MMYOLO

发布 *0.1.1*

MMYOLO Authors

2022 年 10 月 13 日

开启 MMYOLO 之旅

CHAPTER 1

概述

本章向您介绍 MMYOLO 的整体框架，并提供详细的教程链接。

1.1 什么是 MMYOLO

MMYOLO 是一个 YOLO 系列的算法工具箱，目前仅实现了目标检测任务，后续会支持实例分割、全景分割和关键点检测等多种任务。其包括丰富的目标检测算法以及相关的组件和模块，下面是它的整体框架：

MMYOLO 文件结构和 MMDetection 完全一致。为了能够充分复用 MMDetection 代码，MMYOLO 仅包括定制内容，其由 3 个主要部分组成：`datasets`、`models`、`engine`。

- **datasets** 支持用于目标检测的各种数据集。
  - **transforms** 包含各种数据增强变换。
- **models** 是检测器最重要的部分，包含检测器的不同组件。
  - **detectors** 定义所有检测模型类。
  - **data_preprocessors** 用于预处理模型的输入数据。
  - **backbones** 包含各种骨干网络
  - **necks** 包含各种模型颈部组件
  - **dense_heads** 包含执行密集预测的各种检测头。
  - **losses** 包含各种损失函数
  - **task_modules** 为检测任务提供模块。例如 assigners、samplers、box coders 和 prior generators。

  - **layers** 提供了一些基本的神经网络层

- **engine** 是运行时组件的一部分。
  - **optimizers** 提供优化器和优化器封装。
  - **hooks** 提供 runner 的各种钩子。

1.2 如何使用本指南

以下是 MMYOLO 的详细指南：

1. 安装说明见开始你的第一步
2. MMYOLO 的基本使用方法请参考以下教程：
   - 训练和测试
   - 从入门到部署全流程
   - 实用工具
3. YOLO 系列算法实现和全解析教程：
   - 必备基础
   - 原理和实现全解析
4. 参考以下教程深入了解：
   - 数据流
   - How to
5. 解读文章和资源汇总

CHAPTER 2

开始你的第一步

2.1 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。它需要 Python 3.6 以上，CUDA 9.2 以上和 PyTorch 1.7 以上。

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 open-mmlab python=3.8 -y
conda activate open-mmlab
```

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
```

2.2 安装流程

2.2.1 最佳实践

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

```
pip install -U openmim
mim install "mmengine==0.1.0"
mim install "mmcv>=2.0.0rc1,<2.1.0"
mim install "mmdet>=3.0.0rc1,<3.1.0"
```

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
pip 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"
```

2.3 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` and `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
# --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
from mmyolo.utils import register_all_modules

register_all_modules()
```

(下页继续)

(续上页)

```
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。

2.3.1 自定义安装

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）。

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==0.1.0"
```

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>=2.0.0rc1" -f https://download.openmmlab.com/mmcv/dist/cu116/torch1.
↪12.0/index.html
```

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：

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==0.1.0"
!mim install "mmcv>=2.0.0rc1,<2.1.0"
!mim install "mmdet>=3.0.0.rc1"
```

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。

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
```

2.3.2 排除故障

如果你在安装过程中遇到一些问题，请先查看*FAQ* 页面。

如果没有找到解决方案，你也可以在 GitHub 上 打开一个问题。

2.3.3 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代码：

```
PYTHONPATH="$(dirname $0)/..":$PYTHONPATH
```

CHAPTER 3

训练 & 测试

MMYOLO 在 Model Zoo 中提供了诸多检测模型。本文档将展示如何使用这些模型和数据集来执行常见的训练和测试任务：

3.1 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。

3.1.1 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 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：

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
```

(下页继续)

(续上页)

```
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
```

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
```

(下页继续)

(续上页)

```
        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'
        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
```

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
file_client_args = dict(backend='disk')  # 文件读取后端的配置，默认从硬盘读取
```

