此 Head 在模型中也占有较多的参数量。

RTMDet 参考了 NAS-FPN 中的做法，使用了 SepBNHead，在不同层之间共享卷积权重，但是独立计算 BN（BatchNorm）的统计量。

关于不同结构 Head 的实验结果，如下表所示。

同时，RTMDet 也延续了作者之前在 NanoDet 中的思想，使用 Quality Focal Loss ，并去掉 Objectness 分支，进一步将 Head 轻量化。

如果想阅读 Head 中 RTMDetSepBNHeadModule 的源码，可以**点此** 跳转。

注解： 注：MMYOLO 和 MMDetection 中 Neck 和 Head 的具体实现稍有不同。

1.3 正负样本匹配策略

正负样本匹配策略或者称为标签匹配策略 Label Assignment 是目标检测模型训练中最核心的问题之一，更好的标签匹配策略往往能够使得网络更好学习到物体的特征以提高检测能力。

早期的样本标签匹配策略一般都是基于 *空间以及尺度信息的先验* 来决定样本的选取。典型案例如下：

- FCOS 中先限定网格中心点在 GT 内筛选后然后再通过不同特征层限制尺寸来决定正负样本
- RetinaNet 则是通过 Anchor 与 GT 的最大 IOU 匹配来划分正负样本
- YOLOV5 的正负样本则是通过样本的宽高比先筛选一部分，然后通过位置信息选取 GT 中心落在的 Grid 以及临近的两个作为正样本

但是上述方法都是属于基于 *先验* 的静态匹配策略，就是样本的选取方式是根据人的经验规定的。不会随着网络的优化而进行自动优化选取到更好的样本，近些年涌现了许多优秀的动态标签匹配策略：

- OTA 提出使用 Sinkhorn 迭代求解匹配中的最优传输问题
- YOLOX 中使用 OTA 的近似算法 SimOTA，TOOD 将分类分数以及 IOU 相乘计算 Cost 矩阵进行标签匹配等等

这些算法将 *预测的 Bboxes 与 GT 的 IOU 和分类分数* 或者是对应 *分类 Loss* 和 *回归 Loss* 拿来计算 Matching Cost 矩阵再通过 top-k 的方式动态决定样本选取以及样本个数。通过这种方式，在网络优化的过程中会自动选取对分类或者回归更加敏感有效的位置的样本，它不再只依赖先验的静态的信息，而是使用当前的预测结果去动态寻找最优的匹配，只要模型的预测越准确，匹配算法求得的结果也会更优秀。但是在网络训练的初期，网络的分类以及回归是随机初始化，这个时候还是需要 *先验* 来约束，以达到 *冷启动* 的效果。

RTMDet 作者也是采用了动态的 SimOTA 做法，不过其对动态的正负样本分配策略进行了改进。之前的动态匹配策略（HungarianAssigner 、OTA ）往往使用与 Loss 完全一致的代价函数作为匹配的依据，但我们

经过实验发现这并不一定时最优的。使用更多 Soften 的 Cost 以及先验，能够提升性能。

1.3.1 Bbox 编解码过程

RTMDet 的 BBox Coder 采用的是 mmdet.DistancePointBBoxCoder。

该类的 docstring 为 This coder encodes gt bboxes (x1, y1, x2, y2) into (top, bottom, left, right) and decode it back to the original.

编码器将 gt bboxes (x1, y1, x2, y2) 编码为 (top, bottom, left, right)，并且解码至原图像上。

MMDet 编码的核心源码：

```
def bbox2distance(points: Tensor, bbox: Tensor, ...) -> Tensor:
    """
        points (Tensor): 相当于 scale 值 stride ，且每个预测点仅为一个正方形 anchor 的␣
↪anchor point [x, y], Shape (n, 2) or (b, n, 2).
        bbox (Tensor): Bbox 为乘上 stride 的网络预测值，格式为 xyxy, Shape (n, 4) or (b, n,
↪ 4).
    """
    # 计算点距离四边的距离
    left = points[..., 0] - bbox[..., 0]
    top = points[..., 1] - bbox[..., 1]
    right = bbox[..., 2] - points[..., 0]
    bottom = bbox[..., 3] - points[..., 1]

    ...

    return torch.stack([left, top, right, bottom], -1)
```

MMDetection 解码的核心源码：

```
def distance2bbox(points: Tensor, distance: Tensor, ...) -> Tensor:
    """
        通过距离反算 bbox 的 xyxy
        points (Tensor): 正方形的预测 anchor 的 anchor point [x, y], Shape (B, N, 2) or␣
↪(N, 2).
        distance (Tensor): 距离四边的距离。(left, top, right, bottom). Shape (B, N, 4)␣
↪or (N, 4)
    """

    # 反算 bbox xyxy
    x1 = points[..., 0] - distance[..., 0]
    y1 = points[..., 1] - distance[..., 1]
    x2 = points[..., 0] + distance[..., 2]
    y2 = points[..., 1] + distance[..., 3]
```

(下页继续)

(续上页)

```
    bboxes = torch.stack([x1, y1, x2, y2], -1)

    ...

    return bboxes
```

1.3.2 匹配策略

RTMDet 提出了 Dynamic Soft Label Assigner 来实现标签的动态匹配策略, 该方法主要包括使用 **位置先验信息损失**, **样本回归损失**, **样本分类损失**, 同时对三个损失进行了 Soft 处理进行参数调优, 以达到最佳的动态匹配效果。

该方法 Matching Cost 矩阵由如下损失构成:

```
cost_matrix = soft_cls_cost + iou_cost + soft_center_prior
```

1. Soft_Center_Prior

$$C\_center = alpha^{|x\_pred - x\_gt| - beta}$$

```
# valid_prior Tensor[N,4] 表示 anchor point
# 4 分别表示 x, y, 以及对应的特征层的 stride, stride
gt_center = (gt_bboxes[:, :2] + gt_bboxes[:, 2:]) / 2.0
valid_prior = priors[valid_mask]
strides = valid_prior[:, 2]
# 计算 gt 与 anchor point 的中心距离并转换到特征图尺度
distance = (valid_prior[:, None, :2] - gt_center[None, :, :]
                    ).pow(2).sum(-1).sqrt() / strides[:, None]
# 以 10 为底计算位置的软化损失, 限定在 gt 的 6 个单元格以内
soft_center_prior = torch.pow(10, distance - 3)
```

2. IOU_Cost

$$C\_reg = -log(IOU)$$

```
# 计算回归 bboxes 和 gts 的 iou
pairwise_ious = self.iou_calculator(valid_decoded_bbox, gt_bboxes)
# 将 iou 使用 log 进行 soft , iou 越小 cost 更小
iou_cost = -torch.log(pairwise_ious + EPS) * 3
```

3. Soft_Cls_Cost

$$C\_cls = CE(P, Y\_soft)(Y\_soft - P)^2$$

```
# 生成分类标签
 gt_onehot_label = (
    F.one_hot(gt_labels.to(torch.int64),
              pred_scores.shape[-1]).float().unsqueeze(0).repeat(
                  num_valid, 1, 1))
valid_pred_scores = valid_pred_scores.unsqueeze(1).repeat(1, num_gt, 1)
# 不单单将分类标签为 01, 而是换成与 gt 的 iou
soft_label = gt_onehot_label * pairwise_ious[..., None]
# 使用 quality focal loss 计算分类损失 cost , 与实际的分类损失计算保持一致
scale_factor = soft_label - valid_pred_scores.sigmoid()
soft_cls_cost = F.binary_cross_entropy_with_logits(
    valid_pred_scores, soft_label,
    reduction='none') * scale_factor.abs().pow(2.0)
soft_cls_cost = soft_cls_cost.sum(dim=-1)
```

通过计算上述三个损失的和得到最终的 cost_matrix 后, 再使用 SimOTA 决定每一个 GT 匹配的样本的个数并决定最终的样本。具体操作如下所示：

1. 首先通过自适应计算每一个 gt 要选取的样本数量：取每一个 gt 与所有 bboxes 前 13 大的 iou, 得到它们的和取整后作为这个 gt 的 样本数目, 最少为 1 个, 记为 dynamic_ks。
2. 对于每一个 gt , 将其 cost_matrix 矩阵前 dynamic_ks 小的位置作为该 gt 的正样本。
3. 对于某一个 bbox, 如果被匹配到多个 gt 就将与这些 gts 的 cost_marix 中最小的那个作为其 label。

在网络训练初期，因参数初始化，回归和分类的损失值 Cost 往往较大, 这时候 IOU 比较小，选取的样本较少，主要起作用的是 Soft_center_prior 也就是位置信息，优先选取位置距离 GT 比较近的样本作为正样本，这也符合人们的理解，在网络前期给少量并且有足够质量的样本，以达到冷启动。当网络进行训练一段时间过后，分类分支和回归分支都进行了一定的优化后，这时 IOU 变大，选取的样本也逐渐增多，这时网络也有能力学习到更多的样本，同时因为 IOU_Cost 以及 Soft_Cls_Cost 变小，网络也会动态的找到更有利优化分类以及回归的样本点。

在 Resnet50-1x 的三种损失的消融实验：

与其他主流 Assign 方法在 Resnet50-1x 的对比实验：

无论是 Resnet50-1x 还是标准的设置下，还是在 300epoch + havy augmentation, 相比于 SimOTA 、OTA 以及 TOOD 中的 TAL 均有提升。

1.4 Loss 设计

参与 Loss 计算的共有两个值：`loss_cls` 和 `loss_bbox`，其各自使用的 Loss 方法如下：

- `loss_cls`：`mmdet.QualityFocalLoss`
- `loss_bbox`：`mmdet.GIoULoss`

权重比例是：`loss_cls:loss_bbox = 1 : 2`

QualityFocalLoss

Quality Focal Loss (QFL) 是 Generalized Focal Loss: Learning Qualified and Distributed Bounding Boxes for Dense Object Detection 的一部分。

普通的 Focal Loss 公式：

$$FL(p) = -(1-p_t)^{\gamma}\log(p_t), p_t = \begin{cases} p, & when\ y = 1 \\ 1-p, & when\ y = 0 \end{cases}$$

其中 $y \in 1, 0$ 指定真实类，$p \in [0, 1]$ 表示标签 $y = 1$ 的类估计概率。γ 是可调聚焦参数。具体来说，FL 由标准交叉熵部分 $-\log(p_t)$ 和动态比例因子部分 $-(1-p_t)^{\gamma}$ 组成，其中比例因子 $-(1-p_t)^{\gamma}$ 在训练期间自动降低简单类对于 loss 的比重，并且迅速将模型集中在困难类上。

首先 $y = 0$ 表示质量得分为 0 的负样本，$0 < y \leq 1$ 表示目标 IoU 得分为 y 的正样本。为了针对连续的标签，扩展 FL 的两个部分：

1. 交叉熵部分 $-\log(p_t)$ 扩展为完整版本 $-((1-y)\log(1-\sigma) + y\log(\sigma))$
2. 比例因子部分 $-(1-p_t)^{\gamma}$ 被泛化为估计 γ 与其连续标签 y 的绝对距离，即 $|y-\sigma|^{\beta}(\beta \geq 0)$ 。

结合上面两个部分之后，我们得出 QFL 的公式：

$$QFL(\sigma) = -|y-\sigma|^{\beta}((1-y)\log(1-\sigma) + y\log(\sigma))$$

具体作用是：可以将离散标签的 `focal loss` 泛化到连续标签上，将 bboxes 与 gt 的 IoU 的作为分类分数的标签，使得分类分数为表征回归质量的分数。

MMDetection 实现源码的核心部分：

```
@weighted_loss
def quality_focal_loss(pred, target, beta=2.0):
    """
        pred (torch.Tensor): 用形状 (N, C) 联合表示预测分类和质量 (IoU)，C 是类的数量。
        target (tuple([torch.Tensor])): 目标类别标签的形状为 (N,)，目标质量标签的形状是 (N,)。
        beta (float): 计算比例因子的 β 参数.
    """
    ...
```

(下页继续)

(续上页)

```
    # label 表示类别 id, score 表示质量分数
    label, score = target

    # 负样本质量分数 0 来进行监督
    pred_sigmoid = pred.sigmoid()
    scale_factor = pred_sigmoid
    zerolabel = scale_factor.new_zeros(pred.shape)

    # 计算交叉熵部分的值
    loss = F.binary_cross_entropy_with_logits(
        pred, zerolabel, reduction='none') * scale_factor.pow(beta)

    # 得出 IoU 在区间 (0,1] 的 bbox
    # FG cat_id: [0, num_classes -1], BG cat_id: num_classes
    bg_class_ind = pred.size(1)
    pos = ((label >= 0) & (label < bg_class_ind)).nonzero().squeeze(1)
    pos_label = label[pos].long()

    # 正样本由 IoU 范围在 (0,1] 的 bbox 来监督
    # 计算动态比例因子
    scale_factor = score[pos] - pred_sigmoid[pos, pos_label]

    # 计算两部分的 loss
    loss[pos, pos_label] = F.binary_cross_entropy_with_logits(
        pred[pos, pos_label], score[pos],
        reduction='none') * scale_factor.abs().pow(beta)

    # 得出最终 loss
    loss = loss.sum(dim=1, keepdim=False)
    return loss
```

GIoULoss

论文：Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression

GIoU Loss 用于计算两个框重叠区域的关系，重叠区域越大，损失越小，反之越大。而且 GIoU 是在 [0,2] 之间，因为其值被限制在了一个较小的范围内，所以网络不会出现剧烈的波动，证明了其具有比较好的稳定性。

下图是基本的实现流程图：

MMDetection 实现源码的核心部分：

```
def bbox_overlaps(bboxes1, bboxes2, mode='iou', is_aligned=False, eps=1e-6):
    ...
```

(下页继续)

(续上页)

```
    # 求两个区域的面积
    area1 = (bboxes1[..., 2] - bboxes1[..., 0]) * (
        bboxes1[..., 3] - bboxes1[..., 1])
    area2 = (bboxes2[..., 2] - bboxes2[..., 0]) * (
        bboxes2[..., 3] - bboxes2[..., 1])

    if is_aligned:
        # 得出两个 bbox 重合的左上角 lt 和右下角 rb
        lt = torch.max(bboxes1[..., :2], bboxes2[..., :2])  # [B, rows, 2]
        rb = torch.min(bboxes1[..., 2:], bboxes2[..., 2:])  # [B, rows, 2]

        # 求重合面积
        wh = fp16_clamp(rb - lt, min=0)
        overlap = wh[..., 0] * wh[..., 1]

        if mode in ['iou', 'giou']:
            ...
        else:
            union = area1
        if mode == 'giou':
            # 得出两个 bbox 最小凸闭合框的左上角 lt 和右下角 rb
            enclosed_lt = torch.min(bboxes1[..., :2], bboxes2[..., :2])
            enclosed_rb = torch.max(bboxes1[..., 2:], bboxes2[..., 2:])
    else:
        ...

    # 求重合面积 / gt bbox 面积 的比率，即 IoU
    eps = union.new_tensor([eps])
    union = torch.max(union, eps)
    ious = overlap / union

    ...

    # 求最小凸闭合框面积
    enclose_wh = fp16_clamp(enclosed_rb - enclosed_lt, min=0)
    enclose_area = enclose_wh[..., 0] * enclose_wh[..., 1]
    enclose_area = torch.max(enclose_area, eps)

    # 计算 giou
    gious = ious - (enclose_area - union) / enclose_area
    return gious
```

(下页继续)

(续上页)

```
@weighted_loss
def giou_loss(pred, target, eps=1e-7):
    gious = bbox_overlaps(pred, target, mode='giou', is_aligned=True, eps=eps)
    loss = 1 - gious
    return loss
```

1.5 优化策略和训练过程

1.6 推理和后处理过程

(1) 特征图输入

预测的图片输入大小为 640 x 640, 通道数为 3 , 经过 CSPNeXt, CSPNeXtPAFPN 层的 8 倍、16 倍、32 倍下采样得到 80 x 80, 40 x 40, 20 x 20 三个尺寸的特征图。以 rtmdet-l 模型为例，此时三层通道数都为 256，经过 `bbox_head` 层得到两个分支，分别为 `rtm_cls` 类别预测分支，将通道数从 256 变为 80，80 对应所有类别数量; `rtm_reg` 边框回归分支将通道数从 256 变为 4，4 代表框的坐标。

(2) 初始化网格

根据特征图尺寸初始化三个网格，大小分别为 6400 (80 x 80)、1600 (40 x 40)、400 (20 x 20)，如第一个层 shape 为 torch.Size([6400, 2])，最后一个维度是 2，为网格点的横纵坐标，而 6400 表示当前特征层的网格点数量。

(3) 维度变换

经过 `_predict_by_feat_single` 函数，将从 head 提取的单一图像的特征转换为 bbox 结果输入，得到三个列表 `cls_score_list`，`bbox_pred_list`，`mlvl_priors`，详细大小如图所示。之后分别遍历三个特征层，分别对 class 类别预测分支、bbox 回归分支进行处理。以第一层为例，对 bbox 预测分支 [4，80，80] 维度变换为 [6400，4]，对类别预测分支 [80，80，80] 变化为 [6400，80]，并对其做归一化，确保类别置信度在 0 - 1 之间。

(4) 阈值过滤

先使用一个 `nms_pre` 操作，先过滤大部分置信度比较低的预测结果（比如 `score_thr` 阈值设置为 0.05，则去除当前预测置信度低于 0.05 的结果），然后得到 bbox 坐标、所在网格的坐标、置信度、标签的信息。经过三个特征层遍历之后，分别整合这三个层得到的的四个信息放入 results 列表中。

(5) 还原到原图尺度

最后将网络的预测结果映射到整图当中，得到 bbox 在整图中的坐标值

(6) NMS

进行 nms 操作，最终预测得到的返回值为经过后处理的每张图片的检测结果，包含分类置信度，框的 labels，框的四个坐标

11.3.3 2 总结

本文对 RTMDet 原理和在 MMYOLO 实现进行了详细解析，希望能帮助用户理解算法实现过程。同时请注意：由于 RTMDet 本身也在不断更新，本开源库也会不断迭代，请及时阅读和同步最新版本。

11.4 YOLOv8 原理和实现全解析

11.4.1 0 简介

以上结构图由 RangeKing@github 绘制。

YOLOv8 是 Ultralytics 公司在 2023 年 1 月 10 号开源的 YOLOv5 的下一个重大更新版本，目前支持图像分类、物体检测和实例分割任务，在还没有开源时就收到了用户的广泛关注。

按照官方描述，YOLOv8 是一个 SOTA 模型，它建立在以前 YOLO 版本的成功基础上，并引入了新的功能和改进，以进一步提升性能和灵活性。具体创新包括一个新的骨干网络、一个新的 Ancher-Free 检测头和一个新的损失函数，可以在从 CPU 到 GPU 的各种硬件平台上运行。不过 Ultralytics 并没有直接将开源库命名为 YOLOv8，而是直接使用 Ultralytics 这个词，原因是 Ultralytics 将这个库定位为算法框架，而非某一个特定算法，一个主要特点是可扩展性。其希望这个库不仅仅能够用于 YOLO 系列模型，而是能够支持非 YOLO 模型以及分类分割姿态估计等各类任务。总而言之，Ultralytics 开源库的两个主要优点是：

- **融合众多当前 SOTA 技术于一体**
- **未来将支持其他 YOLO 系列以及 YOLO 之外的更多算法**

下表为官方在 COCO Val 2017 数据集上测试的 mAP、参数量和 FLOPs 结果。可以看出 YOLOv8 相比 YOLOv5 精度提升非常多，但是 N/S/M 模型相应的参数量和 FLOPs 都增加了不少，从上图也可以看出相比 YOLOV5 大部分模型推理速度变慢了。

额外提一句，现在各个 YOLO 系列改进算法都在 COCO 上面有明显性能提升，但是在自定义数据集上面的泛化性还没有得到广泛验证，至今依然听到不少关于 YOLOv5 泛化性能较优异的说法。**对各系列 YOLO 泛化性验证也是 MMYOLO 中一个特别关心和重点发力的方向**。

阅读本文前，如果你对 YOLOv5、YOLOv6 和 RTMDet 不熟悉，可以先看下如下文档：

1. YOLOv5 原理和实现全解析
2. YOLOv6 原理和实现全解析
3. RTMDet 原理和实现全解析

11.4.2 1 YOLOv8 概述

YOLOv8 算法的核心特性和改动可以归结为如下：

1. **提供了一个全新的 SOTA 模型，包括 P5 640 和 P6 1280 分辨率的目标检测网络和基于 YOLACT 的实例分割模型。和 YOLOv5 一样，基于缩放系数也提供了 N/S/M/L/X 尺度的不同大小模型，用于满足不同场景需求**
2. **骨干网络和 Neck 部分可能参考了 YOLOv7 ELAN 设计思想，将 YOLOv5 的 C3 结构换成了梯度流更丰富的 C2f 结构，并对不同尺度模型调整了不同的通道数，属于对模型结构精心微调，不再是无脑一套参数应用所有模型，大幅提升了模型性能。不过这个 C2f 模块中存在 Split 等操作对特定硬件部署没有之前那么友好了**
3. **Head 部分相比 YOLOv5 改动较大，换成了目前主流的解耦头结构，将分类和检测头分离，同时也从 Anchor-Based 换成了 Anchor-Free**
4. **Loss 计算方面采用了 TaskAlignedAssigner 正样本分配策略，并引入了 Distribution Focal Loss**
5. **训练的数据增强部分引入了 YOLOX 中的最后 10 epoch 关闭 Mosiac 增强的操作，可以有效地提升精度**

从上面可以看出，YOLOv8 主要参考了最近提出的诸如 YOLOX、YOLOv6、YOLOv7 和 PPYOLOE 等算法的相关设计，本身的创新点不多，偏向工程实践，主推的还是 ultralytics 这个框架本身。

下面将按照模型结构设计、Loss 计算、训练数据增强、训练策略和模型推理过程共 5 个部分详细介绍 YOLOv8 目标检测的各种改进，实例分割部分暂时不进行描述。

11.4.3 2 模型结构设计

模型完整图示可以看图 1。

在暂时不考虑 Head 情况下，对比 YOLOv5 和 YOLOv8 的 yaml 配置文件可以发现改动较小。

左侧为 YOLOv5-s，右侧为 YOLOv8-s

骨干网络和 Neck 的具体变化为：

- 第一个卷积层的 kernel 从 6x6 变成了 3x3
- 所有的 C3 模块换成 C2f，结构如下所示，可以发现多了更多的跳层连接和额外的 Split 操作
- 去掉了 Neck 模块中的 2 个卷积连接层
- Backbone 中 C2f 的 block 数从 3-6-9-3 改成了 3-6-6-3
- 查看 N/S/M/L/X 等不同大小模型，可以发现 N/S 和 L/X 两组模型只是改了缩放系数，但是 S/M/L 等骨干网络的通道数设置不一样，没有遵循同一套缩放系数。如此设计的原因应该是同一套缩放系数下的通道设置不是最优设计，YOLOv7 网络设计时也没有遵循一套缩放系数作用于所有模型

Head 部分变化最大，从原先的耦合头变成了解耦头，并且从 YOLOv5 的 Anchor-Based 变成了 Anchor-Free。其结构如下所示：

可以看出，不再有之前的 objectness 分支，只有解耦的分类和回归分支，并且其回归分支使用了 Distribution Focal Loss 中提出的积分形式表示法。

11.4.4 3 Loss 计算

Loss 计算过程包括 2 个部分：正负样本分配策略和 Loss 计算。现代目标检测器大部分都会在正负样本分配策略上面做文章，典型的如 YOLOX 的 simOTA、TOOD 的 TaskAlignedAssigner 和 RTMDet 的 DynamicSoftLabelAssigner，这类 Assigner 大都是动态分配策略，而 YOLOv5 采用的依然是静态分配策略。考虑到动态分配策略的优异性，YOLOv8 算法中则直接引用了 TOOD 的 TaskAlignedAssigner。TaskAlignedAssigner 的匹配策略简单总结为：根据分类与回归的分数加权的分数选择正样本。

$$t = s^{\alpha} + u^{\beta}$$

`s` 是标注类别对应的预测分值，`u` 是预测框和 gt 框的 iou，两者相乘就可以衡量对齐程度。

1. 对于每一个 GT，对所有的预测框基于 GT 类别对应分类分数，预测框与 GT 的 IoU 的加权得到一个关联分类以及回归的对齐分数 `alignment_metrics`
2. 对于每一个 GT，直接基于 `alignment_metrics` 对齐分数选取 topK 大的作为正样本

Loss 计算包括 2 个分支：**分类和回归分支，没有了之前的 objectness 分支**。

- 分类分支依然采用 BCE Loss
- 回归分支需要和 Distribution Focal Loss 中提出的积分形式表示法绑定，因此使用了 Distribution Focal Loss，同时还使用了 CIoU Loss

3 个 Loss 采用一定权重比例加权即可。

11.4.5 4 训练数据增强

数据增强方面和 YOLOv5 差距不大，只不过引入了 YOLOX 中提出的最后 10 个 epoch 关闭 Mosaic 的操作。假设训练 epoch 是 500，其示意图如下所示：

考虑到不同模型应该采用的数据增强强度不一样，因此对于不同大小模型，有部分超参会进行修改，典型的如大模型会开启 MixUp 和 CopyPaste。数据增强后典型效果如下所示：

上述效果可以运行 browse_dataset 脚本得到。由于每个 pipeline 都是比较常规的操作，本文不再赘述。如果想了解每个 pipeline 的细节，可以查看 MMYOLO 中 YOLOv5 的算法解析文档 。

11.4.6 5 训练策略

YOLOv8 的训练策略和 YOLOv5 没有啥区别，最大区别就是**模型的训练总 epoch 数从 300 提升到了 500**，这也导致训练时间急剧增加。以 YOLOv8-S 为例，其训练策略汇总如下：

11.4.7 6 模型推理过程

YOLOv8 的推理过程和 YOLOv5 几乎一样，唯一差别在于前面需要对 Distribution Focal Loss 中的积分表示 bbox 形式进行解码，变成常规的 4 维度 bbox，后续计算过程就和 YOLOv5 一样了。

以 COCO 80 类为例，假设输入图片大小为 640x640，MMYOLO 中实现的推理过程示意图如下所示：

其推理和后处理过程为：

(1) bbox 积分形式转换为 4d bbox 格式

对 Head 输出的 bbox 分支进行转换，利用 Softmax 和 Conv 计算将积分形式转换为 4 维 bbox 格式

(2) 维度变换

YOLOv8 输出特征图尺度为 `80x80`、`40x40` 和 `20x20` 的三个特征图。Head 部分输出分类和回归共 6 个尺度的特征图。将 3 个不同尺度的类别预测分支、bbox 预测分支进行拼接，并进行维度变换。为了后续方便处理，会将原先的通道维度置换到最后，类别预测分支和 bbox 预测分支 shape 分别为 (b, 80x80+40x40+20x20, 80)=(b,8400,80)，(b,8400,4)。

(3) 解码还原到原图尺度

分类预测分支进行 Sigmoid 计算，而 bbox 预测分支需要进行解码，还原为真实的原图解码后 xyxy 格式。

(4) 阈值过滤

遍历 batch 中的每张图，采用 `score_thr` 进行阈值过滤。在这过程中还需要考虑 **multi_label 和 nms_pre，确保过滤后的检测框数目不会多于 nms_pre**。

(5) 还原到原图尺度和 nms

基于前处理过程，将剩下的检测框还原到网络输出前的原图尺度，然后进行 nms 即可。最终输出的检测框不能多于 **max_per_img**。

有一个特别注意的点：**YOLOv5 中采用的 Batch shape 推理策略，在 YOLOv8 推理中暂时没有开启，不清楚后面是否会开启，在 MMYOLO 中快速测试了下，如果开启 Batch shape 会涨大概 0.1~0.2**。

11.4.8 7 特征图可视化

MMYOLO 中提供了一套完善的特征图可视化工具，可以帮助用户可视化特征的分布情况。为了和官方性能对齐，此处依然采用官方权重进行可视化。

以 YOLOv8-s 模型为例，第一步需要下载官方权重，然后将该权重通过 yolov8_to_mmyolo 脚本将去转换到 MMYOLO 中，注意必须要将脚本置于官方仓库下才能正确运行，假设得到的权重名字为 mmyolov8s.pth。

假设想可视化 backbone 输出的 3 个特征图效果，则只需要

```
cd mmyolo
python demo/featmap_vis_demo.py demo/demo.jpg configs/yolov8/yolov8_s_syncbn_fast_
↪8xb16-500e_coco.py mmyolov8s.pth --channel-reductio squeeze_mean
```

需要特别注意，为了确保特征图和图片叠加显示能对齐效果，需要先将原先的 `test_pipeline` 替换为如下：

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # 这里将␣
↪LetterResize 修改成 mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

从上图可以看出**不同输出特征图层主要负责预测不同尺度的物体**。

我们也可以可视化 Neck 层的 3 个输出层特征图：

```
cd mmyolo
python demo/featmap_vis_demo.py demo/demo.jpg configs/yolov8/yolov8_s_syncbn_fast_
↪8xb16-500e_coco.py mmyolov8s.pth --channel-reductio squeeze_mean --target-layers␣
↪neck
```

从上图可以发现物体处的特征更加聚焦。

11.4.9 总结

本文详细分析和总结了最新的 YOLOv8 算法，从整体设计到模型结构、Loss 计算、训练数据增强、训练策略和推理过程进行了详细的说明，并提供了大量的示意图供大家方便理解。简单来说 YOLOv8 是一个包括了图像分类、Anchor-Free 物体检测和实例分割的高效算法，检测部分设计参考了目前大量优异的最新的 YOLO 改进算法，实现了新的 SOTA。不仅如此还推出了一个全新的框架。不过这个框架还处于早期阶段，还需要不断完善。

MMYOLO 开源地址：https://github.com/open-mmlab/mmyolo/blob/dev/configs/yolov8/README.md

MMYOLO 算法解析教程：https://mmyolo.readthedocs.io/zh_CN/latest/algorithm_descriptions/index.html#id2

CHAPTER 12

MMYOLO 应用范例介绍

12.1 基于 MMYOLO 的频高图实时目标检测 benchmark

12.1.1 数据集构建

数字频高图是获取电离层实时信息最重要的途径。电离层结构检测对精准提取电离层关键参数，具有非常重要的研究意义。

利用中国科学院在海南、武汉、怀来获取的不同季节的 4311 张频高图建立数据集，使用 labelme 人工标注出 E 层、Es-c 层、Es-l 层、F1 层、F2 层、Spread F 层共 6 种结构。数据集下载

使用 labelme 标注的图像预览

1. 数据集准备

下载数据后，放置在 MMYOLO 仓库的根目录下，使用 `unzip test.zip` 命令（linux）解压至当前文件夹。解压后的文件夹结构为：

```
Iono4311/
├── images
|       ├── 20130401005200.png
|       └── ...
└── labels
        ├── 20130401005200.json
        └── ...
```

其中，`images` 目录下存放输入图片，`labels` 目录下存放使用 labelme 标注得到的 json 文件。

2. 数据集格式转换

使用 MMYOLO 提供的 `tools/dataset_converters/labelme2coco.py` 脚本将 labelme 格式的标注文件转换为 COCO 格式的标注文件。

```
python tools/dataset_converters/labelme2coco.py --img-dir ./Iono4311/images \
                                                --labels-dir ./Iono4311/labels \
                                                --out ./Iono4311/annotations/
↪annotations_all.json
```

3. 浏览数据集

使用下面的命令可以将 COCO 的 label 在图片上进行显示，这一步可以验证刚刚转换是否有问题。

```
python tools/analysis_tools/browse_coco_json.py --img-dir ./Iono4311/images \
                                                --ann-file ./Iono4311/annotations/
↪annotations_all.json
```

4. 划分训练集、验证集、测试集

设置 70% 的图片为训练集，15% 作为验证集，15% 为测试集。

```
python tools/misc/coco_split.py --json ./Iono4311/annotations/annotations_all.json \
                                --out-dir ./Iono4311/annotations \
                                --ratios 0.7 0.15 0.15 \
                                --shuffle \
                                --seed 14
```

划分后的文件夹结构:

```
Iono4311/
├── annotations
│   ├── annotations_all.json
│   ├── class_with_id.txt
│   ├── test.json
│   ├── train.json
│   └── val.json
├── classes_with_id.txt
├── images
├── labels
├── test_images
├── train_images
└── val_images
```

12.1.2 配置文件

配置文件存放在目录 `/projects/misc/ionogram_detection/` 下。

1. 数据集分析

使用 `tools/analysis_tools/dataset_analysis.py` 从数据集中采样 200 张图片进行可视化分析：

```
python tools/analysis_tools/dataset_analysis.py projects/misc/ionogram_detection/
↪yolov5/yolov5_s-v61_fast_1xb96-100e_ionogram.py \
                                                --out-dir output
```

得到以下输出：

```
The information obtained is as follows:
+------------------------------+
| Information of dataset class |
+---------------+--------------+
| Class name    | Bbox num     |
+---------------+--------------+
| E             | 98           |
| Es-l          | 27           |
| Es-c          | 46           |
| F1            | 100          |
| F2            | 194          |
| Spread-F      | 6            |
+---------------+--------------+
```

说明本数据集存在样本不均衡的现象。

各类别目标大小统计

根据统计结果，E、Es-l、Esc、F1 类别以小目标居多，F2、Spread F 类主要是中等大小目标。

2. 可视化 config 中的数据处理部分

以 YOLOv5-s 为例，根据配置文件中的 `train_pipeline`，训练时采用的数据增强策略包括：

- 马赛克增强
- 随机仿射变换
- Albumentations 数据增强工具包（包括多种数字图像处理方法）
- HSV 随机增强图像
- 随机水平翻转

使用 `tools/analysis_tools/browse_dataset.py` 脚本的 **‘pipeline’** 模式，可以可视化每个 pipeline 的输出效果：

```
python tools/analysis_tools/browse_dataset.py projects/misc/ionogram_detection/yolov5/
↪yolov5_s-v61_fast_1xb96-100e_ionogram.py \
                                              -m pipeline \
                                              --out-dir output
```

pipeline 输出可视化

3. 优化 Anchor 尺寸

使用分析工具中的 `tools/analysis_tools/optimize_anchors.py` 脚本得到适用于本数据集的先验锚框尺寸。

```
python tools/analysis_tools/optimize_anchors.py projects/misc/ionogram_detection/
↪yolov5/yolov5_s-v61_fast_1xb96-100e_ionogram.py \
                                                --algorithm v5-k-means \
                                                --input-shape 640 640 \
                                                --prior-match-thr 4.0 \
                                                --out-dir work_dirs/dataset_analysis_
↪5_s
```

4. 模型复杂度分析

根据配置文件，使用分析工具中的 `tools/analysis_tools/get_flops.py` 脚本可以得到模型的参数量、浮点计算量等信息。以 YOLOv5-s 为例：

```
python tools/analysis_tools/get_flops.py projects/misc/ionogram_detection/yolov5/
↪yolov5_s-v61_fast_1xb96-100e_ionogram.py
```

得到如下输出，表示模型的浮点运算量为 7.947G，一共有 7.036M 个可学习参数。

```
==============================
Input shape: torch.Size([640, 640])
Model Flops: 7.947G
Model Parameters: 7.036M
==============================
```

12.1.3 训练和测试

1. 训练

训练可视化：本范例按照标注 + 训练 + 测试 + 部署全流程中的步骤安装和配置 wandb。

调试技巧：在调试代码的过程中，有时需要训练几个 epoch，例如调试验证过程或者权重的保存是否符合期望。对于继承自 `BaseDataset` 的数据集（如本范例中的 `YOLOv5CocoDataset`），在 `train_dataloader` 中的 `dataset` 字段增加 `indices` 参数，即可指定每个 epoch 迭代的样本数，减少迭代时间。

```
train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        _delete_=True,
        type='RepeatDataset',
        times=1,
        dataset=dict(
            type=_base_.dataset_type,
            indices=200,  # 设置 indices=200，表示每个 epoch 只迭代 200 个样本
            data_root=data_root,
            metainfo=metainfo,
            ann_file=train_ann_file,
            data_prefix=dict(img=train_data_prefix),
            filter_cfg=dict(filter_empty_gt=False, min_size=32),
            pipeline=_base_.train_pipeline)))
```

启动训练：

```
python tools/train.py projects/misc/ionogram_detection/yolov5/yolov5_s-v61_fast_1xb96-
↪100e_ionogram.py
```

2. 测试

指定配置文件和模型的路径以启动测试：

```
python tools/test.py projects/misc/ionogram_detection/yolov5/yolov5_s-v61_fast_1xb96-
↪100e_ionogram.py \
                     work_dirs/yolov5_s-v61_fast_1xb96-100e_ionogram/xxx
```

12.1.4 实验与结果分析

选择合适的 batch size

- Batch size 主导了训练速度。通常，理想的 batch size 是是硬件能支持的最大 batch size。
- 当显存占用没有达到饱和时，如果 batch size 翻倍，训练吞吐量也应该翻倍（或接近翻倍），训练时间应该减半或接近减半。
- 使用**混合精度训练**可以加快训练速度、减小显存。在执行 `train.py` 脚本时添加 `--amp` 参数即可开启。

硬件信息：

- GPU：V100，显存 32G
- CPU：10 核，内存 40G

实验结果：

不同 batch size 的训练过程中，数据加载时间 `data_time` 占每步总时长的比例

分析结果，可以得出以下结论：

- 混合精度训练对模型的精度几乎没有影响，并且可以明显减少显存占用。
- Batch size 增加 3 倍，和训练时长并没有相应地减小 3 倍。根据训练过程中 `data_time` 的记录，batch size 越大，`data_time` 也越大，说明数据加载成为了限制训练速度的瓶颈。增大加载数据的进程数 `num_workers` 可以加快数据加载。

消融实验

为了得到适用于本数据集的训练流水线，以 YOLOv5-s 模型为例，进行以下消融实验。

不同数据增强方法

结果表明，马赛克增强和随机仿射变换可以对验证集表现带来明显的提升。

是否使用预训练权重

在配置文件中，修改 `load_from = None` 即可不使用预训练权重。对不使用预训练权重的实验，将基础学习率增大四倍，训练轮数增加至 200 轮，使模型得到较为充分的训练。

训练过程中的损失下降对比图

损失下降曲线表明，使用预训练权重时，loss 下降得更快。可见即使是自然图像数据集上预训练的模型，在雷达图像数据集上微调时，也可以加快模型收敛。

频高图结构检测 benchmark

CHAPTER 13

轻松更换主干网络

注解:

1. 使用其他主干网络时，你需要保证主干网络的输出通道与 Neck 的输入通道相匹配。
2. 下面给出的配置文件，仅能确保训练可以正确运行，直接训练性能可能不是最优的。因为某些 backbone 需要配套特定的学习率、优化器等超参数。后续会在“训练技巧章节”补充训练调优相关内容。

13.1 使用 MMYOLO 中注册的主干网络

假设想将 YOLOv6EfficientRep 作为 YOLOv5 的主干网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    backbone=dict(
        type='YOLOv6EfficientRep',
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='ReLU', inplace=True))
)
```

13.2 跨库使用主干网络

OpenMMLab 2.0 体系中 MMYOLO、MMDetection、MMClassification、MMSelfsup 中的模型注册表都继承自 MMEngine 中的根注册表，允许这些 OpenMMLab 开源库直接使用彼此已经实现的模块。因此用户可以在 MMYOLO 中使用来自 MMDetection、MMClassification、MMSelfsup 的主干网络，而无需重新实现。

13.2.1 使用在 MMDetection 中实现的主干网络

1. 假设想将 ResNet-50 作为 YOLOv5 的主干网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [512, 1024, 2048]

model = dict(
    backbone=dict(
        _delete_=True, # 将 _base_ 中关于 backbone 的字段删除
        type='mmdet.ResNet', # 使用 mmdet 中的 ResNet
        depth=50,
        num_stages=4,
        out_indices=(1, 2, 3),
        frozen_stages=1,
        norm_cfg=dict(type='BN', requires_grad=True),
        norm_eval=True,
        style='pytorch',
        init_cfg=dict(type='Pretrained', checkpoint='torchvision://resnet50')),
    neck=dict(
        type='YOLOv5PAFPN',
        widen_factor=widen_factor,
        in_channels=channels, # 注意：ResNet-50 输出的 3 个通道是 [512, 1024, 2048]，和原先的 yolov5-s neck 不匹配，需要更改
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # head 部分输入通道也要做相应更改
            widen_factor=widen_factor))
)
```

2. 假设想将 SwinTransformer-Tiny 作为 YOLOv5 的主干网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [192, 384, 768]
checkpoint_file = 'https://github.com/SwinTransformer/storage/releases/download/v1.0.
↪0/swin_tiny_patch4_window7_224.pth'  # noqa

model = dict(
    backbone=dict(
        _delete_=True, # 将 _base_ 中关于 backbone 的字段删除
        type='mmdet.SwinTransformer', # 使用 mmdet 中的 SwinTransformer
        embed_dims=96,
        depths=[2, 2, 6, 2],
        num_heads=[3, 6, 12, 24],
        window_size=7,
        mlp_ratio=4,
        qkv_bias=True,
        qk_scale=None,
        drop_rate=0.,
        attn_drop_rate=0.,
        drop_path_rate=0.2,
        patch_norm=True,
        out_indices=(1, 2, 3),
        with_cp=False,
        convert_weights=True,
        init_cfg=dict(type='Pretrained', checkpoint=checkpoint_file)),
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # 注意：SwinTransformer-Tiny 输出的 3 个通道是 [192, 384,␣
↪768]，和原先的 yolov5-s neck 不匹配，需要更改
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # head 部分输入通道也要做相应更改
            widen_factor=widen_factor))
)
```

13.2.2 使用在 MMClassification 中实现的主干网络

1. 假设想将 ConvNeXt-Tiny 作为 YOLOv5 的主干网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# 请先使用命令： mim install "mmcls>=1.0.0rc2", 安装 mmcls
# 导入 mmcls.models 使得可以调用 mmcls 中注册的模块
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)
checkpoint_file = 'https://download.openmmlab.com/mmclassification/v0/convnext/
↪downstream/convnext-tiny_3rdparty_32xb128-noema_in1k_20220301-795e9634.pth'  # noqa
deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [192, 384, 768]

model = dict(
    backbone=dict(
        _delete_=True, # 将 _base_ 中关于 backbone 的字段删除
        type='mmcls.ConvNeXt', # 使用 mmcls 中的 ConvNeXt
        arch='tiny',
        out_indices=(1, 2, 3),
        drop_path_rate=0.4,
        layer_scale_init_value=1.0,
        gap_before_final_norm=False,
        init_cfg=dict(
            type='Pretrained', checkpoint=checkpoint_file,
            prefix='backbone.')), # MMCls 中主干网络的预训练权重含义 prefix='backbone.', 为
了正常加载权重，需要把这个 prefix 去掉。
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # 注意：ConvNeXt-Tiny 输出的 3 个通道是 [192, 384, 768]，和原
先的 yolov5-s neck 不匹配，需要更改
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # head 部分输入通道也要做相应更改
            widen_factor=widen_factor))
)
```

2. 假设想将 MobileNetV3-small 作为 YOLOv5 的主干网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# 请先使用命令: mim install "mmcls>=1.0.0rc2", 安装 mmcls
# 导入 mmcls.models 使得可以调用 mmcls 中注册的模块
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)
checkpoint_file = 'https://download.openmmlab.com/mmclassification/v0/mobilenet_v3/
↪convert/mobilenet_v3_small-8427ecf0.pth'  # noqa
deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [24, 48, 96]

model = dict(
    backbone=dict(
        _delete_=True, # 将 _base_ 中关于 backbone 的字段删除
        type='mmcls.MobileNetV3', # 使用 mmcls 中的 MobileNetV3
        arch='small',
        out_indices=(3, 8, 11), # 修改 out_indices
        init_cfg=dict(
            type='Pretrained',
            checkpoint=checkpoint_file,
            prefix='backbone.')), # MMCls 中主干网络的预训练权重含义 prefix='backbone.', 为
了正常加载权重, 需要把这个 prefix 去掉。
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # 注意: MobileNetV3-small 输出的 3 个通道是 [24, 48, 96], 和
原先的 yolov5-s neck 不匹配, 需要更改
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # head 部分输入通道也要做相应更改
            widen_factor=widen_factor))
)
```

13.2.3 通过 MMClassification 使用 timm 中实现的主干网络

由于 MMClassification 提供了 PyTorch Image Models (timm) 主干网络的封装，用户也可以通过 MMClassification 直接使用 timm 中的主干网络。假设想将 EfficientNet-B1 作为 YOLOv5 的主干网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# 请先使用命令： mim install "mmcls>=1.0.0rc2", 安装 mmcls
# 以及： pip install timm, 安装 timm
# 导入 mmcls.models 使得可以调用 mmcls 中注册的模块
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [40, 112, 320]

model = dict(
    backbone=dict(
        _delete_=True,  # 将 _base_ 中关于 backbone 的字段删除
        type='mmcls.TIMMBackbone',  # 使用 mmcls 中的 timm 主干网络
        model_name='efficientnet_b1',  # 使用 TIMM 中的 efficientnet_b1
        features_only=True,
        pretrained=True,
        out_indices=(2, 3, 4)),
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels,  # 注意：EfficientNet-B1 输出的 3 个通道是 [40, 112, 320]，和
原先的 yolov5-s neck 不匹配，需要更改
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels,  # head 部分输入通道也要做相应更改
            widen_factor=widen_factor))
)
```

13.2.4 使用在 MMSelfSup 中实现的主干网络

假设想将 MMSelfSup 中 MoCo v3 自监督训练的 ResNet-50 作为 YOLOv5 的主干网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

# 请先使用命令： mim install "mmselfsup>=1.0.0rc3"，安装 mmselfsup
# 导入 mmselfsup.models 使得可以调用 mmselfsup 中注册的模块
custom_imports = dict(imports=['mmselfsup.models'], allow_failed_imports=False)
checkpoint_file = 'https://download.openmmlab.com/mmselfsup/1.x/mocov3/mocov3_
↪resnet50_8xb512-amp-coslr-800e_in1k/mocov3_resnet50_8xb512-amp-coslr-800e_in1k_
↪20220927-e043f51a.pth'  # noqa
deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [512, 1024, 2048]

model = dict(
    backbone=dict(
        _delete_=True, # 将 _base_ 中关于 backbone 的字段删除
        type='mmselfsup.ResNet',
        depth=50,
        num_stages=4,
        out_indices=(2, 3, 4), # 注意：MMSelfSup 中 ResNet 的 out_indices 比 MMdet 和␣
↪MMCls 的要大 1
        frozen_stages=1,
        norm_cfg=dict(type='BN', requires_grad=True),
        norm_eval=True,
        style='pytorch',
        init_cfg=dict(type='Pretrained', checkpoint=checkpoint_file)),
    neck=dict(
        type='YOLOv5PAFPN',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        in_channels=channels, # 注意：ResNet-50 输出的 3 个通道是 [512, 1024, 2048]，和原先
的 yolov5-s neck 不匹配，需要更改
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # head 部分输入通道也要做相应更改
            widen_factor=widen_factor))
)
```

13.2.5 不使用预训练权重

通常情况下，骨干网络初始化都是优先选择预训练权重。如果你不想使用预训练权重，而是想从头开始训练时模型时，我们可以将 backbone 中的 init_cfg 设置为 None，此时骨干网络将会以默认的初始化方法进行初始化，而不会使用训练好的预训练权重进行初始。以下是以 YOLOv5 使用 resnet 作为主干网络为例子，其余算法也是同样的处理：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = 1.0
channels = [512, 1024, 2048]

model = dict(
    backbone=dict(
        _delete_=True, # 将 _base_ 中关于 backbone 的字段删除
        type='mmdet.ResNet', # 使用 mmdet 中的 ResNet
        depth=50,
        num_stages=4,
        out_indices=(1, 2, 3),
        frozen_stages=1,
        norm_cfg=dict(type='BN', requires_grad=True),
        norm_eval=True,
        style='pytorch',
        init_cfg=None # init_cfg 设置为 None，则 backbone 将不会使用预训练好的权重进行初始化了
    ),
    neck=dict(
        type='YOLOv5PAFPN',
        widen_factor=widen_factor,
        in_channels=channels, # 注意：ResNet-50 输出的 3 个通道是 [512, 1024, 2048]，和原先的 yolov5-s neck 不匹配，需要更改
        out_channels=channels),
    bbox_head=dict(
        type='YOLOv5Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            in_channels=channels, # head 部分输入通道也要做相应更改
            widen_factor=widen_factor))
)
```

CHAPTER 14

模型复杂度分析

我们提供了 tools/analysis_tools/get_flops.py 脚本来帮助进行 MMYOLO 系列中所有模型的复杂度分析。目前支持计算并输出给定模型的 parameters, activation 以及 flops；同时支持以网络结构或表格的形式打印输出每一层网络的复杂度信息。

调用命令如下：

```
python tools/analysis_tools/get_flops.py
    ${CONFIG_FILE} \                                # 配置文件路径
    [--shape ${IMAGE_SIZE}] \                       # 输入图像大小 (int)，默认取 640*640
    [--show-arch ${ARCH_DISPLAY}] \                 # 以网络结构形式逐层展示复杂度信息
    [--not-show-table ${TABLE_DISPLAY}] \           # 以表格形式逐层展示复杂度信息
    [--cfg-options ${CFG_OPTIONS}]                  # 配置文件参数修改选项
# [] 代表可选参数，实际输入命令行时，不用输入 []
```

接下来以 RTMDet 中的 rtmdet_s_syncbn_fast_8xb32-300e_coco.py 配置文件为例，详细展示该脚本的几种使用情形：

14.1 样例 1: 打印模型的 Flops 和 Parameters，并以表格形式展示每层网络复杂度

```
python tools/analysis_tools/get_flops.py  configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-
↪300e_coco.py
```

输出如下：

```
==============================
Input shape: torch.Size([640, 640])
Model Flops: 14.835G
Model Parameters: 8.887M
==============================
```

14.2 样例 2：以网络结构形式逐层展示模型复杂度信息

```
python tools/analysis_tools/get_flops.py  configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-
↪300e_coco.py --show-arch
```

由于该网络结构复杂，输出较长。以下仅展示 bbox_head.head_module.rtm_reg 部分的输出：

```
(rtm_reg): ModuleList(
        #params: 1.55K, #flops: 4.3M, #acts: 33.6K
        (0): Conv2d(
          128, 4, kernel_size=(1, 1), stride=(1, 1)
          #params: 0.52K, #flops: 3.28M, #acts: 25.6K
        )
        (1): Conv2d(
          128, 4, kernel_size=(1, 1), stride=(1, 1)
          #params: 0.52K, #flops: 0.82M, #acts: 6.4K
        )
        (2): Conv2d(
          128, 4, kernel_size=(1, 1), stride=(1, 1)
          #params: 0.52K, #flops: 0.2M, #acts: 1.6K
        )
```

CHAPTER 15

标注 + 训练 + 测试 + 部署全流程

在平时的工作学习中，我们经常会遇到一些任务需要训练自定义的私有数据集，开源数据集去作为上线模型的场景比较少，这就需要我们对自己的私有数据集进行一系列的操作，以确保模型能够上线生产服务于客户。

参见:

本文档配套的视频已发布在 B 站，可前去查看: 自定义数据集从标注到部署保姆级教程

注解: 本教程所有指令是在 Linux 上面完成，Windows 也是完全可用的，但是命令和操作稍有不同。

本教程默认您已经完成 MMYOLO 的安装，如果未安装，请参考文档 开始你的第一步 进行安装。

本教程涵盖从用户自定义图片数据集标注到最终进行训练和部署的整体流程。步骤概览如下:

1. 数据集准备: `tools/misc/download_dataset.py`
2. 使用 labelme 和算法进行辅助和优化数据集标注: `demo/image_demo.py` + labelme
3. 使用脚本转换成 COCO 数据集格式: `tools/dataset_converters/labelme2coco.py`
4. 数据集划分为训练集、验证集和测试集: `tools/misc/coco_split.py`
5. 根据数据集内容新建 config 文件
6. 数据集可视化分析: `tools/analysis_tools/dataset_analysis.py`
7. 优化 Anchor 尺寸: `tools/analysis_tools/optimize_anchors.py`
8. 可视化 config 配置中数据处理部分: `tools/analysis_tools/browse_dataset.py`
9. 训练: `tools/train.py`

10. 推理：demo/image_demo.py
11. 部署

注解： 在训练得到模型权重和验证集的 mAP 后，用户需要对预测错误的 bad case 进行深入分析，以便优化模型，MMYOLO 在后续会增加这个功能，敬请期待。

下面详细介绍每一步。

15.1 1. 数据集准备

- 如果您现在暂时没有自己的数据集，亦或者想尝试用一个小型数据集来跑通我们的整体流程，可以使用本教程提供的一个 144 张图片的 cat 数据集（本 cat 数据集由 @RangeKing 提供原始图片，由 @PeterH0323 进行数据清洗）。本教程的剩余部分都将以此 cat 数据集为例进行讲解。

下载也非常简单，只需要一条命令即可完成（数据集压缩包大小 217 MB）：

```
python tools/misc/download_dataset.py --dataset-name cat --save-dir ./data/cat --
↪unzip --delete
```

该命令会自动下载数据集到 ./data/cat 文件夹中，该文件的目录结构是：

```
.
└── ./data/cat
    ├── images # 图片文件
    │   ├── image1.jpg
    │   ├── image2.png
    │   └── ...
    ├── labels # labelme 标注文件
    │   ├── image1.json
    │   ├── image2.json
    │   └── ...
    ├── annotations # 数据集划分的 COCO 文件
    │   ├── annotations_all.json # 全量数据的 COCO label 文件
    │   ├── trainval.json # 划分比例 80% 的数据
    │   └── test.json # 划分比例 20% 的数据
    └── class_with_id.txt # id + class_name 文件
```

这个数据集可以直接训练，如果您想体验整个流程的话，可以将 images 文件夹**以外的**其余文件都删除。

- 如您已经有数据，可以将其组成下面的结构：

```
.
└── $DATA_ROOT
```

(下页继续)

(续上页)

```
    └── images
        ├── image1.jpg
        ├── image2.png
        └── ...
```

15.2 2. 使用 labelme 和算法进行辅助和优化数据集标注

通常，标注有 2 种方法：

- 软件或者算法辅助 + 人工修正 label（推荐，降本提速）
- 仅人工标注

注解： 目前我们也在考虑接入第三方库来支持通过 GUI 界面调用 MMYOLO 推理接口实现算法辅助标注和人工优化标注一体功能。如果您有兴趣或者想法可以在 issue 留言或直接联系我们！

15.2.1 2.1 软件或者算法辅助 + 人工修正 label

辅助标注的原理是用已有模型进行推理，将得出的推理信息保存为标注软件 label 文件格式。然后人工操作标注软件加载生成好的 label 文件，只需要检查每张图片的目标是否标准，以及是否有漏掉、错标的目标。【软件或者算法辅助 + 人工修正 label】这种方式可以节省很多时间和精力，达到**降本提速**的目的。

注解： 如果已有模型（典型的如 COCO 预训练模型）没有您自定义新数据集的类别，建议先人工打 100 张左右的图片 label，训练个初始模型，然后再进行辅助标注。

下面会分别介绍其过程：

2.1.1 软件或者算法辅助

使用 MMYOLO 提供的模型推理脚本 `demo/image_demo.py`，并设置 `--to-labelme` 则可以将推理结果生成 labelme 格式的 label 文件，具体用法如下：

```
python demo/image_demo.py img \
                          config \
                          checkpoint
                          [--out-dir OUT_DIR] \
                          [--device DEVICE] \
                          [--show] \
```

(下页继续)

(续上页)

```
                            [--deploy] \
                            [--score-thr SCORE_THR] \
                            [--class-name CLASS_NAME]
                            [--to-labelme]
```

其中：

- img：图片的路径，支持文件夹、文件、URL；
- config：用到的模型 config 文件路径；
- checkpoint：用到的模型权重文件路径；
- --out-dir：推理结果输出到指定目录下，默认为 ./output，当 --show 参数存在时，不保存检测结果；
- --device：使用的计算资源，包括 CUDA, CPU 等，默认为 cuda:0；
- --show：使用该参数表示在屏幕上显示检测结果，默认为 False；
- --deploy：是否切换成 deploy 模式；
- --score-thr：置信度阈值，默认为 0.3；
- --to-labelme：是否导出 labelme 格式的 label 文件，不可以与 --show 参数同时存在

例子：

这里使用 YOLOv5-s 作为例子来进行辅助标注刚刚下载的 cat 数据集，先下载 YOLOv5-s 的权重:

```
mkdir work_dirs
wget https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_8xb16-
↪300e_coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth -P .
↪/work_dirs
```

由于 COCO 80 类数据集中已经包括了 cat 这一类，因此我们可以直接加载 COCO 预训练权重进行辅助标注。

```
python demo/image_demo.py ./data/cat/images \
                          ./configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.
↪py \
                          ./work_dirs/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_
↪20220918_084700-86e02187.pth \
                          --out-dir ./data/cat/labels \
                          --class-name cat \
                          --to-labelme
```

小技巧:

- 如果您的数据集需要标注多类，可以采用类似 `--class-name class1 class2` 格式输入；
- 如果全部输出，则删掉 `--class-name` 这个 flag 即可全部类都输出。

生成的 label 文件会在 `--out-dir` 中:

```
.
└── $OUT_DIR
    ├── image1.json
    ├── image1.json
    └── ...
```

这是一张原图及其生成的 json 例子:

2.1.2 人工标注

本教程使用的标注软件是 labelme

- 安装 labelme

```
conda create -n labelme python=3.8
conda activate labelme
pip install labelme==5.1.1
```

- 启动 labelme

```
labelme ${图片文件夹路径（即上一步的图片文件夹）} \
        --output ${label 文件所处的文件夹路径（即上一步的 --out-dir）} \
        --autosave \
        --nodata
```

其中:

- `--output`：labelme 标注文件保存路径，如果该路径下已经存在部分图片的标注文件，则会进行加载；
- `--autosave`：标注文件自动保存，会略去一些繁琐的保存步骤；
- `--nodata`：每张图片的标注文件中不保存图片的 base64 编码，设置了这个 flag 会大大减少标注文件的大小。

例子:

```
cd /path/to/mmyolo
labelme ./data/cat/images --output ./data/cat/labels --autosave --nodata
```

输入命令之后 labelme 就会启动，然后进行 label 检查即可。如果 labelme 启动失败，命令行输入 `export QT_DEBUG_PLUGINS=1` 查看具体缺少什么库，安装一下即可。

警告： 标注的时候务必使用 rectangle，快捷键 Ctrl + R（如下图）

15.2.2 2.2 仅人工标注

步骤和【2.1.2 人工标注】相同，只是这里是直接标注，没有预先生成的 label 。

15.3 3. 使用脚本转换成 COCO 数据集格式

15.3.1 3.1 使用脚本转换

MMYOLO 提供脚本将 labelme 的 label 转换为 COCO label

```
python tools/dataset_converters/labelme2coco.py --img-dir ${图片文件夹路径} \
                                                --labels-dir ${label 文件夹位置} \
                                                --out ${输出 COCO label json 路径} \
                                                [--class-id-txt ${class_with_id.txt 路
径}]
```

其中：--class-id-txt：是数据集 id class_name 的 .txt 文件：

- 如果不指定，则脚本会自动生成，生成在 --out 同级的目录中，保存文件名为 class_with_id.txt；
- 如果指定，脚本仅会进行读取但不会新增或者覆盖，同时，脚本里面还会判断是否存在 .txt 中其他的类，如果出现了会报错提示，届时，请用户检查 .txt 文件并加入新的类及其 id。

