MMDetection

发布 *3.0.0*

MMDetection Authors

2023 年 04 月 06 日

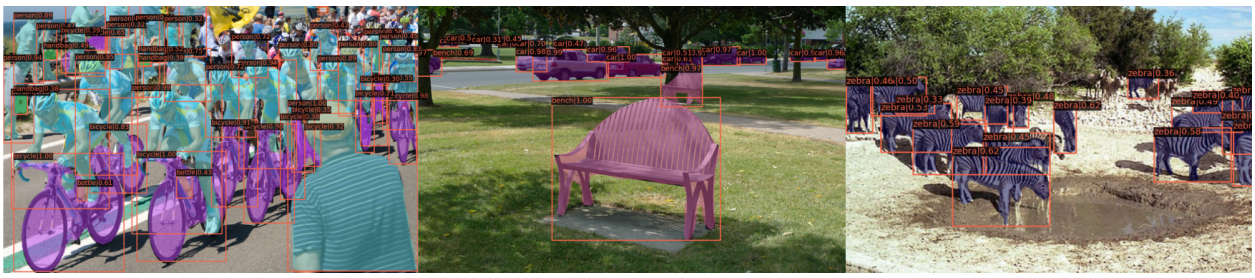
开始你的第一步

CHAPTER 1

概述

本章向您介绍 MMDetection 的整体框架，并提供详细的教程链接。

1.1 什么是 MMDetection



MMDetection 是一个目标检测工具箱，包含了丰富的目标检测、实例分割、全景分割算法以及相关的组件和模块，下面是它的整体框架：

MMDetection 由 7 个主要部分组成，apis、structures、datasets、models、engine、evaluation 和 visualization。

- **apis** 为模型推理提供高级 API。
- **structures** 提供 bbox、mask 和 DetDataSample 等数据结构。
- **datasets** 支持用于目标检测、实例分割和全景分割的各种数据集。
  - **transforms** 包含各种数据增强变换。
  - **samplers** 定义了不同的数据加载器采样策略。

- **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。
  - **roi_heads** 包含从 RoI 预测的各种检测头。
  - **seg_heads** 包含各种分割头。
  - **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。
  - **runner** 为 MMEngine 的执行器提供扩展。
  - **schedulers** 提供用于调整优化超参数的调度程序。
  - **optimizers** 提供优化器和优化器封装。
  - **hooks** 提供执行器的各种钩子。
- **evaluation** 为评估模型性能提供不同的指标。
- **visualization** 用于可视化检测结果。

1.2 如何使用本指南

以下是 MMDetection 的详细指南：

1. 安装说明见开始你的第一步。
2. MMDetection 的基本使用方法请参考以下教程。
   - 训练和测试
   - 实用工具
3. 参考以下教程深入了解：
   - 基础概念
   - 组件定制
4. 对于 MMDetection 2.x 版本的用户，我们提供了迁移指南，帮助您完成新版本的适配。

CHAPTER 2

开始你的第一步

2.1 依赖

本节中，我们将演示如何用 PyTorch 准备一个环境。

MMDetection 支持在 Linux，Windows 和 macOS 上运行。它需要 Python 3.7 以上，CUDA 9.2 以上和 PyTorch 1.6 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 --name openmmlab python=3.8 -y
conda activate openmmlab
```

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

我们推荐用户参照我们的最佳实践安装 MMDetection。不过，整个过程也是可定制化的，更多信息请参考自定义安装章节。

2.2.1 最佳实践

步骤 0. 使用 MIM 安装 MMEngine 和 MMCV。

```
pip install -U openmim
mim install mmengine
mim install "mmcv>=2.0.0"
```

注意： 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。

步骤 1. 安装 MMDetection。

方案 a：如果你开发并直接运行 mmdet，从源码安装它：

```
git clone https://github.com/open-mmlab/mmdetection.git
cd mmdetection
pip 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 b：如果你将 mmdet 作为依赖或第三方 Python 包，使用 MIM 安装：

```
mim install mmdet
```

2.3 验证安装

为了验证 MMDetection 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 mmdet --config rtmdet_tiny_8xb32-300e_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 `rtmdet_tiny_8xb32-300e_coco.py` 和 `rtmdet_tiny_8xb32-300e_coco_20220902_112414-78e30dcc.pth`。

步骤 2. 推理验证。

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

```
python demo/image_demo.py demo/demo.jpg rtmdet_tiny_8xb32-300e_coco.py --weights␣
↪rtmdet_tiny_8xb32-300e_coco_20220902_112414-78e30dcc.pth --device cpu
```

你会在当前文件夹中的 `outputs/vis` 文件夹中看到一个新的图像 `demo.jpg`，图像中包含有网络预测的检测框。

方案 b：如果你通过 MIM 安装的 MMDetection，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 = 'rtmdet_tiny_8xb32-300e_coco.py'
checkpoint_file = 'rtmdet_tiny_8xb32-300e_coco_20220902_112414-78e30dcc.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，包含有检测框，类别和得分。

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，安装 CUDA 运行时库就足够了，这是因为不需要在本地编译 CUDA 代码。但如果你希望从源码编译 MMCV，或是开发其他 CUDA 算子，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
```

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>=2.0.0" -f https://download.openmmlab.com/mmcv/dist/cu116/torch1.12.
↪0/index.html
```

在 CPU 环境中安装

MMDetection 可以在 CPU 环境中构建。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。

但是，以下功能在该模式下不能使用：

- Deformable Convolution
- Modulated Deformable Convolution
- ROI pooling
- Deformable ROI pooling
- CARAFE
- SyncBatchNorm
- CrissCrossAttention
- MaskedConv2d
- Temporal Interlace Shift
- nms_cuda
- sigmoid_focal_loss_cuda
- bbox_overlaps

因此，如果尝试训练/测试/推理包含上述算子的模型，将会报错。下表列出了将会受影响的相关算法。

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 和 MMDetection 即可，命令如下：

步骤 1. 使用 MIM 安装 MMEngine 和 MMCV。

```
!pip3 install openmim
!mim install mmengine
!mim install "mmcv>=2.0.0,<2.1.0"
```

步骤 2. 使用源码安装 MMDetection。

```
!git clone https://github.com/open-mmlab/mmdetection.git
%cd mmdetection
!pip install -e .
```

步骤 3. 验证安装是否成功。

```
import mmdet
print(mmdet.__version__)
# 预期输出：3.0.0 或其他版本号
```

注解： 在 Jupyter Notebook 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 MMDetection

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。

```
# 基于 PyTorch 1.9, CUDA 11.1 构建镜像
# 如果你想要其他版本，只需要修改 Dockerfile
docker build -t mmdetection docker/
```

用以下命令运行 Docker 镜像：

```
docker run --gpus all --shm-size=8g -it -v {DATA_DIR}:/mmdetection/data mmdetection
```

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 上提出一个问题。

2.3.3 使用多个 MMDetection 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 MMDetection。

要使环境中安装默认版本的 MMDetection 而不是当前正在使用的，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

MMDetection 在 Model Zoo 中提供了数百个预训练的检测模型，并支持多种标准数据集格式，包括 Pascal VOC、COCO、CityScapes、LVIS 等。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 学习配置文件

MMDetection 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

MMDetection 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 Mask R-CNN 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：

模型配置

在 mmdetection 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。

```
model = dict(
    type='MaskRCNN',  # 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='DetDataPreprocessor',  # 数据预处理器的类型，参考 https://mmdetection.
↪readthedocs.io/en/latest/api.html#mmdet.models.data_preprocessors.
↪DetDataPreprocessor
        mean=[123.675, 116.28, 103.53],  # 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=[58.395, 57.12, 57.375],  # 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,  # 是否将图片通道从 BGR 转为 RGB
        pad_mask=True,  # 是否填充实例分割掩码
        pad_size_divisor=32),  # padding 后的图像的大小应该可以被 ``pad_size_divisor`` 整除
    backbone=dict(  # 主干网络的配置文件
        type='ResNet',  # 主干网络的类别，可用选项请参考 https://mmdetection.readthedocs.io/
↪en/latest/api.html#mmdet.models.backbones.ResNet
        depth=50,  # 主干网络的深度，对于 ResNet 和 ResNext 通常设置为 50 或 101
        num_stages=4,  # 主干网络状态 (stages) 的数目，这些状态产生的特征图作为后续的 head 的输
入
        out_indices=(0, 1, 2, 3),  # 每个状态产生的特征图输出的索引
        frozen_stages=1,  # 第一个状态的权重被冻结
        norm_cfg=dict(  # 归一化层 (norm layer) 的配置项
            type='BN',  # 归一化层的类别，通常是 BN 或 GN
            requires_grad=True),  # 是否训练归一化里的 gamma 和 beta
        norm_eval=True,  # 是否冻结 BN 里的统计项
        style='pytorch',  # 主干网络的风格，'pytorch' 意思是步长为 2 的层为 3x3 卷积， 'caffe
↪' 意思是步长为 2 的层为 1x1 卷积
        init_cfg=dict(type='Pretrained', checkpoint='torchvision://resnet50')),  # 加载
通过 ImageNet 预训练的模型
    neck=dict(
        type='FPN',  # 检测器的 neck 是 FPN，我们同样支持 'NASFPN', 'PAFPN' 等，更多细节可以参
考 https://mmdetection.readthedocs.io/en/latest/api.html#mmdet.models.necks.FPN
        in_channels=[256, 512, 1024, 2048],  # 输入通道数，这与主干网络的输出通道一致
        out_channels=256,  # 金字塔特征图每一层的输出通道
        num_outs=5),  # 输出的范围 (scales)
    rpn_head=dict(
        type='RPNHead',  # rpn_head 的类型是 'RPNHead', 我们也支持 'GARPNHead' 等，更多细节
可以参考 https://mmdetection.readthedocs.io/en/latest/api.html#mmdet.models.dense_heads.
↪RPNHead
        in_channels=256,  # 每个输入特征图的输入通道，这与 neck 的输出通道一致
        feat_channels=256,  # head 卷积层的特征通道
        anchor_generator=dict(  # 锚点 (Anchor) 生成器的配置
            type='AnchorGenerator',  # 大多数方法使用 AnchorGenerator 作为锚点生成器, SSD␣
↪检测器使用 `SSDAnchorGenerator`。更多细节请参考 https://github.com/open-mmlab/
↪mmdetection/blob/main/mmdet/models/task_modules/prior_generators/anchor_generator.py
↪#L18
```

(下页继续)

(续上页)

```
            scales=[8],  # 锚点的基本比例，特征图某一位置的锚点面积为 scale * base_sizes
            ratios=[0.5, 1.0, 2.0],  # 高度和宽度之间的比率
            strides=[4, 8, 16, 32, 64]),  # 锚生成器的步幅。这与 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
        bbox_coder=dict(  # 在训练和测试期间对框进行编码和解码
            type='DeltaXYWHBBoxCoder',  # 框编码器的类别，'DeltaXYWHBBoxCoder' 是最常用的，
更多细节请参考 https://github.com/open-mmlab/mmdetection/blob/main/mmdet/models/task_
↪modules/coders/delta_xywh_bbox_coder.py#L13
            target_means=[0.0, 0.0, 0.0, 0.0],  # 用于编码和解码框的目标均值
            target_stds=[1.0, 1.0, 1.0, 1.0]),  # 用于编码和解码框的标准差
        loss_cls=dict(  # 分类分支的损失函数配置
            type='CrossEntropyLoss',  # 分类分支的损失类型，我们也支持 FocalLoss 等，更多细节
请参考 https://github.com/open-mmlab/mmdetection/blob/main/mmdet/models/losses/cross_
↪entropy_loss.py#L201
            use_sigmoid=True,  # RPN 通常进行二分类，所以通常使用 sigmoid 函数
            los_weight=1.0),  # 分类分支的损失权重
        loss_bbox=dict(  # 回归分支的损失函数配置
            type='L1Loss',  # 损失类型，我们还支持许多 IoU Losses 和 Smooth L1-loss 等，更多
细节请参考 https://github.com/open-mmlab/mmdetection/blob/main/mmdet/models/losses/
↪smooth_l1_loss.py#L56
            loss_weight=1.0)),  # 回归分支的损失权重
    roi_head=dict(  # RoIHead 封装了两步 (two-stage)/级联 (cascade) 检测器的第二步
        type='StandardRoIHead',  # RoI head 的类型，更多细节请参考 https://github.com/
↪open-mmlab/mmdetection/blob/main/mmdet/models/roi_heads/standard_roi_head.py#L17
        bbox_roi_extractor=dict(  # 用于 bbox 回归的 RoI 特征提取器
            type='SingleRoIExtractor',  # RoI 特征提取器的类型，大多数方法使用␣
↪SingleRoIExtractor，更多细节请参考 https://github.com/open-mmlab/mmdetection/blob/main/
↪mmdet/models/roi_heads/roi_extractors/single_level_roi_extractor.py#L13
            roi_layer=dict(  # RoI 层的配置
                type='RoIAlign',  # RoI 层的类别，也支持 DeformRoIPoolingPack 和␣
↪ModulatedDeformRoIPoolingPack，更多细节请参考 https://mmcv.readthedocs.io/en/latest/
↪api.html#mmcv.ops.RoIAlign
                output_size=7,  # 特征图的输出大小
                sampling_ratio=0),  # 提取 RoI 特征时的采样率。0 表示自适应比率
            out_channels=256,  # 提取特征的输出通道
            featmap_strides=[4, 8, 16, 32]),  # 多尺度特征图的步幅，应该与主干的架构保持一致
        bbox_head=dict(  # RoIHead 中 box head 的配置
            type='Shared2FCBBoxHead',  # bbox head 的类别，更多细节请参考 https://github.
↪com/open-mmlab/mmdetection/blob/main/mmdet/models/roi_heads/bbox_heads/convfc_bbox_
↪head.py#L220
            in_channels=256,  # bbox head 的输入通道。 这与 roi_extractor 中的 out_
↪channels 一致
            fc_out_channels=1024,  # FC 层的输出特征通道
```

(下页继续)

(续上页)

```
            roi_feat_size=7,  # 候选区域 (Region of Interest) 特征的大小
            num_classes=80,  # 分类的类别数量
            bbox_coder=dict(  # 第二阶段使用的框编码器
                type='DeltaXYWHBBoxCoder',  # 框编码器的类别，大多数情况使用
↪'DeltaXYWHBBoxCoder'
                target_means=[0.0, 0.0, 0.0, 0.0],  # 用于编码和解码框的均值
                target_stds=[0.1, 0.1, 0.2, 0.2]),  # 编码和解码的标准差。因为框更准确，所以
值更小，常规设置时 [0.1, 0.1, 0.2, 0.2]。
            reg_class_agnostic=False,  # 回归是否与类别无关
            loss_cls=dict(  # 分类分支的损失函数配
                type='CrossEntropyLoss',  # 分类分支的损失类型，我们也支持 FocalLoss 等
                use_sigmoid=False,  # 是否使用 sigmoid
                loss_weight=1.0),  # 分类分支的损失权重
            loss_bbox=dict(  # 回归分支的损失函数配置
                type='L1Loss',  # 损失类型，我们还支持许多 IoU Losses 和 Smooth L1-loss 等
                loss_weight=1.0)),  # 回归分支的损失权重
        mask_roi_extractor=dict(  # 用于 mask 生成的 RoI 特征提取器
            type='SingleRoIExtractor',  # RoI 特征提取器的类型，大多数方法使用␣
↪SingleRoIExtractor
            roi_layer=dict(  # 提取实例分割特征的 RoI 层配置
                type='RoIAlign',  # RoI 层的类型，也支持 DeformRoIPoolingPack 和␣
↪ModulatedDeformRoIPoolingPack
                output_size=14,  # 特征图的输出大小
                sampling_ratio=0),  # 提取 RoI 特征时的采样率
            out_channels=256,  # 提取特征的输出通道
            featmap_strides=[4, 8, 16, 32]),  # 多尺度特征图的步幅
        mask_head=dict(  # mask 预测 head 模型
            type='FCNMaskHead',  # mask head 的类型，更多细节请参考 https://mmdetection.
↪readthedocs.io/en/latest/api.html#mmdet.models.roi_heads.FCNMaskHead
            num_convs=4,  # mask head 中的卷积层数
            in_channels=256,  # 输入通道，应与 mask roi extractor 的输出通道一致
            conv_out_channels=256,  # 卷积层的输出通道
            num_classes=80,  # 要分割的类别数
            loss_mask=dict(  # mask 分支的损失函数配置
                type='CrossEntropyLoss',  # 用于分割的损失类型
                use_mask=True,  # 是否只在正确的类中训练 mask
                loss_weight=1.0))),  # mask 分支的损失权重
    train_cfg = dict(  # rpn 和 rcnn 训练超参数的配置
        rpn=dict(  # rpn 的训练配置
            assigner=dict(  # 分配器 (assigner) 的配置
                type='MaxIoUAssigner',  # 分配器的类型，MaxIoUAssigner 用于许多常见的检测器，
更多细节请参考 https://github.com/open-mmlab/mmdetection/blob/main/mmdet/models/task_
↪modules/assigners/max_iou_assigner.py#L14
```

(下页继续)

(续上页)

```
                pos_iou_thr=0.7,  # IoU >= 0.7(阈值) 被视为正样本
                neg_iou_thr=0.3,  # IoU < 0.3(阈值) 被视为负样本
                min_pos_iou=0.3,  # 将框作为正样本的最小 IoU 阈值
                match_low_quality=True,  # 是否匹配低质量的框 (更多细节见 API 文档)
                ignore_iof_thr=-1),  # 忽略 bbox 的 IoF 阈值
            sampler=dict(  # 正/负采样器 (sampler) 的配置
                type='RandomSampler',  # 采样器类型，还支持 PseudoSampler 和其他采样器，更多
细节请参考 https://github.com/open-mmlab/mmdetection/blob/main/mmdet/models/task_
↪modules/samplers/random_sampler.py#L14
                num=256,  # 样本数量。
                pos_fraction=0.5,  # 正样本占总样本的比例
                neg_pos_ub=-1,  # 基于正样本数量的负样本上限
                add_gt_as_proposals=False),  # 采样后是否添加 GT 作为 proposal
            allowed_border=-1,  # 填充有效锚点后允许的边框
            pos_weight=-1,  # 训练期间正样本的权重
            debug=False),  # 是否设置调试 (debug) 模式
        rpn_proposal=dict(  # 在训练期间生成 proposals 的配置
            nms_across_levels=False,  # 是否对跨层的 box 做 NMS。仅适用于 `GARPNHead` ，
naive rpn 不支持 nms cross levels
            nms_pre=2000,  # NMS 前的 box 数
            nms_post=1000,  # NMS 要保留的 box 的数量，只在 GARPNHHead 中起作用
            max_per_img=1000,  # NMS 后要保留的 box 数量
            nms=dict( # NMS 的配置
                type='nms',  # NMS 的类别
                iou_threshold=0.7 # NMS 的阈值
                ),
            min_bbox_size=0),  # 允许的最小 box 尺寸
        rcnn=dict(  # roi head 的配置。
            assigner=dict(  # 第二阶段分配器的配置，这与 rpn 中的不同
                type='MaxIoUAssigner',  # 分配器的类型，MaxIoUAssigner 目前用于所有 roi_
↪heads。更多细节请参考 https://github.com/open-mmlab/mmdetection/blob/main/mmdet/models/
↪task_modules/assigners/max_iou_assigner.py#L14
                pos_iou_thr=0.5,  # IoU >= 0.5(阈值) 被认为是正样本
                neg_iou_thr=0.5,  # IoU < 0.5(阈值) 被认为是负样本
                min_pos_iou=0.5,  # 将 box 作为正样本的最小 IoU 阈值
                match_low_quality=False,  # 是否匹配低质量下的 box(有关更多详细信息，请参阅␣
↪API 文档)
                ignore_iof_thr=-1),  # 忽略 bbox 的 IoF 阈值
            sampler=dict(
                type='RandomSampler',  # 采样器的类型，还支持 PseudoSampler 和其他采样器，更
多细节请参考 https://github.com/open-mmlab/mmdetection/blob/main/mmdet/models/task_
↪modules/samplers/random_sampler.py#L14
                num=512,  # 样本数量
```

(下页继续)

(续上页)

```
                pos_fraction=0.25,  # 正样本占总样本的比例
                neg_pos_ub=-1,  # 基于正样本数量的负样本上限
                add_gt_as_proposals=True
            ),  # 采样后是否添加 GT 作为 proposal
            mask_size=28,  # mask 的大小
            pos_weight=-1,  # 训练期间正样本的权重
            debug=False)),  # 是否设置调试模式
    test_cfg = dict(  # 用于测试 rpn 和 rcnn 超参数的配置
        rpn=dict(  # 测试阶段生成 proposals 的配置
            nms_across_levels=False,  # 是否对跨层的 box 做 NMS。仅适用于 `GARPNHead`,
naive rpn 不支持做 NMS cross levels
            nms_pre=1000,  # NMS 前的 box 数
            nms_post=1000,  # NMS 要保留的 box 的数量，只在 `GARPNHead` 中起作用
            max_per_img=1000,  # NMS 后要保留的 box 数量
            nms=dict( # NMS 的配置
                type='nms',  # NMS 的类型
                iou_threshold=0.7 # NMS 阈值
                ),
            min_bbox_size=0),  # box 允许的最小尺寸
        rcnn=dict(  # roi heads 的配置
            score_thr=0.05,  # bbox 的分数阈值
            nms=dict(  # 第二步的 NMS 配置
                type='nms',  # NMS 的类型
                iou_thr=0.5),  # NMS 的阈值
            max_per_img=100,  # 每张图像的最大检测次数
            mask_thr_binary=0.5)))  # mask 预处的阈值
```

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。

```
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。

train_pipeline = [  # 训练数据处理流程
    dict(type='LoadImageFromFile'),  # 第 1 个流程，从文件路径里加载图像。
    dict(
        type='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,  # 是否使用 instance mask，实例分割需要设置为 True。
```

(下页继续)

(续上页)

```
        poly2mask=False),  # 是否将 polygon mask 转化为 instance mask, 设置为 False 以加速和节省内存。
    dict(
        type='Resize',  # 变化图像和其标注大小的流程。
        scale=(1333, 800),  # 图像的最大尺寸
        keep_ratio=True  # 是否保持图像的长宽比。
        ),
    dict(
        type='RandomFlip',  # 翻转图像和其标注的数据增广流程。
        prob=0.5),  # 翻转图像的概率。
    dict(type='PackDetInputs')  # 将数据转换为检测器输入格式的流程
]
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='Resize', scale=(1333, 800), keep_ratio=True),  # 变化图像大小的流程。
    dict(
        type='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'))
]
train_dataloader = dict(  # 训练 dataloader 配置
    batch_size=2,  # 单个 GPU 的 batch size
    num_workers=2,  # 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
    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://
↪mmengine.readthedocs.io/zh_CN/latest/api/generated/mmengine.dataset.DefaultSampler.
↪html#mmengine.dataset.DefaultSampler
        shuffle=True),  # 随机打乱每个轮次训练数据的顺序
    batch_sampler=dict(type='AspectRatioBatchSampler'),  # 批数据采样器, 用于确保每一批次内的数据拥有相似的长宽比, 可用于节省显存
    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/'),  # 图片路径前缀
        filter_cfg=dict(filter_empty_gt=True, min_size=32),  # 图片和标注的过滤配置
        pipeline=train_pipeline))  # 这是由之前创建的 train_pipeline 定义的数据处理流程。
val_dataloader = dict(  # 验证 dataloader 配置
    batch_size=1,  # 单个 GPU 的 Batch size。如果 batch-szie > 1, 组成 batch 时的额外填充会影响模型推理精度
    num_workers=2,  # 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
    drop_last=False,  # 是否丢弃最后未能组成一个批次的数据
    sampler=dict(
        type='DefaultSampler',
        shuffle=False),  # 验证和测试时不打乱数据顺序
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/instances_val2017.json',
        data_prefix=dict(img='val2017/'),
        test_mode=True,  # 开启测试模式，避免数据集过滤图片和标注
        pipeline=test_pipeline))
test_dataloader = val_dataloader  # 测试 dataloader 配置
```

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='CocoMetric',  # 用于评估检测和实例分割的 AR、AP 和 mAP 的 coco 评价指标
    ann_file=data_root + 'annotations/instances_val2017.json',  # 标注文件路径
    metric=['bbox', 'segm'],  # 需要计算的评价指标，`bbox` 用于检测，`segm` 用于实例分割
    format_only=False)
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 MMDetection 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(
```

(下页继续)

(续上页)

```
    type='CocoMetric',
    ann_file=data_root + 'annotations/image_info_test-dev2017.json',
    metric=['bbox', 'segm'],
    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。

```
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=12,  # 最大训练轮次
    val_interval=1)  # 验证间隔。每个 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
```

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=0.02,  # 基础学习率
        momentum=0.9,  # 带动量的随机梯度下降
        weight_decay=0.0001),  # 权重衰减
    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
    )
```

`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。

```
param_scheduler = [
    dict(
```

(下页继续)

(续上页)

```
        type='LinearLR',  # 使用线性学习率预热
        start_factor=0.001, # 学习率预热的系数
        by_epoch=False,  # 按 iteration 更新预热学习率
        begin=0,  # 从第一个 iteration 开始
        end=500),  # 到第 500 个 iteration 结束
    dict(
        type='MultiStepLR',  # 在训练过程中使用 multi step 学习率策略
        by_epoch=True,  # 按 epoch 更新学习率
        begin=0,  # 从第一个 epoch 开始
        end=12,  # 到第 12 个 epoch 结束
        milestones=[8, 11],  # 在哪几个 epoch 进行学习率衰减
        gamma=0.1)  # 学习率衰减系数
]
```

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。更多内容请看 钩子教程 。

```
default_hooks = dict(
    timer=dict(type='IterTimerHook'),
    logger=dict(type='LoggerHook', interval=50),
    param_scheduler=dict(type='ParamSchedulerHook'),
    checkpoint=dict(type='CheckpointHook', interval=1),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    visualization=dict(type='DetVisualizationHook'))
```

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。

```
custom_hooks = []
```

运行相关配置

```
default_scope = 'mmdet'  # 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/advanced_tutorials/registry.html

env_cfg = dict(
    cudnn_benchmark=False,  # 是否启用 cudnn benchmark
    mp_cfg=dict(  # 多进程设置
```

(下页继续)

(续上页)

```
        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='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 Iter-based 配置

MMEngine 的 Runner 除了基于轮次的训练循环（epoch）外，还提供了基于迭代（iteration）的训练循环。要使用基于迭代的训练，用户应该修改 `train_cfg`、`param_scheduler`、`train_dataloader`、`default_hooks` 和 `log_processor`。以下是将基于 epoch 的 RetinaNet 配置更改为基于 iteration 的示例：configs/retinanet/retinanet_r50_fpn_90k_coco.py

```
# iter-based 训练配置
train_cfg = dict(
    _delete_=True,  # 忽略继承的配置文件中的值（可选）
    type='IterBasedTrainLoop',  # iter-based 训练循环
    max_iters=90000,  # 最大迭代次数
    val_interval=10000)  # 每隔多少次进行一次验证


# 将参数调度器修改为 iter-based
param_scheduler = [
    dict(
        type='LinearLR', start_factor=0.001, by_epoch=False, begin=0, end=500),
    dict(
        type='MultiStepLR',
        begin=0,
```

(下页继续)

(续上页)

```
        end=90000,
        by_epoch=False,
        milestones=[60000, 80000],
        gamma=0.1)
]

# 切换至 InfiniteSampler 来避免 dataloader 重启
train_dataloader = dict(sampler=dict(type='InfiniteSampler'))

# 将模型检查点保存间隔设置为按 iter 保存
default_hooks = dict(checkpoint=dict(by_epoch=False, interval=10000))

# 将日志格式修改为 iter-based
log_processor = dict(by_epoch=False)
```

3.1.3 配置文件继承

在 config/_base_ 文件夹下有 4 个基本组件类型，分别是：数据集 (dataset)，模型 (model)，训练策略 (schedule) 和运行时的默认设置 (default runtime)。许多方法，例如 Faster R-CNN、Mask R-CNN、Cascade R-CNN、RPN、SSD 能够很容易地构建出来。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 Faster R-CNN 的基础上做了一些修改，用户首先可以通过指定 _base_ = ../faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py 来继承基础的 Faster R-CNN 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 xxx_rcnn 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_ = './mask-rcnn_r50_fpn_1x_coco.py'
```

当 _base_ 是多个文件路径的列表时，表示继承多个文件。

```
_base_ = [
    '../_base_/models/mask-rcnn_r50_fpn.py',
    '../_base_/datasets/coco_instance.py',
```

(下页继续)

(续上页)

```
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]
```

如果需要检查配置文件，可以通过运行 `python tools/misc/print_config.py /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 MMDetection 里，例如为了改变 Mask R-CNN 的主干网络的某些内容：

```
model = dict(
    type='MaskRCNN',
    backbone=dict(
        type='ResNet',
        depth=50,
        num_stages=4,
        out_indices=(0, 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(...),
    rpn_head=dict(...),
    roi_head=dict(...))
```

基础配置的 `Mask R-CNN` 使用 `ResNet-50`，在需要将主干网络改成 `HRNet` 的时候，因为 `HRNet` 和 `ResNet` 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_ = '../mask_rcnn/mask-rcnn_r50_fpn_1x_coco.py'
model = dict(
    backbone=dict(
        _delete_=True,
        type='HRNet',
        extra=dict(
            stage1=dict(
                num_modules=1,
                num_branches=1,
                block='BOTTLENECK',
                num_blocks=(4, ),
```

(下页继续)

(续上页)

```
                num_channels=(64, )),
            stage2=dict(
                num_modules=1,
                num_branches=2,
                block='BASIC',
                num_blocks=(4, 4),
                num_channels=(32, 64)),
            stage3=dict(
                num_modules=4,
                num_branches=3,
                block='BASIC',
                num_blocks=(4, 4, 4),
                num_channels=(32, 64, 128)),
            stage4=dict(
                num_modules=3,
                num_branches=4,
                block='BASIC',
                num_blocks=(4, 4, 4, 4),
                num_channels=(32, 64, 128, 256))),
        init_cfg=dict(type='Pretrained', checkpoint='open-mmlab://msra/hrnetv2_w32')),
    neck=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 Mask R-CNN 的多尺度策略 (multi scale strategy)，`train_pipeline`/`test_pipeline` 是我们想要修改的中间变量。

```
_base_ = './mask-rcnn_r50_fpn_1x_coco.py'

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True, with_mask=True),
    dict(
        type='RandomResize', scale=[(1333, 640), (1333, 800)],
        keep_ratio=True),
    dict(type='RandomFlip', prob=0.5),
    dict(type='PackDetInputs')
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
```

(下页继续)

(续上页)

```
    dict(type='Resize', scale=(1333, 800), keep_ratio=True),
    dict(
        type='PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor'))
]
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_ = './mask-rcnn_r50_fpn_1x_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。例如：

```
_base_ = './mask-rcnn_r50_fpn_1x_coco.py'

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

3.1.4 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。

3.1.5 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]_[...]}_{training␣
↪settings}_{training dataset information}_{testing dataset information}.py
```

文件名分为五个部分。每个部分用 _ 连接，每个部分内的单词应该用-连接。

- {algorithm name}: 算法的名称。它可以是检测器名称，例如 faster-rcnn、mask-rcnn 等。也可以是半监督或知识蒸馏算法，例如 soft-teacher、lad 等等
- {component names}: 算法中使用的组件名称，如 backbone、neck 等。例如 r50-caffe_fpn_gn-head 表示在算法中使用 caffe 版本的 ResNet50、FPN 和使用了 Group Norm 的检测头。
- {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：4xb4-mixup-giou-coslr-100e 表示使用 8 个 gpu 每个 gpu 4 张图、mixup 数据增强、GIoU loss、余弦退火学习率，并训练 100 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，选项是 1x、2x、20e 等。1x 和 2x 分别代表 12 epoch 和 24 epoch，20e 在级联模型中使用，表示 20 epoch。对于 1x/2x，初始学习率在第 8/16 和第 11/22 epoch 衰减 10 倍；对于 20e ，初始学习率在第 16 和第 19 epoch 衰减 10 倍。
- {training dataset information}: 训练数据集，例如 coco, coco-panoptic, cityscapes, voc-0712, wider-face。
- {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。

3.2 使用已有模型在标准数据集上进行推理

MMDetection 提供了许多预训练好的检测模型，可以在 Model Zoo 查看具体有哪些模型。

推理具体指使用训练好的模型来检测图像上的目标，本文将会展示具体步骤。

在 MMDetection 中，一个模型被定义为一个配置文件 和对应被存储在 checkpoint 文件内的模型参数的集合。

首先，我们建议从 RTMDet 开始，其 配置 文件和 checkpoint 文件在此。我们建议将 checkpoint 文件下载到 `checkpoints` 文件夹内。

3.2.1 推理的高层编程接口

MMDetection 为在图片上推理提供了 Python 的高层编程接口。下面是建立模型和在图像或视频上进行推理的例子。

```
import cv2
import mmcv
from mmcv.transforms import Compose
from mmengine.utils import track_iter_progress
from mmdet.registry import VISUALIZERS
from mmdet.apis import init_detector, inference_detector


# 指定模型的配置文件和 checkpoint 文件路径
config_file = 'configs/rtmdet/rtmdet_l_8xb32-300e_coco.py'
checkpoint_file = 'checkpoints/rtmdet_l_8xb32-300e_coco_20220719_112030-5a0be7c4.pth'

# 根据配置文件和 checkpoint 文件构建模型
model = init_detector(config_file, checkpoint_file, device='cuda:0')

# 初始化可视化工具
visualizer = VISUALIZERS.build(model.cfg.visualizer)
# 从 checkpoint 中加载 Dataset_meta，并将其传递给模型的 init_detector
visualizer.dataset_meta = model.dataset_meta

# 测试单张图片并展示结果
img = 'test.jpg'  # 或者 img = mmcv.imread(img)，这样图片仅会被读一次
result = inference_detector(model, img)

# 显示结果
img = mmcv.imread(img)
img = mmcv.imconvert(img, 'bgr', 'rgb')
```

(下页继续)

(续上页)

```
visualizer.add_datasample(
    'result',
    img,
    data_sample=result,
    draw_gt=False,
    show=True)

# 测试视频并展示结果
# 构建测试 pipeline
model.cfg.test_dataloader.dataset.pipeline[0].type = 'LoadImageFromNDArray'
test_pipeline = Compose(model.cfg.test_dataloader.dataset.pipeline)

# 可视化工具在第 33 行和 35 行已经初完成了初始化，如果直接在一个 jupyter nodebook 中运行这个 demo，

# 这里则不需要再创建一个可视化工具了。
# 初始化可视化工具
visualizer = VISUALIZERS.build(model.cfg.visualizer)
# 从 checkpoint 中加载 Dataset_meta，并将其传递给模型的 init_detector
visualizer.dataset_meta = model.dataset_meta

# 显示间隔 (ms)，0 表示暂停
wait_time = 1

video = mmcv.VideoReader('video.mp4')

cv2.namedWindow('video', 0)

for frame in track_iter_progress(video_reader):
    result = inference_detector(model, frame, test_pipeline=test_pipeline)
    visualizer.add_datasample(
        name='video',
        image=frame,
        data_sample=result,
        draw_gt=False,
        show=False)
    frame = visualizer.get_image()
    mmcv.imshow(frame, 'video', wait_time)

cv2.destroyAllWindows()
```

Jupyter notebook 上的演示样例在 demo/inference_demo.ipynb 。

注意: `inference_detector` 目前仅支持单张图片的推理。

3.2.2 演示样例

我们还提供了三个演示脚本，它们是使用高层编程接口实现的。源码在此 。

图片样例

这是在单张图片上进行推理的脚本。

```
python demo/image_demo.py \
    ${IMAGE_FILE} \
    ${CONFIG_FILE} \
    [--weights ${WEIGHTS}] \
    [--device ${GPU_ID}] \
    [--pred-score-thr ${SCORE_THR}]
```

运行样例：

```
python demo/image_demo.py demo/demo.jpg \
    configs/rtmdet/rtmdet_l_8xb32-300e_coco.py \
    --weights checkpoints/rtmdet_l_8xb32-300e_coco_20220719_112030-5a0be7c4.pth \
    --device cpu
```

摄像头样例

这是使用摄像头实时图片的推理脚本。

```
python demo/webcam_demo.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--device ${GPU_ID}] \
    [--camera-id ${CAMERA-ID}] \
    [--score-thr ${SCORE_THR}]
```

运行样例：

```
python demo/webcam_demo.py \
    configs/rtmdet/rtmdet_l_8xb32-300e_coco.py \
    checkpoints/rtmdet_l_8xb32-300e_coco_20220719_112030-5a0be7c4.pth
```

视频样例

这是在视频样例上进行推理的脚本。

```
python demo/video_demo.py \
    ${VIDEO_FILE} \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--device ${GPU_ID}] \
    [--score-thr ${SCORE_THR}] \
    [--out ${OUT_FILE}] \
    [--show] \
    [--wait-time ${WAIT_TIME}]
```

运行样例：

```
python demo/video_demo.py demo/demo.mp4 \
    configs/rtmdet/rtmdet_l_8xb32-300e_coco.py \
    checkpoints/rtmdet_l_8xb32-300e_coco_20220719_112030-5a0be7c4.pth \
    --out result.mp4
```

视频样例，显卡加速版本

这是在视频样例上进行推理的脚本，使用显卡加速。

```
python demo/video_gpuaccel_demo.py \
     ${VIDEO_FILE} \
     ${CONFIG_FILE} \
     ${CHECKPOINT_FILE} \
     [--device ${GPU_ID}] \
     [--score-thr ${SCORE_THR}] \
     [--nvdecode] \
     [--out ${OUT_FILE}] \
     [--show] \
     [--wait-time ${WAIT_TIME}]
```

运行样例：

```
python demo/video_gpuaccel_demo.py demo/demo.mp4 \
    configs/rtmdet/rtmdet_l_8xb32-300e_coco.py \
    checkpoints/rtmdet_l_8xb32-300e_coco_20220719_112030-5a0be7c4.pth \
    --nvdecode --out result.mp4
```

3.3 数据集准备

MMDetection 支持多个公共数据集，包括 COCO，Pascal VOC，Cityscapes 和 其他更多数据集。

一些公共数据集，比如 Pascal VOC 及其镜像数据集，或者 COCO 等数据集都可以从官方网站或者镜像网站获取。注意：在检测任务中，Pascal VOC 2012 是 Pascal VOC 2007 的无交集扩展，我们通常将两者一起使用。我们建议将数据集下载，然后解压到项目外部的某个文件夹内，然后通过符号链接的方式，将数据集根目录链接到 `$MMDETECTION/data` 文件夹下，如果你的文件夹结构和下方不同的话，你需要在配置文件中改变对应的路径。