(下页继续)

(续上页)

```
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
        file_client_args=file_client_args),  # 文件读取后端的配置，默认从硬盘读取
    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.+
    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'
        }),
```

(下页继续)

(续上页)

```
    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
    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
        file_client_args=file_client_args),  # 文件读取后端的配置，默认从硬盘读取
    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(
```

(下页继续)

(续上页)

```
        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
    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(
        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
```

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
```

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/en/latest/tutorials/optimizer.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 MMYOLO 中，未引入任何参数调度器。

```
param_scheduler = None
```

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
        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)
]
```

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/en/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
)
```

(下页继续)

(续上页)

```
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。
```

3.1.2 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。

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 YOLOv5 的主干网络的某些内容：

```
model = dict(
    type='YOLODetector',
    data_preprocessor=dict(...),
    backbone=dict(
        type='YOLOv5CSPDarknet',
        deepen_factor=deepen_factor,
        widen_factor=widen_factor,
        norm_cfg=dict(type='BN', momentum=0.03, eps=0.001),
        act_cfg=dict(type='SiLU', inplace=True)),
    neck=dict(...),
    bbox_head=dict(...))
```

基础配置的 YOLOv5 使用 YOLOv5CSPDarknet，在需要将主干网络改成 YOLOv6EfficientRep 的时候，因为 YOLOv5CSPDarknet 和 YOLOv6EfficientRep 中有不同的字段，需要使用 _delete_=True 将新的键去替换 backbone 域内所有老的键。

```
_base_ = '../yolov5/yolov5_s-v61_syncbn_8xb16-300e_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(...))
```

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',
        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'))
```

(下页继续)

(续上页)

```
]

test_pipeline = [
    dict(
        type='LoadImageFromFile',
        file_client_args={{_base_.file_client_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),
    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),
    ...)
```

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'

file_client_args = {{_base_.file_client_args}}  # 变量 file_client_args 等于 _base_ 中定义的 file_client_args
```

(下页继续)

(续上页)

```
pre_transform = _base_.pre_transform # 变量 pre_transform 等于 _base_ 中定义的 pre_
↪transform
```

3.1.3 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 'LoadImageFromWebcam'，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。

3.1.4 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。bN 表示每个 GPU 上的 batch 大小为 N。例如 4x4b 是 4 个 GPU 每个 GPU 4 张图的缩写。如果没有注明，默认为 8 卡每卡 2 张图。
  - {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 4

从入门到部署全流程

4.1 YOLOv5 从入门到部署全流程

4.1.1 环境安装

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。

```
conda create -n open-mmlab python=3.8 -y
conda activate open-mmlab
conda install pytorch torchvision -c pytorch
# conda install pytorch torchvision cpuonly -c pytorch
pip install -U openmim
mim install "mmengine==0.1.0"
mim install "mmcv>=2.0.0rc1,<2.1.0"
mim install "mmdet>=3.0.0rc0,<3.1.0"
# for albumentations
git clone https://github.com/open-mmlab/mmyolo.git
cd mmyolo
# Install albumentations
pip 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。
```

详细环境配置操作请查看*get_started*

4.1.2 数据集准备

本文选取不到 40MB 大小的 balloon 气球数据集作为 MMYOLO 的学习数据集。

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

执行以上命令，下载数据集并转化格式后，balloon 数据集在 `data` 文件夹中准备好了，`train.json` 和 `val.json` 便是 coco 格式的标注文件了。

4.1.3 config 文件准备

在 `configs/yolov5` 文件夹下新建 `yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py` 配置文件，并把以下内容复制配置文件中。

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

data_root = 'data/balloon/'

train_batch_size_per_gpu = 4
train_num_workers = 2

metainfo = {
    'CLASSES': ('balloon', ),
    'PALETTE': [
        (220, 20, 60),
    ]
}

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/'),
        ann_file='train.json'))

val_dataloader = dict(
    dataset=dict(
        data_root=data_root,
```