.txt 文件的例子如下（id 可以和 COCO 一样，从 1 开始）：

```
1 cat
2 dog
3 bicycle
4 motorcycle
```

例子：

以本教程的 cat 数据集为例：

```
python tools/dataset_converters/labelme2coco.py --img-dir ./data/cat/images \
                                                --labels-dir ./data/cat/labels \
                                                --out ./data/cat/annotations/
↪annotations_all.json
```

本次演示的 cat 数据集（注意不需要包括背景类），可以看到生成的 class_with_id.txt 中只有 1 类：

```
1 cat
```

15.3.2 3.2 检查转换的 COCO label

使用下面的命令可以将 COCO 的 label 在图片上进行显示，这一步可以验证刚刚转换是否有问题：

```
python tools/analysis_tools/browse_coco_json.py --img-dir ${图片文件夹路径} \
                                                --ann-file ${COCO label json 路径}
```

例子：

```
python tools/analysis_tools/browse_coco_json.py --img-dir ./data/cat/images \
                                                --ann-file ./data/cat/annotations/
↪annotations_all.json
```

参见：

关于 tools/analysis_tools/browse_coco_json.py 的更多用法请参考 可视化 COCO label。

15.4 4. 数据集划分为训练集、验证集和测试集

通常，自定义图片都是一个大文件夹，里面全部都是图片，需要我们自己去对图片进行训练集、验证集、测试集的划分，如果数据量比较少，可以不划分验证集。下面是划分脚本的具体用法：

```
python tools/misc/coco_split.py --json ${COCO label json 路径} \
                                --out-dir ${划分 label json 保存根路径} \
                                --ratios ${划分比例} \
                                [--shuffle] \
                                [--seed ${划分的随机种子}]
```

其中：

- --ratios：划分的比例，如果只设置了 2 个，则划分为 trainval + test，如果设置为 3 个，则划分为 train + val + test。支持两种格式——整数、小数：
  - 整数：按比例进行划分，代码中会进行归一化之后划分数据集。例子：--ratio 2 1 1（代码里面会转换成 0.5 0.25 0.25）or --ratio 3 1（代码里面会转换成 0.75 0.25）
  - 小数：划分为比例。**如果加起来不为 1，则脚本会进行自动归一化修正**。例子：--ratio 0.8 0.1 0.1 or --ratio 0.8 0.2
- --shuffle: 是否打乱数据集再进行划分；
- --seed：设定划分的随机种子，不设置的话自动生成随机种子。

例子：

```
python tools/misc/coco_split.py --json ./data/cat/annotations/annotations_all.json \
                                --out-dir ./data/cat/annotations \
                                --ratios 0.8 0.2 \
                                --shuffle \
                                --seed 10
```

15.5 5. 根据数据集内容新建 config 文件

确保数据集目录是这样的：

```
.
└── $DATA_ROOT
    ├── annotations
    │   ├── trainval.json # 根据上面的指令只划分 trainval + test，如果您使用 3 组划分比例的话，
这里是 train.json、val.json、test.json
    │   └── test.json
    ├── images
    │   ├── image1.jpg
    │   ├── image1.png
    │   └── ...
    └── ...
```

因为是我们自定义的数据集，所以我们需要自己新建一个 config 并加入需要修改的部分信息。

关于新的 config 的命名：

- 这个 config 继承的是 `yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py`；
- 训练的类以本教程提供的数据集中的类 `cat` 为例（如果是自己的数据集，可以自定义类型的总称）；
- 本教程测试的显卡型号是 1 x 3080Ti 12G 显存，电脑内存 32G，可以训练 YOLOv5-s 最大批次是 `batch size = 32`（详细机器资料可见附录）；
- 训练轮次是 `100 epoch`。

综上所述：可以将其命名为 `yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py`，并将其放置在文件夹 `configs/custom_dataset` 中。

我们可以在 configs 目录下新建一个新的目录 `custom_dataset`，同时在里面新建该 config 文件，并添加以下内容：

```
_base_ = '../yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py'

max_epochs = 100  # 训练的最大 epoch
data_root = './data/cat/'  # 数据集目录的绝对路径
# data_root = '/root/workspace/mmyolo/data/cat/'  # Docker 容器里面数据集目录的绝对路径
```

(下页继续)

(续上页)

```
# 结果保存的路径，可以省略，省略保存的文件名位于 work_dirs 下 config 同名的文件夹中
# 如果某个 config 只是修改了部分参数，修改这个变量就可以将新的训练文件保存到其他地方
work_dir = './work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat'

# load_from 可以指定本地路径或者 URL，设置了 URL 会自动进行下载，因为上面已经下载过，我们这里设置本地路径
# 因为本教程是在 cat 数据集上微调，故这里需要使用 `load_from` 来加载 MMYOLO 中的预训练模型，这样可以在加快收敛速度的同时保证精度
load_from = './work_dirs/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth'  # noqa

# 根据自己的 GPU 情况，修改 batch size，YOLOv5-s 默认为 8 卡 x 16bs
train_batch_size_per_gpu = 32
train_num_workers = 4  # 推荐使用 train_num_workers = nGPU x 4

save_epoch_intervals = 2  # 每 interval 轮迭代进行一次保存一次权重

# 根据自己的 GPU 情况，修改 base_lr，修改的比例是 base_lr_default * (your_bs / default_bs)
base_lr = _base_.base_lr / 4

anchors = [  # 此处已经根据数据集特点更新了 anchor，关于 anchor 的生成，后面小节会讲解
    [(68, 69), (154, 91), (143, 162)],  # P3/8
    [(242, 160), (189, 287), (391, 207)],  # P4/16
    [(353, 337), (539, 341), (443, 432)]  # P5/32
]

class_name = ('cat', )  # 根据 class_with_id.txt 类别信息，设置 class_name
num_classes = len(class_name)
metainfo = dict(
    classes=class_name,
    palette=[(220, 20, 60)]  # 画图时候的颜色，随便设置即可
)

train_cfg = dict(
    max_epochs=max_epochs,
    val_begin=20,  # 第几个 epoch 后验证，这里设置 20 是因为前 20 个 epoch 精度不高，测试意义不大，故跳过
    val_interval=save_epoch_intervals  # 每 val_interval 轮迭代进行一次测试评估
)

model = dict(
    bbox_head=dict(
```

(下页继续)

(续上页)

```
        head_module=dict(num_classes=num_classes),
        prior_generator=dict(base_sizes=anchors),

        # loss_cls 会根据 num_classes 动态调整，但是 num_classes = 1 的时候，loss_cls 恒为 0
        loss_cls=dict(loss_weight=0.5 *
                      (num_classes / 80 * 3 / _base_.num_det_layers))))

train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        _delete_=True,
        type='RepeatDataset',
        # 数据量太少的话，可以使用 RepeatDataset ，在每个 epoch 内重复当前数据集 n 次，这里设置
↪5 是重复 5 次
        times=5,
        dataset=dict(
            type=_base_.dataset_type,
            data_root=data_root,
            metainfo=metainfo,
            ann_file='annotations/trainval.json',
            data_prefix=dict(img='images/'),
            filter_cfg=dict(filter_empty_gt=False, min_size=32),
            pipeline=_base_.train_pipeline)))

val_dataloader = dict(
    dataset=dict(
        metainfo=metainfo,
        data_root=data_root,
        ann_file='annotations/trainval.json',
        data_prefix=dict(img='images/')))

test_dataloader = val_dataloader

val_evaluator = dict(ann_file=data_root + 'annotations/trainval.json')
test_evaluator = val_evaluator

optim_wrapper = dict(optimizer=dict(lr=base_lr))

default_hooks = dict(
    # 设置间隔多少个 epoch 保存模型，以及保存模型最多几个，`save_best` 是另外保存最佳模型（推荐）
    checkpoint=dict(
        type='CheckpointHook',
```

(下页继续)

(续上页)

```
        interval=save_epoch_intervals,
        max_keep_ckpts=5,
        save_best='auto'),
    param_scheduler=dict(max_epochs=max_epochs),
    # logger 输出的间隔
    logger=dict(type='LoggerHook', interval=10))
```

注解： 我们在 projects/misc/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py 放了一份相同的 config 文件，用户可以选择复制到 configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py 路径直接开始训练。

15.6 6. 数据集可视化分析

脚本 tools/analysis_tools/dataset_analysis.py 能够帮助用户得到数据集的分析图。该脚本可以生成 4 种分析图：

- 显示类别和 bbox 实例个数的分布图：show_bbox_num
- 显示类别和 bbox 实例宽、高的分布图：show_bbox_wh
- 显示类别和 bbox 实例宽/高比例的分布图：show_bbox_wh_ratio
- 基于面积规则下，显示类别和 bbox 实例面积的分布图：show_bbox_area

脚本使用方式如下：

```
python tools/analysis_tools/dataset_analysis.py ${CONFIG} \
                                                [--val-dataset ${TYPE}] \
                                                [--class-name ${CLASS_NAME}] \
                                                [--area-rule ${AREA_RULE}] \
                                                [--func ${FUNC}] \
                                                [--out-dir ${OUT_DIR}]
```

例子：

以本教程 cat 数据集的 config 为例：

查看训练集数据分布情况：

```
python tools/analysis_tools/dataset_analysis.py configs/custom_dataset/yolov5_s-v61_
↪syncbn_fast_1xb32-100e_cat.py \
                                                --out-dir work_dirs/dataset_analysis_
↪cat/train_dataset
```

查看验证集数据分布情况：

```
python tools/analysis_tools/dataset_analysis.py configs/custom_dataset/yolov5_s-v61_
↪syncbn_fast_1xb32-100e_cat.py \
                                                --out-dir work_dirs/dataset_analysis_
↪cat/val_dataset \
                                                --val-dataset
```

效果（点击图片可查看大图）：

注解： 因为本教程使用的 cat 数据集数量比较少，故 config 里面用了 RepeatDataset，显示的数目实际上都是重复了 5 次。如果您想得到无重复的分析结果，可以暂时将 RepeatDataset 下面的 `times` 参数从 `5` 改成 `1`。

经过输出的图片分析可以得出，本教程使用的 `cat` 数据集的训练集具有以下情况：

- 图片全部是 `large object`；
- 类别 cat 的数量是 `655`；
- bbox 的宽高比例大部分集中在 `1.0 ~ 1.11`，比例最小值是 `0.36`，最大值是 `2.9`；
- bbox 的宽大部分是 `500 ~ 600` 左右，高大部分是 `500 ~ 600` 左右。

参见：

关于 `tools/analysis_tools/dataset_analysis.py` 的更多用法请参考 可视化数据集分析。

15.7 7. 优化 Anchor 尺寸

> **警告：** 该步骤仅适用于 anchor-base 的模型，例如 YOLOv5；
>
> Anchor-free 的模型可以跳过此步骤，例如 YOLOv6、YOLOX。

脚本 `tools/analysis_tools/optimize_anchors.py` 支持 YOLO 系列中三种锚框生成方式，分别是 `k-means`、`Differential Evolution`、`v5-k-means`.

本示例使用的是 YOLOv5 进行训练，使用的是 `640 x 640` 的输入大小，使用 `v5-k-means` 进行描框的优化：

```
python tools/analysis_tools/optimize_anchors.py configs/custom_dataset/yolov5_s-v61_
↪syncbn_fast_1xb32-100e_cat.py \
                                                --algorithm v5-k-means \
                                                --input-shape 640 640 \
```

(下页继续)

(续上页)

```
                                                --prior-match-thr 4.0 \
                                                --out-dir work_dirs/dataset_analysis_
↪cat
```

注解： 因为该命令使用的是 k-means 聚类算法，存在一定的随机性，这与初始化有关。故每次执行得到的 Anchor 都会有些不一样，但是都是基于传递进去的数据集来进行生成的，故不会有什么不良影响。

经过计算的 Anchor 如下：

修改 config 文件里面的 `anchors` 变量：

```
anchors = [
    [(68, 69), (154, 91), (143, 162)],  # P3/8
    [(242, 160), (189, 287), (391, 207)],  # P4/16
    [(353, 337), (539, 341), (443, 432)]  # P5/32
]
```

参见：

关于 `tools/analysis_tools/optimize_anchors.py` 的更多用法请参考 优化锚框尺寸。

15.8 8. 可视化 config 配置中数据处理部分

脚本 `tools/analysis_tools/browse_dataset.py` 能够帮助用户去直接窗口可视化 config 配置中数据处理部分，同时可以选择保存可视化图片到指定文件夹内。

下面演示使用我们刚刚新建的 config 文件 `configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` 来可视化图片，该命令会使得图片直接弹出显示，每张图片持续 3 秒，图片不进行保存：

```
python tools/analysis_tools/browse_dataset.py configs/custom_dataset/yolov5_s-v61_
↪syncbn_fast_1xb32-100e_cat.py \
                                            --show-interval 3
```

参见：

关于 `tools/analysis_tools/browse_dataset.py` 的更多用法请参考 可视化数据集。

15.9 9. 训练

下面会从以下 3 点来进行讲解：

1. 训练可视化
2. YOLOv5 模型训练
3. 切换 YOLO 模型训练

15.9.1 9.1 训练可视化

如果需要采用浏览器对训练过程可视化，MMYOLO 目前提供 2 种方式 wandb 和 TensorBoard，根据自己的情况选择其一即可 (后续会扩展更多可视化后端支持)。

9.1.1 wandb

wandb 可视化需要在官网注册，并在 https://wandb.ai/settings 获取到 wandb 的 API Keys。

然后在命令行进行安装

```
pip install wandb
# 运行了 wandb login 后输入上文中获取到的 API Keys ，便登录成功。
wandb login
```

在我们刚刚新建的 config 文件 `configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` 的最后添加 wandb 配置：

```
visualizer = dict(vis_backends=[dict(type='LocalVisBackend'), dict(type=
↪'WandbVisBackend')])
```

9.1.2 TensorBoard

安装 Tensorboard 环境

```
pip install tensorboard
```

在我们刚刚新建的 config 文件 `configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` 中添加 `tensorboard` 配置

```
visualizer = dict(vis_backends=[dict(type='LocalVisBackend'),dict(type=
↪'TensorboardVisBackend')])
```

运行训练命令后，Tensorboard 文件会生成在可视化文件夹 work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat/${TIMESTAMP}/vis_data 下，运行下面的命令便可以在网页链接使用 Tensorboard 查看 loss、学习率和 coco/bbox_mAP 等可视化数据了：

```
tensorboard --logdir=work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat
```

15.9.2 9.2 执行训练

使用下面命令进行启动训练（训练大约需要 2.5 个小时）：

```
python tools/train.py configs/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.
↪py
```

如果您开启了 wandb 的话，可以登录到自己的账户，在 wandb 中查看本次训练的详细信息了：

下面是 1 x 3080Ti、batch size = 32，训练 100 epoch 最佳精度权重 work_dirs/yolov5_s-v61_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_epoch_98.pth 得出来的精度（详细机器资料可见附录）：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.968
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.968
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.886
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.977
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.977
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.977

bbox_mAP_copypaste: 0.968 1.000 1.000 -1.000 -1.000 0.968
Epoch(val) [98][116/116]  coco/bbox_mAP: 0.9680  coco/bbox_mAP_50: 1.0000  coco/bbox_
↪mAP_75: 1.0000  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_mAP_
↪l: 0.9680
```

小技巧： 在一般的 finetune 最佳实践中都会推荐将 backbone 固定不参与训练，并且学习率 lr 也进行相应缩放，但是在本教程中发现这种做法会出现一定程度掉点。猜测可能原因是 cat 类别已经在 COCO 数据集中，而本教程使用的 cat 数据集数量比较小导致的。

下表是采用 MMYOLO YOLOv5 预训练模型 yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86
pth 在没对 cat 数据集进行 finetune 的测试精度，可以看到 cat 类别的 mAP 只有 0.866，经过我们 finetune

mAP 提升到了 0.968，提升了 10.2 %，可以证明训练是非常成功的：

```
+---------------+-------+--------------+-----+----------------+------+
| category      | AP    | category     | AP  | category       | AP   |
+---------------+-------+--------------+-----+----------------+------+
| person        | nan   | bicycle      | nan | car            | nan  |
| motorcycle    | nan   | airplane     | nan | bus            | nan  |
| train         | nan   | truck        | nan | boat           | nan  |
| traffic light | nan   | fire hydrant | nan | stop sign      | nan  |
| parking meter | nan   | bench        | nan | bird           | nan  |
| cat           | 0.866 | dog          | nan | horse          | nan  |
| sheep         | nan   | cow          | nan | elephant       | nan  |
| bear          | nan   | zebra        | nan | giraffe        | nan  |
| backpack      | nan   | umbrella     | nan | handbag        | nan  |
| tie           | nan   | suitcase     | nan | frisbee        | nan  |
| skis          | nan   | snowboard    | nan | sports ball    | nan  |
| kite          | nan   | baseball bat | nan | baseball glove | nan  |
| skateboard    | nan   | surfboard    | nan | tennis racket  | nan  |
| bottle        | nan   | wine glass   | nan | cup            | nan  |
| fork          | nan   | knife        | nan | spoon          | nan  |
| bowl          | nan   | banana       | nan | apple          | nan  |
| sandwich      | nan   | orange       | nan | broccoli       | nan  |
| carrot        | nan   | hot dog      | nan | pizza          | nan  |
| donut         | nan   | cake         | nan | chair          | nan  |
| couch         | nan   | potted plant | nan | bed            | nan  |
| dining table  | nan   | toilet       | nan | tv             | nan  |
| laptop        | nan   | mouse        | nan | remote         | nan  |
| keyboard      | nan   | cell phone   | nan | microwave      | nan  |
| oven          | nan   | toaster      | nan | sink           | nan  |
| refrigerator  | nan   | book         | nan | clock          | nan  |
| vase          | nan   | scissors     | nan | teddy bear     | nan  |
| hair drier    | nan   | toothbrush   | nan | None           | None |
+---------------+-------+--------------+-----+----------------+------+
```

参见：

关于如何得到预训练权重的精度，可以详见附录【2. 如何测试数据集在预训练权重的精度】

15.9.3 9.3 尝试 MMYOLO 其他模型

MMYOLO 集成了多种 YOLO 算法，切换非常方便，无需重新熟悉一个新的 repo，直接切换 config 文件就可以轻松切换 YOLO 模型，只需简单 3 步即可切换模型：

1. 新建 config 文件
2. 下载预训练权重
3. 启动训练

下面以 YOLOv6-s 为例，进行讲解。

1. 搭建一个新的 config：

```
_base_ = '../yolov6/yolov6_s_syncbn_fast_8xb32-400e_coco.py'

max_epochs = 100  # 训练的最大 epoch
data_root = './data/cat/'  # 数据集目录的绝对路径

# 结果保存的路径，可以省略，省略保存的文件名位于 work_dirs 下 config 同名的文件夹中
# 如果某个 config 只是修改了部分参数，修改这个变量就可以将新的训练文件保存到其他地方
work_dir = './work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat'

# load_from 可以指定本地路径或者 URL，设置了 URL 会自动进行下载，因为上面已经下载过，我们这里设置本地路径
# 因为本教程是在 cat 数据集上微调，故这里需要使用 `load_from` 来加载 MMYOLO 中的预训练模型，这样可以在加快收敛速度的同时保证精度
load_from = './work_dirs/yolov6_s_syncbn_fast_8xb32-400e_coco_20221102_203035-
↪932e1d91.pth'  # noqa

# 根据自己的 GPU 情况，修改 batch size，YOLOv6-s 默认为 8 卡 x 32bs
train_batch_size_per_gpu = 32
train_num_workers = 4  # 推荐使用 train_num_workers = nGPU x 4

save_epoch_intervals = 2  # 每 interval 轮迭代进行一次保存一次权重

# 根据自己的 GPU 情况，修改 base_lr，修改的比例是 base_lr_default * (your_bs / default_bs)
base_lr = _base_.base_lr / 8

class_name = ('cat', )  # 根据 class_with_id.txt 类别信息，设置 class_name
num_classes = len(class_name)
metainfo = dict(
    classes=class_name,
    palette=[(220, 20, 60)]  # 画图时候的颜色，随便设置即可
)
```

(下页继续)

(续上页)

```
train_cfg = dict(
    max_epochs=max_epochs,
    val_begin=20,  # 第几个 epoch 后验证，这里设置 20 是因为前 20 个 epoch 精度不高，测试意义不
大，故跳过
    val_interval=save_epoch_intervals,  # 每 val_interval 轮迭代进行一次测试评估
    dynamic_intervals=[(max_epochs - _base_.num_last_epochs, 1)]
)

model = dict(
    bbox_head=dict(
        head_module=dict(num_classes=num_classes)),
    train_cfg=dict(
        initial_assigner=dict(num_classes=num_classes),
        assigner=dict(num_classes=num_classes))
)

train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    dataset=dict(
        _delete_=True,
        type='RepeatDataset',
        # 数据量太少的话，可以使用 RepeatDataset ，在每个 epoch 内重复当前数据集 n 次，这里设置
↪5 是重复 5 次
        times=5,
        dataset=dict(
            type=_base_.dataset_type,
            data_root=data_root,
            metainfo=metainfo,
            ann_file='annotations/trainval.json',
            data_prefix=dict(img='images/'),
            filter_cfg=dict(filter_empty_gt=False, min_size=32),
            pipeline=_base_.train_pipeline)))

val_dataloader = dict(
    dataset=dict(
        metainfo=metainfo,
        data_root=data_root,
        ann_file='annotations/trainval.json',
        data_prefix=dict(img='images/')))

test_dataloader = val_dataloader
```

(下页继续)

(续上页)

```
val_evaluator = dict(ann_file=data_root + 'annotations/trainval.json')
test_evaluator = val_evaluator

optim_wrapper = dict(optimizer=dict(lr=base_lr))

default_hooks = dict(
    # 设置间隔多少个 epoch 保存模型，以及保存模型最多几个，`save_best` 是另外保存最佳模型（推荐）
    checkpoint=dict(
        type='CheckpointHook',
        interval=save_epoch_intervals,
        max_keep_ckpts=5,
        save_best='auto'),
    param_scheduler=dict(max_epochs=max_epochs),
    # logger 输出的间隔
    logger=dict(type='LoggerHook', interval=10))

custom_hooks = [
    dict(
        type='EMAHook',
        ema_type='ExpMomentumEMA',
        momentum=0.0001,
        update_buffers=True,
        strict_load=False,
        priority=49),
    dict(
        type='mmdet.PipelineSwitchHook',
        switch_epoch=max_epochs - _base_.num_last_epochs,
        switch_pipeline=_base_.train_pipeline_stage2)
]
```

注解： 同样，我们在 `projects/misc/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py` 放了一份相同的 config 文件，用户可以选择复制到 `configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py` 路径直接开始训练。

虽然新的 config 看上去好像很多东西，其实很多都是重复的，用户可以用对比软件对比一下即可看出大部分的配置都是和 `yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` 相同的。因为这 2 个 config 文件需要继承不同的 config，所以还是要添加一些必要的配置。

2. 下载 YOLOv6-s 的预训练权重

```
wget https://download.openmmlab.com/mmyolo/v0/yolov6/yolov6_s_syncbn_fast_8xb32-400e_
↪coco/yolov6_s_syncbn_fast_8xb32-400e_coco_20221102_203035-932e1d91.pth -P work_dirs/
```

3. 训练

```
python tools/train.py configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py
```

在我的实验中，最佳模型是 work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_epoch_96.pth，其精度如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.987
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.987
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.895
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.989
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.989
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.989

bbox_mAP_copypaste: 0.987 1.000 1.000 -1.000 -1.000 0.987
Epoch(val) [96][116/116]  coco/bbox_mAP: 0.9870  coco/bbox_mAP_50: 1.0000  coco/bbox_
↪mAP_75: 1.0000  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_mAP_
↪l: 0.9870
```

以上演示的是如何在 MMYOLO 中切换模型，可以快速对不同模型进行精度对比，精度高的模型可以上线生产。在我的实验中，YOLOv6 最佳精度 0.9870 比 YOLOv5 最佳精度 0.9680 高出 1.9 %，故后续我们使用 YOLOv6 来进行讲解。

15.10 10. 推理

使用最佳的模型进行推理，下面命令中的最佳模型路径是 ./work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_epoch_96.pth，请用户自行修改为自己训练的最佳模型路径。

```
python demo/image_demo.py ./data/cat/images \
                          ./configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_
↪cat.py \
                          ./work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/
↪bbox_mAP_epoch_96.pth \
```

(下页继续)

(续上页)

```
--out-dir ./data/cat/pred_images
```

小技巧： 如果推理结果不理想，这里举例 2 种情况：

1. 模型欠拟合：需要先判断是不是训练 epoch 不够导致的欠拟合，如果是训练不够，则修改 config 文件里面的 `max_epochs` 和 `work_dir` 参数，或者根据上面的命名方式新建一个 config 文件，重新进行训练。
2. 数据集需优化：如果 epoch 加上去了还是不行，可以增加数据集数量，同时可以重新检查并优化数据集的标注，然后重新进行训练。

15.11 11. 部署

MMYOLO 提供两种部署方式：

1. MMDeploy 框架进行部署
2. 使用 `projects/easydeploy` 进行部署

15.11.1 11.1 MMDeploy 框架进行部署

考虑到部署的机器环境千差万别，很多时候在本地机器可以，但是在生产环境则不一定，这里推荐使用 Docker，做到环境一次部署，终身使用，节省运维搭建环境和部署生产的时间。

本小节会从一下几个小点进行展开讲解：

1. 构建 Docker 镜像
2. 创建 Docker 容器
3. 转换 TensorRT 模型
4. 部署模型执行推理

参见：

如果是对 Docker 不熟悉的用户，可以参考 MMDeploy 的 源码手动安装 文档直接在本地编译。安装完之后，可以直接跳到【11.1.3 转换 TensorRT 模型】小节。

11.1.1 构建 Docker 镜像

```
git clone -b dev-1.x https://github.com/open-mmlab/mmdeploy.git
cd mmdeploy
docker build docker/GPU/ -t mmdeploy:gpu --build-arg USE_SRC_INSIDE=true
```

其中 USE_SRC_INSIDE=true 是拉取基础进行之后在内部切换国内源，构建速度会快一些。

执行脚本后，会进行构建，此刻需要等一段时间：

11.1.2 创建 Docker 容器

```
export MMYOLO_PATH=/path/to/local/mmyolo # 先将您机器上 MMYOLO 的路径写入环境变量
docker run --gpus all --name mmyolo-deploy -v ${MMYOLO_PATH}:/root/workspace/mmyolo -
↪it mmdeploy:gpu /bin/bash
```

可以看到本地的 MMYOLO 环境已经挂载到容器里面了

参见:

有关这部分的详细介绍可以看 MMDeploy 官方文档 使用 Docker 镜像

11.1.3 转换 TensorRT 模型

首先需要在 Docker 容器里面安装 MMYOLO 和 pycuda：

```
export MMYOLO_PATH=/root/workspace/mmyolo # 镜像中的路径，这里不需要修改
cd ${MMYOLO_PATH}
export MMYOLO_VERSION=$(python -c "import mmyolo.version as v; print(v.__version__)")␣
↪ # 查看训练使用的 MMYOLO 版本号
echo "Using MMYOLO ${MMYOLO_VERSION}"
mim install --no-cache-dir mmyolo==${MMYOLO_VERSION}
pip install --no-cache-dir pycuda==2022.2
```

进行模型转换

```
cd /root/workspace/mmdeploy
python ./tools/deploy.py \
    ${MMYOLO_PATH}/configs/deploy/detection_tensorrt-fp16_dynamic-192x192-960x960.py \
    ${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py \
    ${MMYOLO_PATH}/work_dirs/yolov6_s_syncbn_fast_1xb32-100e_cat/best_coco/bbox_mAP_
↪epoch_96.pth \
    ${MMYOLO_PATH}/data/cat/images/mmexport1633684751291.jpg \
    --test-img ${MMYOLO_PATH}/data/cat/images/mmexport1633684751291.jpg \
    --work-dir ./work_dir/yolov6_s_syncbn_fast_1xb32-100e_cat_deploy_dynamic_fp16 \
```

(下页继续)

(续上页)

```
    --device cuda:0 \
    --log-level INFO \
    --show \
    --dump-info
```

等待一段时间，出现了 `All process success.` 即为成功：

查看导出的路径，可以看到如下图所示的文件结构：

```
$WORK_DIR
  ├── deploy.json
  ├── detail.json
  ├── end2end.engine
  ├── end2end.onnx
  └── pipeline.json
```

参见：

关于转换模型的详细介绍，请参考 如何转换模型

11.1.4 部署模型执行推理

需要将 `${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py` 里面的 `data_root` 修改为 Docker 容器里面的路径：

```
data_root = '/root/workspace/mmyolo/data/cat/'  # Docker 容器里面数据集目录的绝对路径
```

执行速度和精度测试：

```
python tools/test.py \
    ${MMYOLO_PATH}/configs/deploy/detection_tensorrt-fp16_dynamic-192x192-960x960.py \
    ${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py \
    --model ./work_dir/yolov6_s_syncbn_fast_1xb32-100e_cat_deploy_dynamic_fp16/
↪end2end.engine \
    --speed-test \
    --device cuda
```

速度测试如下，可见平均推理速度是 `24.10 ms`，对比 PyTorch 推理有速度提升，同时显存也下降了很多：

```
Epoch(test) [ 10/116]    eta: 0:00:20  time: 0.1919  data_time: 0.1330  memory: 12
Epoch(test) [ 20/116]    eta: 0:00:15  time: 0.1220  data_time: 0.0939  memory: 12
Epoch(test) [ 30/116]    eta: 0:00:12  time: 0.1168  data_time: 0.0850  memory: 12
Epoch(test) [ 40/116]    eta: 0:00:10  time: 0.1241  data_time: 0.0940  memory: 12
Epoch(test) [ 50/116]    eta: 0:00:08  time: 0.0974  data_time: 0.0696  memory: 12
```

(下页继续)

(续上页)

```
Epoch(test) [ 60/116]    eta: 0:00:06  time: 0.0865  data_time: 0.0547  memory: 16
Epoch(test) [ 70/116]    eta: 0:00:05  time: 0.1521  data_time: 0.1226  memory: 16
Epoch(test) [ 80/116]    eta: 0:00:04  time: 0.1364  data_time: 0.1056  memory: 12
Epoch(test) [ 90/116]    eta: 0:00:03  time: 0.0923  data_time: 0.0627  memory: 12
Epoch(test) [100/116]    eta: 0:00:01  time: 0.0844  data_time: 0.0583  memory: 12
[tensorrt]-110 times per count: 24.10 ms, 41.50 FPS
Epoch(test) [110/116]    eta: 0:00:00  time: 0.1085  data_time: 0.0832  memory: 12
```

精度测试如下。此配置采用 FP16 格式推理，会有一定程度掉点，但是推理速度更快、显存占比更小：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.954
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.975
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.954
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.860
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.965
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.965
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.965

INFO - bbox_mAP_copypaste: 0.954 1.000 0.975 -1.000 -1.000 0.954
INFO - Epoch(test) [116/116]  coco/bbox_mAP: 0.9540  coco/bbox_mAP_50: 1.0000  coco/
↪bbox_mAP_75: 0.9750  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_
↪mAP_l: 0.9540
```

部署模型图片推理演示：

注解： 用户可以参考 MMDeploy 的 SDK 部署方式，使用 C++ 来进行部署，进而进一步提升推理速度。

```
cd ${MMYOLO_PATH}/demo
python deploy_demo.py \
    ${MMYOLO_PATH}/data/cat/images/mmexport1633684900217.jpg \
    ${MMYOLO_PATH}/configs/custom_dataset/yolov6_s_syncbn_fast_1xb32-100e_cat.py \
    /root/workspace/mmdeploy/work_dir/yolov6_s_syncbn_fast_1xb32-100e_cat_deploy_
↪dynamic_fp16/end2end.engine \
    --deploy-cfg ${MMYOLO_PATH}/configs/deploy/detection_tensorrt-fp16_dynamic-
↪192x192-960x960.py \
    --out-dir ${MMYOLO_PATH}/work_dirs/deploy_predict_out \
    --device cuda:0 \
```

(下页继续)

(续上页)

```
    --score-thr 0.5
```

警告： 该脚本 deploy_demo.py 暂时没有做批量推理的处理，而且代码前处理还需要完善，暂时不能完全展现出推理的速度，只能演示推理的结果，后续会优化，敬请期待。

执行之后，可以看到在 --out-dir 下面的推理图片结果：

注解： 您也可以做其他优化调整，例如增大 batch，量化 int8 等等。

11.1.4 保存和加载 Docker 容器

因为如果每次都进行 docker 镜像的构建，特别费时间，此时您可以考虑使用 docker 自带的打包 api 进行打包和加载。

```
# 保存，得到的 tar 包可以放到移动硬盘
docker save mmyolo-deploy > mmyolo-deploy.tar

# 加载镜像到系统
docker load < /path/to/mmyolo-deploy.tar
```

15.11.2 11.2 使用 projects/easydeploy 进行部署

参见：

详见部署文档

TODO: 下个版本会完善这个部分…

15.12 附录

15.12.1 1. 本教程训练机器的详细环境的资料如下：

```
sys.platform: linux
Python: 3.9.13 | packaged by conda-forge | (main, May 27 2022, 16:58:50) [GCC 10.3.0]
CUDA available: True
numpy_random_seed: 2147483648
GPU 0: NVIDIA GeForce RTX 3080 Ti
```

(下页继续)

(续上页)

```
CUDA_HOME: /usr/local/cuda
NVCC: Cuda compilation tools, release 11.5, V11.5.119
GCC: gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
PyTorch: 1.10.0
PyTorch compiling details: PyTorch built with:
  - GCC 7.3
  - C++ Version: 201402
  - Intel(R) oneAPI Math Kernel Library Version 2021.4-Product Build 20210904 for␣
↪Intel(R) 64 architecture applications
  - Intel(R) MKL-DNN v2.2.3 (Git Hash 7336ca9f055cf1bfa13efb658fe15dc9b41f0740)
  - OpenMP 201511 (a.k.a. OpenMP 4.5)
  - LAPACK is enabled (usually provided by MKL)
  - NNPACK is enabled
  - CPU capability usage: AVX2
  - CUDA Runtime 11.3
  - NVCC architecture flags: -gencode;arch=compute_37,code=sm_37;-gencode;
↪arch=compute_50,code=sm_50;-gencode;
                             arch=compute_60,code=sm_60;-gencode;arch=compute_61,
↪code=sm_61;-gencode;arch=compute_70,code=sm_70;
                             -gencode;arch=compute_75,code=sm_75;-gencode;
↪arch=compute_80,code=sm_80;-gencode;
                             arch=compute_86,code=sm_86;-gencode;arch=compute_37,
↪code=compute_37
  - CuDNN 8.2
  - Magma 2.5.2
  - Build settings: BLAS_INFO=mkl, BUILD_TYPE=Release, CUDA_VERSION=11.3, CUDNN_
↪VERSION=8.2.0,
                    CXX_COMPILER=/opt/rh/devtoolset-7/root/usr/bin/c++, CXX_FLAGS= -
↪Wno-deprecated -fvisibility-inlines-hidden
                    -DUSE_PTHREADPOOL -fopenmp -DNDEBUG -DUSE_KINETO -DUSE_FBGEMM -
↪DUSE_QNNPACK -DUSE_PYTORCH_QNNPACK -DUSE_XNNPACK
                    -DSYMBOLICATE_MOBILE_DEBUG_HANDLE -DEDGE_PROFILER_USE_KINETO -O2 -
↪fPIC -Wno-narrowing -Wall -Wextra
                    -Werror=return-type -Wno-missing-field-initializers -Wno-type-
↪limits -Wno-array-bounds -Wno-unknown-pragmas
                    -Wno-sign-compare -Wno-error=deprecated-declarations -Wno-
↪stringop-overflow -Wno-psabi -Wno-error=pedantic
                    -Wno-error=redundant-decls -Wno-error=old-style-cast -
↪fdiagnostics-color=always -faligned-new
                    -Wno-unused-but-set-variable -Wno-maybe-uninitialized -fno-math-
↪errno -fno-trapping-math -Werror=format
                    -Wno-stringop-overflow, LAPACK_INFO=mkl, PERF_WITH_AVX=1, PERF_
↪WITH_AVX2=1, PERF_WITH_AVX512=1,
```

(下页继续)

(续上页)

```
                    TORCH_VERSION=1.10.0, USE_CUDA=ON, USE_CUDNN=ON, USE_EXCEPTION_
↪PTR=1, USE_GFLAGS=OFF, USE_GLOG=OFF, USE_MKL=ON,
                    USE_MKLDNN=ON, USE_MPI=OFF, USE_NCCL=ON, USE_NNPACK=ON, USE_
↪OPENMP=ON,

TorchVision: 0.11.0
OpenCV: 4.6.0
MMEngine: 0.3.1
MMCV: 2.0.0rc3
MMDetection: 3.0.0rc3
MMYOLO: 0.2.0+cf279a5
```

15.12.2 2. 如何测试数据集在预训练权重的精度：

警告： 前提：该类在 COCO 80 类中！

本小节以 cat 数据集为例进行讲解，使用的是：

- config 文件：configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py
- 权重 yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth

1. 修改 config 文件中的路径

因为 configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py 是继承于 configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py，故主要修改 configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py 文件即可。

2. 修改标签

注解： 建议直接复制一份标签，防止弄坏好的标签

将 trainval.json 里面的“categories”字段改为 COCO 原本的：

```
  "categories": [{"supercategory": "person","id": 1,"name": "person"},{"supercategory
↪": "vehicle","id": 2,"name": "bicycle"},{"supercategory": "vehicle","id": 3,"name":
↪"car"},{"supercategory": "vehicle","id": 4,"name": "motorcycle"},{"supercategory":
↪"vehicle","id": 5,"name": "airplane"},{"supercategory": "vehicle","id": 6,"name":
↪"bus"},{"supercategory": "vehicle","id": 7,"name": "train"},{"supercategory":
↪"vehicle","id": 8,"name": "truck"},{"supercategory": "vehicle","id": 9,"name": "boat
↪"},{"supercategory": "outdoor","id": 10,"name": "traffic light"},{"supercategory":
↪"outdoor","id": 11,"name": "fire hydrant"},{"supercategory": "outdoor","id": 13,
↪"name": "stop sign"},{"supercategory": "outdoor","id": 14,"name": "parking meter"},{
↪"supercategory": "outdoor","id": 15,"name": "bench"},{"supercategory": "animal","id
↪": 16,"name": "bird"},{"supercategory": "animal","id": 17,"name": "cat"},{
↪"supercategory": "animal","id": 18,"name": "dog"},{"supercategory": "animal","id":
↪19,"name": "horse"},{"supercategory": "animal","id": 20,"name": "sheep"},{
↪"supercategory": "animal","id": 21,"name": "cow"},{"supercategory": "animal","id":
```

(下页继续)

(续上页)

同时，将 "annotations" 字段里面的 "category_id" 改为 COCO 对应的 id，例如本例子的 cat 是 17，下面展示部分修改结果：

```
  "annotations": [
    {
      "iscrowd": 0,
      "category_id": 17, # 这个 "category_id" 改为 COCO 对应的 id, 例如本例子的 cat 是 17
      "id": 32,
      "image_id": 32,
      "bbox": [
        822.49072265625,
        958.3897094726562,
        1513.693115234375,
        988.3231811523438
      ],
      "area": 1496017.9949368387,
      "segmentation": [
        [
          822.49072265625,
          958.3897094726562,
          822.49072265625,
          1946.712890625,
          2336.183837890625,
          1946.712890625,
          2336.183837890625,
          958.3897094726562
        ]
      ]
    }
  ]
```

3. 执行命令

```
python tools\test.py configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
                     work_dirs/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                     --cfg-options test_evaluator.classwise=True
```

执行之后就可以看到测试后的指标了：

```
+---------------+-------+--------------+-----+----------------+------+
| category      | AP    | category     | AP  | category       | AP   |
```

(下页继续)

(续上页)

person	nan	bicycle	nan	car	nan
motorcycle	nan	airplane	nan	bus	nan
train	nan	truck	nan	boat	nan
traffic light	nan	fire hydrant	nan	stop sign	nan
parking meter	nan	bench	nan	bird	nan
cat	0.866	dog	nan	horse	nan
sheep	nan	cow	nan	elephant	nan
bear	nan	zebra	nan	giraffe	nan
backpack	nan	umbrella	nan	handbag	nan
tie	nan	suitcase	nan	frisbee	nan
skis	nan	snowboard	nan	sports ball	nan
kite	nan	baseball bat	nan	baseball glove	nan
skateboard	nan	surfboard	nan	tennis racket	nan
bottle	nan	wine glass	nan	cup	nan
fork	nan	knife	nan	spoon	nan
bowl	nan	banana	nan	apple	nan
sandwich	nan	orange	nan	broccoli	nan
carrot	nan	hot dog	nan	pizza	nan
donut	nan	cake	nan	chair	nan
couch	nan	potted plant	nan	bed	nan
dining table	nan	toilet	nan	tv	nan
laptop	nan	mouse	nan	remote	nan
keyboard	nan	cell phone	nan	microwave	nan
oven	nan	toaster	nan	sink	nan
refrigerator	nan	book	nan	clock	nan
vase	nan	scissors	nan	teddy bear	nan
hair drier	nan	toothbrush	nan	None	None

CHAPTER 16

关于可视化的一切

本文包括特征图可视化和 Grad-Based 和 Grad-Free CAM 可视化

16.1 特征图可视化

MMYOLO 中，将使用 MMEngine 提供的 `Visualizer` 可视化器进行特征图可视化，其具备如下功能：

- 支持基础绘图接口以及特征图可视化。
- 支持选择模型中的不同层来得到特征图，包含 `squeeze_mean` ，`select_max` ，`topk` 三种显示方式，用户还可以使用 `arrangement` 自定义特征图显示的布局方式。

16.1.1 特征图绘制

你可以调用 `demo/featmap_vis_demo.py` 来简单快捷地得到可视化结果，为了方便理解，将其主要参数的功能梳理如下：

- `img`：选择要用于特征图可视化的图片，支持单张图片或者图片路径列表。
- `config`：选择算法的配置文件。
- `checkpoint`：选择对应算法的权重文件。
- `--out-file`：将得到的特征图保存到本地，并指定路径和文件名。
- `--device`：指定用于推理图片的硬件，`--device cuda：0` 表示使用第 1 张 GPU 推理，`--device cpu` 表示用 CPU 推理。

- `--score-thr`：设置检测框的置信度阈值，只有置信度高于这个值的框才会显示。
- `--preview-model`：可以预览模型，方便用户理解模型的特征层结构。
- `--target-layers`：对指定层获取可视化的特征图。
  - 可以单独输出某个层的特征图，例如：`--target-layers backbone`，`--target-layers neck`，`--target-layers backbone.stage4` 等。
  - 参数为列表时，也可以同时输出多个层的特征图，例如：`--target-layers backbone.stage4 neck` 表示同时输出 backbone 的 stage4 层和 neck 的三层一共四层特征图。
- `--channel-reduction`：输入的 Tensor 一般是包括多个通道的，`channel_reduction` 参数可以将多个通道压缩为单通道，然后和图片进行叠加显示，有以下三个参数可以设置：
  - `squeeze_mean`：将输入的 C 维度采用 mean 函数压缩为一个通道，输出维度变成 (1, H, W)。
  - `select_max`：将输入先在空间维度 sum，维度变成 (C,)，然后选择值最大的通道。
  - `None`：表示不需要压缩，此时可以通过 `topk` 参数可选择激活度最高的 `topk` 个特征图显示。
- `--topk`：只有在 `channel_reduction` 参数为 `None` 的情况下，`topk` 参数才会生效，其会按照激活度排序选择 `topk` 个通道，然后和图片进行叠加显示，并且此时会通过 `--arrangement` 参数指定显示的布局，该参数表示为一个数组，两个数字需要以空格分开，例如：`--topk 5 --arrangement 2 3` 表示以 `2 行 3 列`显示激活度排序最高的 5 张特征图，`--topk 7 --arrangement 3 3` 表示以 `3 行 3 列`显示激活度排序最高的 7 张特征图。
  - 如果 topk 不是 -1，则会按照激活度排序选择 topk 个通道显示。
  - 如果 topk = -1，此时通道 C 必须是 1 或者 3 表示输入数据是图片，否则报错提示用户应该设置 `channel_reduction` 来压缩通道。
- 考虑到输入的特征图通常非常小，函数默认将特征图进行上采样后方便进行可视化。

注意：当图片和特征图尺度不一样时候，`draw_featmap` 函数会自动进行上采样对齐。如果你的图片在推理过程中前处理存在类似 Pad 的操作此时得到的特征图也是 Pad 过的，那么直接上采样就可能会出现不对齐问题。

16.1.2 用法示例

以预训练好的 YOLOv5-s 模型为例：

请提前下载 YOLOv5-s 模型权重到本仓库根路径下：

```
cd mmyolo
wget https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_8xb16-
↪300e_coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth
```

(1) 将多通道特征图采用 `select_max` 参数压缩为单通道并显示，通过提取 `backbone` 层输出进行特征图可视化，将得到 `backbone` 三个输出层的特征图：

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco.py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                                --target-layers backbone \
                                --channel-reduction select_max
```

实际上上述代码存在图片和特征图不对齐问题，解决办法有两个：

1. 修改 YOLOv5 配置，让后处理只是简单的 Resize 即可，这对于可视化是没有啥影响的
2. 可视化时候图片应该用前处理后的，而不能用前处理前的

为了简单目前这里采用第一种解决办法，后续会采用第二种方案修复，让大家可以不修改配置即可使用。具体来说是将原先的 `test_pipeline` 替换为仅仅 Resize 版本。

旧的 `test_pipeline` 为：

```
test_pipeline = [
    dict(
        type='LoadImageFromFile'),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

修改为如下配置：

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # 这里将␣
↪LetterResize 修改成 mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
```

(下页继续)

(续上页)

```
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

正确效果如下：

(2) 将多通道特征图采用 squeeze_mean 参数压缩为单通道并显示，通过提取 neck 层输出进行特征图可视化，将得到 neck 三个输出层的特征图：

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco.py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                                --target-layers neck \
                                --channel-reduction squeeze_mean
```

(3) 将多通道特征图采用 squeeze_mean 参数压缩为单通道并显示，通过提取 backbone.stage4 和 backbone.stage3 层输出进行特征图可视化，将得到两个输出层的特征图：

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco.py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                                --target-layers backbone.stage4 backbone.stage3 \
                                --channel-reduction squeeze_mean
```

(4) 利用 --topk 3 --arrangement 2 2 参数选择多通道特征图中激活度最高的 3 个通道并采用 2x2 布局显示，用户可以通过 arrangement 参数选择自己想要的布局，特征图将自动布局，先按每个层中的 top3 特征图按 2x2 的格式布局，再将每个层按 2x2 布局：

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco.py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                                --target-layers backbone.stage3 backbone.stage4 \
                                --channel-reduction None \
                                --topk 3 \
                                --arrangement 2 2
```

(5) 存储绘制后的图片，在绘制完成后，可以选择本地窗口显示，也可以存储到本地，只需要加入参数 --out-file xxx.jpg：

```
python demo/featmap_vis_demo.py demo/dog.jpg \
                                configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco.py \
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                                --target-layers backbone \
                                --channel-reduction select_max \
                                --out-file featmap_backbone.jpg
```

16.2 Grad-Based 和 Grad-Free CAM 可视化

目标检测 CAM 可视化相比于分类 CAM 复杂很多且差异很大。本文只是简要说明用法，后续会单独开文档详细描述实现原理和注意事项。

你可以调用 `demo/boxmap_vis_demo.py` 来简单快捷地得到 Box 级别的 AM 可视化结果，目前已经支持 `YOLOv5/YOLOv6/YOLOX/RTMDet`。

以 YOLOv5 为例，和特征图可视化绘制一样，你需要先修改 `test_pipeline`，否则会出现特征图和原图不对齐问题。

旧的 `test_pipeline` 为：

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
        backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]
```

修改为如下配置：

```
test_pipeline = [
    dict(
        type='LoadImageFromFile',
```

(下页继续)

(续上页)

```
        backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=img_scale, keep_ratio=False), # 这里将␣
↪LetterResize 修改成 mmdet.Resize
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
```

(1) 使用 GradCAM 方法可视化 neck 模块的最后一个输出层的 AM 图

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth
```

相对应的特征图 AM 图如下：

可以看出 GradCAM 效果可以突出 box 级别的 AM 信息。

你可以通过 --topk 参数选择仅仅可视化预测分值最高的前几个预测框

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth \
        --topk 2
```

(2) 使用 AblationCAM 方法可视化 neck 模块的最后一个输出层的 AM 图

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth \
        --method ablationcam
```

由于 AblationCAM 是通过每个通道对分值的贡献程度来加权，因此无法实现类似 GradCAM 的仅仅可视化 box 级别的 AM 信息，但是你可以使用 --norm-in-bbox 来仅仅显示 bbox 内部 AM

```
python demo/boxam_vis_demo.py \
        demo/dog.jpg \
        configs/yolov5/yolov5_s-v61_syncbn_fast_8xb16-300e_coco.py \
        yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth \
        --method ablationcam \
```

(下页继续)

(续上页)

```
        --norm-in-bbox
```

16.3 可视化 COCO 标签

脚本 tools/analysis_tools/browse_coco_json.py 能够使用可视化显示 COCO 标签在图片的情况。

```
python tools/analysis_tools/browse_coco_json.py [--data-root ${DATA_ROOT}] \
                                                [--img-dir ${IMG_DIR}] \
                                                [--ann-file ${ANN_FILE}] \
                                                [--wait-time ${WAIT_TIME}] \
                                                [--disp-all] [--category-names␣
↪CATEGORY_NAMES [CATEGORY_NAMES ...]] \
                                                [--shuffle]
```

其中，如果图片、标签都在同一个文件夹下的话，可以指定 --data-root 到该文件夹，然后 --img-dir 和 --ann-file 指定该文件夹的相对路径，代码会自动拼接。如果图片、标签文件不在同一个文件夹下的话，则无需指定 --data-root ，直接指定绝对路径的 --img-dir 和 --ann-file 即可。

例子：

1. 查看 COCO 全部类别，同时展示 bbox、mask 等所有类型的标注：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --disp-all
```

如果图片、标签不在同一个文件夹下的话，可以使用绝对路径：

```
python tools/analysis_tools/browse_coco_json.py --img-dir '/dataset/image/coco/
↪train2017' \
                                                --ann-file '/label/instances_
↪train2017.json' \
                                                --disp-all
```

2. 查看 COCO 全部类别，同时仅展示 bbox 类型的标注，并打乱显示：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --shuffle
```

3. 只查看 bicycle 和 person 类别，同时仅展示 bbox 类型的标注：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --category-names 'bicycle' 'person'
```

4. 查看 COCO 全部类别，同时展示 bbox、mask 等所有类型的标注，并打乱显示：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --disp-all \
                                                --shuffle
```

16.4 可视化数据集

```
python tools/analysis_tools/browse_dataset.py \
    ${CONFIG_FILE} \
    [-o, --output-dir ${OUTPUT_DIR}] \
    [-p, --phase ${DATASET_PHASE}] \
    [-n, --show-number ${NUMBER_IMAGES_DISPLAY}] \
    [-i, --show-interval ${SHOW_INTERRVAL}] \
    [-m, --mode ${DISPLAY_MODE}] \
    [--cfg-options ${CFG_OPTIONS}]
```

所有参数的说明：

- config：模型配置文件的路径。
- -o, --output-dir: 保存图片文件夹，如果没有指定，默认为 './output'。
- **-p, --phase**: 可视化数据集的阶段，只能为 ['train', 'val', 'test'] 之一，默认为 'train'。
- **-n, --show-number**: 可视化样本数量。如果没有指定，默认展示数据集的所有图片。
- **-m, --mode**: 可视化的模式，只能为 ['original', 'transformed', 'pipeline'] 之一。默认为 'transformed'。
- --cfg-options：对配置文件的修改，参考学习配置文件。

```
`-m, --mode` 用于设置可视化的模式，默认设置为 'transformed'。
- 如果 `--mode` 设置为 'original'，则获取原始图片；
```

(下页继续)

(续上页)

```
- 如果 `--mode` 设置为 'transformed'，则获取预处理后的图片；
- 如果 `--mode` 设置为 'pipeline'，则获得数据流水线所有中间过程图片。
```

示例：

1. **‘original’**模式：

```
python ./tools/analysis_tools/browse_dataset.py configs/yolov5/yolov5_balloon.py --
↪phase val --output-dir tmp --mode original
```

- `--phase val`: 可视化验证集, 可简化为 `-p val`;
- `--output-dir tmp`: 可视化结果保存在“tmp”文件夹, 可简化为 `-o tmp`;
- `--mode original`: 可视化原图, 可简化为 `-m original`;
- `--show-number 100`: 可视化 100 张图，可简化为 `-n 100`;

2.’**transformed**’模式：

```
python ./tools/analysis_tools/browse_dataset.py configs/yolov5/yolov5_balloon.py
```

3.’**pipeline**’模式：

```
python ./tools/analysis_tools/browse_dataset.py configs/yolov5/yolov5_balloon.py -m␣
↪pipeline
```

16.5 可视化数据集分析

脚本 `tools/analysis_tools/dataset_analysis.py` 能够帮助用户得到四种功能的结果图，并将图片保存到当前运行目录下的 `dataset_analysis` 文件夹中。

关于该脚本的功能的说明：

通过 `main()` 的数据准备，得到每个子函数所需要的数据。

功能一：显示类别和 bbox 实例个数的分布图，通过子函数 `show_bbox_num` 生成。

功能二：显示类别和 bbox 实例宽、高的分布图，通过子函数 `show_bbox_wh` 生成。

功能三：显示类别和 bbox 实例宽/高比例的分布图，通过子函数 `show_bbox_wh_ratio` 生成。

功能四：基于面积规则下，显示类别和 bbox 实例面积的分布图，通过子函数 `show_bbox_area` 生成。

打印列表显示，通过脚本中子函数 `show_class_list` 和 `show_data_list` 生成。

```
python tools/analysis_tools/dataset_analysis.py ${CONFIG} \
                                                [-h] \
```

(下页继续)

(续上页)

```
                                                [--val-dataset ${TYPE}] \
                                                [--class-name ${CLASS_NAME}] \
                                                [--area-rule ${AREA_RULE}] \
                                                [--func ${FUNC}] \
                                                [--out-dir ${OUT_DIR}]
```

例子：

1. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，其中默认设置: 数据加载类型为 train_dataset ，面积规则设置为 [0,32,96,1e5]，生成包含所有类的结果图并将图片保存到当前运行目录下 ./dataset_analysis 文件夹中：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py
```

2. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --val-dataset 设置将数据加载类型由默认的 train_dataset 改为 val_dataset：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --val-dataset
```

3. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --class-name 设置将生成所有类改为特定类显示，以显示 person 为例：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --class-name person
```

4. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --area-rule 重新定义面积规则，以 30 70 125 为例, 面积规则变为 [0,30,70,125,1e5]：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --area-rule 30 70 125
```

5. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --func 设置，将显示四个功能效果图改为只显示 功能一为例：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --func show_bbox_num
```

6. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --out-dir 设置修改图片保存地址，以 work_dirs/dataset_analysis 地址为例：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                                --out-dir work_dirs/dataset_analysis
```

16.6 优化器参数策略可视化

tools/analysis_tools/vis_scheduler.py 旨在帮助用户检查优化器的超参数调度器（无需训练），支持学习率（learning rate）、动量（momentum）和权值衰减（weight decay）。

```
python tools/analysis_tools/vis_scheduler.py \
    ${CONFIG_FILE} \
    [-p, --parameter ${PARAMETER_NAME}] \
    [-d, --dataset-size ${DATASET_SIZE}] \
    [-n, --ngpus ${NUM_GPUs}] \
    [-o, --out-dir ${OUT_DIR}] \
    [--title ${TITLE}] \
    [--style ${STYLE}] \
    [--window-size ${WINDOW_SIZE}] \
    [--cfg-options]
```

所有参数的说明：

- config：模型配置文件的路径。
- **-p, parameter**: 可视化参数名，只能为 ["lr", "momentum", "wd"] 之一，默认为 "lr".
- **-d, --dataset-size**: 数据集的大小。如果指定，DATASETS.build 将被跳过并使用这个数值作为数据集大小，默认使用 DATASETS.build 所得数据集的大小。
- **-n, --ngpus**: 使用 GPU 的数量, 默认为 1。
- **-o, --out-dir**: 保存的可视化图片的文件夹路径，默认不保存。
- --title: 可视化图片的标题，默认为配置文件名。
- --style: 可视化图片的风格，默认为 whitegrid。
- --window-size: 可视化窗口大小，如果没有指定，默认为 12*7。如果需要指定，按照格式 'W*H'。
- --cfg-options: 对配置文件的修改，参考学习配置文件。

注解： 部分数据集在解析标注阶段比较耗时，推荐直接将 -d, dataset-size 指定数据集的大小，以节约时间。

你可以使用如下命令来绘制配置文件 configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_coco.py 将会使用的学习率变化曲线：

```
python tools/analysis_tools/vis_scheduler.py \
    configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_coco.py \
    --dataset-size 118287 \
    --ngpus 8 \
    --out-dir ./output
```

16.7 大图推理 (TODO)

CHAPTER 17

MMDeploy 部署必备教程

17.1 MMDeploy 部署

17.1.1 MMDeploy 介绍

MMDeploy 是 OpenMMLab 模型部署工具箱，**为各算法库提供统一的部署体验**。基于 MMDeploy，开发者可以轻松从训练 repo 生成指定硬件所需 SDK，省去大量适配时间。

更多介绍和使用指南见 https://mmdeploy.readthedocs.io/zh_CN/latest/get_started.html

17.1.2 算法支持列表

目前支持的 model-backend 组合：

ncnn 和其他后端的支持会在后续支持。

17.1.3 安装

按照说明安装 mmdeploy。

注解： 如果安装的是 mmdeploy 预编译包，那么也请通过‘git clone https://github.com/open-mmlab/mmdeploy.git –depth=1’下载 mmdeploy 源码。因为它包含了部署时所需的 tools 文件夹。

17.1.4 MMYOLO 中部署相关配置说明

所有部署配置文件在 configs/deploy 目录下。

您可以部署静态输入或者动态输入的模型，因此您需要修改模型配置文件中与此相关的数据处理流程。

MMDeploy 将后处理整合到自定义的算子中，因此您可以修改 codebase_config 中的 post_processing 参数来调整后处理策略，参数描述如下：

```
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
```

- score_threshold：在 nms 之前筛选候选框的类别分数阈值。
- confidence_threshold：在 nms 之前筛选候选框的置信度分数阈值。
- iou_threshold：在 nms 中去除重复框的 iou 阈值。
- max_output_boxes_per_class：每个类别最大的输出框数量。
- pre_top_k：在 nms 之前对候选框分数排序然后固定候选框的个数。
- keep_top_k：nms 算法最终输出的候选框个数。
- background_label_id：MMYOLO 算法中没有背景类别信息，置为 -1 即可。

静态输入配置

(1) 模型配置文件介绍

以 MMYOLO 中的 YOLOv5 模型配置为例，下面是对部署时使用的模型配置文件参数说明介绍。

```
_base_ = '../../yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'

test_pipeline = [
    dict(type='LoadImageFromFile', backend_args=_base_.backend_args),
    dict(
```

(下页继续)

(续上页)

```
        type='LetterResize',
        scale=_base_.img_scale,
        allow_scale_up=False,
        use_mini_pad=False,
    ),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

test_dataloader = dict(
    dataset=dict(pipeline=test_pipeline, batch_shapes_cfg=None))
```

`_base_ = '../../yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'` 继承了训练时构建模型的配置。

`test_pipeline` 为部署时对输入图像进行处理的流程，`LetterResize` 控制了输入图像的尺寸，同时限制了导出模型所能接受的输入尺寸。

`test_dataloader` 为部署时构建数据加载器配置，`batch_shapes_cfg` 控制了是否启用 `batch_shapes` 策略，详细内容可以参考*yolov5 配置文件说明* 。

(2) 部署配置文件介绍

以 MMYOLO 中的 YOLOv5 部署配置为例，下面是对配置文件参数说明介绍。

ONNXRuntime 部署 YOLOv5 可以使用 `detection_onnxruntime_static.py` 配置。

```
_base_ = ['./base_static.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
```

(下页继续)

(续上页)

```
backend_config = dict(type='onnxruntime')
```

backend_config 中指定了部署后端 type='onnxruntime'，其他信息可参考第三小节。

TensorRT 部署 YOLOv5 可以使用 detection_tensorrt_static-640x640.py 配置。

```
_base_ = ['./base_static.py']
onnx_config = dict(input_shape=(640, 640))
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 640, 640],
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 640, 640])))
    ])
use_efficientnms = False
```

backend_config 中指定了后端 type= ‘tensorrt’。

与 ONNXRuntime 部署配置不同的是，TensorRT 需要指定输入图片尺寸和构建引擎文件需要的参数，包括：

- onnx_config 中指定 input_shape=(640, 640)
- backend_config['common_config'] 中包括 fp16_mode=False 和 max_workspace_size=1 << 30，fp16_mode 表示是否以 fp16 的参数格式构建引擎，max_workspace_size 表示当前 gpu 设备最大显存，单位为 GB。fp16 的详细配置可以参考 detection_tensorrt-fp16_static-640x640.py
- backend_config['model_inputs']['input_shapes']['input'] 中 min_shape /opt_shape/max_shape 对应的值在静态输入下应该保持相同，即默认均为 [1, 3, 640, 640]。

use_efficientnms 是 MMYOLO 系列新引入的配置，表示在导出 onnx 时是否启用 Efficient NMS Plugin 来替换 MMDeploy 中的 TRTBatchedNMS plugin 。

可以参考 TensorRT 官方实现的 Efficient NMS Plugin 获取更多详细信息。

注意，这个功能仅仅在 TensorRT >= 8.0 版本才能使用，无需编译开箱即用。

动态输入配置

(1) 模型配置文件介绍

当您部署动态输入模型时，您无需修改任何模型配置文件，仅需要修改部署配置文件即可。

(2) 部署配置文件介绍

ONNXRuntime 部署 YOLOv5 可以使用 detection_onnxruntime_dynamic.py 配置。

```
_base_ = ['./base_dynamic.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
backend_config = dict(type='onnxruntime')
```

backend_config 中指定了后端 type='onnxruntime'，其他配置与上一节在 ONNXRuntime 部署静态输入模型相同。

TensorRT 部署 YOLOv5 可以使用 detection_tensorrt_dynamic-192x192-960x960.py 配置。

```
_base_ = ['./base_dynamic.py']
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 192, 192],
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 960, 960])))
    ])
use_efficientnms = False
```

backend_config 中指定了后端 type='tensorrt'，由于 TensorRT 动态输入与静态输入有所不同，您可以了解更多动态输入相关信息通过访问 TensorRT dynamic input official introduction。

TensorRT 部署需要配置 min_shape, opt_shape, max_shape ，TensorRT 限制输入图片的尺寸在 min_shape 和 max_shape 之间。

min_shape 为输入图片的最小尺寸，opt_shape 为输入图片常见尺寸，在这个尺寸下推理性能最好，max_shape 为输入图片的最大尺寸。

use_efficientnms 配置与上节 TensorRT 静态输入配置相同。

INT8 量化配置

!!! 部署 TensorRT INT8 模型教程即将发布!!!