我们提供了下载 COCO 等数据集的脚本，你可以运行 `python tools/misc/download_dataset.py --dataset-name coco2017` 下载 COCO 数据集。对于中国境内的用户，我们也推荐通过开源数据平台 OpenDataLab 来下载数据，以获得更好的下载体验。

更多用法请参考数据集下载

```
mmdetection
├── mmdet
├── tools
├── configs
├── data
│   ├── coco
│   │   ├── annotations
│   │   ├── train2017
│   │   ├── val2017
│   │   ├── test2017
│   ├── cityscapes
│   │   ├── annotations
│   │   ├── leftImg8bit
│   │   │   ├── train
│   │   │   ├── val
│   │   ├── gtFine
│   │   │   ├── train
│   │   │   ├── val
│   ├── VOCdevkit
│   │   ├── VOC2007
│   │   ├── VOC2012
```

有些模型需要额外的 COCO-stuff 数据集，比如 HTC，DetectoRS 和 SCNet，你可以下载并解压它们到 `coco` 文件夹下。文件夹会是如下结构：

```
mmdetection
├── data
│   ├── coco
│   │   ├── annotations
```

(下页继续)

(续上页)

```
|   |   ├── train2017
|   |   ├── val2017
|   |   ├── test2017
|   |   ├── stuffthingmaps
```

PanopticFPN 等全景分割模型需要额外的 COCO Panoptic 数据集，你可以下载并解压它们到 coco/annotations 文件夹下。文件夹会是如下结构：

```
mmdetection
├── data
|   ├── coco
|   |   ├── annotations
|   |   |   ├── panoptic_train2017.json
|   |   |   ├── panoptic_train2017
|   |   |   ├── panoptic_val2017.json
|   |   |   ├── panoptic_val2017
|   |   ├── train2017
|   |   ├── val2017
|   |   ├── test2017
```

Cityscape 数据集的标注格式需要转换，以与 COCO 数据集标注格式保持一致，使用 tools/dataset_converters/cityscapes.py 来完成转换：

```
pip install cityscapesscripts

python tools/dataset_converters/cityscapes.py \
    ./data/cityscapes \
    --nproc 8 \
    --out-dir ./data/cityscapes/annotations
```

3.4 测试现有模型

我们提供了测试脚本，能够测试一个现有模型在所有数据集（COCO，Pascal VOC，Cityscapes 等）上的性能。我们支持在如下环境下测试：

- 单 GPU 测试
- CPU 测试
- 单节点多 GPU 测试
- 多节点测试

根据以上测试环境，选择合适的脚本来执行测试过程。

```
# 单 GPU 测试
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--out ${RESULT_FILE}] \
    [--show]

# CPU 测试: 禁用 GPU 并运行单 GPU 测试脚本
export CUDA_VISIBLE_DEVICES=-1
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--out ${RESULT_FILE}] \
    [--show]

# 单节点多 GPU 测试
bash tools/dist_test.sh \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${GPU_NUM} \
    [--out ${RESULT_FILE}]
```

`tools/dist_test.sh` 也支持多节点测试，不过需要依赖 PyTorch 的 启动工具 。

可选参数:

- `RESULT_FILE`: 结果文件名称，需以.pkl 形式存储。如果没有声明，则不将结果存储到文件。
- `--show`: 如果开启，检测结果将被绘制在图像上，以一个新窗口的形式展示。它只适用于单 GPU 的测试，是用于调试和可视化的。请确保使用此功能时，你的 GUI 可以在环境中打开。否则，你可能会遇到这么一个错误 `cannot connect to X server`。
- `--show-dir`: 如果指明，检测结果将会被绘制在图像上并保存到指定目录。它只适用于单 GPU 的测试，是用于调试和可视化的。即使你的环境中没有 GUI，这个选项也可使用。
- `--cfg-options`: 如果指明，这里的键值对将会被合并到配置文件中。

3.4.1 样例

假设你已经下载了 checkpoint 文件到 `checkpoints/` 文件下了。

1. 测试 RTMDet 并可视化其结果。按任意键继续下张图片的测试。配置文件和 checkpoint 文件 在此 。

```
python tools/test.py \
    configs/rtmdet/rtmdet_l_8xb32-300e_coco.py \
```

(下页继续)

(续上页)

```
    checkpoints/rtmdet_l_8xb32-300e_coco_20220719_112030-5a0be7c4.pth \
    --show
```

2. 测试 RTMDet，并为了之后的可视化保存绘制的图像。配置文件和 checkpoint 文件 在此 。

```
python tools/test.py \
    configs/rtmdet/rtmdet_l_8xb32-300e_coco.py \
    checkpoints/rtmdet_l_8xb32-300e_coco_20220719_112030-5a0be7c4.pth \
    --show-dir rtmdet_l_8xb32-300e_coco_results
```

3. 在 Pascal VOC 数据集上测试 Faster R-CNN，不保存测试结果，测试 mAP。配置文件和 checkpoint 文件 在此。

```
python tools/test.py \
    configs/pascal_voc/faster-rcnn_r50_fpn_1x_voc0712.py \
    checkpoints/faster_rcnn_r50_fpn_1x_voc0712_20200624-c9895d40.pth
```

4. 使用 8 块 GPU 测试 Mask R-CNN，测试 bbox 和 mAP 。配置文件和 checkpoint 文件 在此。

```
./tools/dist_test.sh \
    configs/mask-rcnn_r50_fpn_1x_coco.py \
    checkpoints/mask_rcnn_r50_fpn_1x_coco_20200205-d4b0c5d6.pth \
    8 \
    --out results.pkl
```

5. 使用 8 块 GPU 测试 Mask R-CNN，测试**每类**的 bbox 和 mAP。配置文件和 checkpoint 文件 在此。

```
./tools/dist_test.sh \
    configs/mask_rcnn/mask-rcnn_r50_fpn_1x_coco.py \
    checkpoints/mask_rcnn_r50_fpn_1x_coco_20200205-d4b0c5d6.pth \
    8
```

该命令生成两个 JSON 文件 `./work_dirs/coco_instance/test.bbox.json` 和 `./work_dirs/coco_instance/test.segm.json`。

6. 在 COCO test-dev 数据集上，使用 8 块 GPU 测试 Mask R-CNN，并生成 JSON 文件提交到官方评测服务器，配置文件和 checkpoint 文件 在此。你可以在 config 的注释中用 test_evaluator 和 test_dataloader 替换原来的 test_evaluator 和 test_dataloader，然后运行：

```
./tools/dist_test.sh \
    configs/cityscapes/mask-rcnn_r50_fpn_1x_cityscapes.py \
    checkpoints/mask_rcnn_r50_fpn_1x_cityscapes_20200227-afe51d5a.pth \
    8
```

这行命令生成两个 JSON 文件 `mask_rcnn_test-dev_results.bbox.json` 和

mask_rcnn_test-dev_results.segm.json。

7. 在 Cityscapes 数据集上，使用 8 块 GPU 测试 Mask R-CNN，生成 txt 和 png 文件，并上传到官方评测服务器。配置文件和 checkpoint 文件 在此。你可以在 config 的注释中用 test_evaluator 和 test_dataloader 替换原来的 test_evaluator 和 test_dataloader，然后运行：

```
./tools/dist_test.sh \
    configs/cityscapes/mask-rcnn_r50_fpn_1x_cityscapes.py \
    checkpoints/mask_rcnn_r50_fpn_1x_cityscapes_20200227-afe51d5a.pth \
    8
```

生成的 png 和 txt 文件在 ./work_dirs/cityscapes_metric 文件夹下。

3.4.2 不使用 Ground Truth 标注进行测试

MMDetection 支持在不使用 ground-truth 标注的情况下对模型进行测试，这需要用到 CocoDataset。如果你的数据集格式不是 COCO 格式的，请将其转化成 COCO 格式。如果你的数据集格式是 VOC 或者 Cityscapes，你可以使用 tools/dataset_converters 内的脚本直接将其转化成 COCO 格式。如果是其他格式，可以使用 images2coco 脚本 进行转换。

```
python tools/dataset_converters/images2coco.py \
    ${IMG_PATH} \
    ${CLASSES} \
    ${OUT} \
    [--exclude-extensions]
```

参数：

- IMG_PATH: 图片根路径。
- CLASSES: 类列表文本文件名。文本中每一行存储一个类别。
- OUT: 输出 json 文件名。默认保存目录和 IMG_PATH 在同一级。
- exclude-extensions: 待排除的文件后缀名。

在转换完成后，使用如下命令进行测试

```
# 单 GPU 测试
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--show]

# CPU 测试：禁用 GPU 并运行单 GPU 测试脚本
export CUDA_VISIBLE_DEVICES=-1
python tools/test.py \
```

(下页继续)

(续上页)

```
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--out ${RESULT_FILE}] \
    [--show]

# 单节点多 GPU 测试
bash tools/dist_test.sh \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${GPU_NUM} \
    [--show]
```

假设 model zoo 中的 checkpoint 文件被下载到了 `checkpoints/` 文件夹下，我们可以使用以下命令，用 8 块 GPU 在 COCO test-dev 数据集上测试 Mask R-CNN，并且生成 JSON 文件。

```
./tools/dist_test.sh \
    configs/mask_rcnn/mask-rcnn_r50_fpn_1x_coco.py \
    checkpoints/mask_rcnn_r50_fpn_1x_coco_20200205-d4b0c5d6.pth \
    8
```

这行命令生成两个 JSON 文件 `./work_dirs/coco_instance/test.bbox.jso` 和 `./work_dirs/coco_instance/test.segm.jsonn`。

3.4.3 批量推理

MMDetection 在测试模式下，既支持单张图片的推理，也支持对图像进行批量推理。默认情况下，我们使用单张图片的测试，你可以通过修改测试数据配置文件中的 `samples_per_gpu` 来开启批量测试。开启批量推理的配置文件修改方法为：

```
data = dict(train_dataloader=dict(...), val_dataloader=dict(...), test_
↪dataloader=dict(batch_size=2, ...))
```

或者你可以通过将 `--cfg-options` 设置为 `--cfg-options test_dataloader.batch_size=` 来开启它。

3.4.4 测试时增强 (TTA)

测试时增强 (TTA) 是一种在测试阶段使用的数据增强策略。它对同一张图片应用不同的增强，例如翻转和缩放，用于模型推理，然后将每个增强后的图像的预测结果合并，以获得更准确的预测结果。为了让用户更容易使用 TTA，MMEngine 提供了 BaseTTAModel 类，允许用户根据自己的需求通过简单地扩展 BaseTTAModel 类来实现不同的 TTA 策略。

在 MMDetection 中，我们提供了 DetTTAModel 类，它继承自 BaseTTAModel。

使用案例

使用 TTA 需要两个步骤。首先，你需要在配置文件中添加 `tta_model` 和 `tta_pipeline`：

```
tta_model = dict(
    type='DetTTAModel',
    tta_cfg=dict(nms=dict(
                    type='nms',
                    iou_threshold=0.5),
                    max_per_img=100))

tta_pipeline = [
    dict(type='LoadImageFromFile',
        backend_args=None),
    dict(
        type='TestTimeAug',
        transforms=[[
            dict(type='Resize', scale=(1333, 800), keep_ratio=True)
        ], [ # It uses 2 flipping transformations (flipping and not flipping).
            dict(type='RandomFlip', prob=1.),
            dict(type='RandomFlip', prob=0.)
        ], [
            dict(
               type='PackDetInputs',
               meta_keys=('img_id', 'img_path', 'ori_shape',
                       'img_shape', 'scale_factor', 'flip',
                       'flip_direction'))
       ]])]
```

第二步，运行测试脚本时，设置 `--tta` 参数，如下所示：

```
# 单 GPU 测试
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
```

(下页继续)

(续上页)

```
    [--tta]

# CPU 测试: 禁用 GPU 并运行单 GPU 测试脚本
export CUDA_VISIBLE_DEVICES=-1
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    [--out ${RESULT_FILE}] \
    [--tta]

# 多 GPU 测试
bash tools/dist_test.sh \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    ${GPU_NUM} \
    [--tta]
```

你也可以自己修改 TTA 配置，例如添加缩放增强：

```
tta_model = dict(
    type='DetTTAModel',
    tta_cfg=dict(nms=dict(
                    type='nms',
                    iou_threshold=0.5),
                    max_per_img=100))

img_scales = [(1333, 800), (666, 400), (2000, 1200)]
tta_pipeline = [
    dict(type='LoadImageFromFile',
         backend_args=None),
    dict(
        type='TestTimeAug',
        transforms=[[
            dict(type='Resize', scale=s, keep_ratio=True) for s in img_scales
        ], [
            dict(type='RandomFlip', prob=1.),
            dict(type='RandomFlip', prob=0.)
        ], [
            dict(
               type='PackDetInputs',
               meta_keys=('img_id', 'img_path', 'ori_shape',
                       'img_shape', 'scale_factor', 'flip',
                       'flip_direction'))
       ]])]
```

以上数据增强管道将首先对图像执行 3 个多尺度转换，然后执行 2 个翻转转换（翻转和不翻转），最后使用 PackDetInputs 将图像打包到最终结果中。这里有更多的 TTA 使用案例供您参考：

- RetinaNet
- CenterNet
- YOLOX
- RTMDet

更多高级用法和 TTA 的数据流，请参考 MMEngine。我们将在后续支持实例分割 TTA。

3.5 在标准数据集上训练预定义的模型（待更新）

MMDetection 也为训练检测模型提供了开盖即食的工具。本节将展示在标准数据集（比如 COCO）上如何训练一个预定义的模型。

3.5.1 数据集

训练需要准备好数据集，细节请参考 数据集准备。

注意：目前，`configs/cityscapes` 文件夹下的配置文件都是使用 COCO 预训练权值进行初始化的。如果网络连接不可用或者速度很慢，你可以提前下载现存的模型。否则可能在训练的开始会有错误发生。

3.5.2 学习率自动缩放

注意:在配置文件中的学习率是在 8 块 GPU,每块 GPU 有 2 张图像(批大小为 8*2=16)的情况下设置的。其已经设置在 `config/_base_/schedules/schedule_1x.py` 中的 `auto_scale_lr.base_batch_size`。当配置文件的批次大小为 `16` 时，学习率会基于该值进行自动缩放。同时，为了不影响其他基于 mmdet 的 codebase，启用自动缩放标志 `auto_scale_lr.enable` 默认设置为 `False`。

如果要启用此功能，需在命令添加参数 `--auto-scale-lr`。并且在启动命令之前，请检查下即将使用的配置文件的名称，因为配置名称指示默认的批处理大小。在默认情况下，批次大小是 `8 x 2 = 16`，例如：`faster_rcnn_r50_caffe_fpn_90k_coco.py` 或者 `pisa_faster_rcnn_x101_32x4d_fpn_1x_coco.py`；若不是默认批次，你可以在配置文件看到像 `_NxM_` 字样的，例如：`cornernet_hourglass104_mstest_32x3_210e_coco.py` 的批次大小是 `32 x 3 = 96`,或者 `scnet_x101_64x4d_fpn_8x1_20e_coco.py` 的批次大小是 `8 x 1 = 8`。

请记住：如果使用不是默认批次大小为 `16` 的配置文件，请检查配置文件中的底部，会有 `auto_scale_lr.base_batch_size`。如果找不到，可以在其继承的 `_base_=[xxx]` 文件中找到。另外，如果想使用自动缩放学习率的功能，请不要修改这些值。

学习率自动缩放基本用法如下：

```
python tools/train.py \
    ${CONFIG_FILE} \
    --auto-scale-lr \
    [optional arguments]
```

执行命令之后，会根据机器的 GPU 数量和训练的批次大小对学习率进行自动缩放，缩放方式详见 线性扩展规则，比如：在 4 块 GPU 并且每张 GPU 上有 2 张图片的情况下 `lr=0.01`，那么在 16 块 GPU 并且每张 GPU 上有 4 张图片的情况下, LR 会自动缩放至 `lr=0.08`。

如果不启用该功能，则需要根据 线性扩展规则 来手动计算并修改配置文件里面 `optimizer.lr` 的值。

3.5.3 使用单 GPU 训练

我们提供了 `tools/train.py` 来开启在单张 GPU 上的训练任务。基本使用如下：

```
python tools/train.py \
    ${CONFIG_FILE} \
    [optional arguments]
```

在训练期间，日志文件和 checkpoint 文件将会被保存在工作目录下，它需要通过配置文件中的 `work_dir` 或者 CLI 参数中的 `--work-dir` 来指定。

默认情况下，模型将在每轮训练之后在 validation 集上进行测试，测试的频率可以通过设置配置文件来指定：

```
# 每 12 轮迭代进行一次测试评估
evaluation = dict(interval=12)
```

这个工具接受以下参数：

- `--no-validate` (**不建议**): 在训练期间关闭测试.
- `--work-dir ${WORK_DIR}`: 覆盖工作目录.
- `--resume-from ${CHECKPOINT_FILE}`: 从某个 checkpoint 文件继续训练.
- `--options 'Key=value'`: 覆盖使用的配置文件中的其他设置.

注意：`resume-from` 和 `load-from` 的区别：

`resume-from` 既加载了模型的权重和优化器的状态，也会继承指定 checkpoint 的迭代次数，不会重新开始训练。`load-from` 则是只加载模型的权重，它的训练是从头开始的，经常被用于微调模型。

3.5.4 使用 CPU 训练

使用 CPU 训练的流程和使用单 GPU 训练的流程一致，我们仅需要在训练流程开始前禁用 GPU。

```
export CUDA_VISIBLE_DEVICES=-1
```

之后运行单 GPU 训练脚本即可。

注意：

我们不推荐用户使用 CPU 进行训练，这太过缓慢。我们支持这个功能是为了方便用户在没有 GPU 的机器上进行调试。

3.5.5 在多 GPU 上训练

我们提供了 `tools/dist_train.sh` 来开启在多 GPU 上的训练。基本使用如下：

```
bash ./tools/dist_train.sh \
    ${CONFIG_FILE} \
    ${GPU_NUM} \
    [optional arguments]
```

可选参数和单 GPU 训练的可选参数一致。

同时启动多个任务

如果你想在一台机器上启动多个任务的话，比如在一个有 8 块 GPU 的机器上启动 2 个需要 4 块 GPU 的任务，你需要给不同的训练任务指定不同的端口（默认为 29500）来避免冲突。

如果你使用 `dist_train.sh` 来启动训练任务，你可以使用命令来设置端口。

```
CUDA_VISIBLE_DEVICES=0,1,2,3 PORT=29500 ./tools/dist_train.sh ${CONFIG_FILE} 4
CUDA_VISIBLE_DEVICES=4,5,6,7 PORT=29501 ./tools/dist_train.sh ${CONFIG_FILE} 4
```

3.5.6 使用多台机器训练

如果您想使用由 ethernet 连接起来的多台机器，您可以使用以下命令:

在第一台机器上:

```
NNODES=2 NODE_RANK=0 PORT=$MASTER_PORT MASTER_ADDR=$MASTER_ADDR sh tools/dist_train.
↪sh $CONFIG $GPUS
```

在第二台机器上:

```
NNODES=2 NODE_RANK=1 PORT=$MASTER_PORT MASTER_ADDR=$MASTER_ADDR sh tools/dist_train.
↪sh $CONFIG $GPUS
```

但是，如果您不使用高速网路连接这几台机器的话，训练将会非常慢。

3.5.7 使用 Slurm 来管理任务

Slurm 是一个常见的计算集群调度系统。在 Slurm 管理的集群上，你可以使用 `slurm.sh` 来开启训练任务。它既支持单节点训练也支持多节点训练。

基本使用如下：

```
[GPUS=${GPUS}] ./tools/slurm_train.sh ${PARTITION} ${JOB_NAME} ${CONFIG_FILE} ${WORK_
↪DIR}
```

以下是在一个名称为 *dev* 的 Slurm 分区上，使用 16 块 GPU 来训练 Mask R-CNN 的例子，并且将 `work-dir` 设置在了某些共享文件系统下。

```
GPUS=16 ./tools/slurm_train.sh dev mask_r50_1x configs/mask_rcnn_r50_fpn_1x_coco.py /
↪nfs/xxxx/mask_rcnn_r50_fpn_1x
```

你可以查看 源码 来检查全部的参数和环境变量.

在使用 Slurm 时，端口需要以下方的某个方法之一来设置。

1. 通过 `--options` 来设置端口。我们非常建议用这种方法，因为它无需改变原始的配置文件。

   ```
   CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_
   ↪NAME} config1.py ${WORK_DIR} --options 'dist_params.port=29500'
   CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_
   ↪NAME} config2.py ${WORK_DIR} --options 'dist_params.port=29501'
   ```

2. 修改配置文件来设置不同的交流端口。

   在 `config1.py` 中，设置：

   ```
   dist_params = dict(backend='nccl', port=29500)
   ```

   在 `config2.py` 中，设置：

   ```
   dist_params = dict(backend='nccl', port=29501)
   ```

   然后你可以使用 `config1.py` 和 `config2.py` 来启动两个任务了。

```
CUDA_VISIBLE_DEVICES=0,1,2,3 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_
↪NAME} config1.py ${WORK_DIR}
CUDA_VISIBLE_DEVICES=4,5,6,7 GPUS=4 ./tools/slurm_train.sh ${PARTITION} ${JOB_
↪NAME} config2.py ${WORK_DIR}
```

3.6 在自定义数据集上进行训练

通过本文档，你将会知道如何使用自定义数据集对预先定义好的模型进行推理，测试以及训练。我们使用 balloon dataset 作为例子来描述整个过程。

基本步骤如下：

1. 准备自定义数据集
2. 准备配置文件
3. 在自定义数据集上进行训练，测试和推理。

3.6.1 准备自定义数据集

MMDetection 一共支持三种形式应用新数据集：

1. 将数据集重新组织为 COCO 格式。
2. 将数据集重新组织为一个中间格式。
3. 实现一个新的数据集。

我们通常建议使用前面两种方法，因为它们通常来说比第三种方法要简单。

在本文档中，我们展示一个例子来说明如何将数据转化为 COCO 格式。

注意：在 MMDetection 3.0 之后，数据集和指标已经解耦（除了 CityScapes）。因此，用户在验证阶段使用任意的评价指标来评价模型在任意数据集上的性能。比如，用 VOC 评价指标来评价模型在 COCO 数据集的性能，或者同时使用 VOC 评价指标和 COCO 评价指标来评价模型在 OpenImages 数据集上的性能。

COCO 标注格式

用于实例分割的 COCO 数据集格式如下所示，其中的键（key）都是必要的，参考这里来获取更多细节。

```
{
    "images": [image],
    "annotations": [annotation],
    "categories": [category]
}
```

(下页继续)

(续上页)

```
image = {
    "id": int,
    "width": int,
    "height": int,
    "file_name": str,
}

annotation = {
    "id": int,
    "image_id": int,
    "category_id": int,
    "segmentation": RLE or [polygon],
    "area": float,
    "bbox": [x,y,width,height], # (x, y) 为 bbox 左上角的坐标
    "iscrowd": 0 or 1,
}

categories = [{
    "id": int,
    "name": str,
    "supercategory": str,
}]
```

现在假设我们使用 balloon dataset。

下载了数据集之后，我们需要实现一个函数将标注格式转化为 COCO 格式。然后我们就可以使用已经实现的 `CocoDataset` 类来加载数据并进行训练以及评测。

如果你浏览过新数据集，你会发现格式如下：

```
{'base64_img_data': '',
 'file_attributes': {},
 'filename': '34020010494_e5cb88e1c4_k.jpg',
 'fileref': '',
 'regions': {'0': {'region_attributes': {},
   'shape_attributes': {'all_points_x': [1020,
     1000,
     994,
     1003,
     1023,
     1050,
     1089,
     1134,
```

(下页继续)

(续上页)

```
    1190,
    1265,
    1321,
    1361,
    1403,
    1428,
    1442,
    1445,
    1441,
    1427,
    1400,
    1361,
    1316,
    1269,
    1228,
    1198,
    1207,
    1210,
    1190,
    1177,
    1172,
    1174,
    1170,
    1153,
    1127,
    1104,
    1061,
    1032,
    1020],
   'all_points_y': [963,
    899,
    841,
    787,
    738,
    700,
    663,
    638,
    621,
    619,
    643,
    672,
    720,
    765,
```

(下页继续)

(续上页)

```
    800,
    860,
    896,
    942,
    990,
    1035,
    1079,
    1112,
    1129,
    1134,
    1144,
    1153,
    1166,
    1166,
    1150,
    1136,
    1129,
    1122,
    1112,
    1084,
    1037,
    989,
    963],
   'name': 'polygon'}}},
 'size': 1115004}
```

标注文件时是 JSON 格式的，其中所有键（key）组成了一张图片的所有标注。

其中将 balloon dataset 转化为 COCO 格式的代码如下所示。

```
import os.path as osp

import mmcv

from mmengine.fileio import dump, load
from mmengine.utils import track_iter_progress


def convert_balloon_to_coco(ann_file, out_file, image_prefix):
    data_infos = load(ann_file)

    annotations = []
    images = []
    obj_count = 0
```

(下页继续)

(续上页)

```
    for idx, v in enumerate(track_iter_progress(data_infos.values())):
        filename = v['filename']
        img_path = osp.join(image_prefix, filename)
        height, width = mmcv.imread(img_path).shape[:2]

        images.append(
            dict(id=idx, file_name=filename, height=height, width=width))

        for _, obj in v['regions'].items():
            assert not obj['region_attributes']
            obj = obj['shape_attributes']
            px = obj['all_points_x']
            py = obj['all_points_y']
            poly = [(x + 0.5, y + 0.5) for x, y in zip(px, py)]
            poly = [p for x in poly for p in x]

            x_min, y_min, x_max, y_max = (min(px), min(py), max(px), max(py))

            data_anno = dict(
                image_id=idx,
                id=obj_count,
                category_id=0,
                bbox=[x_min, y_min, x_max - x_min, y_max - y_min],
                area=(x_max - x_min) * (y_max - y_min),
                segmentation=[poly],
                iscrowd=0)
            annotations.append(data_anno)
            obj_count += 1

    coco_format_json = dict(
        images=images,
        annotations=annotations,
        categories=[{
            'id': 0,
            'name': 'balloon'
        }])
    dump(coco_format_json, out_file)


if __name__ == '__main__':
    convert_balloon_to_coco(ann_file='data/balloon/train/via_region_data.json',
                            out_file='data/balloon/train/annotation_coco.json',
                            image_prefix='data/balloon/train')
```

(下页继续)

(续上页)

```
    convert_balloon_to_coco(ann_file='data/balloon/val/via_region_data.json',
                            out_file='data/balloon/val/annotation_coco.json',
                            image_prefix='data/balloon/val')
```

使用如上的函数，用户可以成功将标注文件转化为 JSON 格式，之后可以使用 `CocoDataset` 对模型进行训练，并用 `CocoMetric` 评测。

3.6.2 准备配置文件

第二步需要准备一个配置文件来成功加载数据集。假设我们想要用 balloon dataset 来训练配备了 FPN 的 Mask R-CNN，如下是我们的配置文件。假设配置文件命名为 `mask-rcnn_r50-caffe_fpn_ms-poly-1x_balloon.py`，相应保存路径为 `configs/balloon/`，配置文件内容如下所示。

```
# 新配置继承了基本配置，并做了必要的修改
_base_ = '../mask_rcnn/mask-rcnn_r50-caffe_fpn_ms-poly-1x_coco.py'

# 我们还需要更改 head 中的 num_classes 以匹配数据集中的类别数
model = dict(
    roi_head=dict(
        bbox_head=dict(num_classes=1), mask_head=dict(num_classes=1)))

# 修改数据集相关配置
data_root = 'data/balloon/'
metainfo = {
    'classes': ('balloon', ),
    'palette': [
        (220, 20, 60),
    ]
}
train_dataloader = dict(
    batch_size=1,
    dataset=dict(
        data_root=data_root,
        metainfo=metainfo,
        ann_file='train/annotation_coco.json',
        data_prefix=dict(img='train/')))
val_dataloader = dict(
    dataset=dict(
        data_root=data_root,
        metainfo=metainfo,
        ann_file='val/annotation_coco.json',
        data_prefix=dict(img='val/')))
```

(下页继续)

(续上页)

```
test_dataloader = val_dataloader

# 修改评价指标相关配置
val_evaluator = dict(ann_file=data_root + 'val/annotation_coco.json')
test_evaluator = val_evaluator

# 使用预训练的 Mask R-CNN 模型权重来做初始化，可以提高模型性能
load_from = 'https://download.openmmlab.com/mmdetection/v2.0/mask_rcnn/mask_rcnn_r50_
↪caffe_fpn_mstrain-poly_3x_coco/mask_rcnn_r50_caffe_fpn_mstrain-poly_3x_coco_bbox_
↪mAP-0.408__segm_mAP-0.37_20200504_163245-42aa3d00.pth'
```

3.6.3 训练一个新的模型

为了使用新的配置方法来对模型进行训练，你只需要运行如下命令。

```
python tools/train.py configs/balloon/mask-rcnn_r50-caffe_fpn_ms-poly-1x_balloon.py
```

参考 在标准数据集上训练预定义的模型 来获取更多详细的使用方法。

3.6.4 测试以及推理

为了测试训练完毕的模型，你只需要运行如下命令。

```
python tools/test.py configs/balloon/mask-rcnn_r50-caffe_fpn_ms-poly-1x_balloon.py␣
↪work_dirs/mask-rcnn_r50-caffe_fpn_ms-poly-1x_balloon/epoch_12.pth
```

参考 测试现有模型 来获取更多详细的使用方法。

3.7 在标准数据集上训练自定义模型（待更新）

在本文中，你将知道如何在标准数据集上训练、测试和推理自定义模型。我们将在 cityscapes 数据集上以自定义 Cascade Mask R-CNN R50 模型为例演示整个过程，为了方便说明，我们将 neck 模块中的 `FPN` 替换为 `AugFPN`，并且在训练中的自动增强类中增加 `Rotate` 或 `TranslateX`。

基本步骤如下所示：

1. 准备标准数据集
2. 准备你的自定义模型
3. 准备配置文件
4. 在标准数据集上对模型进行训练、测试和推理

3.7.1 准备标准数据集

在本文中，我们使用 cityscapes 标准数据集为例进行说明。

推荐将数据集根路径采用符号链接方式链接到 `$MMDETECTION/data`。

如果你的文件结构不同，你可能需要在配置文件中进行相应的路径更改。标准的文件组织格式如下所示：

```
mmdetection
├── mmdet
├── tools
├── configs
├── data
│   ├── coco
│   │   ├── annotations
│   │   ├── train2017
│   │   ├── val2017
│   │   ├── test2017
│   ├── cityscapes
│   │   ├── annotations
│   │   ├── leftImg8bit
│   │   │   ├── train
│   │   │   ├── val
│   │   ├── gtFine
│   │   │   ├── train
│   │   │   ├── val
│   ├── VOCdevkit
│   │   ├── VOC2007
│   │   ├── VOC2012
```

你也可以通过如下方式设定数据集根路径

```
export MMDET_DATASETS=$data_root
```

我们将会使用环境便变量 `$MMDET_DATASETS` 作为数据集的根目录，因此你无需再修改相应配置文件的路径信息。

你需要使用脚本 `tools/dataset_converters/cityscapes.py` 将 cityscapes 标注转化为 coco 标注格式。

```
pip install cityscapesscripts
python tools/dataset_converters/cityscapes.py ./data/cityscapes --nproc 8 --out-dir ./
↪data/cityscapes/annotations
```

目前在 `cityscapes` 文件夹中的配置文件所对应模型是采用 COCO 预训练权重进行初始化的。

如果你的网络不可用或者比较慢，建议你先手动下载对应的预训练权重，否则可能在训练开始时候出现错误。

3.7.2 准备你的自定义模型

第二步是准备你的自定义模型或者训练相关配置。假设你想在已有的 Cascade Mask R-CNN R50 检测模型基础上，新增一个新的 neck 模块 `AugFPN` 去代替默认的 `FPN`，以下是具体实现：

1 定义新的 neck (例如 AugFPN)

首先创建新文件 `mmdet/models/necks/augfpn.py`.

```
import torch.nn as nn
from mmdet.registry import MODELS

@MODELS.register_module()
class AugFPN(nn.Module):

    def __init__(self,
                 in_channels,
                 out_channels,
                 num_outs,
                 start_level=0,
                 end_level=-1,
                 add_extra_convs=False):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

2 导入模块

你可以采用两种方式导入模块，第一种是在 `mmdet/models/necks/__init__.py` 中添加如下内容

```
from .augfpn import AugFPN
```

第二种是增加如下代码到对应配置中，这种方式的好处是不需要改动代码

```
custom_imports = dict(
    imports=['mmdet.models.necks.augfpn'],
    allow_failed_imports=False)
```

3 修改配置

```
neck=dict(
    type='AugFPN',
    in_channels=[256, 512, 1024, 2048],
    out_channels=256,
    num_outs=5)
```

关于自定义模型其余相关细节例如实现新的骨架网络，头部网络、损失函数，以及运行时训练配置例如定义新的优化器、使用梯度裁剪、定制训练调度策略和钩子等，请参考文档 自定义模型和 自定义运行时训练配置。

3.7.3 准备配置文件

第三步是准备训练配置所需要的配置文件。假设你打算基于 cityscapes 数据集，在 Cascade Mask R-CNN R50 中新增 AugFPN 模块，同时增加 Rotate 或者 Translate 数据增强策略，假设你的配置文件位于 configs/cityscapes/ 目录下，并且取名为 cascade-mask-rcnn_r50_augfpn_autoaug-10e_cityscapes.py，则配置信息如下：

```
# 继承 base 配置，然后进行针对性修改
_base_ = [
    '../_base_/models/cascade-mask-rcnn_r50_fpn.py',
    '../_base_/datasets/cityscapes_instance.py', '../_base_/default_runtime.py'
]

model = dict(
    # 设置 `init_cfg` 为 None，表示不加载 ImageNet 预训练权重，
    # 后续可以设置 `load_from` 参数用来加载 COCO 预训练权重
    backbone=dict(init_cfg=None),
    # 使用新增的 `AugFPN` 模块代替默认的 `FPN`
    neck=dict(
        type='AugFPN',
        in_channels=[256, 512, 1024, 2048],
        out_channels=256,
        num_outs=5),
    # 我们也需要将 num_classes 从 80 修改为 8 来匹配 cityscapes 数据集标注
    # 这个修改包括 `bbox_head` 和 `mask_head`.
    roi_head=dict(
        bbox_head=[
            dict(
                type='Shared2FCBBoxHead',
                in_channels=256,
                fc_out_channels=1024,
```

(下页继续)

(续上页)

```
                roi_feat_size=7,
                # 将 COCO 类别修改为 cityscapes 类别
                num_classes=8,
                bbox_coder=dict(
                    type='DeltaXYWHBBoxCoder',
                    target_means=[0., 0., 0., 0.],
                    target_stds=[0.1, 0.1, 0.2, 0.2]),
                reg_class_agnostic=True,
                loss_cls=dict(
                    type='CrossEntropyLoss',
                    use_sigmoid=False,
                    loss_weight=1.0),
                loss_bbox=dict(type='SmoothL1Loss', beta=1.0,
                               loss_weight=1.0)),
            dict(
                type='Shared2FCBBoxHead',
                in_channels=256,
                fc_out_channels=1024,
                roi_feat_size=7,
                # 将 COCO 类别修改为 cityscapes 类别
                num_classes=8,
                bbox_coder=dict(
                    type='DeltaXYWHBBoxCoder',
                    target_means=[0., 0., 0., 0.],
                    target_stds=[0.05, 0.05, 0.1, 0.1]),
                reg_class_agnostic=True,
                loss_cls=dict(
                    type='CrossEntropyLoss',
                    use_sigmoid=False,
                    loss_weight=1.0),
                loss_bbox=dict(type='SmoothL1Loss', beta=1.0,
                               loss_weight=1.0)),
            dict(
                type='Shared2FCBBoxHead',
                in_channels=256,
                fc_out_channels=1024,
                roi_feat_size=7,
                # 将 COCO 类别修改为 cityscapes 类别
                num_classes=8,
                bbox_coder=dict(
                    type='DeltaXYWHBBoxCoder',
                    target_means=[0., 0., 0., 0.],
                    target_stds=[0.033, 0.033, 0.067, 0.067]),
```

(下页继续)

(续上页)

```
                reg_class_agnostic=True,
                loss_cls=dict(
                    type='CrossEntropyLoss',
                    use_sigmoid=False,
                    loss_weight=1.0),
                loss_bbox=dict(type='SmoothL1Loss', beta=1.0, loss_weight=1.0))
        ],
        mask_head=dict(
            type='FCNMaskHead',
            num_convs=4,
            in_channels=256,
            conv_out_channels=256,
            # 将 COCO 类别修改为 cityscapes 类别
            num_classes=8,
            loss_mask=dict(
                type='CrossEntropyLoss', use_mask=True, loss_weight=1.0))))

# 覆写 `train_pipeline`，然后新增 `AutoAugment` 训练配置
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True, with_mask=True),
    dict(
        type='AutoAugment',
        policies=[
            [dict(
                 type='Rotate',
                 level=5,
                 img_fill_val=(124, 116, 104),
                 prob=0.5,
                 scale=1)
            ],
            [dict(type='Rotate', level=7, img_fill_val=(124, 116, 104)),
             dict(
                 type='TranslateX',
                 level=5,
                 prob=0.5,
                 img_fill_val=(124, 116, 104))
            ],
        ]),
    dict(
        type='RandomResize',
        scale=[(2048, 800), (2048, 1024)],
        keep_ratio=True),
```

(下页继续)

(续上页)

```
    dict(type='RandomFlip', prob=0.5),
    dict(type='PackDetInputs'),
]

# 设置每张显卡的批处理大小，同时设置新的训练 pipeline
data = dict(
    samples_per_gpu=1,
    workers_per_gpu=3,
    train=dict(dataset=dict(pipeline=train_pipeline)))

# 设置优化器
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001))

# 设置定制的学习率策略
param_scheduler = [
    dict(
        type='LinearLR', start_factor=0.001, by_epoch=False, begin=0, end=500),
    dict(
        type='MultiStepLR',
        begin=0,
        end=10,
        by_epoch=True,
        milestones=[8],
        gamma=0.1)
]

# 训练，验证，测试配置
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=10, val_interval=1)
val_cfg = dict(type='ValLoop')
test_cfg = dict(type='TestLoop')

# 我们采用 COCO 预训练过的 Cascade Mask R-CNN R50 模型权重作为初始化权重，可以得到更加稳定的性能
load_from = 'https://download.openmmlab.com/mmdetection/v2.0/cascade_rcnn/cascade_
↪mask_rcnn_r50_fpn_1x_coco/cascade_mask_rcnn_r50_fpn_1x_coco_20200203-9d4dcb24.pth'
```

3.7.4 训练新模型

为了能够使用新增配置来训练模型，你可以运行如下命令：