(下页继续)

(续上页)

```
        metainfo=metainfo,
        data_prefix=dict(img='val/'),
        ann_file='val.json'))

test_dataloader = val_dataloader

val_evaluator = dict(ann_file=data_root + 'val.json')

test_evaluator = val_evaluator

model = dict(bbox_head=dict(head_module=dict(num_classes=1)))

default_hooks = dict(logger=dict(interval=1))
```

以上配置从 `./yolov5_s-v61_syncbn_fast_8xb16-300e_coco.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。我们将 logger 的 `interval` 设置为 1 的原因是，每进行 `interval` 次 iteration 会输出一次 loss 相关的日志，而我们选取气球数据集比较小，`interval` 太大我们将看不到 loss 相关日志的输出。

4.1.4 训练

```
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py
```

运行以上训练命令，`work_dirs/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon` 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。

中断后恢复训练

如果训练中途停止，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_syncbn_fast_1xb4-300e_balloon.py --
↪resume
```

加载预训练权重微调

经过测试，相比不加载预训练模型，加载 YOLOv5-s 预训练模型在气球数据集上训练和验证 coco/bbox_mAP 能涨 30 多个百分点。

1. 下载 COCO 数据集预训练权重

```
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
```

2. 加载预训练模型进行训练

```
cd mmyolo
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py \
                      --cfg-options load_from='yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco_20220918_084700-86e02187.pth' custom_hooks=None
```

注意：原则上在微调阶段应该将 `EMAHook` 的 `strict_load` 初始化参数设置为 `False` 即命令为 `custom_hooks.0.strict_load=False`。但由于 MMEngine v0.1.0 为初期开发版本，目前这样设置会出现问题。因此暂时只能通过命令 `custom_hooks=None`，关闭 `custom_hooks` 使用，从而正确加载预训练权重。预计会在下个版本修复此问题。

3. 冻结 backbone 进行训练

通过 config 文件或者命令行中设置 model.backbone.frozen_stages=4 冻结 backbone 的 4 个 stages。

```
# 命令行中设置 model.backbone.frozen_stages=4
cd mmyolo
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py \
                      --cfg-options load_from='yolov5_s-v61_syncbn_fast_8xb16-300e_
↪coco_20220918_084700-86e02187.pth' model.backbone.frozen_stages=4 custom_hooks=None
```

训练验证中可视化相关

验证阶段可视化

我们将 `configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py` 中的 `default_hooks` 的 `visualization` 进行修改，设置 `draw` 为 `True`，`interval` 为 2。

```
default_hooks = dict(
    logger=dict(interval=1),
    visualization=dict(draw=True, interval=2),
)
```

重新运行以下训练命令，在验证评估的过程中，每 interval 张图片就会保存一张标注结果和预测结果的拼图到 work_dirs/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon/{timestamp}/vis_data/vis_image 文件夹中了。

```
python tools/train.py configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py
```

可视化后端使用

MMEngine 支持本地、TensorBoard 以及 wandb 等多种后端。

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_syncbn_fast_1xb4-300e_balloon.py 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_syncbn_fast_1xb4-300e_balloon.py
```

Tensorboard 可视化使用

安装 Tensorboard 环境

```
pip install tensorboard
```

同上述在配置文件 configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py 中添加 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_syncbn_fast_1xb4-300e_balloon/{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_1xb4-300e_balloon
```

4.1.5 模型测试

如果你训练时候设置了 custom_hooks=None，那么在模型测试过程中依然需要设置 custom_hooks=None

```
python tools/test.py configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py \
                     work_dirs/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon/epoch_300.
↪pth \
                     --show-dir show_results  --cfg-options custom_hooks=None
```

如果你没有设置 custom_hooks=None，那么测试命令如下：