17.1.5 模型转换

使用方法

从源码安装的 MMDeploy

设置 MMDeploy 根目录为环境变量 MMDEPLOY_DIR，例如 export MMDEPLOY_DIR=/the/root/path/of/MMDeploy

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    ${DEPLOY_CFG_PATH} \
    ${MODEL_CFG_PATH} \
    ${MODEL_CHECKPOINT_PATH} \
    ${INPUT_IMG} \
    --test-img ${TEST_IMG} \
    --work-dir ${WORK_DIR} \
    --calib-dataset-cfg ${CALIB_DATA_CFG} \
    --device ${DEVICE} \
    --log-level INFO \
    --show \
    --dump-info
```

参数描述

- deploy_cfg：mmdeploy 针对此模型的部署配置，包含推理框架类型、是否量化、输入 shape 是否动态等。配置文件之间可能有引用关系，configs/deploy/detection_onnxruntime_static.py 是一个示例。
- model_cfg：MMYOLO 算法库的模型配置，例如 configs/deploy/model/yolov5_s-deploy.py, 与 mmdeploy 的路径无关。
- checkpoint：torch 模型路径。可以 http/https 开头，详见 mmengine.fileio 的实现。
- img：模型转换时，用做测试的图像文件路径。
- --test-img：用于测试模型的图像文件路径。默认设置成 None。
- --work-dir：工作目录，用来保存日志和模型文件。
- --calib-dataset-cfg：此参数只有 int8 模式下生效，用于校准数据集配置文件。若在 int8 模式下未传入参数，则会自动使用模型配置文件中的’val’数据集进行校准。
- --device：用于模型转换的设备。默认是 cpu，对于 trt 可使用 cuda:0 这种形式。
- --log-level：设置日记的等级，选项包括'CRITICAL','FATAL','ERROR','WARN','WARNING', 'INFO', 'DEBUG', 'NOTSET'。默认是 INFO。
- --show：是否显示检测的结果。
- --dump-info：是否输出 SDK 信息。

通过 pip install 安装的 MMDeploy

假设当前的工作目录为 mmyolo 的根目录，那么以 YoloV5 模型为例，你可以从此处下载对应的 checkpoint，并使用以下代码将之转换为 onnx 模型：

```
from mmdeploy.apis import torch2onnx
from mmdeploy.backend.sdk.export_info import export2SDK

img = 'demo/demo.jpg'
work_dir = 'mmdeploy_models/mmyolo/onnx'
save_file = 'end2end.onnx'
deploy_cfg = 'configs/deploy/detection_onnxruntime_dynamic.py'
model_cfg = 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model_checkpoint = 'checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth'
device = 'cpu'

# 1. convert model to onnx
torch2onnx(img, work_dir, save_file, deploy_cfg, model_cfg,
```

(下页继续)

(续上页)

```
            model_checkpoint, device)

# 2. extract pipeline info for inference by MMDeploy SDK
export2SDK(deploy_cfg, model_cfg, work_dir, pth=model_checkpoint,
            device=device)
```

17.1.6 模型规范

在使用转换后的模型进行推理之前，有必要了解转换结果的结构。它存放在 `--work-dir` 指定的路路径下。

上例中的 `mmdeploy_models/mmyolo/onnx`，结构如下：

```
mmdeploy_models/mmyolo/onnx
├── deploy.json
├── detail.json
├── end2end.onnx
└── pipeline.json
```

重要的是：

- **end2end.onnx**: 推理引擎文件。可用 ONNX Runtime 推理
- ***xxx*.json**: mmdeploy SDK 推理所需的 meta 信息

整个文件夹被定义为 **mmdeploy SDK model**。换言之，**mmdeploy SDK model** 既包括推理引擎，也包括推理 meta 信息。

17.1.7 模型推理

后端模型推理

以上述模型转换后的 `end2end.onnx` 为例，你可以使用如下代码进行推理：

```
from mmdeploy.apis.utils import build_task_processor
from mmdeploy.utils import get_input_shape, load_config
import torch

deploy_cfg = 'configs/deploy/detection_onnxruntime_dynamic.py'
model_cfg = 'configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
device = 'cpu'
backend_model = ['mmdeploy_models/mmyolo/onnx/end2end.onnx']
image = 'demo/demo.jpg'

# read deploy_cfg and model_cfg
```

(下页继续)

(续上页)

```
deploy_cfg, model_cfg = load_config(deploy_cfg, model_cfg)

# build task and backend model
task_processor = build_task_processor(model_cfg, deploy_cfg, device)
model = task_processor.build_backend_model(backend_model)

# process input image
input_shape = get_input_shape(deploy_cfg)
model_inputs, _ = task_processor.create_input(image, input_shape)

# do model inference
with torch.no_grad():
    result = model.test_step(model_inputs)

# visualize results
task_processor.visualize(
    image=image,
    model=model,
    result=result[0],
    window_name='visualize',
    output_file='work_dir/output_detection.png')
```

运行上述代码后，你可以在 work_dir 中看到推理的结果图片 output_detection.png。

SDK 模型推理

你也可以参考如下代码，对 SDK model 进行推理：

```
from mmdeploy_runtime import Detector
import cv2

img = cv2.imread('demo/demo.jpg')

# create a detector
detector = Detector(model_path='mmdeploy_models/mmyolo/onnx',
                    device_name='cpu', device_id=0)
# perform inference
bboxes, labels, masks = detector(img)

# visualize inference result
indices = [i for i in range(len(bboxes))]
for index, bbox, label_id in zip(indices, bboxes, labels):
    [left, top, right, bottom], score = bbox[0:4].astype(int), bbox[4]
```

(下页继续)

(续上页)

```
    if score < 0.3:
        continue

    cv2.rectangle(img, (left, top), (right, bottom), (0, 255, 0))

cv2.imwrite('work_dir/output_detection.png', img)
```

除了 python API，mmdeploy SDK 还提供了诸如 C、C++、C#、Java 等多语言接口。你可以参考样例学习其他语言接口的使用方法。

17.1.8 模型评测

当您将 PyTorch 模型转换为后端支持的模型后，您可能需要验证模型的精度，使用 `${MMDEPLOY_DIR}/tools/test.py`

```
python3 ${MMDEPLOY_DIR}/tools/test.py \
    ${DEPLOY_CFG} \
    ${MODEL_CFG} \
    --model ${BACKEND_MODEL_FILES} \
    --device ${DEVICE} \
    --work-dir ${WORK_DIR} \
    [--cfg-options ${CFG_OPTIONS}] \
    [--show] \
    [--show-dir ${OUTPUT_IMAGE_DIR}] \
    [--interval ${INTERVAL}] \
    [--wait-time ${WAIT_TIME}] \
    [--log2file work_dirs/output.txt]
    [--speed-test] \
    [--warmup ${WARM_UP}] \
    [--log-interval ${LOG_INTERVERL}] \
    [--batch-size ${BATCH_SIZE}] \
    [--uri ${URI}]
```

参数描述

- `deploy_cfg`: 部署配置文件。
- `model_cfg`: MMYOLO 模型配置文件。
- `--model`: 导出的后端模型。例如, 如果我们导出了 TensorRT 模型，我们需要传入后缀为“.engine”文件路径。
- `--device`: 运行模型的设备。请注意，某些后端会限制设备。例如，TensorRT 必须在 cuda 上运行。

- --work-dir: 模型转换、报告生成的路径。
- --cfg-options: 传入额外的配置，将会覆盖当前部署配置。
- --show: 是否在屏幕上显示评估结果。
- --show-dir: 保存评估结果的目录。(只有给出这个参数才会保存结果)。
- --interval: 屏幕上显示评估结果的间隔。
- --wait-time: 每个窗口的显示时间
- --log2file: 将评估结果（和速度）记录到文件中。
- --speed-test: 是否开启速度测试。
- --warmup: 在计算推理时间之前进行预热，需要先开启 speed-test。
- --log-interval: 每个日志之间的间隔，需要先设置 speed-test。
- --batch-size: 推理的批量大小，它将覆盖数据配置中的 samples_per_gpu。默认为 1。请注意，并非所有模型都支持 batch_size > 1。
- --uri: 在边缘设备上推理时的 ipv4 或 ipv6 端口号。

注意: ${MMDEPLOY_DIR}/tools/test.py 中的其他参数用于速度测试。他们不影响评估。

17.2 YOLOv5 部署全流程说明

请先参考部署必备指南 了解部署配置文件等相关信息。

17.2.1 模型训练和测试

模型训练和测试请参考*YOLOv5 从入门到部署全流程* 。

17.2.2 准备 MMDeploy 运行环境

安装 MMDeploy 请参考 源码手动安装 ，选择您所使用的平台编译 MMDeploy 和自定义算子。

注意！如果环境安装有问题，可以查看 MMDeploy FAQ 或者在 issuse 中提出您的问题。

17.2.3 准备模型配置文件

本例将以基于 coco 数据集预训练的 YOLOv5 配置和权重进行部署的全流程讲解，包括静态/动态输入模型导出和推理，TensorRT / ONNXRuntime 两种后端部署和测试。

静态输入配置

(1) 模型配置文件

当您需要部署静态输入模型时，您应该确保模型的输入尺寸是固定的，比如在测试流程或测试数据集加载时输入尺寸为 640x640。

您可以查看 yolov5_s-static.py 中测试流程或测试数据集加载部分，如下所示：

```
_base_ = '../../yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'

test_pipeline = [
    dict(type='LoadImageFromFile', backend_args=_base_.backend_args),
    dict(
        type='LetterResize',
        scale=_base_.img_scale,
        allow_scale_up=False,
        use_mini_pad=False,
    ),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

test_dataloader = dict(
    dataset=dict(pipeline=test_pipeline, batch_shapes_cfg=None))
```

由于 yolov5 在测试时会开启 allow_scale_up 和 use_mini_pad 改变输入图像的尺寸来取得更高的精度，但是会给部署静态输入模型造成输入尺寸不匹配的问题。

该配置相比与原始配置文件进行了如下修改:

- 关闭 test_pipline 中改变尺寸相关的配置，如 LetterResize 中 allow_scale_up=False 和 use_mini_pad=False 。
- 关闭 test_dataloader 中 batch shapes 策略，即 batch_shapes_cfg=None 。

(2) 部署配置文件

当您部署在 ONNXRuntime 时，您可以查看 detection_onnxruntime_static.py ，如下所示：

```
_base_ = ['./base_static.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
backend_config = dict(type='onnxruntime')
```

默认配置中的 post_processing 后处理参数是当前模型与 pytorch 模型精度对齐的配置，若您需要修改相关参数，可以参考部署必备指南 的详细介绍。

当您部署在 TensorRT 时，您可以查看 detection_tensorrt_static-640x640.py ，如下所示：

```
_base_ = ['./base_static.py']
onnx_config = dict(input_shape=(640, 640))
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 640, 640],
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 640, 640])))
    ])
use_efficientnms = False
```

本例使用了默认的输入尺寸 input_shape=(640, 640) ，构建网络以 fp32 模式即 fp16_mode=False，并且默认构建 TensorRT 构建引擎所使用的显存 max_workspace_size=1 << 30 即最大为 1GB 显存。

动态输入配置

(1) 模型配置文件

当您需要部署动态输入模型时，模型的输入可以为任意尺寸 (TensorRT 会限制最小和最大输入尺寸)，因此使用默认的 yolov5_s-v61_syncbn_8xb16-300e_coco.py 模型配置文件即可，其中数据处理和数据集加载器部分如下所示：

```
batch_shapes_cfg = dict(
    type='BatchShapePolicy',
    batch_size=val_batch_size_per_gpu,
    img_size=img_scale[0],
    size_divisor=32,
    extra_pad_ratio=0.5)

test_pipeline = [
    dict(type='LoadImageFromFile', backend_args=_base_.backend_args),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True, _scope_='mmdet'),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

val_dataloader = dict(
    batch_size=val_batch_size_per_gpu,
    num_workers=val_num_workers,
    persistent_workers=persistent_workers,
    pin_memory=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        test_mode=True,
        data_prefix=dict(img='val2017/'),
        ann_file='annotations/instances_val2017.json',
        pipeline=test_pipeline,
        batch_shapes_cfg=batch_shapes_cfg))
```

其中 LetterResize 类初始化传入了 allow_scale_up=False 控制输入的小图像是否上采样，同时默认 use_mini_pad=False 关闭了图片最小填充策略，val_dataloader['dataset'] 中传入了 batch_shapes_cfg=batch_shapes_cfg，即按照 batch 内的输入尺寸进行最小填充。上述策略会改变输入图像的尺寸，因此动态输入模型在测试时会按照上述数据集加载器动态输入。

(2) 部署配置文件

当您部署在 ONNXRuntime 时，您可以查看 detection_onnxruntime_dynamic.py ，如下所示：

```
_base_ = ['./base_dynamic.py']
codebase_config = dict(
    type='mmyolo',
    task='ObjectDetection',
    model_type='end2end',
    post_processing=dict(
        score_threshold=0.05,
        confidence_threshold=0.005,
        iou_threshold=0.5,
        max_output_boxes_per_class=200,
        pre_top_k=5000,
        keep_top_k=100,
        background_label_id=-1),
    module=['mmyolo.deploy'])
backend_config = dict(type='onnxruntime')
```

与静态输入配置仅有 _base_ = ['./base_dynamic.py'] 不同，动态输入会额外继承 dynamic_axes 属性。其他配置与静态输入配置相同。

当您部署在 TensorRT 时，您可以查看 detection_tensorrt_dynamic-192x192-960x960.py ，如下所示：

```
_base_ = ['./base_dynamic.py']
backend_config = dict(
    type='tensorrt',
    common_config=dict(fp16_mode=False, max_workspace_size=1 << 30),
    model_inputs=[
        dict(
            input_shapes=dict(
                input=dict(
                    min_shape=[1, 3, 192, 192],
                    opt_shape=[1, 3, 640, 640],
                    max_shape=[1, 3, 960, 960])))
    ])
use_efficientnms = False
```

本例构建网络以 fp32 模式即 fp16_mode=False，构建 TensorRT 构建引擎所使用的显存 max_workspace_size=1 << 30 即最大为 1GB 显存。

同时默认配置 min_shape=[1, 3, 192, 192]，opt_shape=[1, 3, 640, 640]，max_shape=[1, 3, 960, 960]，意为该模型所能接受的输入尺寸最小为 192x192，最大为 960x960，最常见尺寸为 640x640。

当您部署自己的模型时，需要根据您的输入图像尺寸进行调整。

17.2.4 模型转换

本教程所使用的 MMDeploy 根目录为 /home/openmmlab/dev/mmdeploy，请注意修改为您的 MMDeploy 目录。预训练权重下载于 yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth，保存在本地的 /home/openmmlab/dev/mmdeploy/yolov5s.pth。

```
wget https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_8xb16-
↪300e_coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth -O /
↪home/openmmlab/dev/mmdeploy/yolov5s.pth
```

命令行执行以下命令配置相关路径：

```
export MMDEPLOY_DIR=/home/openmmlab/dev/mmdeploy
export PATH_TO_CHECKPOINTS=/home/openmmlab/dev/mmdeploy/yolov5s.pth
```

YOLOv5 静态输入模型导出

ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_onnxruntime_static.py \
    configs/deploy/model/yolov5_s-static.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cpu
```

TensorRT

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_tensorrt_static-640x640.py \
    configs/deploy/model/yolov5_s-static.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cuda:0
```

YOLOv5 动态输入模型导出

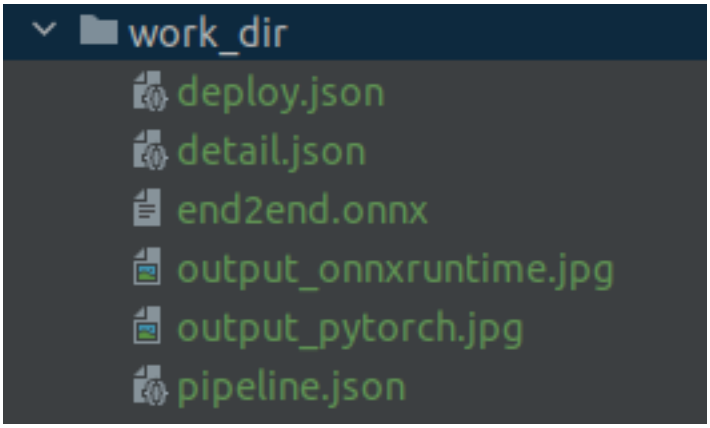
ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_onnxruntime_dynamic.py \
    configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cpu
    --dump-info
```

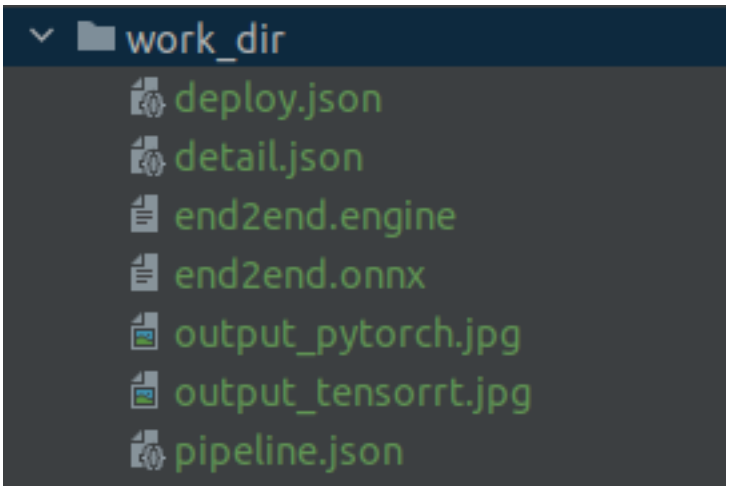
TensorRT

```
python3 ${MMDEPLOY_DIR}/tools/deploy.py \
    configs/deploy/detection_tensorrt_dynamic-192x192-960x960.py \
    configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py \
    ${PATH_TO_CHECKPOINTS} \
    demo/demo.jpg \
    --work-dir work_dir \
    --show \
    --device cuda:0
    --dump-info
```

当您使用上述命令转换模型时，您将会在 `work_dir` 文件夹下发现以下文件：



或者



在导出 `onnxruntime` 模型后，您将得到图 1 的六个文件，其中 `end2end.onnx` 表示导出的 `onnxruntime` 模型，`xxx.json` 表示 `MMDeploy SDK` 推理所需要的 meta 信息。

在导出 `TensorRT` 模型后，您将得到图 2 的七个文件，其中 `end2end.onnx` 表示导出的中间模型，`MMDeploy` 利用该模型自动继续转换获得 `end2end.engine` 模型用于 `TensorRT` 部署，`xxx.json` 表示 `MMDeploy SDK` 推理所需要的 meta 信息。

17.2.5 模型评测

当您转换模型成功后，可以使用 `${MMDEPLOY_DIR}/tools/test.py` 工具对转换后的模型进行评测。下面是对 `ONNXRuntime` 和 `TensorRT` 静态模型的评测，动态模型评测修改传入模型配置即可。

ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/test.py \
        configs/deploy/detection_onnxruntime_static.py \
        configs/deploy/model/yolov5_s-static.py \
        --model work_dir/end2end.onnx  \
        --device cpu \
        --work-dir work_dir
```

执行完成您将看到命令行输出检测结果指标如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.373
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.566
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.405
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.206
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.419
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.489
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.311
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.511
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.565
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.377
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.620
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.718
11/02 10:22:22 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.206 0.419 0.489
```

TensorRT

注意：TensorRT 需要执行设备是 `cuda`

```
python3 ${MMDEPLOY_DIR}/tools/test.py \
        configs/deploy/detection_tensorrt_static-640x640.py \
        configs/deploy/model/yolov5_s-static.py \
        --model work_dir/end2end.engine  \
        --device cuda:0 \
        --work-dir work_dir
```

执行完成您将看到命令行输出检测结果指标如下：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.373
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.566
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.405
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.205
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.418
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.489
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.310
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.507
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.556
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.351
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.610
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.714
11/02 10:21:31 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.205 0.418 0.489
```

未来我们将会支持模型测速等更加实用的脚本

17.3 使用 Docker 部署测试

MMYOLO 提供了一个 Dockerfile 用于构建镜像。请确保您的 docker 版本大于等于 19.03。

温馨提示；国内用户建议取消掉 Dockerfile 里面 Optional 后两行的注释，可以获得火箭一般的下载提速：

```
# (Optional)
RUN sed -i 's/http:\/\/archive.ubuntu.com\/ubuntu\//http:\/\/mirrors.aliyun.com\/
↪ubuntu\//g' /etc/apt/sources.list && \
    pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

构建命令：

```
# build an image with PyTorch 1.12, CUDA 11.6, TensorRT 8.2.4 ONNXRuntime 1.8.1
docker build -f docker/Dockerfile_deployment -t mmyolo:v1 .
```

用以下命令运行 Docker 镜像：

```
export DATA_DIR=/path/to/your/dataset
docker run --gpus all --shm-size=8g -it --name mmyolo -v ${DATA_DIR}:/openmmlab/
↪mmyolo/data/coco mmyolo:v1
```

DATA_DIR 是 COCO 数据的路径。

复制以下脚本到 docker 容器 /openmmlab/mmyolo/script.sh:

```
#!/bin/bash
wget -q https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_s-v61_syncbn_fast_
↪8xb16-300e_coco/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187
↪pth \
```

(下页继续)

(续上页)

```
  -O yolov5s.pth
export MMDEPLOY_DIR=/openmmlab/mmdeploy
export PATH_TO_CHECKPOINTS=/openmmlab/mmyolo/yolov5s.pth

python3 ${MMDEPLOY_DIR}/tools/deploy.py \
  configs/deploy/detection_tensorrt_static-640x640.py \
  configs/deploy/model/yolov5_s-static.py \
  ${PATH_TO_CHECKPOINTS} \
  demo/demo.jpg \
  --work-dir work_dir_trt \
  --device cuda:0

python3 ${MMDEPLOY_DIR}/tools/test.py \
  configs/deploy/detection_tensorrt_static-640x640.py \
  configs/deploy/model/yolov5_s-static.py \
  --model work_dir_trt/end2end.engine \
  --device cuda:0 \
  --work-dir work_dir_trt

python3 ${MMDEPLOY_DIR}/tools/deploy.py \
  configs/deploy/detection_onnxruntime_static.py \
  configs/deploy/model/yolov5_s-static.py \
  ${PATH_TO_CHECKPOINTS} \
  demo/demo.jpg \
  --work-dir work_dir_ort \
  --device cpu

python3 ${MMDEPLOY_DIR}/tools/test.py \
  configs/deploy/detection_onnxruntime_static.py \
  configs/deploy/model/yolov5_s-static.py \
  --model work_dir_ort/end2end.onnx \
  --device cpu \
  --work-dir work_dir_ort
```

在 /openmmlab/mmyolo 下运行：

```
sh script.sh
```

脚本会自动下载 MMYOLO 的 YOLOv5 预训练权重并使用 MMDeploy 进行模型转换和测试。您将会看到以下输出：

- TensorRT：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.37294
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.56565
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.40474
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.20472
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.41857
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.48867
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.31000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.50722
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.55619
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.35140
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.61063
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.71398
11/03 05:31:50 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.205 0.419 0.489
```

- ONNXRuntime：

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.37329
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.56650
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.40494
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.20608
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.41881
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.48873
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.31057
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.51073
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.56509
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.37725
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.62035
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.71803
11/03 05:47:44 - mmengine - INFO - bbox_mAP_copypaste: 0.373 0.566 0.405 0.206 0.419 0.489
```

可以看到，经过 MMDeploy 部署的模型与 MMYOLO-YOLOv5 的 mAP-37.7 差距在 1% 以内。

如果您需要测试您的模型推理速度，可以使用以下命令：

- TensorRT

```
python3 ${MMDEPLOY_DIR}/tools/profiler.py \
  configs/deploy/detection_tensorrt_static-640x640.py \
  configs/deploy/model/yolov5_s-static.py \
  data/coco/val2017 \
  --model work_dir_trt/end2end.engine \
  --device cuda:0
```

- ONNXRuntime

```
python3 ${MMDEPLOY_DIR}/tools/profiler.py \
  configs/deploy/detection_onnxruntime_static.py \
  configs/deploy/model/yolov5_s-static.py \
```

(下页继续)

(续上页)

```
  data/coco/val2017 \
  --model work_dir_ort/end2end.onnx \
  --device cpu
```

17.3.1 模型推理

后端模型推理

ONNXRuntime

以上述模型转换后的 end2end.onnx 为例，您可以使用如下代码进行推理：

```
from mmdeploy.apis.utils import build_task_processor
from mmdeploy.utils import get_input_shape, load_config
import torch

deploy_cfg = './configs/deploy/detection_onnxruntime_dynamic.py'
model_cfg = '../mmyolo/configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
device = 'cpu'
backend_model = ['./work_dir/end2end.onnx']
image = '../mmyolo/demo/demo.jpg'

# read deploy_cfg and model_cfg
deploy_cfg, model_cfg = load_config(deploy_cfg, model_cfg)

# build task and backend model
task_processor = build_task_processor(model_cfg, deploy_cfg, device)
model = task_processor.build_backend_model(backend_model)

# process input image
input_shape = get_input_shape(deploy_cfg)
model_inputs, _ = task_processor.create_input(image, input_shape)

# do model inference
with torch.no_grad():
    result = model.test_step(model_inputs)

# visualize results
task_processor.visualize(
    image=image,
    model=model,
    result=result[0],
```

(下页继续)

(续上页)

```
    window_name='visualize',
    output_file='work_dir/output_detection.png')
```

TensorRT

以上述模型转换后的 end2end.engine 为例，您可以使用如下代码进行推理：

```
from mmdeploy.apis.utils import build_task_processor
from mmdeploy.utils import get_input_shape, load_config
import torch

deploy_cfg = './configs/deploy/detection_tensorrt_dynamic-192x192-960x960.py'
model_cfg = '../mmyolo/configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py'
device = 'cuda:0'
backend_model = ['./work_dir/end2end.engine']
image = '../mmyolo/demo/demo.jpg'

# read deploy_cfg and model_cfg
deploy_cfg, model_cfg = load_config(deploy_cfg, model_cfg)

# build task and backend model
task_processor = build_task_processor(model_cfg, deploy_cfg, device)
model = task_processor.build_backend_model(backend_model)

# process input image
input_shape = get_input_shape(deploy_cfg)
model_inputs, _ = task_processor.create_input(image, input_shape)

# do model inference
with torch.no_grad():
    result = model.test_step(model_inputs)

# visualize results
task_processor.visualize(
    image=image,
    model=model,
    result=result[0],
    window_name='visualize',
    output_file='work_dir/output_detection.png')
```

SDK 模型推理

ONNXRuntime

以上述模型转换后的 end2end.onnx 为例，您可以使用如下代码进行 SDK 推理：

```
from mmdeploy_runtime import Detector
import cv2

img = cv2.imread('../mmyolo/demo/demo.jpg')

# create a detector
detector = Detector(model_path='work_dir',
                    device_name='cpu', device_id=0)
# perform inference
bboxes, labels, masks = detector(img)

# visualize inference result
indices = [i for i in range(len(bboxes))]
for index, bbox, label_id in zip(indices, bboxes, labels):
    [left, top, right, bottom], score = bbox[0:4].astype(int), bbox[4]
    if score < 0.3:
        continue

    cv2.rectangle(img, (left, top), (right, bottom), (0, 255, 0))

cv2.imwrite('work_dir/output_detection.png', img)
```

TensorRT

以上述模型转换后的 end2end.engine 为例，您可以使用如下代码进行 SDK 推理：

```
from mmdeploy_runtime import Detector
import cv2

img = cv2.imread('../mmyolo/demo/demo.jpg')

# create a detector
detector = Detector(model_path='work_dir',
                    device_name='cuda', device_id=0)
# perform inference
bboxes, labels, masks = detector(img)
```

(下页继续)

(续上页)

```
# visualize inference result
indices = [i for i in range(len(bboxes))]
for index, bbox, label_id in zip(indices, bboxes, labels):
    [left, top, right, bottom], score = bbox[0:4].astype(int), bbox[4]
    if score < 0.3:
        continue

    cv2.rectangle(img, (left, top), (right, bottom), (0, 255, 0))

cv2.imwrite('work_dir/output_detection.png', img)
```

除了 python API，mmdeploy SDK 还提供了诸如 C、C++、C#、Java 等多语言接口。你可以参考样例学习其他语言接口的使用方法。

CHAPTER 18

EasyDeploy 部署必备教程

18.1 EasyDeploy 部署

本项目作为 MMYOLO 的部署 project 单独存在，意图剥离 MMDeploy 当前的体系，独自支持用户完成模型训练后的转换和部署功能，使用户的学习和工程成本下降。

当前支持对 ONNX 格式和 TensorRT 格式的转换，后续对其他推理平台也会支持起来。

CHAPTER 19

常见错误排除步骤

本文档收集用户经常碰到的常见错误情况，并提供详细的排查步骤。如果你发现阅读本文你没有找到正确的解决方案，请联系我们或者提 PR 进行更新。提 PR 请参考*如何给 MMYOLO 贡献代码*

19.1 xxx is not in the model registry

这个错误信息是指某个模块没有被注册到 model 中。这个错误出现的原因非常多，典型的情况有：

1. 你新增的模块没有在类别前面加上注册器装饰器 @MODELS.register_module()
2. 虽然注册了，但是注册错了位置，例如你实际想注册到 MMYOLO 中，但是你导入的 MODELS 是 MMDet 包里面的
3. 你注册了且注册正确了，但是没有在对应的 `__init__.py` 中加入导致没有被导入
4. 以上 3 个步骤都确认没问题，但是你是新增 py 文件来自定义模块的却没有重新安装 MMYOLO 导致没有生效，此时你可以重新安装一遍，即使你是 -e 模式安装也需要重新安装
5. 如果你是在 mmyolo 包路径下新增了一个 package，除上述步骤外，你还需要在 register_all_modules 函数中增加其导包代码，否则该 package 不会被自动触发
6. 你的环境中有多个版本 MMYOLO，你注册的和实际运行的实际上不是同一套代码，导致没有生效。此时你可以在程序运行前输入 `PYTHONPATH="$(dirname $0)/..":$PYTHONPATH` 强行使用当前代码

19.2 loss_bbox 始终为 0

该原因出现主要有两个原因：

1. 训练过程中没有 GT 标注数据
2. 参数设置不合理导致训练中没有正样本

第一种情况出现的概率更大。`loss_bbox` 通常是只考虑正样本的 loss，如果训练中没有正样本则始终为 0。如果是第一种原因照成的 `loss_bbox` 始终为 0，那么通常意味着你配置不对，特别是 dataset 部分的配置不正确。一个非常典型的情况是用户的 `dataset` 中 `metainfo` 设置不正确或者设置了但是没有传给 dataset 导致加载后没有找到对应类别的 GT Bbox 标注。这种情况请仔细阅读我们提供的 示例配置 。验证 dataset 配置是否正确的一个最直接的途径是运行 browse_dataset 脚本，如果可视化效果正确则说明是正确的。

19.3 MMCV 安装时间非常久

这通常意味着你在自己编译 MMCV 而不是直接下载使用我们提供的预编译包。MMCV 中包括了大量的自定义的 CUDA 算子，如果从源码安装则需要非常久的时间去编译，并且由于其安装成功依赖于严格的底层环境信息，需要多个库的版本一致才可以。如果用户自己编译大概率会失败。我们不推荐用户自己去编译 MMCV 而应该优先选择预编译包。如果你当前的环境中我们没有提供对应的预编译包，那么建议你可以快速换一个 Conda 环境，并安装有预编译包的 Torch。以 torch1.8.0+cu102 为例，如果你想查看目前查看所有的预编译包，可以查看 https://download.openmmlab.com/mmcv/dist/cu102/torch1.8.0/index.html。

19.4 基于官方配置继承新建的配置出现 unexpected keyword argument

这通常是由于你没有删除 base 配置中的额外参数。可以在你新建配置所修改的字典中增加 `_delete_=True` 删掉 base 中该类之前的所有参数。

19.5 The testing results of the whole dataset is empty

这通常说明训练效果太差导致网络没有预测出任何符合阈值要求的检测框。出现这种现象有多个原因，典型的为：

1. 当前为前几个 epoch，网络当前训练效果还较差，等后续训练久一点后可能就不会出现该警告了
2. 配置设置不正确，网络虽然正常训练但是实际上无效训练，例如前面的 `loss_bbox` 始终为 0 就会导致上述警告
3. 超参设置不合理

19.6 ValueError: not enough values to unpack(expected 2, got 0)

这个错误通常是在 epoch 切换时候出现。这是 PyTorch 1.7 的已知问题，在 PyTorch 1.8+ 中已经修复。如果在 PyTorch 1.7 中想修复这个问题，可以简单的设置 dataloader 参数 `persistent_workers` 为 False。

19.7 ValueError: need at least one array to concatenate

这个是一个非常常见的错误，可能出现在训练一开始或者训练正常但是评估时候。不管出现在何阶段，均说明你的配置不对：

1. 最常见的错误就是 `num_classes` 参数设置不对。在 MMYOLO 或者 MMDet 中大部分配置都是以 COCO 数据为例，因此配置中默认的 `num_classes` 是 80, 如果用户自定义数据集没有正确修改这个字段则会出现上述错误。MMYOLO 中有些算法配置会在多个模块中都需要 `num_classes` 参数，用户经常出现的错误就是仅仅修改了某一个地方的 `num_classes` 而没有将所有的 `num_classes` 都修改。想快速解决这个问题，可以使用 print_config 脚本打印下全配置，然后全局搜索 `num_classes` 确认是否有没有修改的模块。
2. 该错误还可能会出现在对 `dataset.metainfo.classes` 参数设置不对造成的。当用户希望训练自己的数据集但是未能正确的修改 `dataset.metainfo.classes` 参数，而默认的使用 `COCO` 数据集中的类别时，且用户自定义数据集的所有类别不在 `COCO` 数据集的类别里就会出现该错误。这时需要用户核对并修改正确的 `dataset.metainfo.classes` 信息。

19.8 评估时候 IndexError: list index out of range

具体输出信息是

```
  File "site-packages/mmdet/evaluation/metrics/coco_metric.py", line 216, in␣
↪results2json
    data['category_id'] = self.cat_ids[label]
IndexError: list index out of range
```

可以看出是评估时候类别索引越界，这个通常的原因是配置中的 `num_classes` 设置不正确，默认的 `num_classes` 是 80，如果你自定义类别小于 80，那么就有可能出现类别越界。注意算法配置的 `num_classes` 一般会用到多个模块，你可能只改了某几个而漏掉了一些。想快速解决这个问题，可以使用 print_config 脚本打印下全配置，然后全局搜索 `num_classes` 确认是否有没有修改的模块。

19.9 训练中不打印 loss，但是程序依然正常训练和评估

这通常是因为一个训练 epoch 没有超过 50 个迭代，而 MMYOLO 中默认的打印间隔是 50。你可以修改 `default_hooks.logger.interval` 参数。

19.10 GPU out of memory

1. 存在大量 ground truth boxes 或者大量 anchor 的场景，可能在 assigner 会 OOM。
2. 使用–amp 来开启混合精度训练。
3. 你也可以尝试使用 MMDet 中的 AvoidCUDAOOM 来避免该问题。首先它将尝试调用 torch.cuda.empty_cache()。如果失败，将会尝试把输入类型转换到 FP16。如果仍然失败，将会把输入从 GPUs 转换到 CPUs 进行计算。这里提供了两个使用的例子：

```
from mmdet.utils import AvoidCUDAOOM

output = AvoidCUDAOOM.retry_if_cuda_oom(some_function)(input1, input2)
```

你也可也使用 AvoidCUDAOOM 作为装饰器让代码遇到 OOM 的时候继续运行：

```
from mmdet.utils import AvoidCUDAOOM

@AvoidCUDAOOM.retry_if_cuda_oom
def function(*args, **kwargs):
    ...
    return xxx
```

19.11 Loss goes Nan

1. 检查数据的标注是否正常，长或宽为 0 的框可能会导致回归 loss 变为 nan，一些小尺寸（宽度或高度小于 1）的框在数据增强后也会导致此问题。因此，可以检查标注并过滤掉那些特别小甚至面积为 0 的框，并关闭一些可能会导致 0 面积框出现数据增强。
2. 降低学习率：由于某些原因，例如 batch size 大小的变化，导致当前学习率可能太大。您可以降低为可以稳定训练模型的值。
3. 延长 warm up 的时间：一些模型在训练初始时对学习率很敏感。
4. 添加 gradient clipping: 一些模型需要梯度裁剪来稳定训练过程。你可以在 config 设置 `optim_wrapper.clip_grad=dict(max_norm=xx)`

19.12 训练中其他不符合预期或者错误

如果训练或者评估中出现了不属于上述描述的问题，由于原因不明，现提供常用的排除流程：

1. 首先确认配置是否正确，可以使用 print_config 脚本打印全部配置，如果运行成功则说明配置语法没有错误
2. 确认 COCO 格式的 json 标注是否正确，可以使用 browse_coco_json.py 脚本确认
3. 确认 dataset 部分配置是否正确，这一步骤几乎是必须要提前运行的，可以提前排查很多问题，可以使用 browse_dataset.py 脚本确认
4. 如果以上 3 步都没有问题，那么出问题可能在 model 部分了。这个部分的排除没有特别的办法，你可以单独写一个脚本来仅运行 model 部分并通过调试来确认，如果对于 model 中多个模块的输入构建存在困惑，可以参考对应模块的单元测试写法

CHAPTER 20

MM 系列仓库必备基础

CHAPTER 21

数据集格式准备和说明

21.1 DOTA 数据集

21.1.1 下载 DOTA 数据集

数据集可以从 DOTA 数据集的主页 DOTA 或 OpenDataLab 下载。

我们推荐使用 OpenDataLab 下载，其中的文件夹结构已经按照需要排列好了，只需要解压即可，不需要费心去调整文件夹结构。

下载后解压数据集，并按如下文件夹结构存放。

```
${DATA_ROOT}
├── train
|   ├── images
|   |   ├── P0000.png
|   |   ├── ...
|   ├── labelTxt-v1.0
|   |   ├── labelTxt
|   |   |   ├── P0000.txt
|   |   |   ├── ...
|   |   ├── trainset_reclabelTxt
|   |   |   ├── P0000.txt
|   |   |   ├── ...
├── val
```

(下页继续)

(续上页)

```
|   ├── images
|   |   ├── P0003.png
|   |   ├── ...
|   ├── labelTxt-v1.0
|   |   ├── labelTxt
|   |   |   ├── P0003.txt
|   |   |   ├── ...
|   |   ├── valset_reclabelTxt
|   |   |   ├── P0003.txt
|   |   |   ├── ...
├── test
|   ├── images
|   |   ├── P0006.png
|   |   ├── ...
```

其中，以 `reclabelTxt` 为结尾的文件夹存放了水平检测框的标注，目前仅使用了 `labelTxt-v1.0` 中旋转框的标注。

21.1.2 数据集切片

我们提供了 `tools/dataset_converters/dota/dota_split.py` 脚本用于 DOTA 数据集的准备和切片。

```
python tools/dataset_converters/dota/dota_split.py \
    [--splt-config ${SPLIT_CONFIG}] \
    [--data-root ${DATA_ROOT}] \
    [--out-dir ${OUT_DIR}] \
    [--ann-subdir ${ANN_SUBDIR}] \
    [--phase ${DATASET_PHASE}] \
    [--nproc ${NPROC}] \
    [--save-ext ${SAVE_EXT}] \
    [--overwrite]
```

脚本依赖于 shapely 包，请先通过 `pip install shapely` 安装 shapely。

参数说明:

- `--splt-config` : 切片参数的配置文件。
- `--data-root`: DOTA 数据集的存放位置。
- `--out-dir`: 切片后的输出位置。
- `--ann-subdir`: 标注文件夹的名字。默认为 `labelTxt-v1.0` 。

- --phase: 数据集的阶段。默认为 trainval test 。
- --nproc: 进程数量。默认为 8 。
- --save-ext: 输出图像的扩展名，如置空则与原图保持一致。默认为 None 。
- --overwrite: 如果目标文件夹已存在，是否允许覆盖。

基于 DOTA 数据集论文中提供的配置，我们提供了两种切片配置。

./split_config/single_scale.json 用于单尺度 single-scale 切片 ./split_config/multi_scale.json 用于多尺度 multi-scale 切片

DOTA 数据集通常使用 trainval 集进行训练，然后使用 test 集进行在线验证，大多数论文提供的也是在线验证的精度。如果你需要进行本地验证，可以准备 train 集和 val 集进行训练和测试。

示例:

使用单尺度切片配置准备 trainval 和 test 集

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/single_scale.json'
    --data-root ${DATA_ROOT} \
    --out-dir ${OUT_DIR}
```

准备 DOTA-v1.5 数据集，它的标注文件夹名字是 labelTxt-v1.5

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/single_scale.json'
    --data-root ${DATA_ROOT} \
    --out-dir ${OUT_DIR} \
    --ann-subdir 'labelTxt-v1.5'
```

使用单尺度切片配置准备 train 和 val 集

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/single_scale.json'
    --data-root ${DATA_ROOT} \
    --phase train val \
    --out-dir ${OUT_DIR}
```

使用多尺度切片配置准备 trainval 和 test 集

```
python tools/dataset_converters/dota/dota_split.py
    --split-config 'tools/dataset_converters/dota/split_config/multi_scale.json'
    --data-root ${DATA_ROOT} \
    --out-dir ${OUT_DIR}
```

在运行完成后，输出的结构如下：

```
${OUT_DIR}
├── trainval
│   ├── images
│   │   ├── P0000__1024__0___0.png
│   │   ├── ...
│   ├── annfiles
│   │   ├── P0000__1024__0___0.txt
│   │   ├── ...
├── test
│   ├── images
│   │   ├── P0006__1024__0___0.png
│   │   ├── ...
│   ├── annfiles
│   │   ├── P0006__1024__0___0.txt
│   │   ├── ...
```

此时将配置文件中的 `data_root` 修改为 ${OUT_DIR} 即可开始使用 DOTA 数据集训练。

CHAPTER 22

恢复训练

恢复训练是指从之前某次训练保存下来的状态开始继续训练，这里的状态包括模型的权重、优化器和优化器参数调整策略的状态。

用户可以在训练命令最后加上 `--resume` 恢复训练，程序会自动从 `work_dirs` 中加载最新的权重文件恢复训练。如果 `work_dir` 中有最新的 checkpoint（例如该训练在上一次训练时被中断），则会从该 checkpoint 恢复训练，否则（例如上一次训练还没来得及保存 checkpoint 或者启动了新的训练任务）会重新开始训练。下面是一个恢复训练的示例:

```
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py --resume
```

CHAPTER 23

自动混合精度（AMP）训练

如果要开启自动混合精度（AMP）训练，在训练命令最后加上 --amp 即可，命令如下：

```
python tools/train.py python ./tools/train.py ${CONFIG} --amp
```

具体例子如下：

```
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py --amp
```

CHAPTER 24

多尺度训练和测试

24.1 多尺度训练

MMYOLO 中目前支持了主流的 YOLOv5、YOLOv6、YOLOv7、YOLOv8 和 RTMDet 等算法，其默认配置均为单尺度 640x640 训练。在 MM 系列开源库中常用的多尺度训练有两种实现方式：

1. 在 `train_pipeline` 中输出的每张图都是不定尺度的，然后在 DataPreprocessor 中将不同尺度的输入图片通过 stack_batch 函数填充到同一尺度，从而组成 batch 进行训练。MMDet 中大部分算法都是采用这个实现方式。
2. 在 `train_pipeline` 中输出的每张图都是固定尺度的，然后直接在 `DataPreprocessor` 中进行 batch 张图片的上下采样，从而实现多尺度训练功能

在 MMYOLO 中两种多尺度训练方式都是支持的。理论上第一种实现方式所生成的尺度会更加丰富，但是由于其对单张图进行独立增强，训练效率不如第二种方式。所以我们更推荐使用第二种方式。

以 `configs/yolov5/yolov5_s-v61_fast_1xb12-40e_cat.py` 配置为例，其默认配置采用的是 640x640 固定尺度训练，假设想实现以 32 为倍数，且多尺度范围为 (480, 800) 的训练方式，则可以参考 YOLOX 做法通过 DataPreprocessor 中的 YOLOXBatchSyncRandomResize 实现。

在 `configs/yolov5` 路径下新建配置，命名为 `configs/yolov5/yolov5_s-v61_fast_1xb12-ms-40e_cat.py`，其内容如下：

```
_base_ = 'yolov5_s-v61_fast_1xb12-40e_cat.py'

model = dict(
```

(下页继续)

(续上页)

```
    data_preprocessor=dict(
        type='YOLOv5DetDataPreprocessor',
        pad_size_divisor=32,
        batch_augments=[
            dict(
                type='YOLOXBatchSyncRandomResize',
                # 多尺度范围是 480~800
                random_size_range=(480, 800),
                # 输出尺度需要被 32 整除
                size_divisor=32,
                # 每隔 1 个迭代改变一次输出输出
                interval=1)
        ])
)
```

上述配置就可以实现多尺度训练了。为了方便，我们已经在 `configs/yolov5/` 下已经提供了该配置。其余 YOLO 系列算法也是类似做法。

24.2 多尺度测试

MMYOLO 多尺度测试功能等同于测试时增强 TTA，目前已经支持，详情请查看*测试时增强 TTA* 。

CHAPTER 25

测试时增强相关说明

25.1 测试时增强 TTA

MMYOLO 在 v0.5.0+ 版本中增加对 TTA 的支持，用户可以在进行评估时候指定 `--tta` 参数使用。以 `YOLOv5-s` 为例，其单卡 TTA 测试命令为：

```
python tools/test.py configs/yolov5/yolov5_n-v61_syncbn_fast_8xb16-300e_coco.py␣
↪https://download.openmmlab.com/mmyolo/v0/yolov5/yolov5_n-v61_syncbn_fast_8xb16-300e_
↪coco/yolov5_n-v61_syncbn_fast_8xb16-300e_coco_20220919_090739-b804c1ad.pth  --tta
```

TTA 功能的正常运行必须确保配置中存在 `tta_model` 和 `tta_pipeline` 两个变量，详情可以参考 det_p5_tta.py。

MMYOLO 中默认的 TTA 会先执行 3 个多尺度增强，然后在每个尺度中执行 2 种水平翻转增强，一共 6 个并行的 pipeline。以 `YOLOv5-s` 为例，其 TTA 配置为：

```
img_scales = [(640, 640), (320, 320), (960, 960)]

_multiscale_resize_transforms = [
    dict(
        type='Compose',
        transforms=[
            dict(type='YOLOv5KeepRatioResize', scale=s),
            dict(
                type='LetterResize',
```

(下页继续)

(续上页)

```
                scale=s,
                allow_scale_up=False,
                pad_val=dict(img=114))
        ]) for s in img_scales
]

tta_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='TestTimeAug',
        transforms=[
            _multiscale_resize_transforms,
            [
                dict(type='mmdet.RandomFlip', prob=1.),
                dict(type='mmdet.RandomFlip', prob=0.)
            ], [dict(type='mmdet.LoadAnnotations', with_bbox=True)],
            [
                dict(
                    type='mmdet.PackDetInputs',
                    meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                               'scale_factor', 'pad_param', 'flip',
                               'flip_direction'))
            ]
        ])
]
```

其示意图如下所示：

```
                              LoadImageFromFile
                    /                 |                     \
(RatioResize,LetterResize) (RatioResize,LetterResize) (RatioResize,LetterResize)
       /      \                   /      \                   /        \
 RandomFlip RandomFlip      RandomFlip RandomFlip        RandomFlip RandomFlip
     |          |               |          |                |          |
 LoadAnn    LoadAnn          LoadAnn    LoadAnn          LoadAnn    LoadAnn
     |          |               |          |                |          |
 PackDetIn  PackDetIn        PackDetIn  PackDetIn        PackDetIn  PackDetIn
```

你可以修改 `img_scales` 来支持不同的多尺度增强，也可以插入新的 pipeline 从而实现自定义 TTA 需求。假设你只想进行水平翻转增强，则配置应该修改为如下：

```
tta_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
```

(下页继续)

(续上页)

```
        type='TestTimeAug',
        transforms=[
            [
                dict(type='mmdet.RandomFlip', prob=1.),
                dict(type='mmdet.RandomFlip', prob=0.)
            ], [dict(type='mmdet.LoadAnnotations', with_bbox=True)],
            [
                dict(
                    type='mmdet.PackDetInputs',
                    meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                               'scale_factor', 'pad_param', 'flip',
                               'flip_direction'))
            ]
        ])
]
```

CHAPTER 26

给主干网络增加插件

MMYOLO 支持在 Backbone 的不同 Stage 后增加如 none_local、dropblock 等插件，用户可以直接通过修改 config 文件中 backbone 的 plugins 参数来实现对插件的管理。例如为 YOLOv5 增加 GeneralizedAttention 插件，其配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    backbone=dict(
        plugins=[
            dict(
                cfg=dict(
                    type='GeneralizedAttention',
                    spatial_range=-1,
                    num_heads=8,
                    attention_type='0011',
                    kv_stride=2),
                stages=(False, False, True, True))
        ]))
```

cfg 参数表示插件的具体配置，stages 参数表示是否在 backbone 对应的 stage 后面增加插件，长度需要和 backbone 的 stage 数量相同。

目前 MMYOLO 支持了如下插件：

1. CBAM

2. GeneralizedAttention
3. NonLocal2d
4. ContextBlock

CHAPTER 27

冻结指定网络层权重

27.1 冻结 backbone 权重

在 MMYOLO 中我们可以通过设置 frozen_stages 参数去冻结主干网络的部分 stage, 使这些 stage 的参数不参与模型的更新。需要注意的是：frozen_stages = i 表示的意思是指从最开始的 stage 开始到第 i 层 stage 的所有参数都会被冻结。下面是 YOLOv5 的例子，其他算法也是同样的逻辑：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    backbone=dict(
        frozen_stages=1 # 表示第一层 stage 以及它之前的所有 stage 中的参数都会被冻结
    ))
```

27.2 冻结 neck 权重

MMYOLO 中也可以通过参数 freeze_all 去冻结整个 neck 的参数。下面是 YOLOv5 的例子，其他算法也是同样的逻辑：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    neck=dict(
```

(下页继续)

(续上页)

```
        freeze_all=True # freeze_all=True 时表示整个 neck 的参数都会被冻结
    ))
```

CHAPTER 28

输出模型预测结果

如果想将预测结果保存为特定的文件，用于离线评估，目前 MMYOLO 支持 json 和 pkl 两种格式。

注解： json 文件仅保存 image_id、bbox、score 和 category_id；json 文件可以使用 json 库读取。pkl 保存内容比 json 文件更多，还会保存预测图片的文件名和尺寸等一系列信息；pkl 文件可以使用 pickle 库读取。

28.1 输出为 json 文件

如果想将预测结果输出为 json 文件，则命令如下：

```
python tools/test.py ${CONFIG} ${CHECKPOINT} --json-prefix ${JSON_PREFIX}
```

--json-prefix 后的参数输入为文件名前缀（无需输入 .json 后缀），也可以包含路径。举一个具体例子：

```
python tools/test.py configs\yolov5\yolov5_s-v61_syncbn_8xb16-300e_coco.py yolov5_s-
↪v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth --json-prefix work_
↪dirs/demo/json_demo
```

运行以上命令会在 work_dirs/demo 文件夹下，输出 json_demo.bbox.json 文件。

28.2 输出为 pkl 文件

如果想将预测结果输出为 pkl 文件，则命令如下：

```
python tools/test.py ${CONFIG} ${CHECKPOINT} --out ${OUTPUT_FILE} [--cfg-options $
↪{OPTIONS [OPTIONS...]}]
```

--out 后的参数输入为完整文件名（**必须输入** .pkl 或 .pickle 后缀），也可以包含路径。举一个具体例子：

```
python tools/test.py configs\yolov5\yolov5_s-v61_syncbn_8xb16-300e_coco.py yolov5_s-
↪v61_syncbn_fast_8xb16-300e_coco_20220918_084700-86e02187.pth --out work_dirs/demo/
↪pkl_demo.pkl
```

运行以上命令会在 work_dirs/demo 文件夹下，输出 pkl_demo.pkl 文件。

CHAPTER 29

设置随机种子

如果想要在训练时指定随机种子，可以使用以下命令：

```
python ./tools/train.py \
    ${CONFIG} \                                    # 配置文件路径
    --cfg-options randomness.seed=2023 \           # 设置随机种子为 2023
    [randomness.diff_rank_seed=True] \             # 根据 rank 来设置不同的种子。
    [randomness.deterministic=True]                # 把 cuDNN 后端确定性选项设置为 True
# [] 代表可选参数，实际输入命令行时，不用输入 []
```

randomness 有三个参数可设置，具体含义如下：

- randomness.seed=2023，设置随机种子为 2023。
- randomness.diff_rank_seed=True，根据 rank 来设置不同的种子，diff_rank_seed 默认为 False。
- randomness.deterministic=True，把 cuDNN 后端确定性选项设置为 True，即把 torch.backends.cudnn.deterministic 设为 True，把 torch.backends.cudnn.benchmark 设为 False。deterministic 默认为 False。更多细节见 https://pytorch.org/docs/stable/notes/randomness.html。

CHAPTER 30

算法组合替换教程

30.1 Loss 组合替换教程

OpenMMLab 2.0 体系中 MMYOLO、MMDetection、MMClassification 中的 loss 注册表都继承自 MMEngine 中的根注册表。因此用户可以在 MMYOLO 中使用来自 MMDetection、MMClassification 中实现的 loss 而无需重新实现。

30.1.1 替换 YOLOv5 Head 中的 loss_cls 函数

1. 假设我们想使用 `LabelSmoothLoss` 作为 `loss_cls` 的损失函数。因为 `LabelSmoothLoss` 已经在 MMClassification 中实现了，所以可以直接在配置文件中进行替换。配置文件如下：

```
# 请先使用命令： mim install "mmcls>=1.0.0rc2", 安装 mmcls
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model = dict(
    bbox_head=dict(
      loss_cls=dict(
        _delete_=True,
        _scope_='mmcls', #  临时替换 scope 为 mmcls
        type='LabelSmoothLoss',
        label_smooth_val=0.1,
        mode='multi_label',
        reduction='mean',
        loss_weight=0.5)))
```

2. 假设我们想使用 VarifocalLoss 作为 loss_cls 的损失函数。因为 VarifocalLoss 在 MMDetection 已经实现好了，所以可以直接替换。配置文件如下：

```
model = dict(
    bbox_head=dict(
        loss_cls=dict(
            _delete_=True,
            _scope_='mmdet',
            type='VarifocalLoss',
            loss_weight=1.0)))
```

3. 假设我们想使用 FocalLoss 作为 loss_cls 的损失函数。配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model = dict(
    bbox_head=dict(
        loss_cls= dict(
            _delete_=True,
            _scope_='mmdet',
            type='FocalLoss',
            loss_weight=1.0)))
```

4. 假设我们想使用 QualityFocalLoss 作为 loss_cls 的损失函数。配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model = dict(
    bbox_head=dict(
      loss_cls= dict(
        _delete_=True,
        _scope_='mmdet',
        type='QualityFocalLoss',
        loss_weight=1.0)))
```

30.1.2 替换 YOLOv5 Head 中的 loss_obj 函数

loss_obj 的替换与 loss_cls 的替换类似，我们可以使用已经实现好的损失函数对 loss_obj 的损失函数进行替换

1. 假设我们想使用 VarifocalLoss 作为 loss_obj 的损失函数

```
model = dict(
    bbox_head=dict(
        loss_obj=dict(
            _delete_=True,
```