```
python tools/train.py configs/cityscapes/cascade-mask-rcnn_r50_augfpn_autoaug-10e_
↪cityscapes.py
```

如果想了解更多用法，可以参考 例子 1。

3.7.5 测试和推理

为了能够测试训练好的模型，你可以运行如下命令：

```
python tools/test.py configs/cityscapes/cascade-mask-rcnn_r50_augfpn_autoaug-10e_
↪cityscapes.py work_dirs/cascade-mask-rcnn_r50_augfpn_autoaug-10e_cityscapes/epoch_
↪10.pth
```

如果想了解更多用法，可以参考 例子 1。

3.8 模型微调

在 COCO 数据集上预训练的检测器可以作为其他数据集（例如 CityScapes 和 KITTI 数据集）优质的预训练模型。本教程将指导用户如何把*ModelZoo* 中提供的模型用于其他数据集中并使得当前所训练的模型获得更好性能。

以下是在新数据集中微调模型需要的两个步骤。

- 按 教程 2：自定义数据集的方法中的方法对新数据集添加支持中的方法对新数据集添加支持
- 按照本教程中所讨论方法，修改配置信息

接下来将会以 Cityscapes Dataset 上的微调过程作为例子，具体讲述用户需要在配置中修改的五个部分。

3.8.1 继承基础配置

为了减轻编写整个配置的负担并减少漏洞的数量，MMDetection V3.0 支持从多个现有配置中继承配置信息。微调 MaskRCNN 模型的时候，新的配置信息需要使用从 `_base_/models/mask_rcnn_r50_fpn.py` 中继承的配置信息来构建模型的基本结构。当使用 Cityscapes 数据集时，新的配置信息可以简便地从 `_base_/datasets/cityscapes_instance.py` 中继承。对于训练过程的运行设置部分，例如 `logger settings`，配置文件可以从 `_base_/default_runtime.py` 中继承。对于训练计划的配置则可以从 `_base_/schedules/schedule_1x.py` 中继承。这些配置文件存放于 `configs` 目录下，用户可以选择全部内容的重新编写而不是使用继承方法。

```
_base_ = [
    '../_base_/models/mask_rcnn_r50_fpn.py',
    '../_base_/datasets/cityscapes_instance.py', '../_base_/default_runtime.py',
    '../_base_/schedules/schedule_1x.py'
]
```

3.8.2 Head 的修改

接下来新的配置还需要根据新数据集的类别数量对 Head 进行修改。只需要对 roi_head 中的 `num_classes` 进行修改。修改后除了最后的预测模型的 Head 之外，预训练模型的权重的大部分都会被重新使用。

```
model = dict(
    roi_head=dict(
        bbox_head=dict(
            type='Shared2FCBBoxHead',
            in_channels=256,
            fc_out_channels=1024,
            roi_feat_size=7,
            num_classes=8,
            bbox_coder=dict(
                type='DeltaXYWHBBoxCoder',
                target_means=[0., 0., 0., 0.],
                target_stds=[0.1, 0.1, 0.2, 0.2]),
            reg_class_agnostic=False,
            loss_cls=dict(
                type='CrossEntropyLoss', use_sigmoid=False, loss_weight=1.0),
            loss_bbox=dict(type='SmoothL1Loss', beta=1.0, loss_weight=1.0)),
        mask_head=dict(
            type='FCNMaskHead',
            num_convs=4,
            in_channels=256,
            conv_out_channels=256,
            num_classes=8,
            loss_mask=dict(
                type='CrossEntropyLoss', use_mask=True, loss_weight=1.0))))
```

3.8.3 数据集的修改

用户可能还需要准备数据集并编写有关数据集的配置，可在*Customize Datasets* 中获取更多信息。目前 MMDetection V3.0 的配置文件已经支持 VOC、WIDERFACE、COCO、LIVS、OpenImages、DeepFashion、Objects365 和 Cityscapes Dataset 的数据集信息。

3.8.4 训练策略的修改

微调超参数与默认的训练策略不同。它通常需要更小的学习率和更少的训练回合。

```
# 优化器
# batch size 为 8 时的 lr 配置
optim_wrapper = dict(optimizer=dict(lr=0.01))

# 学习率
param_scheduler = [
    dict(
        type='LinearLR', start_factor=0.001, by_epoch=False, begin=0, end=500),
    dict(
        type='MultiStepLR',
        begin=0,
        end=8,
        by_epoch=True,
        milestones=[7],
        gamma=0.1)
]

# 设置 max epoch
train_cfg = dict(max_epochs=8)

# 设置 log config
default_hooks = dict(logger=dict(interval=100)),
```

3.8.5 使用预训练模型

如果要使用预训练模型，可以在 `load_from` 中查阅新的配置信息，用户需要在训练开始之前下载好需要的模型权重，从而避免在训练过程中浪费了宝贵时间。

```
load_from = 'https://download.openmmlab.com/mmdetection/v2.0/mask_rcnn/mask_rcnn_r50_
↪caffe_fpn_mstrain-poly_3x_coco/mask_rcnn_r50_caffe_fpn_mstrain-poly_3x_coco_bbox_
↪mAP-0.408__segm_mAP-0.37_20200504_163245-42aa3d00.pth'  # noqa
```

3.9 提交测试结果

3.9.1 全景分割测试结果提交

下面几节介绍如何在 COCO 测试开发集上生成泛视分割模型的预测结果，并将预测提交到 COCO 评估服务器

前提条件

- 下载 COCO 测试数据集图像，测试图像信息，和全景训练/相关注释，然后解压缩它们，把 `test2017` 放到 `data/coco/`，把 json 文件和注释文件放到 `data/coco/annotations/` 。

```
# 假设 data/coco/ 不存在
mkdir -pv data/coco/
# 下载 test2017
wget -P data/coco/ http://images.cocodataset.org/zips/test2017.zip
wget -P data/coco/ http://images.cocodataset.org/annotations/image_info_test2017.zip
wget -P data/coco/ http://images.cocodataset.org/annotations/panoptic_annotations_
↪trainval2017.zip
# 解压缩它们
unzip data/coco/test2017.zip -d data/coco/
unzip data/coco/image_info_test2017.zip -d data/coco/
unzip data/coco/panoptic_annotations_trainval2017.zip -d data/coco/
# 删除 zip 文件 (可选)
rm -rf data/coco/test2017.zip data/coco/image_info_test2017.zip data/coco/panoptic_
↪annotations_trainval2017.zip
```

- 运行以下代码更新测试图像信息中的类别信息。由于 `image_info_test-dev2017.json` 的类别信息中缺少属性 `isthing` ，我们需要用 `panoptic_val2017.json` 中的类别信息更新它。

```
python tools/misc/gen_coco_panoptic_test_info.py data/coco/annotations
```

在完成上述准备之后，你的 `data` 目录结构应该是这样:

```
data
`-- coco
    |-- annotations
    |   |-- image_info_test-dev2017.json
    |   |-- image_info_test2017.json
    |   |-- panoptic_image_info_test-dev2017.json
    |   |-- panoptic_train2017.json
    |   |-- panoptic_train2017.zip
    |   |-- panoptic_val2017.json
```

(下页继续)

(续上页)

```
|   `-- panoptic_val2017.zip
`-- test2017
```

coco 测试开发的推理

要在 coco test-dev 上进行推断，我们应该首先更新 `test_dataloder` 和 `test_evaluator` 的设置。有两种方法可以做到这一点:1. 在配置文件中更新它们;2. 在命令行中更新它们。

在配置文件中更新它们

相关的设置在 `configs/_base_/datasets/ coco_panoptical .py` 的末尾，如下所示。

```
test_dataloader = dict(
    batch_size=1,
    num_workers=1,
    persistent_workers=True,
    drop_last=False,
    sampler=dict(type='DefaultSampler', shuffle=False),
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        ann_file='annotations/panoptic_image_info_test-dev2017.json',
        data_prefix=dict(img='test2017/'),
        test_mode=True,
        pipeline=test_pipeline))
test_evaluator = dict(
    type='CocoPanopticMetric',
    format_only=True,
    ann_file=data_root + 'annotations/panoptic_image_info_test-dev2017.json',
    outfile_prefix='./work_dirs/coco_panoptic/test')
```

以下任何一种方法都可以用于更新 coco test-dev 集上的推理设置

情况 1: 直接取消注释 `configs/_base_/datasets/ coco_panoptical .py` 中的设置。

情况 2: 将以下设置复制到您现在使用的配置文件中。

```
test_dataloader = dict(
    dataset=dict(
        ann_file='annotations/panoptic_image_info_test-dev2017.json',
        data_prefix=dict(img='test2017/', _delete_=True)))
test_evaluator = dict(
    format_only=True,
```

(下页继续)

(续上页)

```
    ann_file=data_root + 'annotations/panoptic_image_info_test-dev2017.json',
    outfile_prefix='./work_dirs/coco_panoptic/test')
```

然后通过以下命令对 coco test-dev et 进行推断。

```
python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE}
```

在命令行中更新它们

coco test-dev 上更新相关设置和推理的命令如下所示。

```
# 用一个 gpu 测试
CUDA_VISIBLE_DEVICES=0 python tools/test.py \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    --cfg-options \
    test_dataloader.dataset.ann_file=annotations/panoptic_image_info_test-dev2017.
↪json \
    test_dataloader.dataset.data_prefix.img=test2017 \
    test_dataloader.dataset.data_prefix._delete_=True \
    test_evaluator.format_only=True \
    test_evaluator.ann_file=data/coco/annotations/panoptic_image_info_test-dev2017.
↪json \
    test_evaluator.outfile_prefix=${WORK_DIR}/results
# 用四个 gpu 测试
CUDA_VISIBLE_DEVICES=0,1,3,4 bash tools/dist_test.sh \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    8 \  # eights gpus
    --cfg-options \
    test_dataloader.dataset.ann_file=annotations/panoptic_image_info_test-dev2017.
↪json \
    test_dataloader.dataset.data_prefix.img=test2017 \
    test_dataloader.dataset.data_prefix._delete_=True \
    test_evaluator.format_only=True \
    test_evaluator.ann_file=data/coco/annotations/panoptic_image_info_test-dev2017.
↪json \
    test_evaluator.outfile_prefix=${WORK_DIR}/results
# 用 slurm 测试
GPUS=8 tools/slurm_test.sh \
    ${Partition} \
```

(下页继续)

(续上页)

```
    ${JOB_NAME} \
    ${CONFIG_FILE} \
    ${CHECKPOINT_FILE} \
    --cfg-options \
    test_dataloader.dataset.ann_file=annotations/panoptic_image_info_test-dev2017.
↪json \
    test_dataloader.dataset.data_prefix.img=test2017 \
    test_dataloader.dataset.data_prefix._delete_=True \
    test_evaluator.format_only=True \
    test_evaluator.ann_file=data/coco/annotations/panoptic_image_info_test-dev2017.
↪json \
    test_evaluator.outfile_prefix=${WORK_DIR}/results
```

例子: 假设我们使用预先训练的带有 ResNet-50 骨干网的 MaskFormer 对 `test2017` 执行推断。

```
# 单 gpu 测试
CUDA_VISIBLE_DEVICES=0 python tools/test.py \
    configs/maskformer/maskformer_r50_mstrain_16x1_75e_coco.py \
    checkpoints/maskformer_r50_mstrain_16x1_75e_coco_20220221_141956-bc2699cb.pth \
    --cfg-options \
    test_dataloader.dataset.ann_file=annotations/panoptic_image_info_test-dev2017.
↪json \
    test_dataloader.dataset.data_prefix.img=test2017 \
    test_dataloader.dataset.data_prefix._delete_=True \
    test_evaluator.format_only=True \
    test_evaluator.ann_file=data/coco/annotations/panoptic_image_info_test-dev2017.
↪json \
    test_evaluator.outfile_prefix=work_dirs/maskformer/results
```

重命名文件并压缩结果

推理之后，全景分割结果 (一个 json 文件和一个存储掩码的目录) 将在 `WORK_DIR` 中。我们应该按照 COCO's Website上的命名约定重新命名它们。最后，我们需要将 json 和存储掩码的目录压缩到 zip 文件中，并根据命名约定重命名该 zip 文件。注意，zip 文件应该**直接**包含上述两个文件。

重命名文件和压缩结果的命令:

```
# 在 WORK_DIR 中，我们有 panoptic 分割结果: 'panoptic' 和 'results. panoptical .json'。
cd ${WORK_DIR}
# 将 '[algorithm_name]' 替换为您使用的算法名称
mv ./panoptic ./panoptic_test-dev2017_[algorithm_name]_results
mv ./results.panoptic.json ./panoptic_test-dev2017_[algorithm_name]_results.json
zip panoptic_test-dev2017_[algorithm_name]_results.zip -ur panoptic_test-dev2017_
↪[algorithm_name]_results panoptic_test-dev2017_[algorithm_name]_results.json
```

(下页继续)

(续上页)

3.10 权重初始化

在训练过程中，适当的初始化策略有利于加快训练速度或获得更高的性能。MMCV 提供了一些常用的初始化模块的方法，如 `nn.Conv2d`。MMdetection 中的模型初始化主要使用 `init_cfg`。用🄵可以通过以下两个步骤来初始化模型：

1. 在 `model_cfg` 中为模型或其组件定义 `init_cfg`，但子组件的 `init_cfg` 优先级更高，会覆盖父模块的 `init_cfg` 。
2. 像往常一样构建模型，然后显式调用 `model.init_weights()` 方法，此时模型参数将会被按照配置文件写法进行初始化。

MMdetection 初始化工作流的高层 API 调用流程是：

model_cfg(init_cfg) -> build_from_cfg -> model -> init_weight() -> initialize(self, self.init_cfg) -> children's init_weight()

3.10.1 描述

它的数据类型是 dict 或者 list[dict]，包含了下列键值:

- `type` (str)，包含 `INTIALIZERS` 中的初始化器名称，后面跟着初始化器的参数。
- `layer`（str 或 list[str]），包含 Pytorch 或 MMCV 中基本层的名称，以及将被初始化的可学习参数，例如 `'Conv2d'`, `'DeformConv2d'`。
- `override` (dict 或 list[dict])，包含不继承自 `BaseModule` 且其初始化配置与 `layer` 键中的其他层不同的子模块。`type` 中定义的初始化器将适用于 `layer` 中定义的所有层，因此如果子模块不是 `BaseModule` 的派生类但可以与 `layer` 中的层相同的方式初始化，则不需要使用 `override`。`override` 包含了：
  - `type` 后跟初始化器的参数；
  - `name` 用以指示将被初始化的子模块。

3.10.2 初始化参数

从 mmcv.runner.BaseModule 或 mmdet.models 继承一个新模型。这里我们用 FooModel 来举个例子。

```
import torch.nn as nn
from mmcv.runner import BaseModule

class FooModel(BaseModule)
        def __init__(self,
                     arg1,
                     arg2,
                     init_cfg=None):
                super(FooModel, self).__init__(init_cfg)
                    ...
```

- 直接在代码中使用 init_cfg 初始化模型

```
import torch.nn as nn
from mmcv.runner import BaseModule
# or directly inherit mmdet models

class FooModel(BaseModule)
        def __init__(self,
                     arg1,
                     arg2,
                     init_cfg=XXX):
                       super(FooModel, self).__init__(init_cfg)
                    ...
```

- 在 mmcv.Sequential 或 mmcv.ModuleList 代码中直接使用 init_cfg 初始化模型

```
from mmcv.runner import BaseModule, ModuleList

class FooModel(BaseModule)
        def __init__(self,
                           arg1,
                           arg2,
                           init_cfg=None):
                       super(FooModel, self).__init__(init_cfg)
                    ...
                    self.conv1 = ModuleList(init_cfg=XXX)
```

- 使用配置文件中的 init_cfg 初始化模型

```
model = dict(
        ...
          model = dict(
              type='FooModel',
              arg1=XXX,
              arg2=XXX,
              init_cfg=XXX),
          ...
```

3.10.3 init_cfg 的使用

1. 用 layer 键初始化模型

   如果我们只定义了 layer, 它只会在 layer 键中初始化网络层。

   注意: layer 键对应的值是 Pytorch 的带有 weights 和 bias 属性的类名（因此不支持 MultiheadAttention 层）。

- 定义用于初始化具有相同配置的模块的 layer 键。

```
init_cfg = dict(type='Constant', layer=['Conv1d', 'Conv2d', 'Linear'], val=1)
# 用相同的配置初始化整个模块
```

- 定义用于初始化具有不同配置的层的 layer 键。

```
init_cfg = [dict(type='Constant', layer='Conv1d', val=1),
            dict(type='Constant', layer='Conv2d', val=2),
            dict(type='Constant', layer='Linear', val=3)]
# nn.Conv1d 将被初始化为 dict(type='Constant', val=1)
# nn.Conv2d 将被初始化为 dict(type='Constant', val=2)
# nn.Linear 将被初始化为 dict(type='Constant', val=3)
```

2. 使用 override 键初始化模型

- 当使用属性名初始化某些特定部分时，我们可以使用 override 键，override 中的值将忽略 init_cfg 中的值。

```
# layers:
# self.feat = nn.Conv1d(3, 1, 3)
# self.reg = nn.Conv2d(3, 3, 3)
# self.cls = nn.Linear(1,2)

init_cfg = dict(type='Constant',
                layer=['Conv1d','Conv2d'], val=1, bias=2,
                override=dict(type='Constant', name='reg', val=3, bias=4))
```

(下页继续)

(续上页)

```
# self.feat and self.cls 将被初始化为 dict(type='Constant', val=1, bias=2)
# 叫 'reg' 的模块将被初始化为 dict(type='Constant', val=3, bias=4)
```

- 如果 init_cfg 中的 `layer` 为 None，则只会初始化 override 中有 name 的子模块，而 override 中的 type 和其他参数可以省略。

```
# layers:
# self.feat = nn.Conv1d(3, 1, 3)
# self.reg = nn.Conv2d(3, 3, 3)
# self.cls = nn.Linear(1,2)

init_cfg = dict(type='Constant', val=1, bias=2,          override=dict(name='reg'))

# self.feat and self.cls 将被 Pytorch 初始化
# 叫 'reg' 的模块将被 dict(type='Constant', val=1, bias=2) 初始化
```

- 如果我们不定义 `layer` 或 `override` 键，它不会初始化任何东西。
- 无效的使用

```
# override 没有 name 键的话是无效的
init_cfg = dict(type='Constant', layer=['Conv1d','Conv2d'], val=1, bias=2,
                    override=dict(type='Constant', val=3, bias=4))

# override 有 name 键和其他参数但是没有 type 键也是无效的
init_cfg = dict(type='Constant', layer=['Conv1d','Conv2d'], val=1, bias=2,
                override=dict(name='reg', val=3, bias=4))
```

3. 使用预训练模型初始化模型

```
init_cfg = dict(type='Pretrained',
            checkpoint='torchvision://resnet50')
```

更多细节可以参考 MMEngine 的文档

3.11 将单阶段检测器作为 RPN

候选区域网络 (Region Proposal Network, RPN) 作为 Faster R-CNN 的一个子模块，将为 Faster R-CNN 的第二阶段产生候选区域。在 MMDetection 里大多数的二阶段检测器使用 `RPNHead` 作为候选区域网络来产生候选区域。然而，任何的单阶段检测器都可以作为候选区域网络，是因为他们对边界框的预测可以被视为是一种候选区域，并且因此能够在 R-CNN 中得到改进。因此在 MMDetection v3.0 中会支持将单阶段检测器作为 RPN 使用。

接下来我们通过一个例子，即如何在 Faster R-CNN 中使用一个无锚框的单阶段的检测器模型 FCOS 作为 RPN，详细阐述具体的全部流程。

主要流程如下:

1. 在 Faster R-CNN 中使用 `FCOSHead` 作为 `RPNHead`
2. 评估候选区域
3. 用预先训练的 FCOS 训练定制的 Faster R-CNN

3.11.1 在 Faster R-CNN 中使用 `FCOSHead` 作为 `RPNHead`

为了在 Faster R-CNN 中使用 `FCOSHead` 作为 `RPNHead`，我们应该创建一个名为 `configs/faster_rcnn/faster-rcnn_r50_fpn_fcos-rpn_1x_coco.py` 的配置文件，并且在 `configs/faster_rcnn/faster-rcnn_r50_fpn_fcos-rpn_1x_coco.py` 中将 `rpn_head` 的设置替换为 `bbox_head` 的设置，此外我们仍然使用 FCOS 的瓶颈设置，步幅为 `[8,16,32,64,128]`，并且更新 `bbox_roi_extractor` 的 `featmap_stride` 为 `[8,16,32,64,128]`。为了避免损失变慢，我们在前 1000 次迭代而不是前 500 次迭代中应用预热，这意味着 lr 增长得更慢。相关配置如下:

```
_base_ = [
    '../_base_/models/faster-rcnn_r50_fpn.py',
    '../_base_/datasets/coco_detection.py',
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]
model = dict(
    # 从 configs/fcos/fcos_r50-caffe_fpn_gn-head_1x_coco.py 复制
    neck=dict(
        start_level=1,
        add_extra_convs='on_output',  # 使用 P5
        relu_before_extra_convs=True),
    rpn_head=dict(
        _delete_=True,  # 忽略未使用的旧设置
        type='FCOSHead',
        num_classes=1,  # 对于 rpn, num_classes = 1, 如果 num_classes > 1, 它将在␣
↪TwoStageDetector 中自动设置为 1
        in_channels=256,
        stacked_convs=4,
        feat_channels=256,
        strides=[8, 16, 32, 64, 128],
        loss_cls=dict(
            type='FocalLoss',
            use_sigmoid=True,
            gamma=2.0,
            alpha=0.25,
```

(下页继续)

(续上页)

```
            loss_weight=1.0),
        loss_bbox=dict(type='IoULoss', loss_weight=1.0),
        loss_centerness=dict(
            type='CrossEntropyLoss', use_sigmoid=True, loss_weight=1.0)),
    roi_head=dict(  # featmap_strides 的更新取决于于颈部的步伐
        bbox_roi_extractor=dict(featmap_strides=[8, 16, 32, 64, 128])))
# 学习率
param_scheduler = [
    dict(
        type='LinearLR', start_factor=0.001, by_epoch=False, begin=0,
        end=1000),  # 慢慢增加 lr，否则损失变成 NAN
    dict(
        type='MultiStepLR',
        begin=0,
        end=12,
        by_epoch=True,
        milestones=[8, 11],
        gamma=0.1)
]
```

然后，我们可以使用下面的命令来训练我们的定制模型。更多训练命令，请参考这里。

```
# 使用 8 个 GPU 进行训练
bash
tools/dist_train.sh
configs/faster_rcnn/faster-rcnn_r50_fpn_fcos-rpn_1x_coco.py
--work-dir  /work_dirs/faster-rcnn_r50_fpn_fcos-rpn_1x_coco
```

3.11.2 评估候选区域

候选区域的质量对检测器的性能有重要影响，因此，我们也提供了一种评估候选区域的方法。和上面一样创建一个新的名为 configs/rpn/fcos-rpn_r50_fpn_1x_coco.py 的配置文件，并且在 configs/rpn/fcos-rpn_r50_fpn_1x_coco.py 中将 rpn_head 的设置替换为 bbox_head 的设置。

```
_base_ = [
    '../_base_/models/rpn_r50_fpn.py', '../_base_/datasets/coco_detection.py',
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]
val_evaluator = dict(metric='proposal_fast')
test_evaluator = val_evaluator
model = dict(
    # 从 configs/fcos/fcos_r50-caffe_fpn_gn-head_1x_coco.py 复制
```

(下页继续)

(续上页)

```
    neck=dict(
        start_level=1,
        add_extra_convs='on_output',  # 使用 P5
        relu_before_extra_convs=True),
    rpn_head=dict(
        _delete_=True,  # 忽略未使用的旧设置
        type='FCOSHead',
        num_classes=1,  # 对于 rpn, num_classes = 1, 如果 num_classes > 为 1, 它将在 rpn
↪中自动设置为 1
        in_channels=256,
        stacked_convs=4,
        feat_channels=256,
        strides=[8, 16, 32, 64, 128],
        loss_cls=dict(
            type='FocalLoss',
            use_sigmoid=True,
            gamma=2.0,
            alpha=0.25,
            loss_weight=1.0),
        loss_bbox=dict(type='IoULoss', loss_weight=1.0),
        loss_centerness=dict(
            type='CrossEntropyLoss', use_sigmoid=True, loss_weight=1.0)))
```

假设我们在训练之后有检查点 `./work_dirs/faster-rcnn_r50_fpn_fcos-rpn_1x_coco/epoch_12.pth`，然后，我们可以使用下面的命令来评估建议的质量。

```
# 使用 8 个 GPU 进行测试
bash
tools/dist_test.sh
configs/rpn/fcos-rpn_r50_fpn_1x_coco.py
--work_dirs /faster-rcnn_r50_fpn_fcos-rpn_1x_coco/epoch_12.pth
```

3.11.3 用预先训练的 FCOS 训练定制的 Faster R-CNN

预训练不仅加快了训练的收敛速度，而且提高了检测器的性能。因此，我们在这里给出一个例子来说明如何使用预先训练的 FCOS 作为 RPN 来加速训练和提高精度。假设我们想在 Faster R-CNN 中使用 `FCOSHead` 作为 `rpn_head`，并加载预先训练权重来进行训练 `fcos_r50-caffe_fpn_gn-head_1x_coco`。配置文件 `configs/faster_rcnn/faster-rcnn_r50-caffe_fpn_fcos- rpn_1x_copy .py` 的内容如下所示。注意，`fcos_r50-caffe_fpn_gn-head_1x_coco` 使用 ResNet50 的 caffe 版本，因此需要更新 `data_preprocessor` 中的像素平均值和 std。

```
_base_ = [
    '../_base_/models/faster-rcnn_r50_fpn.py',
    '../_base_/datasets/coco_detection.py',
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]
model = dict(
    data_preprocessor=dict(
        mean=[103.530, 116.280, 123.675],
        std=[1.0, 1.0, 1.0],
        bgr_to_rgb=False),
    backbone=dict(
        norm_cfg=dict(type='BN', requires_grad=False),
        style='caffe',
        init_cfg=None),  # the checkpoint in ``load_from`` contains the weights of␣
↪backbone
    neck=dict(
        start_level=1,
        add_extra_convs='on_output',  # 使用 P5
        relu_before_extra_convs=True),
    rpn_head=dict(
        _delete_=True,  # 忽略未使用的旧设置
        type='FCOSHead',
        num_classes=1,  # 对于 rpn, num_classes = 1, 如果 num_classes > 1, 它将在␣
↪TwoStageDetector 中自动设置为 1
        in_channels=256,
        stacked_convs=4,
        feat_channels=256,
        strides=[8, 16, 32, 64, 128],
        loss_cls=dict(
            type='FocalLoss',
            use_sigmoid=True,
            gamma=2.0,
            alpha=0.25,
            loss_weight=1.0),
        loss_bbox=dict(type='IoULoss', loss_weight=1.0),
        loss_centerness=dict(
            type='CrossEntropyLoss', use_sigmoid=True, loss_weight=1.0)),
    roi_head=dict(  # update featmap_strides due to the strides in neck
        bbox_roi_extractor=dict(featmap_strides=[8, 16, 32, 64, 128])))
load_from = 'https://download.openmmlab.com/mmdetection/v2.0/fcos/fcos_r50_caffe_fpn_
↪gn-head_1x_coco/fcos_r50_caffe_fpn_gn-head_1x_coco-821213aa.pth'
```

训练命令如下。

```
bash
tools/dist_train.sh
configs/faster_rcnn/faster-rcnn_r50-caffe_fpn_fcos-rpn_1x_coco.py  \
--work-dir /work_dirs/faster-rcnn_r50-caffe_fpn_fcos-rpn_1x_coco
```

3.12 半监督目标检测

半监督目标检测同时利用标签数据和无标签数据进行训练，一方面可以减少模型对检测框数量的依赖，另一方面也可以利用大量的未标记数据进一步提高模型。

按照以下流程进行半监督目标检测：

- 半监督目标检测
  - 准备和拆分数据集
  - 配置多分支数据流程
  - 配置半监督数据加载
  - 配置半监督模型
  - 配置 *MeanTeacherHook*
  - 配置 *TeacherStudentValLoop*

3.12.1 准备和拆分数据集

我们提供了数据集下载脚本，默认下载 coco2017 数据集，并且自动解压。

```
python tools/misc/download_dataset.py
```

解压后的数据集目录如下：

```
mmdetection
├── data
│   ├── coco
│   │   ├── annotations
│   │   │   ├── image_info_unlabeled2017.json
│   │   │   ├── instances_train2017.json
│   │   │   ├── instances_val2017.json
│   │   ├── test2017
│   │   ├── train2017
│   │   ├── unlabeled2017
│   │   ├── val2017
```

半监督目标检测在 coco 数据集上有两种比较通用的实验设置：

（1）将 train2017 按照固定百分比（1%，2%，5% 和 10%）划分出一部分数据作为标签数据集，剩余的训练集数据作为无标签数据集，同时考虑划分不同的训练集数据作为标签数据集对半监督训练的结果影响较大，所以采用五折交叉验证来评估算法性能。我们提供了数据集划分脚本：

```
python tools/misc/split_coco.py
```

该脚本默认会按照 1%，2%，5% 和 10% 的标签数据占比划分 train2017，每一种划分会随机重复 5 次，用于交叉验证。生成的半监督标注文件名称格式如下：

- 标签数据集标注名称格式：instances_train2017.{fold}@{percent}.json
- 无标签数据集名称标注：instances_train2017.{fold}@{percent}-unlabeled.json

其中，fold 用于交叉验证，percent 表示标签数据的占比。划分后的数据集目录结构如下：

```
mmdetection
├── data
│   ├── coco
│   │   ├── annotations
│   │   │   ├── image_info_unlabeled2017.json
│   │   │   ├── instances_train2017.json
│   │   │   ├── instances_val2017.json
│   │   ├── semi_anns
│   │   │   ├── instances_train2017.1@1.json
│   │   │   ├── instances_train2017.1@1-unlabeled.json
│   │   │   ├── instances_train2017.1@2.json
│   │   │   ├── instances_train2017.1@2-unlabeled.json
│   │   │   ├── instances_train2017.1@5.json
│   │   │   ├── instances_train2017.1@5-unlabeled.json
│   │   │   ├── instances_train2017.1@10.json
│   │   │   ├── instances_train2017.1@10-unlabeled.json
│   │   │   ├── instances_train2017.2@1.json
│   │   │   ├── instances_train2017.2@1-unlabeled.json
│   │   ├── test2017
│   │   ├── train2017
│   │   ├── unlabeled2017
│   │   ├── val2017
```

（2）将 train2017 作为标签数据集，unlabeled2017 作为无标签数据集。由于 image_info_unlabeled2017.json 没有 categories 信息，无法初始化 CocoDataset，所以需要将 instances_train2017.json 的 categories 写入 image_info_unlabeled2017.json，另存为 instances_unlabeled2017.json，相关脚本如下：

```
from mmengine.fileio import load, dump

```

(下页继续)

(续上页)

```
anns_train = load('instances_train2017.json')
anns_unlabeled = load('image_info_unlabeled2017.json')
anns_unlabeled['categories'] = anns_train['categories']
dump(anns_unlabeled, 'instances_unlabeled2017.json')
```

处理后的数据集目录如下：

```
mmdetection
├── data
│   ├── coco
│   │   ├── annotations
│   │   │   ├── image_info_unlabeled2017.json
│   │   │   ├── instances_train2017.json
│   │   │   ├── instances_unlabeled2017.json
│   │   │   ├── instances_val2017.json
│   │   ├── test2017
│   │   ├── train2017
│   │   ├── unlabeled2017
│   │   ├── val2017
```

3.12.2 配置多分支数据流程

半监督学习有两个主要的方法，分别是 一致性正则化 和伪标签 。一致性正则化往往需要一些精心的设计，而伪标签的形式比较简单，更容易拓展到下游任务。我们主要采用了基于伪标签的教师学生联合训练的半监督目标检测框架，对于标签数据和无标签数据需要配置不同的数据流程：（1）标签数据的数据流程：

```
# pipeline used to augment labeled data,
# which will be sent to student model for supervised training.
sup_pipeline = [
    dict(type='LoadImageFromFile',backend_args = backend_args),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='RandomResize', scale=scale, keep_ratio=True),
    dict(type='RandomFlip', prob=0.5),
    dict(type='RandAugment', aug_space=color_space, aug_num=1),
    dict(type='FilterAnnotations', min_gt_bbox_wh=(1e-2, 1e-2)),
    dict(type='MultiBranch', sup=dict(type='PackDetInputs'))
]
```

（2）无标签的数据流程：

```
# pipeline used to augment unlabeled data weakly,
# which will be sent to teacher model for predicting pseudo instances.
weak_pipeline = [
```

(下页继续)

(续上页)

```
    dict(type='RandomResize', scale=scale, keep_ratio=True),
    dict(type='RandomFlip', prob=0.5),
    dict(
        type='PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'flip', 'flip_direction',
                   'homography_matrix')),
]

# pipeline used to augment unlabeled data strongly,
# which will be sent to student model for unsupervised training.
strong_pipeline = [
    dict(type='RandomResize', scale=scale, keep_ratio=True),
    dict(type='RandomFlip', prob=0.5),
    dict(
        type='RandomOrder',
        transforms=[
            dict(type='RandAugment', aug_space=color_space, aug_num=1),
            dict(type='RandAugment', aug_space=geometric, aug_num=1),
        ]),
    dict(type='RandomErasing', n_patches=(1, 5), ratio=(0, 0.2)),
    dict(type='FilterAnnotations', min_gt_bbox_wh=(1e-2, 1e-2)),
    dict(
        type='PackDetInputs',
        meta_keys=('img_id', 'img_path', 'ori_shape', 'img_shape',
                   'scale_factor', 'flip', 'flip_direction',
                   'homography_matrix')),
]

# pipeline used to augment unlabeled data into different views
unsup_pipeline = [
    dict(type='LoadImageFromFile', backend_args = backend_args),
    dict(type='LoadEmptyAnnotations'),
    dict(
        type='MultiBranch',
        unsup_teacher=weak_pipeline,
        unsup_student=strong_pipeline,
    )
]
```

3.12.3 配置半监督数据加载

（1）构建半监督数据集。使用 ConcatDataset 拼接标签数据集和无标签数据集。

```
labeled_dataset = dict(
    type=dataset_type,
    data_root=data_root,
    ann_file='annotations/instances_train2017.json',
    data_prefix=dict(img='train2017/'),
    filter_cfg=dict(filter_empty_gt=True, min_size=32),
    pipeline=sup_pipeline)

unlabeled_dataset = dict(
    type=dataset_type,
    data_root=data_root,
    ann_file='annotations/instances_unlabeled2017.json',
    data_prefix=dict(img='unlabeled2017/'),
    filter_cfg=dict(filter_empty_gt=False),
    pipeline=unsup_pipeline)

train_dataloader = dict(
    batch_size=batch_size,
    num_workers=num_workers,
    persistent_workers=True,
    sampler=dict(
        type='GroupMultiSourceSampler',
        batch_size=batch_size,
        source_ratio=[1, 4]),
    dataset=dict(
        type='ConcatDataset', datasets=[labeled_dataset, unlabeled_dataset]))
```

（2）使用多源数据集采样器。使用 GroupMultiSourceSampler 从 labeled_dataset 和 labeled_dataset 采样数据组成 batch，source_ratio 控制 batch 中标签数据和无标签数据的占比。GroupMultiSourceSampler 还保证了同一个 batch 中的图片具有相近的长宽比例，如果不需要保证 batch 内图片的长宽比例，可以使用 MultiSourceSampler。GroupMultiSourceSampler 采样示意图如下：

sup=1000 表示标签数据集的规模为 1000，sup_h=200 表示标签数据集中长宽比大于等于 1 的图片规模为 200，sup_w=800 表示标签数据集中长宽比小于 1 的图片规模为 800，unsup=9000 表示无标签数据集的规模为 9000，unsup_h=1800 表示无标签数据集中长宽比大于等于 1 的图片规模为 1800，unsup_w=7200 表示标签数据集中长宽比小于 1 的图片规模为 7200，GroupMultiSourceSampler 每次按照标签数据集和无标签数据集的图片的总体长宽比分布随机选择一组，然后按照 source_ratio 从两个数据集中采样组成 batch，因此标签数据集和无标签数据集重复采样次数不同。

3.12.4 配置半监督模型

我们选择 Faster R-CNN 作为 detector 进行半监督训练，以半监督目标检测算法 SoftTeacher 为例，模型的配置可以继承 _base_/models/faster-rcnn_r50_fpn.py，将检测器的骨干网络替换成 caffe 风格。注意，与监督训练的配置文件不同的是，Faster R-CNN 作为 detector，是作为 model 的一个属性，而不是 model 。此外，还需要将 data_preprocessor 设置为 MultiBranchDataPreprocessor，用于处理不同数据流程图片的填充和归一化。最后，可以通过 semi_train_cfg 和 semi_test_cfg 配置半监督训练和测试需要的参数。

```
_base_ = [
    '../_base_/models/faster-rcnn_r50_fpn.py', '../_base_/default_runtime.py',
    '../_base_/datasets/semi_coco_detection.py'
]

detector = _base_.model
detector.data_preprocessor = dict(
    type='DetDataPreprocessor',
    mean=[103.530, 116.280, 123.675],
    std=[1.0, 1.0, 1.0],
    bgr_to_rgb=False,
    pad_size_divisor=32)
detector.backbone = dict(
    type='ResNet',
    depth=50,
    num_stages=4,
    out_indices=(0, 1, 2, 3),
    frozen_stages=1,
    norm_cfg=dict(type='BN', requires_grad=False),
    norm_eval=True,
    style='caffe',
    init_cfg=dict(
        type='Pretrained',
        checkpoint='open-mmlab://detectron2/resnet50_caffe'))

model = dict(
    _delete_=True,
    type='SoftTeacher',
    detector=detector,
    data_preprocessor=dict(
        type='MultiBranchDataPreprocessor',
        data_preprocessor=detector.data_preprocessor),
    semi_train_cfg=dict(
        freeze_teacher=True,
        sup_weight=1.0,
```

(下页继续)

(续上页)

```
        unsup_weight=4.0,
        pseudo_label_initial_score_thr=0.5,
        rpn_pseudo_thr=0.9,
        cls_pseudo_thr=0.9,
        reg_pseudo_thr=0.02,
        jitter_times=10,
        jitter_scale=0.06,
        min_pseudo_bbox_wh=(1e-2, 1e-2)),
    semi_test_cfg=dict(predict_on='teacher'))
```

此外，我们也支持其他检测模型进行半监督训练，比如，RetinaNet 和 Cascade R-CNN。由于 SoftTeacher 仅支持 Faster R-CNN，所以需要将其替换为 SemiBaseDetector，示例如下：

```
_base_ = [
    '../_base_/models/retinanet_r50_fpn.py', '../_base_/default_runtime.py',
    '../_base_/datasets/semi_coco_detection.py'
]

detector = _base_.model

model = dict(
    _delete_=True,
    type='SemiBaseDetector',
    detector=detector,
    data_preprocessor=dict(
        type='MultiBranchDataPreprocessor',
        data_preprocessor=detector.data_preprocessor),
    semi_train_cfg=dict(
        freeze_teacher=True,
        sup_weight=1.0,
        unsup_weight=1.0,
        cls_pseudo_thr=0.9,
        min_pseudo_bbox_wh=(1e-2, 1e-2)),
    semi_test_cfg=dict(predict_on='teacher'))
```

沿用 SoftTeacher 的半监督训练配置，将 batch_size 改为 2 ，source_ratio 改为 [1, 1]，RetinaNet，Faster R-CNN，Cascade R-CNN 以及 SoftTeacher 在 10% coco 训练集上的监督训练和半监督训练的实验结果如下：

3.12.5 配置 MeanTeacherHook

通常，教师模型采用对学生模型指数滑动平均（EMA）的方式进行更新，进而教师模型随着学生模型的优化而优化，可以通过配置 `custom_hooks` 实现：

```
custom_hooks = [dict(type='MeanTeacherHook')]
```

3.12.6 配置 TeacherStudentValLoop

由于教师学生联合训练框架存在两个模型，我们可以用 `TeacherStudentValLoop` 替换 `ValLoop`，在训练的过程中同时检验两个模型的精度。

```
val_cfg = dict(type='TeacherStudentValLoop')
```

CHAPTER 4

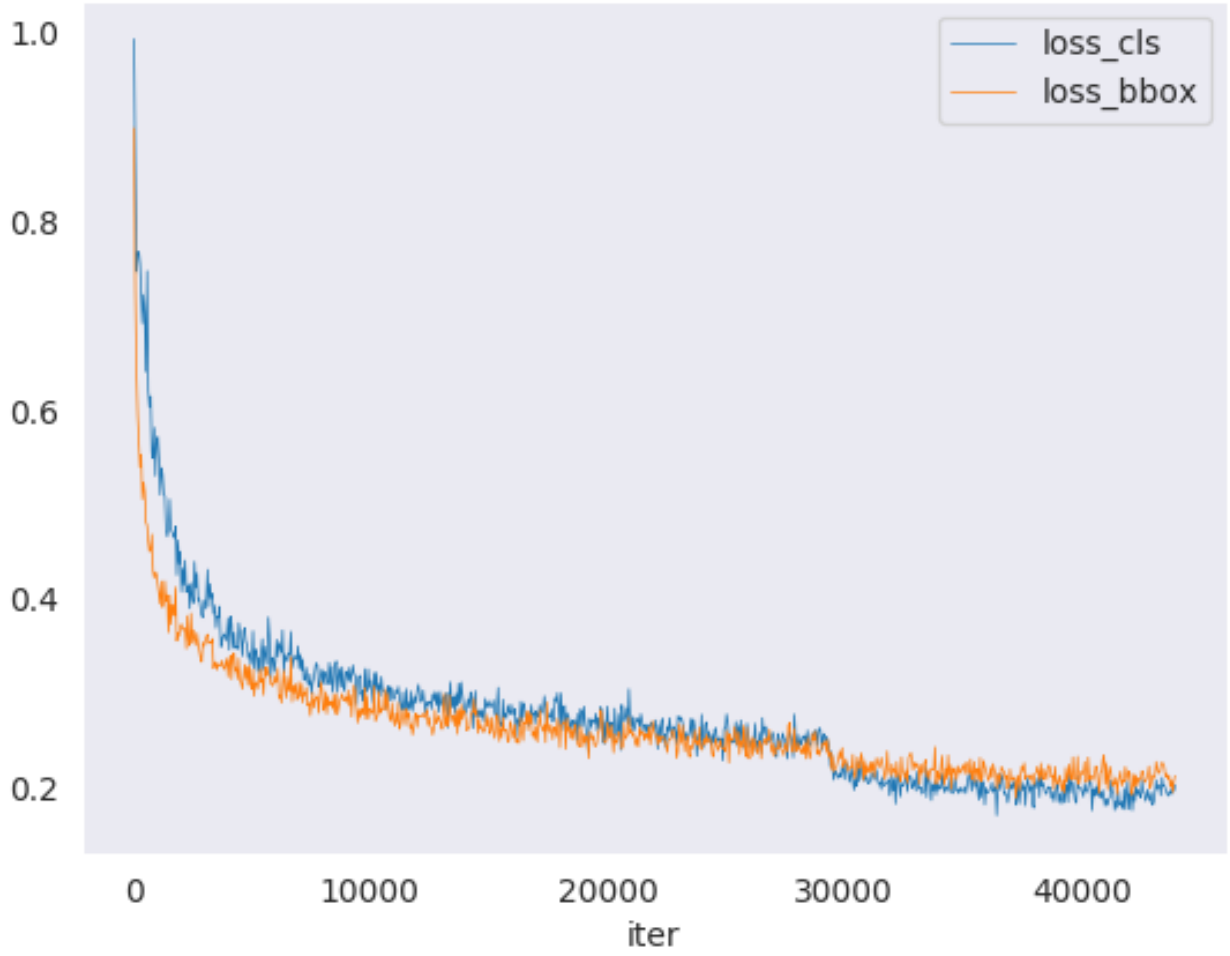
实用工具

除了训练和测试脚本，我们还在 tools/ 目录下提供了许多有用的工具。

4.1 日志分析

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.