```
python tools/test.py configs/yolov5/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon.py \
                     work_dirs/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon/epoch_300.
↪pth \
                     --show-dir show_results
```

运行以上测试命令，推理结果图片会自动保存至 work_dirs/yolov5_s-v61_syncbn_fast_1xb4-300e_balloon/{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。

4.1.6 模型部署

正在准备中，敬请期待！

CHAPTER 5

实用工具

5.1 可视化

5.1.1 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。

5.1.2 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`：从输入的 C 维度中先在空间维度 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。

5.1.3 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 \
```

(下页继续)

(续上页)

```
                                yolov5_s-v61_syncbn_fast_8xb16-300e_coco_20220918_
↪084700-86e02187.pth \
                                --target-layers backbone \
                                --channel-reduction select_max
```

(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_fast_syncbn_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 \
                                --out-file 4.jpg
```

(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
```

5.2 实用工具

我们在 tools/ 文件夹下提供很多实用工具。除此之外，你也可以通过 MIM 来快速运行 OpenMMLab 的其他开源库。

以 MMDetection 为例，如果想利用 print_config.py，你可以直接采用如下命令，而无需复制源码到 MMYOLO 库中。

```
mim run mmdet print_config [CONFIG]
```

注意：上述命令能够成功的前提是 MMDetection 库必须通过 MIM 来安装。

5.2.1 可视化

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} \
                                                [--ann_file ${ANN_FILE}] \
                                                [--img_dir ${IMG_DIR}] \
                                                [--wait-time ${WAIT_TIME}] \
                                                [--disp-all] [--category-names␣
↪CATEGORY_NAMES [CATEGORY_NAMES ...]] \
                                                [--shuffle]
```

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/coco/' \
                                                --ann_file 'annotations/instances_
↪train2017.json' \
```

(下页继续)

(续上页)

```
                                                --img_dir 'train2017' \
                                                --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/coco/' \
                                                --ann_file 'annotations/instances_
↪train2017.json' \
                                                --img_dir 'train2017' \
                                                --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/coco/' \
                                                --ann_file 'annotations/instances_
↪train2017.json' \
                                                --img_dir 'train2017' \
                                                --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/coco/' \
                                                --ann_file 'annotations/instances_
↪train2017.json' \
                                                --img_dir 'train2017' \
                                                --disp-all \
                                                --shuffle
```

可视化数据集

脚本 tools/analysis_tools/browse_dataset.py 能够帮助用户去直接窗口可视化数据集的原始图片 + 展示标签的图片，或者保存可视化图片到指定文件夹内。

```
python tools/analysis_tools/browse_dataset.py ${CONFIG} \
                                              [-h] \
                                              [--output-dir ${OUTPUT_DIR}] \
                                              [--not-show] \
                                              [--show-interval ${SHOW_INTERVAL}]
```

例子：

1. 使用 config 文件 configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py 可视化图片，图片直接弹出显示，同时保存到目录 work-dir/browse_dataset：

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_
↪8xb16-300e_coco.py' \
                                              --output-dir 'work-dir/browse_dataset'
```

2. 使用 config 文件 configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py 可视化图片，图片直接弹出显示，每张图片持续 10 秒，同时保存到目录 work-dir/browse_dataset：

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_
↪8xb16-300e_coco.py' \
                                              --output-dir 'work-dir/browse_dataset'␣
↪\
                                              --show-interval 10
```

3. 使用 config 文件 configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py 可视化图片，图片直接弹出显示，每张图片持续 10 秒，图片不进行保存：

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_
↪8xb16-300e_coco.py' \
                                              --show-interval 10
```

4. 使用 config 文件 configs/yolov5/yolov5_s-v61_syncbn_8xb16-300e_coco.py 可视化图片，图片不直接弹出显示，仅保存到目录 work-dir/browse_dataset：