(下页继续)

(续上页)

```
            _scope_='mmdet',
            type='VarifocalLoss',
            loss_weight=1.0)))
```

2. 假设我们想使用 FocalLoss 作为 loss_obj 的损失函数。

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model = dict(
    bbox_head=dict(
        loss_cls= dict(
            _delete_=True,
            _scope_='mmdet',
            type='FocalLoss',
            loss_weight=1.0)))
```

3. 假设我们想使用 QualityFocalLoss 作为 loss_obj 的损失函数。

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model = dict(
    bbox_head=dict(
      loss_cls= dict(
        _delete_=True,
        _scope_='mmdet',
        type='QualityFocalLoss',
        loss_weight=1.0)))
```

注意

1. 在本教程中损失函数的替换是运行不报错的，但无法保证性能一定会上升。
2. 本次损失函数的替换都是以 YOLOv5 算法作为例子的，但是 MMYOLO 下的多个算法，如 YOLOv6，YOLOX 等算法都可以按照上述的例子进行替换。

30.2 Model 和 Loss 组合替换

在 MMYOLO 中，model 即网络本身和 loss 是解耦的，用户可以简单的通过修改配置文件中 model 和 loss 来组合不同模块。下面给出两个具体例子。

(1) YOLOv5 model 组合 YOLOv7 loss，配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
model = dict(
```

(下页继续)

(续上页)

```
    bbox_head=dict(
        _delete_=True,
        type='YOLOv7Head',
        head_module=dict(
            type='YOLOv5HeadModule',
            num_classes=80,
            in_channels=[256, 512, 1024],
            widen_factor=0.5,
            featmap_strides=[8, 16, 32],
            num_base_priors=3)))
```

(2) RTMDet model 组合 YOLOv6 loss，配置文件如下：

```
_base_ = './rtmdet_l_syncbn_8xb32-300e_coco.py'
model = dict(
    bbox_head=dict(
        _delete_=True,
        type='YOLOv6Head',
        head_module=dict(
            type='RTMDetSepBNHeadModule',
            num_classes=80,
            in_channels=256,
            stacked_convs=2,
            feat_channels=256,
            norm_cfg=dict(type='BN'),
            act_cfg=dict(type='SiLU', inplace=True),
            share_conv=True,
            pred_kernel_size=1,
            featmap_strides=[8, 16, 32]),
        loss_bbox=dict(
            type='IoULoss',
            iou_mode='giou',
            bbox_format='xyxy',
            reduction='mean',
            loss_weight=2.5,
            return_iou=False)),
    train_cfg=dict(
        _delete_=True,
        initial_epoch=4,
        initial_assigner=dict(
            type='BatchATSSAssigner',
            num_classes=80,
            topk=9,
            iou_calculator=dict(type='mmdet.BboxOverlaps2D')),
```

(下页继续)

(续上页)

```
        assigner=dict(
            type='BatchTaskAlignedAssigner',
            num_classes=80,
            topk=13,
            alpha=1,
            beta=6)
    ))
```

30.3 Backbone + Neck + HeadModule 的组合替换

30.3.1 1. YOLOv5 Backbone 替换

(1) 假设想将 RTMDet backbone + yolov5 neck + yolov5 head 作为 YOLOv5 的完整网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

widen_factor = 0.5
deepen_factor = 0.33

model = dict(
    backbone=dict(
        _delete_=True,
        type='CSPNeXt',
        arch='P5',
        expand_ratio=0.5,
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        channel_attention=True,
        norm_cfg=dict(type='BN'),
        act_cfg=dict(type='SiLU', inplace=True))
)
```

(2) YOLOv6EfficientRep backbone + yolov5 neck + yolov5 head 作为 YOLOv5 的完整网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    backbone=dict(
        type='YOLOv6EfficientRep',
```

(下页继续)

(续上页)

```
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='ReLU', inplace=True))
)
```

30.3.2 2. YOLOv5 Neck 替换

(1) 假设想将 yolov5 backbone + yolov6 neck + yolov5 head 作为 YOLOv5 的完整网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

model = dict(
    neck = dict(
        type = 'YOLOv6RepPAFPN',
        in_channels = [256, 512, 1024],
        out_channels = [128, 256, 512], # 注意 YOLOv6RepPAFPN 的输出通道是 [128, 256,␣
↪512]
        num_csp_blocks = 12,
        act_cfg = dict(type='ReLU', inplace = True),
    ),
    bbox_head = dict(
        head_module = dict(
            in_channels = [128, 256, 512])) # head 部分输入通道要做相应更改
)
```

(2) 假设想将 yolov5 backbone + yolov7 neck + yolov5 head 作为 YOLOv5 的完整网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = _base_.widen_factor

model = dict(
    neck = dict(
        _delete_=True, # 将 _base_ 中关于 neck 的字段删除
        type = 'YOLOv7PAFPN',
        deepen_factor = deepen_factor,
        widen_factor = widen_factor,
        upsample_feats_cat_first = False,
        in_channels = [256, 512, 1024],
        out_channels = [128, 256, 512],
```

(下页继续)

(续上页)

```
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg = dict(type='SiLU', inplace=True),
    ),
    bbox_head = dict(
        head_module = dict(
            in_channels = [256, 512, 1024])) # 注意使用 YOLOv7PAFPN 后 head 部分输入通道数是 neck 输出通道数的两倍
)
```

30.3.3 3. YOLOv5 HeadModule 替换

(1) 假设想将 yolov5 backbone + yolov5 neck + yolo7 headmodule 作为 YOLOv5 的完整网络，则配置文件如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

strides = [8, 16, 32]
num_classes = 1 # 根据自己的数据集调整

model = dict(
    bbox_head=dict(
        type='YOLOv7Head',
        head_module=dict(
            type='YOLOv7HeadModule',
            num_classes=num_classes,
            in_channels=[256, 512, 1024],
            featmap_strides=strides,
            num_base_priors=3)))
```

CHAPTER 31

使用 mim 跨库调用其他 OpenMMLab 仓库的脚本

注解:

1. 目前暂不支持跨库调用所有脚本，正在修复中。等修复完成，本文档会添加更多的例子。
2. 绘制 mAP 和计算平均训练速度两项功能在 MMDetection dev-3.x 分支中修复，目前需要通过源码安装该分支才能成功调用。

31.1 日志分析

31.1.1 曲线图绘制

MMDetection 中的 `tools/analysis_tools/analyze_logs.py` 可利用指定的训练 log 文件绘制 loss/mAP 曲线图，第一次运行前请先运行 `pip install seaborn` 安装必要依赖。

```
mim run mmdet analyze_logs plot_curve \
    ${LOG} \                                    # 日志文件路径
    [--keys ${KEYS}] \                          # 需要绘制的指标，默认为 'bbox_mAP'
    [--start-epoch ${START_EPOCH}]              # 起始的 epoch，默认为 1
    [--eval-interval ${EVALUATION_INTERVAL}] \  # 评估间隔，默认为 1
    [--title ${TITLE}] \                        # 图片标题，无默认值
    [--legend ${LEGEND}] \                      # 图例，默认为 None
```

(下页继续)

(续上页)

```
    [--backend ${BACKEND}] \                    # 绘制后端，默认为 None
    [--style ${STYLE}] \                        # 绘制风格，默认为 'dark'
    [--out ${OUT_FILE}]                         # 输出文件路径
# [] 代表可选参数，实际输入命令行时，不用输入 []
```

样例：

- 绘制分类损失曲线图

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls \
    --legend loss_cls
```

- 绘制分类损失、回归损失曲线图，保存图片为对应的 pdf 文件

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls loss_bbox \
    --legend loss_cls loss_bbox \
    --out losses_yolov5_s.pdf
```

- 在同一图像中比较两次运行结果的 bbox mAP

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    yolov5_n-v61_syncbn_fast_8xb16-300e_coco_20220919_090739.log.json \
    --keys bbox_mAP \
    --legend yolov5_s yolov5_n \
    --eval-interval 10 # 注意评估间隔必须和训练时设置的一致，否则会报错
```

31.1.2 计算平均训练速度

```
mim run mmdet analyze_logs cal_train_time \
    ${LOG} \                                    # 日志文件路径
    [--include-outliers]                        # 计算时包含每个 epoch 的第一个数据
```

样例：

```
mim run mmdet analyze_logs cal_train_time \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json
```

输出以如下形式展示：

```
-----Analyze train time of yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.
↪log.json-----
slowest epoch 278, average time is 0.1705 s/iter
fastest epoch 300, average time is 0.1510 s/iter
time std over epochs is 0.0026
average iter time: 0.1556 s/iter
```

CHAPTER 32

应用多个 Neck

如果你想堆叠多个 Neck，可以直接在配置文件中的 Neck 参数，MMYOLO 支持以 `List` 形式拼接多个 Neck 配置，你需要保证上一个 Neck 的输出通道与下一个 Neck 的输入通道相匹配。如需要调整通道，可以插入 `mmdet.ChannelMapper` 模块用来对齐多个 Neck 之间的通道数量。具体配置如下：

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

deepen_factor = _base_.deepen_factor
widen_factor = _base_.widen_factor
model = dict(
    type='YOLODetector',
    neck=[
        dict(
            type='YOLOv5PAFPN',
            deepen_factor=deepen_factor,
            widen_factor=widen_factor,
            in_channels=[256, 512, 1024],
            out_channels=[256, 512, 1024],
            # 因为 out_channels 由 widen_factor 控制，YOLOv5PAFPN 的 out_channels = out_
↪channels * widen_factor
            num_csp_blocks=3,
            norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
            act_cfg=dict(type='SiLU', inplace=True)),
        dict(
            type='mmdet.ChannelMapper',
```

(下页继续)

(续上页)

```
            in_channels=[128, 256, 512],
            out_channels=128,
        ),
        dict(
            type='mmdet.DyHead',
            in_channels=128,
            out_channels=256,
            num_blocks=2,
            # disable zero_init_offset to follow official implementation
            zero_init_offset=False)
    ],
    bbox_head=dict(head_module=dict(in_channels=[512, 512, 512]))
    # 因为 out_channels 由 widen_factor 控制，YOLOv5HeadModuled 的 in_channels * widen_
↪factor 才会等于最后一个 neck 的 out_channels
)
```

CHAPTER 33

指定特定设备训练或推理

如果你有多张 GPU，比如 8 张，其编号分别为 0，1，2，3，4，5，6，7，使用单卡训练或推理时会默认使用卡 0。如果想指定其他卡进行训练或推理，可以使用以下命令：

```
CUDA_VISIBLE_DEVICES=5 python ./tools/train.py ${CONFIG} #train
CUDA_VISIBLE_DEVICES=5 python ./tools/test.py ${CONFIG} ${CHECKPOINT_FILE} #test
```

如果设置 CUDA_VISIBLE_DEVICES 为 -1 或者一个大于 GPU 最大编号的数，比如 8，将会使用 CPU 进行训练或者推理。

如果你想使用其中几张卡并行训练，可以使用如下命令：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 ./tools/dist_train.sh ${CONFIG} ${GPU_NUM}
```

这里 GPU_NUM 为 4。另外如果在一台机器上多个任务同时多卡训练，需要设置不同的端口，比如以下命令：

```
CUDA_VISIBLE_DEVICES=0,1,2,3 PORT=29500 ./tools/dist_train.sh ${CONFIG} 4
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 ./tools/dist_train.sh ${CONFIG} 4
```

CHAPTER 34

单通道和多通道应用案例

34.1 在单通道图像数据集上训练示例

MMYOLO 中默认的训练图片均为彩色三通道数据，如果希望采用单通道数据集进行训练和测试，预计需要修改的地方包括：

1. 所有的图片处理 pipeline 都要支持单通道运算
2. 模型的骨干网络的第一个卷积层输入通道需要从 3 改成 1
3. 如果希望加载 COCO 预训练权重，则需要处理第一个卷积层权重尺寸不匹配问题

下面以 cat 数据集为例，描述整个修改过程，如果你使用的是自定义灰度图像数据集，你可以跳过数据集预处理这一步。

34.1.1 1 数据集预处理

自定义数据集的处理训练可参照自定义数据集标注 + 训练 + 测试 + 部署全流程。

cat 是一个三通道彩色图片数据集，为了方便演示，你可以运行下面的代码和命令，将数据集图片替换为单通道图片，方便后续验证。

1. 下载 cat 数据集进行解压

```
python tools/misc/download_dataset.py --dataset-name cat --save-dir ./data/cat --
↪unzip --delete
```

2. 将数据集转换为灰度图

```
import argparse
import imghdr
import os
from typing import List
import cv2

def parse_args():
    parser = argparse.ArgumentParser(description='data_path')
    parser.add_argument('path', type=str, help='Original dataset path')
    return parser.parse_args()

def main():
    args = parse_args()

    path = args.path + '/images/'
    save_path = path
    file_list: List[str] = os.listdir(path)
    # Grayscale conversion of each imager
    for file in file_list:
        if imghdr.what(path + '/' + file) != 'jpeg':
            continue
        img = cv2.imread(path + '/' + file)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        cv2.imwrite(save_path + '/' + file, img)

if __name__ == '__main__':
    main()
```

将上述脚本命名为 `cvt_single_channel.py`, 运行命令为:

```
python cvt_single_channel.py data/cat
```

34.1.2 2 修改 base 配置文件

目前 MMYOLO 的一些图像处理函数例如颜色空间变换还不兼容单通道图片，如果直接采用单通道数据训练需要修改部分 pipeline，工作量较大。为了解决不兼容问题，推荐的做法是将单通道图片作为采用三通道图片方式读取将其加载为三通道数据，但是在输入到网络前将其转换为单通道格式。这种做法会稍微增加一些运算负担，但是用户基本不需要修改代码即可使用。

以 `projects/misc/custom_dataset/yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py` 为 base 配置，将其复制到 `configs/yolov5` 目录下，在同级配置路径下新增 `yolov5_s-v61_syncbn_fast_1xb32-100e_cat_single_channel.py` 文件。我们

可以 mmyolo/models/data_preprocessors/data_preprocessor.py 文件中继承 YOLOv5DetDataPreprocessor 并命名新类为 YOLOv5SCDetDataPreprocessor，在其中将图片转成单通道，添加依赖库并在 mmyolo/models/data_preprocessors/__init__.py 中注册新类。YOLOv5SCDetDataPreprocessor 示例代码为：

```
@MODELS.register_module()
class YOLOv5SCDetDataPreprocessor(YOLOv5DetDataPreprocessor):
    """Rewrite collate_fn to get faster training speed.

    Note: It must be used together with `mmyolo.datasets.utils.yolov5_collate`
    """

    def forward(self, data: dict, training: bool = False) -> dict:
        """Perform normalization, padding, bgr2rgb conversion and convert to single␣
↪channel image based on ``DetDataPreprocessor``.

        Args:
            data (dict): Data sampled from dataloader.
            training (bool): Whether to enable training time augmentation.

        Returns:
            dict: Data in the same format as the model input.
        """
        if not training:
            return super().forward(data, training)

        data = self.cast_data(data)
        inputs, data_samples = data['inputs'], data['data_samples']
        assert isinstance(data['data_samples'], dict)

        # TODO: Supports multi-scale training
        if self._channel_conversion and inputs.shape[1] == 3:
            inputs = inputs[:, [2, 1, 0], ...]

        if self._enable_normalize:
            inputs = (inputs - self.mean) / self.std

        if self.batch_augments is not None:
            for batch_aug in self.batch_augments:
                inputs, data_samples = batch_aug(inputs, data_samples)

        img_metas = [{'batch_input_shape': inputs.shape[2:]}] * len(inputs)
        data_samples = {
            'bboxes_labels': data_samples['bboxes_labels'],
```

(下页继续)

(续上页)

```
            'img_metas': img_metas
        }

        # Convert to single channel image
        inputs = inputs.mean(dim=1, keepdim=True)

        return {'inputs': inputs, 'data_samples': data_samples}
```

此时 yolov5_s-v61_syncbn_fast_1xb32-100e_cat_single_channel.py 配置文件内容为如下所示：

```
_base_ = 'yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py'

_base_.model.data_preprocessor.type = 'YOLOv5SCDetDataPreprocessor'
```

34.1.3 3 预训练模型加载问题

直接使用原三通道的预训练模型，理论上会导致精度有所降低（未实验验证）。可采用的解决思路：将输入层 3 通道每个通道的权重调整为原 3 通道权重的平均值, 或将输入层每个通道的权重调整为原 3 通道某一通道权重，也可以对输入层权重不做修改直接训练，具体效果根据实际情况有所不同。这里采用将输入层 3 个通道权重调整为预训练 3 通道权重平均值的方式。

```
import torch

def main():
    # 加载权重文件
    state_dict = torch.load(
        'checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187.pth'
    )

    # 修改输入层权重
    weights = state_dict['state_dict']['backbone.stem.conv.weight']
    avg_weight = weights.mean(dim=1, keepdim=True)
    state_dict['state_dict']['backbone.stem.conv.weight'] = avg_weight

    # 保存修改后的权重到新文件
    torch.save(
        state_dict,
        'checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187_single_channel.pth'
    )
```

(下页继续)

(续上页)

```
if __name__ == '__main__':
    main()
```

此时 yolov5_s-v61_syncbn_fast_1xb32-100e_cat_single_channel.py 配置文件内容为如下所示:

```
_base_ = 'yolov5_s-v61_syncbn_fast_1xb32-100e_cat.py'

_base_.model.data_preprocessor.type = 'YOLOv5SCDetDataPreprocessor'

load_from = './checkpoints/yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700-
↪86e02187_single_channel.pth'
```

34.1.4 4 模型训练效果

左图是实际标签，右图是目标检测结果。

```
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.958
 Average Precision  (AP) @[ IoU=0.50       | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.75       | area=   all | maxDets=100 ] = 1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.958
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.881
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.969
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.969
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = -1.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.969
bbox_mAP_copypaste: 0.958 1.000 1.000 -1.000 -1.000 0.958
Epoch(val) [100][116/116]  coco/bbox_mAP: 0.9580  coco/bbox_mAP_50: 1.0000  coco/bbox_
↪mAP_75: 1.0000  coco/bbox_mAP_s: -1.0000  coco/bbox_mAP_m: -1.0000  coco/bbox_mAP_
↪l: 0.9580
```

34.2 在多通道图像数据集上训练示例

TODO

CHAPTER 35

MM 系列开源库注册表

（注意：本文档是通过.dev_scripts/print_registers.py 脚本自动生成）

35.1 MMdetection (3.0.0rc6)

35.2 MMclassification (1.0.0rc5)

35.3 MMsegmentation (1.0.0rc5)

35.4 MMengine (0.6.0)

35.5 MMCV (2.0.0rc4)

CHAPTER 36

可视化 COCO 标签

脚本 tools/analysis_tools/browse_coco_json.py 能够使用可视化显示 COCO 标签在图片的情况。

```
python tools/analysis_tools/browse_coco_json.py [--data-root ${DATA_ROOT}] \
                                                 [--img-dir ${IMG_DIR}] \
                                                 [--ann-file ${ANN_FILE}] \
                                                 [--wait-time ${WAIT_TIME}] \
                                                 [--disp-all] [--category-names␣
↪CATEGORY_NAMES [CATEGORY_NAMES ...]] \
                                                 [--shuffle]
```

其中，如果图片、标签都在同一个文件夹下的话，可以指定 --data-root 到该文件夹，然后 --img-dir 和 --ann-file 指定该文件夹的相对路径，代码会自动拼接。如果图片、标签文件不在同一个文件夹下的话，则无需指定 --data-root ，直接指定绝对路径的 --img-dir 和 --ann-file 即可。

例子：

1. 查看 COCO 全部类别，同时展示 bbox、mask 等所有类型的标注：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                 --img-dir 'train2017' \
                                                 --ann-file 'annotations/instances_
↪train2017.json' \
                                                 --disp-all
```

如果图片、标签不在同一个文件夹下的话，可以使用绝对路径：

```
python tools/analysis_tools/browse_coco_json.py --img-dir '/dataset/image/coco/
↪train2017' \
                                                --ann-file '/label/instances_
↪train2017.json' \
                                                --disp-all
```

2. 查看 COCO 全部类别，同时仅展示 bbox 类型的标注，并打乱显示：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --shuffle
```

3. 只查看 bicycle 和 person 类别，同时仅展示 bbox 类型的标注：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --category-names 'bicycle' 'person'
```

4. 查看 COCO 全部类别，同时展示 bbox、mask 等所有类型的标注，并打乱显示：

```
python tools/analysis_tools/browse_coco_json.py --data-root './data/coco' \
                                                --img-dir 'train2017' \
                                                --ann-file 'annotations/instances_
↪train2017.json' \
                                                --disp-all \
                                                --shuffle
```

CHAPTER 37

可视化数据集

```
python tools/analysis_tools/browse_dataset.py \
    ${CONFIG_FILE} \
    [-o, --out-dir ${OUTPUT_DIR}] \
    [-p, --phase ${DATASET_PHASE}] \
    [-n, --show-number ${NUMBER_IMAGES_DISPLAY}] \
    [-i, --show-interval ${SHOW_INTERRVAL}] \
    [-m, --mode ${DISPLAY_MODE}] \
    [--cfg-options ${CFG_OPTIONS}]
```

所有参数的说明:

- `config`: 模型配置文件的路径。
- `-o, --out-dir`: 保存图片文件夹，如果没有指定，默认为 `'./output'`。
- **`-p, --phase`**: 可视化数据集的阶段，只能为 `['train', 'val', 'test']` 之一，默认为 `'train'`。
- **`-n, --show-number`**: 可视化样本数量。如果没有指定，默认展示数据集的所有图片。
- **`-m, --mode`**: 可视化的模式，只能为 `['original', 'transformed', 'pipeline']` 之一。默认为 `'transformed'`。
- `--cfg-options`: 对配置文件的修改，参考学习配置文件。

```
`-m, --mode` 用于设置可视化的模式，默认设置为 'transformed'。
- 如果 `--mode` 设置为 'original'，则获取原始图片；
```

(下页继续)

(续上页)

```
- 如果 `--mode` 设置为 'transformed'，则获取预处理后的图片；
- 如果 `--mode` 设置为 'pipeline'，则获得数据流水线所有中间过程图片。
```

示例：

1. **‘original’**模式：

```
python ./tools/analysis_tools/browse_dataset.py configs/yolov5/yolov5_s-v61_syncbn_
↪fast_1xb4-300e_balloon.py --phase val --out-dir tmp --mode original
```

- `--phase val`: 可视化验证集, 可简化为 `-p val`；
- `--out-dir tmp`: 可视化结果保存在“tmp”文件夹, 可简化为 `-o tmp`；
- `--mode original`: 可视化原图, 可简化为 `-m original`；
- `--show-number 100`: 可视化 100 张图，可简化为 `-n 100`；

2. **‘transformed’**模式：

```
python ./tools/analysis_tools/browse_dataset.py configs/yolov5/yolov5_s-v61_syncbn_
↪fast_1xb4-300e_balloon.py
```

3. **‘pipeline’**模式：

```
python ./tools/analysis_tools/browse_dataset.py configs/yolov5/yolov5_s-v61_syncbn_
↪fast_1xb4-300e_balloon.py -m pipeline
```

CHAPTER 38

打印完整配置文件

MMDetection 中的 tools/misc/print_config.py 脚本可将所有配置继承关系展开，打印相应的完整配置文件。调用命令如下：

```
mim run mmdet print_config \
    ${CONFIG} \                                 # 需要打印的配置文件路径
    [--save-path] \                             # 保存文件路径，必须以 .py, .json 或者 .yml 结尾
    [--cfg-options ${OPTIONS [OPTIONS...]}]   # 通过命令行参数修改配置文件
```

样例：

```
mim run mmdet print_config \
    configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py \
    --save-path ./work_dirs/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon_whole.py
```

运行以上命令，会将 yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py 继承关系展开后的配置文件保存到 ./work_dirs 文件夹内的 yolov5_s-v61_syncbn_fast_1xb4-300e_balloon_whole.py 文件中。

CHAPTER 39

可视化数据集分析结果

脚本 tools/analysis_tools/dataset_analysis.py 能够帮助用户得到四种功能的结果图，并将图片保存到当前运行目录下的 dataset_analysis 文件夹中。

关于该脚本的功能的说明:

通过 main() 的数据准备，得到每个子函数所需要的数据。

功能一：显示类别和 bbox 实例个数的分布图，通过子函数 show_bbox_num 生成。

功能二：显示类别和 bbox 实例宽、高的分布图，通过子函数 show_bbox_wh 生成。

功能三：显示类别和 bbox 实例宽/高比例的分布图，通过子函数 show_bbox_wh_ratio 生成。

功能四：基于面积规则下，显示类别和 bbox 实例面积的分布图，通过子函数 show_bbox_area 生成。

打印列表显示，通过脚本中子函数 show_class_list 和 show_data_list 生成。

```
python tools/analysis_tools/dataset_analysis.py ${CONFIG} \
                                                [-h] \
                                                [--val-dataset ${TYPE}] \
                                                [--class-name ${CLASS_NAME}] \
                                                [--area-rule ${AREA_RULE}] \
                                                [--func ${FUNC}] \
                                                [--out-dir ${OUT_DIR}]
```

例子:

1. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，其中默认设置: 数据加载类型为 train_dataset ，面积规则设置为 [0,32,96,1e5] ,生成包含

所有类的结果图并将图片保存到当前运行目录下 ./dataset_analysis 文件夹中：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py
```

2. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --val-dataset 设置将数据加载类型由默认的 train_dataset 改为 val_dataset：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                            --val-dataset
```

3. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --class-name 设置将生成所有类改为特定类显示，以显示 person 为例：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                            --class-name person
```

4. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --area-rule 重新定义面积规则，以 30 70 125 为例, 面积规则变为 [0,30,70,125,1e5]：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                            --area-rule 30 70 125
```

5. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --func 设置，将显示四个功能效果图改为只显示 功能一为例：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                            --func show_bbox_num
```

6. 使用 config 文件 configs/yolov5/voc/yolov5_s-v61_fast_1xb64-50e_voc.py 分析数据集，通过 --out-dir 设置修改图片保存地址，以 work_dirs/dataset_analysis 地址为例：

```
python tools/analysis_tools/dataset_analysis.py configs/yolov5/voc/yolov5_s-v61_fast_
↪1xb64-50e_voc.py \
                                            --out-dir work_dirs/dataset_analysis
```

CHAPTER 40

优化锚框尺寸

脚本 tools/analysis_tools/optimize_anchors.py 支持 YOLO 系列中三种锚框生成方式，分别是 k-means、Differential Evolution、v5-k-means.

40.1 k-means

在 k-means 方法中，使用的是基于 IoU 表示距离的聚类方法，具体使用命令如下:

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} \
                                                --algorithm k-means \
                                                --input-shape ${INPUT_SHAPE [WIDTH␣
↪HEIGHT]} \
                                                --out-dir ${OUT_DIR}
```

40.2 Differential Evolution

在 Differential Evolution 方法中，使用的是基于差分进化算法（简称 DE 算法）的聚类方式，其最小化目标函数为 avg_iou_cost，具体使用命令如下:

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} \
                                                --algorithm DE \
```

(下页继续)

(续上页)

```
                                                --input-shape ${INPUT_SHAPE [WIDTH␣
↪HEIGHT]} \
                                                --out-dir ${OUT_DIR}
```

40.3 v5-k-means

在 v5-k-means 方法中，使用的是 YOLOv5 中基于 `shape-match` 的聚类方式，具体使用命令如下:

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} \
                                                --algorithm v5-k-means \
                                                --input-shape ${INPUT_SHAPE [WIDTH␣
↪HEIGHT]} \
                                                --prior-match-thr ${PRIOR_MATCH_THR} \
                                                --out-dir ${OUT_DIR}
```

CHAPTER 41

提取 COCO 子集

COCO2017 数据集训练数据集包括 118K 张图片，验证集包括 5K 张图片，数据集比较大。在调试或者快速验证程序是否正确的场景下加载 json 会需要消耗较多资源和带来较慢的启动速度，这会导致程序体验不好。

`extract_subcoco.py` 脚本提供了按指定图片数量、类别、锚框尺寸来切分图片的功能，用户可以通过 `--num-img`, `--classes`, `--area-size` 参数来得到指定条件的 COCO 子集，从而满足上述需求。

例如通过以下脚本切分图片：

```
python tools/misc/extract_subcoco.py \
    ${ROOT} \
    ${OUT_DIR} \
    --num-img 20 \
    --classes cat dog person \
    --area-size small
```

会切分出 20 张图片，且这 20 张图片只会保留同时满足类别条件和锚框尺寸条件的标注信息, 没有满足条件的标注信息的图片不会被选择，保证了这 20 张图都是有 annotation info 的。

注意：本脚本目前仅仅支持 COCO2017 数据集，未来会支持更加通用的 COCO JSON 格式数据集

输入 root 根路径文件夹格式如下所示：

```
├── root
|   ├── annotations
|   ├── train2017
|   ├── val2017
|   ├── test2017
```

1. 仅仅使用 5K 张验证集切分出 10 张训练图片和 10 张验证图片

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --num-img 10
```

2. 使用训练集切分出 20 张训练图片，使用验证集切分出 20 张验证图片

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --num-img 20 --use-training-
↪set
```

3. 设置全局种子，默认不设置

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --num-img 20 --use-training-
↪set --seed 1
```

4. 按指定类别切分图片

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --classes cat dog person
```

5. 按指定锚框尺寸切分图片

```
python tools/misc/extract_subcoco.py ${ROOT} ${OUT_DIR} --area-size small
```

CHAPTER 42

可视化优化器参数策略

`tools/analysis_tools/vis_scheduler.py` 旨在帮助用户检查优化器的超参数调度器（无需训练），支持学习率（learning rate）、动量（momentum）和权值衰减（weight decay）。

```
python tools/analysis_tools/vis_scheduler.py \
    ${CONFIG_FILE} \
    [-p, --parameter ${PARAMETER_NAME}] \
    [-d, --dataset-size ${DATASET_SIZE}] \
    [-n, --ngpus ${NUM_GPUs}] \
    [-o, --out-dir ${OUT_DIR}] \
    [--title ${TITLE}] \
    [--style ${STYLE}] \
    [--window-size ${WINDOW_SIZE}] \
    [--cfg-options]
```

所有参数的说明:

- `config` : 模型配置文件的路径。
- **`-p, parameter`**: 可视化参数名，只能为 `["lr", "momentum", "wd"]` 之一，默认为 `"lr"`.
- **`-d, --dataset-size`**: 数据集的大小。如果指定，`DATASETS.build` 将被跳过并使用这个数值作为数据集大小，默认使用 `DATASETS.build` 所得数据集的大小。
- **`-n, --ngpus`**: 使用 GPU 的数量, 默认为 1。
- **`-o, --out-dir`**: 保存的可视化图片的文件夹路径，默认不保存。
- `--title`: 可视化图片的标题，默认为配置文件名。

- --style: 可视化图片的风格，默认为 whitegrid。
- --window-size: 可视化窗口大小，如果没有指定，默认为 12*7。如果需要指定，按照格式 'W*H'。
- --cfg-options: 对配置文件的修改，参考学习配置文件。

注解： 部分数据集在解析标注阶段比较耗时，推荐直接将 -d, dataset-size 指定数据集的大小，以节约时间。

你可以使用如下命令来绘制配置文件 configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_coco.py 将会使用的学习率变化曲线：

```
python tools/analysis_tools/vis_scheduler.py \
    configs/rtmdet/rtmdet_s_syncbn_fast_8xb32-300e_coco.py \
    --dataset-size 118287 \
    --ngpus 8 \
    --out-dir ./output
```

CHAPTER 43

数据集转换

文件夹 tools/data_converters/ 目前包含 ballon2coco.py、yolo2coco.py 和 labelme2coco.py 三个数据集转换工具。

- ballon2coco.py 将 balloon 数据集（该小型数据集仅作为入门使用）转换成 COCO 的格式。

```
python tools/dataset_converters/balloon2coco.py
```

- yolo2coco.py 将 yolo-style **.txt** 格式的数据集转换成 COCO 的格式，请按如下方式使用：

```
python tools/dataset_converters/yolo2coco.py /path/to/the/root/dir/of/your_dataset
```

使用说明：

1. image_dir 是需要你传入的待转换的 yolo 格式数据集的根目录，内应包含 images 、labels 和 classes.txt 文件，classes.txt 是当前 dataset 对应的类的声明，一行一个类别。image_dir 结构如下例所示：

```
.
└── $ROOT_PATH
    ├── classes.txt
    ├── labels
    │    ├── a.txt
    │    ├── b.txt
    │    └── ...
    ├── images
    │    ├── a.jpg
```

(下页继续)

(续上页)

```
    |    ├── b.png
    |    └── ...
    └── ...
```

2. 脚本会检测 image_dir 下是否已有的 train.txt 、val.txt 和 test.txt 。若检测到文件，则会按照类别进行整理，否则默认不需要分类。故请确保对应的 train.txt 、val.txt 和 test.txt 要在 image_dir 内。文件内的图片路径必须是**绝对路径**。

3. 脚本会默认在 image_dir 目录下创建 annotations 文件夹并将转换结果存在这里。如果在 image_dir 下没找到分类文件，输出文件即为一个 result.json，反之则会生成需要的 train.json 、val.json、test.json，脚本完成后 annotations 结构可如下例所示：

```
.
└── $ROOT_PATH
    ├── annotations
    |    ├── result.json
    |    └── ...
    ├── classes.txt
    ├── labels
    |    ├── a.txt
    |    ├── b.txt
    |    └── ...
    ├── images
    |    ├── a.jpg
    |    ├── b.png
    |    └── ...
    └── ...
```

CHAPTER 44

数据集下载

脚本 tools/misc/download_dataset.py 支持下载数据集，例如 COCO、VOC、LVIS 和 Balloon.

```
python tools/misc/download_dataset.py --dataset-name coco2017
python tools/misc/download_dataset.py --dataset-name voc2007
python tools/misc/download_dataset.py --dataset-name voc2012
python tools/misc/download_dataset.py --dataset-name lvis
python tools/misc/download_dataset.py --dataset-name balloon [--save-dir ${SAVE_DIR}]␣
↪[--unzip]
```

CHAPTER 45

日志分析

45.1 曲线图绘制

MMDetection 中的 tools/analysis_tools/analyze_logs.py 可利用指定的训练 log 文件绘制 loss/mAP 曲线图，第一次运行前请先运行 pip install seaborn 安装必要依赖。

```
mim run mmdet analyze_logs plot_curve \
    ${LOG} \                                    # 日志文件路径
    [--keys ${KEYS}] \                          # 需要绘制的指标，默认为 'bbox_mAP'
    [--start-epoch ${START_EPOCH}]              # 起始的 epoch，默认为 1
    [--eval-interval ${EVALUATION_INTERVAL}] \  # 评估间隔，默认为 1
    [--title ${TITLE}] \                        # 图片标题，无默认值
    [--legend ${LEGEND}] \                      # 图例，默认为 None
    [--backend ${BACKEND}] \                    # 绘制后端，默认为 None
    [--style ${STYLE}] \                        # 绘制风格，默认为 'dark'
    [--out ${OUT_FILE}]                         # 输出文件路径
# [] 代表可选参数，实际输入命令行时，不用输入 []
```

样例：

- 绘制分类损失曲线图

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls \
    --legend loss_cls
```

- 绘制分类损失、回归损失曲线图，保存图片为对应的 pdf 文件

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    --keys loss_cls loss_bbox \
    --legend loss_cls loss_bbox \
    --out losses_yolov5_s.pdf
```

- 在同一图像中比较两次运行结果的 bbox mAP

```
mim run mmdet analyze_logs plot_curve \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json \
    yolov5_n-v61_syncbn_fast_8xb16-300e_coco_20220919_090739.log.json \
    --keys bbox_mAP \
    --legend yolov5_s yolov5_n \
    --eval-interval 10 # 注意评估间隔必须和训练时设置的一致，否则会报错
```

45.2 计算平均训练速度

```
mim run mmdet analyze_logs cal_train_time \
    ${LOG} \                                    # 日志文件路径
    [--include-outliers]                        # 计算时包含每个 epoch 的第一个数据
```

样例：

```
mim run mmdet analyze_logs cal_train_time \
    yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.log.json
```

输出以如下形式展示：

```
-----Analyze train time of yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_084700.
↪log.json-----
slowest epoch 278, average time is 0.1705 s/iter
fastest epoch 300, average time is 0.1510 s/iter
time std over epochs is 0.0026
average iter time: 0.1556 s/iter
```

CHAPTER 46

模型转换

文件夹 tools/model_converters/ 下的六个脚本能够帮助用户将对应 YOLO 官方的预训练模型中的键转换成 MMYOLO 格式，并使用 MMYOLO 对模型进行微调。

46.1 YOLOv5

下面以转换 yolov5s.pt 为例：

1. 将 YOLOv5 官方代码克隆到本地（目前支持的最高版本为 v6.1 ）：

```
git clone -b v6.1 https://github.com/ultralytics/yolov5.git
cd yolov5
```

2. 下载官方权重：

```
wget https://github.com/ultralytics/yolov5/releases/download/v6.1/yolov5s.pt
```

3. 将 tools/model_converters/yolov5_to_mmyolo.py 文件复制到 YOLOv5 官方代码克隆的路径：

```
cp ${MMDET_YOLO_PATH}/tools/model_converters/yolov5_to_mmyolo.py yolov5_to_mmyolo.py
```

4. 执行转换：

```
python yolov5_to_mmyolo.py --src ${WEIGHT_FILE_PATH} --dst mmyolov5.pt
```

转换好的 mmyolov5.pt 即可以为 MMYOLO 所用。YOLOv6 官方权重转化也是采用一样的使用方式。

46.2 YOLOX

YOLOX 模型的转换不需要下载 YOLOX 官方代码，只需要下载权重即可。下面以转换 yolox_s.pth 为例：

1. 下载权重：

```
wget https://github.com/Megvii-BaseDetection/YOLOX/releases/download/0.1.1rc0/yolox_s.
↪pth
```

2. 执行转换：

```
python tools/model_converters/yolox_to_mmyolo.py --src yolox_s.pth --dst mmyolox.pt
```

转换好的 mmyolox.pt 即可以在 MMYOLO 中使用。

CHAPTER 47

学习 YOLOv5 配置文件

MMYOLO 和其他 OpenMMLab 仓库使用 MMEngine 的配置文件系统。配置文件使用了模块化和继承设计，以便于进行各类实验。

47.1 配置文件的内容

MMYOLO 采用模块化设计，所有功能的模块都可以通过配置文件进行配置。以 yolov5_s-v61_syncbn_8xb16-300e_coco.py 为例，我们将根据不同的功能模块介绍配置文件中的各个字段：

47.1.1 重要参数

如下参数是修改训练配置时经常需要修改的参数。例如缩放因子 `deepen_factor` 和 `widen_factor`，MMYOLO 中的网络基本都使用它们来控制模型的大小。所以我们推荐在配置文件中单独定义这些参数。

```
img_scale = (640, 640)              # 高度，宽度
deepen_factor = 0.33                # 控制网络结构深度的缩放因子，YOLOv5-s 为 0.33
widen_factor = 0.5                  # 控制网络结构宽度的缩放因子，YOLOv5-s 为 0.5
max_epochs = 300                    # 最大训练轮次 300 轮
save_epoch_intervals = 10           # 验证间隔，每 10 个 epoch 验证一次
train_batch_size_per_gpu = 16       # 训练时单个 GPU 的 Batch size
train_num_workers = 8               # 训练时单个 GPU 分配的数据加载线程数
val_batch_size_per_gpu = 1          # 验证时单个 GPU 的 Batch size
val_num_workers = 2                 # 验证时单个 GPU 分配的数据加载线程数
```

47.1.2 模型配置

在 MMYOLO 的配置中，我们使用 model 字段来配置检测算法的组件。除了 backbone、neck 等神经网络组件外，还需要 data_preprocessor、train_cfg 和 test_cfg。data_preprocessor 负责对 dataloader 输出的每一批数据进行预处理。模型配置中的 train_cfg 和 test_cfg 用于设置训练和测试组件的超参数。

```
anchors = [[(10, 13), (16, 30), (33, 23)], # 多尺度的先验框基本尺寸
           [(30, 61), (62, 45), (59, 119)],
           [(116, 90), (156, 198), (373, 326)]]
strides = [8, 16, 32] # 先验框生成器的步幅

model = dict(
    type='YOLODetector', # 检测器名
    data_preprocessor=dict(  # 数据预处理器的配置，通常包括图像归一化和 padding
        type='mmdet.DetDataPreprocessor',  # 数据预处理器的类型，还可以选择
↪'YOLOv5DetDataPreprocessor' 训练速度更快
        mean=[0., 0., 0.],  # 用于预训练骨干网络的图像归一化通道均值，按 R、G、B 排序
        std=[255., 255., 255.], # 用于预训练骨干网络的图像归一化通道标准差，按 R、G、B 排序
        bgr_to_rgb=True),  # 是否将图像通道从 BGR 转为 RGB
    backbone=dict(  # 主干网络的配置文件
        type='YOLOv5CSPDarknet',  # 主干网络的类别，目前可选用 'YOLOv5CSPDarknet',
↪'YOLOv6EfficientRep', 'YOLOXCSPDarknet' 3 种
        deepen_factor=deepen_factor, # 控制网络结构深度的缩放因子
        widen_factor=widen_factor, # 控制网络结构宽度的缩放因子
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001), # 归一化层 (norm layer) 的配
置项
        act_cfg=dict(type='SiLU', inplace=True)), # 激活函数 (activation function) 的配置
项
    neck=dict(
        type='YOLOv5PAFPN',  # 检测器的 neck 是 YOLOv5FPN，我们同样支持 'YOLOv6RepPAFPN',
↪'YOLOXPAFPN'
        deepen_factor=deepen_factor, # 控制网络结构深度的缩放因子
        widen_factor=widen_factor, # 控制网络结构宽度的缩放因子
        in_channels=[256, 512, 1024], # 输入通道数，与 Backbone 的输出通道一致
        out_channels=[256, 512, 1024], # 输出通道数，与 Head 的输入通道一致
        num_csp_blocks=3, # CSPLayer 中 bottlenecks 的数量
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001), # 归一化层 (norm layer) 的配
置项
        act_cfg=dict(type='SiLU', inplace=True)), # 激活函数 (activation function) 的配置
项
    bbox_head=dict(
        type='YOLOv5Head', # bbox_head 的类型是 'YOLOv5Head', 我们目前也支持 'YOLOv6Head',
↪ 'YOLOXHead'
```

(下页继续)

(续上页)

```
        head_module=dict(
            type='YOLOv5HeadModule', # head_module 的类型是 'YOLOv5HeadModule', 我们目前
也支持 'YOLOv6HeadModule', 'YOLOXHeadModule'
            num_classes=80, # 分类的类别数量
            in_channels=[256, 512, 1024], # 输入通道数，与 Neck 的输出通道一致
            widen_factor=widen_factor, # 控制网络结构宽度的缩放因子
            featmap_strides=[8, 16, 32], # 多尺度特征图的步幅
            num_base_priors=3), # 在一个点上，先验框的数量
        prior_generator=dict( # 先验框 (prior) 生成器的配置
            type='mmdet.YOLOAnchorGenerator', # 先验框生成器的类型是 mmdet 中的
↪'YOLOAnchorGenerator'
            base_sizes=anchors, # 多尺度的先验框基本尺寸
            strides=strides), # 先验框生成器的步幅，与 FPN 特征步幅一致。如果未设置 base_
↪sizes，则当前步幅值将被视为 base_sizes。
    ),
    test_cfg=dict(
        multi_label=True, # 对于多类别预测来说是否考虑多标签，默认设置为 True
        nms_pre=30000,  # NMS 前保留的最大检测框数目
        score_thr=0.001, # 过滤类别的分值，低于 score_thr 的检测框当做背景处理
        nms=dict(type='nms', # NMS 的类型
                 iou_threshold=0.65), # NMS 的阈值
        max_per_img=300)) # 每张图像 NMS 后保留的最大检测框数目
```

47.1.3 数据集和评测器配置

在使用 执行器 进行训练、测试、验证时，我们需要配置 Dataloader 。构建数据 dataloader 需要设置数据集（dataset）和数据处理流程（data pipeline）。由于这部分的配置较为复杂，我们使用中间变量来简化 dataloader 配置的编写。由于 MMYOLO 中各类轻量目标检测算法使用了更加复杂的数据增强方法，因此会比 MMDetection 中的其他模型拥有更多样的数据集配置。

YOLOv5 的训练与测试的数据流存在一定差异，这里我们分别进行介绍。

```
dataset_type = 'CocoDataset'  # 数据集类型，这将被用来定义数据集
data_root = 'data/coco/'  # 数据的根路径

pre_transform = [ # 训练数据读取流程
    dict(
        type='LoadImageFromFile'), # 第 1 个流程，从文件路径里加载图像
    dict(type='LoadAnnotations', # 第 2 个流程，对于当前图像，加载它的注释信息
         with_bbox=True) # 是否使用标注框 (bounding box)，目标检测需要设置为 True
]

albu_train_transforms = [                # YOLOv5-v6.1 仓库中，引入了 Albumentation 代码库
进行图像的数据增广，请确保其版本为 1.0.+
```

(下页继续)

(续上页)

```
    dict(type='Blur', p=0.01),        # 图像模糊，模糊概率 0.01
    dict(type='MedianBlur', p=0.01), # 均值模糊，模糊概率 0.01
    dict(type='ToGray', p=0.01),          # 随机转换为灰度图像，转灰度概率 0.01
    dict(type='CLAHE', p=0.01)                  # CLAHE(限制对比度自适应直方图均衡化) 图像增
强方法，直方图均衡化概率 0.01
]
train_pipeline = [                                  # 训练数据处理流程
    *pre_transform,                                  # 引入前述定义的训练数据读取流程
    dict(
        type='Mosaic',           # Mosaic 数据增强方法
        img_scale=img_scale,     # Mosaic 数据增强后的图像尺寸
        pad_val=114.0,           # 空区域填充像素值
        pre_transform=pre_transform), # 之前创建的 pre_transform 训练数据读取流程
    dict(
        type='YOLOv5RandomAffine',             # YOLOv5 的随机仿射变换
        max_rotate_degree=0.0,           # 最大旋转角度
        max_shear_degree=0.0,            # 最大错切角度
        scaling_ratio_range=(0.5, 1.5), # 图像缩放系数的范围
        border=(-img_scale[0] // 2, -img_scale[1] // 2), # 从输入图像的高度和宽度两侧调整输
出形状的距离
        border_val=(114, 114, 114)), # 边界区域填充像素值
    dict(
        type='mmdet.Albu',                          # mmdet 中的 Albumentation 数据增强
        transforms=albu_train_transforms, # 之前创建的 albu_train_transforms 数据增强流程
        bbox_params=dict(
            type='BboxParams',
            format='pascal_voc',
            label_fields=['gt_bboxes_labels', 'gt_ignore_flags']),
        keymap={
            'img': 'image',
            'gt_bboxes': 'bboxes'
        }),
    dict(type='YOLOv5HSVRandomAug'),             # HSV 通道随机增强
    dict(type='mmdet.RandomFlip', prob=0.5),         # 随机翻转，翻转概率 0.5
    dict(
        type='mmdet.PackDetInputs',                                  # 将数据转换为检测器输
入格式的流程
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'flip',
                   'flip_direction'))
]
train_dataloader = dict( # 训练 dataloader 配置
    batch_size=train_batch_size_per_gpu, # 训练时单个 GPU 的 Batch size
    num_workers=train_num_workers, # 训练时单个 GPU 分配的数据加载线程数
```

(下页继续)

(续上页)

```
    persistent_workers=True, # 如果设置为 True，dataloader 在迭代完一轮之后不会关闭数据读取的子
进程，可以加速训练
    pin_memory=True, # 开启锁页内存，节省 CPU 内存拷贝时间
    sampler=dict( # 训练数据的采样器
        type='DefaultSampler', # 默认的采样器，同时支持分布式和非分布式训练。请参考 https://
↪github.com/open-mmlab/mmengine/blob/main/mmengine/dataset/sampler.py
        shuffle=True), # 随机打乱每个轮次训练数据的顺序
    dataset=dict( # 训练数据集的配置
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_train2017.json', # 标注文件路径
        data_prefix=dict(img='train2017/'), # 图像路径前缀
        filter_cfg=dict(filter_empty_gt=False, min_size=32), # 图像和标注的过滤配置
        pipeline=train_pipeline)) # 这是由之前创建的 train_pipeline 定义的数据处理流程
```

YOLOv5 测试阶段采用 Letter Resize 的方法来将所有的测试图像统一到相同尺度，进而有效保留了图像的长宽比。因此我们在验证和评测时，都采用相同的数据流进行推理。

```
test_pipeline = [ # 测试数据处理流程
    dict(
        type='LoadImageFromFile'), # 第 1 个流程，从文件路径里加载图像
    dict(type='YOLOv5KeepRatioResize', # 第 2 个流程，保持长宽比的图像大小缩放
         scale=img_scale), # 图像缩放的目标尺寸
    dict(
        type='LetterResize', # 第 3 个流程，满足多种步幅要求的图像大小缩放
        scale=img_scale, # 图像缩放的目标尺寸
        allow_scale_up=False, # 当 ratio > 1 时，是否允许放大图像，
        pad_val=dict(img=114)), # 空区域填充像素值
    dict(type='LoadAnnotations', with_bbox=True), # 第 4 个流程，对于当前图像，加载它的注释信
息
    dict(
        type='mmdet.PackDetInputs', # 将数据转换为检测器输入格式的流程
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

val_dataloader = dict(
    batch_size=val_batch_size_per_gpu, # 验证时单个 GPU 的 Batch size
    num_workers=val_num_workers, # 验证时单个 GPU 分配的数据加载线程数
    persistent_workers=True, # 如果设置为 True，dataloader 在迭代完一轮之后不会关闭数据读取的子
进程，可以加速训练
    pin_memory=True, # 开启锁页内存，节省 CPU 内存拷贝时间
    drop_last=False, # 是否丢弃最后未能组成一个批次的数据
```

(下页继续)

(续上页)

```
    sampler=dict(
        type='DefaultSampler', # 默认的采样器，同时支持分布式和非分布式训练
        shuffle=False), # 验证和测试时不打乱数据顺序
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        test_mode=True, # 开启测试模式，避免数据集过滤图像和标注
        data_prefix=dict(img='val2017/'), # 图像路径前缀
        ann_file='annotations/instances_val2017.json', # 标注文件路径
        pipeline=test_pipeline, # 这是由之前创建的 test_pipeline 定义的数据处理流程
        batch_shapes_cfg=dict(  # batch shapes 配置
            type='BatchShapePolicy', # 确保在 batch 推理过程中同一个 batch 内的图像 pad 像素最少，不要求整个验证过程中所有 batch 的图像尺度一样
            batch_size=val_batch_size_per_gpu, # batch shapes 策略的 batch size，等于验证时单个 GPU 的 Batch size
            img_size=img_scale[0], # 图像的尺寸
            size_divisor=32, # padding 后的图像的大小应该可以被 pad_size_divisor 整除
            extra_pad_ratio=0.5))) # 额外需要 pad 的像素比例

test_dataloader = val_dataloader
```

评测器 用于计算训练模型在验证和测试数据集上的指标。评测器的配置由一个或一组评价指标（Metric）配置组成：

```
val_evaluator = dict(  # 验证过程使用的评测器
    type='mmdet.CocoMetric',  # 用于评估检测的 AR、AP 和 mAP 的 coco 评价指标
    proposal_nums=(100, 1, 10),         # 用于评估检测任务时，选取的 Proposal 数量
    ann_file=data_root + 'annotations/instances_val2017.json',  # 标注文件路径
    metric='bbox',  # 需要计算的评价指标，`bbox` 用于检测
)
test_evaluator = val_evaluator  # 测试过程使用的评测器
```

由于测试数据集没有标注文件，因此 MMYOLO 中的 `test_dataloader` 和 `test_evaluator` 配置通常等于 `val`。如果要保存在测试数据集上的检测结果，则可以像这样编写配置：

```
# 在测试集上推理，
# 并将检测结果转换格式以用于提交结果
test_dataloader = dict(
    batch_size=1,
    num_workers=2,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
```

(下页继续)

(续上页)

```
        type=dataset_type,
        data_root=data_root,
        ann_file=data_root + 'annotations/image_info_test-dev2017.json',
        data_prefix=dict(img='test2017/'),
        test_mode=True,
        pipeline=test_pipeline))
test_evaluator = dict(
    type='mmdet.CocoMetric',
    ann_file=data_root + 'annotations/image_info_test-dev2017.json',
    metric='bbox',
    format_only=True,  # 只将模型输出转换为 coco 的 JSON 格式并保存
    outfile_prefix='./work_dirs/coco_detection/test')  # 要保存的 JSON 文件的前缀
```

47.1.4 训练和测试的配置

MMEngine 的 Runner 使用 Loop 来控制训练，验证和测试过程。用户可以使用这些字段设置最大训练轮次和验证间隔。

```
max_epochs = 300 # 最大训练轮次 300 轮
save_epoch_intervals = 10 # 验证间隔，每 10 轮验证一次

train_cfg = dict(
    type='EpochBasedTrainLoop',  # 训练循环的类型，请参考 https://github.com/open-mmlab/
↪mmengine/blob/main/mmengine/runner/loops.py
    max_epochs=max_epochs,  # 最大训练轮次 300 轮
    val_interval=save_epoch_intervals)  # 验证间隔，每 10 个 epoch 验证一次
val_cfg = dict(type='ValLoop')  # 验证循环的类型
test_cfg = dict(type='TestLoop')  # 测试循环的类型
```

MMEngine 也支持动态评估间隔，例如你可以在前面 280 epoch 训练阶段中，每间隔 10 个 epoch 验证一次，到最后 20 epoch 训练中每隔 1 个 epoch 验证一次，则配置写法为：

```
max_epochs = 300 # 最大训练轮次 300 轮
save_epoch_intervals = 10 # 验证间隔，每 10 轮验证一次

train_cfg = dict(
    type='EpochBasedTrainLoop',  # 训练循环的类型，请参考 https://github.com/open-mmlab/
↪mmengine/blob/main/mmengine/runner/loops.py
    max_epochs=max_epochs,  # 最大训练轮次 300 轮
    val_interval=save_epoch_intervals,  # 验证间隔，每 10 个 epoch 验证一次
    dynamic_intervals=[(280, 1)]) # 到 280 epoch 开始切换为间隔 1 的评估方式
val_cfg = dict(type='ValLoop')  # 验证循环的类型
test_cfg = dict(type='TestLoop')  # 测试循环的类型
```

47.1.5 优化相关配置

optim_wrapper 是配置优化相关设置的字段。优化器封装（OptimWrapper）不仅提供了优化器的功能，还支持梯度裁剪、混合精度训练等功能。更多内容请看优化器封装教程.

```
optim_wrapper = dict(  # 优化器封装的配置
    type='OptimWrapper',  # 优化器封装的类型。可以切换至 AmpOptimWrapper 来启用混合精度训练
    optimizer=dict(  # 优化器配置。支持 PyTorch 的各种优化器。请参考 https://pytorch.org/
↪docs/stable/optim.html#algorithms
        type='SGD',  # 随机梯度下降优化器
        lr=0.01,  # 基础学习率
        momentum=0.937, # 带动量的随机梯度下降
        weight_decay=0.0005, # 权重衰减
        nesterov=True, # 开启 Nesterov momentum，公式详见 http://www.cs.toronto.edu/~
↪hinton/absps/momentum.pdf
        batch_size_per_gpu=train_batch_size_per_gpu),  # 该选项实现了自动权重衰减系数缩放
    clip_grad=None,  # 梯度裁剪的配置，设置为 None 关闭梯度裁剪。使用方法请见 https://mmengine.
↪readthedocs.io/zh_CN/latest/tutorials/optim_wrapper.html
    constructor='YOLOv5OptimizerConstructor') # YOLOv5 优化器构建器
```

param_scheduler 字段用于配置参数调度器（Parameter Scheduler）来调整优化器的超参数（例如学习率和动量）。用户可以组合多个调度器来创建所需的参数调整策略。在参数调度器教程 和参数调度器 API 文档中查找更多信息。在 YOLOv5 中，参数调度实现比较复杂，难以通过 param_scheduler 实现。所以我们采用了 YOLOv5ParamSchedulerHook 来实现（见下节），这样做更简单但是通用性较差。

```
param_scheduler = None
```

47.1.6 钩子配置

用户可以在训练、验证和测试循环上添加钩子，以便在运行期间插入一些操作。配置中有两种不同的钩子字段，一种是 default_hooks，另一种是 custom_hooks。

default_hooks 是一个字典，用于配置运行时必须使用的钩子。这些钩子具有默认优先级，如果未设置，runner 将使用默认值。如果要禁用默认钩子，用户可以将其配置设置为 None。

```
default_hooks = dict(
    param_scheduler=dict(
        type='YOLOv5ParamSchedulerHook', # MMYOLO 中默认采用 Hook 方式进行优化器超参数的调节
        scheduler_type='linear',
        lr_factor=0.01,
        max_epochs=max_epochs),
    checkpoint=dict(
        type='CheckpointHook', # 按照给定间隔保存模型的权重的 Hook
```

(下页继续)

(续上页)

```
        interval=save_epoch_intervals, # 每 10 轮保存 1 次权重文件
        max_keep_ckpts=3)) # 最多保存 3 个权重文件
```

custom_hooks 是一个列表。用户可以在这个字段中加入自定义的钩子，例如 EMAHook。

```
custom_hooks = [
    dict(
        type='EMAHook', # 实现权重 EMA(指数移动平均) 更新的 Hook
        ema_type='ExpMomentumEMA', # YOLO 中使用的带动量 EMA
        momentum=0.0001, # EMA 的动量参数
        update_buffers=True, # 是否计算模型的参数和缓冲的 running averages
        priority=49) # 优先级略高于 NORMAL(50)
]
```

47.1.7 运行相关配置

```
default_scope = 'mmyolo'  # 默认的注册器域名，默认从此注册器域中寻找模块。请参考 https://
↪mmengine.readthedocs.io/zh_CN/latest/tutorials/registry.html

env_cfg = dict(
    cudnn_benchmark=True,  # 是否启用 cudnn benchmark，推荐单尺度训练时开启，可加速训练
    mp_cfg=dict(  # 多进程设置
        mp_start_method='fork',  # 使用 fork 来启动多进程。'fork' 通常比 'spawn' 更快，但可
能存在隐患。请参考 https://github.com/pytorch/pytorch/issues/1355
        opencv_num_threads=0),  # 关闭 opencv 的多线程以避免系统超负荷
    dist_cfg=dict(backend='nccl'),  # 分布式相关设置
)

vis_backends = [dict(type='LocalVisBackend')]  # 可视化后端，请参考 https://mmengine.
↪readthedocs.io/zh_CN/latest/advanced_tutorials/visualization.html
visualizer = dict(
    type='mmdet.DetLocalVisualizer', vis_backends=vis_backends, name='visualizer')
log_processor = dict(
    type='LogProcessor',  # 日志处理器用于处理运行时日志
    window_size=50,  # 日志数值的平滑窗口
    by_epoch=True)  # 是否使用 epoch 格式的日志。需要与训练循环的类型保存一致。

log_level = 'INFO'  # 日志等级
load_from = None  # 从给定路径加载模型检查点作为预训练模型。这不会恢复训练。
resume = False  # 是否从 `load_from` 中定义的检查点恢复。 如果 `load_from` 为 None，它将恢复␣
↪`work_dir` 中的最新检查点。
```