```
python tools/analysis_tools/analyze_logs.py plot_curve [--keys ${KEYS}] [--eval-
↪interval ${EVALUATION_INTERVAL}] [--title ${TITLE}] [--legend ${LEGEND}] [--backend
↪${BACKEND}] [--style ${STYLE}] [--out ${OUT_FILE}]
```



样例:

- 绘制分类损失曲线图

```
python tools/analysis_tools/analyze_logs.py plot_curve 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

```
python tools/analysis_tools/analyze_logs.py plot_curve log.json --keys loss_cls␣
↪loss_bbox --out losses.pdf
```

- 在相同图像中比较两次运行结果的 bbox mAP

```
python tools/analysis_tools/analyze_logs.py plot_curve log1.json log2.json --keys␣
↪bbox_mAP --legend run1 run2
```

- 计算平均训练速度

```
python tools/analysis_tools/analyze_logs.py cal_train_time log.json [--include-
↪outliers]
```

输出以如下形式展示

```
-----Analyze train time of work_dirs/some_exp/20190611_192040.log.json-----
slowest epoch 11, average time is 1.2024
fastest epoch 1, average time is 1.1909
time std over epochs is 0.0028
average iter time: 1.1959 s/iter
```

4.2 结果分析

使用 `tools/analysis_tools/analyze_results.py` 可计算每个图像 mAP，随后根据真实标注框与预测框的比较结果，展示或保存最高与最低 top-k 得分的预测图像。

使用方法

```
python tools/analysis_tools/analyze_results.py \
      ${CONFIG} \
      ${PREDICTION_PATH} \
      ${SHOW_DIR} \
      [--show] \
      [--wait-time ${WAIT_TIME}] \
      [--topk ${TOPK}] \
      [--show-score-thr ${SHOW_SCORE_THR}] \
      [--cfg-options ${CFG_OPTIONS}]
```

各个参数选项的作用:

- `config`: model config 文件的路径。
- `prediction_path`: 使用 `tools/test.py` 输出的 pickle 格式结果文件。
- `show_dir`: 绘制真实标注框与预测框的图像存放目录。
- `--show`：决定是否展示绘制 box 后的图片，默认值为 `False`。
- `--wait-time`: show 时间的间隔，若为 0 表示持续显示。
- `--topk`: 根据最高或最低 `topk` 概率排序保存的图片数量，若不指定，默认设置为 `20`。
- `--show-score-thr`: 能够展示的概率阈值，默认为 `0`。
- `--cfg-options`: 如果指定，可根据指定键值对覆盖更新配置文件的对应选项

样例: 假设你已经通过 `tools/test.py` 得到了 pickle 格式的结果文件，路径为‘./result.pkl’。

1. 测试 Faster R-CNN 并可视化结果，保存图片至 `results/`

```
python tools/analysis_tools/analyze_results.py \
       configs/faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py \
       result.pkl \
       results \
       --show
```

2. 测试 Faster R-CNN 并指定 top-k 参数为 50，保存结果图片至 `results/`

```
python tools/analysis_tools/analyze_results.py \
       configs/faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py \
       result.pkl \
       results \
       --topk 50
```

3. 如果你想过滤低概率的预测结果，指定 `show-score-thr` 参数

```
python tools/analysis_tools/analyze_results.py \
       configs/faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py \
       result.pkl \
       results \
       --show-score-thr 0.3
```

4.3 可视化

4.3.1 可视化数据集

`tools/analysis_tools/browse_dataset.py` 可帮助使用者检查所使用的检测数据集（包括图像和标注），或保存图像至指定目录。

```
python tools/misc/browse_dataset.py ${CONFIG} [-h] [--skip-type ${SKIP_TYPE[SKIP_TYPE.
↪..]}] [--output-dir ${OUTPUT_DIR}] [--not-show] [--show-interval ${SHOW_INTERVAL}]
```

4.3.2 可视化模型

在可视化之前，需要先转换模型至 ONNX 格式，可参考此处。注意，现在只支持 RetinaNet，之后的版本将会支持其他模型转换后的模型可以被其他工具可视化Netron。

4.3.3 可视化预测结果

如果你想要一个轻量 GUI 可视化检测结果，你可以参考 DetVisGUI project。

4.4 误差分析

`tools/analysis_tools/coco_error_analysis.py` 使用不同标准分析每个类别的 COCO 评估结果。同时将一些有帮助的信息体现在图表上。

```
python tools/analysis_tools/coco_error_analysis.py ${RESULT} ${OUT_DIR} [-h] [--ann ${ANN}] [--types ${TYPES[TYPES...]}]
```

样例:

假设你已经把 Mask R-CNN checkpoint file 放置在文件夹‘checkpoint’中（其他模型请在 model zoo 中获取）。

为了保存 bbox 结果信息，我们需要用下列方式修改 `test_evaluator`：

1. 查找当前 config 文件相对应的‘configs/base/datasets’数据集信息。
2. 用当前数据集 config 中的 test_evaluator 以及 test_dataloader 替换原始文件的 test_evaluator 以及 test_dataloader。
3. 使用以下命令得到 bbox 或 segmentation 的 json 格式文件。

```
python tools/test.py \
       configs/mask_rcnn/mask-rcnn_r50_fpn_1x_coco.py \
       checkpoint/mask_rcnn_r50_fpn_1x_coco_20200205-d4b0c5d6.pth \
```

1. 得到每一类的 COCO bbox 误差结果，并保存分析结果图像至指定目录。（在 config 中默认目录是‘./work_dirs/coco_instance/test’）

```
python tools/analysis_tools/coco_error_analysis.py \
       results.bbox.json \
       results \
       --ann=data/coco/annotations/instances_val2017.json \
```

2. 得到每一类的 COCO 分割误差结果，并保存分析结果图像至指定目录。

```
python tools/analysis_tools/coco_error_analysis.py \
       results.segm.json \
       results \
       --ann=data/coco/annotations/instances_val2017.json \
       --types='segm'
```

4.5 模型服务部署

如果你想使用 TorchServe 搭建一个 MMDetection 模型服务，可以参考以下步骤：

4.5.1 1. 安装 TorchServe

假设你已经成功安装了包含 PyTorch 和 MMDetection 的 Python 环境，那么你可以运行以下命令来安装 TorchServe 及其依赖项。有关更多其他安装选项，请参考快速入门。

```
python -m pip install torchserve torch-model-archiver torch-workflow-archiver nvgpu
```

注意: 如果你想在 docker 中使用 TorchServe，请参考torchserve docker。

4.5.2 2. 把 MMDetection 模型转换至 TorchServe

```
python tools/deployment/mmdet2torchserve.py ${CONFIG_FILE} ${CHECKPOINT_FILE} \
--output-folder ${MODEL_STORE} \
--model-name ${MODEL_NAME}
```

4.5.3 3. 启动 TorchServe

```
torchserve --start --ncs \
  --model-store ${MODEL_STORE} \
  --models  ${MODEL_NAME}.mar
```

4.5.4 4. 测试部署效果

```
curl -O curl -O https://raw.githubusercontent.com/pytorch/serve/master/docs/images/
↪3dogs.jpg
curl http://127.0.0.1:8080/predictions/${MODEL_NAME} -T 3dogs.jpg
```

你可以得到下列 json 信息：

```
[
  {
    "class_label": 16,
    "class_name": "dog",
    "bbox": [
      294.63409423828125,
```

(下页继续)

(续上页)


```
      203.99111938476562,
      417.048583984375,
      281.62744140625
    ],
    "score": 0.9987992644309998
  },
  {
    "class_label": 16,
    "class_name": "dog",
    "bbox": [
      404.26019287109375,
      126.0080795288086,
      574.5091552734375,
      293.6662292480469
    ],
    "score": 0.9979367256164551
  },
  {
    "class_label": 16,
    "class_name": "dog",
    "bbox": [
      197.2144775390625,
      93.3067855834961,
      307.8505554199219,
      276.7560119628906
    ],
    "score": 0.993338406085968
  }
]
```


结果对比

你也可以使用 test_torchserver.py 来比较 TorchServe 和 PyTorch 的结果，并可视化：

```
python tools/deployment/test_torchserver.py ${IMAGE_FILE} ${CONFIG_FILE} ${CHECKPOINT_
↪FILE} ${MODEL_NAME}
[--inference-addr ${INFERENCE_ADDR}] [--device ${DEVICE}] [--score-thr ${SCORE_THR}]␣
↪[--work-dir ${WORK_DIR}]
```

样例：

```
python tools/deployment/test_torchserver.py \
demo/demo.jpg \
```

(下页继续)

(续上页)

```
configs/yolo/yolov3_d53_8xb8-320-273e_coco.py \
checkpoint/yolov3_d53_320_273e_coco-421362b6.pth \
yolov3 \
--work-dir ./work-dir
```

4.5.5 5. 停止 TorchServe

```
torchserve --stop
```

4.6 模型复杂度

tools/analysis_tools/get_flops.py 工具可用于计算指定模型的 FLOPs、参数量大小（改编自 flops-counter.pytorch ）。

```
python tools/analysis_tools/get_flops.py ${CONFIG_FILE} [--shape ${INPUT_SHAPE}]
```

获得的结果如下：

```
==============================
Input shape: (3, 1280, 800)
Flops: 239.32 GFLOPs
Params: 37.74 M
==============================
```

注意：这个工具还只是实验性质，我们不保证这个数值是绝对正确的。你可以将他用于简单的比较，但如果用于科技论文报告需要再三检查确认。

1. FLOPs 与输入的形状大小相关，参数量没有这个关系，默认的输入形状大小为 (1, 3, 1280, 800) 。
2. 一些算子并不计入 FLOPs，比如 GN 或其他自定义的算子。你可以参考 mmcv.cnn.get_model_complexity_info() 查看更详细的说明。
3. 两阶段检测的 FLOPs 大小取决于 proposal 的数量。

4.7 模型转换

4.7.1 MMDetection 模型转换至 ONNX 格式

我们提供了一个脚本用于转换模型至 ONNX 格式。同时还支持比较 Pytorch 与 ONNX 模型的输出结果以便对照。更详细的内容可以参考 mmdeploy。

4.7.2 MMDetection 1.x 模型转换至 MMDetection 2.x 模型

`tools/model_converters/upgrade_model_version.py` 可将旧版本的 MMDetection checkpoints 转换至新版本。但要注意此脚本不保证在新版本加入非兼容更新后还能正常转换，建议您直接使用新版本的 checkpoints。

```
python tools/model_converters/upgrade_model_version.py ${IN_FILE} ${OUT_FILE} [-h] [--
↪num-classes NUM_CLASSES]
```

4.7.3 RegNet 模型转换至 MMDetection 模型

`tools/model_converters/regnet2mmdet.py` 将 pycls 编码的预训练 RegNet 模型转换为 MMDetection 风格。

```
python tools/model_converters/regnet2mmdet.py ${SRC} ${DST} [-h]
```

4.7.4 Detectron ResNet 模型转换至 Pytorch 模型

`tools/model_converters/detectron2pytorch.py` 将 detectron 的原始预训练 RegNet 模型转换为 MMDetection 风格。

```
python tools/model_converters/detectron2pytorch.py ${SRC} ${DST} ${DEPTH} [-h]
```

4.7.5 制作发布用模型

`tools/model_converters/publish_model.py` 可用来制作一个发布用的模型。

在发布模型至 AWS 之前，你可能需要：

1. 将模型转换至 CPU 张量
2. 删除优化器状态
3. 计算 checkpoint 文件的 hash 值，并将 hash 号码记录至文件名。

```
python tools/model_converters/publish_model.py ${INPUT_FILENAME} ${OUTPUT_FILENAME}
```

样例：

```
python tools/model_converters/publish_model.py work_dirs/faster_rcnn/latest.pth␣
↪faster_rcnn_r50_fpn_1x_20190801.pth
```

最后输出的文件名如下所示：`faster_rcnn_r50_fpn_1x_20190801-{hash id}.pth`.

4.8 数据集转换

`tools/data_converters/` 提供了将 Cityscapes 数据集与 Pascal VOC 数据集转换至 COCO 数据集格式的工具

```
python tools/dataset_converters/cityscapes.py ${CITYSCAPES_PATH} [-h] [--img-dir $
↪{IMG_DIR}] [--gt-dir ${GT_DIR}] [-o ${OUT_DIR}] [--nproc ${NPROC}]
python tools/dataset_converters/pascal_voc.py ${DEVKIT_PATH} [-h] [-o ${OUT_DIR}]
```

4.9 数据集下载

`tools/misc/download_dataset.py` 可以下载各类形如 COCO，VOC，LVIS 数据集。

```
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
```

对于中国境内的用户，我们也推荐使用开源数据平台 OpenDataLab 来获取这些数据集，以获得更好的下载体验:

- COCO2017
- VOC2007
- VOC2012
- LVIS

4.10 基准测试

4.10.1 鲁棒性测试基准

tools/analysis_tools/test_robustness.py 及 tools/analysis_tools/robustness_eval.py 帮助使用者衡量模型的鲁棒性。其核心思想来源于 Benchmarking Robustness in Object Detection: Autonomous Driving when Winter is Coming。如果你想了解如何在污损图像上评估模型的效果，以及参考该基准的一组标准模型，请参照*robustness_benchmarking.md*。

4.10.2 FPS 测试基准

tools/analysis_tools/benchmark.py 可帮助使用者计算 FPS，FPS 计算包括了模型向前传播与后处理过程。为了得到更精确的计算值，现在的分布式计算模式只支持一个 GPU。

```
python -m torch.distributed.launch --nproc_per_node=1 --master_port=${PORT} tools/
↪analysis_tools/benchmark.py \
    ${CONFIG} \
    [--checkpoint ${CHECKPOINT}] \
    [--repeat-num ${REPEAT_NUM}] \
    [--max-iter ${MAX_ITER}] \
    [--log-interval ${LOG_INTERVAL}] \
    --launcher pytorch
```

样例：假设你已经下载了 Faster R-CNN 模型 checkpoint 并放置在 checkpoints/ 目录下。

```
python -m torch.distributed.launch --nproc_per_node=1 --master_port=29500 tools/
↪analysis_tools/benchmark.py \
       configs/faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py \
       checkpoints/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth \
       --launcher pytorch
```

4.11 更多工具

4.11.1 以某个评估标准进行评估

tools/analysis_tools/eval_metric.py 根据配置文件中的评估方式对 pkl 结果文件进行评估。

```
python tools/analysis_tools/eval_metric.py ${CONFIG} ${PKL_RESULTS} [-h] [--format-
↪only] [--eval ${EVAL[EVAL ...]}]
                      [--cfg-options ${CFG_OPTIONS [CFG_OPTIONS ...]}]
                      [--eval-options ${EVAL_OPTIONS [EVAL_OPTIONS ...]}]
```

4.11.2 打印全部 config

tools/misc/print_config.py 可将所有配置继承关系展开，完全打印相应的配置文件。

```
python tools/misc/print_config.py ${CONFIG} [-h] [--options ${OPTIONS [OPTIONS...]}]
```

4.12 超参数优化

4.12.1 YOLO Anchor 优化

tools/analysis_tools/optimize_anchors.py 提供了两种方法优化 YOLO 的 anchors。

其中一种方法使用 K 均值 anchor 聚类（k-means anchor cluster），源自 darknet。

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

另一种方法使用差分进化算法优化 anchors。

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

样例：

```
python tools/analysis_tools/optimize_anchors.py configs/yolo/yolov3_d53_8xb8-320-273e_
↪coco.py --algorithm differential_evolution --input-shape 608 608 --device cuda --
↪output-dir work_dirs
```

你可能会看到如下结果：

```
loading annotations into memory...
Done (t=9.70s)
creating index...
index created!
2021-07-19 19:37:20,951 - mmdet - INFO - Collecting bboxes from annotation...
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>] 117266/117266, 15874.5 task/s,␣
↪elapsed: 7s, ETA:     0s

2021-07-19 19:37:28,753 - mmdet - INFO - Collected 849902 bboxes.
differential_evolution step 1: f(x)= 0.506055
differential_evolution step 2: f(x)= 0.506055
......
```

(下页继续)

(续上页)

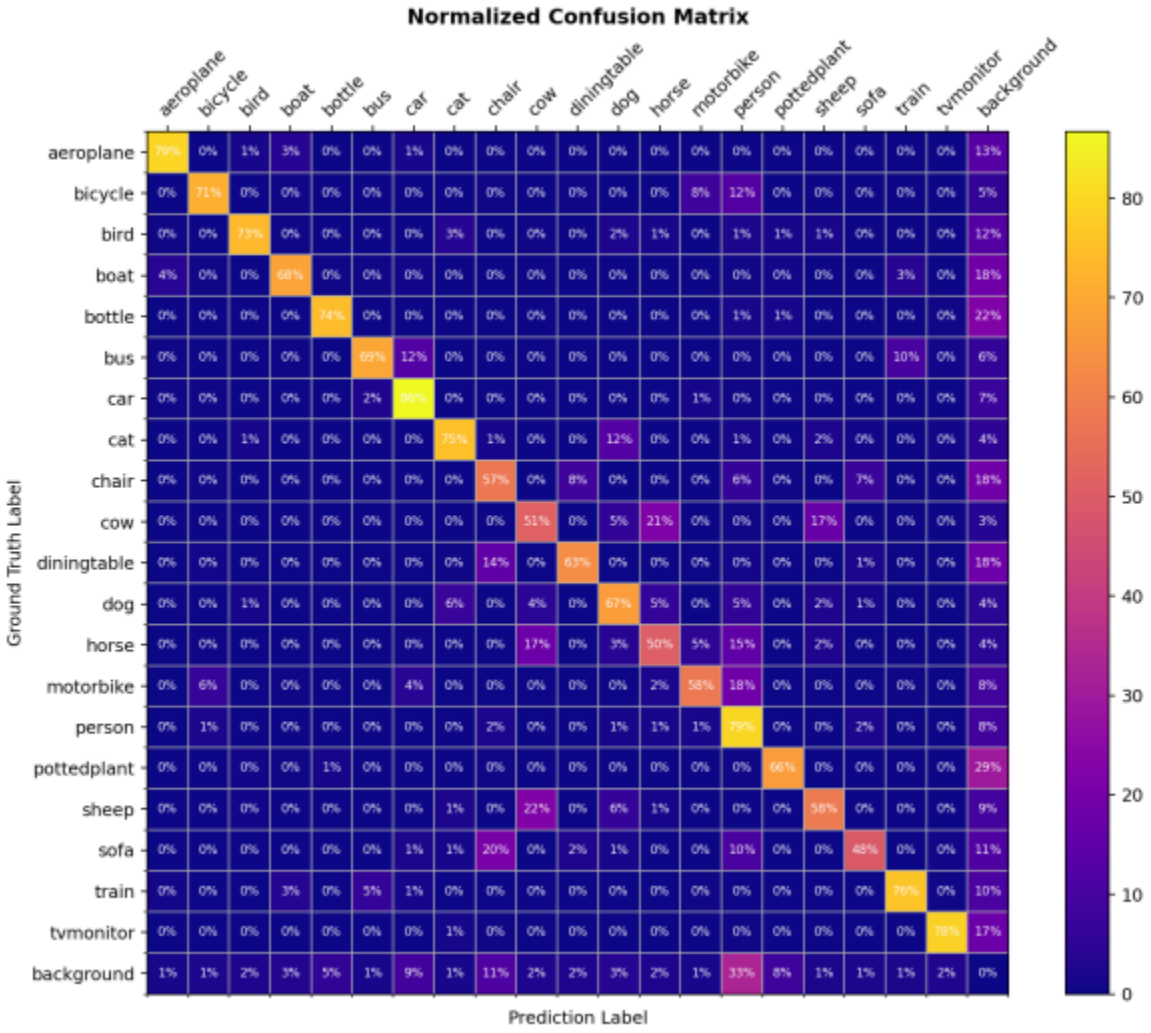
```
differential_evolution step 489: f(x)= 0.386625
2021-07-19 19:46:40,775 - mmdet - INFO Anchor evolution finish. Average IOU: 0.
↪6133754253387451
2021-07-19 19:46:40,776 - mmdet - INFO Anchor differential evolution result:[[10, 12],
↪ [15, 30], [32, 22], [29, 59], [61, 46], [57, 116], [112, 89], [154, 198], [349,␣
↪336]]
2021-07-19 19:46:40,798 - mmdet - INFO Result saved in work_dirs/anchor_optimize_
↪result.json
```

4.13 混淆矩阵

混淆矩阵是对检测结果的概览。tools/analysis_tools/confusion_matrix.py 可对预测结果进行分析，绘制成混淆矩阵表。首先，运行 tools/test.py 保存 .pkl 预测结果。之后再运行：

```
python tools/analysis_tools/confusion_matrix.py ${CONFIG}  ${DETECTION_RESULTS}  $
↪{SAVE_DIR} --show
```

最后你可以得到如图的混淆矩阵：



4.14 COCO 分离和遮挡实例分割性能评估

对于最先进的目标检测器来说，检测被遮挡的物体仍然是一个挑战。我们实现了论文 A Tri-Layer Plugin to Improve Occluded Detection 中提出的指标来计算分离和遮挡目标的召回率。

使用此评价指标有两种方法：

4.14.1 离线评测

我们提供了一个脚本对存储后的检测结果文件计算指标。

首先，使用 tools/test.py 脚本存储检测结果：

```
python tools/test.py ${CONFIG} ${MODEL_PATH} --out results.pkl
```

然后，运行 tools/analysis_tools/coco_occluded_separated_recall.py 脚本来计算分离和遮挡目标的掩码的召回率:

```
python tools/analysis_tools/coco_occluded_separated_recall.py results.pkl --out␣
↪occluded_separated_recall.json
```

输出如下：

```
loading annotations into memory...
Done (t=0.51s)
creating index...
index created!
processing detection results...
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>] 5000/5000, 109.3 task/s,␣
↪elapsed: 46s, ETA:     0s
computing occluded mask recall...
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>] 5550/5550, 780.5 task/s,␣
↪elapsed: 7s, ETA:     0s
COCO occluded mask recall: 58.79%
COCO occluded mask success num: 3263
computing separated mask recall...
[>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>] 3522/3522, 778.3 task/s,␣
↪elapsed: 5s, ETA:     0s
COCO separated mask recall: 31.94%
COCO separated mask success num: 1125

+-----------+--------+-------------+
| mask type | recall | num correct |
+-----------+--------+-------------+
| occluded  | 58.79% | 3263        |
| separated | 31.94% | 1125        |
+-----------+--------+-------------+
Evaluation results have been saved to occluded_separated_recall.json.
```

4.14.2 在线评测

我们实现继承自 CocoMetic 的 CocoOccludedSeparatedMetric。要在训练期间评估分离和遮挡掩码的召回率，只需在配置中将 evaluator 类型替换为 CocoOccludedSeparatedMetric：

```
val_evaluator = dict(
    type='CocoOccludedSeparatedMetric',  # 修改此处
    ann_file=data_root + 'annotations/instances_val2017.json',
    metric=['bbox', 'segm'],
    format_only=False)
test_evaluator = val_evaluator
```

如果您使用了此指标，请引用论文：

```
@article{zhan2022triocc,
    title={A Tri-Layer Plugin to Improve Occluded Detection},
    author={Zhan, Guanqi and Xie, Weidi and Zisserman, Andrew},
    journal={British Machine Vision Conference},
    year={2022}
}
```

4.15 实用的钩子

MMDetection 和 MMEngine 为用户提供了多种多样实用的钩子（Hook），包括 MemoryProfilerHook、NumClassCheckHook 等等。这篇教程介绍了 MMDetection 中实现的钩子功能及使用方式。若使用 MMEngine 定义的钩子请参考 MMEngine 的钩子 API 文档.

4.15.1 CheckInvalidLossHook

4.15.2 NumClassCheckHook

4.15.3 MemoryProfilerHook

内存分析钩子 记录了包括虚拟内存、交换内存、当前进程在内的所有内存信息，它能够帮助捕捉系统的使用状况与发现隐藏的内存泄露问题。为了使用这个钩子，你需要先通过 pip install memory_profiler psutil 命令安装 memory_profiler 和 psutil。

使用

为了使用这个钩子，使用者需要添加如下代码至 config 文件

```
custom_hooks = [
    dict(type='MemoryProfilerHook', interval=50)
]
```

结果

在训练中，你会看到 MemoryProfilerHook 记录的如下信息：

```
The system has 250 GB (246360 MB + 9407 MB) of memory and 8 GB (5740 MB + 2452 MB) of␣
↪swap memory in total. Currently 9407 MB (4.4%) of memory and 5740 MB (29.9%) of␣
↪swap memory were consumed. And the current training process consumed 5434 MB of␣
↪memory.
```

```
2022-04-21 08:49:56,881 - mmengine - INFO - Memory information available_memory:␣
↪246360 MB, used_memory: 9407 MB, memory_utilization: 4.4 %, available_swap_memory:␣
↪5740 MB, used_swap_memory: 2452 MB, swap_memory_utilization: 29.9 %, current_
↪process_memory: 5434 MB
```

4.15.4 SetEpochInfoHook

4.15.5 SyncNormHook

4.15.6 SyncRandomSizeHook

4.15.7 YOLOXLrUpdaterHook

4.15.8 YOLOXModeSwitchHook

4.15.9 如何实现自定义钩子

通常，从模型训练的开始到结束，共有 20 个点位可以执行钩子。我们可以实现自定义钩子在不同点位执行，以便在训练中实现自定义操作。

- global points: `before_run`, `after_run`
- points in training: `before_train`, `before_train_epoch`, `before_train_iter`, `after_train_iter`, `after_train_epoch`, `after_train`
- points in validation: `before_val`, `before_val_epoch`, `before_val_iter`, `after_val_iter`, `after_val_epoch`, `after_val`

- points at testing: `before_test`, `before_test_epoch`, `before_test_iter`, `after_test_iter`, `after_test_epoch`, `after_test`
- other points: `before_save_checkpoint`, `after_save_checkpoint`

比如，我们要实现一个检查 loss 的钩子，当损失为 NaN 时自动结束训练。我们可以把这个过程分为三步：

1. 在 MMEngine 实现一个继承于 `Hook` 类的新钩子，并实现 `after_train_iter` 方法用于检查每 n 次训练迭代后损失是否变为 NaN 。
2. 使用 `@HOOKS.register_module()` 注册实现好了的自定义钩子，如下列代码所示。
3. 在配置文件中添加 `custom_hooks = [dict(type='MemoryProfilerHook', interval=50)]`

```python
from typing import Optional

import torch
from mmengine.hooks import Hook
from mmengine.runner import Runner

from mmdet.registry import HOOKS


@HOOKS.register_module()
class CheckInvalidLossHook(Hook):
    """Check invalid loss hook.

    This hook will regularly check whether the loss is valid
    during training.

    Args:
        interval (int): Checking interval (every k iterations).
            Default: 50.
    """

    def __init__(self, interval: int = 50) -> None:
        self.interval = interval

    def after_train_iter(self,
                         runner: Runner,
                         batch_idx: int,
                         data_batch: Optional[dict] = None,
                         outputs: Optional[dict] = None) -> None:
        """Regularly check whether the loss is valid every n iterations.

        Args:
```

(下页继续)

(续上页)

```
            runner (:obj:`Runner`): The runner of the training process.
            batch_idx (int): The index of the current batch in the train loop.
            data_batch (dict, Optional): Data from dataloader.
                Defaults to None.
            outputs (dict, Optional): Outputs from model. Defaults to None.
        """
        if self.every_n_train_iters(runner, self.interval):
            assert torch.isfinite(outputs['loss']), \
                runner.logger.info('loss become infinite or NaN!')
```

请参考自定义训练配置 了解更多与自定义钩子相关的内容。

4.16 可视化

在阅读本教程之前，建议先阅读 MMEngine 的 Visualization 文档，以对 `Visualizer` 的定义和用法有一个初步的了解。

简而言之，`Visualizer` 在 MMEngine 中实现以满足日常可视化需求，并包含以下三个主要功能：

- 实现通用的绘图 API，例如 `draw_bboxes` 实现了绘制边界框的功能，`draw_lines` 实现了绘制线条的功能。
- 支持将可视化结果、学习率曲线、损失函数曲线以及验证精度曲线写入到各种后端中，包括本地磁盘以及常见的深度学习训练日志工具，例如 TensorBoard 和 Wandb。
- 支持在代码的任何位置调用以可视化或记录模型在训练或测试期间的中间状态，例如特征图和验证结果。

基于 MMEngine 的 `Visualizer`，MMDet 提供了各种预构建的可视化工具，用户可以通过简单地修改以下配置文件来使用它们。

- `tools/analysis_tools/browse_dataset.py` 脚本提供了一个数据集可视化功能，可以在数据经过数据转换后绘制图像和相应的注释，具体描述请参见 `browse_dataset.py`。
- MMEngine 实现了 `LoggerHook`，使用 `Visualizer` 将学习率、损失和评估结果写入由 `Visualizer` 设置的后端。因此，通过修改配置文件中的 `Visualizer` 后端，例如修改为 `TensorBoardVISBackend` 或 `WandbVISBackend`，可以实现日志记录到常用的训练日志工具，如 `TensorBoard` 或 `WandB`，从而方便用户使用这些可视化工具来分析和监控训练过程。
- 在 MMDet 中实现了 `VisualizerHook`，它使用 `Visualizer` 将验证或预测阶段的预测结果可视化或存储到由 `Visualizer` 设置的后端。因此，通过修改配置文件中的 `Visualizer` 后端，例如修改为 `TensorBoardVISBackend` 或 `WandbVISBackend`，可以将预测图像存储到 `TensorBoard` 或 `Wandb` 中。

4.16.1 配置

由于使用了注册机制，在 MMDet 中我们可以通过修改配置文件来设置 Visualizer 的行为。通常，我们会在 configs/_base_/default_runtime.py 中为可视化器定义默认配置，详细信息请参见配置教程。

```
vis_backends = [dict(type='LocalVisBackend')]
visualizer = dict(
    type='DetLocalVisualizer',
    vis_backends=vis_backends,
    name='visualizer')
```

基于上面的例子，我们可以看到 Visualizer 的配置由两个主要部分组成，即 Visualizer 类型和其使用的可视化后端 vis_backends。

- 用户可直接使用 DetLocalVisualizer 来可视化支持任务的标签或预测结果。
- MMDet 默认将可视化后端 vis_backend 设置为本地可视化后端 LocalVisBackend，将所有可视化结果和其他训练信息保存在本地文件夹中。

4.16.2 存储

MMDet 默认使用本地可视化后端 LocalVisBackend，VisualizerHook 和 LoggerHook 中存储的模型损失、学习率、模型评估精度和可视化信息，包括损失、学习率、评估精度将默认保存到 {work_dir}/{config_name}/{time}/{vis_data} 文件夹中。此外，MMDet 还支持其他常见的可视化后端，例如 TensorboardVisBackend 和 WandbVisBackend，您只需要在配置文件中更改 vis_backends 类型为相应的可视化后端即可。例如，只需在配置文件中插入以下代码块即可将数据存储到 TensorBoard 和 Wandb 中。

```
# https://mmengine.readthedocs.io/en/latest/api/visualization.html
_base_.visualizer.vis_backends = [
    dict(type='LocalVisBackend'), #
    dict(type='TensorboardVisBackend'),
    dict(type='WandbVisBackend'),]
```

4.16.3 绘图

绘制预测结果

MMDet 主要使用*DetVisualizationHook*来绘制验证和测试的预测结果，默认情况下 DetVisualizationHook 是关闭的，其默认配置如下。

```
visualization=dict( # 用户可视化验证和测试结果
    type='DetVisualizationHook',
```

(下页继续)

(续上页)

```
    draw=False,
    interval=1,
    show=False)
```

以下表格展示了 DetVisualizationHook 支持的参数。

如果您想在训练或测试期间启用 DetVisualizationHook 相关功能和配置，您只需要修改配置文件，以 configs/rtmdet/rtmdet_tiny_8xb32-300e_coco.py 为例，同时绘制注释和预测，并显示图像，配置文件可以修改如下：

```
visualization = _base_.default_hooks.visualization
visualization.update(dict(draw=True, show=True))
```

test.py 程序提供了--show 和--show-dir 参数，可以在测试过程中可视化注释和预测结果，而不需要修改配置文件，从而进一步简化了测试过程。

```
# 展示测试结果
python tools/test.py configs/rtmdet/rtmdet_tiny_8xb32-300e_coco.py https://download.
↪openmmlab.com/mmdetection/v3.0/rtmdet/rtmdet_tiny_8xb32-300e_coco/rtmdet_tiny_8xb32-
↪300e_coco_20220902_112414-78e30dcc.pth --show

# 指定存储预测结果的位置
python tools/test.py configs/rtmdet/rtmdet_tiny_8xb32-300e_coco.py https://download.
↪openmmlab.com/mmdetection/v3.0/rtmdet/rtmdet_tiny_8xb32-300e_coco/rtmdet_tiny_8xb32-
↪300e_coco_20220902_112414-78e30dcc.pth --show-dir imgs/
```