```
python tools/analysis_tools/browse_dataset.py 'configs/yolov5/yolov5_s-v61_syncbn_
↪8xb16-300e_coco.py' \
                                              --output-dir 'work-dir/browse_dataset'␣
↪\
                                              --not-show
```

5.2.2 数据集转换

文件夹 tools/data_converters/ 包含工具将 balloon 数据集（该小型数据集仅作为入门使用）转换成 COCO 的格式。

关于该脚本的详细说明，请看*YOLOv5 从入门到部署全流程* 中 数据集准备小节。

```
python tools/dataset_converters/balloon2coco.py
```

5.2.3 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 lvis
python tools/misc/download_dataset.py --dataset-name balloon [--save-dir ${SAVE_DIR}]␣
↪[--unzip]
```

5.2.4 模型转换

文件夹 tools/analysis_tools/ 下的三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。

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。

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 6

必备基础

6.1 模型设计相关说明

6.1.1 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。

BaseBackbone

如上图所示，对于 P5 而言，BaseBackbone 包括 1 个 stem 层 + 4 个 stage 层的类似 ResNet 的基础结构，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。

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。

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。

6.1.2 HeadModule

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),
```

(下页继续)

(续上页)

```
    act_cfg=dict(type='Swish'),
    ...
    loss_obj=dict(
        type='CrossEntropyLoss',
        use_sigmoid=True,
        reduction='sum',
        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 7

算法原理和实现全解析

7.1 YOLOv5 原理和实现全解析

7.1.1 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 12 epoch 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

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后续文档。

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，因此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

7.1.2 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如下数据增强、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，也是属于混合图片类增强，其是随机从另外一张图，然后两种图随机混合而成。其实现方法有多种，常见的做法是：要么 label 直接拼接起来，要么 label 也采用 alpha 混合，作者的做法非常简单，对 label 直接拼接即可，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

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，使得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 Mosiac 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 Mosiac 等混合类数据增强，会额外实现一个 `get_indexes` 方法用来获取其他图片索引，然后得到 4 张图片信息后就可以进行 Mosiac 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 数据集包装其传入一个包括 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)
```

(下页继续)

(续上页)

```
]

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),
    ...
]
```

此时就不再需要 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',
        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),
```

(下页继续)

(续上页)

```
        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，整体结构图如上所示。

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，可以直接将 MMDeploy 导出的 ONNX 文件格式使用 netron 打开。

1.2.1 Backbone

在 MMYOLO 中 CSPDarknet 继承自 BaseBackbone，整体结构和 ResNet 类似，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。
- 前 3 个 Stage Layer 由 1 个 ConvModule 和 1 个 CSPLayer 组成。如上图 Details 部分，其中 ConvModule 为 3x3 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。
- 第 4 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 模型结构会在 `Stage Layer` 2-4 之后分别输出，进入 `Neck` 结构，共抽取三个输出特征图，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)，stride 为 8/16/32。

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 模块输出特征图和 Backbone 完全一致即为 (B,256,80,80)、(B,512,40,40) 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，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。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，一个好的匹配策略明显可以提升算法性能。

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 过滤暂时丢掉，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，可以发现粗略估计正样本数相比前 YOLO 系列，至少增加了三倍

下面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，Anchor size 的获取方式与 YOLOv3 相同，是使用 kmeans 算法进行聚类获得。

在用户更换了数据集后，可以使用 MMDetection 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。

若你的 MMDetection 通过 mim 安装，可使用以下命令分析 Anchor：

```
mim run mmdet optimize_anchors ${CONFIG} --algorithm k-means
--input-shape ${INPUT_SHAPE [WIDTH HEIGHT]} --output-dir ${OUTPUT_DIR}
```

若 MMDetection 为其他方式安装，可进入 MMDetection 所在目录，使用以下命令分析 Anchor:

```
python tools/analysis_tools/optimize_anchors.py ${CONFIG} --algorithm 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：