47.2 配置文件继承

在 config/_base_ 文件夹目前有运行时的默认设置 (default runtime)。由 _base_ 下的组件组成的配置，被我们称为 *原始配置 (primitive)*。

对于同一文件夹下的所有配置，推荐**只有一个**对应的**原始配置**文件。所有其他的配置文件都应该继承自这个**原始配置**文件。这样就能保证配置文件的最大继承深度为 3。

为了便于理解，我们建议贡献者继承现有方法。例如，如果在 YOLOv5s 的基础上做了一些修改，比如修改网络深度，用户首先可以通过指定 _base_ = ./yolov5_s-v61_syncbn_8xb16-300e_coco.py 来集成基础的 YOLOv5 结构，然后修改配置文件中的必要参数以完成继承。

如果你在构建一个与任何现有方法不共享结构的全新方法，那么可以在 configs 文件夹下创建一个新的例如 yolov100 文件夹。

更多细节请参考 MMEngine 配置文件教程。

通过设置 _base_ 字段，我们可以设置当前配置文件继承自哪些文件。

当 _base_ 为文件路径字符串时，表示继承一个配置文件的内容。

```
_base_ = '../_base_/default_runtime.py'
```

当 _base_ 是多个文件路径的列表时，表示继承多个文件。

```
_base_ = [
    './yolov5_s-v61_syncbn_8xb16-300e_coco.py',
    '../_base_/default_runtime.py'
]
```

如果需要检查配置文件，可以通过运行 mim run mmdet print_config /PATH/TO/CONFIG 来查看完整的配置。

47.2.1 忽略基础配置文件里的部分内容

有时，您也许会设置 _delete_=True 去忽略基础配置文件里的一些域内容。您也许可以参照 MMEngine 配置文件教程 来获得一些简单的指导。

在 MMYOLO 里，例如为了改变 RTMDet 的主干网络的某些内容：

```
model = dict(
    type='YOLODetector',
    data_preprocessor=dict(...),
    backbone=dict(
        type='CSPNeXt',
        arch='P5',
```

(下页继续)

(续上页)

```
        expand_ratio=0.5,
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        channel_attention=True,
        norm_cfg=dict(type='BN'),
        act_cfg=dict(type='SiLU', inplace=True)),
    neck=dict(...),
    bbox_head=dict(...))
```

如果想把 RTMDet 主干网络的 CSPNeXt 改成 YOLOv6EfficientRep，因为 CSPNeXt 和 YOLOv6EfficientRep 中有不同的字段 (channel_attention 和 expand_ratio)，这时候就需要使用 _delete_=True 将新的键去替换 backbone 域内所有老的键。

```
_base_ = '../rtmdet/rtmdet_l_syncbn_8xb32-300e_coco.py'
model = dict(
    backbone=dict(
        _delete_=True,
        type='YOLOv6EfficientRep',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='ReLU', inplace=True)),
    neck=dict(...),
    bbox_head=dict(...))
```

47.2.2 使用配置文件里的中间变量

配置文件里会使用一些中间变量，例如数据集里的 train_pipeline/test_pipeline。我们在定义新的 train_pipeline/test_pipeline 之后，需要将它们传递到 data 里。例如，我们想在训练或测试时，改变 YOLOv5 网络的 img_scale 训练尺度并在训练时添加 YOLOv5MixUp 数据增强，img_scale/train_pipeline/test_pipeline 是我们想要修改的中间变量。

注：使用 YOLOv5MixUp 数据增强时，需要将 YOLOv5MixUp 之前的训练数据处理流程定义在其 pre_transform 中。详细过程和图解可参见*YOLOv5 原理和实现全解析*。

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'

img_scale = (1280, 1280)  # 高度，宽度
affine_scale = 0.9        # 仿射变换尺度

mosaic_affine_pipeline = [
    dict(
        type='Mosaic',
```

(下页继续)

(续上页)

```
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='YOLOv5RandomAffine',
        max_rotate_degree=0.0,
        max_shear_degree=0.0,
        scaling_ratio_range=(1 - affine_scale, 1 + affine_scale),
        border=(-img_scale[0] // 2, -img_scale[1] // 2),
        border_val=(114, 114, 114))
]

train_pipeline = [
    *pre_transform, *mosaic_affine_pipeline,
    dict(
        type='YOLOv5MixUp',         # YOLOv5 的 MixUp (图像混合) 数据增强
        prob=0.1, # MixUp 概率
        pre_transform=[*pre_transform,*mosaic_affine_pipeline]), # MixUp 之前的训练数据处
理流程，包含 数据预处理流程、 'Mosaic' 和 'YOLOv5RandomAffine'
    dict(
        type='mmdet.Albu',
        transforms=albu_train_transforms,
        bbox_params=dict(
            type='BboxParams',
            format='pascal_voc',
            label_fields=['gt_bboxes_labels', 'gt_ignore_flags']),
        keymap={
            'img': 'image',
            'gt_bboxes': 'bboxes'
        }),
    dict(type='YOLOv5HSVRandomAug'),
    dict(type='mmdet.RandomFlip', prob=0.5),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'flip',
                   'flip_direction'))
]

test_pipeline = [
    dict(
        type='LoadImageFromFile'),
    dict(type='YOLOv5KeepRatioResize', scale=img_scale),
    dict(
```

(下页继续)

(续上页)

```
        type='LetterResize',
        scale=img_scale,
        allow_scale_up=False,
        pad_val=dict(img=114)),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'pad_param'))
]

train_dataloader = dict(dataset=dict(pipeline=train_pipeline))
val_dataloader = dict(dataset=dict(pipeline=test_pipeline))
test_dataloader = dict(dataset=dict(pipeline=test_pipeline))
```

我们首先定义新的 train_pipeline/test_pipeline 然后传递到 data 里。

同样的，如果我们想从 SyncBN 切换到 BN 或者 MMSyncBN，我们需要修改配置文件里的每一个 norm_cfg。

```
_base_ = './yolov5_s-v61_syncbn_8xb16-300e_coco.py'
norm_cfg = dict(type='BN', requires_grad=True)
model = dict(
    backbone=dict(norm_cfg=norm_cfg),
    neck=dict(norm_cfg=norm_cfg),
    ...)
```

47.2.3 复用 _base_ 文件中的变量

如果用户希望在当前配置中复用 _base_ 文件中的变量，则可以通过使用 {{_base_.xxx}} 的方式来获取对应变量的拷贝。而在新版 MMEngine 中，还支持省略 {{}} 的写法。例如：

```
_base_ = '../_base_/default_runtime.py'

pre_transform = _base_.pre_transform # 变量 pre_transform 等于 _base_ 中定义的 pre_
↪transform
```

47.3 通过脚本参数修改配置

当运行 tools/train.py 和 tools/test.py 时，可以通过 --cfg-options 来修改配置文件。

- 更新字典链中的配置

  可以按照原始配置文件中的 dict 键顺序地指定配置预选项。例如，使用 --cfg-options model.backbone.norm_eval=False 将模型主干网络中的所有 BN 模块都改为 train 模式。

- 更新配置列表中的键

  在配置文件里，一些字典型的配置被包含在列表中。例如，数据训练流程 data.train.pipeline 通常是一个列表，比如 [dict(type='LoadImageFromFile'), ...]。如果需要将 'LoadImageFromFile' 改成 'LoadImageFromNDArray'，需要写成下述形式：--cfg-options data.train.pipeline.0.type=LoadImageFromNDArray.

- 更新列表或元组的值

  如果要更新的值是列表或元组。例如，配置文件通常设置 model.data_preprocessor.mean=[123.675, 116.28, 103.53]。如果需要改变这个键，可以通过 --cfg-options model.data_preprocessor.mean="[127,127,127]" 来重新设置。需要注意，引号 " 是支持列表或元组数据类型所必需的，并且在指定值的引号内**不允许**有空格。

47.4 配置文件名称风格

我们遵循以下样式来命名配置文件。建议贡献者遵循相同的风格。

```
{algorithm name}_{model component names [component1]_[component2]_[...]}-[version id]_
↪[norm setting]_[data preprocessor type]_{training settings}_{training dataset␣
↪information}_[testing dataset information].py
```

文件名分为 8 个部分，其中 4 个必填部分、4 个可选部分。每个部分用 _ 连接，每个部分内的单词应该用 - 连接。{} 表示必填部分，[] 表示选填部分。

- {algorithm name}：算法的名称。它可以是检测器名称，例如 yolov5, yolov6, yolox 等。
- {component names}：算法中使用的组件名称，如 backbone、neck 等。例如 yolov5_s 代表其深度缩放因子 deepen_factor=0.33 以及其宽度缩放因子 widen_factor=0.5。
- [version_id] (可选)：由于 YOLO 系列算法迭代速度远快于传统目标检测算法，因此采用 version id 来区分不同子版本之间的差异。例如 YOLOv5 的 3.0 版本采用 Focus 层作为第一个下采样层，而 6.0 以后的版本采用 Conv 层作为第一个下采样层。
- [norm_setting] (可选)：bn 表示 Batch Normalization，syncbn 表示 Synchronized Batch Normalization。

- [data preprocessor type] (可选): fast 表示调用 YOLOv5DetDataPreprocessor 并配合 yolov5_collate 进行数据预处理，训练速度比默认的 mmdet.DetDataPreprocessor 更快，但是对多任务处理的灵活性较低。
- {training settings}: 训练设置的信息，例如 batch 大小、数据增强、损失、参数调度方式和训练最大轮次/迭代。例如: 8xb16-300e_coco 表示使用 8 个 GPU 每个 GPU 16 张图，并训练 300 个 epoch。缩写介绍:
  - {gpu x batch_per_gpu}: GPU 数和每个 GPU 的样本数。例如 4x4b 是 4 个 GPU 每个 GPU 4 张图的缩写。
  - {schedule}: 训练方案，MMYOLO 中默认为 300 个 epoch。
- {training dataset information}: 训练数据集，例如 coco, cityscapes, voc-0712, wider-face, balloon。
- [testing dataset information] (可选): 测试数据集，用于训练和测试在不同数据集上的模型配置。如果没有注明，则表示训练和测试的数据集类型相同。

CHAPTER 48

混合类图片数据增强更新

混合类图片数据增强是指类似 Mosaic 和 MixUp 一样，在运行过程中需要获取多张图片的标注信息进行融合。在 OpenMMLab 数据增强 pipeline 中一般是获取不到数据集其他索引的。为了实现上述功能，在 MMDetection 复现的 YOLOX 中提出了 MultiImageMixDataset 数据集包装器的概念。

`MultiImageMixDataset` 数据集包装器会传入一个包括 `Mosaic` 和 `RandAffine` 等数据增强，而 `CocoDataset` 中也需要传入一个包括图片和标注加载的 `pipeline`。通过这种方式就可以快速的实现混合类数据增强。其配置用法如下所示：

```
train_pipeline = [
    dict(type='Mosaic', img_scale=img_scale, pad_val=114.0),
    dict(
        type='RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
    dict(
        type='MixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0),
    ...
]
train_dataset = dict(
    # use MultiImageMixDataset wrapper to support mosaic and mixup
    type='MultiImageMixDataset',
    dataset=dict(
```

(下页继续)

(续上页)

```
        type='CocoDataset',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations', with_bbox=True)
        ]),
    pipeline=train_pipeline)
```

但是上述实现起来会有一个缺点：对于不熟悉 MMDetection 的用户来说，其经常会忘记 Mosaic 必须要和 `MultiImageMixDataset` 配合使用，而且这样会加大复杂度和理解难度。

为了解决这个问题，在 MMYOLO 中进一步进行了简化。直接让 `pipeline` 获取到 `dataset` 对象，此时就可以将 `Mosaic` 等混合类数据增强的实现和使用随机翻转的操作一样，不再需要数据集包装器。新的配置写法为：

```
pre_transform = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True)
]
train_pipeline = [
    *pre_transform,
    dict(
        type='Mosaic',
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='mmdet.RandomAffine',
        scaling_ratio_range=(0.1, 2),
        border=(-img_scale[0] // 2, -img_scale[1] // 2)),
    dict(
        type='YOLOXMixUp',
        img_scale=img_scale,
        ratio_range=(0.8, 1.6),
        pad_val=114.0,
        pre_transform=pre_transform),
    ...
]
```

一个稍微复杂点的包括 MixUp 的 YOLOv5-m 配置如下所示：

```
mosaic_affine_pipeline = [
    dict(
        type='Mosaic',
```

(下页继续)

(续上页)

```
        img_scale=img_scale,
        pad_val=114.0,
        pre_transform=pre_transform),
    dict(
        type='YOLOv5RandomAffine',
        max_rotate_degree=0.0,
        max_shear_degree=0.0,
        scaling_ratio_range=(1 - affine_scale, 1 + affine_scale),
        border=(-img_scale[0] // 2, -img_scale[1] // 2),
        border_val=(114, 114, 114))
]

# enable mixup
train_pipeline = [
    *pre_transform, *mosaic_affine_pipeline,
    dict(
        type='YOLOv5MixUp',
        prob=0.1,
        pre_transform=[*pre_transform, *mosaic_affine_pipeline]),
    dict(
        type='mmdet.Albu',
        transforms=albu_train_transforms,
        bbox_params=dict(
            type='BboxParams',
            format='pascal_voc',
            label_fields=['gt_bboxes_labels', 'gt_ignore_flags']),
        keymap={
            'img': 'image',
            'gt_bboxes': 'bboxes'
        }),
    dict(type='YOLOv5HSVRandomAug'),
    dict(type='mmdet.RandomFlip', prob=0.5),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'flip',
                   'flip_direction'))
]
```

其实现过程非常简单，只需要在 Dataset 中将本身对象传给 pipeline 即可，具体代码如下：

```
def prepare_data(self, idx) -> Any:
    """Pass the dataset to the pipeline during training to support mixed
    data augmentation, such as Mosaic and MixUp."""
    if self.test_mode is False:
```

(下页继续)

(续上页)

```
        data_info = self.get_data_info(idx)
        data_info['dataset'] = self
        return self.pipeline(data_info)
    else:
        return super().prepare_data(idx)
```

CHAPTER 49

旋转目标检测

旋转目标检测（Rotated Object Detection），又称为有向目标检测（Oriented Object Detection），试图在检测出目标位置的同时得到目标的方向信息。它通过重新定义目标表示形式，以及增加回归自由度数量的操作，实现旋转矩形、四边形甚至任意形状的目标检测。旋转目标检测在人脸识别、场景文字、遥感影像、自动驾驶、医学图像、机器人抓取等领域都有广泛应用。

关于旋转目标检测的详细介绍请参考文档 MMRotate 基础知识

MMYOLO 中的旋转目标检测依赖于 MMRotate 1.x，请参考文档 开始你的第一步 安装 MMRotate 1.x。

本教程将介绍如何在 MMYOLO 中训练和使用旋转目标检测模型，目前支持了 RTMDet-R。

49.1 数据集准备

对于旋转目标检测数据集，目前最常用的数据集是 DOTA 数据集，由于 DOTA 数据集中的图像分辨率较大，因此需要进行切片处理，数据集准备请参考 Preparing DOTA Dataset.

处理后的数据集结构如下：

```
mmyolo
├── data
│   ├── split_ss_dota
│   │   ├── trainval
│   │   │   ├── images
│   │   │   ├── annfiles
│   │   ├── test
```

(下页继续)

(续上页)

```
|   |   |   ├── images
|   |   |   ├── annfiles
|   ├── split_ms_dota
|   |   ├── trainval
|   |   |   ├── images
|   |   |   ├── annfiles
|   |   ├── test
|   |   |   ├── images
|   |   |   ├── annfiles
```

其中 split_ss_dota 是单尺度切片，split_ms_dota 是多尺度切片，可以根据需要选择。

对于自定义数据集，我们建议将数据转换为 DOTA 格式并离线进行转换，如此您只需在数据转换后修改 config 的数据标注路径和类别即可。

为了方便使用，我们同样提供了基于 COCO 格式的旋转标注格式，将多边形检测框储存在 COCO 标注的 segmentation 标签中，示例如下：

```
{
    "id": 131,
    "image_id": 72,
    "bbox": [123, 167, 11, 37],
    "area": 271.5,
    "category_id": 1,
    "segmentation": [[123, 167, 128, 204, 134, 201, 132, 167]],
    "iscrowd": 0,
}
```

49.2 配置文件

这里以 RTMDet-R 为例介绍旋转目标检测的配置文件，其中大部分和水平检测模型相同，主要介绍它们的差异，包括数据集和评测器配置、检测头、可视化等。

得益于 MMEngine 的配置文件系统，大部分模块都可以调用 MMRotate 中的模块。

49.2.1 数据集和评测器配置

关于配置文件的基础请先阅读*学习 YOLOV5 配置文件*. 下面介绍旋转目标检测的一些必要设置。

```
dataset_type = 'YOLOv5DOTADataset'  # 数据集类型，这将被用来定义数据集
data_root = 'data/split_ss_dota/'  # 数据的根路径

angle_version = 'le90' # 角度范围的定义，目前支持 oc, le90 和 le135

train_pipeline = [
    # 训练数据读取流程
    dict(
        type='LoadImageFromFile'), # 第 1 个流程，从文件路径里加载图像
    dict(type='LoadAnnotations', # 第 2 个流程，对于当前图像，加载它的注释信息
         with_bbox=True, # 是否使用标注框 (bounding box)，目标检测需要设置为 True
         box_type='qbox'), # 指定读取的标注格式，旋转框数据集默认的数据格式为四边形
    dict(type='mmrotate.ConvertBoxType', # 第 3 个流程，转换标注格式
         box_type_mapping=dict(gt_bboxes='rbox')), # 将四边形标注转化为旋转框标注

    # 训练数据处理流程
    dict(type='mmdet.Resize', scale=(1024, 1024), keep_ratio=True),
    dict(type='mmdet.RandomFlip',
         prob=0.75,
         direction=['horizontal', 'vertical', 'diagonal']),
    dict(type='mmrotate.RandomRotate', # 旋转数据增强
         prob=0.5, # 旋转概率 0.5
         angle_range=180, # 旋转范围 180
         rotate_type='mmrotate.Rotate', # 旋转方法
         rect_obj_labels=[9, 11]), # 由于 DOTA 数据集中标号为 9 的 'storage-tank' 和标号␣
↪11 的 'roundabout' 两类为正方形标注，无需角度信息，旋转中将这两类保持为水平
    dict(type='mmdet.Pad', size=img_scale, pad_val=dict(img=(114, 114, 114))),
    dict(type='RegularizeRotatedBox', # 统一旋转框表示形式
         angle_version=angle_version), # 根据角度的定义方式进行
    dict(type='mmdet.PackDetInputs')
]

train_dataloader = dict(
    batch_size=train_batch_size_per_gpu,
    num_workers=train_num_workers,
    persistent_workers=persistent_workers,
    pin_memory=True,
    collate_fn=dict(type='yolov5_collate'),
    sampler=dict(type='DefaultSampler', shuffle=True),
    dataset=dict( # 训练数据集的配置
        type=dataset_type,
```

(下页继续)

(续上页)

```
        data_root=data_root,
        ann_file='trainval/annfiles/', # 标注文件夹路径
        data_prefix=dict(img_path='trainval/images/'), # 图像路径前缀
        img_shape=(1024, 1024), # 图像大小
        filter_cfg=dict(filter_empty_gt=True), # 标注的过滤配置
        pipeline=train_pipeline)) # 这是由之前创建的 train_pipeline 定义的数据处理流程
```

RTMDet-R 保持论文内的配置，默认仅采用随机旋转增强，得益于 BoxType 设计，在数据增强阶段，大部分增强无需改动代码即可直接支持，例如 MixUp 和 Mosaic 等，可以直接在 pipeline 中使用。

警告: 目前已知 Albu 数据增强仅支持水平框，在使用其他的数据增强时建议先使用可视化数据集脚本 `browse_dataset.py` 验证数据增强是否正确。

RTMDet-R 测试阶段仅采用 Resize 和 Pad，在验证和评测时，都采用相同的数据流进行推理。

```
val_pipeline = [
    dict(type='LoadImageFromFile', backend_args=_base_.backend_args),
    dict(type='mmdet.Resize', scale=(1024, 1024), keep_ratio=True),
    dict(
        type='mmdet.Pad', size=(1024, 1024),
        pad_val=dict(img=(114, 114, 114))),
    # 和训练时一致，先读取标注再转换标注格式
    dict(
        type='LoadAnnotations',
        with_bbox=True,
        box_type='qbox',
        _scope_='mmdet'),
    dict(
        type='mmrotate.ConvertBoxType',
        box_type_mapping=dict(gt_bboxes='rbox')),
    dict(
        type='mmdet.PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]

val_dataloader = dict(
    batch_size=val_batch_size_per_gpu,
    num_workers=val_num_workers,
    persistent_workers=persistent_workers,
    pin_memory=True,
    drop_last=False,
```

(下页继续)

(续上页)

```
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='trainval/annfiles/',
        data_prefix=dict(img_path='trainval/images/'),
        img_shape=(1024, 1024),
        test_mode=True,
        batch_shapes_cfg=batch_shapes_cfg,
        pipeline=val_pipeline))
```

评测器 用于计算训练模型在验证和测试数据集上的指标。评测器的配置由一个或一组评价指标（Metric）配置组成：

```
val_evaluator = dict( # 验证过程使用的评测器
    type='mmrotate.DOTAMetric', # 用于评估旋转目标检测的 mAP 的 dota 评价指标
    metric='mAP' # 需要计算的评价指标
)
test_evaluator = val_evaluator  # 测试过程使用的评测器
```

由于 DOTA 测试数据集没有标注文件，如果要保存在测试数据集上的检测结果，则可以像这样编写配置：

```
# 在测试集上推理，
# 并将检测结果转换格式以用于提交结果
test_dataloader = dict(
    batch_size=val_batch_size_per_gpu,
    num_workers=val_num_workers,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        data_prefix=dict(img_path='test/images/'),
        img_shape=(1024, 1024),
        test_mode=True,
        batch_shapes_cfg=batch_shapes_cfg,
        pipeline=test_pipeline))
test_evaluator = dict(
    type='mmrotate.DOTAMetric',
    format_only=True, # 只将模型输出转换为 DOTA 的 txt 提交格式并压缩成 zip
    merge_patches=True, # 将切片结果合并成大图检测结果
    outfile_prefix='./work_dirs/dota_detection/submission') # 输出测试文件夹的路径
```

如果使用基于 COCO 格式的旋转框标注，只需要修改 pipeline 中数据读取流程和训练数据集的配置，以训练数据为例：

```
dataset_type='YOLOv5CocoDataset'

train_pipeline = [
    # 训练数据读取流程
    dict(
        type='LoadImageFromFile'), # 第 1 个流程，从文件路径里加载图像
    dict(type='LoadAnnotations', # 第 2 个流程，对于当前图像，加载它的注释信息
         with_bbox=True, # 是否使用标注框 (bounding box)，目标检测需要设置为 True
         with_mask=True, # 读取储存在 segmentation 标注中的多边形标注
         poly2mask=False) # 不执行 poly2mask，后续会将 poly 转化成检测框
    dict(type='ConvertMask2BoxType', # 第 3 个流程，将 mask 标注转化为 boxtype
         box_type='rbox'), # 目标类型是 rbox 旋转框
    # 剩余的其他 pipeline
    ...
]

metainfo = dict( # DOTA 数据集的 metainfo
    classes=('plane', 'baseball-diamond', 'bridge', 'ground-track-field',
             'small-vehicle', 'large-vehicle', 'ship', 'tennis-court',
             'basketball-court', 'storage-tank', 'soccer-ball-field',
             'roundabout', 'harbor', 'swimming-pool', 'helicopter'))

train_dataloader = dict(
    dataset=dict( # 训练数据集的配置
        type=dataset_type,
        metainfo=metainfo,
        data_root=data_root,
        ann_file='train/train.json', # 标注文件路径
        data_prefix=dict(img='train/images/'), # 图像路径前缀
        filter_cfg=dict(filter_empty_gt=True), # 标注的过滤配置
        pipeline=train_pipeline), # 数据处理流程
)
```

49.2.2 模型配置

对于旋转目标检测器，在模型配置中 backbone 和 neck 的配置和其他模型是一致的，主要差异在检测头上。目前仅支持 RTMDet-R 旋转目标检测，下面介绍新增的参数：

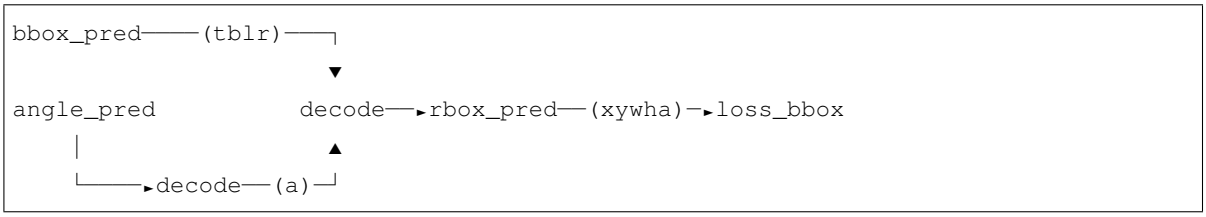
1. `angle_version` 角度范围，用于在训练时限制角度的范围，可选的角度范围有 `le90`, `le135` 和 `oc`。
2. `angle_coder` 角度编码器，和 bbox coder 类似，用于编码和解码角度。

   默认使用的角度编码器是 `PseudoAngleCoder`，即”伪角度编码器“，并不进行编解码，直接回归角度参数。这样设计的目标是能更好的自定义角度编码方式，而无需重写代码，例如 CSL，DCL，PSC 等方法。
3. `use_hbbox_loss` 是否使用水平框 loss。考虑到部分角度编码解码过程不可导，直接使用旋转框的损失函数无法学习角度，因此引入该参数用于将框和角度分开训练。
4. `loss_angle` 角度损失函数。在设定 `use_hbbox_loss=True` 时必须设定，而使用旋转框损失时可选，此时可以作为回归损失的辅助。

通过组合 `use_hbbox_loss` 和 `loss_angle` 可以控制旋转框训练时的回归损失计算方式，共有三种组合方式：

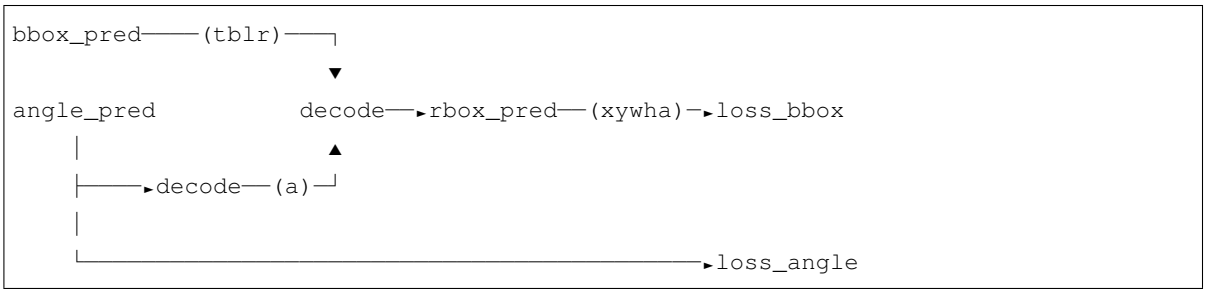
- `use_hbbox_loss=False` 且 `loss_angle` 为 None.

  此时框预测和角度预测进行合并，直接对旋转框预测进行回归，此时 `loss_bbox` 应当设定为旋转框损失，例如 `RotatedIoULoss`。这种方案和水平检测模型的回归方式基本一致，只是多了额外的角度编解码过程。



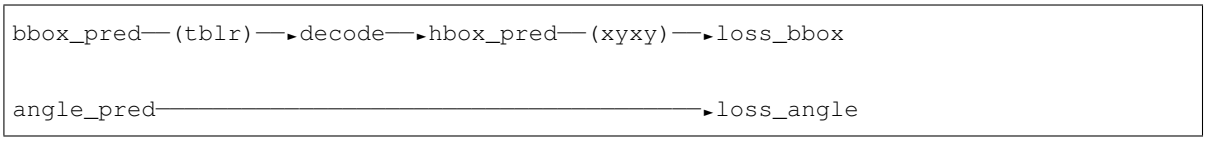
- `use_hbbox_loss=False`，同时设定 `loss_angle`.

  此时会增加额外的角度回归和分类损失，具体的角度损失类型需要根据角度编码器 `angle_code` 进行选择。



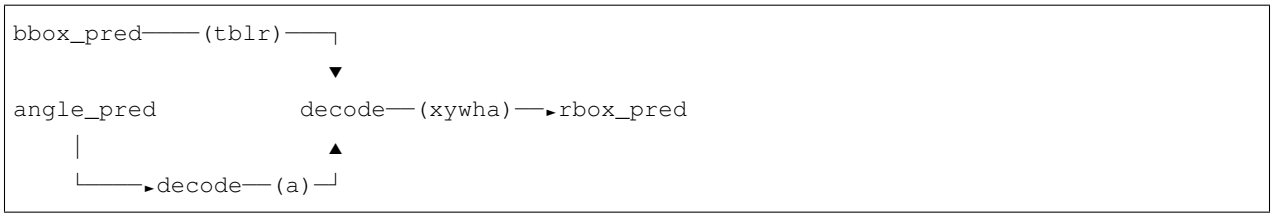
- `use_hbbox_loss=True` 且 `loss_angle` 为 None.

此时框预测和角度预测完全分离，将两个分支视作两个任务进行训练。此时 loss_bbox 要设定为水平框的损失函数，例如 IoULoss 。

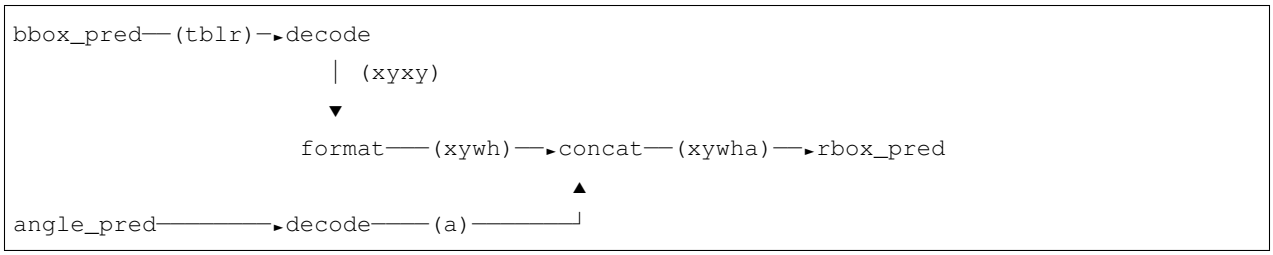


除了检测头中的参数，在 test_cfg 中还增加了 decoded_with_angle 参数用来控制推理时角度的处理逻辑，默认设定为 True 。设计这个参数的目标是让训练过程和推理过程的逻辑对齐，该参数会影响最终的精度。

当 decoded_with_angle=True 时，将框和角度同时送入 bbox_coder 中。此时要使用旋转框的编解码器，例如 DistanceAnglePointCoder。



当 decoded_with_angle=False 时，首先解码出水平检测框，之后将角度 concat 到检测框。此时要使用水平框的编解码器，例如 DistancePointBBoxCoder。



49.2.3 可视化器

由于旋转框和水平框的差异，旋转目标检测模型需要使用 MMRotate 中的 RotLocalVisualizer，配置如下：

```
vis_backends = [dict(type='LocalVisBackend')]  # 可视化后端，请参考 https://mmengine.
↪readthedocs.io/zh_CN/latest/advanced_tutorials/visualization.html
visualizer = dict(
    type='mmrotate.RotLocalVisualizer', vis_backends=vis_backends, name='visualizer')
```

49.3 实用工具

目前测试可用的工具包括：

可视化数据集

CHAPTER 50

自定义安装

50.1 CUDA 版本

在安装 PyTorch 时，你需要指定 CUDA 的版本。如果你不清楚应该选择哪一个，请遵循我们的建议。

- 对于 Ampere 架构的 NVIDIA GPU，例如 GeForce 30 系列以及 NVIDIA A100，CUDA 11 是必需的。
- 对于更早的 NVIDIA GPU，CUDA 11 是向后兼容 (backward compatible) 的，但 CUDA 10.2 能够提供更好的兼容性，也更加轻量。

请确保你的 GPU 驱动版本满足最低的版本需求，参阅 NVIDIA 官方的 CUDA 工具箱和相应的驱动版本关系表。

注解： 如果按照我们的最佳实践进行安装，CUDA 运行时库就足够了，因为我们提供相关 CUDA 代码的预编译，不需要进行本地编译。但如果你希望从源码进行 MMCV 的编译，或是进行其他 CUDA 算子的开发，那么就必须安装完整的 CUDA 工具链，参见 NVIDIA 官网，另外还需要确保该 CUDA 工具链的版本与 PyTorch 安装时的配置相匹配（如用 `conda install` 安装 PyTorch 时指定的 cudatoolkit 版本）。

50.2 不使用 MIM 安装 MMEngine

要使用 pip 而不是 MIM 来安装 MMEngine，请遵照 MMEngine 安装指南。

例如，你可以通过以下命令安装 MMEngine：

```
pip install "mmengine>=0.6.0"
```

50.3 不使用 MIM 安装 MMCV

MMCV 包含 C++ 和 CUDA 扩展，因此其对 PyTorch 的依赖比较复杂。MIM 会自动解析这些依赖，选择合适的 MMCV 预编译包，使安装更简单，但它并不是必需的。

要使用 pip 而不是 MIM 来安装 MMCV，请遵照 MMCV 安装指南。它需要您用指定 URL 的形式手动指定对应的 PyTorch 和 CUDA 版本。

例如，下述命令将会安装基于 PyTorch 1.12.x 和 CUDA 11.6 编译的 mmcv：

```
pip install "mmcv>=2.0.0rc4" -f https://download.openmmlab.com/mmcv/dist/cu116/torch1.
↪12.0/index.html
```

50.4 在 CPU 环境中安装

我们的代码能够建立在只使用 CPU 的环境（CUDA 不可用）。

在 CPU 模式下，可以进行模型训练（需要 MMCV 版本 >= `2.0.0rc1`）、测试或者推理，然而以下功能将在 CPU 模式下不能使用：

- Deformable Convolution
- Modulated Deformable Convolution
- ROI pooling
- Deformable ROI pooling
- CARAFE: Content-Aware ReAssembly of FEatures
- SyncBatchNorm
- CrissCrossAttention: Criss-Cross Attention
- MaskedConv2d
- Temporal Interlace Shift
- nms_cuda

- sigmoid_focal_loss_cuda
- bbox_overlaps

因此，如果尝试使用包含上述操作的模型进行训练/测试/推理，将会报错。下表列出了由于依赖上述算子而无法在 CPU 上运行的相关模型：

50.5 在 Google Colab 中安装

Google Colab 通常已经包含了 PyTorch 环境，因此我们只需要安装 MMEngine、MMCV、MMDetection 和 MMYOLO 即可，命令如下：

步骤 1. 使用 MIM 安装 MMEngine 、MMCV 和 MMDetection 。

```
!pip3 install openmim
!mim install "mmengine>=0.6.0"
!mim install "mmcv>=2.0.0rc4,<2.1.0"
!mim install "mmdet>=3.0.0,<4.0.0"
```

步骤 2. 使用源码安装 MMYOLO：

```
!git clone https://github.com/open-mmlab/mmyolo.git
%cd mmyolo
!pip install -e .
```

步骤 3. 验证安装是否成功：

```
import mmyolo
print(mmyolo.__version__)
# 预期输出：0.1.0 或其他版本号
```

注解： 在 Jupyter 中，感叹号 ！用于执行外部命令，而 `%cd` 是一个魔术命令，用于切换 Python 的工作路径。

50.6 使用多个 MMYOLO 版本进行开发

训练和测试的脚本已经在 `PYTHONPATH` 中进行了修改，以确保脚本使用当前目录中的 MMYOLO。

要使环境中安装默认的 MMYOLO 而不是当前正在在使用的，可以删除出现在相关脚本中的如下代码：

```
PYTHONPATH="$(dirname $0)/..":$PYTHONPATH
```

CHAPTER 51

常见警告说明

本文档收集用户经常疑惑的警告信息说明，方便大家理解。

51.1 xxx registry in mmyolo did not set import location

完整信息为 The xxx registry in mmyolo did not set import location. Fallback to call `mmyolo.utils.register_all_modules` instead.。这个警告的含义说某个模块在导入时候发现没有设置导入的 location，导致无法确定其位置，因此会自动调用 `mmyolo.utils.register_all_modules` 触发包的导入。这个警告属于 MMEngine 中非常底层的模块警告，用户理解起来可能比较困难，不过对大家使用没有任何影响，可以直接忽略。

51.2 save_param_schedulers is true but self.param_schedulers is None

以 YOLOv5 算法为例，这是因为 YOLOv5 中重新写了参数调度器策略 `YOLOv5ParamSchedulerHook`，因此 MMEngine 中设计的 ParamScheduler 是没有使用的，但是 YOLOv5 配置中也没有设置 `save_param_schedulers` 为 False。首先这个警告对性能和恢复训练没有任何影响，用户如果觉得这个警告会影响体验，可以设置 `default_hooks.checkpoint.save_param_scheduler` 为 False 或者训练时候通过命令行设置 `--cfg-options default_hooks.checkpoint.save_param_scheduler=False` 即可。

51.3 The loss_cls will be 0. This is a normal phenomenon.

这个和具体算法有关。以 YOLOv5 为例，其分类 loss 是只考虑正样本的，如果类别是 1，那么分类 loss 和 obj loss 就是功能重复的了，因此在设计上当类别是 1 的时候 loss_cls 是不计算的，因此始终是 0，这是正常现象。

51.4 The model and loaded state dict do not match exactly

这个警告是否会影响性能要根据进一步的打印信息来确定。如果是在微调模式下，由于用户自定义类别不一样无法加载 Head 模块的 COCO 预训练权重，这是一个正常现象，不会影响性能。

CHAPTER 52

常见问题解答

我们在这里列出了使用时的一些常见问题及其相应的解决方案。如果您发现有一些问题被遗漏，请随时提 PR 丰富这个列表。如果您无法在此获得帮助，请创建 issue 提问，但是请在模板中填写所有必填信息，这有助于我们更快定位问题。

52.1 为什么要推出 MMYOLO？

为什么要推出 MMYOLO？为何要单独开一个仓库而不是直接放到 MMDetection 中？自从开源后，不断收到社区小伙伴们类似的疑问，答案可以归纳为以下三点：

(1) 统一运行和推理平台

目前目标检测领域出现了非常多 YOLO 的改进算法，并且非常受大家欢迎，但是这类算法基于不同框架不同后端实现，存在较大差异，缺少统一便捷的从训练到部署的公平评测流程。

(2) 协议限制

众所周知，YOLOv5 以及其衍生的 YOLOv6 和 YOLOv7 等算法都是 GPL 3.0 协议，不同于 MMDetection 的 Apache 协议。由于协议问题，无法将 MMYOLO 直接并入 MMDetection 中。

(3) 多任务支持

还有一层深远的原因：**MMYOLO 任务不局限于 MMDetection**，后续会支持更多任务例如基于 MMPose 实现关键点相关的应用，基于 MMTracking 实现追踪相关的应用，因此不太适合直接并入 MMDetection 中。

52.2 projects 文件夹是用来干什么的？

projects 文件夹是 OpenMMLab 2.0 中引入的一个全新文件夹。其初衷有如下 3 点：

1. 便于社区贡献。由于 OpenMMLab 系列代码库对于代码合入有一套规范严谨的流程，这不可避免的会导致算法复现周期很长，不利于社区贡献
2. 便于快速支持新算法。算法开发周期过长同样会导致用户无法尽快体验最新算法
3. 便于快速支持新方向和新特性。新发展方向或者一些新的特性可能和现如今代码库中的设计有些不兼容，没法快速合入到代码库中

综上所述，projects 文件夹的引入主要是解决算法复现周期过长导致的新算法支持速度较慢，新特性支持较复杂等多个问题。projects 中每个文件夹属于一个完全独立的工程，社区用户可以通过 projects 快速支持一些在当前版本中较难支持或者想快速支持的新算法和新特性。等后续设计稳定或者代码符合合入规范，则会考虑合入到主分支中。

52.3 YOLOv5 backbone 替换为 Swin 后效果很差

在轻松更换主干网络 一文中我们提供了大量替换 backbone 的教程，但是该文档只是教用户如何替换 backbone，直接训练不一定能得到比较优异的结果。原因是不同 backbone 所需要的训练超参是不一样的，以 Swin 和 YOLOv5 backbone 为例两者差异较大，Swin 属于 transformer 系列算法，而 YOLOv5 backbone 属于卷积系列算法，其训练的优化器、学习率以及其他超参差异较大。如果强行将 Swin 作为 YOLOv5 backbone 且想取得不错的效果，需要同时调整诸多参数。

52.4 MM 系列开源库中有很多组件，如何在 MMYOLO 中使用？

在 OpenMMLab 2.0 中对多个 MM 系列开源库之间的模块跨库调用功能进行增强。目前在 MMYOLO 中可以在配置文件中通过 `MM 算法库 A. 模块名` 来之间调用 MM 算法库 A 中已经被注册的任意模块。具体例子可以参考轻松更换主干网络 中使用在 MMClassification 中实现的主干网络章节，其他模块调用也是相同的用法。

52.5 MMYOLO 中是否可以加入纯背景图片进行训练？

将纯背景图片加入训练大部分情况可以抑制误报率，是否将纯背景图片加入训练功能已经大部分数据集上支持了。以 `YOLOv5CocoDataset` 为例，核心控制参数是 `train_dataloader.dataset.filter_cfg.filter_empty_gt`，如果 `filter_empty_gt` 为 True 表示将纯背景图片过滤掉不加入训练，反之将纯背景图片加入到训练中。目前 MMYOLO 中大部分算法都是默认将纯背景图片加入训练中。

52.6 MMYOLO 是否有计算模型推理 FPS 脚本？

MMYOLO 是基于 MMDet 3.x 来开发的，在 MMDet 3.x 中提供了计算模型推理 FPS 的脚本。具体脚本为 benchmark。我们推荐大家使用 mim 直接跨库启动 MMDet 中的脚本而不是直接复制到 MMYOLO 中。关于如果通过 mim 启动 MMDet 中脚本，可以查看使用 *mim* 跨库调用其他 *OpenMMLab* 仓库的脚本。

52.7 MMDeploy 和 EasyDeploy 有啥区别？

MMDeploy 是由 OpenMMLab 中部署团队开发的针对 OpenMMLab 系列算法库提供部署支持的开源库，支持各种后端和自定义等等强大功能。EasyDeploy 是由社区小伙伴提供的一个相比 MMDeploy 更加简单易用的部署 projects。EasyDeploy 支持的功能目前没有 MMDeploy 多，但是使用上更加简单。MMYOLO 中同时提供对 MMDeploy 和 EasyDeploy 的支持，用户可以根据自己需求选择。

52.8 COCOMetric 中如何查看每个类的 AP

只需要在配置中设置 `test_evaluator.classwise` 为 True，或者在 test.py 运行时候增加 `--cfg-options test_evaluator.classwise=True` 即可。

52.9 MMYOLO 中为何没有支持 MMDet 类似的自动学习率缩放功能？

原因是实验发现 YOLO 系列算法不是非常满足线性缩放功能。在多个数据集上验证发现会出现不基于 batch size 自动学习率缩放效果好于缩放的情形。因此暂时 MMYOLO 还没有支持自动学习率缩放功能。

52.10 自己训练的模型权重尺寸为啥比官方发布的大？

原因是用户自己训练的权重通常包括 `optimizer`、`ema_state_dict` 和 `message_hub` 等额外数据，这部分数据我们会在模型发布时候自动删掉，而用户直接基于框架跑的模型权重是全部保留的，所以用户自己训练的模型权重尺寸会比官方发布的大。你可以使用 publish_model.py 脚本删掉额外字段。

52.11 RTMDet 为何训练所占显存比 YOLOv5 多很多？

训练显存较多的原因主要是 assigner 部分的差异。YOLOv5 采用的是非常简单且高效的 shape 匹配 assigner，而 RTMDet 中采用的是动态的全 batch 计算的 dynamic soft label assigner，其内部的 Cost 矩阵需要消耗比较多的显存，特别是当前 batch 中标注框过多时候。后续我们会考虑解决这个问题。

52.12 修改一些代码后是否需要重新安装 MMYOLO

在不新增 py 代码情况下，如果你遵循最佳实践，即使用 `mim install -v -e .` 安装的 MMYOLO，则对本地代码所作的任何修改都会生效，无需重新安装。但是如果你是新增了 py 文件然后在里面新增的代码，则依然需要重新安装即运行 `mim install -v -e .`。

52.13 如何使用多个 MMYOLO 版本进行开发

若你拥有多个 MMYOLO 工程文件夹，例如 mmyolo-v1, mmyolo-v2。在使用不同版本 MMYOLO 时候，你可以在终端运行前设置

```
PYTHONPATH="$(dirname $0)/..":$PYTHONPATH
```

使得当前环境生效。如果要使用环境中安装默认的 MMYOLO 而不是当前正在在使用的，可以删除出现上述命令或者通过如下命令重置

```
unset PYTHONPATH
```

52.14 训练中保存最好模型

用户可以通过在配置中设置 `default_hooks.checkpoint.save_best` 参数来选择根据什么指标来筛选最优模型。以 COCO 数据集检测任务为例，`default_hooks.checkpoint.save_best` 可以选择输入的参数有:

1. `auto` 将会根据验证集中的第一个评价指标作为筛选条件。
2. `coco/bbox_mAP` 将会根据 `bbox_mAP` 作为筛选条件。
3. `coco/bbox_mAP_50` 将会根据 `bbox_mAP_50` 作为筛选条件。
4. `coco/bbox_mAP_75` 将会根据 `bbox_mAP_75` 作为筛选条件。
5. `coco/bbox_mAP_s` 将会根据 `bbox_mAP_s` 作为筛选条件。
6. `coco/bbox_mAP_m` 将会根据 `bbox_mAP_m` 作为筛选条件。
7. `coco/bbox_mAP_l` 将会根据 `bbox_mAP_l` 作为筛选条件。

此外用户还可以选择筛选的逻辑，通过设置配置中的 `default_hooks.checkpoint.rule` 来选择判断逻辑，如：`default_hooks.checkpoint.rule=greater` 表示指标越大越好。更详细的使用可以参考 checkpoint_hook 来修改

52.15 如何进行非正方形输入尺寸训练和测试?

在 YOLO 系列算法中默认配置基本上都是 640x640 或者 1280x1280 正方形尺度输入训练的。用户如果想进行非正方形尺度训练，你可以修改配置中 `image_scale` 参数，并将其他对应位置进行修改即可。用户可以参考我们提供的 yolov5_s-v61_fast_1xb12-40e_608x352_cat.py 配置。

CHAPTER 53

MMYOLO 跨库应用解析

CHAPTER 54

模型库和评测

本页面用于汇总 MMYOLO 中支持的各类模型性能和相关评测指标，方便用户对比分析。

54.1 COCO 数据集

- 所有模型均使用 COCO train2017 作为训练集，在 COCO val2017 上验证精度
- TRT-FP16-GPU-Latency(ms) 是指在 NVIDIA Tesla T4 设备上采用 TensorRT 8.4，batch size 为 1，测试 shape 为 640x640 且仅包括模型 forward 的 GPU Compute time (YOLOX-tiny 测试 shape 是 416x416)
- 模型参数量和 FLOPs 是采用 get_flops 脚本得到，不同的运算方式可能略有不同
- RTMDet 性能是通过 MMRazor 知识蒸馏 训练后的结果
- MMYOLO 中暂时只实现了 YOLOv6 2.0 版本，并且 L 和 M 为没有经过知识蒸馏的结果
- YOLOv8 是引入了实例分割标注优化后的结果，YOLOv5、YOLOv6 和 YOLOv7 没有采用实例分割标注优化
- PPYOLOE+ 使用 Obj365 作为预训练权重，因此 COCO 训练的 epoch 数只需要 80
- YOLOX-tiny、YOLOX-s 和 YOLOX-m 为采用了 RTMDet 中所提优化器参数训练所得，性能相比原始实现有不同程度提升

详情见如下内容

- RTMDet
- YOLOv5

- YOLOv6
- YOLOv7
- YOLOv8
- YOLOX
- PPYOLO-E

54.2 VOC 数据集

详情见如下内容

- YOLOv5

54.3 CrowdHuman 数据集

详情见如下内容

- YOLOv5

54.4 DOTA 1.0 数据集

CHAPTER 55

更新日志

55.1 v0.6.0 (15/8/2023)

55.1.1 亮点

- 支持 YOLOv5 实例分割
- 基于 MMPose 支持 YOLOX-Pose
- 添加 15 分钟的实例分割教程
- YOLOv5 支持使用 mask 标注来优化边界框
- 添加多尺度训练和测试文档

55.1.2 新特性

- 添加训练和测试技巧文档 (#659)
- 支持设置 `cache_size_limit` 参数，并支持 mmdet 3.0.0 (#707)
- 支持 YOLOv5u 和 YOLOv6 3.0 推理 (#624, #744)
- 支持仅模型推断 (#733)
- 添加 YOLOv8 deepstream 配置 (#633)
- 在 MMYOLO 应用程序中添加电离图示例 (#643)

55.1.3 Bug 修复

- 修复 browse_dataset 以可视化测试和验证集的问题 (#641)
- 修复安装文档错误 (#662)
- 修复 yolox-l ckpt 链接 (#677)
- 修正 YOLOv7 和 YOLOv8 图表中的拼写错误 (#621, #710)
- 调整 `boxam_vis_demo.py` 中包导入的顺序 (#655)

55.1.4 完善

- 优化 `convert_kd_ckpt_to_student.py` 文件 (#647)
- 添加 FAQ 和 training_testing_tricks 的英文文档 (#691, #693)

55.1.5 贡献者

总共 21 位开发者参与了本次版本

感谢 @Lum1104,@azure-wings,@FeiGeChuanShu,@Lingrui Gu,@Nioolek,@huayuan4396,@RangeKing,@danielhonies,@yechenzhi,@J
Mu,@kikefdezl,@zhangrui-wolf,@xin-li-67,@Ben-Louis,@zgzhengSEU,@VoyagerXvoyagerx,@tang576225574,@hhaAndroid

55.2 v0.5.0 (2/3/2023)

55.2.1 亮点

1. 支持了 RTMDet-R 旋转框目标检测任务和算法
2. YOLOv8 支持使用 mask 标注提升目标检测模型性能
3. 支持 MMRazor 搜索的 NAS 子网络作为 YOLO 系列算法的 backbone
4. 支持调用 MMRazor 对 RTMDet 进行知识蒸馏
5. MMYOLO 文档结构优化，内容全面升级
6. 基于 RTMDet 训练超参提升 YOLOX 精度和训练速度
7. 支持模型参数量、FLOPs 计算和提供 T4 设备上 GPU 延时数据，并更新了 Model Zoo
8. 支持测试时增强 TTA
9. 支持 RTMDet、YOLOv8 和 YOLOv7 assigner 可视化

55.2.2 新特性

1. 支持 RTMDet 实例分割任务的推理 (#583)
2. 美化 MMYOLO 中配置文件并增加更多注释 (#501, #506, #516, #529, #531, #539)
3. 重构并优化中英文文档 (#568, #573, #579, #584, #587, #589, #596, #599, #600)
4. 支持 fast 版本的 YOLOX (#518)
5. EasyDeploy 中支持 DeepStream，并添加说明文档 (#485, #545, #571)
6. 新增混淆矩阵绘制脚本 (#572)
7. 新增单通道应用案例 (#460)
8. 支持 auto registration (#597)
9. Box CAM 支持 YOLOv7、YOLOv8 和 PPYOLOE (#601)
10. 新增自动化生成 MM 系列 repo 注册信息和 tools 脚本 (#559)
11. 新增 YOLOv7 模型结构图 (#504)
12. 新增如何指定特定 GPU 训练和推理文档 (#503)
13. 新增训练或者测试时检查 `metainfo` 是否全为小写 (#535)
14. 增加 Twitter、Discord、Medium 和 YouTube 等链接 (#555)

55.2.3 Bug 修复

1. 修复 isort 版本问题 (#492, #497)
2. 修复 assigner 可视化模块的 type 错误 (#509)
3. 修复 YOLOv8 文档链接错误 (#517)
4. 修复 EasyDeploy 中的 RTMDet Decoder 错误 (#519)
5. 修复一些文档链接错误 (#537)
6. 修复 RTMDet-Tiny 权重路径错误 (#580)

55.2.4 完善

1. 完善更新 `contributing.md`
2. 优化 `DetDataPreprocessor` 支使其支持多任务 (#511)
3. 优化 `gt_instances_preprocess` 使其可以用于其他 YOLO 算法 (#532)
4. 新增 `yolov7-e6e` 权重转换脚本 (#570)
5. 参考 YOLOv8 推理代码修改 PPYOLOE (#614)

55.2.5 贡献者

总共 22 位开发者参与了本次版本

@triple-Mu, @isLinXu, @Audrey528, @TianWen580, @yechenzhi, @RangeKing, @lyviva, @Nioolek, @PeterH0323, @tianleiSHI, @aptsunny, @satuoqaq, @vansin, @xin-li-67, @VoyagerXvoyagerx, @landhill, @kitecats, @tang576225574, @HIT-cwh, @AI-Tianlong, @RangiLyu, @hhaAndroid

55.3 v0.4.0 (18/1/2023)

55.3.1 亮点

1. 实现了 YOLOv8 目标检测模型，并通过 projects/easydeploy 支持了模型部署
2. 新增了中英文版本的 YOLOv8 原理和实现全解析文档

55.3.2 新特性

1. 新增 YOLOv8 和 PPYOLOE 模型结构图 (#459, #471)
2. 调整最低支持 Python 版本从 3.6 升级为 3.7 (#449)
3. TensorRT-8 中新增新的 YOLOX decoder 写法 (#450)
4. 新增学习率可视化曲线脚本 (#479)
5. 新增脚本命令速查表 (#481)

55.3.3 Bug 修复

1. 修复 `optimize_anchors.py` 脚本导入错误问题 (#452)
2. 修复 `get_started.md` 中安装步骤错误问题 (#474)
3. 修复使用 `RTMDet` P6 模型时候 neck 报错问题 (#480)

55.3.4 视频

1. 发布了 玩转 MMYOLO 之实用篇（四）：顶会第一步 · 模块自定义

55.3.5 贡献者

总共 9 位开发者参与了本次版本

谢谢 @VoyagerXvoyagerx, @tianleiSHI, @RangeKing, @PeterH0323, @Nioolek, @triple-Mu, @lyviva, @Zheng-LinXiao, @hhaAndroid

55.4 v0.3.0 (8/1/2023)

55.4.1 亮点

1. 实现了 RTMDet 的快速版本。RTMDet-s 8xA100 训练只需要 14 个小时，训练速度相比原先版本提升 2.6 倍。
2. 支持 PPYOLOE 训练。
3. 支持 YOLOv5 的 `iscrowd` 属性训练。
4. 支持 YOLOv5 正样本分配结果可视化
5. 新增 YOLOv6 原理和实现全解析文档

55.4.2 新特性

1. 新增 `crowdhuman` 数据集 (#368)
2. EasyDeploy 中支持 TensorRT 推理 (#377)
3. 新增 `YOLOX` 结构图描述 (#402)
4. 新增视频推理脚本 (#392)
5. EasyDeploy 中支持 `YOLOv7` 部署 (#427)
6. 支持从 CLI 中的特定检查点恢复训练 (#393)
7. 将元信息字段设置为小写（#362、#412）
8. 新增模块组合文档 (#349, #352, #345)
9. 新增关于如何冻结 backbone 或 neck 权重的文档 (#418)
10. 在 `how_to.md` 中添加不使用预训练权重的文档 (#404)
11. 新增关于如何设置随机种子的文档 (#386)
12. 将 `rtmdet_description.md` 文档翻译成英文 (#353)

55.4.3 Bug 修复

1. 修复设置 `--class-id-txt` 时输出注释文件中的错误 (#430)
2. 修复 `YOLOv5` head 中的批量推理错误 (#413)
3. 修复某些 head 的类型提示（#415、#416、#443）
4. 修复 expected a non-empty list of Tensors 错误 (#376)
5. 修复 `YOLOv7` 训练中的设备不一致错误（#397）
6. 修复 `LetterResize` 中的 `scale_factor` 和 `pad_param` 值 (#387)
7. 修复 readthedocs 的 docstring 图形渲染错误 (#400)
8. 修复 `YOLOv6` 从训练到验证时的断言错误 (#378)
9. 修复 `np.int` 和旧版 builder.py 导致的 CI 错误 (#389)
10. 修复 MMDeploy 重写器 (#366)
11. 修复 MMYOLO 单元测试错误 (#351)
12. 修复 `pad_param` 错误 (#354)
13. 修复 head 推理两次的错误（#342）
14. 修复自定义数据集训练 (#428)

55.4.4 完善

1. 更新 `useful_tools.md` (#384)
2. 更新英文版 `custom_dataset.md` (#381)
3. 重写函数删除上下文参数 (#395)
4. 弃用 `np.bool` 类型别名 (#396)
5. 为自定义数据集添加新的视频链接 (#365)
6. 仅为模型导出 onnx (#361)
7. 添加 MMYOLO 回归测试 yml (#359)
8. 更新 `article.md` 中的视频教程 (#350)
9. 添加部署 demo (#343)
10. 优化 debug 模式下大图的可视化效果 (#346)
11. 改进 `browse_dataset` 的参数并支持 `RepeatDataset` (#340, #338)

55.4.5 视频

1. 发布了 基于 sahi 的大图推理
2. 发布了 自定义数据集从标注到部署保姆级教程

55.4.6 贡献者

总共 28 位开发者参与了本次版本

谢谢 @RangeKing, @PeterH0323, @Nioolek, @triple-Mu, @matrixgame2018, @xin-li-67, @tang576225574, @kitecats, @Seperendity, @diplomatist, @vaew, @wzr-skn, @VoyagerXvoyagerx, @MambaWong, @tianleiSHI, @cajgithub, @zhubochao, @lvhan028, @dsghaonan, @lyviva, @yuewangg, @wang-tf, @satuoqaq, @grimoire, @RunningLeon, @hanrui1sensetime, @RangiLyu, @hhaAndroid

55.5 v0.2.0（1/12/2022)

55.5.1 亮点

1. 支持 YOLOv7 P5 和 P6 模型
2. 支持 YOLOv6 中的 ML 大模型
3. 支持 Grad-Based CAM 和 Grad-Free CAM
4. 基于 sahi 支持 大图推理
5. projects 文件夹下新增 easydeploy 项目
6. 新增 自定义数据集教程

55.5.2 新特性

1. `browse_dataset.py` 脚本支持可视化原图、数据增强后和中间结果功能 (#304)
2. `image_demo.py` 新增预测结果保存为 labelme 格式功能 (#288, #314)
3. 新增 labelme 格式转 COCO 格式脚本 `labelme2coco` (#308, #313)
4. 新增 COCO 数据集切分脚本 `coco_split.py` (#311)
5. `how-to.md` 文档中新增两个 backbone 替换案例以及更新 `plugin.md` (#291)
6. 新增贡献者文档 `contributing.md` and 代码规范文档 `code_style.md` (#322)
7. 新增如何通过 mim 跨库调用脚本文档 (#321)
8. `YOLOv5` 支持 RV1126 设备部署 (#262)

55.5.3 Bug 修复

1. 修复 MixUp padding 错误 (#319)
2. 修复 LetterResize 和 YOLOv5KeepRatioResize 中 scale_factor 参数顺序错误 (#305)
3. 修复 YOLOX Nano 模型训练错误问题 (#285)
4. 修复 RTMDet 部署没有导包的错误 (#287)
5. 修复 int8 部署配置错误 (#315)
6. 修复 basebackbone 中 make_stage_plugins 注释 (#296)
7. 部署模块支持切换为 deploy 模式功能 (#324)
8. 修正 RTMDet 模型结构图中的错误 (#317)

55.5.4 完善

1. test.py 中新增 json 格式导出选项 (#316)
2. extract_subcoco.py 脚本中新增基于面积阈值过滤规则 (#286)
3. 部署相关中文文档翻译为英文 (#289)
4. 新增 YOLOv6 算法描述大纲文档 (#252)
5. 完善 config.md (#297, #303)
6. 完善 mosiac9 的 docstring (#307)
7. 完善 browse_coco_json.py 脚本输入参数 (#309)
8. 重构 dataset_analysis.py 中部分函数使其更加通用 (#294)

55.5.5 视频

1. 发布了 工程文件结构简析
2. 发布了 10 分钟换遍主干网络文档

55.5.6 贡献者

总共 14 位开发者参与了本次版本

谢谢 @fcakyon, @matrixgame2018, @MambaWong, @imAzhou, @triple-Mu, @RangeKing, @PeterH0323, @xin-li-67, @kitecats, @hanrui1sensetime, @AllentDan, @Zheng-LinXiao, @hhaAndroid, @wanghonglie

55.6 v0.1.3（10/11/2022)

55.6.1 新特性

1. 支持 CBAM 插件并提供插件文档 (#246)
2. 新增 YOLOv5 P6 模型结构图和相关说明 (#273)

55.6.2 Bug 修复

1. 基于 mmengine 0.3.1 修复保存最好权重时训练失败问题
2. 基于 mmdet 3.0.0rc3 修复 `add_dump_metric` 报错 (#253)
3. 修复 backbone 不支持 `init_cfg` 问题 (#272)
4. 基于 mmdet 3.0.0rc3 改变 typing 导入方式 (#261)

55.6.3 完善

1. `featmap_vis_demo` 支持文件夹和 url 输入 (#248)
2. 部署 docker 文件完善 (#242)

55.6.4 贡献者

总共 10 位开发者参与了本次版本

谢谢 @kitecats, @triple-Mu, @RangeKing, @PeterH0323, @Zheng-LinXiao, @tkhe, @weikai520, @zytx121, @wanghonglie, @hhaAndroid

55.7 v0.1.2（3/11/2022)

55.7.1 亮点

1. 支持 ONNXRuntime 和 TensorRT 的 YOLOv5/YOLOv6/YOLOX/RTMDet 部署
2. 支持 YOLOv6 s/t/n 模型训练
3. YOLOv5 支持 P6 大分辨率 1280 尺度训练
4. YOLOv5 支持 VOC 数据集训练
5. 支持 PPYOLOE 和 YOLOv7 模型推理和官方权重转化
6. How-to 文档中新增 YOLOv5 替换 backbone 教程

55.7.2 新特性

1. 新增 `optimize_anchors` 脚本 (#175)
2. 新增 `extract_subcoco` 脚本 (#186)
3. 新增 `yolo2coco` 转换脚本 (#161)
4. 新增 `dataset_analysis` 脚本 (#172)
5. 移除 Albu 版本限制 (#187)

55.7.3 Bug 修复

1. 修复当设置 `cfg.resume` 时候不生效问题 (#221)
2. 修复特征图可视化脚本中不显示 bbox 问题 (#204)
3. 更新 RTMDet 的 metafile (#188)
4. 修复 test_pipeline 中的可视化错误 (#166)
5. 更新 badges (#140)

55.7.4 完善

1. 优化 Readthedoc 显示页面 (#209)
2. 为 base model 添加模块结构图的 docstring (#196)
3. 支持 LoadAnnotations 中不包括任何实例逻辑 (#161)
4. 更新 `image_demo` 脚本以支持文件夹和 url 路径 (#128)
5. 更新 pre-commit hook (#129)

55.7.5 文档

1. 将 `yolov5_description.md`、`yolov5_tutorial.md` 和 `visualization.md` 翻译为英文 (#138, #198, #206)
2. 新增部署相关中文文档 (#220)
3. 更新 `config.md`、`faq.md` 和 `pull_request_template.md` (#190, #191, #200)
4. 更新 `article` 页面 (#133)

55.7.6 视频

1. 发布了特征图可视化视频
2. 发布了 YOLOv5 配置文件解读视频
3. 发布了 RTMDet-s 特征图可视化 demo 视频
4. 发布了源码解读和必备调试技巧视频

55.7.7 贡献者

总共 14 位开发者参与了本次版本

谢谢 @imAzhou, @triple-Mu, @RangeKing, @PeterH0323, @xin-li-67, @Nioolek, @kitecats, @Bin-ze, @JiayuXu0, @cydiachen, @zhiqwang, @Zheng-LinXiao, @hhaAndroid, @wanghonglie

55.8 v0.1.1（29/9/2022)

基于 MMDetection 的 RTMDet 高精度低延时目标检测算法，我们也同步发布了 RTMDet，并提供了 RTMDet 原理和实现全解析中文文档

55.8.1 亮点

1. 支持了 RTMDet
2. 新增了 RTMDet 原理和实现全解析中文文档
3. 支持对 backbone 自定义插件，并更新了 How-to 文档 (#75)

55.8.2 Bug 修复

1. 修复一些文档错误 (#66, #72, #76, #83, #86)
2. 修复权重链接错误 (#63)
3. 修复 `LetterResize` 使用 `imscale` api 时候输出不符合预期的 bug (#105)

55.8.3 完善

1. 缩减 docker 镜像尺寸 (#67)
2. 简化 BaseMixImageTransform 中 Compose 逻辑 (#71)
3. test 脚本支持 dump 结果 (#84)

贡献者

总共 13 位开发者参与了本次版本

谢谢 @wanghonglie, @hhaAndroid, @yang-0201, @PeterH0323, @RangeKing, @satuoqaq, @Zheng-LinXiao, @xin-li-67, @suibe-qingtian, @MambaWong, @MichaelCai0912, @rimoire, @Nioolek

55.9 v0.1.0（21/9/2022)

我们发布了 MMYOLO 开源库，其基于 MMEngine, MMCV 2.x 和 MMDetection 3.x 库. 目前实现了目标检测功能，后续会扩展为多任务。

55.9.1 亮点

1. 支持 YOLOv5/YOLOX 训练，支持 YOLOv6 推理。部署即将支持。
2. 重构了 MMDetection 的 YOLOX，提供了更快的训练和推理速度。
3. 提供了详细入门和进阶教程, 包括 YOLOv5 从入门到部署、YOLOv5 算法原理和实现全解析、特征图可视化等教程。

CHAPTER 56

MMYOLO 兼容性说明

56.1 MMYOLO v0.3.0

56.1.1 METAINFO 修改

为了和 OpenMMLab 其他仓库统一，将 Dataset 里 `METAINFO` 的所有键从大写改为小写。

56.1.2 关于图片 shape 顺序的说明

在 OpenMMLab 2.0 中，为了与 OpenCV 的输入参数相一致，图片处理 pipeline 中关于图像 shape 的输入参数总是以 `(width, height)` 的顺序排列。相反，为了计算方便，经过 pipeline 和 model 的字段的顺序是 `(height, width)`。具体来说在每个数据 pipeline 处理的结果中，字段和它们的值含义如下：

- img_shape: (height, width)
- ori_shape: (height, width)
- pad_shape: (height, width)
- batch_input_shape: (height, width)

以 `Mosaic` 为例，其初始化参数如下所示：

```
@TRANSFORMS.register_module()
class Mosaic(BaseTransform):
    def __init__(self,
```

(下页继续)

(续上页)

```
                img_scale: Tuple[int, int] = (640, 640),
                center_ratio_range: Tuple[float, float] = (0.5, 1.5),
                bbox_clip_border: bool = True,
                pad_val: float = 114.0,
                prob: float = 1.0) -> None:
        ...