4.17 检测器鲁棒性检查

4.17.1 介绍

我们提供了在 Benchmarking Robustness in Object Detection: Autonomous Driving when Winter is Coming 中定义的「图像损坏基准测试」上测试目标检测和实例分割模型的工具。此页面提供了如何使用该基准测试的基本教程。

```
@article{michaelis2019winter,
  title={Benchmarking Robustness in Object Detection:
    Autonomous Driving when Winter is Coming},
  author={Michaelis, Claudio and Mitzkus, Benjamin and
    Geirhos, Robert and Rusak, Evgenia and
    Bringmann, Oliver and Ecker, Alexander S. and
    Bethge, Matthias and Brendel, Wieland},
  journal={arXiv:1907.07484},
```

(下页继续)

(续上页)

```
  year={2019}
}
```

4.17.2 关于基准测试

要将结果提交到基准测试，请访问基准测试主页

基准测试是仿照 imagenet-c 基准测试，由 Dan Hendrycks 和 Thomas Dietterich 在Benchmarking Neural Network Robustness to Common Corruptions and Perturbations(ICLR 2019) 中发表。

图像损坏变换功能包含在此库中，但可以使用以下方法单独安装：

```
pip install imagecorruptions
```

与 imagenet-c 相比，我们必须进行一些更改以处理任意大小的图像和灰度图像。我们还修改了“运动模糊”和“雪”损坏，以解除对于 linux 特定库的依赖，否则必须单独安装这些库。有关详细信息，请参阅 imagecorruptions。

4.17.3 使用预训练模型进行推理

我们提供了一个测试脚本来评估模型在基准测试中提供的各种损坏变换组合下的性能。

在数据集上测试

- [x] 单张 GPU 测试
- [ ] 多张 GPU 测试
- [ ] 可视化检测结果

您可以使用以下命令在基准测试中使用 15 种损坏变换来测试模型性能。

```
# single-gpu testing
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}]
```

也可以选择其它不同类型的损坏变换。

```
# noise
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}] --corruptions noise

# blur
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}] --corruptions blur

# wetaher
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}] --corruptions weather

# digital
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}] --corruptions digital
```

或者使用一组自定义的损坏变换，例如：

```
# gaussian noise, zoom blur and snow
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}] --corruptions gaussian_noise zoom_blur␣
↪snow
```

最后，我们也可以选择施加在图像上的损坏变换的严重程度。严重程度从 1 到 5 逐级增强，0 表示不对图像施加损坏变换，即原始图像数据。

```
# severity 1
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}] --severities 1
```

(下页继续)

(续上页)

```
# severities 0,2,4
python tools/analysis_tools/test_robustness.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [--
↪out ${RESULT_FILE}] [--eval ${EVAL_METRICS}] --severities 0 2 4
```

4.17.4 模型测试结果

下表是各模型在 COCO 2017val 上的测试结果。

由于对图像的损坏变换存在随机性，测试结果可能略有不同。

4.18 模型部署

MMDeploy 是 OpenMMLab 的部署仓库，负责包括 MMClassification、MMDetection 等在内的各算法库的部署工作。你可以从这里获取 MMDeploy 对 MMDetection 部署支持的最新文档。

本文的结构如下：

- 安装
- 模型转换
- 模型规范
- 模型推理
  - 后端模型推理
  - *SDK* 模型推理
- 模型支持列表
-

4.18.1 安装

请参考此处安装 mmdet。然后，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要用到的配置文件

4.18.2 模型转换

假设在安装步骤中，mmdetection 和 mmdeploy 代码库在同级目录下，并且当前的工作目录为 mmdetection 的根目录，那么以 Faster R-CNN 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/mmdet/onnx'
save_file = 'end2end.onnx'
deploy_cfg = '../mmdeploy/configs/mmdet/detection/detection_onnxruntime_dynamic.py'
model_cfg = 'configs/faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py'
model_checkpoint = 'faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth'
device = 'cpu'

# 1. convert model to onnx
torch2onnx(img, work_dir, save_file, deploy_cfg, model_cfg,
           model_checkpoint, device)

# 2. extract pipeline info for inference by MMDeploy SDK
export2SDK(deploy_cfg, model_cfg, work_dir, pth=model_checkpoint,
           device=device)
```

转换的关键之一是使用正确的配置文件。项目中已内置了各后端部署配置文件。文件的命名模式是：

```
{task}/{task}_{backend}-{precision}_{static | dynamic}_{shape}.py
```

其中：

- **{task}:** mmdet 中的任务

  mmdet 任务有 2 种：物体检测（detection）、实例分割（instance-seg）。例如，RetinaNet、Faster R-CNN、DETR 等属于前者。Mask R-CNN、SOLO 等属于后者。更多模型-任务的划分，请参考章节模型支持列表。

  请务必使用 detection/detection_*.py 转换检测模型，使用 instance-seg/instance-seg_*.py 转换实例分割模型。

- **{backend}:** 推理后端名称。比如，onnxruntime、tensorrt、pplnn、ncnn、openvino、coreml 等等
- **{precision}:** 推理精度。比如，fp16、int8。不填表示 fp32
- **{static | dynamic}:** 动态、静态 shape
- **{shape}:** 模型输入的 shape 或者 shape 范围

在上例中，你也可以把 Faster R-CNN 转为其他后端模型。比如使用 detection_tensorrt-fp16_dynamic-320x320-1344x1344.py，把模型转为 tensorrt-fp16 模型。

小技巧： 当转 tensorrt 模型时, –device 需要被设置为“cuda”

4.18.3 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/mmdet/onnx，结构如下：

```
mmdeploy_models/mmdet/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。

4.18.4 模型推理

4.18.5 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 = '../mmdeploy/configs/mmdet/detection/detection_onnxruntime_dynamic.py'
model_cfg = 'configs/faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py'
device = 'cpu'
backend_model = ['mmdeploy_models/mmdet/onnx/end2end.onnx']
image = 'demo/demo.jpg'
```

(下页继续)

(续上页)

```
# 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='output_detection.png')
```

4.18.6 SDK 模型推理

你也可以参考如下代码，对 SDK model 进行推理：

```
from mmdeploy_python import Detector
import cv2

img = cv2.imread('demo/demo.jpg')

# create a detector
detector = Detector(model_path='mmdeploy_models/mmdet/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('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。

4.18.7 模型支持列表

请参考这里

4.19 使用 MMDetection 和 Label-Studio 进行半自动化目标检测标注

标注数据是一个费时费力的任务，本文介绍了如何使用 MMDetection 中的 RTMDet 算法联合 Label-Studio 软件进行半自动化标注。具体来说，使用 RTMDet 预测图片生成标注，然后使用 Label-Studio 进行微调标注，社区用户可以参考此流程和方法，将其应用到其他领域。

- RTMDet：RTMDet 是 OpenMMLab 自研的高精度单阶段的目标检测算法，开源于 MMDetection 目标检测工具箱中，其开源协议为 Apache 2.0，工业界的用户可以不受限的免费使用。
- Label Studio 是一款优秀的标注软件，覆盖图像分类、目标检测、分割等领域数据集标注的功能。

本文将使用喵喵数据集的图片，进行半自动化标注。

4.19.1 环境配置

首先需要创建一个虚拟环境，然后安装 PyTorch 和 MMCV。在本文中，我们将指定 PyTorch 和 MMCV 的版本。接下来安装 MMDetection、Label-Studio 和 label-studio-ml-backend，具体步骤如下：

创建虚拟环境：

```
conda create -n rtmdet python=3.9 -y
conda activate rtmdet
```

安装 PyTorch

```
# Linux and Windows CPU only
pip install torch==1.10.1+cpu torchvision==0.11.2+cpu torchaudio==0.10.1 -f https://
↪download.pytorch.org/whl/cpu/torch_stable.html
# Linux and Windows CUDA 11.3
```

(下页继续)

(续上页)

```
pip install torch==1.10.1+cu113 torchvision==0.11.2+cu113 torchaudio==0.10.1 -f␣
↪https://download.pytorch.org/whl/cu113/torch_stable.html
# OSX
pip install torch==1.10.1 torchvision==0.11.2 torchaudio==0.10.1
```

安装 MMCV

```
pip install -U openmim
mim install "mmcv>=2.0.0"
# 安装 mmcv 的过程中会自动安装 mmengine
```

安装 MMDetection

```
git clone https://github.com/open-mmlab/mmdetection
cd mmdetection
pip install -v -e .
```

安装 Label-Studio 和 label-studio-ml-backend

```
# 安装 label-studio 需要一段时间，如果找不到版本请使用官方源
pip install label-studio==1.7.2
pip install label-studio-ml==1.0.9
```

下载 rtmdet 权重

```
cd path/to/mmetection
mkdir work_dirs
cd work_dirs
wget https://download.openmmlab.com/mmdetection/v3.0/rtmdet/rtmdet_m_8xb32-300e_coco/
↪rtmdet_m_8xb32-300e_coco_20220719_112220-229f527c.pth
```

4.19.2 启动服务

启动 RTMDet 后端推理服务：

```
cd path/to/mmetection

label-studio-ml start projects/LabelStudio/backend_template --with \
config_file=configs/rtmdet/rtmdet_m_8xb32-300e_coco.py \
checkpoint_file=./work_dirs/rtmdet_m_8xb32-300e_coco_20220719_112220-229f527c.pth \
device=cpu \
--port 8003
# device=cpu 为使用 CPU 推理，如果使用 GPU 推理，将 cpu 替换为 cuda:0
```

```
(rtmdet) ➜ mmdetection git:(label-studio) X label-studio-ml start projects/LabelStudio/backend_template --with \
config_file=configs/rtmdet/rtmdet_m_8xb32-300e_coco.py \
checkpoint_file=./work_dirs/rtmdet_m_8xb32-300e_coco_20220719_112220-229f527c.pth \
device=cpu \
--port 8003
 * Serving Flask app "label_studio_ml.api" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
[2023-03-30 13:15:45,825] [WARNING] [werkzeug::_log::225]  * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
[2023-03-30 13:15:45,825] [INFO] [werkzeug::_log::225]  * Running on http://172.29.211.69:8003/ (Press CTRL+C to quit)
```

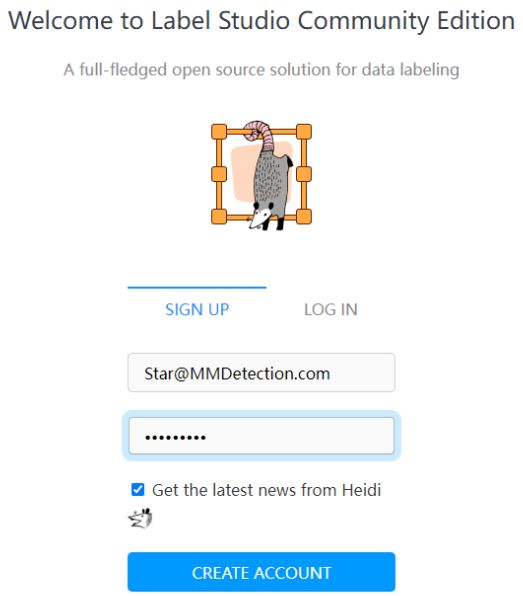
此时，RTMDet 后端推理服务已经启动，后续在 Label-Studio Web 系统中配置 http://localhost:8003 后端推理服务即可。

现在启动 Label-Studio 网页服务：

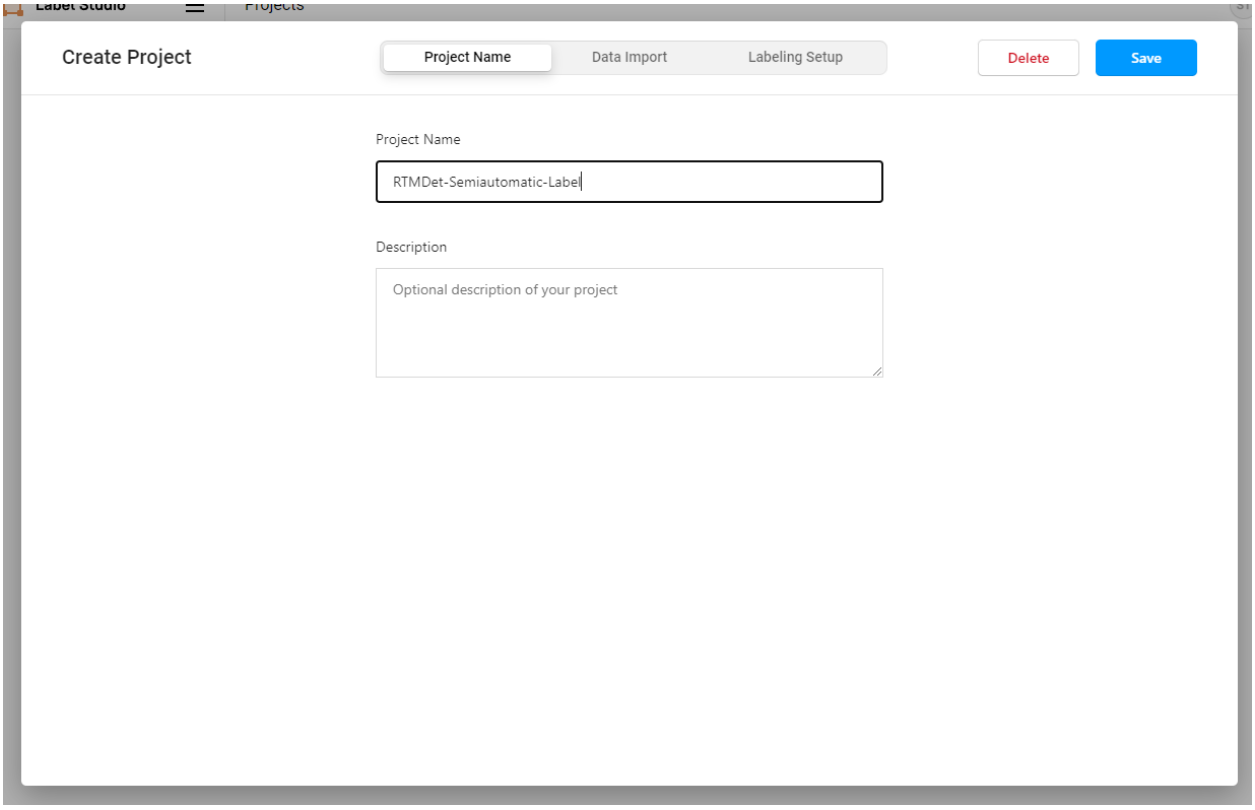
```
label-studio start
```

```
(rtmdet) ➜ mmdetection git:(label-studio) X label-studio start

=> Database and media directory: /home/vansin/.local/share/label-studio
=> Static URL is set to: /static/
=> Database and media directory: /home/vansin/.local/share/label-studio
=> Static URL is set to: /static/
Starting new HTTPS connection (1): pypi.org:443
https://pypi.org:443 "GET /pypi/label-studio/json HTTP/1.1" 200 56156
Performing system checks...

[2023-03-30 05:28:48,234] [django::register_actions_from_dir::97] [INFO] No module named 'data_manager.actions.__pycache_'
[2023-03-30 05:28:48,234] [django::register_actions_from_dir::97] [INFO] No module named 'data_manager.actions.__pycache_'
System check identified no issues (1 silenced).
March 30, 2023 - 05:28:48
Django version 3.2.16, using settings 'label_studio.core.settings.label_studio'
Starting development server at http://0.0.0.0:8080/
Quit the server with CONTROL-C.
```

打开浏览器访问 http://localhost:8080/ 即可看到 Label-Studio 的界面。

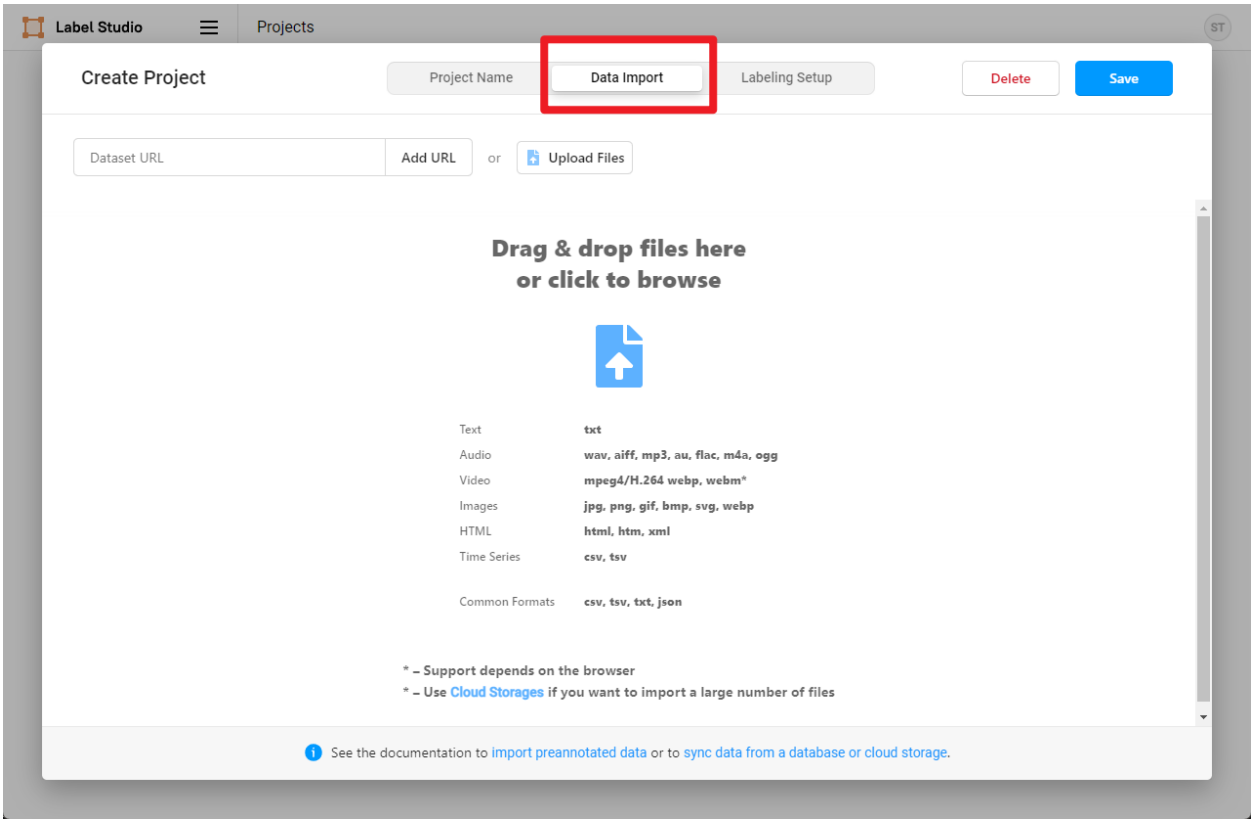


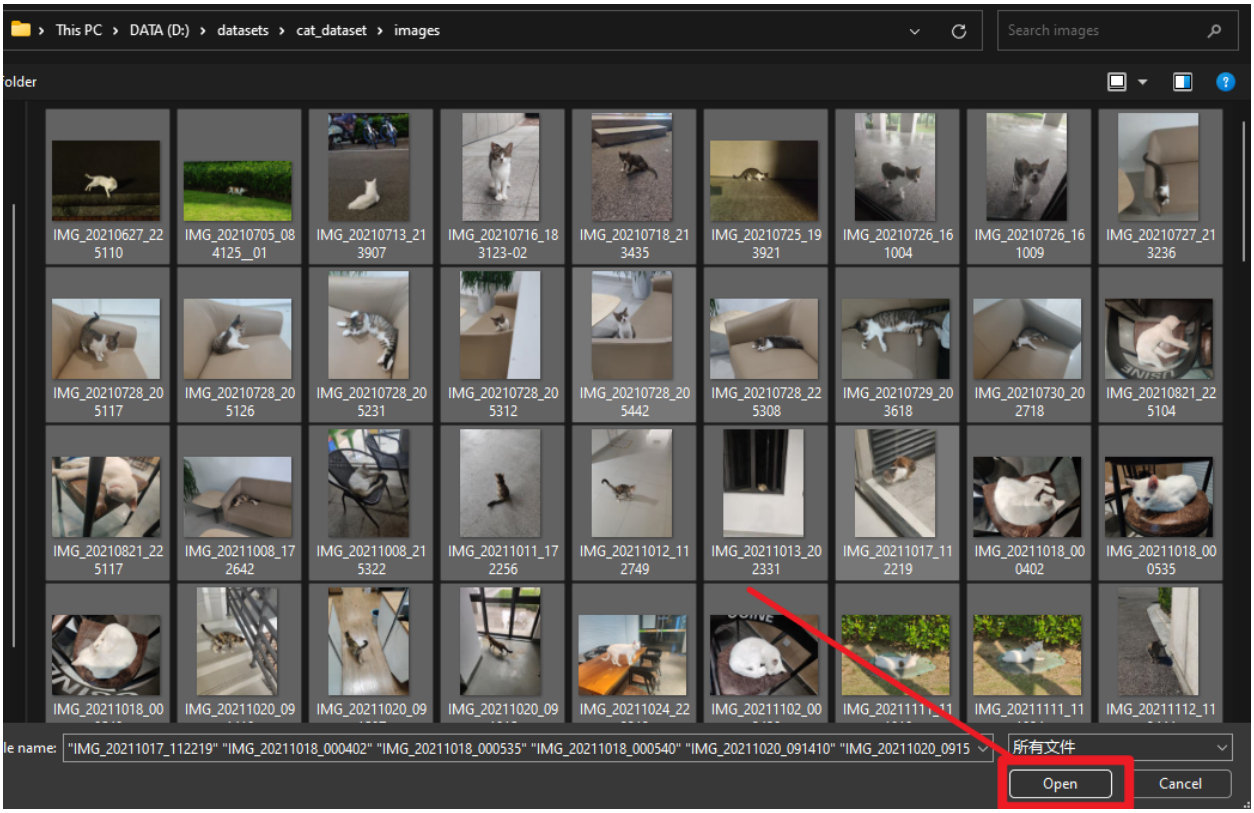
我们注册一个用户，然后创建一个 RTMDet-Semiautomatic-Label 项目。



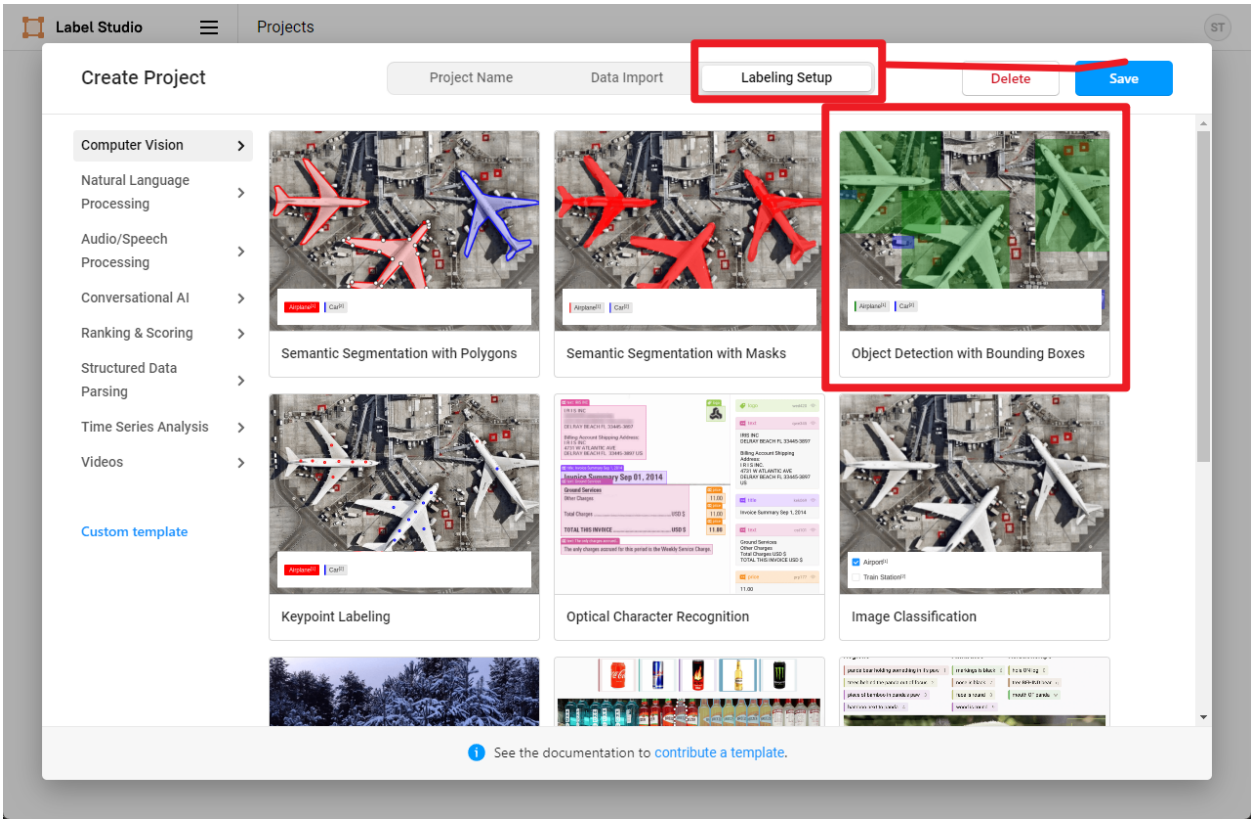
我们通过下面的方式下载好示例的喵喵图片，点击 Data Import 导入需要标注的猫图片。

```
cd path/to/mmetection
mkdir data && cd data

wget https://download.openmmlab.com/mmyolo/data/cat_dataset.zip && unzip cat_dataset.
↪zip
```





然后选择 Object Detection With Bounding Boxes 模板




```
airplane
apple
backpack
banana
baseball_bat
baseball_glove
bear
bed
bench
bicycle
bird
boat
book
bottle
bowl
broccoli
bus
cake
car
carrot
cat
cell_phone
```

(下页继续)

(续上页)

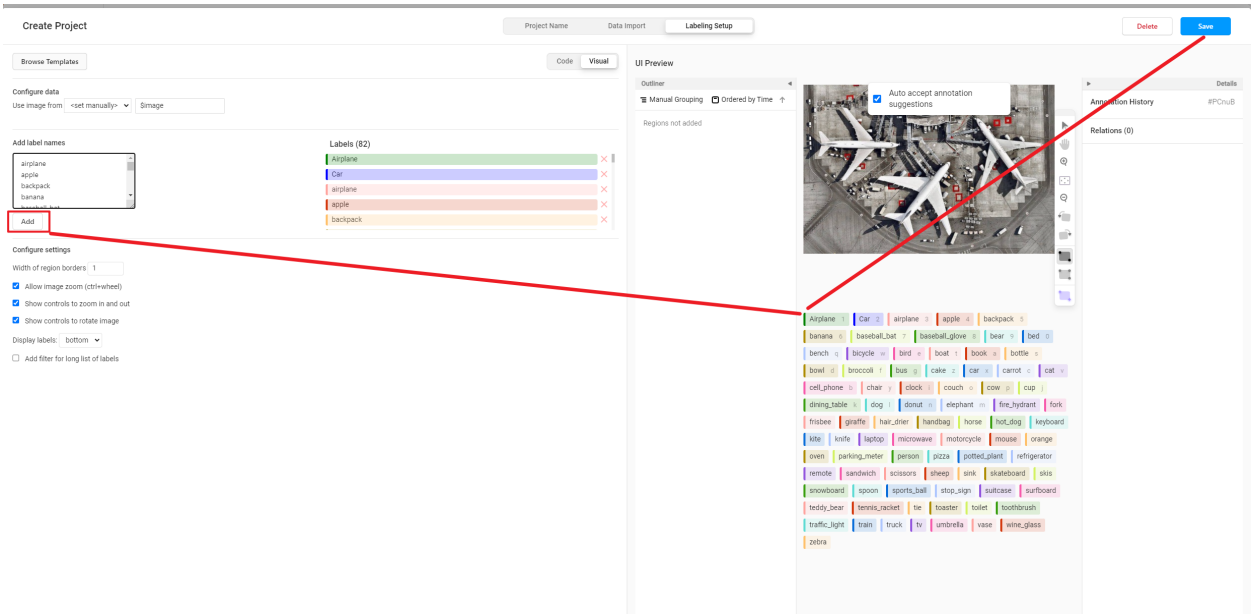
```
chair
clock
couch
cow
cup
dining_table
dog
donut
elephant
fire_hydrant
fork
frisbee
giraffe
hair_drier
handbag
horse
hot_dog
keyboard
kite
knife
laptop
microwave
motorcycle
mouse
orange
oven
parking_meter
person
pizza
potted_plant
refrigerator
remote
sandwich
scissors
sheep
sink
skateboard
skis
snowboard
spoon
sports_ball
stop_sign
suitcase
```

(下页继续)

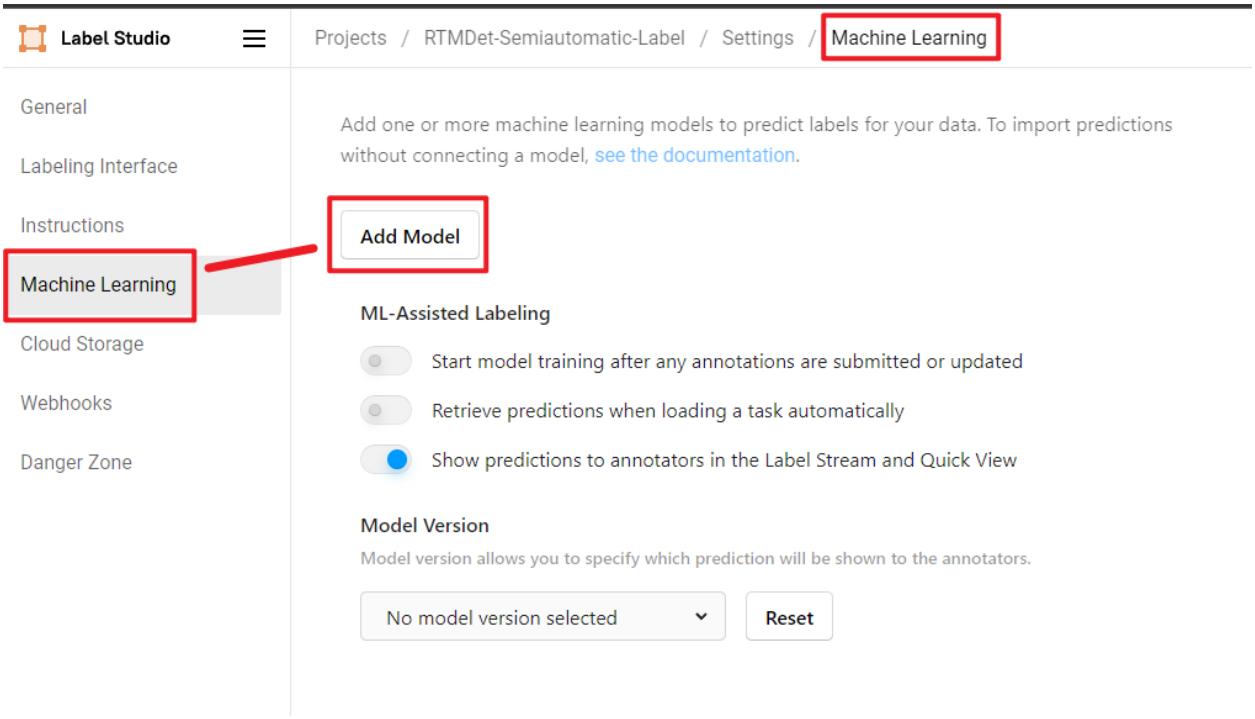
(续上页)

```
surfboard
teddy_bear
tennis_racket
tie
toaster
toilet
toothbrush
traffic_light
train
truck
tv
umbrella
vase
wine_glass
zebra
```