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

公式中，

a_w□□$Anchor$□□□

c_x□□$Grid$□□□□□

σ□□$Sigmoid$□□□

而在 YOLOv5 中，回归公式为：

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

改进之处主要有以下几点：

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

$$(0, +\infty)$$

调整至

$$(0, 4a_{wh})$$

这个改进具有以下好处：

- **中心点能更好的预测到 0 和 1**。有助于更精准回归出 box 坐标。
- 宽高回归公式 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 + 图示：

prior1 匹配失败的原因是

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

(2) 将步骤 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)]，其中在 P3 特征图上，stride 为 8，通过计算 prior2，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

$$GT_x^{center_grid} = 26/8 = 3.25$$
$$GT_y^{center_grid} = 37/8 = 4.625$$

(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。

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

在 Objectness loss 中，P3,P4,P5 层的 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。

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

1.6.1 推理过程

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。其核心控制参数为：

1. **multi_label**

对于多类别预测来说是否考虑多标签，也就是同一个预测位置中预测的多个类别概率，是否当做单类处理。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.int)  # 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]
```

(下页继续)

(续上页)

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

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

7.1.3 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。

7.2 RTMDet 原理和实现全解析

7.2.1 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

7.2.2 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川练方式，但由于引入了旋转，切片导致 box 标注产生误差，需要在第二阶段引入额外的 L1oss 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).
```

(下页继续)

(续上页)

```
    """
    # 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]

    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,
```

(下页继续)

(续上页)

```
    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.
    """
    ...

    # 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()
```

(下页继续)

(续上页)

```
    # 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):
    ...

    # 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']:
            ...
```

(下页继续)

(续上页)

```
        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

@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

7.2.3 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。

CHAPTER 8

数据流

8.1 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(
```

(下页继续)

(续上页)

```
    # 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)
```

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',
        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:
```

(下页继续)

(续上页)

```
    """Pass the dataset to the pipeline during training to support mixed
    data augmentation, such as Mosaic and MixUp."""
    if self.test_mode is False:
        data_info = self.get_data_info(idx)
        data_info['dataset'] = self
        return self.pipeline(data_info)
    else:
        return super().prepare_data(idx)
```

CHAPTER 9

How to

本教程收集了任何如何使用 MMYOLO 进行 xxx 的答案。如果您遇到有关如何做的问题及答案，请随时更新此文档!

9.1 给骨干网络增加插件

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='mmdet.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。

9.2 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，你需要保证的是上一个 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'

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],
            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',
            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)
    ]
```

CHAPTER 10

解读文章和资源汇总

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。

10.1 MMYOLO 解读文章和资源

- MMYOLO 社区倾情贡献，RTMDet 原理社区开发者解读来啦！

10.2 MMEngine 解读文章和资源

10.3 MMCV 解读文章和资源

- 手把手教你如何高效地在 MMCV 中贡献算子

10.4 MMDetection 解读文章和资源

10.5 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?

10.6 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

10.7 其他

- Type Hints 入门教程，让代码更加规范整洁

CHAPTER 11

mmyolo.datasets

11.1 datasets

11.2 transforms

CHAPTER 12

mmyolo.engine

12.1 hooks

12.2 optimizers

CHAPTER 13

mmyolo.models

CHAPTER 14

mmyolo.utils

CHAPTER 15

模型库

15.1 共同设置

- 所有模型都是在 `coco_2017_train` 上训练，在 `coco_2017_val` 上测试。
- 我们使用分布式训练。

15.2 Baselines

15.2.1 YOLOv5

请参考 YOLOv5。

15.2.2 YOLOv6

请参考 YOLOv6。

15.2.3 YOLOX

请参考 YOLOX。

15.2.4 RTMDet

请参考 RTMDet。

CHAPTER 16

常见问题解答

CHAPTER 17

更新日志

17.1 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

17.1.1 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)

17.1.2 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)

17.1.3 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

17.2 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。

17.2.1 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 18

English

CHAPTER 19

简体中文

CHAPTER 20

Indices and tables

- genindex
- search