        # img_scale 顺序应该是 (width, height)
        self.img_scale = img_scale

    def transform(self, results: dict) -> dict:
        ...

        results['img'] = mosaic_img
        # (height, width)
        results['img_shape'] = mosaic_img.shape[:2]
```

CHAPTER 57

默认约定

如果你想把 MMYOLO 修改为自己的项目，请遵循下面的约定。

57.1 关于图片 shape 顺序的说明

在 OpenMMLab 2.0 中，为了与 OpenCV 的输入参数相一致，图片处理 pipeline 中关于图像 shape 的输入参数总是以 `(width, height)` 的顺序排列。相反，为了计算方便，经过 pipeline 和 model 的字段的顺序是 `(height, width)`。具体来说在每个数据 pipeline 处理的结果中，字段和它们的值含义如下：

- img_shape: (height, width)
- ori_shape: (height, width)
- pad_shape: (height, width)
- batch_input_shape: (height, width)

以 `Mosaic` 为例，其初始化参数如下所示：

```
@TRANSFORMS.register_module()
class Mosaic(BaseTransform):
    def __init__(self,
                 img_scale: Tuple[int, int] = (640, 640),
                 center_ratio_range: Tuple[float, float] = (0.5, 1.5),
                 bbox_clip_border: bool = True,
                 pad_val: float = 114.0,
```

(下页继续)

(续上页)

```
                prob: float = 1.0) -> None:
        ...

        # img_scale 顺序应该是 (width, height)
        self.img_scale = img_scale

    def transform(self, results: dict) -> dict:
        ...

        results['img'] = mosaic_img
        # (height, width)
        results['img_shape'] = mosaic_img.shape[:2]
```

CHAPTER 58

代码规范

58.1 代码规范标准

58.1.1 PEP 8 ——Python 官方代码规范

Python 官方的代码风格指南，包含了以下几个方面的内容：

- 代码布局，介绍了 Python 中空行、断行以及导入相关的代码风格规范。比如一个常见的问题：当我的代码较长，无法在一行写下时，何处可以断行？
- 表达式，介绍了 Python 中表达式空格相关的一些风格规范。
- 尾随逗号相关的规范。当列表较长，无法一行写下而写成如下逐行列表时，推荐在末项后加逗号，从而便于追加选项、版本控制等。

```
# Correct:
FILES = ['setup.cfg', 'tox.ini']
# Correct:
FILES = [
    'setup.cfg',
    'tox.ini',
]
# Wrong:
FILES = ['setup.cfg', 'tox.ini',]
# Wrong:
FILES = [
```

(下页继续)

(续上页)

```
    'setup.cfg',
    'tox.ini'
]
```

- 命名相关规范、注释相关规范、类型注解相关规范，我们将在后续章节中做详细介绍。

  “A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.” PEP 8 –Style Guide for Python Code

---

注解： PEP 8 的代码规范并不是绝对的，项目内的一致性要优先于 PEP 8 的规范。OpenMMLab 各个项目都在 setup.cfg 设定了一些代码规范的设置，请遵照这些设置。一个例子是在 PEP 8 中有如下一个例子：

```
# Correct:
hypot2 = x*x + y*y
# Wrong:
hypot2 = x * x + y * y
```

这一规范是为了指示不同优先级，但 OpenMMLab 的设置中通常没有启用 yapf 的 `ARITHMETIC_PRECEDENCE_INDICATION` 选项，因而格式规范工具不会按照推荐样式格式化，以设置为准。

58.1.2 Google 开源项目风格指南

Google 使用的编程风格指南，包括了 Python 相关的章节。相较于 PEP 8，该指南提供了更为详尽的代码指南。该指南包括了语言规范和风格规范两个部分。

其中，语言规范对 Python 中很多语言特性进行了优缺点的分析，并给出了使用指导意见，如异常、Lambda 表达式、列表推导式、metaclass 等。

风格规范的内容与 PEP 8 较为接近，大部分约定建立在 PEP 8 的基础上，也有一些更为详细的约定，如函数长度、TODO 注释、文件与 socket 对象的访问等。

推荐将该指南作为参考进行开发，但不必严格遵照，一来该指南存在一些 Python 2 兼容需求，例如指南中要求所有无基类的类应当显式地继承 Object, 而在仅使用 Python 3 的环境中，这一要求是不必要的，依本项目中的惯例即可。二来 OpenMMLab 的项目作为框架级的开源软件，不必对一些高级技巧过于避讳，尤其是 MMCV。但尝试使用这些技巧前应当认真考虑是否真的有必要，并寻求其他开发人员的广泛评估。

另外需要注意的一处规范是关于包的导入，在该指南中，要求导入本地包时必须使用路径全称，且导入的每一个模块都应当单独成行，通常这是不必要的，而且也不符合目前项目的开发惯例，此处进行如下约定：

```
# Correct
from mmcv.cnn.bricks import (Conv2d, build_norm_layer, DropPath, MaxPool2d,
```

(下页继续)

(续上页)

```
                              Linear)
from ..utils import ext_loader

# Wrong
from mmcv.cnn.bricks import Conv2d, build_norm_layer, DropPath, MaxPool2d, \
                            Linear  # 使用括号进行连接，而不是反斜杠
from ...utils import is_str  # 最多向上回溯一层，过多的回溯容易导致结构混乱
```

OpenMMLab 项目使用 pre-commit 工具自动格式化代码，详情见贡献代码。

58.2 命名规范

58.2.1 命名规范的重要性

优秀的命名是良好代码可读的基础。基础的命名规范对各类变量的命名做了要求，使读者可以方便地根据代码名了解变量是一个类 / 局部变量 / 全局变量等。而优秀的命名则需要代码作者对于变量的功能有清晰的认识，以及良好的表达能力，从而使读者根据名称就能了解其含义，甚至帮助了解该段代码的功能。

58.2.2 基础命名规范

注意：

- 尽量避免变量名与保留字冲突，特殊情况下如不可避免，可使用一个后置下划线，如 class_
- 尽量不要使用过于简单的命名，除了约定俗成的循环变量 i，文件变量 f，错误变量 e 等。
- 不会被用到的变量可以命名为 _，逻辑检查器会将其忽略。

58.2.3 命名技巧

良好的变量命名需要保证三点：

1. 含义准确，没有歧义
2. 长短适中
3. 前后统一

```
# Wrong
class Masks(metaclass=ABCMeta):  # 命名无法表现基类；Instance or Semantic？
    pass

# Correct
```

(下页继续)

(续上页)

```
class BaseInstanceMasks(metaclass=ABCMeta):
    pass

# Wrong，不同地方含义相同的变量尽量用统一的命名
def __init__(self, inplanes, planes):
    pass

def __init__(self, in_channels, out_channels):
    pass
```

常见的函数命名方法：

- 动宾命名法：crop_img, init_weights
- 动宾倒置命名法：imread, bbox_flip

注意函数命名与参数的顺序，保证主语在前，符合语言习惯：

- check_keys_exist(key, container)
- check_keys_contain(container, key)

注意避免非常规或统一约定的缩写，如 nb -> num_blocks，in_nc -> in_channels

58.3 docstring 规范

58.3.1 为什么要写 docstring

docstring 是对一个类、一个函数功能与 API 接口的详细描述，有两个功能，一是帮助其他开发者了解代码功能，方便 debug 和复用代码；二是在 Readthedocs 文档中自动生成相关的 API reference 文档，帮助不了解源代码的社区用户使用相关功能。

58.3.2 如何写 docstring

与注释不同，一份规范的 docstring 有着严格的格式要求，以便于 Python 解释器以及 sphinx 进行文档解析，详细的 docstring 约定参见 PEP 257。此处以例子的形式介绍各种文档的标准格式，参考格式为 Google 风格。

1. 模块文档

   代码风格规范推荐为每一个模块（即 Python 文件）编写一个 docstring，但目前 OpenMMLab 项目大部分没有此类 docstring，因此不做硬性要求。

   ```
   """A one line summary of the module or program, terminated by a period.

   Leave one blank line. The rest of this docstring should contain an
   ```

(下页继续)

(续上页)

```
overall description of the module or program. Optionally, it may also
contain a brief description of exported classes and functions and/or usage
examples.

Typical usage example:

foo = ClassFoo()
bar = foo.FunctionBar()
"""
```

2. 类文档

类文档是我们最常需要编写的，此处，按照 OpenMMLab 的惯例，我们使用了与 Google 风格不同的写法。如下例所示，文档中没有使用 Attributes 描述类属性，而是使用 Args 描述 **init** 函数的参数。

在 Args 中，遵照 `parameter (type): Description.` 的格式，描述每一个参数类型和功能。其中，多种类型可使用 `(float or str)` 的写法，可以为 None 的参数可以写为 `(int, optional)`。

```
class BaseRunner(metaclass=ABCMeta):
    """The base class of Runner, a training helper for PyTorch.

    All subclasses should implement the following APIs:

    - ``run()``
    - ``train()``
    - ``val()``
    - ``save_checkpoint()``

    Args:
        model (:obj:`torch.nn.Module`): The model to be run.
        batch_processor (callable, optional): A callable method that process
            a data batch. The interface of this method should be
            ``batch_processor(model, data, train_mode) -> dict``.
            Defaults to None.
        optimizer (dict or :obj:`torch.optim.Optimizer`, optional): It can be
            either an optimizer (in most cases) or a dict of optimizers
            (in models that requires more than one optimizer, e.g., GAN).
            Defaults to None.
        work_dir (str, optional): The working directory to save checkpoints
            and logs. Defaults to None.
        logger (:obj:`logging.Logger`): Logger used during training.
             Defaults to None. (The default value is just for backward
             compatibility)
        meta (dict, optional): A dict records some import information such as
```

(下页继续)

(续上页)

```
            environment info and seed, which will be logged in logger hook.
            Defaults to None.
        max_epochs (int, optional): Total training epochs. Defaults to None.
        max_iters (int, optional): Total training iterations. Defaults to None.
    """

    def __init__(self,
                 model,
                 batch_processor=None,
                 optimizer=None,
                 work_dir=None,
                 logger=None,
                 meta=None,
                 max_iters=None,
                 max_epochs=None):
        ...
```

另外，在一些算法实现的主体类中，建议加入原论文的链接；如果参考了其他开源代码的实现，则应加入 modified from，而如果是直接复制了其他代码库的实现，则应加入 copied from ，并注意源码的 License。如有必要，也可以通过.. math:: 来加入数学公式

```
# 参考实现
# This func is modified from `detectron2
# <https://github.com/facebookresearch/detectron2/blob/
↪ffff8acc35ea88ad1cb1806ab0f00b4c1c5dbfd9/detectron2/structures/masks.py#L387>`_.

# 复制代码
# This code was copied from the `ubelt
# library<https://github.com/Erotemic/ubelt>`_.

# 引用论文 & 添加公式
class LabelSmoothLoss(nn.Module):
    r"""Initializer for the label smoothed cross entropy loss.

    Refers to `Rethinking the Inception Architecture for Computer Vision
    <https://arxiv.org/abs/1512.00567>`_.

    This decreases gap between output scores and encourages generalization.
    Labels provided to forward can be one-hot like vectors (NxC) or class
    indices (Nx1).
    And this accepts linear combination of one-hot like labels from mixup or
    cutmix except multi-label task.
```

(下页继续)

(续上页)

```
    Args:
        label_smooth_val (float): The degree of label smoothing.
        num_classes (int, optional): Number of classes. Defaults to None.
        mode (str): Refers to notes, Options are "original", "classy_vision",
            "multi_label". Defaults to "classy_vision".
        reduction (str): The method used to reduce the loss.
            Options are "none", "mean" and "sum". Defaults to 'mean'.
        loss_weight (float):  Weight of the loss. Defaults to 1.0.

    Note:
        if the ``mode`` is "original", this will use the same label smooth
        method as the original paper as:

        .. math::
            (1-\epsilon)\delta_{k, y} + \frac{\epsilon}{K}

        where :math:`\epsilon` is the ``label_smooth_val``, :math:`K` is
        the ``num_classes`` and :math:`\delta_{k,y}` is Dirac delta,
        which equals 1 for k=y and 0 otherwise.

        if the ``mode`` is "classy_vision", this will use the same label
        smooth method as the `facebookresearch/ClassyVision
        <https://github.com/facebookresearch/ClassyVision/blob/main/classy_vision/
↪losses/label_smoothing_loss.py>`_ repo as:

        .. math::
            \frac{\delta_{k, y} + \epsilon/K}{1+\epsilon}

        if the ``mode`` is "multi_label", this will accept labels from
        multi-label task and smoothing them as:

        .. math::
            (1-2\epsilon)\delta_{k, y} + \epsilon
```

注解： 注意 “here“、‘here‘、” here” 三种引号功能是不同。

在 reStructured 语法中，“here“ 表示一段代码；‘here‘ 表示斜体；” here” 无特殊含义，一般可用来表示字符串。其中 ‘here‘ 的用法与 Markdown 中不同，需要多加留意。另外还有:obj:‘type‘ 这种更规范的表示类的写法，但鉴于长度，不做特别要求，一般仅用于表示非常用类型。

3. 方法（函数）文档

函数文档与类文档的结构基本一致，但需要加入返回值文档。对于较为复杂的函数和类，可以使用

Examples 字段加入示例；如果需要对参数加入一些较长的备注，可以加入 Note 字段进行说明。

对于使用较为复杂的类或函数，比起看大段大段的说明文字和参数文档，添加合适的示例更能帮助用户迅速了解其用法。需要注意的是，这些示例最好是能够直接在 Python 交互式环境中运行的，并给出一些相对应的结果。如果存在多个示例，可以使用注释简单说明每段示例，也能起到分隔作用。

```
def import_modules_from_strings(imports, allow_failed_imports=False):
    """Import modules from the given list of strings.

    Args:
        imports (list | str | None): The given module names to be imported.
        allow_failed_imports (bool): If True, the failed imports will return
            None. Otherwise, an ImportError is raise. Defaults to False.

    Returns:
        List[module] | module | None: The imported modules.
        All these three lines in docstring will be compiled into the same
        line in readthedocs.

    Examples:
        >>> osp, sys = import_modules_from_strings(
        ...     ['os.path', 'sys'])
        >>> import os.path as osp_
        >>> import sys as sys_
        >>> assert osp == osp_
        >>> assert sys == sys_
    """
    ...
```

如果函数接口在某个版本发生了变化，需要在 docstring 中加入相关的说明，必要时添加 Note 或者 Warning 进行说明，例如：

```
class CheckpointHook(Hook):
    """Save checkpoints periodically.

    Args:
        out_dir (str, optional): The root directory to save checkpoints. If
            not specified, ``runner.work_dir`` will be used by default. If
            specified, the ``out_dir`` will be the concatenation of
            ``out_dir`` and the last level directory of ``runner.work_dir``.
            Defaults to None. `Changed in version 1.3.15.`

    Warning:
        Before v1.3.15, the ``out_dir`` argument indicates the path where the
        checkpoint is stored. However, in v1.3.15 and later, ``out_dir``
```

(下页继续)

(续上页)

```
        indicates the root directory and the final path to save checkpoint is
        the concatenation of out_dir and the last level directory of
        ``runner.work_dir``. Suppose the value of ``out_dir`` is
        "/path/of/A" and the value of ``runner.work_dir`` is "/path/of/B",
        then the final path will be "/path/of/A/B".
```

如果参数或返回值里带有需要展开描述字段的 dict，则应该采用如下格式：

```
def func(x):
    r"""
    Args:
        x (None): A dict with 2 keys, ``padded_targets``, and ``targets``.

            - ``targets`` (list[Tensor]): A list of tensors.
              Each tensor has the shape of :math:`(T_i)`. Each
              element is the index of a character.
            - ``padded_targets`` (Tensor): A tensor of shape :math:`(N)`.
              Each item is the length of a word.

    Returns:
        dict: A dict with 2 keys, ``padded_targets``, and ``targets``.

        - ``targets`` (list[Tensor]): A list of tensors.
          Each tensor has the shape of :math:`(T_i)`. Each
          element is the index of a character.
        - ``padded_targets`` (Tensor): A tensor of shape :math:`(N)`.
          Each item is the length of a word.
    """
    return x
```

重要： 为了生成 readthedocs 文档，文档的编写需要按照 ReStructrued 文档格式，否则会产生文档渲染错误，在提交 PR 前，最好生成并预览一下文档效果。语法规范参考：

- reStructuredText Primer - Sphinx documentation
- Example Google Style Python Docstrings – napoleon 0.7 documentation

58.4 注释规范

58.4.1 为什么要写注释

对于一个开源项目，团队合作以及社区之间的合作是必不可少的，因而尤其要重视合理的注释。不写注释的代码，很有可能过几个月自己也难以理解，造成额外的阅读和修改成本。

58.4.2 如何写注释

最需要写注释的是代码中那些技巧性的部分。如果你在下次代码审查的时候必须解释一下，那么你应该现在就给它写注释。对于复杂的操作，应该在其操作开始前写上若干行注释。对于不是一目了然的代码，应在其行尾添加注释。——Google 开源项目风格指南

```
# We use a weighted dictionary search to find out where i is in
# the array. We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.
if i & (i-1) == 0:  # True if i is 0 or a power of 2.
```

为了提高可读性, 注释应该至少离开代码 2 个空格. 另一方面, 绝不要描述代码. 假设阅读代码的人比你更懂 Python, 他只是不知道你的代码要做什么. ——Google 开源项目风格指南

```
# Wrong:
# Now go through the b array and make sure whenever i occurs
# the next element is i+1

# Wrong:
if i & (i-1) == 0:  # True if i bitwise and i-1 is 0.
```

在注释中，可以使用 Markdown 语法，因为开发人员通常熟悉 Markdown 语法，这样可以便于交流理解，如可使用单反引号表示代码和变量（注意不要和 docstring 中的 ReStructured 语法混淆）

```
# `_reversed_padding_repeated_twice` is the padding to be passed to
# `F.pad` if needed (e.g., for non-zero padding types that are
# implemented as two ops: padding + conv). `F.pad` accepts paddings in
# reverse order than the dimension.
self._reversed_padding_repeated_twice = _reverse_repeat_tuple(self.padding, 2)
```

58.4.3 注释示例

1. 出自 mmcv/utils/registry.py，对于较为复杂的逻辑结构，通过注释，明确了优先级关系。

```
# self.build_func will be set with the following priority:
# 1. build_func
# 2. parent.build_func
# 3. build_from_cfg
if build_func is None:
    if parent is not None:
        self.build_func = parent.build_func
    else:
        self.build_func = build_from_cfg
else:
    self.build_func = build_func
```

2. 出自 mmcv/runner/checkpoint.py，对于 bug 修复中的一些特殊处理，可以附带相关的 issue 链接，帮助其他人了解 bug 背景。

```
def _save_ckpt(checkpoint, file):
    # The 1.6 release of PyTorch switched torch.save to use a new
    # zipfile-based file format. It will cause RuntimeError when a
    # checkpoint was saved in high version (PyTorch version>=1.6.0) but
    # loaded in low version (PyTorch version<1.6.0). More details at
    # https://github.com/open-mmlab/mmpose/issues/904
    if digit_version(TORCH_VERSION) >= digit_version('1.6.0'):
        torch.save(checkpoint, file, _use_new_zipfile_serialization=False)
    else:
        torch.save(checkpoint, file)
```

58.5 类型注解

58.5.1 为什么要写类型注解

类型注解是对函数中变量的类型做限定或提示，为代码的安全性提供保障、增强代码的可读性、避免出现类型相关的错误。Python 没有对类型做强制限制，类型注解只起到一个提示作用，通常你的 IDE 会解析这些类型注解，然后在你调用相关代码时对类型做提示。另外也有类型注解检查工具，这些工具会根据类型注解，对代码中可能出现的问题进行检查，减少 bug 的出现。需要注意的是，通常我们不需要注释模块中的所有函数：

1. 公共的 API 需要注释
2. 在代码的安全性，清晰性和灵活性上进行权衡是否注释

3. 对于容易出现类型相关的错误的代码进行注释
4. 难以理解的代码请进行注释
5. 若代码中的类型已经稳定，可以进行注释. 对于一份成熟的代码，多数情况下，即使注释了所有的函数，也不会丧失太多的灵活性.

58.5.2 如何写类型注解

1. 函数 / 方法类型注解，通常不对 self 和 cls 注释。

```
from typing import Optional, List, Tuple

# 全部位于一行
def my_method(self, first_var: int) -> int:
    pass

# 另起一行
def my_method(
        self, first_var: int,
        second_var: float) -> Tuple[MyLongType1, MyLongType1, MyLongType1]:
    pass

# 单独成行（具体的应用场合与行宽有关，建议结合 yapf 自动化格式使用）
def my_method(
    self, first_var: int, second_var: float
) -> Tuple[MyLongType1, MyLongType1, MyLongType1]:
    pass

# 引用尚未被定义的类型
class MyClass:
    def __init__(self,
                 stack: List["MyClass"]) -> None:
        pass
```

注：类型注解中的类型可以是 Python 内置类型，也可以是自定义类，还可以使用 Python 提供的 wrapper 类对类型注解进行装饰，一些常见的注解如下：

```
# 数值类型
from numbers import Number

# 可选类型，指参数可以为 None
from typing import Optional
def foo(var: Optional[int] = None):
    pass
```

(下页继续)

(续上页)

```
# 联合类型，指同时接受多种类型
from typing import Union
def foo(var: Union[float, str]):
    pass

from typing import Sequence  # 序列类型
from typing import Iterable  # 可迭代类型
from typing import Any  # 任意类型
from typing import Callable  # 可调用类型

from typing import List, Dict  # 列表和字典的泛型类型
from typing import Tuple  # 元组的特殊格式
# 虽然在 Python 3.9 中，list, tuple 和 dict 本身已支持泛型，但为了支持之前的版本
# 我们在进行类型注解时还是需要使用 List, Tuple, Dict 类型
# 另外，在对参数类型进行注解时，尽量使用 Sequence & Iterable & Mapping
# List, Tuple, Dict 主要用于返回值类型注解
# 参见 https://docs.python.org/3/library/typing.html#typing.List
```

2. 变量类型注解，一般用于难以直接推断其类型时

```
# Recommend: 带类型注解的赋值
a: Foo = SomeUndecoratedFunction()
a: List[int]: [1, 2, 3]          # List 只支持单一类型泛型，可使用 Union
b: Tuple[int, int] = (1, 2)      # 长度固定为 2
c: Tuple[int, ...] = (1, 2, 3)  # 变长
d: Dict[str, int] = {'a': 1, 'b': 2}

# Not Recommend: 行尾类型注释
# 虽然这种方式被写在了 Google 开源指南中，但这是一种为了支持 Python 2.7 版本
# 而补充的注释方式，鉴于我们只支持 Python 3，为了风格统一，不推荐使用这种方式。
a = SomeUndecoratedFunction()  # type: Foo
a = [1, 2, 3]  # type: List[int]
b = (1, 2, 3)  # type: Tuple[int, ...]
c = (1, "2", 3.5)  # type: Tuple[int, Text, float]
```

3. 泛型

   上文中我们知道，typing 中提供了 list 和 dict 的泛型类型，那么我们自己是否可以定义类似的泛型呢?

```
from typing import TypeVar, Generic

KT = TypeVar('KT')
VT = TypeVar('VT')
```

(下页继续)

(续上页)

```
class Mapping(Generic[KT, VT]):
    def __init__(self, data: Dict[KT, VT]):
        self._data = data

    def __getitem__(self, key: KT) -> VT:
        return self._data[key]
```

使用上述方法，我们定义了一个拥有泛型能力的映射类，实际用法如下：

```
mapping = Mapping[str, float]({'a': 0.5})
value: float = example['a']
```

另外，我们也可以利用 TypeVar 在函数签名中指定联动的多个类型：

```
from typing import TypeVar, List

T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes


def repeat(x: T, n: int) -> List[T]:
    """Return a list containing n references to x."""
    return [x]*n


def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

更多关于类型注解的写法请参考 typing。

58.5.3 类型注解检查工具

mypy 是一个 Python 静态类型检查工具。根据你的类型注解，mypy 会检查传参、赋值等操作是否符合类型注解，从而避免可能出现的 bug。

例如如下的一个 Python 脚本文件 test.py:

```
def foo(var: int) -> float:
    return float(var)

a: str = foo('2.0')
b: int = foo('3.0')  # type: ignore
```

运行 mypy test.py 可以得到如下检查结果，分别指出了第 4 行在函数调用和返回值赋值两处类型错误。而第 5 行同样存在两个类型错误，由于使用了 type: ignore 而被忽略了，只有部分特殊情况可能需要此类忽略。

```
test.py:4: error: Incompatible types in assignment (expression has type "float",␣
↪variable has type "int")
test.py:4: error: Argument 1 to "foo" has incompatible type "str"; expected "int"
Found 2 errors in 1 file (checked 1 source file)
```

CHAPTER 59

mmyolo.datasets

59.1 datasets

class mmyolo.datasets.**BatchShapePolicy**(*batch_size: int = 32*, *img_size: int = 640*, *size_divisor: int = 32*, *extra_pad_ratio: float = 0.5*)

BatchShapePolicy is only used in the testing phase, which can reduce the number of pad pixels during batch inference.

参数

- **batch_size** (*int*) –Single GPU batch size during batch inference. Defaults to 32.
- **img_size** (*int*) –Expected output image size. Defaults to 640.
- **size_divisor** (*int*) –The minimum size that is divisible by size_divisor. Defaults to 32.
- **extra_pad_ratio** (*float*) –Extra pad ratio. Defaults to 0.5.

class mmyolo.datasets.**YOLOv5CocoDataset**(**args*, *batch_shapes_cfg: Optional[dict] = None*, ***kwargs*)

Dataset for YOLOv5 COCO Dataset.

We only add *BatchShapePolicy* function compared with CocoDataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

class mmyolo.datasets.**YOLOv5CrowdHumanDataset**(**args*, *batch_shapes_cfg: Optional[dict] = None*, ***kwargs*)

Dataset for YOLOv5 CrowdHuman Dataset.

We only add *BatchShapePolicy* function compared with CrowdHumanDataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

class mmyolo.datasets.**YOLOv5DOTADataset**(**args*, ***kwargs*)

Dataset for YOLOv5 DOTA Dataset.

We only add *BatchShapePolicy* function compared with DOTADataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

class mmyolo.datasets.**YOLOv5VOCDataset**(**args*, *batch_shapes_cfg: Optional[dict] = None*, ***kwargs*)

Dataset for YOLOv5 VOC Dataset.

We only add *BatchShapePolicy* function compared with VOCDataset. See *mmyolo/datasets/utils.py#BatchShapePolicy* for details

mmyolo.datasets.**yolov5_collate**(*data_batch: Sequence*, *use_ms_training: bool = False*) → dict

Rewrite collate_fn to get faster training speed.

参数

- **data_batch** (*Sequence*) –Batch of data.
- **use_ms_training** (*bool*) –Whether to use multi-scale training.

59.2 transforms

class mmyolo.datasets.transforms.**FilterAnnotations**(*by_keypoints: bool = False*, ***kwargs*)

Filter invalid annotations.

In addition to the conditions checked by `FilterDetAnnotations`, this filter adds a new condition requiring instances to have at least one visible keypoints.

class mmyolo.datasets.transforms.**LetterResize**(*scale: Union[int, Tuple[int, int]]*, *pad_val: dict = {'img': 0, 'mask': 0, 'seg': 255}*, *use_mini_pad: bool = False*, *stretch_only: bool = False*, *allow_scale_up: bool = True*, *half_pad_param: bool = False*, ***kwargs*)

Resize and pad image while meeting stride-multiple constraints.

Required Keys:

- img (np.uint8)
- batch_shape (np.int64) (optional)

Modified Keys:

- img (np.uint8)
- img_shape (tuple)

- gt_bboxes (optional)

Added Keys: - pad_param (np.float32)

参数

- **scale** (*Union[int, Tuple[int, int]]*) –Images scales for resizing.
- **pad_val** (*dict*) –Padding value. Defaults to dict(img=0, seg=255).
- **use_mini_pad** (*bool*) –Whether using minimum rectangle padding. Defaults to True
- **stretch_only** (*bool*) –Whether stretch to the specified size directly. Defaults to False
- **allow_scale_up** (*bool*) –Allow scale up when ratio > 1. Defaults to True
- **half_pad_param** (*bool*) –If set to True, left and right pad_param will be given by dividing padding_h by 2. If set to False, pad_param is in int format. We recommend setting this to False for object detection tasks, and True for instance segmentation tasks. Default to False.

transform(*results: dict*) → dict

Transform function to resize images, bounding boxes, semantic segmentation map and keypoints.

参数 **results** (*dict*) –Result dict from loading pipeline.

返回 Resized results, 'img', 'gt_bboxes', 'gt_seg_map', 'gt_keypoints', 'scale', 'scale_factor', 'img_shape', and 'keep_ratio' keys are updated in result dict.

返回类型 dict

class mmyolo.datasets.transforms.**LoadAnnotations**(*mask2bbox: bool = False*, *poly2mask: bool = False*, *merge_polygons: bool = True*, ***kwargs*)

Because the yolo series does not need to consider ignore bboxes for the time being, in order to speed up the pipeline, it can be excluded in advance.

参数

- **mask2bbox** (*bool*) –Whether to use mask annotation to get bbox. Defaults to False.
- **poly2mask** (*bool*) –Whether to transform the polygons to bitmaps. Defaults to False.
- **merge_polygons** (*bool*) –Whether to merge polygons into one polygon. If merged, the storage structure is simpler and training is more effcient, especially if the mask inside a bbox is divided into multiple polygons. Defaults to True.

merge_multi_segment(*gt_masks: List[numpy.ndarray]*) → List[numpy.ndarray]

Merge multi segments to one list.

Find the coordinates with min distance between each segment, then connect these coordinates with one thin line to merge all segments into one. :param gt_masks: original segmentations in coco's json file.

like [segmentation1, segmentation2,⋯], each segmentation is a list of coordinates.

返回 merged gt_masks

返回类型 gt_masks(List(np.array))

min_index(*arr1: numpy.ndarray*, *arr2: numpy.ndarray*) → Tuple[int, int]

Find a pair of indexes with the shortest distance.

参数

- **arr1** –(N, 2).
- **arr2** –(M, 2).

返回 a pair of indexes.

返回类型 tuple

transform(*results: dict*) → dict

Function to load multiple types annotations.

参数 **results** (*dict*) –Result dict from :obj:mmengine.BaseDataset.

返回 The dict contains loaded bounding box, label and semantic segmentation.

返回类型 dict

class mmyolo.datasets.transforms.**Mosaic**(*img_scale: Tuple[int, int] = (640, 640)*, *center_ratio_range: Tuple[float, float] = (0.5, 1.5)*, *bbox_clip_border: bool = True*, *pad_val: float = 114.0*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 40*, *random_pop: bool = True*, *max_refetch: int = 15*)

Mosaic augmentation.

Given 4 images, mosaic transform combines them into one output image. The output image is composed of the parts from each sub- image.

```
                  mosaic transform
                     center_x
           +------------------------------+
           |       pad        |           |
           |      +-----------+    pad    |
           |      |           |           |
           |      |  image1   +-----------+
           |      |           |           |
           |      |           |   image2  |
center_y   |----+-+-----------+-----------+
           |    |   cropped   |           |
           |pad |   image3    |   image4  |
```

(下页继续)

(续上页)

```
          |    |             |           |
          +----|-------------+-----------+
               |             |
               +-------------+

The mosaic transform steps are as follows:

    1. Choose the mosaic center as the intersections of 4 images
    2. Get the left top image according to the index, and randomly
       sample another 3 images from the custom dataset.
    3. Sub image will be cropped if image is larger than mosaic patch
```

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

参数

- **img_scale** (*Sequence[int]*) –Image size after mosaic pipeline of single image. The shape order should be (width, height). Defaults to (640, 640).
- **center_ratio_range** (*Sequence[float]*) –Center ratio range of mosaic output. Defaults to (0.5, 1.5).
- **bbox_clip_border** (*bool, optional*) –Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **pad_val** (*int*) –Pad value. Defaults to 114.
- **pre_transform** (*Sequence[dict]*) –Sequence of transform object or config dict to be composed.

- **prob** (*float*) –Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) –Whether to use cache. Defaults to False.
- **max_cached_images** (*int*) –The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 10 caches for each image suffices for randomness. Defaults to 40.
- **random_pop** (*bool*) –Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) –The maximum number of retry iterations for getting valid results from the pipeline. If the number of iterations is greater than *max_refetch*, but results is still None, then the iteration is terminated and raise the error. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → list

Call function to collect indexes.

参数 **dataset** (Dataset or list) –The dataset or cached list.

返回 indexes.

返回类型 list

mix_img_transform(*results: dict*) → dict

Mixed image data transformation.

参数 **results** (*dict*) –Result dict.

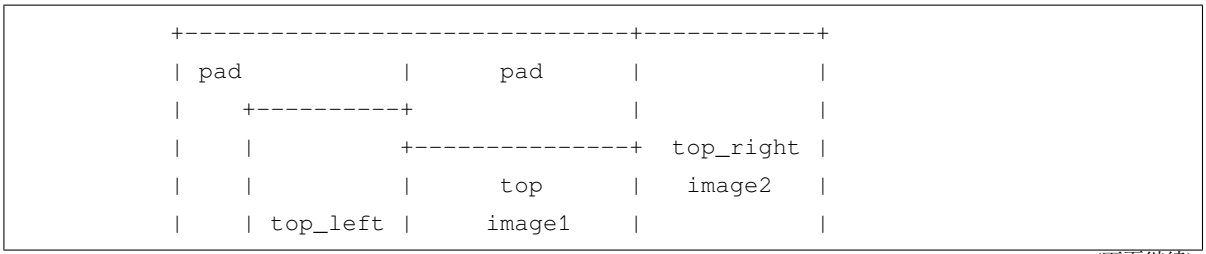
返回 Updated result dict.

返回类型 results (dict)

class mmyolo.datasets.transforms.**Mosaic9**(*img_scale: Tuple[int, int] = (640, 640)*, *bbox_clip_border: bool = True*, *pad_val: Union[float, int] = 114.0*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 50*, *random_pop: bool = True*, *max_refetch: int = 15*)

Mosaic9 augmentation.

Given 9 images, mosaic transform combines them into one output image. The output image is composed of the parts from each sub- image.



(下页继续)

(续上页)

```
          |    |  image8  o--------+------+--------+---+
          |    |          |        |                |   |
          +----+----------+        |     right      |pad|
          |               | center |     image3     |   |
          |     left      | image0 +----------------+---|
          |    image7     |        |                |   |
      +---+-----------+---+--------+                |   |
      |   |  cropped  |            |  bottom_right |pad|
      |   |bottom_left|            |    image4      |   |
      |   |  image6   |   bottom   |                |   |
      +---|-----------+   image5   +----------------+---|
          |    pad    |            |        pad         |
          +-----------+------------+--------------------+

The mosaic transform steps are as follows:

    1. Get the center image according to the index, and randomly
       sample another 8 images from the custom dataset.
    2. Randomly offset the image after Mosaic
```

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

参数

- **img_scale** (*Sequence[int]*) –Image size after mosaic pipeline of single image. The shape order should be (width, height). Defaults to (640, 640).

- **bbox_clip_border** (*bool, optional*) –Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **pad_val** (*int*) –Pad value. Defaults to 114.
- **pre_transform** (*Sequence[dict]*) –Sequence of transform object or config dict to be composed.
- **prob** (*float*) –Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) –Whether to use cache. Defaults to False.
- **max_cached_images** (*int*) –The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 5 caches for each image suffices for randomness. Defaults to 50.
- **random_pop** (*bool*) –Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) –The maximum number of retry iterations for getting valid results from the pipeline. If the number of iterations is greater than *max_refetch*, but results is still None, then the iteration is terminated and raise the error. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → list

Call function to collect indexes.

参数 **dataset** (Dataset or list) –The dataset or cached list.

返回 indexes.

返回类型 list

mix_img_transform(*results: dict*) → dict

Mixed image data transformation.

参数 **results** (*dict*) –Result dict.

返回 Updated result dict.

返回类型 results (dict)

class mmyolo.datasets.transforms.**PPYOLOERandomCrop**(*aspect_ratio: List[float] = [0.5, 2.0]*, *thresholds: List[float] = [0.0, 0.1, 0.3, 0.5, 0.7, 0.9]*, *scaling: List[float] = [0.3, 1.0]*, *num_attempts: int = 50*, *allow_no_crop: bool = True*, *cover_all_box: bool = False*)

Random crop the img and bboxes. Different thresholds are used in PPYOLOE to judge whether the clipped image meets the requirements. This implementation is different from the implementation of RandomCrop in mmdet.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

Added Keys: - pad_param (np.float32)

参数

- **aspect_ratio** (*List[float]*) –Aspect ratio of cropped region. Default to [.5, 2].
- **thresholds** (*List[float]*) –Iou thresholds for deciding a valid bbox crop in [min, max] format. Defaults to [.0, .1, .3, .5, .7, .9].
- **scaling** (*List[float]*) –Ratio between a cropped region and the original image in [min, max] format. Default to [.3, 1.].
- **num_attempts** (*int*) –Number of tries for each threshold before giving up. Default to 50.
- **allow_no_crop** (*bool*) –Allow return without actually cropping them. Default to True.
- **cover_all_box** (*bool*) –Ensure all bboxes are covered in the final crop. Default to False.

class mmyolo.datasets.transforms.**PPYOLOERandomDistort**(*hue_cfg: dict = {'max': 18, 'min': -18, 'prob': 0.5}*, *saturation_cfg: dict = {'max': 1.5, 'min': 0.5, 'prob': 0.5}*, *contrast_cfg: dict = {'max': 1.5, 'min': 0.5, 'prob': 0.5}*, *brightness_cfg: dict = {'max': 1.5, 'min': 0.5, 'prob': 0.5}*, *num_distort_func: int = 4*)

Random hue, saturation, contrast and brightness distortion.

Required Keys:

- img

Modified Keys:

- img (np.float32)

参数

- **hue_cfg** (*dict*) –Hue settings. Defaults to dict(min=-18, max=18, prob=0.5).
- **saturation_cfg** (*dict*) –Saturation settings. Defaults to dict(min=0.5, max=1.5, prob=0.5).
- **contrast_cfg** (*dict*) –Contrast settings. Defaults to dict(min=0.5, max=1.5, prob=0.5).
- **brightness_cfg** (*dict*) –Brightness settings. Defaults to dict(min=0.5, max=1.5, prob=0.5).
- **num_distort_func** (*int*) –The number of distort function. Defaults to 4.

transform(*results: dict*) → dict

The hue, saturation, contrast and brightness distortion function.

参数 **results** (*dict*) –The result dict.

返回 The result dict.

返回类型 dict

transform_brightness(*results*)

Transform brightness randomly.

transform_contrast(*results*)

Transform contrast randomly.

transform_hue(*results*)

Transform hue randomly.

transform_saturation(*results*)

Transform saturation randomly.

class mmyolo.datasets.transforms.**PackDetInputs**(*meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape', 'scale_factor', 'flip', 'flip_direction')*)

Pack the inputs data for the detection / semantic segmentation / panoptic segmentation.

Compared to mmdet, we just add the *gt_panoptic_seg* field and logic.

transform(*results: dict*) → dict

Method to pack the input data. :param results: Result dict from the data pipeline. :type results: dict

返回

- ‘inputs’ (obj:*torch.Tensor*): The forward data of models.
- **’ data_sample’ (obj:*DetDataSample*): The annotation info of the** sample.

返回类型 dict

class mmyolo.datasets.transforms.**Polygon2Mask**(*downsample_ratio: int = 4*, *mask_overlap: bool = True*, *coco_style: bool = False*)

Polygons to bitmaps in YOLOv5.

参数

- **downsample_ratio** (*int*) –Downsample ratio of mask.
- **mask_overlap** (*bool*) –Whether to use maskoverlap in mask process. When set to True, the implementation here is the same as the official, with higher training speed. If set to True, all gt masks will compress into one overlap mask, the value of mask indicates the index of gt masks. If set to False, one mask is a binary mask. Default to True.
- **coco_style** (*bool*) –Whether to use coco_style to convert the polygons to bitmaps. Note that this option is only used to test if there is an improvement in training speed and we recommend setting it to False.

polygon2mask(*img_shape: Tuple[int, int]*, *polygons: numpy.ndarray*, *color: int = 1*) → numpy.ndarray

参数

- **img_shape** (*tuple*) –The image size.
- **polygons** (*np.ndarray*) –[N, M], N is the number of polygons, M is the number of points(Be divided by 2).
- **color** (*int*) –color in fillPoly.

返回 the overlap mask.

返回类型 np.ndarray

polygons2masks(*img_shape: Tuple[int, int]*, *polygons: mmdet.structures.mask.structures.PolygonMasks*, *color: int = 1*) → numpy.ndarray

Return a list of bitmap masks.

参数

- **img_shape** (*tuple*) –The image size.
- **polygons** (*PolygonMasks*) –The mask annotations.
- **color** (*int*) –color in fillPoly.

返回 the list of masks in bitmaps.

返回类型 List[np.ndarray]

polygons2masks_overlap(*img_shape: Tuple[int, int]*, *polygons: mmdet.structures.mask.structures.PolygonMasks*) → Tuple[numpy.ndarray, numpy.ndarray]

Return a overlap mask and the sorted idx of area.

参数

- **img_shape** (*tuple*) –The image size.
- **polygons** (*PolygonMasks*) –The mask annotations.
- **color** (*int*) –color in fillPoly.

返回 the overlap mask and the sorted idx of area.

返回类型 Tuple[np.ndarray, np.ndarray]

transform(*results: dict*) → dict

The transform function. All subclass of BaseTransform should override this method.

This function takes the result dict as the input, and can add new items to the dict or modify existing items in the dict. And the result dict will be returned in the end, which allows to concate multiple transforms into a pipeline.

参数 **results** (*dict*) –The result dict.

返回 The result dict.

返回类型 dict

class mmyolo.datasets.transforms.**RandomAffine**(***kwargs*)

class mmyolo.datasets.transforms.**RandomFlip**(*prob: Optional[Union[float, Iterable[float]]] = None*, *direction: Union[str, Sequence[Optional[str]]] = 'horizontal'*, *swap_seg_labels: Optional[Sequence] = None*)

class mmyolo.datasets.transforms.**RegularizeRotatedBox**(*angle_version='le90'*)

Regularize rotated boxes.

Due to the angle periodicity, one rotated box can be represented in many different (x, y, w, h, t). To make each rotated box unique, `regularize_boxes` will take the remainder of the angle divided by 180 degrees.

For convenience, three angle_version can be used here:

- **'oc': OpenCV Definition. Has the same box representation as** `cv2.minAreaRect` the angle ranges in [-90, 0).
- **'le90': Long Edge Definition (90). the angle ranges in [-90, 90).** The width is always longer than the height.
- **'le135': Long Edge Definition (135). the angle ranges in [-45, 135).** The width is always longer than the height.

Required Keys:

- gt_bboxes (RotatedBoxes[torch.float32])

Modified Keys:

- gt_bboxes

参数 **angle_version** (*str*) –Angle version. Can only be ‘oc’, ‘le90’, or ‘le135’. Defaults to ‘le90.

transform (*results: dict*) → dict

The transform function. All subclass of BaseTransform should override this method.

This function takes the result dict as the input, and can add new items to the dict or modify existing items in the dict. And the result dict will be returned in the end, which allows to concate multiple transforms into a pipeline.

参数 **results** (*dict*) –The result dict.

返回 The result dict.

返回类型 dict

class mmyolo.datasets.transforms.**RemoveDataElement** (*keys: Union[str, Sequence[str]]*)

Remove unnecessary data element in results.

参数 **keys** (*Union[str, Sequence[str]]*) –Keys need to be removed.

transform (*results: dict*) → dict

The transform function. All subclass of BaseTransform should override this method.

This function takes the result dict as the input, and can add new items to the dict or modify existing items in the dict. And the result dict will be returned in the end, which allows to concate multiple transforms into a pipeline.

参数 **results** (*dict*) –The result dict.

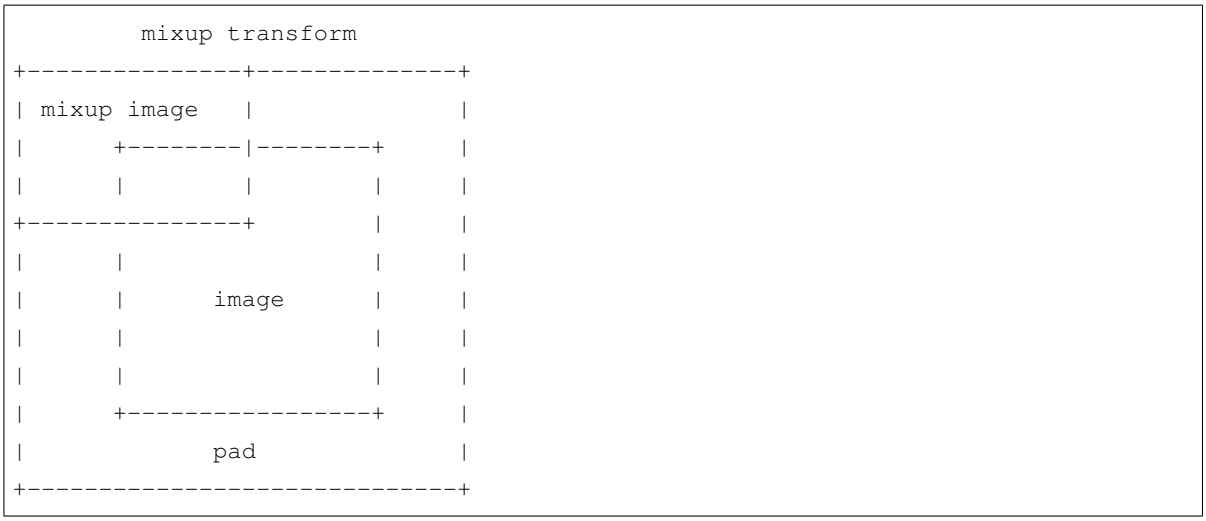
返回 The result dict.

返回类型 dict

class mmyolo.datasets.transforms.**Resize** (*scale: Optional[Union[int, Tuple[int, int]]] = None*, *scale_factor: Optional[Union[float, Tuple[float, float]]] = None*, *keep_ratio: bool = False*, *clip_object_border: bool = True*, *backend: str = 'cv2'*, *interpolation='bilinear'*)

class mmyolo.datasets.transforms.**YOLOXMixUp** (*img_scale: Tuple[int, int] = (640, 640)*, *ratio_range: Tuple[float, float] = (0.5, 1.5)*, *flip_ratio: float = 0.5*, *pad_val: float = 114.0*, *bbox_clip_border: bool = True*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 20*, *random_pop: bool = True*, *max_refetch: int = 15*)

MixUp data augmentation for YOLOX.



The mixup transform steps are as follows:

1. Another random image is picked by dataset and embedded in the top left patch(after padding and resizing)
2. The target of mixup transform is the weighted average of mixup image and origin image.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

参数

- **img_scale** (*Sequence[int]*) –Image output size after mixup pipeline. The shape order should be (width, height). Defaults to (640, 640).
- **ratio_range** (*Sequence[float]*) –Scale ratio of mixup image. Defaults to (0.5, 1.5).
- **flip_ratio** (*float*) –Horizontal flip ratio of mixup image. Defaults to 0.5.
- **pad_val** (*int*) –Pad value. Defaults to 114.

- **bbox_clip_border** (*bool, optional*) –Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **pre_transform** (*Sequence[dict]*) –Sequence of transform object or config dict to be composed.
- **prob** (*float*) –Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) –Whether to use cache. Defaults to False.
- **max_cached_images** (*int*) –The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 10 caches for each image suffices for randomness. Defaults to 20.
- **random_pop** (*bool*) –Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) –The maximum number of iterations. If the number of iterations is greater than *max_refetch*, but gt_bbox is still empty, then the iteration is terminated. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → int

Call function to collect indexes.

参数 **dataset** (Dataset or list) –The dataset or cached list.

返回 indexes.

返回类型 int

mix_img_transform(*results: dict*) → dict

YOLOX MixUp transform function.

参数 **results** (*dict*) –Result dict.

返回 Updated result dict.

返回类型 results (dict)

class mmyolo.datasets.transforms.**YOLOv5CopyPaste**(*ioa_thresh: float = 0.3*, *prob: float = 0.5*)

Copy-Paste used in YOLOv5 and YOLOv8.

This transform randomly copy some objects in the image to the mirror position of the image.It is different from the *CopyPaste* in mmdet.

Required Keys:

- img (np.uint8)
- gt_bboxes (BaseBoxes[torch.float32])
- gt_bboxes_labels (np.int64) (optional)

- gt_ignore_flags (bool) (optional)
- gt_masks (PolygonMasks) (optional)

Modified Keys:

- img
- gt_bboxes
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (optional)
- gt_masks (optional)

参数

- **ioa_thresh** (*float*) –Ioa thresholds for deciding valid bbox.
- **prob** (*float*) –Probability of choosing objects. Defaults to 0.5.

static bbox_ioa(*gt_bboxes_flip: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*, *gt_bboxes: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*, *eps: float = 1e-07*) → numpy.ndarray

Calculate ioa between gt_bboxes_flip and gt_bboxes.

参数

- **gt_bboxes_flip** (*HorizontalBoxes*) –Flipped ground truth bounding boxes.
- **gt_bboxes** (*HorizontalBoxes*) –Ground truth bounding boxes.
- **eps** (*float*) –Default to 1e-10.

返回 Ioa.

返回类型 (Tensor)

class mmyolo.datasets.transforms.**YOLOv5HSVRandomAug**(*hue_delta: Union[int, float] = 0.015*, *saturation_delta: Union[int, float] = 0.7*, *value_delta: Union[int, float] = 0.4*)

Apply HSV augmentation to image sequentially.

Required Keys:

- img

Modified Keys:

- img

参数

- **hue_delta** (*[int, float]*) –delta of hue. Defaults to 0.015.
- **saturation_delta** (*[int, float]*) –delta of saturation. Defaults to 0.7.
- **value_delta** (*[int, float]*) –delta of value. Defaults to 0.4.

transform(*results: dict*) → dict

The HSV augmentation transform function.

参数 **results** (*dict*) –The result dict.

返回 The result dict.

返回类型 dict

class mmyolo.datasets.transforms.**YOLOv5KeepRatioResize**(*scale: Union[int, Tuple[int, int]]*, *keep_ratio: bool = True*, ***kwargs*)

Resize images & bbox(if existed).

This transform resizes the input image according to `scale`. Bboxes (if existed) are then resized with the same scale factor.

Required Keys:

- img (np.uint8)
- gt_bboxes (BaseBoxes[torch.float32]) (optional)

Modified Keys:

- img (np.uint8)
- img_shape (tuple)
- gt_bboxes (optional)
- scale (float)

Added Keys:

- scale_factor (np.float32)

参数 **scale** (*Union[int, Tuple[int, int]]*) –Images scales for resizing.

class mmyolo.datasets.transforms.**YOLOv5MixUp**(*alpha: float = 32.0*, *beta: float = 32.0*, *pre_transform: Optional[Sequence[dict]] = None*, *prob: float = 1.0*, *use_cached: bool = False*, *max_cached_images: int = 20*, *random_pop: bool = True*, *max_refetch: int = 15*)

MixUp data augmentation for YOLOv5.