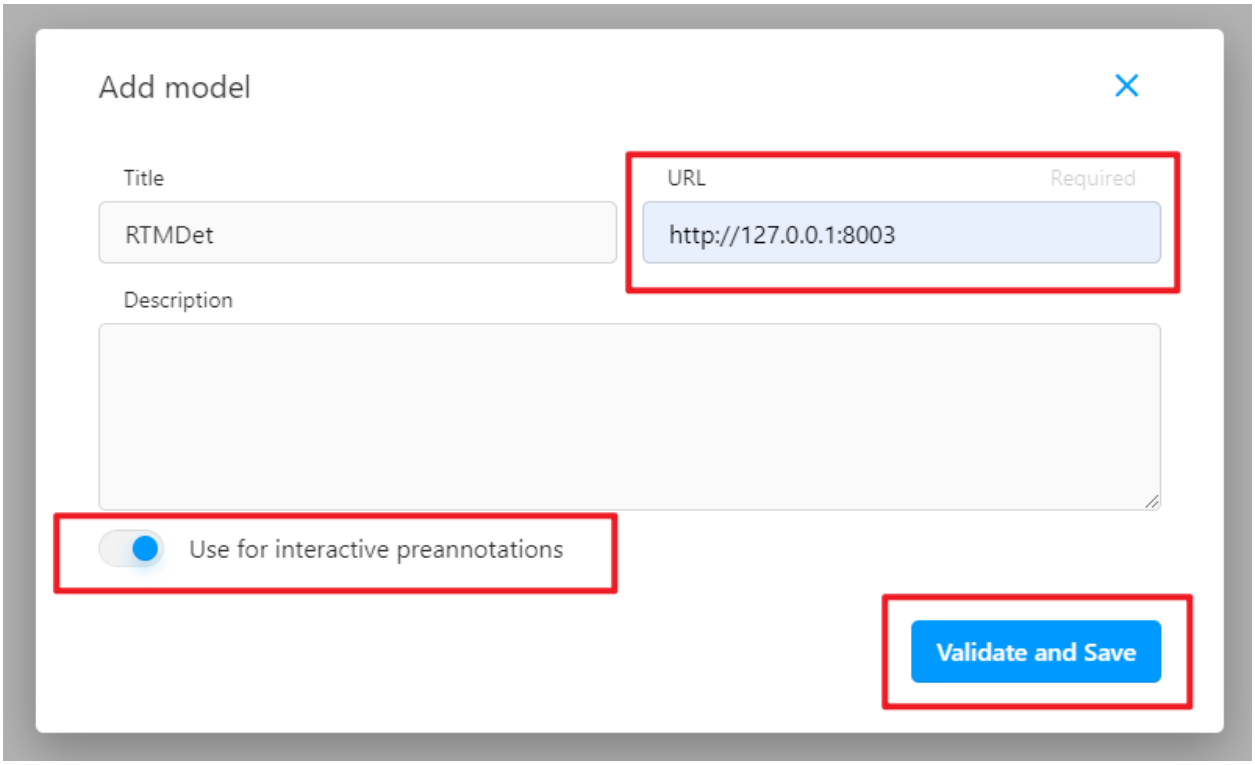
然后将上述类别复制添加到 Label-Studio，然后点击 Save。



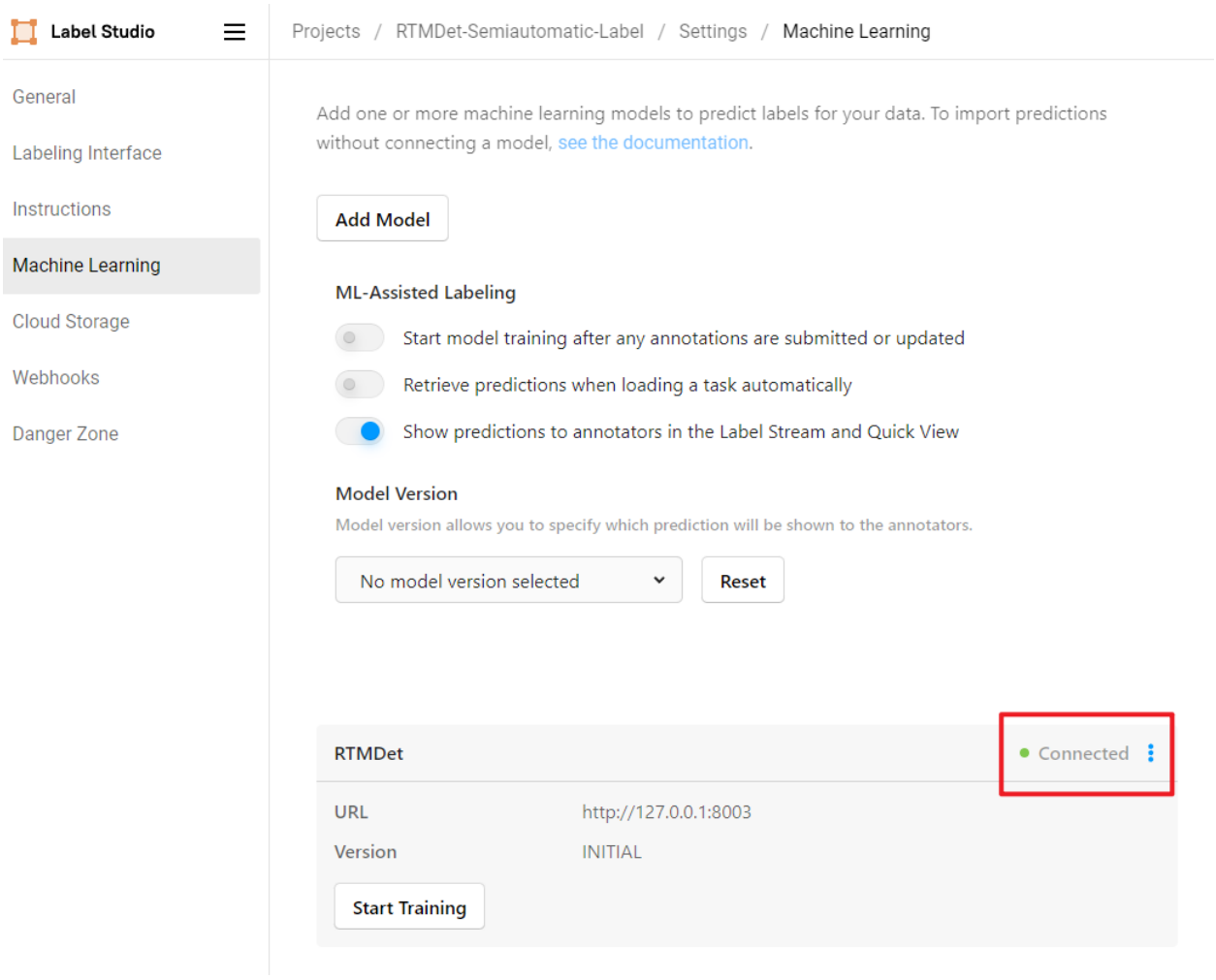
然后在设置中点击 Add Model 添加 RTMDet 后端推理服务。



点击 Validate and Save，然后点击 Start Labeling。

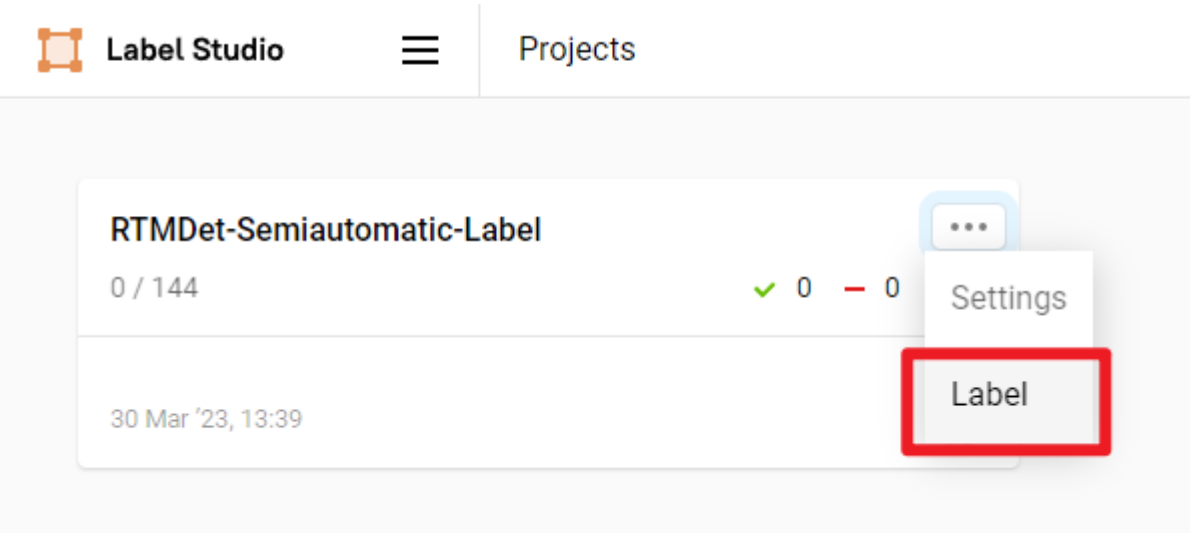


看到如下 Connected 就说明后端推理服务添加成功。

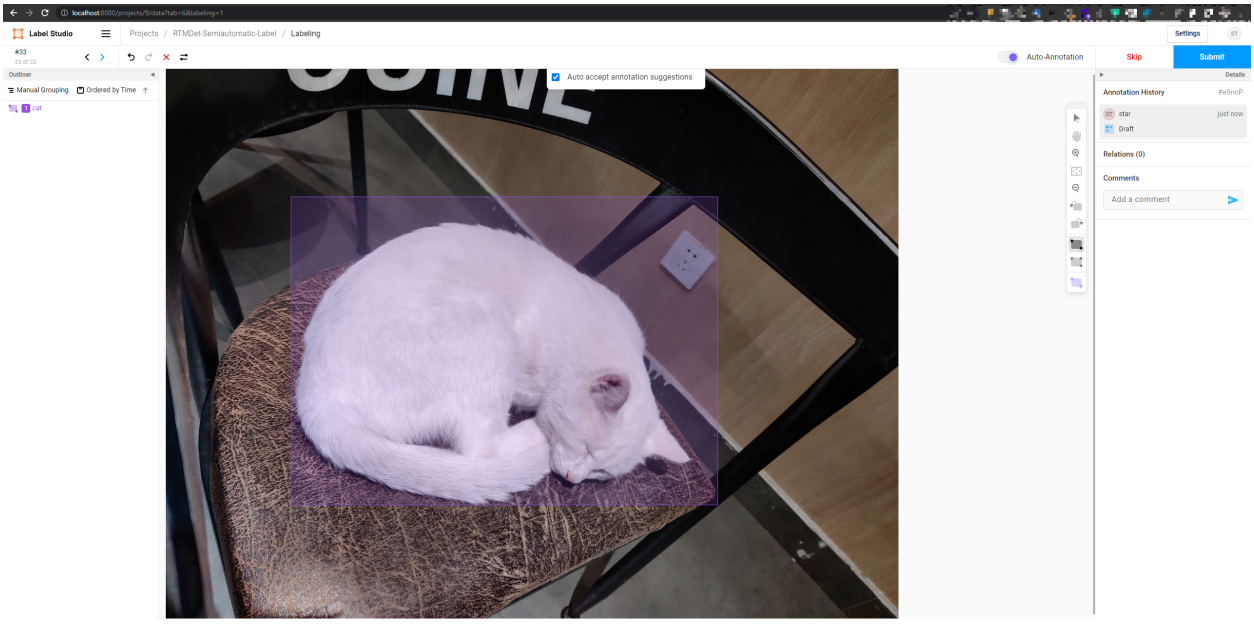


4.19.3 开始半自动化标注

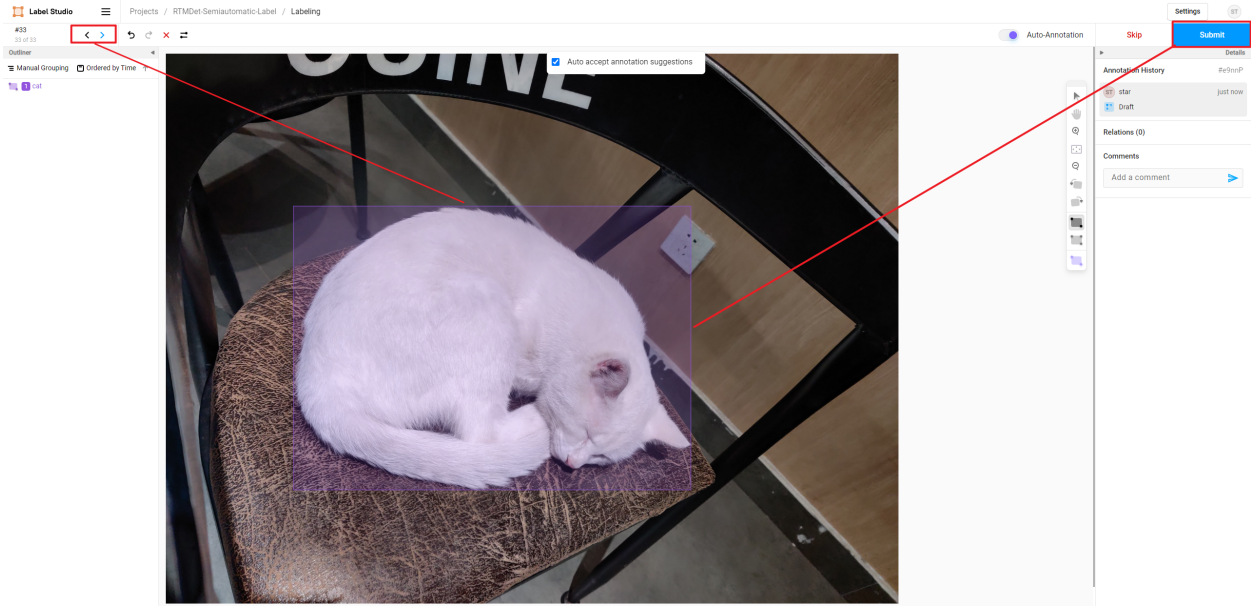
点击 Label 开始标注



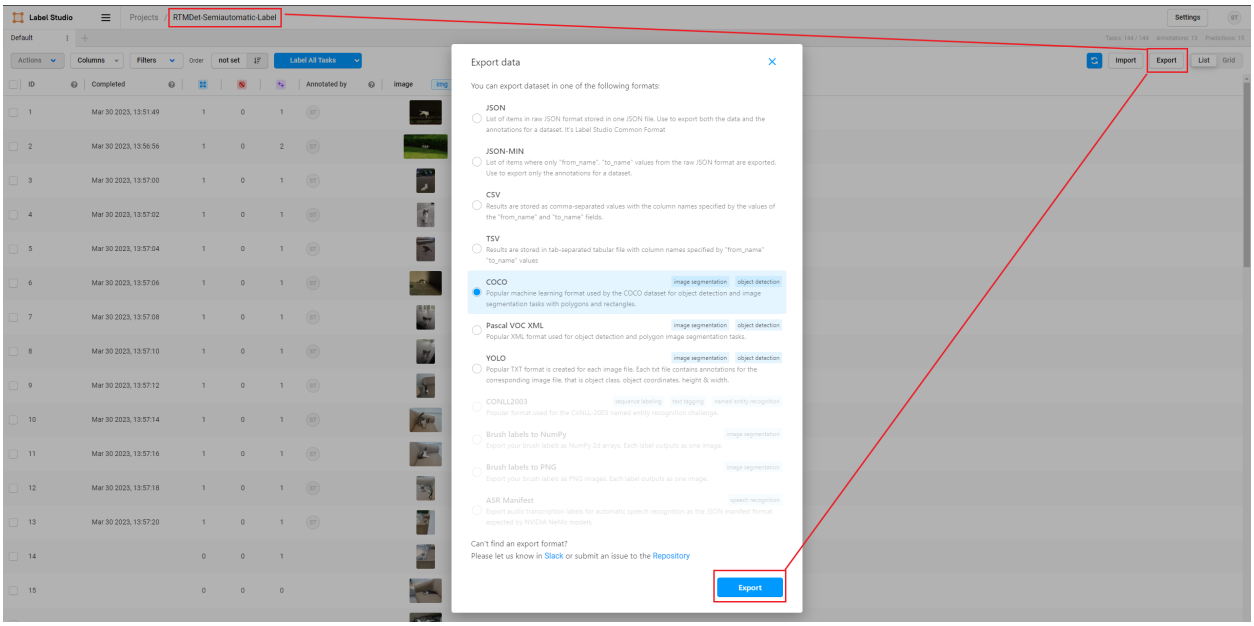
我们可以看到 RTMDet 后端推理服务已经成功返回了预测结果并显示在图片上，我们可以发现这个喵喵预测的框有点大。



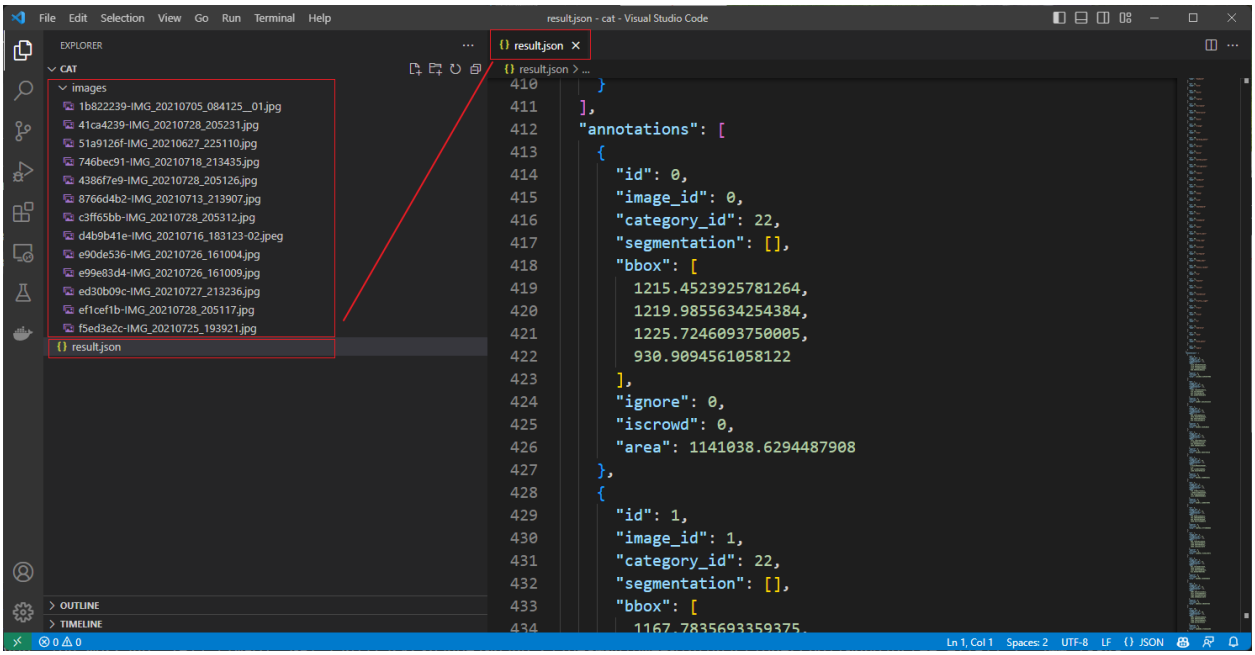
我们手工拖动框，修正一下框的位置，得到以下修正过后的标注，然后点击 Submit，本张图片就标注完毕了。



我们 submit 完毕所有图片后，点击 exprot 导出 COCO 格式的数据集，就能把标注好的数据集的压缩包导出来了。



用 vscode 打开解压后的文件夹，可以看到标注好的数据集，包含了图片和 json 格式的标注文件。



到此半自动化标注就完成了，我们可以用这个数据集在 MMDetection 训练精度更高的模型了，训练出更好的模型，然后再用这个模型继续半自动化标注新采集的图片，这样就可以不断迭代，扩充高质量数据集，提高模型的精度。

4.19.4 使用 MMYOLO 作为后端推理服务

如果想在 MMYOLO 中使用 Label-Studio，可以参考在启动后端推理服务时，将 config_file 和 checkpoint_file 替换为 MMYOLO 的配置文件和权重文件即可。

```
cd path/to/mmetection

label-studio-ml start projects/LabelStudio/backend_template --with \
config_file= path/to/mmyolo_config.py \
checkpoint_file= path/to/mmyolo_weights.pth \
device=cpu \
--port 8003
# device=cpu 为使用 CPU 推理，如果使用 GPU 推理，将 cpu 替换为 cuda:0
```

旋转目标检测和实例分割还在支持中，敬请期待。

CHAPTER 5

基础概念

5.1 数据流（待更新）

5.2 数据结构（待更新）

5.3 模型（待更新）

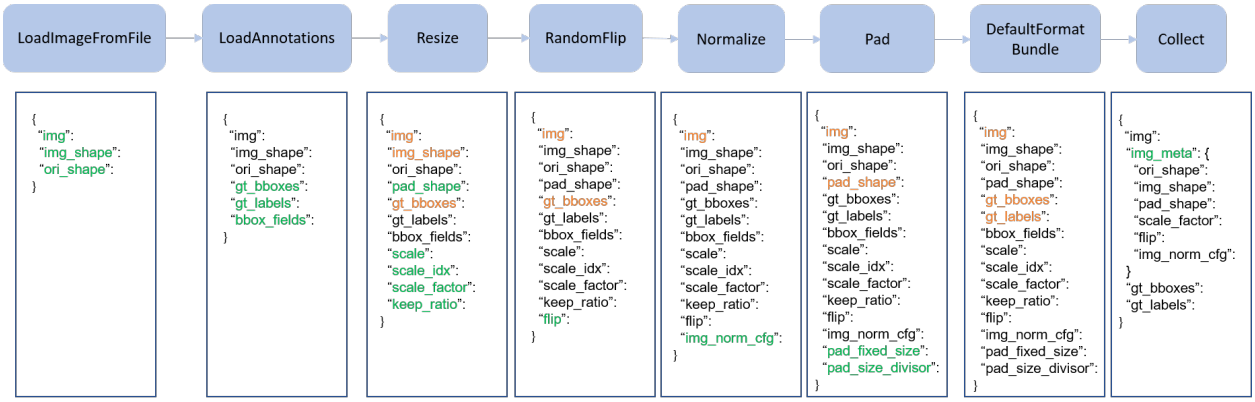
5.4 数据集（待更新）

5.5 数据变换（待更新）

按照惯例，我们使用 `Dataset` 和 `DataLoader` 进行多进程的数据加载。`Dataset` 返回字典类型的数据，数据内容为模型 `forward` 方法的各个参数。由于在目标检测中，输入的图像数据具有不同的大小，我们在 `MMCV` 里引入一个新的 `DataContainer` 类去收集和分发不同大小的输入数据。更多细节请参考这里。

数据的准备流程和数据集是解耦的。通常一个数据集定义了如何处理标注数据（annotations）信息，而一个数据流程定义了准备一个数据字典的所有步骤。一个流程包括一系列的操作，每个操作都把一个字典作为输入，然后再输出一个新的字典给下一个变换操作。

我们在下图展示了一个经典的数据处理流程。蓝色块是数据处理操作，随着数据流程的处理，每个操作都可以在结果字典中加入新的键（标记为绿色）或更新现有的键（标记为橙色）。



这些操作可以分为数据加载（data loading）、预处理（pre-processing）、格式变化（formatting）和测试时数据增强（test-time augmentation）。

下面的例子是 `Faster R-CNN` 的一个流程：

```
img_norm_cfg = dict(
    mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='Resize', img_scale=(1333, 800), keep_ratio=True),
    dict(type='RandomFlip', flip_ratio=0.5),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels']),
]
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(
        type='MultiScaleFlipAug',
        img_scale=(1333, 800),
        flip=False,
        transforms=[
            dict(type='Resize', keep_ratio=True),
            dict(type='RandomFlip'),
            dict(type='Normalize', **img_norm_cfg),
            dict(type='Pad', size_divisor=32),
            dict(type='ImageToTensor', keys=['img']),
            dict(type='Collect', keys=['img']),
        ])
]
```

5.6 精度评测（待更新）

5.7 执行引擎（待更新）

5.8 默认约定

如果你想把 MMDetection 修改为自己的项目，请遵循下面的约定。

5.8.1 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,
                 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]
```

5.8.2 损失

在 MMDetection 中，`model(**data)` 的返回值是一个字典，包含着所有的损失和评价指标，他们将会由 `model(**data)` 返回。

例如，在 bbox head 中，

```
class BBoxHead(nn.Module):
    ...
    def loss(self, ...):
        losses = dict()
        # 分类损失
        losses['loss_cls'] = self.loss_cls(...)
        # 分类准确率
        losses['acc'] = accuracy(...)
        # 边界框损失
        losses['loss_bbox'] = self.loss_bbox(...)
        return losses
```

`'bbox_head.loss()'` 在模型 forward 阶段会被调用。返回的字典中包含了 `'loss_bbox','loss_cls','acc'`。只有 `'loss_bbox'`, `'loss_cls'` 会被用于反向传播，`'acc'` 只会被作为评价指标来监控训练过程。

我们默认，只有那些键的名称中包含 `'loss'` 的值会被用于反向传播。这个行为可以通过修改 `BaseDetector.train_step()` 来改变。

5.8.3 空 proposals

在 MMDetection 中，我们为两阶段方法中空 proposals 的情况增加了特殊处理和单元测试。我们同时需要处理整个 batch 和单一图片中空 proposals 的情况。例如，在 CascadeRoIHead 中，

```
# 简单的测试
...

# 在整个 batch 中 都没有 proposals
if rois.shape[0] == 0:
    bbox_results = [[
        np.zeros((0, 5), dtype=np.float32)
        for _ in range(self.bbox_head[-1].num_classes)
    ]] * num_imgs
    if self.with_mask:
        mask_classes = self.mask_head[-1].num_classes
        segm_results = [[[] for _ in range(mask_classes)]
                        for _ in range(num_imgs)]
        results = list(zip(bbox_results, segm_results))
```

(下页继续)

(续上页)

```
    else:
        results = bbox_results
    return results
...

# 在单张图片中没有 proposals
for i in range(self.num_stages):
    ...
    if i < self.num_stages - 1:
          for j in range(num_imgs):
                # 处理空 proposals
                if rois[j].shape[0] > 0:
                    bbox_label = cls_score[j][:, :-1].argmax(dim=1)
                    refine_roi = self.bbox_head[i].regress_by_class(
                         rois[j], bbox_label[j], bbox_pred[j], img_metas[j])
                    refine_roi_list.append(refine_roi)
```

如果你有自定义的 RoIHead, 你可以参考上面的方法来处理空 proposals 的情况。

5.8.4 全景分割数据集

在 MMDetection 中，我们支持了 COCO 全景分割数据集 CocoPanopticDataset。对于它的实现，我们在这里声明一些默认约定。

1. 在 mmdet<=2.16.0 时，语义分割标注中的前景和背景标签范围与 MMDetection 中的默认规定有所不同。标签 0 代表 VOID 标签。从 mmdet=2.17.0 开始，为了和框的类别标注保持一致，语义分割标注的类别标签也改为从 0 开始，标签 255 代表 VOID 类。为了达成这一目标，我们在流程 Pad 里支持了设置 seg 的填充值的功能。
2. 在评估中，全景分割结果必须是一个与原图大小相同的图。结果图中每个像素的值有如此形式: instance_id * INSTANCE_OFFSET + category_id。

CHAPTER 6

组件定制

6.1 自定义模型

我们简单地把模型的各个组件分为五类：

- 主干网络 (backbone)：通常是一个用来提取特征图 (feature map) 的全卷积网络 (FCN network)，例如：ResNet, MobileNet。
- Neck：主干网络和 Head 之间的连接部分，例如：FPN, PAFPN。
- Head：用于具体任务的组件，例如：边界框预测和掩码预测。
- 区域提取器 (roi extractor)：从特征图中提取 RoI 特征，例如：RoI Align。
- 损失 (loss)：在 Head 组件中用于计算损失的部分，例如：FocalLoss, L1Loss, GHMLoss.

6.1.1 开发新的组件

添加一个新的主干网络

这里，我们以 MobileNet 为例来展示如何开发新组件。

1. 定义一个新的主干网络（以 MobileNet 为例）

新建一个文件 mmdet/models/backbones/mobilenet.py

```
import torch.nn as nn

from mmdet.registry import MODELS


@MODELS.register_module()
class MobileNet(nn.Module):

    def __init__(self, arg1, arg2):
        pass

    def forward(self, x):  # should return a tuple
        pass
```

2. 导入该模块

你可以添加下述代码到 mmdet/models/backbones/__init__.py

```
from .mobilenet import MobileNet
```

或添加：

```
custom_imports = dict(
    imports=['mmdet.models.backbones.mobilenet'],
    allow_failed_imports=False)
```

到配置文件以避免原始代码被修改。

3. 在你的配置文件中使用该主干网络

```
model = dict(
    ...
    backbone=dict(
        type='MobileNet',
        arg1=xxx,
        arg2=xxx),
    ...
```

添加新的 Neck

1. 定义一个 Neck（以 PAFPN 为例）

新建一个文件 mmdet/models/necks/pafpn.py

```
import torch.nn as nn

from mmdet.registry import MODELS


@MODELS.register_module()
class PAFPN(nn.Module):

    def __init__(self,
                 in_channels,
                 out_channels,
                 num_outs,
                 start_level=0,
                 end_level=-1,
                 add_extra_convs=False):
        pass

    def forward(self, inputs):
        # implementation is ignored
        pass
```

2. 导入该模块

你可以添加下述代码到 mmdet/models/necks/__init__.py

```
from .pafpn import PAFPN
```

或添加：

```
custom_imports = dict(
    imports=['mmdet.models.necks.pafpn'],
    allow_failed_imports=False)
```

到配置文件以避免原始代码被修改。

3. 修改配置文件

```
neck=dict(
    type='PAFPN',
    in_channels=[256, 512, 1024, 2048],
    out_channels=256,
    num_outs=5)
```

添加新的 Head

我们以 Double Head R-CNN 为例来展示如何添加一个新的 Head。

首先，添加一个新的 bbox head 到 `mmdet/models/roi_heads/bbox_heads/double_bbox_head.py`。Double Head R-CNN 在目标检测上实现了一个新的 bbox head。为了实现 bbox head，我们需要使用如下的新模块中三个函数。

```
from typing import Tuple

import torch.nn as nn
from mmcv.cnn import ConvModule
from mmengine.model import BaseModule, ModuleList
from torch import Tensor

from mmdet.models.backbones.resnet import Bottleneck
from mmdet.registry import MODELS
from mmdet.utils import ConfigType, MultiConfig, OptConfigType, OptMultiConfig
from .bbox_head import BBoxHead


@MODELS.register_module()
class DoubleConvFCBBoxHead(BBoxHead):
    r"""Bbox head used in Double-Head R-CNN

    .. code-block:: none

                                         /-> cls
                     /-> shared convs ->
                                         \-> reg
        roi features
                                         /-> cls
                     \-> shared fc    ->
                                         \-> reg
    """  # noqa: W605
```

(下页继续)

(续上页)

```
    def __init__(self,
                 num_convs: int = 0,
                 num_fcs: int = 0,
                 conv_out_channels: int = 1024,
                 fc_out_channels: int = 1024,
                 conv_cfg: OptConfigType = None,
                 norm_cfg: ConfigType = dict(type='BN'),
                 init_cfg: MultiConfig = dict(
                     type='Normal',
                     override=[
                         dict(type='Normal', name='fc_cls', std=0.01),
                         dict(type='Normal', name='fc_reg', std=0.001),
                         dict(
                             type='Xavier',
                             name='fc_branch',
                             distribution='uniform')
                     ]),
                 **kwargs) -> None:
        kwargs.setdefault('with_avg_pool', True)
        super().__init__(init_cfg=init_cfg, **kwargs)

    def forward(self, x_cls: Tensor, x_reg: Tensor) -> Tuple[Tensor]:
```

然后，如有必要，实现一个新的 bbox head。我们打算从 StandardRoIHead 来继承新的 DoubleHeadRoIHead。我们可以发现 StandardRoIHead 已经实现了下述函数。

```
from typing import List, Optional, Tuple

import torch
from torch import Tensor

from mmdet.registry import MODELS, TASK_UTILS
from mmdet.structures import DetDataSample
from mmdet.structures.bbox import bbox2roi
from mmdet.utils import ConfigType, InstanceList
from ..task_modules.samplers import SamplingResult
from ..utils import empty_instances, unpack_gt_instances
from .base_roi_head import BaseRoIHead


@MODELS.register_module()
class StandardRoIHead(BaseRoIHead):
```

(下页继续)

(续上页)

```
    """Simplest base roi head including one bbox head and one mask head."""

    def init_assigner_sampler(self) -> None:

    def init_bbox_head(self, bbox_roi_extractor: ConfigType,
                       bbox_head: ConfigType) -> None:

    def init_mask_head(self, mask_roi_extractor: ConfigType,
                       mask_head: ConfigType) -> None:

    def forward(self, x: Tuple[Tensor],
                rpn_results_list: InstanceList) -> tuple:

    def loss(self, x: Tuple[Tensor], rpn_results_list: InstanceList,
             batch_data_samples: List[DetDataSample]) -> dict:

    def _bbox_forward(self, x: Tuple[Tensor], rois: Tensor) -> dict:

    def bbox_loss(self, x: Tuple[Tensor],
                  sampling_results: List[SamplingResult]) -> dict:

    def mask_loss(self, x: Tuple[Tensor],
                  sampling_results: List[SamplingResult], bbox_feats: Tensor,
                  batch_gt_instances: InstanceList) -> dict:

    def _mask_forward(self,
                      x: Tuple[Tensor],
                      rois: Tensor = None,
                      pos_inds: Optional[Tensor] = None,
                      bbox_feats: Optional[Tensor] = None) -> dict:

    def predict_bbox(self,
                     x: Tuple[Tensor],
                     batch_img_metas: List[dict],
                     rpn_results_list: InstanceList,
                     rcnn_test_cfg: ConfigType,
                     rescale: bool = False) -> InstanceList:

    def predict_mask(self,
                     x: Tuple[Tensor],
                     batch_img_metas: List[dict],
                     results_list: InstanceList,
                     rescale: bool = False) -> InstanceList:
```

Double Head 的修改主要在 bbox_forward 的逻辑中，且它从 `StandardRoIHead` 中继承了其他逻辑。在 `mmdet/models/roi_heads/double_roi_head.py` 中，我们用下述代码实现新的 bbox head：

```python
from typing import Tuple

from torch import Tensor

from mmdet.registry import MODELS
from .standard_roi_head import StandardRoIHead


@MODELS.register_module()
class DoubleHeadRoIHead(StandardRoIHead):
    """RoI head for `Double Head RCNN <https://arxiv.org/abs/1904.06493>`_.

    Args:
        reg_roi_scale_factor (float): The scale factor to extend the rois
            used to extract the regression features.
    """

    def __init__(self, reg_roi_scale_factor: float, **kwargs):
        super().__init__(**kwargs)
        self.reg_roi_scale_factor = reg_roi_scale_factor

    def _bbox_forward(self, x: Tuple[Tensor], rois: Tensor) -> dict:
        """Box head forward function used in both training and testing.

        Args:
            x (tuple[Tensor]): List of multi-level img features.
            rois (Tensor): RoIs with the shape (n, 5) where the first
                column indicates batch id of each RoI.

        Returns:
             dict[str, Tensor]: Usually returns a dictionary with keys:

                - `cls_score` (Tensor): Classification scores.
                - `bbox_pred` (Tensor): Box energies / deltas.
                - `bbox_feats` (Tensor): Extract bbox RoI features.
        """
        bbox_cls_feats = self.bbox_roi_extractor(
            x[:self.bbox_roi_extractor.num_inputs], rois)
        bbox_reg_feats = self.bbox_roi_extractor(
            x[:self.bbox_roi_extractor.num_inputs],
            rois,
            roi_scale_factor=self.reg_roi_scale_factor)
```

(下页继续)

(续上页)

```
        if self.with_shared_head:
            bbox_cls_feats = self.shared_head(bbox_cls_feats)
            bbox_reg_feats = self.shared_head(bbox_reg_feats)
        cls_score, bbox_pred = self.bbox_head(bbox_cls_feats, bbox_reg_feats)

        bbox_results = dict(
            cls_score=cls_score,
            bbox_pred=bbox_pred,
            bbox_feats=bbox_cls_feats)
        return bbox_results
```

最终，用户需要把该模块添加到 mmdet/models/bbox_heads/__init__.py 和 mmdet/models/roi_heads/__init__.py 以使相关的注册表可以找到并加载他们。

或者，用户可以添加：

```
custom_imports=dict(
    imports=['mmdet.models.roi_heads.double_roi_head', 'mmdet.models.roi_heads.bbox_
↪heads.double_bbox_head'])
```

到配置文件并实现相同的目的。

Double Head R-CNN 的配置文件如下：

```
_base_ = '../faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py'
model = dict(
    roi_head=dict(
        type='DoubleHeadRoIHead',
        reg_roi_scale_factor=1.3,
        bbox_head=dict(
            _delete_=True,
            type='DoubleConvFCBBoxHead',
            num_convs=4,
            num_fcs=2,
            in_channels=256,
            conv_out_channels=1024,
            fc_out_channels=1024,
            roi_feat_size=7,
            num_classes=80,
            bbox_coder=dict(
                type='DeltaXYWHBBoxCoder',
                target_means=[0., 0., 0., 0.],
                target_stds=[0.1, 0.1, 0.2, 0.2]),
            reg_class_agnostic=False,
            loss_cls=dict(
```

(下页继续)

(续上页)

```
            type='CrossEntropyLoss', use_sigmoid=False, loss_weight=2.0),
        loss_bbox=dict(type='SmoothL1Loss', beta=1.0, loss_weight=2.0))))
```

从 MMDetection 2.0 版本起，配置系统支持继承配置以使用户可以专注于修改。Double Head R-CNN 主要使用了一个新的 `DoubleHeadRoIHead` 和一个新的 `DoubleConvFCBBoxHead`，参数需要根据每个模块的 `__init__` 函数来设置。

添加新的损失

假设你想添加一个新的损失 `MyLoss` 用于边界框回归。为了添加一个新的损失函数，用户需要在 `mmdet/models/losses/my_loss.py` 中实现。装饰器 `weighted_loss` 可以使损失每个部分加权。

```
import torch
import torch.nn as nn

from mmdet.registry import LOSSES
from .utils import weighted_loss


@weighted_loss
def my_loss(pred, target):
    assert pred.size() == target.size() and target.numel() > 0
    loss = torch.abs(pred - target)
    return loss

@LOSSES.register_module()
class MyLoss(nn.Module):

    def __init__(self, reduction='mean', loss_weight=1.0):
        super(MyLoss, self).__init__()
        self.reduction = reduction
        self.loss_weight = loss_weight

    def forward(self,
                pred,
                target,
                weight=None,
                avg_factor=None,
                reduction_override=None):
        assert reduction_override in (None, 'none', 'mean', 'sum')
        reduction = (
            reduction_override if reduction_override else self.reduction)
```

(下页继续)

(续上页)

```
        loss_bbox = self.loss_weight * my_loss(
            pred, target, weight, reduction=reduction, avg_factor=avg_factor)
        return loss_bbox
```

然后，用户需要把它加到 mmdet/models/losses/__init__.py。

```
from .my_loss import MyLoss, my_loss
```

或者，你可以添加：

```
custom_imports=dict(
    imports=['mmdet.models.losses.my_loss'])
```

到配置文件来实现相同的目的。

如使用，请修改 loss_xxx 字段。因为 MyLoss 是用于回归的，你需要在 Head 中修改 loss_xxx 字段。

```
loss_bbox=dict(type='MyLoss', loss_weight=1.0))
```

6.2 自定义损失函数

MMDetection 为用户提供了不同的损失函数。但是默认的配置可能无法适应不同的数据和模型，所以用户可能会希望修改某一个损失函数来适应新的情况。

本教程首先详细的解释计算损失的过程然后给出一些关于如何修改每一个步骤的指导。对损失的修改可以被分为微调和加权。

6.2.1 一个损失的计算过程

给定输入（包括预测和目标，以及权重），损失函数会把输入的张量映射到最后的损失标量。映射过程可以分为下面五个步骤：

1. 设置采样方法为对正负样本进行采样。
2. 通过损失核函数获取**元素**或者**样本**损失。
3. 通过权重张量来给损失**逐元素**权重。
4. 把损失张量归纳为一个**标量**。
5. 用一个**张量**给当前损失一个权重。

6.2.2 设置采样方法（步骤 1）

对于一些损失函数，需要采样策略来避免正负样本之间的不平衡。

例如，在 RPN head 中使用 `CrossEntropyLoss` 时，我们需要在 `train_cfg` 中设置 `RandomSampler`

```
train_cfg=dict(
    rpn=dict(
        sampler=dict(
            type='RandomSampler',
            num=256,
            pos_fraction=0.5,
            neg_pos_ub=-1,
            add_gt_as_proposals=False))
```

对于其他一些具有正负样本平衡机制的损失，例如 Focal Loss、GHMC 和 QualityFocalLoss，不再需要进行采样。

6.2.3 微调损失

微调一个损失主要与步骤 2，4，5 有关，大部分的修改可以在配置文件中指定。这里我们用 Focal Loss (FL) 作为例子。下面的代码分别是构建 FL 的方法和它的配置文件，他们是一一对应的。

```
@LOSSES.register_module()
class FocalLoss(nn.Module):

    def __init__(self,
                 use_sigmoid=True,
                 gamma=2.0,
                 alpha=0.25,
                 reduction='mean',
                 loss_weight=1.0):
```

```
loss_cls=dict(
    type='FocalLoss',
    use_sigmoid=True,
    gamma=2.0,
    alpha=0.25,
    loss_weight=1.0)
```

微调超参数（步骤 2）

gamma 和 beta 是 Focal Loss 中的两个超参数。如果我们想把 gamma 的值设为 1.5，把 alpha 的值设为 0.5，我们可以在配置文件中按照如下指定：

```
loss_cls=dict(
    type='FocalLoss',
    use_sigmoid=True,
    gamma=1.5,
    alpha=0.5,
    loss_weight=1.0)
```

微调归纳方式（步骤 4）

Focal Loss 默认的归纳方式是 mean。如果我们想把归纳方式从 mean 改成 sum，我们可以在配置文件中按照如下指定：

```
loss_cls=dict(
    type='FocalLoss',
    use_sigmoid=True,
    gamma=2.0,
    alpha=0.25,
    loss_weight=1.0,
    reduction='sum')
```

微调损失权重（步骤 5）

这里的损失权重是一个标量，他用来控制多任务学习中不同损失的重要程度，例如，分类损失和回归损失。如果我们想把分类损失的权重设为 0.5，我们可以在配置文件中如下指定：

```
loss_cls=dict(
    type='FocalLoss',
    use_sigmoid=True,
    gamma=2.0,
    alpha=0.25,
    loss_weight=0.5)
```

6.2.4 加权损失（步骤 3）

加权损失就是我们逐元素修改损失权重。更具体来说，我们给损失张量乘以一个与他有相同形状的权重张量。所以，损失中不同的元素可以被赋予不同的比例，所以这里叫做逐元素。损失的权重在不同模型中变化很大，而且与上下文相关，但是总的来说主要有两种损失权重：分类损失的 `label_weights` 和边界框的 `bbox_weights`。你可以在相应的头中的 `get_target` 方法中找到他们。这里我们使用 ATSSHead 作为一个例子。它继承了 AnchorHead ，但是我们重写它的 `get_targets` 方法来产生不同的 `label_weights` 和 `bbox_weights`。

```
class ATSSHead(AnchorHead):

    ...

    def get_targets(self,
                    anchor_list,
                    valid_flag_list,
                    gt_bboxes_list,
                    img_metas,
                    gt_bboxes_ignore_list=None,
                    gt_labels_list=None,
                    label_channels=1,
                    unmap_outputs=True):
```

6.3 自定义数据集

6.3.1 支持新的数据格式

为了支持新的数据格式，可以选择将数据转换成现成的格式（COCO 或者 PASCAL）或将其转换成中间格式。当然也可以选择以离线的形式（在训练之前使用脚本转换）或者在线的形式（实现一个新的 dataset 在训练中进行转换）来转换数据。

在 MMDetection 中，建议将数据转换成 COCO 格式并以离线的方式进行，因此在完成数据转换后只需修改配置文件中的标注数据的路径和类别即可。

将新的数据格式转换为现有的数据格式

最简单的方法就是将你的数据集转换成现有的数据格式（COCO 或者 PASCAL VOC）

COCO 格式的 JSON 标注文件有如下必要的字段：

```
'images': [
    {
```

(下页继续)

(续上页)

```
        'file_name': 'COCO_val2014_000000001268.jpg',
        'height': 427,
        'width': 640,
        'id': 1268
    },
    ...
],

'annotations': [
    {
        'segmentation': [[192.81,
            247.09,
            ...
            219.03,
            249.06]],  # 如果有 mask 标签且为多边形 XY 点坐标格式，则需要保证至少包括 3 个点坐标，否则为无效多边形
        'area': 1035.749,
        'iscrowd': 0,
        'image_id': 1268,
        'bbox': [192.81, 224.8, 74.73, 33.43],
        'category_id': 16,
        'id': 42986
    },
    ...
],