The mixup transform steps are as follows:

1. Another random image is picked by dataset.
2. **Randomly obtain the fusion ratio from the beta distribution,** then fuse the target

of the original image and mixup image through this ratio.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- mix_results (List[dict])

Modified Keys:

- img
- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)

参数

- **alpha** (*float*) –parameter of beta distribution to get mixup ratio. Defaults to 32.
- **beta** (*float*) –parameter of beta distribution to get mixup ratio. Defaults to 32.
- **pre_transform** (*Sequence[dict]*) –Sequence of transform object or config dict to be composed.
- **prob** (*float*) –Probability of applying this transformation. Defaults to 1.0.
- **use_cached** (*bool*) –Whether to use cache. Defaults to False.
- **max_cached_images** (*int*) –The maximum length of the cache. The larger the cache, the stronger the randomness of this transform. As a rule of thumb, providing 10 caches for each image suffices for randomness. Defaults to 20.
- **random_pop** (*bool*) –Whether to randomly pop a result from the cache when the cache is full. If set to False, use FIFO popping method. Defaults to True.
- **max_refetch** (*int*) –The maximum number of iterations. If the number of iterations is greater than *max_refetch*, but gt_bbox is still empty, then the iteration is terminated. Defaults to 15.

get_indexes(*dataset: Union[mmengine.dataset.base_dataset.BaseDataset, list]*) → int

Call function to collect indexes.

参数 **dataset** (Dataset or list) –The dataset or cached list.

返回 indexes.

返回类型 int

mix_img_transform(*results: dict*) → dict

YOLOv5 MixUp transform function.

参数 **results** (*dict*) –Result dict

返回 Updated result dict.

返回类型 results (dict)

class mmyolo.datasets.transforms.**YOLOv5RandomAffine**(*max_rotate_degree: float = 10.0*, *max_translate_ratio: float = 0.1*, *scaling_ratio_range: Tuple[float, float] = (0.5, 1.5)*, *max_shear_degree: float = 2.0*, *border: Tuple[int, int] = (0, 0)*, *border_val: Tuple[int, int, int] = (114, 114, 114)*, *bbox_clip_border: bool = True*, *min_bbox_size: int = 2*, *min_area_ratio: float = 0.1*, *use_mask_refine: bool = False*, *max_aspect_ratio: float = 20.0*, *resample_num: int = 1000*)

Random affine transform data augmentation in YOLOv5 and YOLOv8. It is different from the implementation in YOLOX.

This operation randomly generates affine transform matrix which including rotation, translation, shear and scaling transforms. If you set use_mask_refine == True, the code will use the masks annotation to refine the bbox. Our implementation is slightly different from the official. In COCO dataset, a gt may have multiple mask tags. The official YOLOv5 annotation file already combines the masks that an object has, but our code takes into account the fact that an object has multiple masks.

Required Keys:

- img
- gt_bboxes (BaseBoxes[torch.float32]) (optional)
- gt_bboxes_labels (np.int64) (optional)
- gt_ignore_flags (bool) (optional)
- gt_masks (PolygonMasks) (optional)

Modified Keys:

- img

- img_shape
- gt_bboxes (optional)
- gt_bboxes_labels (optional)
- gt_ignore_flags (optional)
- gt_masks (PolygonMasks) (optional)

参数

- **max_rotate_degree** (*float*) –Maximum degrees of rotation transform. Defaults to 10.
- **max_translate_ratio** (*float*) –Maximum ratio of translation. Defaults to 0.1.
- **scaling_ratio_range** (*tuple[float]*) –Min and max ratio of scaling transform. Defaults to (0.5, 1.5).
- **max_shear_degree** (*float*) –Maximum degrees of shear transform. Defaults to 2.
- **border** (*tuple[int]*) –Distance from width and height sides of input image to adjust output shape. Only used in mosaic dataset. Defaults to (0, 0).
- **border_val** (*tuple[int]*) –Border padding values of 3 channels. Defaults to (114, 114, 114).
- **bbox_clip_border** (*bool, optional*) –Whether to clip the objects outside the border of the image. In some dataset like MOT17, the gt bboxes are allowed to cross the border of images. Therefore, we don't need to clip the gt bboxes in these cases. Defaults to True.
- **min_bbox_size** (*float*) –Width and height threshold to filter bboxes. If the height or width of a box is smaller than this value, it will be removed. Defaults to 2.
- **min_area_ratio** (*float*) –Threshold of area ratio between original bboxes and wrapped bboxes. If smaller than this value, the box will be removed. Defaults to 0.1.
- **use_mask_refine** (*bool*) –Whether to refine bbox by mask. Deprecated.
- **max_aspect_ratio** (*float*) –Aspect ratio of width and height threshold to filter bboxes. If max(h/w, w/h) larger than this value, the box will be removed. Defaults to 20.
- **resample_num** (*int*) –Number of poly to resample to.

clip_polygons (*gt_masks: mmdet.structures.mask.structures.PolygonMasks*, *height: int*, *width: int*) → mmdet.structures.mask.structures.PolygonMasks

Function to clip points of polygons with height and width.

参数

- **gt_masks** (*PolygonMasks*) –Annotations of instance segmentation.
- **height** (*int*) –height of clip border.

- **width** (*int*) –width of clip border.

返回 Clip annotations of instance segmentation.

返回类型 clipped_masks (PolygonMasks)

filter_gt_bboxes(*origin_bboxes: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*, *wrapped_bboxes: mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes*) → torch.Tensor

Filter gt bboxes.

参数

- **origin_bboxes** (*HorizontalBoxes*) –Origin bboxes.
- **wrapped_bboxes** (*HorizontalBoxes*) –Wrapped bboxes

返回 The result dict.

返回类型 dict

resample_masks(*gt_masks: mmdet.structures.mask.structures.PolygonMasks*) → mmdet.structures.mask.structures.PolygonMasks

Function to resample each mask annotation with shape (2 * n,) to shape (resample_num * 2,).

参数 **gt_masks** (*PolygonMasks*) –Annotations of semantic segmentation.

segment2box(*gt_masks: mmdet.structures.mask.structures.PolygonMasks*, *height: int*, *width: int*) → mmdet.structures.bbox.horizontal_boxes.HorizontalBoxes

Convert 1 segment label to 1 box label, applying inside-image constraint i.e. (xy1, xy2, ···) to (xyxy) :param gt_masks: the segment label :type gt_masks: torch.Tensor :param width: the width of the image. Defaults to 640 :type width: int :param height: The height of the image. Defaults to 640 :type height: int

返回 the clip bboxes from gt_masks.

返回类型 HorizontalBoxes

warp_mask(*gt_masks: mmdet.structures.mask.structures.PolygonMasks*, *warp_matrix: numpy.ndarray*, *img_w: int*, *img_h: int*) → mmdet.structures.mask.structures.PolygonMasks

Warp masks by warp_matrix and retain masks inside image after warping.

参数

- **gt_masks** (*PolygonMasks*) –Annotations of semantic segmentation.
- **warp_matrix** (*np.ndarray*) –Affine transformation matrix. Shape: (3, 3).
- **img_w** (*int*) –Width of output image.
- **img_h** (*int*) –Height of output image.

返回 Masks after warping.

返回类型 PolygonMasks

static warp_poly(*poly: numpy.ndarray*, *warp_matrix: numpy.ndarray*, *img_w: int*, *img_h: int*) → numpy.ndarray

Function to warp one mask and filter points outside image.

参数

- **poly** (*np.ndarray*) –Segmentation annotation with shape (n,) and with format (x1, y1, x2, y2, ···).
- **warp_matrix** (*np.ndarray*) –Affine transformation matrix. Shape: (3, 3).
- **img_w** (*int*) –Width of output image.
- **img_h** (*int*) –Height of output image.

CHAPTER 60

mmyolo.engine

60.1 hooks

60.2 optimizers

CHAPTER 61

mmyolo.models

61.1 backbones

class mmyolo.models.backbones.**BaseBackbone**(*arch_setting: list*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Sequence[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *plugins: Optional[Union[dict, List[dict]]] = None*, *norm_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *act_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

BaseBackbone backbone used in YOLO series.

```
Backbone model structure diagram
+-----------+
|   input   |
+-----------+
      v
+-----------+
|   stem    |
```

(下页继续)

(续上页)

```
|   layer   |
+-----------+
      v
+-----------+
|   stage   |
|  layer 1  |
+-----------+
      v
+-----------+
|   stage   |
|  layer 2  |
+-----------+
      v
    ......
      v
+-----------+
|   stage   |
|  layer n  |
+-----------+
In P5 model, n=4
In P6 model, n=5
```

参数

- **arch_setting** (*list*) –Architecture of BaseBackbone.
- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains:
  - cfg (dict, required): Cfg dict to build plugin.
  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as ‘num_stages’ .
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** –Number of input image channels. Defaults to 3.
- **out_indices** (*Sequence[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) –Dictionary to construct and config norm layer. Defaults to None.
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to None.

- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

abstract build_stage_layer(*stage_idx: int*, *setting: list*)

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

abstract build_stem_layer()

Build a stem layer.

forward(*x: torch.Tensor*) → tuple

Forward batch_inputs from the data_preprocessor.

make_stage_plugins(*plugins*, *stage_idx*, *setting*)

Make plugins for backbone `stage_idx` th stage.

Currently we support to insert `context_block`, `empirical_attention_block`, `nonlocal_block`, `dropout_block` into the backbone.

An example of plugins format could be:

实际案例

```
>>> plugins=[
...     dict(cfg=dict(type='xxx', arg1='xxx'),
...          stages=(False, True, True, True)),
...     dict(cfg=dict(type='yyy'),
...          stages=(True, True, True, True)),
... ]
>>> model = YOLOv5CSPDarknet()
>>> stage_plugins = model.make_stage_plugins(plugins, 0, setting)
>>> assert len(stage_plugins) == 1
```

Suppose `stage_idx=0`, the structure of blocks in the stage would be:

```
conv1 -> conv2 -> conv3 -> yyy
```

Suppose `stage_idx=1`, the structure of blocks in the stage would be:

```
conv1 -> conv2 -> conv3 -> xxx -> yyy
```

参数

- **plugins** (*list[dict]*) –List of plugins cfg to build. The postfix is required if multiple same type plugins are inserted.
- **stage_idx** (*int*) –Index of stage to build If stages is missing, the plugin would be applied to all stages.
- **setting** (*list*) –The architecture setting of a stage layer.

返回 Plugins for current stage

返回类型 list[nn.Module]

train(*mode: bool = True*)

Convert the model into training mode while keep normalization layer frozen.

class mmyolo.models.backbones.**CSPNeXt**(*arch: str = 'P5'*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Sequence[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *plugins: Optional[Union[dict, List[dict]]] = None*, *use_depthwise: bool = False*, *expand_ratio: float = 0.5*, *arch_ovewrite: Optional[dict] = None*, *channel_attention: bool = True*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = {'a': 2.23606797749979, 'distribution': 'uniform', 'layer': 'Conv2d', 'mode': 'fan_in', 'nonlinearity': 'leaky_relu', 'type': 'Kaiming'}*)

CSPNeXt backbone used in RTMDet.

参数

- **arch** (*str*) –Architecture of CSPNeXt, from {P5, P6}. Defaults to P5.
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.

- **out_indices** (*Sequence[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin.Defaults to - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as ‘num_stages’.
- **use_depthwise** (*bool*) –Whether to use depthwise separable convolution. Defaults to False.
- **expand_ratio** (*float*) –Ratio to adjust the number of channels of the hidden layer. Defaults to 0.5.
- **arch_ovewrite** (*list*) –Overwrite default arch settings. Defaults to None.
- **channel_attention** (*bool*) –Whether to add channel attention in each stage. Defaults to True.
- **conv_cfg** (ConfigDict or dict, optional) –Config dict for convolution layer. Defaults to None.
- **norm_cfg** (ConfigDict or dict) –Dictionary to construct and config norm layer. Defaults to dict(type=’BN’, requires_grad=True).
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Defaults to dict(type=’SiLU’, inplace=True).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict]): Initialization config dict.

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

class mmyolo.models.backbones.**PPYOLOECSPResNet**(*arch: str = 'P5'*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *plugins: Optional[Union[dict, List[dict]]] = None*, *arch_ovewrite: Optional[dict] = None*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'shortcut': True, 'type': 'PPYOLOEBasicBlock', 'use_alpha': True}*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *attention_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'act_cfg': {'type': 'HSigmoid'}, 'type': 'EffectiveSELayer'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, *use_large_stem: bool = False*)

CSP-ResNet backbone used in PPYOLOE.

参数

- **arch** (*str*) –Architecture of CSPNeXt, from {P5, P6}. Defaults to P5.
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **out_indices** (*Sequence[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as ‘num_stages’.

- **arch_ovewrite** (*list*) –Overwrite default arch settings. Defaults to None.
- **block_cfg** (*dict*) –Config dict for block. Defaults to dict(type='PPYOLOEBasicBlock', shortcut=True, use_alpha=True)
- **norm_cfg** (ConfigDict or dict) –Dictionary to construct and config norm layer. Defaults to dict(type='BN', momentum=0.1, eps=1e-5).
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **attention_cfg** (*dict*) –Config dict for *EffectiveSELayer*. Defaults to dict(type='EffectiveSELayer', act_cfg=dict(type='HSigmoid')).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict]): Initialization config dict. :param use_large_stem: Whether to use large stem layer.

Defaults to False.

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

class mmyolo.models.backbones.**YOLOXCSPDarknet**(*arch: str = 'P5'*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *use_depthwise: bool = False*, *spp_kernal_sizes: Tuple[int] = (5, 9, 13)*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSP-Darknet backbone used in YOLOX.

参数

- **arch** (*str*) –Architecture of CSP-Darknet, from {P5, P6}. Defaults to P5.
- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains:
  - cfg (dict, required): Cfg dict to build plugin.
  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as 'num_stages'.
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) –Number of input image channels. Defaults to 3.
- **out_indices** (*Tuple[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **use_depthwise** (*bool*) –Whether to use depthwise separable convolution. Defaults to False.
- **spp_kernal_sizes** –(tuple[int]): Sequential of kernel sizes of SPP layers. Defaults to (5, 9, 13).
- **norm_cfg** (*dict*) –Dictionary to construct and config norm layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).

- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type=’SiLU’, inplace=True).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.
- **init_cfg** (*Union[dict,list[dict]], optional*) –Initialization config dict. Defaults to None.

示例

```
>>> from mmyolo.models import YOLOXCSPDarknet
>>> import torch
>>> model = YOLOXCSPDarknet()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

class mmyolo.models.backbones.**YOLOv5CSPDarknet**(*arch: str = 'P5'*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSP-Darknet backbone used in YOLOv5. :param arch: Architecture of CSP-Darknet, from {P5, P6}.

Defaults to P5.

参数

- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as ‘num_stages’.

- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) –Number of input image channels. Defaults to: 3.
- **out_indices** (*Tuple[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) –Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.

- **init_cfg** (*Union[dict,list[dict]], optional*) –Initialization config dict. Defaults to None.

示例

```
>>> from mmyolo.models import YOLOv5CSPDarknet
>>> import torch
>>> model = YOLOv5CSPDarknet()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

init_weights()

Initialize the parameters.

class mmyolo.models.backbones.**YOLOv6CSPBep**(*arch: str = 'P5', plugins: Optional[Union[dict, List[dict]]] = None, deepen_factor: float = 1.0, widen_factor: float = 1.0, input_channels: int = 3, hidden_ratio: float = 0.5, out_indices: Tuple[int] = (2, 3, 4), frozen_stages: int = - 1, use_cspsppf: bool = False, norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}, act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}, norm_eval: bool = False, block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'ConvWrapper'}, init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSPBep backbone used in YOLOv6. :param arch: Architecture of BaseDarknet, from {P5, P6}.

Defaults to P5.

参数

- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as ‘num_stages’.

- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) –Number of input image channels. Defaults to 3.
- **out_indices** (*Tuple[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) –Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='LeakyReLU', negative_slope=0.1).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.

- **block_cfg** (*dict*) –Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **block_act_cfg** (*dict*) –Config dict for activation layer used in each stage. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*Union[dict, list[dict]], optional*) –Initialization config dict. Defaults to None.

示例

```
>>> from mmyolo.models import YOLOv6CSPBep
>>> import torch
>>> model = YOLOv6CSPBep()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

class mmyolo.models.backbones.**YOLOv6EfficientRep**(*arch: str = 'P5', plugins: Optional[Union[dict, List[dict]]] = None, deepen_factor: float = 1.0, widen_factor: float = 1.0, input_channels: int = 3, out_indices: Tuple[int] = (2, 3, 4), frozen_stages: int = - 1, use_cspsppf: bool = False, norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}, act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}, norm_eval: bool = False, block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}, init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

EfficientRep backbone used in YOLOv6. :param arch: Architecture of BaseDarknet, from {P5, P6}.

Defaults to P5.

参数

- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as ‘num_stages’.

- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) –Number of input image channels. Defaults to 3.
- **out_indices** (*Tuple[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **norm_cfg** (*dict*) –Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).

- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type=’LeakyReLU’, negative_slope=0.1).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **block_cfg** (*dict*) –Config dict for the block used to build each layer. Defaults to dict(type=’RepVGGBlock’).
- **init_cfg** (*Union[dict, list[dict]], optional*) –Initialization config dict. Defaults to None.

示例

```
>>> from mmyolo.models import YOLOv6EfficientRep
>>> import torch
>>> model = YOLOv6EfficientRep()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

init_weights()

Initialize the weights.

class mmyolo.models.backbones.**YOLOv7Backbone**(*arch: str = 'L'*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *plugins: Optional[Union[dict, List[dict]]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Backbone used in YOLOv7.

参数

- **arch** (*str*) –Architecture of YOLOv7Defaults to L.
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **out_indices** (*Sequence[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.
- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains:
  - cfg (dict, required): Cfg dict to build plugin.
  - stages (tuple[bool], optional): Stages to apply plugin, length should be same as ‘num_stages’ .
- **norm_cfg** (ConfigDict or dict) –Dictionary to construct and config norm layer. Defaults to dict(type=’ BN’ , requires_grad=True).
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Defaults to dict(type=’ SiLU’ , inplace=True).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only.

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict]): Initialization config dict.

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

class mmyolo.models.backbones.**YOLOv8CSPDarknet**(*arch: str = 'P5'*, *last_stage_out_channels: int = 1024*, *plugins: Optional[Union[dict, List[dict]]] = None*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *input_channels: int = 3*, *out_indices: Tuple[int] = (2, 3, 4)*, *frozen_stages: int = - 1*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *norm_eval: bool = False*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

CSP-Darknet backbone used in YOLOv8.

参数

- **arch** (*str*) –Architecture of CSP-Darknet, from {P5}. Defaults to P5.
- **last_stage_out_channels** (*int*) –Final layer output channel. Defaults to 1024.
- **plugins** (*list[dict]*) –List of plugins for stages, each dict contains: - cfg (dict, required): Cfg dict to build plugin. - stages (tuple[bool], optional): Stages to apply plugin, length

  should be same as ‘num_stages’.
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **input_channels** (*int*) –Number of input image channels. Defaults to: 3.
- **out_indices** (*Tuple[int]*) –Output from which stages. Defaults to (2, 3, 4).
- **frozen_stages** (*int*) –Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Defaults to -1.

- **norm_cfg** (*dict*) –Dictionary to construct and config norm layer. Defaults to dict(type='BN', requires_grad=True).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **norm_eval** (*bool*) –Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Defaults to False.
- **init_cfg** (*Union[dict,list[dict]], optional*) –Initialization config dict. Defaults to None.

示例

```
>>> from mmyolo.models import YOLOv8CSPDarknet
>>> import torch
>>> model = YOLOv8CSPDarknet()
>>> model.eval()
>>> inputs = torch.rand(1, 3, 416, 416)
>>> level_outputs = model(inputs)
>>> for level_out in level_outputs:
...     print(tuple(level_out.shape))
...
(1, 256, 52, 52)
(1, 512, 26, 26)
(1, 1024, 13, 13)
```

build_stage_layer(*stage_idx: int*, *setting: list*) → list

Build a stage layer.

参数

- **stage_idx** (*int*) –The index of a stage layer.
- **setting** (*list*) –The architecture setting of a stage layer.

build_stem_layer() → torch.nn.modules.module.Module

Build a stem layer.

init_weights()

Initialize the parameters.

61.2 data_preprocessor

61.3 dense_heads

class mmyolo.models.dense_heads.**PPYOLOEHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0.5, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'alpha': 0.75, 'gamma': 2.0, 'iou_weighted': True, 'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.VarifocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xyxy', 'iou_mode': 'giou', 'loss_weight': 2.5, 'reduction': 'mean', 'return_iou': False, 'type': 'IoULoss'}*, *loss_dfl: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 0.125, 'reduction': 'mean', 'type': 'mmdet.DistributionFocalLoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

PPYOLOEHead head used in PPYOLOE. The YOLOv6 head and the PPYOLOE head are only slightly different. Distribution focal loss is extra used in PPYOLOE, but not in YOLOv6.

参数

- **head_module** (*ConfigType*) –Base module used for YOLOv5Head
- **prior_generator** (*dict*) –Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) –Config of bbox coder.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.

- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **loss_dfl** (ConfigDict or dict) –Config of distribution focal loss.
- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat (*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *bbox_dist_preds: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

参数

- **cls_scores** (*Sequence[Tensor]*) –Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) –Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **bbox_dist_preds** (*Sequence[Tensor]*) –Box distribution logits for each scale level with shape (bs, reg_max + 1, H*W, 4).
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of losses.

返回类型 dict[str, Tensor]

class mmyolo.models.dense_heads.**PPYOLOEHeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *reg_max: int = 16*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

PPYOLOEHead head module used in ‘PPYOLOE.

<https://arxiv.org/abs/2203.16250>‘_.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **in_channels** (*int*) –Number of channels in the input feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) –The number of priors (points) at a point on the feature grid.
- **featmap_strides** (*Sequence[int]*) –Downsample factor of each feature map. Defaults to (8, 16, 32).
- **reg_max** (*int*) –Max value of integral set :math: `{0, ..., reg_max}` in QFL setting. Defaults to 16.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type=’BN’, momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type=’SiLU’, inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → torch.Tensor

Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions.

返回类型 Tuple[List]

forward_single(*x: torch.Tensor*, *cls_stem: torch.nn.modules.container.ModuleList*, *cls_pred: torch.nn.modules.container.ModuleList*, *reg_stem: torch.nn.modules.container.ModuleList*, *reg_pred: torch.nn.modules.container.ModuleList*) → torch.Tensor

Forward feature of a single scale level.

init_weights(*prior_prob=0.01*)

Initialize the weight and bias of PPYOLOE head.

class mmyolo.models.dense_heads.**RTMDetHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'beta': 2.0, 'loss_weight': 1.0, 'type': 'mmdet.QualityFocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 2.0, 'type': 'mmdet.GIoULoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

RTMDet head.

参数

- **head_module** (*ConfigType*) –Base module used for RTMDetHead
- **prior_generator** –Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) –Config of bbox coder.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.
- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to

None.

- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions, and objectnesses.

返回类型 Tuple[List]

loss_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: List[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict
Compute losses of the head.

参数

- **cls_scores** (*list[Tensor]*) –Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) –Decoded box for each scale level with shape (N, num_anchors * 4, H, W) in [tl_x, tl_y, br_x, br_y] format.
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], Optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of loss components.

返回类型 dict[str, Tensor]

special_init()
Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**RTMDetInsSepBNHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'beta': 2.0, 'loss_weight': 1.0, 'type': 'mmdet.QualityFocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 2.0, 'type': 'mmdet.GIoULoss'}*, *loss_mask={'eps': 5e-06, 'loss_weight': 2.0, 'reduction': 'mean', 'type': 'mmdet.DiceLoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

RTMDet Instance Segmentation head.

参数

- **head_module** (*ConfigType*) –Base module used for RTMDetInsSepBNHead
- **prior_generator** –Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) –Config of bbox coder.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.
- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **loss_mask** (ConfigDict or dict) –Config of mask loss.
- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to None.

- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat (*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: List[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Compute losses of the head.

参数

- **cls_scores** (*list[Tensor]*) –Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) –Decoded box for each scale level with shape (N, num_anchors * 4, H, W) in [tl_x, tl_y, br_x, br_y] format.
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], Optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of loss components.

返回类型 dict[str, Tensor]

parse_dynamic_params (*flatten_kernels: torch.Tensor*) → tuple

split kernel head prediction to conv weight and bias.

predict_by_feat (*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *kernel_preds: List[torch.Tensor]*, *mask_feats: torch.Tensor*, *score_factors: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted from the head into bbox results.

Note: When score_factors is not None, the cls_scores are usually multiplied by it then obtain the real score used in NMS.

参数

- **cls_scores** (*list[Tensor]*) –Classification scores for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).
- **bbox_preds** (*list[Tensor]*) –Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **kernel_preds** (*list[Tensor]*) –Kernel predictions of dynamic convs for all scale levels, each is a 4D-tensor, has shape (batch_size, num_params, H, W).
- **mask_feats** (*Tensor*) –Mask prototype features extracted from the mask head, has shape (batch_size, num_prototypes, H, W).
- **score_factors** (*list[Tensor], optional*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, num_priors * 1, H, W). Defaults to None.
- **batch_img_metas** (*list[dict], Optional*) –Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) –Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) –If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) –If True, do nms before return boxes. Defaults to True.

返回

Object detection and instance segmentation results of each image after the post process. Each item usually contains following keys.

- scores (Tensor): Classification scores, has a shape (num_instance,)
- labels (Tensor): Labels of bboxes, has a shape (num_instances,).
- bboxes (Tensor): Has a shape (num_instances, 4), the last dimension 4 arrange as (x1, y1, x2, y2).
- masks (Tensor): Has a shape (num_instances, h, w).

返回类型 list[InstanceData]

class mmyolo.models.dense_heads.**RTMDetInsSepBNHeadModule**(*num_classes: int*, **args*, *num_prototypes: int = 8*, *dyconv_channels: int = 8*, *num_dyconvs: int = 3*, *use_sigmoid_cls: bool = True*, ***kwargs*)

Detection and Instance Segmentation Head of RTMDet.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.

- **num_prototypes** (*int*) –Number of mask prototype features extracted from the mask head. Defaults to 8.
- **dyconv_channels** (*int*) –Channel of the dynamic conv layers. Defaults to 8.
- **num_dyconvs** (*int*) –Number of the dynamic convolution layers. Defaults to 3.
- **use_sigmoid_cls** (*bool*) –Use sigmoid for class prediction. Defaults to True.

forward(*feats: Tuple[torch.Tensor, ...]*) → tuple

Forward features from the upstream network.

参数 **feats** (*tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回

Usually a tuple of classification scores and bbox prediction - cls_scores (list[Tensor]): Classification scores for all scale

levels, each is a 4D-tensor, the channels number is num_base_priors * num_classes.

- bbox_preds (list[Tensor]): Box energies / deltas for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * 4.
- kernel_preds (list[Tensor]): Dynamic conv kernels for all scale levels, each is a 4D-tensor, the channels number is num_gen_params.
- **mask_feat (Tensor): Mask prototype features.** Has shape (batch_size, num_prototypes, H, W).

返回类型 tuple

init_weights() → None

Initialize weights of the head.

class mmyolo.models.dense_heads.**RTMDetRotatedHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistanceAnglePointCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'beta': 2.0, 'loss_weight': 1.0, 'type': 'mmdet.QualityFocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 2.0, 'mode': 'linear', 'type': 'mmrotate.RotatedIoULoss'}*, *angle_version: str = 'le90'*, *use_hbbox_loss: bool = False*, *angle_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'mmrotate.PseudoAngleCoder'}*, *loss_angle: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

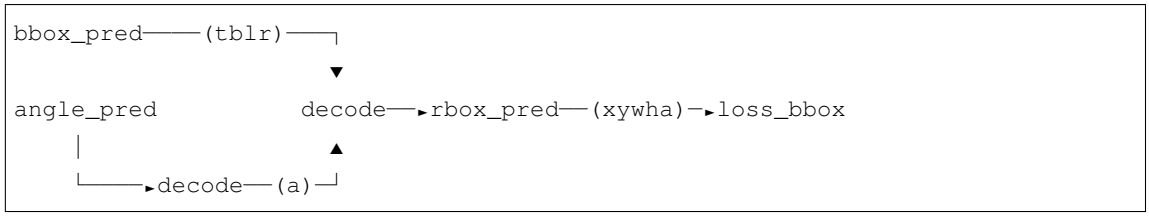
RTMDet-R head.

Compared with RTMDetHead, RTMDetRotatedHead add some args to support rotated object detection.

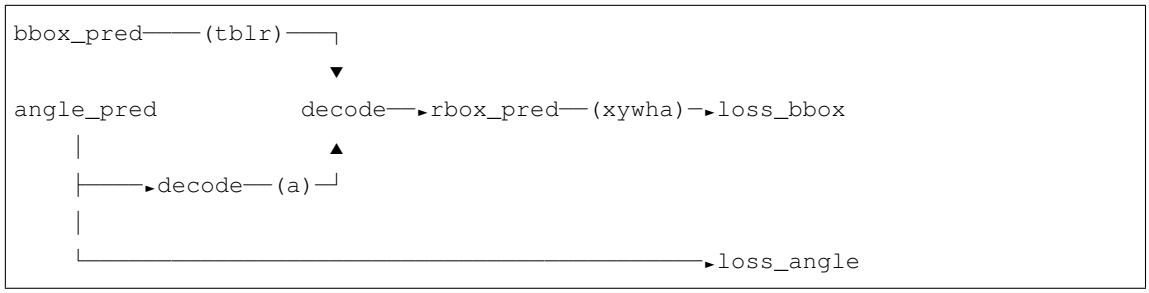
- *angle_version* used to limit angle_range during training.
- *angle_coder* used to encode and decode angle, which is similar to bbox_coder.
- *use_hbbox_loss* and *loss_angle* allow custom regression loss calculation for rotated box.

  There are three combination options for regression:

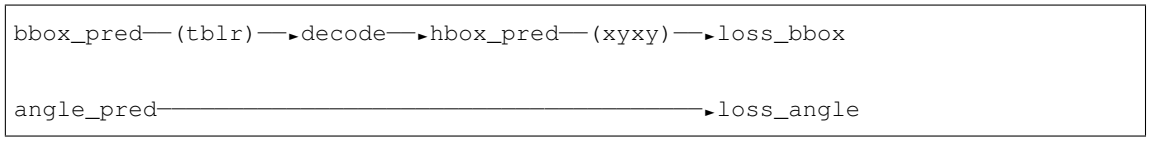
1. *use_hbbox_loss=False* and loss_angle is None.



2. *use_hbbox_loss=False* and loss_angle is specified. A angle loss is added on angle_pred.

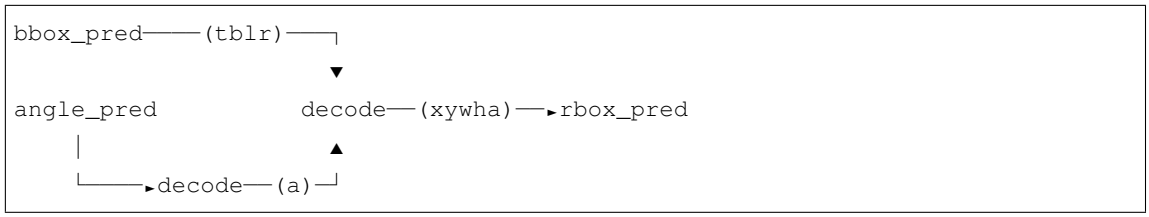


3. *use_hbbox_loss=True* and loss_angle is specified. In this case the loss_angle must be set.

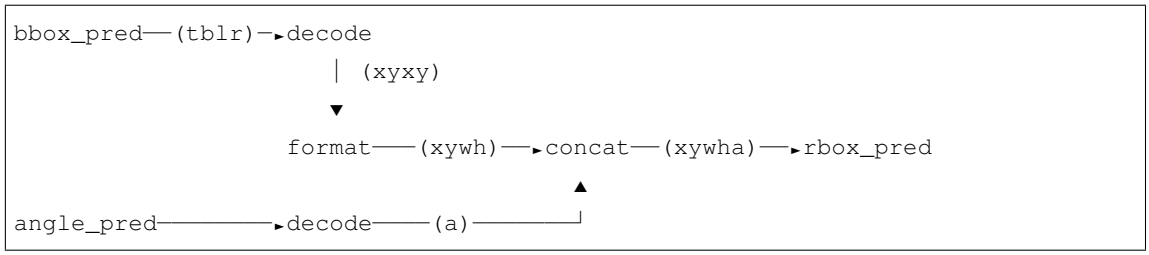


- There’s a *decoded_with_angle* flag in test_cfg, which is similar to training process.

  When *decoded_with_angle=True*:



  When *decoded_with_angle=False*:



参数

- **head_module** (*ConfigType*) –Base module used for RTMDetRotatedHead.
- **prior_generator** –Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) –Config of bbox coder.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.
- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **angle_version** (*str*) –Angle representations. Defaults to 'le90'.
- **use_hbbox_loss** (*bool*) –If true, use horizontal bbox loss and loss_angle should not be None. Default to False.
- **angle_coder** (ConfigDict or dict) –Config of angle coder.
- **loss_angle** (ConfigDict or dict, optional) –Config of angle loss.
- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat (*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *angle_preds: List[torch.Tensor]*, *batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: List[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Compute losses of the head.

参数

- **cls_scores** (*list[Tensor]*) –Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) –Decoded box for each scale level with shape (N, num_anchors * 4, H, W) in [tl_x, tl_y, br_x, br_y] format.
- **angle_preds** (*list[Tensor]*) –Angle prediction for each scale level with shape (N, num_anchors * angle_out_dim, H, W).
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.

- **batch_gt_instances_ignore** (list[InstanceData], Optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of loss components.

返回类型 dict[str, Tensor]

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *angle_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted by the head into bbox results.

参数

- **cls_scores** (*list[Tensor]*) –Classification scores for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).
- **bbox_preds** (*list[Tensor]*) –Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **angle_preds** (*list[Tensor]*) –Box angle for each scale level with shape (N, num_points * angle_dim, H, W)
- **objectnesses** (*list[Tensor], Optional*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_img_metas** (*list[dict], Optional*) –Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) –Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) –If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) –If True, do nms before return boxes. Defaults to True.

返回

Object detection results of each image after the post process. Each item usually contains following keys. - scores (Tensor): Classification scores, has a shape

(num_instance,)

- labels (Tensor): Labels of bboxes, has a shape (num_instances,).
- bboxes (Tensor): Has a shape (num_instances, 5), the last dimension 4 arrange as (x, y, w, h, angle).

返回类型 list[InstanceData]

class mmyolo.models.dense_heads.**RTMDetRotatedSepBNHeadModule**(*num_classes: int*, *in_channels: int*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *feat_channels: int = 256*, *stacked_convs: int = 2*, *featmap_strides: Sequence[int] = [8, 16, 32]*, *share_conv: bool = True*, *pred_kernel_size: int = 1*, *angle_out_dim: int = 1*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Detection Head Module of RTMDet-R.

Compared with RTMDet Detection Head Module, RTMDet-R adds a conv for angle prediction. An *angle_out_dim* arg is added, which is generated by the angle_coder module and controls the angle pred dim.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **in_channels** (*int*) –Number of channels in the input feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) –The number of priors (points) at a point on the feature grid. Defaults to 1.
- **feat_channels** (*int*) –Number of hidden channels. Used in child classes. Defaults to 256
- **stacked_convs** (*int*) –Number of stacking convs of the head. Defaults to 2.

- **featmap_strides** (*Sequence[int]*) –Downsample factor of each feature map. Defaults to (8, 16, 32).
- **share_conv** (*bool*) –Whether to share conv layers between stages. Defaults to True.
- **pred_kernel_size** (*int*) –Kernel size of `nn.Conv2d`. Defaults to 1.
- **angle_out_dim** (*int*) –Encoded length of angle, will passed by head. Defaults to 1.
- **conv_cfg** (`ConfigDict` or dict, optional) –Config dict for convolution layer. Defaults to None.
- **norm_cfg** (`ConfigDict` or dict) –Config dict for normalization layer. Defaults to `dict(type='BN')`.
- **act_cfg** (`ConfigDict` or dict) –Config dict for activation layer. Default: dict(type='SiLU', inplace=True).

:param init_cfg (`ConfigDict` or list[`ConfigDict`] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*feats: Tuple[torch.Tensor, ...]*) → tuple
Forward features from the upstream network.

参数 **feats** (*tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回

Usually a tuple of classification scores and bbox prediction - cls_scores (list[Tensor]): Classification scores for all scale

levels, each is a 4D-tensor, the channels number is num_base_priors * num_classes.

- bbox_preds (list[Tensor]): Box energies / deltas for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * 4.
- angle_preds (list[Tensor]): Angle prediction for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * angle_out_dim.

返回类型 tuple

init_weights() → None
Initialize weights of the head.

class mmyolo.models.dense_heads.**RTMDetSepBNHeadModule**(*num_classes: int*, *in_channels: int*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *feat_channels: int = 256*, *stacked_convs: int = 2*, *featmap_strides: Sequence[int] = [8, 16, 32]*, *share_conv: bool = True*, *pred_kernel_size: int = 1*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Detection Head of RTMDet.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **in_channels** (*int*) –Number of channels in the input feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) –The number of priors (points) at a point on the feature grid. Defaults to 1.
- **feat_channels** (*int*) –Number of hidden channels. Used in child classes. Defaults to 256
- **stacked_convs** (*int*) –Number of stacking convs of the head. Defaults to 2.
- **featmap_strides** (*Sequence[int]*) –Downsample factor of each feature map. Defaults to (8, 16, 32).
- **share_conv** (*bool*) –Whether to share conv layers between stages. Defaults to True.
- **pred_kernel_size** (*int*) –Kernel size of `nn.Conv2d`. Defaults to 1.
- **conv_cfg** (`ConfigDict` or dict, optional) –Config dict for convolution layer. Defaults to None.

- **norm_cfg** (ConfigDict or dict) –Config dict for normalization layer. Defaults to dict(type='BN').
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Default: dict(type='SiLU', inplace=True).

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*feats: Tuple[torch.Tensor, ...]*) → tuple
Forward features from the upstream network.

参数 **feats** (*tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回

Usually a tuple of classification scores and bbox prediction - cls_scores (list[Tensor]): Classification scores for all scale

levels, each is a 4D-tensor, the channels number is num_base_priors * num_classes.

- bbox_preds (list[Tensor]): Box energies / deltas for all scale levels, each is a 4D-tensor, the channels number is num_base_priors * 4.

返回类型 tuple

init_weights() → None
Initialize weights of the head.

class mmyolo.models.dense_heads.**YOLOXHead**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'YOLOXBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-16, 'loss_weight': 5.0, 'mode': 'square', 'reduction': 'sum', 'type': 'mmdet.IoULoss'}*, *loss_obj: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox_aux: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.L1Loss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOXHead head used in YOLOX.

参数

- **head_module** (*ConfigType*) –Base module used for YOLOXHead
- **prior_generator** –Points generator feature maps in 2D points-based detectors.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.
- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **loss_obj** (ConfigDict or dict) –Config of objectness loss.
- **loss_bbox_aux** (ConfigDict or dict) –Config of bbox aux loss.
- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to None.

- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions, and objectnesses.

返回类型 Tuple[List]

static gt_instances_preprocess(*batch_gt_instances: torch.Tensor*, *batch_size: int*) → List[mmengine.structures.instance_data.InstanceData]
Split batch_gt_instances with batch size.

参数

- **batch_gt_instances** (*Tensor*) –Ground truth a 2D-Tensor for whole batch, shape [all_gt_bboxes, 6]
- **batch_size** (*int*) –Batch size.

返回 batch gt instances data, shape [batch_size, InstanceData]

返回类型 List

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *batch_gt_instances: torch.Tensor*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict
Calculate the loss based on the features extracted by the detection head.

参数

- **cls_scores** (*Sequence[Tensor]*) –Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) –Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.

- **batch_gt_instances_ignore** (list[InstanceData], optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of losses.

返回类型 dict[str, Tensor]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOXHeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *feat_channels: int = 256*, *stacked_convs: int = 2*, *featmap_strides: Sequence[int] = [8, 16, 32]*, *use_depthwise: bool = False*, *dcn_on_last_conv: bool = False*, *conv_bias: Union[bool, str] = 'auto'*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOXHead head module used in ‘YOLOX.

https://arxiv.org/abs/2107.08430

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) –Number of channels in the input feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.

- **num_base_priors** (*int*) –The number of priors (points) at a point on the feature grid
- **stacked_convs** (*int*) –Number of stacking convs of the head. Defaults to 2.
- **featmap_strides** (*Sequence[int]*) –Downsample factor of each feature map. Defaults to [8, 16, 32].
- **use_depthwise** (*bool*) –Whether to depthwise separable convolution in blocks. Defaults to False.
- **dcn_on_last_conv** (*bool*) –If true, use dcn in the last layer of towers. Defaults to False.
- **conv_bias** (*bool or str*) –If specified as *auto*, it will be decided by the norm_cfg. Bias of conv will be set as True if *norm_cfg* is None, otherwise False. Defaults to "auto".
- **conv_cfg** (ConfigDict or dict, optional) –Config dict for convolution layer. Defaults to None.
- **norm_cfg** (ConfigDict or dict) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.

Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions, and objectnesses.

返回类型 Tuple[List]

forward_single(*x: torch.Tensor*, *cls_convs: torch.nn.modules.module.Module*, *reg_convs: torch.nn.modules.module.Module*, *conv_cls: torch.nn.modules.module.Module*, *conv_reg: torch.nn.modules.module.Module*, *conv_obj: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Forward feature of a single scale level.

init_weights()

Initialize weights of the head.

class mmyolo.models.dense_heads.**YOLOXPoseHead**(*loss_pose: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, **args*, ***kwargs*)

YOLOXPoseHead head used in 'YOLO-Pose.

<https://arxiv.org/abs/2204.06806>'_. :param loss_pose: Config of keypoint OKS loss. :type loss_pose: ConfigDict, optional

decode_pose(*grids: torch.Tensor*, *offsets: torch.Tensor*, *strides: Union[torch.Tensor, int]*) → torch.Tensor

Decode regression offsets to keypoints.

参数

- **grids** (*torch.Tensor*) –The coordinates of the feature map grids.
- **offsets** (*torch.Tensor*) –The predicted offset of each keypoint relative to its corresponding grid.
- **strides** (*torch.Tensor | int*) –The stride of the feature map for each instance.

返回 The decoded keypoints coordinates.

返回类型 torch.Tensor

static gt_instances_preprocess(*batch_gt_instances: List[mmengine.structures.instance_data.InstanceData]*, **args*, ***kwargs*) → List[mmengine.structures.instance_data.InstanceData]

Split batch_gt_instances with batch size.

参数

- **batch_gt_instances** (*Tensor*) –Ground truth a 2D-Tensor for whole batch, shape [all_gt_bboxes, 6]
- **batch_size** (*int*) –Batch size.

返回 batch gt instances data, shape [batch_size, InstanceData]

返回类型 List

static gt_kps_instances_preprocess(*batch_gt_instances: torch.Tensor*, *batch_gt_keypoints*, *batch_gt_keypoints_visible*, *batch_size: int*) → List[mmengine.structures.instance_data.InstanceData]

Split batch_gt_instances with batch size.

参数

- **batch_gt_instances** (*Tensor*) –Ground truth a 2D-Tensor for whole batch, shape [all_gt_bboxes, 6]
- **batch_size** (*int*) –Batch size.

返回 batch gt instances data, shape [batch_size, InstanceData]

返回类型 List

loss(*x: Tuple[torch.Tensor]*, *batch_data_samples: Union[list, dict]*) → dict

Perform forward propagation and loss calculation of the detection head on the features of the upstream network.

参数

- **x** (*tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.
- **batch_data_samples** (List[DetDataSample], dict) –The Data Samples. It usually includes information such as *gt_instance*, *gt_panoptic_seg* and *gt_sem_seg*.

返回 A dictionary of loss components.

返回类型 dict

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *kpt_preds: Sequence[torch.Tensor]*, *vis_preds: Sequence[torch.Tensor]*, *batch_gt_instances: torch.Tensor*, *batch_gt_keypoints: torch.Tensor*, *batch_gt_keypoints_visible: torch.Tensor*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

In addition to the base class method, keypoint losses are also calculated in this method.

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *kpt_preds: Optional[List[torch.Tensor]] = None*, *vis_preds: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted by the head into bbox and keypoint results.

In addition to the base class method, keypoint predictions are also calculated in this method.

class mmyolo.models.dense_heads.**YOLOXPoseHeadModule**(*num_keypoints: int*, **args*, ***kwargs*)

YOLOXPoseHeadModule serves as a head module for *YOLOX-Pose*.

In comparison to *YOLOXHeadModule*, this module introduces branches for keypoint prediction.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

init_weights()

Initialize weights of the head.

class mmyolo.models.dense_heads.**YOLOv5Head**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'base_sizes': [[(10, 13), (16, 30), (33, 23)], [(30, 61), (62, 45), (59, 119)], [(116, 90), (156, 198), (373, 326)]], 'strides': [8, 16, 32], 'type': 'mmdet.YOLOAnchorGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'YOLOv5BBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 0.5, 'reduction': 'mean', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xywh', 'eps': 1e-07, 'iou_mode': 'ciou', 'loss_weight': 0.05, 'reduction': 'mean', 'return_iou': True, 'type': 'IoULoss'}*, *loss_obj: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 1.0, 'reduction': 'mean', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *prior_match_thr: float = 4.0*, *near_neighbor_thr: float = 0.5*, *ignore_iof_thr: float = - 1.0*, *obj_level_weights: List[float] = [4.0, 1.0, 0.4]*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv5Head head used in *YOLOv5*.

参数

- **head_module** (*ConfigType*) –Base module used for YOLOv5Head
- **prior_generator** (*dict*) –Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) –Config of bbox coder.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.
- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **loss_obj** (ConfigDict or dict) –Config of objectness loss.

- **prior_match_thr** (*float*) –Defaults to 4.0.
- **ignore_iof_thr** (*float*) –Defaults to -1.0.
- **obj_level_weights** (*List[float]*) –Defaults to [4.0, 1.0, 0.4].
- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]
Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions, and objectnesses.

返回类型 Tuple[List]

loss(*x: Tuple[torch.Tensor]*, *batch_data_samples: Union[list, dict]*) → dict
Perform forward propagation and loss calculation of the detection head on the features of the upstream network.

参数

- **x** (*tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.
- **batch_data_samples** (List[DetDataSample], dict) –The Data Samples. It usually includes information such as *gt_instance*, *gt_panoptic_seg* and *gt_sem_seg*.

返回 A dictionary of loss components.

返回类型 dict

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict
Calculate the loss based on the features extracted by the detection head.

参数

- **cls_scores** (*Sequence[Tensor]*) –Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.

- **bbox_preds** (*Sequence[Tensor]*) –Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_gt_instances** (*Sequence[InstanceData]*) –Batch of gt_instance. It usually includes `bboxes` and `labels` attributes.
- **batch_img_metas** (*Sequence[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[`InstanceData`], optional) –Batch of gt_instances_ignore. It includes `bboxes` attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of losses.

返回类型 dict[str, Tensor]

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted by the head into bbox results. :param cls_scores: Classification scores for all

scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).

参数

- **bbox_preds** (*list[Tensor]*) –Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **objectnesses** (*list[Tensor], Optional*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_img_metas** (*list[dict], Optional*) –Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) –Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) –If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) –If True, do nms before return boxes. Defaults to True.

返回

Object detection results of each image after the post process. Each item usually contains following keys.

- scores (Tensor): Classification scores, has a shape (num_instance,)
- labels (Tensor): Labels of bboxes, has a shape (num_instances,).
- bboxes (Tensor): Has a shape (num_instances, 4), the last dimension 4 arrange as (x1, y1, x2, y2).

返回类型 list[`InstanceData`]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOv5HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 3*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv5Head head module used in *YOLOv5*.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) –Number of channels in the input feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) –The number of priors (points) at a point on the feature grid.
- **featmap_strides** (*Sequence[int]*) –Downsample factor of each feature map. Defaults to (8, 16, 32).

:param init_cfg (`ConfigDict` or list[`ConfigDict`] or dict or: list[dict], optional): Initialization config dict.

Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions, and objectnesses.

返回类型 Tuple[List]

forward_single(*x: torch.Tensor*, *convs: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Forward feature of a single scale level.

init_weights()

Initialize the bias of YOLOv5 head.

class mmyolo.models.dense_heads.**YOLOv5InsHead**(**args*, *mask_overlap: bool = True*, *loss_mask: Union[mmengine.config.config.ConfigDict, dict] = {'reduction': 'none', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_mask_weight=0.05*, ***kwargs*)

YOLOv5 Instance Segmentation and Detection head.

参数

- **mask_overlap** (*bool*) –Defaults to True.
- **loss_mask** (ConfigDict or dict) –Config of mask loss.
- **loss_mask_weight** (*float*) –The weight of mask loss.

crop_mask(*masks: torch.Tensor*, *boxes: torch.Tensor*) → torch.Tensor

Crop mask by the bounding box.

参数

- **masks** (*Tensor*) –Predicted mask results. Has shape (1, num_instance, H, W).
- **boxes** (*Tensor*) –Tensor of the bbox. Has shape (num_instance, 4).

返回 The masks are being cropped to the bounding box.

返回类型 (torch.Tensor)

loss(*x: Tuple[torch.Tensor]*, *batch_data_samples: Union[list, dict]*) → dict

Perform forward propagation and loss calculation of the detection head on the features of the upstream network.

参数

- **x** (*tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.
- **batch_data_samples** (List[DetDataSample], dict) –The Data Samples. It usually includes information such as *gt_instance*, *gt_panoptic_seg* and *gt_sem_seg*.

返回 A dictionary of loss components.

返回类型 dict

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *objectnesses: Sequence[torch.Tensor]*, *coeff_preds: Sequence[torch.Tensor]*, *proto_preds: torch.Tensor*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_gt_masks: Sequence[torch.Tensor]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

参数

- **cls_scores** (*Sequence[Tensor]*) –Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) –Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **coeff_preds** (*Sequence[Tensor]*) –Mask coefficient for each scale level, each is a 4D-tensor, the channel number is num_priors * mask_channels.
- **proto_preds** (*Tensor*) –Mask prototype features extracted from the mask head, has shape (batch_size, mask_channels, H, W).
- **batch_gt_instances** (*Sequence[InstanceData]*) –Batch of gt_instance. It usually includes `bboxes` and `labels` attributes.
- **batch_gt_masks** (*Sequence[Tensor]*) –Batch of gt_mask.
- **batch_img_metas** (*Sequence[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[`InstanceData`], optional) –Batch of gt_instances_ignore. It includes `bboxes` attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of losses.

返回类型 dict[str, Tensor]

predict_by_feat(*cls_scores: List[torch.Tensor]*, *bbox_preds: List[torch.Tensor]*, *objectnesses: Optional[List[torch.Tensor]] = None*, *coeff_preds: Optional[List[torch.Tensor]] = None*, *proto_preds: Optional[torch.Tensor] = None*, *batch_img_metas: Optional[List[dict]] = None*, *cfg: Optional[mmengine.config.config.ConfigDict] = None*, *rescale: bool = True*, *with_nms: bool = True*) → List[mmengine.structures.instance_data.InstanceData]

Transform a batch of output features extracted from the head into bbox results. Note: When score_factors is not None, the cls_scores are usually multiplied by it then obtain the real score used in NMS. :param cls_scores: Classification scores for all

scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).

参数

- **bbox_preds** (*list[Tensor]*) –Box energies / deltas for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * 4, H, W).
- **objectnesses** (*list[Tensor], Optional*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **coeff_preds** (*list[Tensor]*) –Mask coefficients predictions for all scale levels, each is a 4D-tensor, has shape (batch_size, mask_channels, H, W).
- **proto_preds** (*Tensor*) –Mask prototype features extracted from the mask head, has shape (batch_size, mask_channels, H, W).
- **batch_img_metas** (*list[dict], Optional*) –Batch image meta info. Defaults to None.
- **cfg** (*ConfigDict, optional*) –Test / postprocessing configuration, if None, test_cfg would be used. Defaults to None.
- **rescale** (*bool*) –If True, return boxes in original image space. Defaults to False.
- **with_nms** (*bool*) –If True, do nms before return boxes. Defaults to True.

返回

Object detection and instance segmentation results of each image after the post process. Each item usually contains following keys.

- scores (Tensor): Classification scores, has a shape (num_instance,)
- labels (Tensor): Labels of bboxes, has a shape (num_instances,).
- bboxes (Tensor): Has a shape (num_instances, 4), the last dimension 4 arrange as (x1, y1, x2, y2).
- masks (Tensor): Has a shape (num_instances, h, w).

返回类型 list[InstanceData]

process_mask(*mask_proto: torch.Tensor*, *mask_coeff_pred: torch.Tensor*, *bboxes: torch.Tensor*, *shape: Tuple[int, int]*, *upsample: bool = False*) → torch.Tensor

Generate mask logits results.

参数

- **mask_proto** (*Tensor*) –Mask prototype features. Has shape (num_instance, mask_channels).
- **mask_coeff_pred** (*Tensor*) –Mask coefficients prediction for single image. Has shape (mask_channels, H, W)

- **bboxes** (*Tensor*) –Tensor of the bbox. Has shape (num_instance, 4).
- **shape** (*Tuple*) –Batch input shape of image.
- **upsample** (*bool*) –Whether upsample masks results to batch input shape. Default to False.

返回

Instance segmentation masks for each instance. Has shape (num_instance, H, W).

返回类型 Tensor

class mmyolo.models.dense_heads.**YOLOv5InsHeadModule**(**args*, *num_classes: int*, *mask_channels: int = 32*, *proto_channels: int = 256*, *widen_factor: float = 1.0*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, ***kwargs*)

Detection and Instance Segmentation Head of YOLOv5.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **mask_channels** (*int*) –Number of channels in the mask feature map. This is the channel count of the mask.
- **proto_channels** (*int*) –Number of channels in the proto feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **norm_cfg** (ConfigDict or dict) –Config dict for normalization layer. Defaults to `dict(type='BN', momentum=0.03, eps=0.001)`.
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Default: dict(type='SiLU', inplace=True).

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions, objectnesses, and mask predictions.

返回类型 Tuple[List]

forward_single(*x: torch.Tensor*, *convs_pred: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]

Forward feature of a single scale level.

class mmyolo.models.dense_heads.**YOLOv6Head**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0.5, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'alpha': 0.75, 'gamma': 2.0, 'iou_weighted': True, 'loss_weight': 1.0, 'reduction': 'sum', 'type': 'mmdet.VarifocalLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xyxy', 'iou_mode': 'giou', 'loss_weight': 2.5, 'reduction': 'mean', 'return_iou': False, 'type': 'IoULoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv6Head head used in YOLOv6.

参数

- **head_module** (*ConfigType*) –Base module used for YOLOv6Head
- **prior_generator** (*dict*) –Points generator feature maps in 2D points-based detectors.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.
- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.

Defaults to None.

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *bbox_dist_preds: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

参数

- **cls_scores** (*Sequence[Tensor]*) –Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) –Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of losses.

返回类型 dict[str, Tensor]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOv6HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *reg_max=0*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv6Head head module used in ‘YOLOv6.

<https://arxiv.org/pdf/2209.02976>‘_.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) –Number of channels in the input feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** –(int): The number of priors (points) at a point on the feature grid.
- **featmap_strides** (*Sequence[int]*) –

  Downsample factor of each feature map. Defaults to [8, 16, 32].

  None, otherwise False. Defaults to “auto”.
- **norm_cfg** (ConfigDict or dict) –Config dict for normalization layer. Defaults to dict(type=’BN’, momentum=0.03, eps=0.001).
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions.

返回类型 Tuple[List]

forward_single(*x: torch.Tensor*, *stem: torch.nn.modules.module.Module*, *cls_conv: torch.nn.modules.module.Module*, *cls_pred: torch.nn.modules.module.Module*, *reg_conv: torch.nn.modules.module.Module*, *reg_pred: torch.nn.modules.module.Module*) → Tuple[torch.Tensor, torch.Tensor]

Forward feature of a single scale level.

init_weights()

Initialize the weights.

class mmyolo.models.dense_heads.**YOLOv7Head**(**args*, *simota_candidate_topk: int = 20*, *simota_iou_weight: float = 3.0*, *simota_cls_weight: float = 1.0*, *aux_loss_weights: float = 0.25*, ***kwargs*)

YOLOv7Head head used in YOLOv7.

参数

- **simota_candidate_topk** (*int*) –The candidate top-k which used to get top-k ious to calculate dynamic-k in BatchYOLOv7Assigner. Defaults to 10.
- **simota_iou_weight** (*float*) –The scale factor for regression iou cost in BatchYOLOv7Assigner. Defaults to 3.0.
- **simota_cls_weight** (*float*) –The scale factor for classification cost in BatchYOLOv7Assigner. Defaults to 1.0.

loss_by_feat(*cls_scores: Sequence[Union[torch.Tensor, List]]*, *bbox_preds: Sequence[Union[torch.Tensor, List]]*, *objectnesses: Sequence[Union[torch.Tensor, List]]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

参数

- **cls_scores** (*Sequence[Tensor]*) –Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) –Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **objectnesses** (*Sequence[Tensor]*) –Score factor for all scale level, each is a 4D-tensor, has shape (batch_size, 1, H, W).
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.

- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of losses.

返回类型 dict[str, Tensor]

class mmyolo.models.dense_heads.**YOLOv7HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 3*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv7Head head module used in YOLOv7.

init_weights()

Initialize the bias of YOLOv7 head.

class mmyolo.models.dense_heads.**YOLOv7p6HeadModule**(**args*, *main_out_channels: Sequence[int] = [256, 512, 768, 1024]*, *aux_out_channels: Sequence[int] = [320, 640, 960, 1280]*, *use_aux: bool = True*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, ***kwargs*)

YOLOv7Head head module used in YOLOv7.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions, and objectnesses.

返回类型 Tuple[List]

forward_single(*x: torch.Tensor*, *convs: torch.nn.modules.module.Module*, *aux_convs: Optional[torch.nn.modules.module.Module]*) → Tuple[Union[torch.Tensor, List], Union[torch.Tensor, List], Union[torch.Tensor, List]]

Forward feature of a single scale level.

init_weights()

Initialize the bias of YOLOv5 head.

class mmyolo.models.dense_heads.**YOLOv8Head**(*head_module: Union[mmengine.config.config.ConfigDict, dict]*, *prior_generator: Union[mmengine.config.config.ConfigDict, dict] = {'offset': 0.5, 'strides': [8, 16, 32], 'type': 'mmdet.MlvlPointGenerator'}*, *bbox_coder: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'DistancePointBBoxCoder'}*, *loss_cls: Union[mmengine.config.config.ConfigDict, dict] = {'loss_weight': 0.5, 'reduction': 'none', 'type': 'mmdet.CrossEntropyLoss', 'use_sigmoid': True}*, *loss_bbox: Union[mmengine.config.config.ConfigDict, dict] = {'bbox_format': 'xyxy', 'iou_mode': 'ciou', 'loss_weight': 7.5, 'reduction': 'sum', 'return_iou': False, 'type': 'IoULoss'}*, *loss_dfl={'loss_weight': 0.375, 'reduction': 'mean', 'type': 'mmdet.DistributionFocalLoss'}*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv8Head head used in *YOLOv8*.