'categories': [
    {'id': 0, 'name': 'car'},
 ]
```

在 JSON 文件中有三个必要的键：

- `images`: 包含多个图片以及它们的信息的数组，例如 `file_name`、`height`、`width` 和 `id`。
- `annotations`: 包含多个实例标注信息的数组。
- `categories`: 包含多个类别名字和 ID 的数组。

在数据预处理之后，使用现有的数据格式来训练自定义的新数据集有如下两步（以 COCO 为例）：

1. 为自定义数据集修改配置文件。
2. 检查自定义数据集的标注。

这里我们举一个例子来展示上面的两个步骤，这个例子使用包括 5 个类别的 COCO 格式的数据集来训练一个现有的 Cascade Mask R-CNN R50-FPN 检测器

1. 为自定义数据集修改配置文件

配置文件的修改涉及两个方面：

1. dataloaer 部分。需要在 train_dataloader.dataset、val_dataloader.dataset 和 test_dataloader.dataset 中添加 metainfo=dict(classes=classes)，其中 classes 必须是 tuple 类型。
2. model 部分中的 num_classes。需要将默认值（COCO 数据集中为 80）修改为自定义数据集中的类别数。

configs/my_custom_config.py 内容如下：

```
# 新的配置来自基础的配置以更好地说明需要修改的地方
_base_ = './cascade_mask_rcnn_r50_fpn_1x_coco.py'

# 1. 数据集设定
dataset_type = 'CocoDataset'
classes = ('a', 'b', 'c', 'd', 'e')
data_root='path/to/your/'

train_dataloader = dict(
    batch_size=2,
    num_workers=2,
    dataset=dict(
        type=dataset_type,
        # 将类别名字添加至 `metainfo` 字段中
        metainfo=dict(classes=classes),
        data_root=data_root,
        ann_file='train/annotation_data',
        data_prefix=dict(img='train/image_data')
        )
    )

val_dataloader = dict(
    batch_size=1,
    num_workers=2,
    dataset=dict(
        type=dataset_type,
        test_mode=True,
        # 将类别名字添加至 `metainfo` 字段中
        metainfo=dict(classes=classes),
        data_root=data_root,
        ann_file='val/annotation_data',
        data_prefix=dict(img='val/image_data')
```

(下页继续)

(续上页)

```
    )

test_dataloader = dict(
    batch_size=1,
    num_workers=2,
    dataset=dict(
        type=dataset_type,
        test_mode=True,
        # 将类别名字添加至 `metainfo` 字段中
        metainfo=dict(classes=classes),
        data_root=data_root,
        ann_file='test/annotation_data',
        data_prefix=dict(img='test/image_data')
        )
    )

# 2. 模型设置

# 将所有的 `num_classes` 默认值修改为 5 (原来为 80)
model = dict(
    roi_head=dict(
        bbox_head=[
            dict(
                type='Shared2FCBBoxHead',
                # 将所有的 `num_classes` 默认值修改为 5 (原来为 80)
                num_classes=5),
            dict(
                type='Shared2FCBBoxHead',
                # 将所有的 `num_classes` 默认值修改为 5 (原来为 80)
                num_classes=5),
            dict(
                type='Shared2FCBBoxHead',
                # 将所有的 `num_classes` 默认值修改为 5 (原来为 80)
                num_classes=5)],
    # 将所有的 `num_classes` 默认值修改为 5 (原来为 80)
    mask_head=dict(num_classes=5)))
```

2. 检查自定义数据集的标注

假设你自己的数据集是 COCO 格式，那么需要保证数据的标注没有问题：

1. 标注文件中 `categories` 的长度要与配置中的 `classes` 元组长度相匹配，它们都表示有几类。(如例子中有 5 个类别)
2. 配置文件中 `classes` 字段应与标注文件里 `categories` 下的 `name` 有相同的元素且顺序一致。MMDetection 会自动将 `categories` 中不连续的 `id` 映射成连续的索引，因此 `categories` 下的 `name` 的字符串顺序会影响标签的索引。同时，配置文件中的 `classes` 的字符串顺序也会影响到预测框可视化时的标签。
3. `annotations` 中的 `category_id` 必须是有效的值。比如所有 `category_id` 的值都应该属于 `categories` 中的 `id`。

下面是一个有效标注的例子：

```
'annotations': [
    {
        'segmentation': [[192.81,
            247.09,
            ...
            219.03,
            249.06]],  # 如果有 mask 标签。
        'area': 1035.749,
        'iscrowd': 0,
        'image_id': 1268,
        'bbox': [192.81, 224.8, 74.73, 33.43],
        'category_id': 16,
        'id': 42986
    },
    ...
],

# MMDetection 会自动将 `categories` 中不连续的 `id` 映射成连续的索引。
'categories': [
    {'id': 1, 'name': 'a'}, {'id': 3, 'name': 'b'}, {'id': 4, 'name': 'c'}, {'id': 16,
↪ 'name': 'd'}, {'id': 17, 'name': 'e'},
 ]
```

我们使用这种方式来支持 CityScapes 数据集。脚本在 cityscapes.py 并且我们提供了微调的 configs.

注意

1. 对于实例分割数据集, **MMDetection 目前只支持评估 COCO 格式的 mask AP**.
2. 推荐训练之前进行离线转换，这样就可以继续使用 `CocoDataset` 且只需修改标注文件的路径以及训

练的种类。

调整新的数据格式为中间格式

如果不想将标注格式转换为 COCO 或者 PASCAL 格式也是可行的。实际上，我们在 MMEngine 的 BaseDataset 中定义了一种简单的标注格式并且与所有现有的数据格式兼容，也能进行离线或者在线转换。

数据集的标注必须为 `json` 或 `yaml`，`yml` 或 `pickle`，`pkl` 格式；标注文件中存储的字典必须包含 `metainfo` 和 `data_list` 两个字段。其中 `metainfo` 是一个字典，里面包含数据集的元信息，例如类别信息；`data_list` 是一个列表，列表中每个元素是一个字典，该字典定义了一个原始数据（raw data），每个原始数据包含一个或若干个训练/测试样本。

以下是一个 JSON 标注文件的例子:

```
{
    'metainfo':
        {
            'classes': ('person', 'bicycle', 'car', 'motorcycle'),
            ...
        },
    'data_list':
        [
            {
                "img_path": "xxx/xxx_1.jpg",
                "height": 604,
                "width": 640,
                "instances":
                [
                  {
                    "bbox": [0, 0, 10, 20],
                    "bbox_label": 1,
                    "ignore_flag": 0
                  },
                  {
                    "bbox": [10, 10, 110, 120],
                    "bbox_label": 2,
                    "ignore_flag": 0
                  }
                ]
              },
            {
                "img_path": "xxx/xxx_2.jpg",
                "height": 320,
                "width": 460,
                "instances":
```

(下页继续)

(续上页)

```
                [
                  {
                    "bbox": [10, 0, 20, 20],
                    "bbox_label": 3,
                    "ignore_flag": 1
                  }
                ]
              },
            ...
        ]
}
```

有些数据集可能会提供如：crowd/difficult/ignored bboxes 标注，那么我们使用 `ignore_flag` 来包含它们。

在得到上述标准的数据标注格式后，可以直接在配置中使用 MMDetection 的 BaseDetDataset ，而无需进行转换。

自定义数据集例子

假设文本文件中表示的是一种全新的标注格式。边界框的标注信息保存在 `annotation.txt` 中，内容如下：

```
#
000001.jpg
1280 720
2
10 20 40 60 1
20 40 50 60 2
#
000002.jpg
1280 720
3
50 20 40 60 2
20 40 30 45 2
30 40 50 60 3
```

我们可以在 `mmdet/datasets/my_dataset.py` 中创建一个新的 dataset 用以加载数据。

```
import mmengine
from mmdet.base_det_dataset import BaseDetDataset
from mmdet.registry import DATASETS


@DATASETS.register_module()
class MyDataset(BaseDetDataset):
```

(下页继续)

(续上页)

```
    METAINFO = {
        'classes': ('person', 'bicycle', 'car', 'motorcycle'),
         'palette': [(220, 20, 60), (119, 11, 32), (0, 0, 142), (0, 0, 230)]
    }

    def load_data_list(self, ann_file):
        ann_list = mmengine.list_from_file(ann_file)

        data_infos = []
        for i, ann_line in enumerate(ann_list):
            if ann_line != '#':
                continue

            img_shape = ann_list[i + 2].split(' ')
            width = int(img_shape[0])
            height = int(img_shape[1])
            bbox_number = int(ann_list[i + 3])

            instances = []
            for anns in ann_list[i + 4:i + 4 + bbox_number]:
                instance = {}
                instance['bbox'] = [float(ann) for ann in anns.split(' ')[:4]]
                instance['bbox_label']=int(anns[4])
                                instances.append(instance)

            data_infos.append(
                dict(
                    img_path=ann_list[i + 1],
                    img_id=i,
                    width=width,
                    height=height,
                    instances=instances
                ))

        return data_infos
```

配置文件中，可以使用 MyDataset 进行如下修改

```
dataset_A_train = dict(
    type='MyDataset',
    ann_file = 'image_list.txt',
    pipeline=train_pipeline
)
```

6.3.2 使用 dataset 包装器自定义数据集

MMEngine 也支持非常多的数据集包装器（wrapper）来混合数据集或在训练时修改数据集的分布，其支持如下三种数据集包装：

- `RepeatDataset`：将整个数据集简单地重复。
- `ClassBalancedDataset`：以类别均衡的方式重复数据集。
- `ConcatDataset`：合并数据集。

具体使用方式见 MMEngine 数据集包装器。

6.3.3 修改数据集的类别

根据现有数据集的类型，我们可以修改它们的类别名称来训练其标注的子集。例如，如果只想训练当前数据集中的三个类别，那么就可以修改数据集的 `metainfo` 字典，数据集就会自动屏蔽掉其他类别的真实框。

```
classes = ('person', 'bicycle', 'car')
train_dataloader = dict(
    dataset=dict(
        metainfo=dict(classes=classes))
    )
val_dataloader = dict(
    dataset=dict(
        metainfo=dict(classes=classes))
    )
test_dataloader = dict(
    dataset=dict(
        metainfo=dict(classes=classes))
    )
```

注意

- 在 MMDetection v2.5.0 之前，如果类别为集合时数据集将自动过滤掉不包含 GT 的图片，且没办法通过修改配置将其关闭。这是一种不可取的行为而且会引起混淆，因为当类别不是集合时数据集时，只有在 `filter_empty_gt=True` 以及 `test_mode=False` 的情况下才会过滤掉不包含 GT 的图片。在 MMDetection v2.5.0 之后，我们将图片的过滤以及类别的修改进行解耦，数据集只有在 `filter_cfg=dict(filter_empty_gt=True)` 和 `test_mode=False` 的情况下才会过滤掉不包含 GT 的图片，无论类别是否为集合。设置类别只会影响用于训练的标注类别，用户可以自行决定是否过滤不包含 GT 的图片。
- 直接使用 MMEngine 中的 `BaseDataset` 或者 MMDetection 中的 `BaseDetDataset` 时用户不能通过修改配置来过滤不含 GT 的图片，但是可以通过离线的方式来解决。
- 当设置数据集中的 `classes` 时，记得修改 `num_classes`。从 v2.9.0 (PR#4508) 之后，我们实现了 NumClassCheckHook 来检查类别数是否一致。

6.3.4 COCO 全景分割数据集

现在我们也支持 COCO Panoptic Dataset，全景注释的格式与 COCO 格式不同，其前景和背景都将存在于注释文件中。COCO Panoptic 格式的注释 JSON 文件具有以下必要的键：

```
'images': [
    {
        'file_name': '000000001268.jpg',
        'height': 427,
        'width': 640,
        'id': 1268
    },
    ...
]

'annotations': [
    {
        'filename': '000000001268.jpg',
        'image_id': 1268,
        'segments_info': [
            {
                'id':8345037,  # One-to-one correspondence with the id in the␣
↪annotation map.
                'category_id': 51,
                'iscrowd': 0,
                'bbox': (x1, y1, w, h),  # The bbox of the background is the outer␣
↪rectangle of its mask.
                'area': 24315
            },
            ...
        ]
    },
    ...
]

'categories': [  # including both foreground categories and background categories
    {'id': 0, 'name': 'person'},
    ...
 ]
```

此外，`seg` 必须设置为全景注释图像的路径。

```
dataset_type = 'CocoPanopticDataset'
data_root='path/to/your/'
```

(下页继续)

(续上页)

```
train_dataloader = dict(
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        data_prefix=dict(
            img='train/image_data/', seg='train/panoptic/image_annotation_data/')
    )
)
val_dataloader = dict(
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        data_prefix=dict(
            img='val/image_data/', seg='val/panoptic/image_annotation_data/')
    )
)
test_dataloader = dict(
    dataset=dict(
        type=dataset_type,
        data_root=data_root,
        data_prefix=dict(
            img='test/image_data/', seg='test/panoptic/image_annotation_data/')
    )
)
```

6.4 自定义数据预处理流程

1. 在任意文件里写一个新的流程，例如在 `my_pipeline.py`，它以一个字典作为输入并且输出一个字典：

```
import random
from mmcv.transforms import BaseTransform
from mmdet.registry import TRANSFORMS


@TRANSFORMS.register_module()
class MyTransform(BaseTransform):
    """Add your transform

    Args:
        p (float): Probability of shifts. Default 0.5.
    """
```

(下页继续)

(续上页)

```
    def __init__(self, prob=0.5):
        self.prob = prob

    def transform(self, results):
        if random.random() > self.prob:
            results['dummy'] = True
        return results
```

2. 在配置文件里调用并使用你写的数据处理流程，需要确保你的训练脚本能够正确导入新增模块：

```
custom_imports = dict(imports=['path.to.my_pipeline'], allow_failed_imports=False)

train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(type='Resize', scale=(1333, 800), keep_ratio=True),
    dict(type='RandomFlip', prob=0.5),
    dict(type='MyTransform', prob=0.2),
    dict(type='PackDetInputs')
]
```

3. 可视化数据增强处理流程的结果

   如果想要可视化数据增强处理流程的结果，可以使用 `tools/misc/browse_dataset.py` 直观地浏览检测数据集（图像和标注信息），或将图像保存到指定目录。使用方法请参考可视化文档

6.5 自定义训练配置

6.5.1 自定义优化相关的配置

优化相关的配置现在已全部集成到 `optim_wrapper` 中，通常包含三个域：`optimizer`, `paramwise_cfg`, `clip_grad`，具体细节见 OptimWrapper。下面这个例子中，使用了 `AdamW` 作为优化器，主干部分的学习率缩小到原来的十分之一，以及添加了梯度裁剪。

```
optim_wrapper = dict(
    type='OptimWrapper',
    # 优化器
    optimizer=dict(
        type='AdamW',
        lr=0.0001,
        weight_decay=0.05,
        eps=1e-8,
```

(下页继续)

(续上页)

```
        betas=(0.9, 0.999)),

    # 参数层面的学习率和正则化设置
    paramwise_cfg=dict(
        custom_keys={
            'backbone': dict(lr_mult=0.1, decay_mult=1.0),
        },
        norm_decay_mult=0.0),

    # 梯度裁剪
    clip_grad=dict(max_norm=0.01, norm_type=2))
```

自定义 Pytorch 中优化器设置

我们已经支持了 Pytorch 中实现的所有优化器，要使用这些优化器唯一要做就是修改配置文件中的 optimi_wrapper 中的 optimzer 域。比如，如果想要使用 ADAM 作为优化器（可能会导致性能下降），所需要做的修改如下。

```
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=dict(type='Adam', lr=0.0003, weight_decay=0.0001))
```

要修改模型的学习率，用户只需要修改 optimizer 中的 lr 域。用户可以直接参考 PyToch 的 API doc 来进行参数的设置。

自定义优化器

1. 定义一个新优化器

自定义优化器可以定义的方式如下：

假设你想要添加一个名为 MyOptimizer 的优化器，它包含三个参数 a，b，c。你需要新建一个名为 mmdet/engine/optimizers 的文件夹。然后在文件（比如，mmdet/engine/optimizers/my_optimizer.py）实现一个新的优化器。

```
from mmdet.registry import OPTIMIZERS
from torch.optim import Optimizer


@OPTIMIZERS.register_module()
class MyOptimizer(Optimizer):

```

(下页继续)

(续上页)

```
    def __init__(self, a, b, c)

```

2. 导入自定义的优化器

为了能找到上面的所定义的模块，这个模块必须要先导入到主命名空间中。有两种方式可以实现这一点。

- 修改 mmdet/engine/optimizers/__init__.py 来导入模块。

  新定义的模块必须导入到 mmdet/engine/optimizers/__init__.py，这样注册器才能找到该模块并添加它。

```
from .my_optimizer import MyOptimizer
```

- 在配置文件使用 custom_imports 来手动导入模块。

```
custom_imports = dict(imports=['mmdet.engine.optimizers.my_optimizer'], allow_failed_
↪imports=False)
```

mmdet.engine.optimizers.my_optimizer 模块将在程序开始时导入，之后 MyOptimizer 类会被自动注册。注意：应该导入 MyOptimizer 所在的文件，即 mmdet.engine.optimizers.my_optimizer，而不是 mmdet.engine.optimizers.my_optimizer.MyOptimizer。

实际上，用户也可以在别的目录结构下来进行导入模块，只要改模块可以在 PYTHONPATH 中找到。

3. 在配置文件中指定优化器

接下来，你可以在配置文件中的 optim_wrapper 域中的中 optimizer 域中设置你实现的优化器 MyOptimizer。在配置文件中，优化器在 optimizer 域中的配置方式如下：

```
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001))
```

为了使用你的优化器，可以进行如下修改

```
optim_wrapper = dict(
    type='OptimWrapper',
    optimizer=dict(type='MyOptimizer', a=a_value, b=b_value, c=c_value))
```

自定义优化器包装构造类

一些模型可能存在一些特定参数的优化设置，比如，BN 层的权重衰减。用户可以通过自定义优化器包装构造类来实现这些精细化的参数调整。

```
from mmengine.optim import DefaultOptiWrapperConstructor

from mmdet.registry import OPTIM_WRAPPER_CONSTRUCTORS
from .my_optimizer import MyOptimizer


@OPTIM_WRAPPER_CONSTRUCTORS.register_module()
class MyOptimizerWrapperConstructor(DefaultOptimWrapperConstructor):

    def __init__(self,
                 optim_wrapper_cfg: dict,
                 paramwise_cfg: Optional[dict] = None):

    def __call__(self, model: nn.Module) -> OptimWrapper:

        return optim_wrapper
```

优化器包装构造类的具体实现见这里，用户以它为模板，来实现新的优化器包装构造类。

额外的设置

一些没有被优化器实现的技巧（比如，参数层面的学习率设置）应该通过优化器包装构造类来实现或者钩子。我们列出了一些常用的设置用于稳定训练或者加速训练。请随意创建 PR，发布更多设置。

- **使用梯度裁剪来稳定训练**: 一些模型需要进行梯度裁剪来稳定训练过程，例子如下：

  ```
  optim_wrapper = dict(
      _delete_=True, clip_grad=dict(max_norm=35, norm_type=2))
  ```

  如果你的配置已经集成了基础配置（包含了 optim_wrapper 的配置），那么你需要添加 _delete_=True 来覆盖掉不需要的设置。具体见配置相关的文档。

- **使用动量调度加速模型收敛**: 我们支持动量调度器根据学习率修改模型的动量，这可以使模型以更快的方式收敛。动量调度器通常与学习率调度器一起使用，例如 3D 检测 中使用以下配置以加速收敛。更多细节请参考 CosineAnnealingLR 和 CosineAnnealingMomentum 的具体实现。

  ```
  param_scheduler = [
      # 学习率调度器
      # 在前 8 个 epoch，学习率从 0 增大到 lr * 10
  ```

(下页继续)

(续上页)

```
    # 在接下来 12 个 epoch, 学习率从 lr * 10 减小到 lr * 1e-4
    dict(
        type='CosineAnnealingLR',
        T_max=8,
        eta_min=lr * 10,
        begin=0,
        end=8,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingLR',
        T_max=12,
        eta_min=lr * 1e-4,
        begin=8,
        end=20,
        by_epoch=True,
        convert_to_iter_based=True),
    # 动量调度器
    # 在前 8 个 epoch, 动量从 0 增大到 0.85 / 0.95
    # 在接下来 12 个 epoch, 学习率从 0.85 / 0.95 增大到 1
    dict(
        type='CosineAnnealingMomentum',
        T_max=8,
        eta_min=0.85 / 0.95,
        begin=0,
        end=8,
        by_epoch=True,
        convert_to_iter_based=True),
    dict(
        type='CosineAnnealingMomentum',
        T_max=12,
        eta_min=1,
        begin=8,
        end=20,
        by_epoch=True,
        convert_to_iter_based=True)
]
```

6.5.2 自定义训练策略

默认情况下，我们使用 1x 的学习率调整策略，这会条用 MMEngine 中的 MultiStepLR。我们支持许多其他学习率调整策略，具体见这里，例如 `CosineAnnealingLR` 和 `PolyLR` 策略。下面有些例子

- 多项式学习率调整策略:

```
param_scheduler = [
    dict(
        type='PolyLR',
        power=0.9,
        eta_min=1e-4,
        begin=0,
        end=8,
        by_epoch=True)]
```

- 余弦退火学习率调整策略

```
param_scheduler = [
    dict(
        type='CosineAnnealingLR',
        T_max=8,
        eta_min=lr * 1e-5,
        begin=0,
        end=8,
        by_epoch=True)]
```

6.5.3 自定义训练循环

默认情况下，在 `train_cfg` 中使用 `EpochBasedTrainLoop`，并且在每个 epoch 训练之后进行验证，如下所示。

```
train_cfg = dict(type='EpochBasedTrainLoop', max_epochs=12, val_begin=1, val_
↪interval=1)
```

实际上，`IterBasedTrainLoop` 和 [EpochBasedTrainLoop](https://github.com/open-mmlab/mmengine/blob/main/mmengine/runner/loops.py#L18) 支持动态区间的方式进行验证，见下例。

```
# 在第 365001 次迭代之前，我们每 5000 次迭代进行一次评估。
# 在第 365000 次迭代后，我们每 368750 次迭代进行一次评估，
# 这意味着我们在训练结束时进行评估。

interval = 5000
```

(下页继续)

(续上页)

```
max_iters = 368750
dynamic_intervals = [(max_iters // interval * interval + 1, max_iters)]
train_cfg = dict(
    type='IterBasedTrainLoop',
    max_iters=max_iters,
    val_interval=interval,
    dynamic_intervals=dynamic_intervals)
```

6.5.4 自定义钩子

自定义自行实现的钩子

1. 实现一个新的钩子

MMEngine 提供了许多有用的钩子，但在某些情况下用户可能需要实现新的钩子。MMDetection 在 v3.0 中支持自定义钩子。因此，用户可以直接在 mmdet 或其基于 mmdet 的代码库中实现钩子，并通过仅在训练中修改配置来使用钩子。这里我们给出一个在 mmdet 中创建一个新的钩子并在训练中使用它的例子。

```
from mmengine.hooks import Hook
from mmdet.registry import HOOKS


@HOOKS.register_module()
class MyHook(Hook):

    def __init__(self, a, b):

    def before_run(self, runner) -> None:

    def after_run(self, runner) -> None:

    def before_train(self, runner) -> None:

    def after_train(self, runner) -> None:

    def before_train_epoch(self, runner) -> None:

    def after_train_epoch(self, runner) -> None:

    def before_train_iter(self,
                          runner,
                          batch_idx: int,
```

(下页继续)

(续上页)

```
                          data_batch: DATA_BATCH = None) -> None:

    def after_train_iter(self,
                         runner,
                         batch_idx: int,
                         data_batch: DATA_BATCH = None,
                         outputs: Optional[dict] = None) -> None:
```

根据钩子的功能，用户需要在 before_run、after_run、before_train、after_train、before_train_epoch、after_train_epoch、before_train_iter 和 after_train_iter。还有更多可以插入钩子的点，更多细节请参考 base hook class。

2. 注册新钩子

然后我们需要导入 MyHook。假设该文件位于 mmdet/engine/hooks/my_hook.py 中，有两种方法可以做到这一点：

- 修改 mmdet/engine/hooks/__init__.py 以导入它。

  新定义的模块应该在 mmdet/engine/hooks/__init__.py 中导入，以便注册表找到新模块并添加它：

```
from .my_hook import MyHook
```

- 在配置中使用 custom_imports 手动导入它

```
custom_imports = dict(imports=['mmdet.engine.hooks.my_hook'], allow_failed_
↪imports=False)
```

3. 修改配置

```
custom_hooks = [
    dict(type='MyHook', a=a_value, b=b_value)
]
```

你还可以通过修改键 priority 的值为 NORMAL 或 HIGHEST 来设置挂钩的优先级，如下所示

```
custom_hooks = [
    dict(type='MyHook', a=a_value, b=b_value, priority='NORMAL')
]
```

默认情况下，钩子的优先级在注册期间设置为 NORMAL。

使用 MMDetection 中实现的钩子

如果 MMDetection 中已经实现了该钩子，你可以直接修改配置以使用该钩子，如下所示

例子: `NumClassCheckHook`

我们实现了一个名为 NumClassCheckHook 的自定义钩子来检查 `num_classes` 是否在 head 中和 `dataset` 中的 `classes` 的长度相匹配。

我们在 default_runtime.py 中设置它。

```
custom_hooks = [dict(type='NumClassCheckHook')]
```

修改默认运行时钩子

有一些常见的钩子是通过 `default_hooks` 注册的，它们是

- `IterTimerHook`：记录“data_time”用于加载数据和“time”用于模型训练步骤的钩子。
- `LoggerHook`：从 `Runner` 的不同组件收集日志并将它们写入终端、JSON 文件、tensorboard 和 wandb 等的钩子。
- `ParamSchedulerHook`：更新优化器中一些超参数的钩子，例如学习率和动量。
- `CheckpointHook`：定期保存检查点的钩子。
- `DistSamplerSeedHook`：为采样器和批处理采样器设置种子的钩子。
- `DetVisualizationHook`：用于可视化验证和测试过程预测结果的钩子。

`IterTimerHook`、`ParamSchedulerHook` 和 `DistSamplerSeedHook` 很简单，通常不需要修改，所以这里我们将展示如何使用 `LoggerHook`、`CheckpointHook` 和 `DetVisualizationHook`。

CheckpointHook

除了定期保存检查点，`CheckpointHook` 提供了其他选项，例如 `max_keep_ckpts`、`save_optimizer` 等。用户可以设置 `max_keep_ckpts` 只保存少量检查点或通过 `save_optimizer` 决定是否存储优化器的状态字典。参数的更多细节在这里可以找到。

```
default_hooks = dict(
    checkpoint=dict(
        type='CheckpointHook',
        interval=1,
        max_keep_ckpts=3,
        save_optimizer=True))
```

LoggerHook

LoggerHook 可以设置间隔。详细用法可以在 docstring 中找到。

```
default_hooks = dict(logger=dict(type='LoggerHook', interval=50))
```

DetVisualizationHook

DetVisualizationHook 使用 DetLocalVisualizer 来可视化预测结果，DetLocalVisualizer 支持不同的后端，例如 TensorboardVisBackend 和 WandbVisBackend （见 docstring 了解更多细节）。用户可以添加多个后端来进行可视化，如下所示。

```
default_hooks = dict(
    visualization=dict(type='DetVisualizationHook', draw=True))

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

CHAPTER 7

How to

本教程收集了任何如何使用 MMDetection 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!

7.1 使用 MMClassification 的骨干网络

MMDet、MMCls、MMSeg 中的模型注册表都继承自 MMEngine 中的根注册表，允许这些存储库直接使用彼此已经实现的模块。因此用户可以在 MMDetection 中使用来自 MMClassification 的骨干网络，而无需实现 MMClassification 中已经存在的网络。

7.1.1 使用在 MMClassification 中实现的骨干网络

假设想将 `MobileNetV3-small` 作为 `RetinaNet` 的骨干网络，则配置文件如下。

```
_base_ = [
    '../_base_/models/retinanet_r50_fpn.py',
    '../_base_/datasets/coco_detection.py',
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]
# please install mmcls>=1.0.0rc0
# import mmcls.models to trigger register_module in mmcls
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)
pretrained = 'https://download.openmmlab.com/mmclassification/v0/mobilenet_v3/convert/
↪mobilenet_v3_small-8427ecf0.pth'
```

(下页继续)

(续上页)

```
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=pretrained,
            prefix='backbone.')), # MMCls 中骨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。
    # 修改 in_channels
    neck=dict(in_channels=[24, 48, 96], start_level=0))
```

7.1.2 通过 MMClassification 使用 TIMM 中实现的骨干网络

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

```
# https://github.com/open-mmlab/mmdetection/blob/main/configs/timm_example/retinanet_
↪timm_efficientnet_b1_fpn_1x_coco.py
_base_ = [
    '../_base_/models/retinanet_r50_fpn.py',
    '../_base_/datasets/coco_detection.py',
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]

# please install mmcls>=1.0.0rc0
# import mmcls.models to trigger register_module in mmcls
custom_imports = dict(imports=['mmcls.models'], allow_failed_imports=False)
model = dict(
    backbone=dict(
        _delete_=True, # 将 _base_ 中关于 backbone 的字段删除
        type='mmcls.TIMMBackbone', # 使用 mmcls 中 timm 骨干网络
        model_name='efficientnet_b1',
        features_only=True,
        pretrained=True,
        out_indices=(1, 2, 3, 4)), # 修改 out_indices
    neck=dict(in_channels=[24, 40, 112, 320])) # 修改 in_channels

optimizer = dict(type='SGD', lr=0.01, momentum=0.9, weight_decay=0.0001)
```

`type='mmcls.TIMMBackbone'` 表示在 MMDetection 中使用 MMClassification 中的 `TIMMBackbone` 类，并且使用的模型为 `EfficientNet-B1`，其中 `mmcls` 表示 MMClassification 库，而 `TIMMBackbone` 表示 MMClassification 中实现的 TIMMBackbone 包装器。

关于层次注册器的具体原理可以参考 MMEngine 文档，关于如何使用 MMClassification 中的其他 backbone，可以参考 MMClassification 文档。

7.2 使用马赛克数据增强

如果你想在训练中使用 `Mosaic`，那么请确保你同时使用 `MultiImageMixDataset`。以 `Faster R-CNN` 算法为例，你可以通过如下做法实现：

```
# 直接打开 configs/faster_rcnn/faster-rcnn_r50_fpn_1x_coco.py , 增添如下字段
data_root = 'data/coco/'
dataset_type = 'CocoDataset'
img_scale=(1333, 800)

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)), # 图像经过马赛克处理后会放大 4 倍,
所以我们使用仿射变换来恢复图像的大小。
    dict(type='RandomFlip', prob=0.5),
    dict(type='PackDetInputs'))
]

train_dataset = dict(
    _delete_ = True, # 删除不必要的设置
    type='MultiImageMixDataset',
    dataset=dict(
        type=dataset_type,
        ann_file=data_root + 'annotations/instances_train2017.json',
        img_prefix=data_root + 'train2017/',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations', with_bbox=True)
        ],
        filter_empty_gt=False,
    ),
    pipeline=train_pipeline
    )
```

(下页继续)

(续上页)

```
data = dict(
    train=train_dataset
    )
```

7.3 在配置文件中冻结骨干网络后在训练中解冻骨干网络

如果你在配置文件中已经冻结了骨干网络并希望在几个训练周期后解冻它，你可以通过 hook 来实现这个功能。以用 ResNet 为骨干网络的 Faster R-CNN 为例，你可以冻结一个骨干网络的一个层并在配置文件中添加如下 `custom_hooks`:

```
_base_ = [
    '../_base_/models/faster-rcnn_r50_fpn.py',
    '../_base_/datasets/coco_detection.py',
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]
model = dict(
    # freeze one stage of the backbone network.
    backbone=dict(frozen_stages=1),
)
custom_hooks = [dict(type="UnfreezeBackboneEpochBasedHook", unfreeze_epoch=1)]
```

同时在 `mmdet/core/hook/unfreeze_backbone_epoch_based_hook.py` 当中书写 `UnfreezeBackboneEpochBasedHook` 类

```
from mmengine.model import is_model_wrapper
from mmengine.hooks import Hook
from mmdet.registry import HOOKS


@HOOKS.register_module()
class UnfreezeBackboneEpochBasedHook(Hook):
    """Unfreeze backbone network Hook.

    Args:
        unfreeze_epoch (int): The epoch unfreezing the backbone network.
    """

    def __init__(self, unfreeze_epoch=1):
        self.unfreeze_epoch = unfreeze_epoch

    def before_train_epoch(self, runner):
        # Unfreeze the backbone network.
```

(下页继续)

(续上页)

```
        # Only valid for resnet.
        if runner.epoch == self.unfreeze_epoch:
            model = runner.model
            if is_module_wrapper(model):
                model = model.module
            backbone = model.backbone
            if backbone.frozen_stages >= 0:
                if backbone.deep_stem:
                    backbone.stem.train()
                    for param in backbone.stem.parameters():
                        param.requires_grad = True
                else:
                    backbone.norm1.train()
                    for m in [backbone.conv1, backbone.norm1]:
                        for param in m.parameters():
                            param.requires_grad = True

            for i in range(1, backbone.frozen_stages + 1):
                m = getattr(backbone, f'layer{i}')
                m.train()
                for param in m.parameters():
                    param.requires_grad = True
```

7.4 获得新的骨干网络的通道数

如果你想获得一个新骨干网络的通道数，你可以单独构建这个骨干网络并输入一个伪造的图片来获取每一个阶段的输出。

以 ResNet 为例：

```
from mmdet.models import ResNet
import torch
self = ResNet(depth=18)
self.eval()
inputs = torch.rand(1, 3, 32, 32)
level_outputs = self.forward(inputs)
for level_out in level_outputs:
    print(tuple(level_out.shape))
```

以上脚本的输出为:

```
(1, 64, 8, 8)
(1, 128, 4, 4)
(1, 256, 2, 2)
(1, 512, 1, 1)
```

用户可以通过将脚本中的 `ResNet(depth=18)` 替换为自己的骨干网络配置来得到新的骨干网络的通道数。

7.5 MMDetection 中训练 Detectron2 的模型

用户可以使用 `Detectron2Wrapper` 从而在 MMDetection 中使用 Detectron2 的模型。我们提供了 Faster R-CNN, Mask R-CNN 和 RetinaNet 的示例来在 MMDetection 中训练/测试 Detectron2 的模型。

使用过程中需要注意配置文件中算法组件要和 Detectron2 中的相同。模型初始化时，我们首先初始化 Detectron2 的默认设置，然后配置文件中的设置将覆盖默认设置，模型将基于更新过的设置来建立。输入数据首先转换成 Detectron2 的类型并输入进 Detectron2 的模型中。在推理阶段，Detectron2 的模型结果将会转换回 MMDetection 的类型。

7.5.1 使用 Detectron2 的预训练权重

`Detectron2Wrapper` 中的权重初始化将不使用 MMDetection 的逻辑。用户可以设置 `model.d2_detector.weights=xxx` 来加载预训练的权重。例如，我们可以使用 `model.d2_detector.weights='detectron2://ImageNetPretrained/MSRA/R-50.pkl'` 来加载 ResNet-50 的预训练权重，或者使用 `model.d2_detector.weights='detectron2://COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_1x/137260431/model_final_a54504.pkl'` 来加载 Detectron2 中提出的预训练的 Mask R-CNN 权重。

注意： 不能直接使用 `load_from` 来加载 Detectron2 的预训练模型，但可以通过 `tools/model_converters/detectron2_to_mmdet.py` 先对该预训练模型进行转换。

在测试时，用户应该首先使用 `tools/model_converters/detectron2_to_mmdet.py` 将 Detectron2 的预训练权重转换为 MMDetection 可读取的结构。

```
python tools/model_converters/detectron2_to_mmdet.py ${Detectron2 ckpt path} $
↪{MMDetectron ckpt path}。
```

CHAPTER 8

从 MMDetection 2.x 迁移至 3.x

MMDetection 3.x 版本是一个重大更新，包含了许多 API 和配置文件的变化。本文档旨在帮助用户从 MMDetection 2.x 版本迁移到 3.x 版本。我们将迁移指南分为以下几个部分：

- 配置文件迁移
- API 和 Registry 迁移
- 数据集迁移
- 模型迁移
- 常见问题

如果您在迁移过程中遇到任何问题，欢迎在 issue 中提出。我们也欢迎您为本文档做出贡献。

CHAPTER 9

mmdet.apis

CHAPTER 10

mmdet.datasets

10.1 datasets

10.2 api_wrappers

class mmdet.datasets.api_wrappers.**COCO**(**args: Any*, ***kwargs: Any*)

This class is almost the same as official pycocotools package.

It implements some snake case function aliases. So that the COCO class has the same interface as LVIS class.

class mmdet.datasets.api_wrappers.**COCOPanoptic**(**args: Any*, ***kwargs: Any*)

This wrapper is for loading the panoptic style annotation file.

The format is shown in the CocoPanopticDataset class.

参数 **annotation_file** (*str, optional*) –Path of annotation file. Defaults to None.

createIndex() → None

Create index.

load_anns(*ids: Union[List[int], int] = []*) → Optional[List[dict]]

Load anns with the specified ids.

`self.anns` is a list of annotation lists instead of a list of annotations.

参数 **ids** (*Union[List[int], int]*) –Integer ids specifying anns.

返回 Loaded ann objects.

返回类型 anns (List[dict], optional)

10.3 samplers

class mmdet.datasets.samplers.**AspectRatioBatchSampler**(*sampler: torch.utils.data.sampler.Sampler*, *batch_size: int*, *drop_last: bool = False*)

A sampler wrapper for grouping images with similar aspect ratio (< 1 or.

>= 1) into a same batch.

参数

- **sampler** (*Sampler*) –Base sampler.
- **batch_size** (*int*) –Size of mini-batch.
- **drop_last** (*bool*) –If `True`, the sampler will drop the last batch if its size would be less than `batch_size`.

class mmdet.datasets.samplers.**ClassAwareSampler**(*dataset: mmengine.dataset.base_dataset.BaseDataset*, *seed: Optional[int] = None*, *num_sample_class: int = 1*)

Sampler that restricts data loading to the label of the dataset.

A class-aware sampling strategy to effectively tackle the non-uniform class distribution. The length of the training data is consistent with source data. Simple improvements based on Relay Backpropagation for Effective Learning of Deep Convolutional Neural Networks

The implementation logic is referred to https://github.com/Sense-X/TSD/blob/master/mmdet/datasets/samplers/distributed_classaware_sampler.py

参数

- **dataset** –Dataset used for sampling.
- **seed** (*int, optional*) –random seed used to shuffle the sampler. This number should be identical across all processes in the distributed group. Defaults to None.
- **num_sample_class** (*int*) –The number of samples taken from each per-label list. Defaults to 1.

get_cat2imgs() → Dict[int, list]

Get a dict with class as key and img_ids as values.

返回 A dict of per-label image list, the item of the dict indicates a label index, corresponds to the image index that contains the label.

返回类型 dict[int, list]

set_epoch(*epoch: int*) → None

Sets the epoch for this sampler.

When `shuffle=True`, this ensures all replicas use a different random ordering for each epoch. Otherwise, the next iteration of this sampler will yield the same ordering.