参数

- **head_module** (ConfigDict or dict) –Base module used for YOLOv8Head
- **prior_generator** (*dict*) –Points generator feature maps in 2D points-based detectors.
- **bbox_coder** (ConfigDict or dict) –Config of bbox coder.
- **loss_cls** (ConfigDict or dict) –Config of classification loss.
- **loss_bbox** (ConfigDict or dict) –Config of localization loss.
- **loss_dfl** (ConfigDict or dict) –Config of Distribution Focal Loss.

- **train_cfg** (ConfigDict or dict, optional) –Training config of anchor head. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) –Testing config of anchor head. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.
Defaults to None.

loss_by_feat(*cls_scores: Sequence[torch.Tensor]*, *bbox_preds: Sequence[torch.Tensor]*, *bbox_dist_preds: Sequence[torch.Tensor]*, *batch_gt_instances: Sequence[mmengine.structures.instance_data.InstanceData]*, *batch_img_metas: Sequence[dict]*, *batch_gt_instances_ignore: Optional[List[mmengine.structures.instance_data.InstanceData]] = None*) → dict

Calculate the loss based on the features extracted by the detection head.

参数

- **cls_scores** (*Sequence[Tensor]*) –Box scores for each scale level, each is a 4D-tensor, the channel number is num_priors * num_classes.
- **bbox_preds** (*Sequence[Tensor]*) –Box energies / deltas for each scale level, each is a 4D-tensor, the channel number is num_priors * 4.
- **bbox_dist_preds** (*Sequence[Tensor]*) –Box distribution logits for each scale level with shape (bs, reg_max + 1, H*W, 4).
- **batch_gt_instances** (list[InstanceData]) –Batch of gt_instance. It usually includes bboxes and labels attributes.
- **batch_img_metas** (*list[dict]*) –Meta information of each image, e.g., image size, scaling factor, etc.
- **batch_gt_instances_ignore** (list[InstanceData], optional) –Batch of gt_instances_ignore. It includes bboxes attribute data that is ignored during training and testing. Defaults to None.

返回 A dictionary of losses.

返回类型 dict[str, Tensor]

special_init()

Since YOLO series algorithms will inherit from YOLOv5Head, but different algorithms have special initialization process.

The special_init function is designed to deal with this situation.

class mmyolo.models.dense_heads.**YOLOv8HeadModule**(*num_classes: int*, *in_channels: Union[int, Sequence]*, *widen_factor: float = 1.0*, *num_base_priors: int = 1*, *featmap_strides: Sequence[int] = (8, 16, 32)*, *reg_max: int = 16*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

YOLOv8HeadModule head module used in *YOLOv8*.

参数

- **num_classes** (*int*) –Number of categories excluding the background category.
- **in_channels** (*Union[int, Sequence]*) –Number of channels in the input feature map.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_base_priors** (*int*) –The number of priors (points) at a point on the feature grid.
- **featmap_strides** (*Sequence[int]*) –Downsample factor of each feature map. Defaults to [8, 16, 32].
- **reg_max** (*int*) –Max value of integral set :math: `{0, ..., reg_max-1}` in QFL setting. Defaults to 16.
- **norm_cfg** (ConfigDict or dict) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (ConfigDict or dict) –Config dict for activation layer. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.

Defaults to None.

forward(*x: Tuple[torch.Tensor]*) → Tuple[List]

Forward features from the upstream network.

参数 **x** (*Tuple[Tensor]*) –Features from the upstream network, each is a 4D-tensor.

返回 A tuple of multi-level classification scores, bbox predictions

返回类型 Tuple[List]

forward_single(*x: torch.Tensor*, *cls_pred: torch.nn.modules.container.ModuleList*, *reg_pred: torch.nn.modules.container.ModuleList*) → Tuple

Forward feature of a single scale level.

init_weights(*prior_prob=0.01*)

Initialize the weight and bias of PPYOLOE head.

61.4 detectors

class mmyolo.models.detectors.**YOLODetector**(*backbone: Union[mmengine.config.config.ConfigDict, dict]*, *neck: Union[mmengine.config.config.ConfigDict, dict]*, *bbox_head: Union[mmengine.config.config.ConfigDict, dict]*, *train_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *test_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *data_preprocessor: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, *use_syncbn: bool = True*)

Implementation of YOLO Series

参数

- **backbone** (ConfigDict or dict) –The backbone config.
- **neck** (ConfigDict or dict) –The neck config.
- **bbox_head** (ConfigDict or dict) –The bbox head config.
- **train_cfg** (ConfigDict or dict, optional) –The training config of YOLO. Defaults to None.
- **test_cfg** (ConfigDict or dict, optional) –The testing config of YOLO. Defaults to None.
- **data_preprocessor** (ConfigDict or dict, optional) –Config of DetDataPreprocessor to process the input data. Defaults to None.

:param init_cfg (ConfigDict or list[ConfigDict] or dict or: list[dict], optional): Initialization config dict.

Defaults to None.

参数 **use_syncbn** (*bool*) –whether to use SyncBatchNorm. Defaults to True.

61.5 layers

class mmyolo.models.layers.**BepC3StageBlock**(*in_channels: int*, *out_channels: int*, *num_blocks: int = 1*, *hidden_ratio: float = 0.5*, *concat_all_layer: bool = True*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*)

Beer-mug RepC3 Block.

参数

- **in_channels** (*int*) –Number of channels in the input image
- **out_channels** (*int*) –Number of channels produced by the convolution
- **num_blocks** (*int*) –Number of blocks. Defaults to 1
- **hidden_ratio** (*float*) –Hidden channel expansion. Default: 0.5
- **concat_all_layer** (*bool*) –Concat all layer when forward calculate. Default: True
- **block_cfg** (*dict*) –Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **norm_cfg** (*ConfigType*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*ConfigType*) –Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

注解: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmyolo.models.layers.**BiFusion**(*in_channels0: int*, *in_channels1: int*, *out_channels: int*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*)

BiFusion Block in YOLOv6.

BiFusion fuses current-, high- and low-level features. Compared with concatenation in PAN, it fuses an extra low-level feature.

参数

- **in_channels0** (*int*) –The channels of current-level feature.
- **in_channels1** (*int*) –The input channels of lower-level feature.
- **out_channels** (*int*) –The out channels of the BiFusion module.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

forward(*x: List[torch.Tensor]*) → torch.Tensor

Forward process :param x: The tensor list of length 3.

x[0]: The high-level feature. x[1]: The current-level feature. x[2]: The low-level feature.

class mmyolo.models.layers.**CSPLayerWithTwoConv**(*in_channels: int*, *out_channels: int*, *expand_ratio: float = 0.5*, *num_blocks: int = 1*, *add_identity: bool = True*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Cross Stage Partial Layer with 2 convolutions.

参数

- **in_channels** (*int*) –The input channels of the CSP layer.
- **out_channels** (*int*) –The output channels of the CSP layer.
- **expand_ratio** (*float*) –Ratio to adjust the number of channels of the hidden layer. Defaults to 0.5.
- **num_blocks** (*int*) –Number of blocks. Defaults to 1
- **add_identity** (*bool*) –Whether to add identity in blocks. Defaults to True.
- **conv_cfg** (*dict, optional*) –Config dict for convolution layer. Defaults to None, which means using conv2d.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN').
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).

:param init_cfg (ConfigDict or dict or list[dict] or: list[ConfigDict], optional): Initialization config dict.

Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process.

class mmyolo.models.layers.**DarknetBottleneck**(*in_channels: int*, *out_channels: int*, *expansion: float = 0.5*, *kernel_size: Sequence[int] = (1, 3)*, *padding: Sequence[int] = (0, 1)*, *add_identity: bool = True*, *use_depthwise: bool = False*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

The basic bottleneck block used in Darknet.

Each ResBlock consists of two ConvModules and the input is added to the final output. Each ConvModule is composed of Conv, BN, and LeakyReLU. The first convLayer has filter size of k1Xk1 and the second one has the filter size of k2Xk2.

Note: This DarknetBottleneck is little different from MMDet's, we can change the kernel size and padding for each conv.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The output channels of this Module.
- **expansion** (*float*) –The kernel size for hidden channel. Defaults to 0.5.
- **kernel_size** (*Sequence[int]*) –The kernel size of the convolution. Defaults to (1, 3).
- **padding** (*Sequence[int]*) –The padding size of the convolution. Defaults to (0, 1).
- **add_identity** (*bool*) –Whether to add identity to the out. Defaults to True
- **use_depthwise** (*bool*) –Whether to use depthwise separable convolution. Defaults to False
- **conv_cfg** (*dict*) –Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN').
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='Swish').

class mmyolo.models.layers.**EELANBlock**(*num_elan_block: int*, ***kwargs*)

Expand efficient layer aggregation networks for YOLOv7.

参数 **num_elan_block** (*int*) –The number of ELANBlock.

forward(*x: torch.Tensor*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

注解: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class mmyolo.models.layers.**ELANBlock**(*in_channels: int*, *out_channels: int*, *middle_ratio: float*, *block_ratio: float*, *num_blocks: int = 2*, *num_convs_in_block: int = 1*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Efficient layer aggregation networks for YOLOv7.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The out channels of this Module.
- **middle_ratio** (*float*) –The scaling ratio of the middle layer based on the in_channels.
- **block_ratio** (*float*) –The scaling ratio of the block layer based on the in_channels.
- **num_blocks** (*int*) –The number of blocks in the main branch. Defaults to 2.
- **num_convs_in_block** (*int*) –The number of convs pre block. Defaults to 1.
- **conv_cfg** (*dict*) –Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**EffectiveSELayer**(*channels: int*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'HSigmoid'}*)

Effective Squeeze-Excitation.

From *CenterMask : Real-Time Anchor-Free Instance Segmentation* arxiv (https://arxiv.org/abs/1911.06667) This code referenced to https://github.com/youngwanLEE/CenterMask/blob/

72147e8aae673fcaf4103ee90a6a6b73863e7fa1/maskrcnn_benchmark/modeling/backbone/vovnet.py#L108-L121 # noqa

参数

- **channels** (*int*) –The input and output channels of this Module.
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='HSigmoid').

forward(*x: torch.Tensor*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**ExpMomentumEMA**(*model: torch.nn.modules.module.Module*, *momentum: float = 0.0002*, *gamma: int = 2000*, *interval=1*, *device: Optional[torch.device] = None*, *update_buffers: bool = False*)

Exponential moving average (EMA) with exponential momentum strategy, which is used in YOLO.

参数

- **model** (*nn.Module*) –The model to be averaged.
- **momentum** (*float*) –

  The momentum used for updating ema parameter. Ema's parameters are updated with the formula:

  *averaged_param = (1-momentum) * averaged_param + momentum * source_param*. Defaults to 0.0002.

- **gamma** (*int*) –Use a larger momentum early in training and gradually annealing to a smaller value to update the ema model smoothly. The momentum is calculated as *(1 - momentum) * exp(-(1 + steps) / gamma) + momentum*. Defaults to 2000.
- **interval** (*int*) –Interval between two updates. Defaults to 1.
- **device** (*torch.device, optional*) –If provided, the averaged model will be stored on the `device`. Defaults to None.
- **update_buffers** (*bool*) –if True, it will compute running averages for both the parameters and the buffers of the model. Defaults to False.

avg_func(*averaged_param: torch.Tensor*, *source_param: torch.Tensor*, *steps: int*)

Compute the moving average of the parameters using the exponential momentum strategy.

参数

- **averaged_param** (*Tensor*) –The averaged parameters.
- **source_param** (*Tensor*) –The source parameters.
- **steps** (*int*) –The number of times the parameters have been updated.

update_parameters(*model: torch.nn.modules.module.Module*)

Update the parameters after each training step.

参数 **model** (*nn.Module*) –The model of the parameter needs to be updated.

class mmyolo.models.layers.**ImplicitA**(*in_channels: int*, *mean: float = 0.0*, *std: float = 0.02*)

Implicit add layer in YOLOv7.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **mean** (*float*) –Mean value of implicit module. Defaults to 0.
- **std** (*float*) –Std value of implicit module. Defaults to 0.02

forward(*x*)

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**ImplicitM**(*in_channels: int*, *mean: float = 1.0*, *std: float = 0.02*)

Implicit multiplier layer in YOLOv7.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **mean** (*float*) –Mean value of implicit module. Defaults to 1.
- **std** (*float*) –Std value of implicit module. Defaults to 0.02.

forward(*x*)

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**MaxPoolAndStrideConvBlock**(*in_channels: int*, *out_channels: int*, *maxpool_kernel_sizes: int = 2*, *use_in_channels_of_middle: bool = False*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Max pooling and stride conv layer for YOLOv7.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The out channels of this Module.
- **maxpool_kernel_sizes** (*int*) –kernel sizes of pooling layers. Defaults to 2.
- **use_in_channels_of_middle** (*bool*) –Whether to calculate middle channels based on in_channels. Defaults to False.
- **conv_cfg** (*dict*) –Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**PPYOLOEBasicBlock**(*in_channels: int*, *out_channels: int*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *shortcut: bool = True*, *use_alpha: bool = False*)

PPYOLOE Backbone BasicBlock.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The output channels of this Module.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.1, eps=1e-5).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **shortcut** (*bool*) –Whether to add inputs and outputs together
- **the end of this layer. Defaults to True.** (*at*) –
- **use_alpha** (*bool*) –Whether to use *alpha* parameter at 1x1 conv.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process. :param inputs: The input tensor. :type inputs: Tensor

返回 The output tensor.

返回类型 Tensor

class mmyolo.models.layers.**RepStageBlock**(*in_channels: int*, *out_channels: int*, *num_blocks: int = 1*, *bottle_block: torch.nn.modules.module.Module = <class 'mmyolo.models.layers.yolo_bricks.RepVGGBlock'>*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*)

RepStageBlock is a stage block with rep-style basic block.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The output channels of this Module.
- **num_blocks** (*int, tuple[int]*) –Number of blocks. Defaults to 1.
- **bottle_block** (*nn.Module*) –Basic unit of RepStage. Defaults to RepVGGBlock.
- **block_cfg** (*ConfigType*) –Config of RepStage. Defaults to 'RepVGGBlock'.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process.

参数 **x** (*Tensor*) –The input tensor.

返回 The output tensor.

返回类型 Tensor

class mmyolo.models.layers.**RepVGGBlock**(*in_channels: int*, *out_channels: int*, *kernel_size: Union[int, Tuple[int]] = 3*, *stride: Union[int, Tuple[int]] = 1*, *padding: Union[int, Tuple[int]] = 1*, *dilation: Union[int, Tuple[int]] = 1*, *groups: Optional[int] = 1*, *padding_mode: Optional[str] = 'zeros'*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *use_se: bool = False*, *use_alpha: bool = False*, *use_bn_first=True*, *deploy: bool = False*)

RepVGGBlock is a basic rep-style block, including training and deploy status This code is based on https://github.com/DingXiaoH/RepVGG/blob/main/repvgg.py.

参数

- **in_channels** (*int*) –Number of channels in the input image

- **out_channels** (*int*) –Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) –Size of the convolving kernel
- **stride** (*int or tuple*) –Stride of the convolution. Default: 1
- **padding** (*int, tuple*) –Padding added to all four sides of the input. Default: 1
- **dilation** (*int or tuple*) –Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) –Number of blocked connections from input channels to output channels. Default: 1
- **padding_mode** (*string, optional*) –Default: 'zeros'
- **use_se** (*bool*) –Whether to use se. Default: False
- **use_alpha** (*bool*) –Whether to use *alpha* parameter at 1x1 conv. In PPYOLOE+ model backbone, *use_alpha* will be set to True. Default: False.
- **use_bn_first** (*bool*) –Whether to use bn layer before conv. In YOLOv6 and YOLOv7, this will be set to True. In PPYOLOE, this will be set to False. Default: True.
- **deploy** (*bool*) –Whether in deploy mode. Default: False

forward(*inputs: torch.Tensor*) → torch.Tensor

Forward process. :param inputs: The input tensor. :type inputs: Tensor

返回 The output tensor.

返回类型 Tensor

get_equivalent_kernel_bias()

Derives the equivalent kernel and bias in a differentiable way.

返回 Equivalent kernel and bias

返回类型 tuple

switch_to_deploy()

Switch to deploy mode.

class mmyolo.models.layers.**SPPFBottleneck**(*in_channels: int*, *out_channels: int*, *kernel_sizes: Union[int, Sequence[int]] = 5*, *use_conv_first: bool = True*, *mid_channels_scale: float = 0.5*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Spatial pyramid pooling - Fast (SPPF) layer for YOLOv5, YOLOX and PPYOLOE by Glenn Jocher

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The output channels of this Module.
- **kernel_sizes** (*int, tuple[int]*) –Sequential or number of kernel sizes of pooling layers. Defaults to 5.
- **use_conv_first** (*bool*) –Whether to use conv before pooling layer. In YOLOv5 and YOLOX, the para set to True. In PPYOLOE, the para set to False. Defaults to True.
- **mid_channels_scale** (*float*) –Channel multiplier, multiply in_channels by this amount to get mid_channels. This parameter is valid only when use_conv_fist=True.Defaults to 0.5.
- **conv_cfg** (*dict*) –Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

forward(*x: torch.Tensor*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**SPPFCSPBlock**(*in_channels: int*, *out_channels: int*, *expand_ratio: float = 0.5*, *kernel_sizes: Union[int, Sequence[int]] = 5*, *is_tiny_version: bool = False*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Spatial pyramid pooling - Fast (SPPF) layer with CSP for YOLOv7

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The output channels of this Module.
- **expand_ratio** (*float*) –Expand ratio of SPPCSPBlock. Defaults to 0.5.
- **kernel_sizes** (*int, tuple[int]*) –Sequential or number of kernel sizes of pooling layers. Defaults to 5.
- **is_tiny_version** (*bool*) –Is tiny version of SPPFCSPBlock. If True, it means it is a yolov7 tiny model. Defaults to False.
- **conv_cfg** (*dict*) –Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

forward(*x*) → torch.Tensor

Forward process :param x: The input tensor. :type x: Tensor

class mmyolo.models.layers.**TinyDownSampleBlock**(*in_channels: int*, *out_channels: int*, *middle_ratio: float = 1.0*, *kernel_sizes: Union[int, Sequence[int]] = 3*, *conv_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'negative_slope': 0.1, 'type': 'LeakyReLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Down sample layer for YOLOv7-tiny.

参数

- **in_channels** (*int*) –The input channels of this Module.
- **out_channels** (*int*) –The out channels of this Module.
- **middle_ratio** (*float*) –The scaling ratio of the middle layer based on the in_channels. Defaults to 1.0.
- **kernel_sizes** (*int, tuple[int]*) –Sequential or number of kernel sizes of pooling layers. Defaults to 3.
- **conv_cfg** (*dict*) –Config dict for convolution layer. Defaults to None. which means using conv2d. Defaults to None.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='LeakyReLU', negative_slope=0.1).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

forward(*x*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

注解: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while

the latter silently ignores them.

61.6 losses

class mmyolo.models.losses.**IoULoss**(*iou_mode: str = 'ciou'*, *bbox_format: str = 'xywh'*, *eps: float = 1e-07*, *reduction: str = 'mean'*, *loss_weight: float = 1.0*, *return_iou: bool = True*)

IoULoss.

Computing the IoU loss between a set of predicted bboxes and target bboxes. :param iou_mode: Options are "ciou".

Defaults to "ciou".

参数

- **bbox_format** (*str*) –Options are "xywh" and "xyxy". Defaults to "xywh".
- **eps** (*float*) –Eps to avoid log(0).
- **reduction** (*str*) –Options are "none", "mean" and "sum".
- **loss_weight** (*float*) –Weight of loss.
- **return_iou** (*bool*) –If True, return loss and iou.

forward(*pred: torch.Tensor*, *target: torch.Tensor*, *weight: Optional[torch.Tensor] = None*, *avg_factor: Optional[float] = None*, *reduction_override: Optional[Union[str, bool]] = None*) → Tuple[torch.Tensor, torch.Tensor]

Forward function.

参数

- **pred** (*Tensor*) –Predicted bboxes of format (x1, y1, x2, y2) or (x, y, w, h),shape (n, 4).
- **target** (*Tensor*) –Corresponding gt bboxes, shape (n, 4).
- **weight** (*Tensor, optional*) –Element-wise weights.
- **avg_factor** (*float, optional*) –Average factor when computing the mean of losses.
- **reduction_override** (*str, bool, optional*) –Same as built-in losses of PyTorch. Defaults to None.

返回

返回类型 loss or tuple(loss, iou)

class mmyolo.models.losses.**OksLoss**(*metainfo: Optional[str] = None*, *loss_weight: float = 1.0*)

A PyTorch implementation of the Object Keypoint Similarity (OKS) loss as described in the paper "YOLO-Pose: Enhancing YOLO for Multi Person Pose Estimation Using Object Keypoint Similarity Loss" by Debapriya et al.

(2022). The OKS loss is used for keypoint-based object recognition and consists of a measure of the similarity between predicted and ground truth keypoint locations, adjusted by the size of the object in the image. The loss function takes as input the predicted keypoint locations, the ground truth keypoint locations, a mask indicating which keypoints are valid, and bounding boxes for the objects. :param metainfo: Path to a JSON file containing information

about the dataset's annotations.

参数 **loss_weight** (*float*) –Weight for the loss.

compute_oks(*output: torch.Tensor*, *target: torch.Tensor*, *target_weights: torch.Tensor*, *bboxes: Optional[torch.Tensor] = None*) → torch.Tensor

Calculates the OKS loss.

参数

- **output** (*Tensor*) –Predicted keypoints in shape N x k x 2, where N is batch size, k is the number of keypoints, and 2 are the xy coordinates.
- **target** (*Tensor*) –Ground truth keypoints in the same shape as output.
- **target_weights** (*Tensor*) –Mask of valid keypoints in shape N x k, with 1 for valid and 0 for invalid.
- **bboxes** (*Optional[Tensor]*) –Bounding boxes in shape N x 4, where 4 are the xyxy coordinates.

返回 The calculated OKS loss.

返回类型 Tensor

forward(*output: torch.Tensor*, *target: torch.Tensor*, *target_weights: torch.Tensor*, *bboxes: Optional[torch.Tensor] = None*) → torch.Tensor

Defines the computation performed at every call.

Should be overridden by all subclasses.

注解: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

mmyolo.models.losses.**bbox_overlaps**(*pred: torch.Tensor*, *target: torch.Tensor*, *iou_mode: str = 'ciou'*, *bbox_format: str = 'xywh'*, *siou_theta: float = 4.0*, *eps: float = 1e-07*) → torch.Tensor

Calculate overlap between two set of bboxes. Implementation of paper 'Enhancing Geometric Factors into Model Learning and Inference for Object Detection and Instance Segmentation.

In the CIoU implementation of YOLOv5 and MMDetection, there is a slight difference in the way the alpha parameter is computed.

mmdet version: alpha = (ious > 0.5).float() * v / (1 - ious + v)

YOLOv5 version: alpha = v / (v - ious + (1 + eps)

参数

- **pred** (*Tensor*) –Predicted bboxes of format (x1, y1, x2, y2) or (x, y, w, h),shape (n, 4).
- **target** (*Tensor*) –Corresponding gt bboxes, shape (n, 4).
- **iou_mode** (*str*) –Options are ('iou', 'ciou', 'giou', 'siou'). Defaults to "ciou".
- **bbox_format** (*str*) –Options are "xywh" and "xyxy". Defaults to "xywh".
- **siou_theta** (*float*) –siou_theta for SIoU when calculate shape cost. Defaults to 4.0.
- **eps** (*float*) –Eps to avoid log(0).

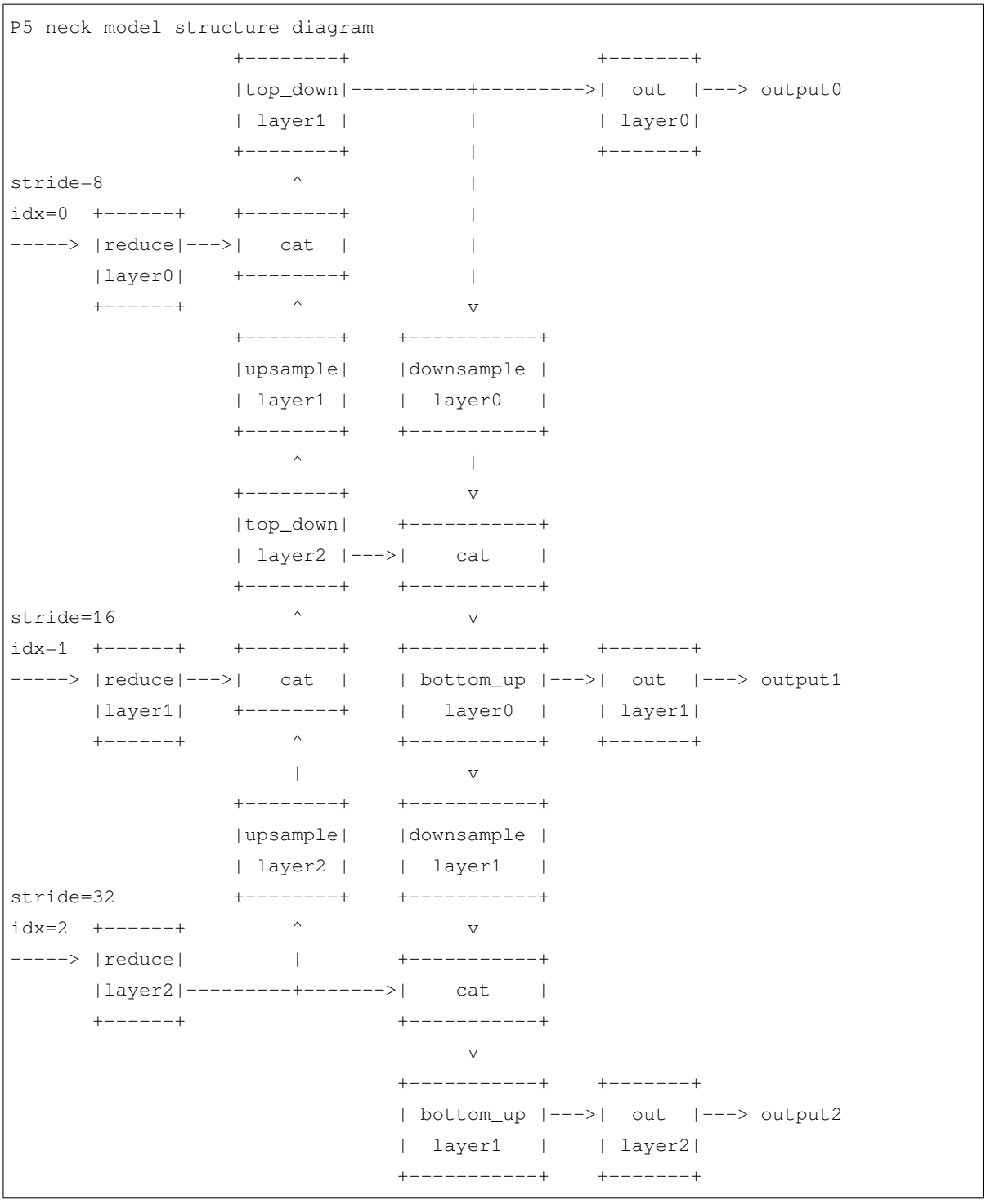
返回 shape (n,).

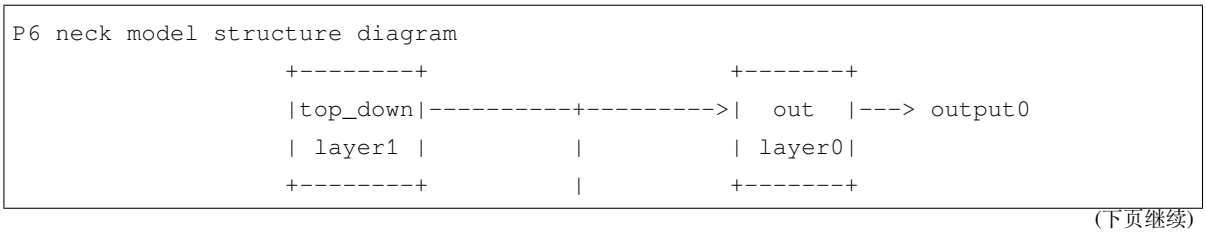
返回类型 Tensor

61.7 necks

class mmyolo.models.necks.**BaseYOLONeck**(*in_channels: List[int]*, *out_channels: Union[int, List[int]]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *upsample_feats_cat_first: bool = True*, *freeze_all: bool = False*, *norm_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *act_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, ***kwargs*)

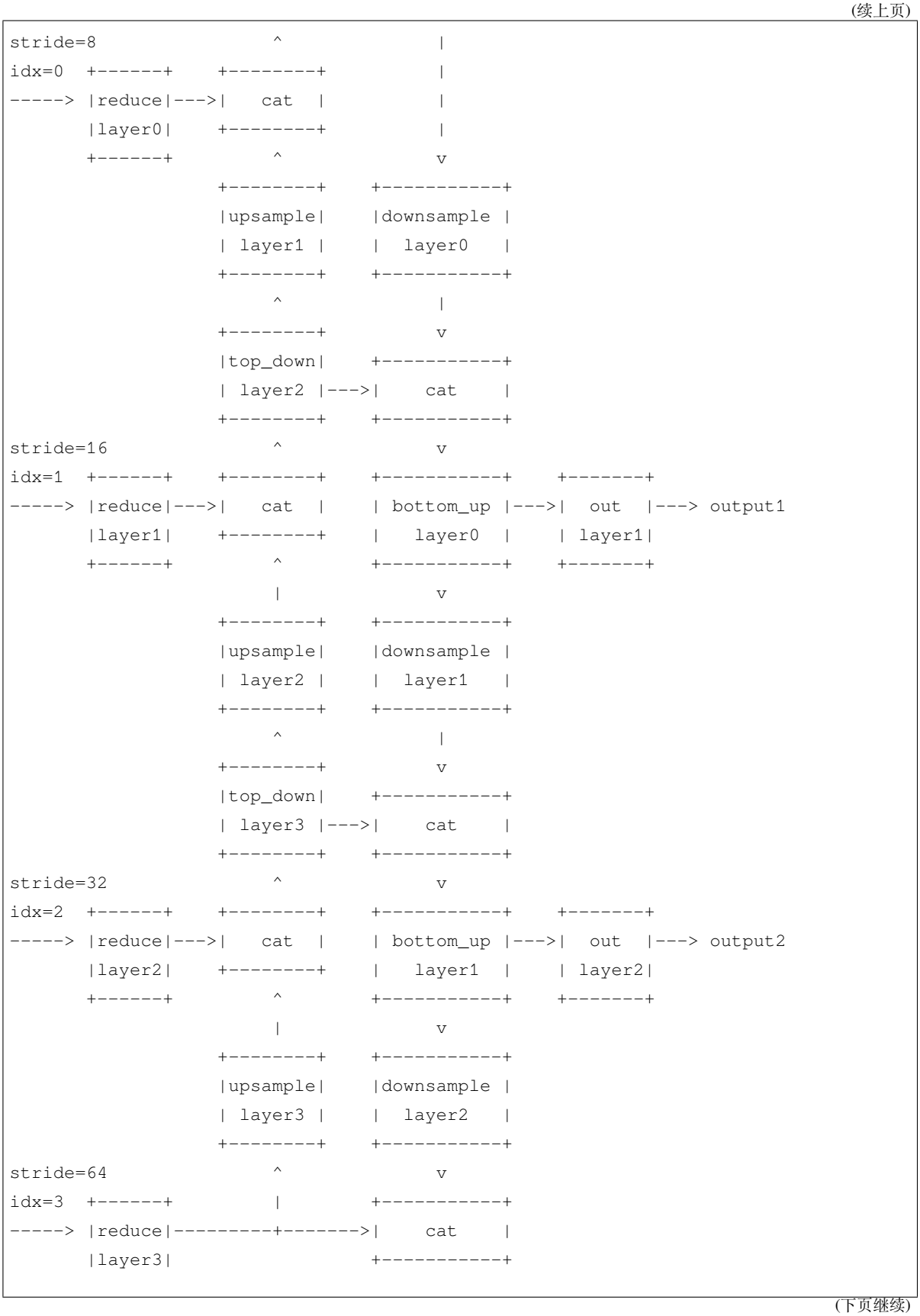
Base neck used in YOLO series.


P5 neck model structure diagram
top_down layer1
out layer0
output0
stride=8
idx=0
reduce layer0
cat
upsample layer1
downsample layer0
top_down layer2
cat
stride=16
idx=1
reduce layer1
cat
bottom_up layer0
out layer1
output1
upsample layer2
downsample layer1
stride=32
idx=2
reduce layer2
cat
bottom_up layer1
out layer2
output2


P6 neck model structure diagram
top_down layer1
out layer0
output0

(下页继续)

(续上页)


stride=8
idx=0
reduce
layer0
cat
upsample
layer1
top_down
layer2
downsample
layer0
stride=16
idx=1
reduce
layer1
cat
upsample
layer2
top_down
layer3
bottom_up
layer0
downsample
layer1
out
output1
stride=32
idx=2
reduce
layer2
cat
upsample
layer3
bottom_up
layer1
downsample
layer2
out
output2
stride=64
idx=3
reduce
layer3
cat

(下页继续)

(续上页)

```
      +------+                      v
                                +-----------+    +-------+
                                | bottom_up |--->|  out  |---> output3
                                |  layer2   |    | layer3|
                                +-----------+    +-------+
```

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **upsample_feats_cat_first** (*bool*) –Whether the output features are concat first after upsampling in the topdown module. Defaults to True. Currently only YOLOv7 is false.
- **freeze_all** (*bool*) –Whether to freeze the model. Defaults to False
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to None.
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to None.
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

abstract build_bottom_up_layer(*idx: int*)

build bottom up layer.

abstract build_downsample_layer(*idx: int*)

build downsample layer.

abstract build_out_layer(*idx: int*)

build out layer.

abstract build_reduce_layer(*idx: int*)

build reduce layer.

abstract build_top_down_layer(*idx: int*)

build top down layer.

abstract build_upsample_layer(*idx: int*)

build upsample layer.

forward(*inputs: List[torch.Tensor]*) → tuple

Forward function.

train(*mode=True*)

Convert the model into training mode while keep the normalization layer freezed.

class mmyolo.models.necks.**CSPNeXtPAFPN**(*in_channels: Sequence[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 3*, *freeze_all: bool = False*, *use_depthwise: bool = False*, *expand_ratio: float = 0.5*, *upsample_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'mode': 'nearest', 'scale_factor': 2}*, *conv_cfg: Optional[bool] = None*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = {'a': 2.23606797749979, 'distribution': 'uniform', 'layer': 'Conv2d', 'mode': 'fan_in', 'nonlinearity': 'leaky_relu', 'type': 'Kaiming'}*)

Path Aggregation Network with CSPNeXt blocks.

参数

- **in_channels** (*Sequence[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 3.
- **use_depthwise** (*bool*) –Whether to use depthwise separable convolution in blocks. Defaults to False.
- **expand_ratio** (*float*) –Ratio to adjust the number of channels of the hidden layer. Defaults to 0.5.
- **upsample_cfg** (*dict*) –Config dict for interpolate layer. Default: *dict(scale_factor=2, mode='nearest')*
- **conv_cfg** (*dict, optional*) –Config dict for convolution layer. Default: None, which means using conv2d.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Default: dict(type='BN')

- **act_cfg** (*dict*) –Config dict for activation layer. Default: dict(type='SiLU', inplace=True)
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Default: None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

参数 **idx** (*int*) –layer idx.

返回 The downsample layer.

返回类型 nn.Module

build_out_layer(*idx: int*) → torch.nn.modules.module.Module

build out layer.

参数 **idx** (*int*) –layer idx.

返回 The out layer.

返回类型 nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module

build reduce layer.

参数 **idx** (*int*) –layer idx.

返回 The reduce layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

build_upsample_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module

build upsample layer.

class mmyolo.models.necks.**PPYOLOECSPPAFPN**(*in_channels: List[int] = [256, 512, 1024]*, *out_channels: List[int] = [256, 512, 1024]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *freeze_all: bool = False*, *num_csplayer: int = 1*, *num_blocks_per_layer: int = 3*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'shortcut': False, 'type': 'PPYOLOEBasicBlock', 'use_alpha': False}*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 1e-05, 'momentum': 0.1, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *drop_block_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict]] = None*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*, *use_spp: bool = False*)

CSPPAN in PPYOLOE.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*List[int]*) –Number of output channels (used at each scale).
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **freeze_all** (*bool*) –Whether to freeze the model.
- **num_csplayer** (*int*) –Number of *CSPResLayer* in per layer. Defaults to 1.
- **num_blocks_per_layer** (*int*) –Number of blocks per *CSPResLayer*. Defaults to 3.
- **block_cfg** (*dict*) –Config dict for block. Defaults to dict(type=' PPYOLOEBasicBlock' , shortcut=True, use_alpha=False)
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type=' BN' , momentum=0.1, eps=1e-5).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type=' SiLU' , inplace=True).
- **drop_block_cfg** (*dict, optional*) –Drop block config. Defaults to None. If you want to use Drop block after *CSPResLayer*, you can set this para as dict(type='

mmdet.DropBlock', drop_prob=0.1, block_size=3, warm_iters=0).

- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.
- **use_spp** (*bool*) –Whether to use *SPP* in reduce layer. Defaults to False.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

参数 **idx** (*int*) –layer idx.

返回 The downsample layer.

返回类型 nn.Module

build_out_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module

build out layer.

build_reduce_layer(*idx: int*)

build reduce layer.

参数 **idx** (*int*) –layer idx.

返回 The reduce layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module

build upsample layer.

class mmyolo.models.necks.**YOLOXPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 3*, *use_depthwise: bool = False*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOX.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale).
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 1.
- **use_depthwise** (*bool*) –Whether to use depthwise separable convolution. Defaults to False.
- **freeze_all** (*bool*) –Whether to freeze the model. Defaults to False.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

参数 **idx** (*int*) –layer idx.

返回 The downsample layer.

返回类型 nn.Module

build_out_layer(*idx: int*) → torch.nn.modules.module.Module
build out layer.

参数 **idx** (*int*) –layer idx.

返回 The out layer.

返回类型 nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module
build reduce layer.

参数 **idx** (*int*) –layer idx.

返回 The reduce layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module
build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

build_upsample_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build upsample layer.

class mmyolo.models.necks.**YOLOv5PAFPN**(*in_channels: List[int]*, *out_channels: Union[List[int], int]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 1*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv5.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)

- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) –Whether to freeze the model
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

参数 **idx** (*int*) –layer idx.

返回 The downsample layer.

返回类型 nn.Module

build_out_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module

build out layer.

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module

build reduce layer.

参数 **idx** (*int*) –layer idx.

返回 The reduce layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*)

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

build_upsample_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module
build upsample layer.

init_weights()
Initialize the weights.

class mmyolo.models.necks.**YOLOv6CSPRepBiPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *hidden_ratio: float = 0.5*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6 3.0.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) –Whether to freeze the model.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) –Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').

- **block_act_cfg** (*dict*) –Config dict for activation layer used in each stage. Defaults to dict(type=’SiLU’, inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

class mmyolo.models.necks.**YOLOv6CSPRepPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *hidden_ratio: float = 0.5*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.

- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) –Whether to freeze the model.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) –Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **block_act_cfg** (*dict*) –Config dict for activation layer used in each stage. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

class mmyolo.models.necks.**YOLOv6RepBiPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6 3.0.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) –Whether to freeze the model.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) –Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module

build upsample layer.

参数 **idx** (*int*) –layer idx.

返回 The upsample layer.

返回类型 nn.Module

forward(*inputs: List[torch.Tensor]*) → tuple

Forward function.

class mmyolo.models.necks.**YOLOv6RepPAFPN**(*in_channels: List[int]*, *out_channels: int*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 12*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'ReLU'}*, *block_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'RepVGGBlock'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv6.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) –Whether to freeze the model.
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='ReLU', inplace=True).
- **block_cfg** (*dict*) –Config dict for the block used to build each layer. Defaults to dict(type='RepVGGBlock').

- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

参数 **idx** (*int*) –layer idx.

返回 The downsample layer.

返回类型 nn.Module

build_out_layer(**args*, ***kwargs*) → torch.nn.modules.module.Module

build out layer.

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module

build reduce layer.

参数 **idx** (*int*) –layer idx.

返回 The reduce layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module

build upsample layer.

参数 **idx** (*int*) –layer idx.

返回 The upsample layer.

返回类型 nn.Module

init_weights()

Initialize the weights.

class mmyolo.models.necks.**YOLOv7PAFPN**(*in_channels: List[int]*, *out_channels: List[int]*, *block_cfg: dict = {'block_ratio': 0.25, 'middle_ratio': 0.5, 'num_blocks': 4, 'num_convs_in_block': 1, 'type': 'ELANBlock'}*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *spp_expand_ratio: float = 0.5*, *is_tiny_version: bool = False*, *use_maxpool_in_downsample: bool = True*, *use_in_channels_in_downsample: bool = False*, *use_repconv_outs: bool = True*, *upsample_feats_cat_first: bool = False*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv7.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale).
- **block_cfg** (*dict*) –Config dict for block.
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **spp_expand_ratio** (*float*) –Expand ratio of SPPCSPBlock. Defaults to 0.5.
- **is_tiny_version** (*bool*) –Is tiny version of neck. If True, it means it is a yolov7 tiny model. Defaults to False.
- **use_maxpool_in_downsample** (*bool*) –Whether maxpooling is used in downsample layers. Defaults to True.
- **use_in_channels_in_downsample** (*bool*) –MaxPoolAndStrideConvBlock module input parameters. Defaults to False.
- **use_repconv_outs** (*bool*) –Whether to use *repconv* in the output layer. Defaults to True.
- **upsample_feats_cat_first** (*bool*) –Whether the output features are concat first after upsampling in the topdown module. Defaults to True. Currently only YOLOv7 is false.
- **freeze_all** (*bool*) –Whether to freeze the model. Defaults to False.

- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_downsample_layer(*idx: int*) → torch.nn.modules.module.Module

build downsample layer.

参数 **idx** (*int*) –layer idx.

返回 The downsample layer.

返回类型 nn.Module

build_out_layer(*idx: int*) → torch.nn.modules.module.Module

build out layer.

参数 **idx** (*int*) –layer idx.

返回 The out layer.

返回类型 nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module

build reduce layer.

参数 **idx** (*int*) –layer idx.

返回 The reduce layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

build_upsample_layer(*idx: int*) → torch.nn.modules.module.Module

build upsample layer.

class mmyolo.models.necks.**YOLOv8PAFPN**(*in_channels: List[int]*, *out_channels: Union[List[int], int]*, *deepen_factor: float = 1.0*, *widen_factor: float = 1.0*, *num_csp_blocks: int = 3*, *freeze_all: bool = False*, *norm_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'eps': 0.001, 'momentum': 0.03, 'type': 'BN'}*, *act_cfg: Union[mmengine.config.config.ConfigDict, dict] = {'inplace': True, 'type': 'SiLU'}*, *init_cfg: Optional[Union[mmengine.config.config.ConfigDict, dict, List[Union[dict, mmengine.config.config.ConfigDict]]]] = None*)

Path Aggregation Network used in YOLOv8.

参数

- **in_channels** (*List[int]*) –Number of input channels per scale.
- **out_channels** (*int*) –Number of output channels (used at each scale)
- **deepen_factor** (*float*) –Depth multiplier, multiply number of blocks in CSP layer by this amount. Defaults to 1.0.
- **widen_factor** (*float*) –Width multiplier, multiply number of channels in each layer by this amount. Defaults to 1.0.
- **num_csp_blocks** (*int*) –Number of bottlenecks in CSPLayer. Defaults to 1.
- **freeze_all** (*bool*) –Whether to freeze the model
- **norm_cfg** (*dict*) –Config dict for normalization layer. Defaults to dict(type='BN', momentum=0.03, eps=0.001).
- **act_cfg** (*dict*) –Config dict for activation layer. Defaults to dict(type='SiLU', inplace=True).
- **init_cfg** (*dict or list[dict], optional*) –Initialization config dict. Defaults to None.

build_bottom_up_layer(*idx: int*) → torch.nn.modules.module.Module

build bottom up layer.

参数 **idx** (*int*) –layer idx.

返回 The bottom up layer.

返回类型 nn.Module

build_reduce_layer(*idx: int*) → torch.nn.modules.module.Module

build reduce layer.

参数 **idx** (*int*) –layer idx.

返回 The reduce layer.

返回类型 nn.Module

build_top_down_layer(*idx: int*) → torch.nn.modules.module.Module

build top down layer.

参数 **idx** (*int*) –layer idx.

返回 The top down layer.

返回类型 nn.Module

61.8 task_modules

class mmyolo.models.task_modules.**BatchATSSAssigner**(*num_classes: int*, *iou_calculator: Union[mmengine.config.config.ConfigDict, dict] = {'type': 'mmdet.BboxOverlaps2D'}*, *topk: int = 9*)

Assign a batch of corresponding gt bboxes or background to each prior.

This code is based on https://github.com/meituan/YOLOv6/blob/main/yolov6/assigners/atss_assigner.py

Each proposal will be assigned with *0* or a positive integer indicating the ground truth index.

- 0: negative sample, no assigned gt
- positive integer: positive sample, index (1-based) of assigned gt

参数

- **num_classes** (*int*) –number of class
- **iou_calculator** (ConfigDict or dict) –Config dict for iou calculator. Defaults to dict(type='BboxOverlaps2D')
- **topk** (*int*) –number of priors selected in each level

forward(*pred_bboxes: torch.Tensor*, *priors: torch.Tensor*, *num_level_priors: List*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *pad_bbox_flag: torch.Tensor*) → dict

Assign gt to priors.

The assignment is done in following steps

1. compute iou between all prior (prior of all pyramid levels) and gt
2. compute center distance between all prior and gt
3. on each pyramid level, for each gt, select k prior whose center are closest to the gt center, so we total select k*l prior as candidates for each gt
4. get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold

5. select these candidates whose iou are greater than or equal to the threshold as positive
6. limit the positive sample’s center in gt

参数

- **pred_bboxes** (*Tensor*) –Predicted bounding boxes, shape(batch_size, num_priors, 4)
- **priors** (*Tensor*) –Model priors with stride, shape(num_priors, 4)
- **num_level_priors** (*List*) –Number of bboxes in each level, len(3)
- **gt_labels** (*Tensor*) –Ground truth label, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) –Ground truth bbox, shape(batch_size, num_gt, 4)
- **pad_bbox_flag** (*Tensor*) –Ground truth bbox mask, 1 means bbox, 0 means no bbox, shape(batch_size, num_gt, 1)

返回

Assigned result ’assigned_labels’(Tensor): shape(batch_size, num_gt) ‘assigned_bboxes’(Tensor): shape(batch_size, num_gt, 4) ‘assigned_scores’(Tensor):

shape(batch_size, num_gt, number_classes)

’fg_mask_pre_prior’(Tensor): shape(bs, num_gt)

返回类型 assigned_result (dict)

get_targets(*gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *assigned_gt_inds: torch.Tensor*, *fg_mask_pre_prior: torch.Tensor*, *num_priors: int*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Get target info.

参数

- **gt_labels** (*Tensor*) –Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) –Ground true bboxes, shape(batch_size, num_gt, 4)
- **assigned_gt_inds** (*Tensor*) –Assigned ground truth indexes, shape(batch_size, num_priors)
- **fg_mask_pre_prior** (*Tensor*) –Force ground truth matching mask, shape(batch_size, num_priors)
- **num_priors** (*int*) –Number of priors.
- **batch_size** (*int*) –Batch size.
- **num_gt** (*int*) –Number of ground truth.

返回

Assigned labels, shape(batch_size, num_priors)

assigned_bboxes (Tensor): Assigned bboxes, shape(batch_size, num_priors)

assigned_scores (Tensor): Assigned scores, shape(batch_size, num_priors)

返回类型 assigned_labels (Tensor)

select_topk_candidates(*distances: torch.Tensor*, *num_level_priors: List[int]*, *pad_bbox_flag: torch.Tensor*) → Tuple[torch.Tensor, torch.Tensor]

Selecting candidates based on the center distance.

参数

- **distances** (*Tensor*) –Distance between all bbox and gt, shape(batch_size, num_gt, num_priors)
- **num_level_priors** (*List[int]*) –Number of bboxes in each level, len(3)
- **pad_bbox_flag** (*Tensor*) –Ground truth bbox mask, shape(batch_size, num_gt, 1)

返回

Flag show that each level have topk candidates or not, shape(batch_size, num_gt, num_priors)

candidate_idxs (Tensor): Candidates index, shape(batch_size, num_gt, num_gt)

返回类型 is_in_candidate_list (Tensor)

static threshold_calculator(*is_in_candidate: List*, *candidate_idxs: torch.Tensor*, *overlaps: torch.Tensor*, *num_priors: int*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor]

Get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold.

参数

- **is_in_candidate** (*Tensor*) –Flag show that each level have topk candidates or not, shape(batch_size, num_gt, num_priors).
- **candidate_idxs** (*Tensor*) –Candidates index, shape(batch_size, num_gt, num_gt)
- **overlaps** (*Tensor*) –Overlaps area, shape(batch_size, num_gt, num_priors).
- **num_priors** (*int*) –Number of priors.
- **batch_size** (*int*) –Batch size.
- **num_gt** (*int*) –Number of ground truth.

返回

Overlap threshold of per ground truth, shape(batch_size, num_gt, 1).

candidate_overlaps (Tensor): Candidate overlaps, shape(batch_size, num_gt, num_priors).

返回类型 overlaps_thr_per_gt (Tensor)

class mmyolo.models.task_modules.**BatchTaskAlignedAssigner**(*num_classes: int*, *topk: int = 13*, *alpha: float = 1.0*, *beta: float = 6.0*, *eps: float = 1e-07*, *use_ciou: bool = False*)

This code referenced to https://github.com/meituan/YOLOv6/blob/main/yolov6/ assigners/tal_assigner.py. Batch Task aligned assigner base on the paper: TOOD: Task-aligned One-stage Object Detection.. Assign a corresponding gt bboxes or background to a batch of predicted bboxes. Each bbox will be assigned with *0* or a positive integer indicating the ground truth index. - 0: negative sample, no assigned gt - positive integer: positive sample, index (1-based) of assigned gt :param num_classes: number of class :type num_classes: int :param topk: number of bbox selected in each level :type topk: int :param alpha: Hyper-parameters related to alignment_metrics.

Defaults to 1.0

参数

- **beta** (*float*) –Hyper-parameters related to alignment_metrics. Defaults to 6.
- **eps** (*float*) –Eps to avoid log(0). Default set to 1e-9
- **use_ciou** (*bool*) –Whether to use ciou while calculating iou. Defaults to False.

forward(*pred_bboxes: torch.Tensor*, *pred_scores: torch.Tensor*, *priors: torch.Tensor*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *pad_bbox_flag: torch.Tensor*) → dict

Assign gt to bboxes.

The assignment is done in following steps 1. compute alignment metric between all bbox (bbox of all pyramid levels) and gt

2. select top-k bbox as candidates for each gt
3. limit the positive sample’s center in gt (because the anchor-free detector only can predict positive distance)

参数

- **pred_bboxes** (*Tensor*) –Predict bboxes, shape(batch_size, num_priors, 4)
- **pred_scores** (*Tensor*) –Scores of predict bboxes, shape(batch_size, num_priors, num_classes)
- **priors** (*Tensor*) –Model priors, shape (num_priors, 4)
- **gt_labels** (*Tensor*) –Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) –Ground true bboxes, shape(batch_size, num_gt, 4)
- **pad_bbox_flag** (*Tensor*) –Ground truth bbox mask, 1 means bbox, 0 means no bbox, shape(batch_size, num_gt, 1)

返回

assigned_labels (Tensor): Assigned labels, shape(batch_size, num_priors)

assigned_bboxes (Tensor): Assigned boxes, shape(batch_size, num_priors, 4)

assigned_scores (Tensor): Assigned scores, shape(batch_size, num_priors, num_classes)

fg_mask_pre_prior (Tensor): Force ground truth matching mask, shape(batch_size, num_priors)

返回类型 assigned_result (dict) Assigned result

get_box_metrics(*pred_bboxes: torch.Tensor*, *pred_scores: torch.Tensor*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor]

Compute alignment metric between all bbox and gt.

参数

- **pred_bboxes** (*Tensor*) –Predict bboxes, shape(batch_size, num_priors, 4)
- **pred_scores** (*Tensor*) –Scores of predict bbox, shape(batch_size, num_priors, num_classes)
- **gt_labels** (*Tensor*) –Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) –Ground true bboxes, shape(batch_size, num_gt, 4)
- **batch_size** (*int*) –Batch size.
- **num_gt** (*int*) –Number of ground truth.

返回

Align metric, shape(batch_size, num_gt, num_priors)

overlaps (Tensor): Overlaps, shape(batch_size, num_gt, num_priors)

返回类型 alignment_metrics (Tensor)

get_pos_mask(*pred_bboxes: torch.Tensor*, *pred_scores: torch.Tensor*, *priors: torch.Tensor*, *gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *pad_bbox_flag: torch.Tensor*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Get possible mask.

参数

- **pred_bboxes** (*Tensor*) –Predict bboxes, shape(batch_size, num_priors, 4)
- **pred_scores** (*Tensor*) –Scores of predict bbox, shape(batch_size, num_priors, num_classes)
- **priors** (*Tensor*) –Model priors, shape (num_priors, 2)
- **gt_labels** (*Tensor*) –Ground true labels, shape(batch_size, num_gt, 1)

- **gt_bboxes** (*Tensor*) –Ground true bboxes, shape(batch_size, num_gt, 4)
- **pad_bbox_flag** (*Tensor*) –Ground truth bbox mask, 1 means bbox, 0 means no bbox, shape(batch_size, num_gt, 1)
- **batch_size** (*int*) –Batch size.
- **num_gt** (*int*) –Number of ground truth.

返回

Possible mask, shape(batch_size, num_gt, num_priors)

alignment_metrics (Tensor): Alignment metrics, shape(batch_size, num_gt, num_priors)

overlaps (Tensor): Overlaps of gt_bboxes and pred_bboxes, shape(batch_size, num_gt, num_priors)

返回类型 pos_mask (Tensor)

get_targets(*gt_labels: torch.Tensor*, *gt_bboxes: torch.Tensor*, *assigned_gt_idxs: torch.Tensor*, *fg_mask_pre_prior: torch.Tensor*, *batch_size: int*, *num_gt: int*) → Tuple[torch.Tensor, torch.Tensor, torch.Tensor]

Get assigner info.

参数

- **gt_labels** (*Tensor*) –Ground true labels, shape(batch_size, num_gt, 1)
- **gt_bboxes** (*Tensor*) –Ground true bboxes, shape(batch_size, num_gt, 4)
- **assigned_gt_idxs** (*Tensor*) –Assigned ground truth indexes, shape(batch_size, num_priors)
- **fg_mask_pre_prior** (*Tensor*) –Force ground truth matching mask, shape(batch_size, num_priors)
- **batch_size** (*int*) –Batch size.
- **num_gt** (*int*) –Number of ground truth.

返回

Assigned labels, shape(batch_size, num_priors)

assigned_bboxes (Tensor): Assigned bboxes, shape(batch_size, num_priors)

assigned_scores (Tensor): Assigned scores, shape(batch_size, num_priors)

返回类型 assigned_labels (Tensor)

select_topk_candidates(*alignment_gt_metrics: torch.Tensor*, *using_largest_topk: bool = True*, *topk_mask: Optional[torch.Tensor] = None*) → torch.Tensor

Compute alignment metric between all bbox and gt.

参数

- **alignment_gt_metrics** (*Tensor*) –Alignment metric of gt candidates, shape(batch_size, num_gt, num_priors)
- **using_largest_topk** (*bool*) –Controls whether to using largest or smallest elements.
- **topk_mask** (*Tensor*) –Topk mask, shape(batch_size, num_gt, self.topk)

返回

Topk candidates mask, shape(batch_size, num_gt, num_priors)

返回类型 Tensor

class mmyolo.models.task_modules.**YOLOXBBoxCoder**(*use_box_type: bool = False*, ***kwargs*)

YOLOX BBox coder.

This decoder decodes pred bboxes (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

decode(*priors: torch.Tensor*, *pred_bboxes: torch.Tensor*, *stride: Union[torch.Tensor, int]*) → torch.Tensor

Decode regression results (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

参数

- **priors** (*torch.Tensor*) –Basic boxes or points, e.g. anchors.
- **pred_bboxes** (*torch.Tensor*) –Encoded boxes with shape
- **stride** (*torch.Tensor | int*) –Strides of bboxes.

返回 Decoded boxes.

返回类型 torch.Tensor

encode(***kwargs*)

Encode deltas between bboxes and ground truth boxes.

class mmyolo.models.task_modules.**YOLOv5BBoxCoder**(*use_box_type: bool = False*, ***kwargs*)

YOLOv5 BBox coder.

This decoder decodes pred bboxes (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

decode(*priors: torch.Tensor*, *pred_bboxes: torch.Tensor*, *stride: Union[torch.Tensor, int]*) → torch.Tensor

Decode regression results (delta_x, delta_x, w, h) to bboxes (tl_x, tl_y, br_x, br_y).

参数

- **priors** (*torch.Tensor*) –Basic boxes or points, e.g. anchors.
- **pred_bboxes** (*torch.Tensor*) –Encoded boxes with shape
- **stride** (*torch.Tensor | int*) –Strides of bboxes.

返回 Decoded boxes.

返回类型 torch.Tensor

encode(***kwargs*)

Encode deltas between bboxes and ground truth boxes.

61.9 utils

class mmyolo.models.utils.**OutputSaveFunctionWrapper**(*func: Callable*, *spec: Optional[Dict]*)

A class that wraps a function and saves its outputs.

This class can be used to decorate a function to save its outputs. It wraps the function with a *__call__* method that calls the original function and saves the results in a log attribute. :param func: A function to wrap. :type func: Callable :param spec: A dictionary of global variables to use as the

namespace for the wrapper. If *None*, the global namespace of the original function is used.

class mmyolo.models.utils.**OutputSaveObjectWrapper**(*obj: Any*)

A wrapper class that saves the output of function calls on an object.

clear()

Clears the log of function call outputs.

mmyolo.models.utils.**gt_instances_preprocess**(*batch_gt_instances: Union[torch.Tensor, Sequence]*, *batch_size: int*) → torch.Tensor

Split batch_gt_instances with batch size.

From [all_gt_bboxes, box_dim+2] to [batch_size, number_gt, box_dim+1]. For horizontal box, box_dim=4, for rotated box, box_dim=5

If some shape of single batch smaller than gt bbox len, then using zeros to fill.

参数

- **batch_gt_instances** (*Sequence[Tensor]*) –Ground truth instances for whole batch, shape [all_gt_bboxes, box_dim+2]
- **batch_size** (*int*) –Batch size.

返回

batch gt instances data, shape [batch_size, number_gt, box_dim+1]

返回类型 Tensor

mmyolo.models.utils.**make_divisible**(*x: float*, *widen_factor: float = 1.0*, *divisor: int = 8*) → int

Make sure that x*widen_factor is divisible by divisor.

mmyolo.models.utils.**make_round**(*x: float*, *deepen_factor: float = 1.0*) → int

Make sure that x*deepen_factor becomes an integer not less than 1.

CHAPTER 62

mmyolo.utils

CHAPTER 63

English

CHAPTER 64

简体中文

CHAPTER 65

Indices and tables

- genindex
- search

Python 模块索引

m

索引

C

D

E

F

G

I

L

M

O

P

R

S

T

U

W

Y

模块