参数 **epoch** (*int*) –Epoch number.

class mmdet.datasets.samplers.**GroupMultiSourceSampler**(*dataset: mmengine.dataset.base_dataset.BaseDataset*, *batch_size: int*, *source_ratio: List[Union[int, float]]*, *shuffle: bool = True*, *seed: Optional[int] = None*)

Group Multi-Source Infinite Sampler.

According to the sampling ratio, sample data from different datasets but the same group to form batches.

参数

- **dataset** (*Sized*) –The dataset.
- **batch_size** (*int*) –Size of mini-batch.
- **source_ratio** (*list[int | float]*) –The sampling ratio of different source datasets in a mini-batch.
- **shuffle** (*bool*) –Whether shuffle the dataset or not. Defaults to True.
- **seed** (*int, optional*) –Random seed. If None, set a random seed. Defaults to None.

class mmdet.datasets.samplers.**MultiSourceSampler**(*dataset: Sized*, *batch_size: int*, *source_ratio: List[Union[int, float]]*, *shuffle: bool = True*, *seed: Optional[int] = None*)

Multi-Source Infinite Sampler.

According to the sampling ratio, sample data from different datasets to form batches.

参数

- **dataset** (*Sized*) –The dataset.
- **batch_size** (*int*) –Size of mini-batch.
- **source_ratio** (*list[int | float]*) –The sampling ratio of different source datasets in a mini-batch.
- **shuffle** (*bool*) –Whether shuffle the dataset or not. Defaults to True.
- **seed** (*int, optional*) –Random seed. If None, set a random seed. Defaults to None.

实际案例

```
>>> dataset_type = 'ConcatDataset'
>>> sub_dataset_type = 'CocoDataset'
>>> data_root = 'data/coco/'
>>> sup_ann = '../coco_semi_annos/instances_train2017.1@10.json'
>>> unsup_ann = '../coco_semi_annos/' \
>>>              'instances_train2017.1@10-unlabeled.json'
>>> dataset = dict(type=dataset_type,
>>>     datasets=[
>>>         dict(
>>>             type=sub_dataset_type,
>>>             data_root=data_root,
>>>             ann_file=sup_ann,
>>>             data_prefix=dict(img='train2017/'),
>>>             filter_cfg=dict(filter_empty_gt=True, min_size=32),
>>>             pipeline=sup_pipeline),
>>>         dict(
>>>             type=sub_dataset_type,
>>>             data_root=data_root,
>>>             ann_file=unsup_ann,
>>>             data_prefix=dict(img='train2017/'),
>>>             filter_cfg=dict(filter_empty_gt=True, min_size=32),
>>>             pipeline=unsup_pipeline),
>>>         ])
>>>     train_dataloader = dict(
>>>         batch_size=5,
>>>         num_workers=5,
>>>         persistent_workers=True,
>>>         sampler=dict(type='MultiSourceSampler',
>>>             batch_size=5, source_ratio=[1, 4]),
>>>         batch_sampler=None,
>>>         dataset=dataset)
```

set_epoch(*epoch: int*) → None

Not supported in ‘epoch-based runner.

10.4 transforms

CHAPTER 11

mmdet.engine

11.1 hooks

class mmdet.engine.hooks.**CheckInvalidLossHook**(*interval: int = 50*)

Check invalid loss hook.

This hook will regularly check whether the loss is valid during training.

参数 **interval** (*int*) –Checking interval (every k iterations). Default: 50.

after_train_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: Optional[dict] = None*, *outputs: Optional[dict] = None*) → None

Regularly check whether the loss is valid every n iterations.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*dict, Optional*) –Data from dataloader. Defaults to None.
- **outputs** (*dict, Optional*) –Outputs from model. Defaults to None.

class mmdet.engine.hooks.**DetVisualizationHook**(*draw: bool = False*, *interval: int = 50*, *score_thr: float = 0.3*, *show: bool = False*, *wait_time: float = 0.0*, *test_out_dir: Optional[str] = None*, *backend_args: Optional[dict] = None*)

Detection Visualization Hook. Used to visualize validation and testing process prediction results.

In the testing phase:

1. **If `show` is True, it means that only the prediction results are** visualized without storing data, so `vis_backends` needs to be excluded.
2. **If `test_out_dir` is specified, it means that the prediction results** need to be saved to `test_out_dir`. In order to avoid vis_backends also storing data, so `vis_backends` needs to be excluded.
3. **`vis_backends` takes effect if the user does not specify `show`** and *test_out_dir*'. You can set `vis_backends` to WandbVisBackend or TensorboardVisBackend to store the prediction result in Wandb or Tensorboard.

参数

- **`draw`** (*bool*) –whether to draw prediction results. If it is False, it means that no drawing will be done. Defaults to False.
- **`interval`** (*int*) –The interval of visualization. Defaults to 50.
- **`score_thr`** (*float*) –The threshold to visualize the bboxes and masks. Defaults to 0.3.
- **`show`** (*bool*) –Whether to display the drawn image. Default to False.
- **`wait_time`** (*float*) –The interval of show (s). Defaults to 0.
- **`test_out_dir`** (*str, optional*) –directory where painted images will be saved in testing process.
- **`backend_args`** (*dict, optional*) –Arguments to instantiate the corresponding backend. Defaults to None.

`after_test_iter`(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs: Sequence[*mmdet.structures.det_data_sample.DetDataSample*]*) → None

Run after every testing iterations.

参数

- **`runner`** (`Runner`) –The runner of the testing process.
- **`batch_idx`** (*int*) –The index of the current batch in the val loop.
- **`data_batch`** (*dict*) –Data from dataloader.
- **`outputs`** (Sequence[`DetDataSample`]) –A batch of data samples that contain annotations and predictions.

`after_val_iter`(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: dict*, *outputs: Sequence[*mmdet.structures.det_data_sample.DetDataSample*]*) → None

Run after every `self.interval` validation iterations.

参数

- **runner** (Runner) –The runner of the validation process.
- **batch_idx** (*int*) –The index of the current batch in the val loop.
- **data_batch** (*dict*) –Data from dataloader.
- **outputs** (Sequence[DetDataSample]]) –A batch of data samples that contain annotations and predictions.

class mmdet.engine.hooks.**MeanTeacherHook**(*momentum: float = 0.001*, *interval: int = 1*, *skip_buffer=True*)

Mean Teacher Hook.

Mean Teacher is an efficient semi-supervised learning method in Mean Teacher. This method requires two models with exactly the same structure, as the student model and the teacher model, respectively. The student model updates the parameters through gradient descent, and the teacher model updates the parameters through exponential moving average of the student model. Compared with the student model, the teacher model is smoother and accumulates more knowledge.

参数

- **momentum** (*float*) –

  The momentum used for updating teacher's parameter. Teacher's parameter are updated with the formula:

  teacher = (1-momentum) * teacher + momentum * student. Defaults to 0.001.

- **interval** (*int*) –Update teacher's parameter every interval iteration. Defaults to 1.
- **skip_buffers** (*bool*) –Whether to skip the model buffers, such as batchnorm running stats (running_mean, running_var), it does not perform the ema operation. Default to True.

after_train_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: Optional[dict] = None*, *outputs: Optional[dict] = None*) → None

Update teacher's parameter every self.interval iterations.

before_train(*runner: mmengine.runner.runner.Runner*) → None

To check that teacher model and student model exist.

momentum_update(*model: torch.nn.modules.module.Module*, *momentum: float*) → None

Compute the moving average of the parameters using exponential moving average.

class mmdet.engine.hooks.**MemoryProfilerHook**(*interval: int = 50*)

Memory profiler hook recording memory information including virtual memory, swap memory, and the memory of the current process.

参数 **interval** (*int*) –Checking interval (every k iterations). Default: 50.

after_test_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: Optional[dict] = None*, *outputs: Optional[Sequence[mmdet.structures.det_data_sample.DetDataSample]] = None*) → None

Regularly record memory information.

参数

- **runner** (Runner) –The runner of the testing process.
- **batch_idx** (*int*) –The index of the current batch in the test loop.
- **data_batch** (*dict, optional*) –Data from dataloader. Defaults to None.
- **outputs** (Sequence[DetDataSample], optional) –Outputs from model. Defaults to None.

after_train_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: Optional[dict] = None*, *outputs: Optional[dict] = None*) → None

Regularly record memory information.

参数

- **runner** (Runner) –The runner of the training process.
- **batch_idx** (*int*) –The index of the current batch in the train loop.
- **data_batch** (*dict, optional*) –Data from dataloader. Defaults to None.
- **outputs** (*dict, optional*) –Outputs from model. Defaults to None.

after_val_iter(*runner: mmengine.runner.runner.Runner*, *batch_idx: int*, *data_batch: Optional[dict] = None*, *outputs: Optional[Sequence[mmdet.structures.det_data_sample.DetDataSample]] = None*) → None

Regularly record memory information.

参数

- **runner** (Runner) –The runner of the validation process.
- **batch_idx** (*int*) –The index of the current batch in the val loop.
- **data_batch** (*dict, optional*) –Data from dataloader. Defaults to None.
- **outputs** (Sequence[DetDataSample], optional) –Outputs from model. Defaults to None.

class mmdet.engine.hooks.**NumClassCheckHook**

Check whether the *num_classes* in head matches the length of *classes* in *dataset.metainfo*.

before_train_epoch(*runner: mmengine.runner.runner.Runner*) → None

Check whether the training dataset is compatible with head.

参数 **runner** (Runner) –The runner of the training or evaluation process.

before_val_epoch(*runner: mmengine.runner.runner.Runner*) → None

Check whether the dataset in val epoch is compatible with head.

参数 **runner** (Runner) –The runner of the training or evaluation process.

class mmdet.engine.hooks.**PipelineSwitchHook**(*switch_epoch*, *switch_pipeline*)

Switch data pipeline at switch_epoch.

参数

- **switch_epoch** (*int*) –switch pipeline at this epoch.
- **switch_pipeline** (*list[dict]*) –the pipeline to switch to.

before_train_epoch(*runner*)

switch pipeline.

class mmdet.engine.hooks.**SetEpochInfoHook**

Set runner’s epoch information to the model.

before_train_epoch(*runner*)

All subclasses should override this method, if they need any operations before each training epoch.

参数 **runner** (*Runner*) –The runner of the training process.

class mmdet.engine.hooks.**SyncNormHook**

Synchronize Norm states before validation, currently used in YOLOX.

before_val_epoch(*runner*)

Synchronizing norm.

class mmdet.engine.hooks.**YOLOXModeSwitchHook**(*num_last_epochs: int = 15*, *skip_type_keys: Sequence[str] = ('Mosaic', 'RandomAffine', 'MixUp')*)

Switch the mode of YOLOX during training.

This hook turns off the mosaic and mixup data augmentation and switches to use L1 loss in bbox_head.

参数 **num_last_epochs** –The number of latter epochs in the end of the training to close the data augmentation and switch to L1 loss. Defaults to 15.

before_train_epoch(*runner*) → None

Close mosaic and mixup augmentation and switches to use L1 loss.

11.2 optimizers

class mmdet.engine.optimizers.**LearningRateDecayOptimizerConstructor**(*optim_wrapper_cfg: dict*, *paramwise_cfg: Optional[dict] = None*)

add_params(*params: List[dict]*, *module: torch.nn.modules.module.Module*, ***kwargs*) → None

Add all parameters of module to the params list.

The parameters of the given module will be added to the list of param groups, with specific rules defined by paramwise_cfg.

参数

- **params** (*list[dict]*) –A list of param groups, it will be modified in place.
- **module** (*nn.Module*) –The module to be added.

11.3 runner

class mmdet.engine.runner.**TeacherStudentValLoop**(*runner*, *dataloader: Union[torch.utils.data.dataloader.DataLoader, Dict]*, *evaluator: Union[mmengine.evaluator.evaluator.Evaluator, Dict, List]*, *fp16: bool = False*)

Loop for validation of model teacher and student.

run()

Launch validation for model teacher and student.

11.4 schedulers

class mmdet.engine.schedulers.**QuadraticWarmupLR**(*optimizer*, **args*, ***kwargs*)

Warm up the learning rate of each parameter group by quadratic formula.

参数

- **optimizer** (*Optimizer*) –Wrapped optimizer.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.

- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

class mmdet.engine.schedulers.**QuadraticWarmupMomentum**(*optimizer*, **args*, ***kwargs*)

Warm up the momentum value of each parameter group by quadratic formula.

参数

- **optimizer** (*Optimizer*) –Wrapped optimizer.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.
- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

class mmdet.engine.schedulers.**QuadraticWarmupParamScheduler**(*optimizer: torch.optim.optimizer.Optimizer*, *param_name: str*, *begin: int = 0*, *end: int = 1000000000*, *last_step: int = - 1*, *by_epoch: bool = True*, *verbose: bool = False*)

Warm up the parameter value of each parameter group by quadratic formula:

$$X_t = X_{t-1} + \frac{2t+1}{(end-begin)^2} \times X_{base}$$

参数

- **optimizer** (*Optimizer*) –Wrapped optimizer.
- **param_name** (*str*) –Name of the parameter to be adjusted, such as `lr`, `momentum`.
- **begin** (*int*) –Step at which to start updating the parameters. Defaults to 0.
- **end** (*int*) –Step at which to stop updating the parameters. Defaults to INF.
- **last_step** (*int*) –The index of last step. Used for resume without state dict. Defaults to -1.

- **by_epoch** (*bool*) –Whether the scheduled parameters are updated by epochs. Defaults to True.
- **verbose** (*bool*) –Whether to print the value for each update. Defaults to False.

classmethod build_iter_from_epoch(**args*, *begin=0*, *end=1000000000*, *by_epoch=True*, *epoch_length=None*, ***kwargs*)

Build an iter-based instance of this scheduler from an epoch-based config.

CHAPTER 12

mmdet.evaluation

12.1 functional

mmdet.evaluation.functional.**average_precision**(*recalls*, *precisions*, *mode='area'*)

Calculate average precision (for single or multiple scales).

参数

- **recalls** (*ndarray*) –shape (num_scales, num_dets) or (num_dets,)
- **precisions** (*ndarray*) –shape (num_scales, num_dets) or (num_dets,)
- **mode** (*str*) – 'area' or '11points', 'area' means calculating the area under precision-recall curve, '11points' means calculating the average precision of recalls at [0, 0.1, ···, 1]

返回 calculated average precision

返回类型 float or ndarray

mmdet.evaluation.functional.**bbox_overlaps**(*bboxes1*, *bboxes2*, *mode='iou'*, *eps=1e-06*, *use_legacy_coordinate=False*)

Calculate the ious between each bbox of bboxes1 and bboxes2.

参数

- **bboxes1** (*ndarray*) –Shape (n, 4)
- **bboxes2** (*ndarray*) –Shape (k, 4)
- **mode** (*str*) –IOU (intersection over union) or IOF (intersection over foreground)

- **use_legacy_coordinate** (*bool*) –Whether to use coordinate system in mmdet v1.x. which means width, height should be calculated as 'x2 - x1 + 1' and 'y2 - y1 + 1' respectively. Note when function is used in *VOCDataset*, it should be True to align with the official implementation *http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCdevkit_18-May-2011.tar* Default: False.

返回 Shape (n, k)

返回类型 ious (ndarray)

mmdet.evaluation.functional.**cityscapes_classes**() → list

Class names of Cityscapes.

mmdet.evaluation.functional.**coco_classes**() → list

Class names of COCO.

mmdet.evaluation.functional.**coco_panoptic_classes**() → list

Class names of COCO panoptic.

mmdet.evaluation.functional.**eval_map**(*det_results*, *annotations*, *scale_ranges=None*, *iou_thr=0.5*, *ioa_thr=None*, *dataset=None*, *logger=None*, *tpfp_fn=None*, *nproc=4*, *use_legacy_coordinate=False*, *use_group_of=False*, *eval_mode='area'*)

Evaluate mAP of a dataset.

参数

- **det_results** (*list[list]*) –[[cls1_det, cls2_det, ···], ···]. The outer list indicates images, and the inner list indicates per-class detected bboxes.
- **annotations** (*list[dict]*) –Ground truth annotations where each item of the list indicates an image. Keys of annotations are:
  - *bboxes*: numpy array of shape (n, 4)
  - *labels*: numpy array of shape (n,)
  - *bboxes_ignore* (optional): numpy array of shape (k, 4)
  - *labels_ignore* (optional): numpy array of shape (k,)
- **scale_ranges** (*list[tuple] | None*) –Range of scales to be evaluated, in the format [(min1, max1), (min2, max2), ···]. A range of (32, 64) means the area range between (32**2, 64**2). Defaults to None.
- **iou_thr** (*float*) –IoU threshold to be considered as matched. Defaults to 0.5.
- **ioa_thr** (*float | None*) –IoA threshold to be considered as matched, which only used in OpenImages evaluation. Defaults to None.
- **dataset** (*list[str] | str | None*) –Dataset name or dataset classes, there are minor differences in metrics for different datasets, e.g. "voc", "imagenet_det", etc. Defaults

to None.

- **logger** (*logging.Logger | str | None*) –The way to print the mAP summary. See *mmengine.logging.print_log()* for details. Defaults to None.
- **tpfp_fn** (*callable | None*) –The function used to determine true/ false positives. If None, `tpfp_default()` is used as default unless dataset is 'det' or 'vid' (`tpfp_imagenet()` in this case). If it is given as a function, then this function is used to evaluate tp & fp. Default None.
- **nproc** (*int*) –Processes used for computing TP and FP. Defaults to 4.
- **use_legacy_coordinate** (*bool*) –Whether to use coordinate system in mmdet v1.x. which means width, height should be calculated as 'x2 - x1 + 1' and 'y2 - y1 + 1' respectively. Defaults to False.
- **use_group_of** (*bool*) –Whether to use group of when calculate TP and FP, which only used in OpenImages evaluation. Defaults to False.
- **eval_mode** (*str*) – 'area' or '11points', 'area' means calculating the area under precision-recall curve, '11points' means calculating the average precision of recalls at [0, 0.1, ···, 1], PASCAL VOC2007 uses *11points* as default evaluate mode, while others are 'area'. Defaults to 'area'.

返回 (mAP, [dict, dict, ···])

返回类型 tuple

mmdet.evaluation.functional.**eval_recalls**(*gts*, *proposals*, *proposal_nums=None*, *iou_thrs=0.5*, *logger=None*, *use_legacy_coordinate=False*)

Calculate recalls.

参数

- **gts** (*list[ndarray]*) –a list of arrays of shape (n, 4)
- **proposals** (*list[ndarray]*) –a list of arrays of shape (k, 4) or (k, 5)
- **proposal_nums** (*int | Sequence[int]*) –Top N proposals to be evaluated.
- **iou_thrs** (*float | Sequence[float]*) –IoU thresholds. Default: 0.5.
- **logger** (*logging.Logger | str | None*) –The way to print the recall summary. See *mmengine.logging.print_log()* for details. Default: None.
- **use_legacy_coordinate** (*bool*) –Whether use coordinate system in mmdet v1.x. "1" was added to both height and width which means w, h should be computed as 'x2 - x1 + 1' and 'y2 - y1 + 1'. Default: False.

返回 recalls of different ious and proposal nums

返回类型 ndarray

mmdet.evaluation.functional.**evaluateImgLists**(*prediction_list: list*, *groundtruth_list: list*, *args: object*, *backend_args: Optional[dict] = None*, *dump_matches: bool = False*) → dict

A wrapper of obj:“cityscapesscripts.evaluation.

evalInstanceLevelSemanticLabeling.evaluateImgLists“. Support loading groundtruth image from file backend. :param prediction_list: A list of prediction txt file. :type prediction_list: list :param groundtruth_list: A list of groundtruth image file. :type groundtruth_list: list :param args: A global object setting in

obj:cityscapesscripts.evaluation. evalInstanceLevelSemanticLabeling

参数

- **backend_args** (*dict, optional*) –Arguments to instantiate the preifx of uri corresponding backend. Defaults to None.
- **dump_matches** (*bool*) –whether dump matches.json. Defaults to False.

返回 The computed metric.

返回类型 dict

mmdet.evaluation.functional.**get_classes**(*dataset*) → list

Get class names of a dataset.

mmdet.evaluation.functional.**imagenet_det_classes**() → list

Class names of ImageNet Det.

mmdet.evaluation.functional.**imagenet_vid_classes**() → list

Class names of ImageNet VID.

mmdet.evaluation.functional.**objects365v1_classes**() → list

Class names of Objects365 V1.

mmdet.evaluation.functional.**objects365v2_classes**() → list

Class names of Objects365 V2.

mmdet.evaluation.functional.**oid_challenge_classes**() → list

Class names of Open Images Challenge.

mmdet.evaluation.functional.**oid_v6_classes**() → list

Class names of Open Images V6.

mmdet.evaluation.functional.**plot_iou_recall**(*recalls*, *iou_thrs*)

Plot IoU-Recalls curve.

参数

- **recalls** (*ndarray or list*) –shape (k,)
- **iou_thrs** (*ndarray or list*) –same shape as *recalls*

mmdet.evaluation.functional.**plot_num_recall**(*recalls*, *proposal_nums*)

Plot Proposal_num-Recalls curve.

参数

- **recalls** (*ndarray or list*) –shape (k,)
- **proposal_nums** (*ndarray or list*) –same shape as *recalls*

mmdet.evaluation.functional.**pq_compute_multi_core**(*matched_annotations_list*, *gt_folder*, *pred_folder*, *categories*, *backend_args=None*, *nproc=32*)

Evaluate the metrics of Panoptic Segmentation with multithreading.

Same as the function with the same name in *panopticapi*.

参数

- **matched_annotations_list** (*list*) –The matched annotation list. Each element is a tuple of annotations of the same image with the format (gt_anns, pred_anns).
- **gt_folder** (*str*) –The path of the ground truth images.
- **pred_folder** (*str*) –The path of the prediction images.
- **categories** (*str*) –The categories of the dataset.
- **backend_args** (*object*) –The file client of the dataset. If None, the backend will be set to *local*.
- **nproc** (*int*) –Number of processes for panoptic quality computing. Defaults to 32. When *nproc* exceeds the number of cpu cores, the number of cpu cores is used.

mmdet.evaluation.functional.**pq_compute_single_core**(*proc_id*, *annotation_set*, *gt_folder*, *pred_folder*, *categories*, *backend_args=None*, *print_log=False*)

The single core function to evaluate the metric of Panoptic Segmentation.

Same as the function with the same name in *panopticapi*. Only the function to load the images is changed to use the file client.

参数

- **proc_id** (*int*) –The id of the mini process.
- **gt_folder** (*str*) –The path of the ground truth images.
- **pred_folder** (*str*) –The path of the prediction images.
- **categories** (*str*) –The categories of the dataset.
- **backend_args** (*object*) –The Backend of the dataset. If None, the backend will be set to *local*.

- **print_log** (*bool*) –Whether to print the log. Defaults to False.

mmdet.evaluation.functional.**print_map_summary**(*mean_ap*, *results*, *dataset=None*, *scale_ranges=None*, *logger=None*)

Print mAP and results of each class.

A table will be printed to show the gts/dets/recall/AP of each class and the mAP.

参数

- **mean_ap** (*float*) –Calculated from *eval_map()*.
- **results** (*list[dict]*) –Calculated from *eval_map()*.
- **dataset** (*list[str] | str | None*) –Dataset name or dataset classes.
- **scale_ranges** (*list[tuple] | None*) –Range of scales to be evaluated.
- **logger** (*logging.Logger | str | None*) –The way to print the mAP summary. See *mmengine.logging.print_log()* for details. Defaults to None.

mmdet.evaluation.functional.**print_recall_summary**(*recalls*, *proposal_nums*, *iou_thrs*, *row_idxs=None*, *col_idxs=None*, *logger=None*)

Print recalls in a table.

参数

- **recalls** (*ndarray*) –calculated from *bbox_recalls*
- **proposal_nums** (*ndarray or list*) –top N proposals
- **iou_thrs** (*ndarray or list*) –iou thresholds
- **row_idxs** (*ndarray*) –which rows(proposal nums) to print
- **col_idxs** (*ndarray*) –which cols(iou thresholds) to print
- **logger** (*logging.Logger | str | None*) –The way to print the recall summary. See *mmengine.logging.print_log()* for details. Default: None.

mmdet.evaluation.functional.**voc_classes**() → list

Class names of PASCAL VOC.

12.2 metrics

CHAPTER 13

mmdet.models

13.1 backbones

13.2 data_preprocessors

13.3 dense_heads

13.4 detectors

13.5 layers

13.6 losses

13.7 necks

13.8 roi_heads

13.9 seg_heads

13.10 task_modules

13.11 test_time_augs

CHAPTER 14

mmdet.structures

14.1 structures

class mmdet.structures.**DetDataSample**(*, *metainfo: Optional[dict] = None*, ***kwargs*)

A data structure interface of MMDetection. They are used as interfaces between different components.

The attributes in `DetDataSample` are divided into several parts:

- **"proposals"(InstanceData): Region proposals used in two-stage** detectors.
- "gt_instances"(InstanceData): Ground truth of instance annotations.
- "pred_instances"(InstanceData): Instances of model predictions.
- **"ignored_instances"(InstanceData): Instances to be ignored during** training/testing.
- **"gt_panoptic_seg"(PixelData): Ground truth of panoptic** segmentation.
- **"pred_panoptic_seg"(PixelData): Prediction of panoptic** segmentation.
- "gt_sem_seg"(PixelData): Ground truth of semantic segmentation.
- "pred_sem_seg"(PixelData): Prediction of semantic segmentation.

实际案例

```
>>> import torch
>>> import numpy as np
>>> from mmengine.structures import InstanceData
>>> from mmdet.structures import DetDataSample
```

```
>>> data_sample = DetDataSample()
>>> img_meta = dict(img_shape=(800, 1196),
...                 pad_shape=(800, 1216))
>>> gt_instances = InstanceData(metainfo=img_meta)
>>> gt_instances.bboxes = torch.rand((5, 4))
>>> gt_instances.labels = torch.rand((5,))
>>> data_sample.gt_instances = gt_instances
>>> assert 'img_shape' in data_sample.gt_instances.metainfo_keys()
>>> len(data_sample.gt_instances)
5
>>> print(data_sample)
```

<DetDataSample(

META INFORMATION

DATA FIELDS gt_instances: <InstanceData(

META INFORMATION pad_shape: (800, 1216) img_shape: (800, 1196)

DATA FIELDS labels: tensor([0.8533, 0.1550, 0.5433, 0.7294, 0.5098])
bboxes: tensor([[9.7725e-01, 5.8417e-01, 1.7269e-01, 6.5694e-01],

[1.7894e-01, 5.1780e-01, 7.0590e-01, 4.8589e-01], [7.0392e-01, 6.6770e-01, 1.7520e-01, 1.4267e-01], [2.2411e-01, 5.1962e-01, 9.6953e-01, 6.6994e-01], [4.1338e-01, 2.1165e-01, 2.7239e-04, 6.8477e-01]])

) at 0x7f21fb1b9190>

) at 0x7f21fb1b9880>

```
>>> pred_instances = InstanceData(metainfo=img_meta)
>>> pred_instances.bboxes = torch.rand((5, 4))
>>> pred_instances.scores = torch.rand((5,))
>>> data_sample = DetDataSample(pred_instances=pred_instances)
>>> assert 'pred_instances' in data_sample
```

```
>>> data_sample = DetDataSample()
>>> gt_instances_data = dict(
...                         bboxes=torch.rand(2, 4),
...                         labels=torch.rand(2),
...                         masks=np.random.rand(2, 2, 2))
>>> gt_instances = InstanceData(**gt_instances_data)
>>> data_sample.gt_instances = gt_instances
>>> assert 'gt_instances' in data_sample
>>> assert 'masks' in data_sample.gt_instances
```

```
>>> data_sample = DetDataSample()
>>> gt_panoptic_seg_data = dict(panoptic_seg=torch.rand(2, 4))
>>> gt_panoptic_seg = PixelData(**gt_panoptic_seg_data)
>>> data_sample.gt_panoptic_seg = gt_panoptic_seg
>>> print(data_sample)
```

<DetDataSample(

META INFORMATION

DATA FIELDS _gt_panoptic_seg: <BaseDataElement(

META INFORMATION

DATA FIELDS panoptic_seg: tensor([[0.7586, 0.1262, 0.2892, 0.9341],

[0.3200, 0.7448, 0.1052, 0.5371]])

) at 0x7f66c2bb7730>

gt_panoptic_seg: <BaseDataElement(

META INFORMATION

DATA FIELDS panoptic_seg: tensor([[0.7586, 0.1262, 0.2892, 0.9341],

[0.3200, 0.7448, 0.1052, 0.5371]])

) at 0x7f66c2bb7730>

) at 0x7f66c2bb7280> »> data_sample = DetDataSample() »> gt_segm_seg_data = dict(segm_seg=torch.rand(2, 2, 2)) »> gt_segm_seg = PixelData(**gt_segm_seg_data) »> data_sample.gt_segm_seg = gt_segm_seg »> assert 'gt_segm_seg' in data_sample »> assert 'segm_seg' in data_sample.gt_segm_seg

14.2 bbox

14.3 mask

CHAPTER 15

mmdet.testing

CHAPTER 16

mmdet.visualization

CHAPTER 17

mmdet.utils

class mmdet.utils.**AvoidOOM**(*to_cpu=True*, *test=False*)

Try to convert inputs to FP16 and CPU if got a PyTorch's CUDA Out of Memory error. It will do the following steps:

1. First retry after calling *torch.cuda.empty_cache()*.
2. If that still fails, it will then retry by converting inputs

   to FP16.

3. If that still fails trying to convert inputs to CPUs.

   In this case, it expects the function to dispatch to CPU implementation.

参数

- **to_cpu** (*bool*) –Whether to convert outputs to CPU if get an OOM error. This will slow down the code significantly. Defaults to True.
- **test** (*bool*) –Skip *_ignore_torch_cuda_oom* operate that can use lightweight data in unit test, only used in test unit. Defaults to False.

实际案例

```
>>> from mmdet.utils.memory import AvoidOOM
>>> AvoidCUDAOOM = AvoidOOM()
>>> output = AvoidOOM.retry_if_cuda_oom(
>>>     some_torch_function)(input1, input2)
>>> # To use as a decorator
>>> # from mmdet.utils import AvoidCUDAOOM
>>> @AvoidCUDAOOM.retry_if_cuda_oom
>>> def function(*args, **kwargs):
>>>     return None
```

“‘

注解:

1. **The output may be on CPU even if inputs are on GPU. Processing** on CPU will slow down the code significantly.
2. **When converting inputs to CPU, it will only look at each argument** and check if it has *.device* and *.to* for conversion. Nested structures of tensors are not supported.
3. **Since the function might be called more than once, it has to be** stateless.

---

retry_if_cuda_oom(*func*)

Makes a function retry itself after encountering pytorch’s CUDA OOM error.

The implementation logic is referred to https://github.com/facebookresearch/detectron2/blob/main/detectron2/utils/memory.py

参数 **func** –a stateless callable that takes tensor-like objects as arguments.

返回 a callable which retries *func* if OOM is encountered.

返回类型 func

mmdet.utils.**all_reduce_dict**(*py_dict*, *op='sum'*, *group=None*, *to_float=True*)

Apply all reduce function for python dict object.

The code is modified from https://github.com/Megvii- BaseDetection/YOLOX/blob/main/yolox/utils/allreduce_norm.py.

NOTE: make sure that py_dict in different ranks has the same keys and the values should be in the same shape. Currently only supports nccl backend.

参数

- **py_dict** (*dict*) –Dict to be applied all reduce op.
- **op** (*str*) –Operator, could be ‘sum’ or ‘mean’. Default: ‘sum’

- **group** (torch.distributed.group, optional) –Distributed group, Default: None.
- **to_float** (*bool*) –Whether to convert all values of dict to float. Default: True.

返回 reduced python dict object.

返回类型 OrderedDict

mmdet.utils.**allreduce_grads**(*params*, *coalesce=True*, *bucket_size_mb=- 1*)

Allreduce gradients.

参数

- **params** (*list[torch.Parameters]*) –List of parameters of a model
- **coalesce** (*bool, optional*) –Whether allreduce parameters as a whole. Defaults to True.
- **bucket_size_mb** (*int, optional*) –Size of bucket, the unit is MB. Defaults to -1.

mmdet.utils.**collect_env**()

Collect the information of the running environments.

mmdet.utils.**compat_cfg**(*cfg*)

This function would modify some filed to keep the compatibility of config.

For example, it will move some args which will be deprecated to the correct fields.

mmdet.utils.**find_latest_checkpoint**(*path*, *suffix='pth'*)

Find the latest checkpoint from the working directory.

参数

- **path** (*str*) –The path to find checkpoints.
- **suffix** (*str*) –File extension. Defaults to pth.

返回 File path of the latest checkpoint.

返回类型 latest_path(str | None)

引用

mmdet.utils.**get_caller_name**()

Get name of caller method.

mmdet.utils.**get_test_pipeline_cfg**(*cfg: Union[str, mmengine.config.config.ConfigDict]*) → mmengine.config.config.ConfigDict

Get the test dataset pipeline from entire config.

参数 **cfg** (str or ConfigDict) –the entire config. Can be a config file or a ConfigDict.

返回 the config of test dataset.

返回类型 ConfigDict

mmdet.utils.**log_img_scale**(*img_scale*, *shape_order='hw'*, *skip_square=False*)

Log image size.

参数

- **img_scale** (*tuple*) –Image size to be logged.
- **shape_order** (*str, optional*) –The order of image shape. 'hw' for (height, width) and 'wh' for (width, height). Defaults to 'hw'.
- **skip_square** (*bool, optional*) –Whether to skip logging for square img_scale. Defaults to False.

返回 Whether to have done logging.

返回类型 bool

mmdet.utils.**reduce_mean**(*tensor*)

"Obtain the mean of tensor on different GPUs.

mmdet.utils.**register_all_modules**(*init_default_scope: bool = True*) → None

Register all modules in mmdet into the registries.

参数 **init_default_scope** (*bool*) –Whether initialize the mmdet default scope. When *init_default_scope=True*, the global default scope will be set to *mmdet*, and all registries will build modules from mmdet's registry node. To understand more about the registry, please refer to https://github.com/open-mmlab/mmengine/blob/main/docs/en/tutorials/registry.md Defaults to True.

mmdet.utils.**replace_cfg_vals**(*ori_cfg*)

Replace the string "${key}" with the corresponding value.

Replace the "${key}" with the value of ori_cfg.key in the config. And support replacing the chained ${key}. Such as, replace "${key0.key1}" with the value of cfg.key0.key1. Code is modified from **'vars.py < https://github.com/microsoft/SoftTeacher/blob/main/ssod/utils/vars.py>'_** # noqa: E501

参数 **ori_cfg** (*mmengine.config.Config*) –The origin config with "${key}" generated from a file.

返回 The config with "${key}" replaced by the corresponding value.

返回类型 updated_cfg [mmengine.config.Config]

mmdet.utils.**setup_cache_size_limit_of_dynamo**()

Setup cache size limit of dynamo.

Note: Due to the dynamic shape of the loss calculation and post-processing parts in the object detection algorithm, these functions must be compiled every time they are run. Setting a large value for torch._dynamo.config.cache_size_limit may result in repeated compilation, which can slow down training and testing speed. Therefore, we need to set the default value of cache_size_limit smaller. An empirical value is 4.

mmdet.utils.**setup_multi_processes**(*cfg*)

Setup multi-processing environment variables.

mmdet.utils.**split_batch**(*img*, *img_metas*, *kwargs*)

Split data_batch by tags.

Code is modified from <https://github.com/microsoft/SoftTeacher/blob/main/ssod/utils/structure_utils.py> # noqa: E501

参数

- **img** (*Tensor*) –of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) –List of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys, see mmdet.datasets.pipelines.Collect.
- **kwargs** (*dict*) –Specific to concrete implementation.

返回

a dict that data_batch splited by tags, such as 'sup', 'unsup_teacher', and 'unsup_student'.

返回类型 data_groups (dict)

mmdet.utils.**sync_random_seed**(*seed=None*, *device='cuda'*)

Make sure different ranks share the same seed.

All workers must call this function, otherwise it will deadlock. This method is generally used in *DistributedSampler*, because the seed should be identical across all processes in the distributed group.

In distributed sampling, different ranks should sample non-overlapped data in the dataset. Therefore, this function is used to make sure that each rank shuffles the data indices in the same order based on the same seed. Then different ranks could use different indices to select non-overlapped data from the same data list.

参数

- **seed** (*int, Optional*) –The seed. Default to None.
- **device** (*str*) –The device where the seed will be put on. Default to 'cuda'.

返回 Seed to be used.

返回类型 int

mmdet.utils.**update_data_root**(*cfg*, *logger=None*)

Update data root according to env MMDET_DATASETS.

If set env MMDET_DATASETS, update cfg.data_root according to MMDET_DATASETS. Otherwise, using cfg.data_root as default.

参数

- **cfg** (Config) –The model config need to modify
- **logger** (*logging.Logger | str | None*) –the way to print msg

CHAPTER 18

模型库

18.1 镜像地址

从 MMDetection V2.0 起，我们只通过阿里云维护模型库。V1.x 版本的模型已经弃用。

18.2 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。
- 所有 pytorch-style 的 ImageNet 预训练主干网络来自 PyTorch 的模型库，caffe-style 的预训练主干网络来自 detectron2 最新开源的模型。
- 为了与其他代码库公平比较，文档中所写的 GPU 内存是 8 个 GPU 的 `torch.cuda.max_memory_allocated()` 的最大值，此值通常小于 nvidia-smi 显示的值。
- 我们以网络 forward 和后处理的时间加和作为推理时间，不包含数据加载时间。所有结果通过 benchmark.py 脚本计算所得。该脚本会计算推理 2000 张图像的平均时间。

18.3 ImageNet 预训练模型

通过 ImageNet 分类任务预训练的主干网络进行初始化是很常见的操作。所有预训练模型的链接都可以在 open_mmlab 中找到。根据 `img_norm_cfg` 和原始权重，我们可以将所有 ImageNet 预训练模型分为以下几种情况：

- TorchVision：torchvision 模型权重，包含 ResNet50, ResNet101。`img_norm_cfg` 为 `dict(mean=[123.675, 116.28, 103.53], std=[58.395, 57.12, 57.375], to_rgb=True)`。
- Pycls：pycls 模型权重，包含 RegNetX。`img_norm_cfg` 为 `dict( mean=[103.530, 116.280, 123.675], std=[57.375, 57.12, 58.395], to_rgb=False)`。
- MSRA styles：MSRA 模型权重，包含 ResNet50_Caffe，ResNet101_Caffe。`img_norm_cfg` 为 `dict( mean=[103.530, 116.280, 123.675], std=[1.0, 1.0, 1.0], to_rgb=False)`。
- Caffe2 styles：现阶段只包含 ResNext101_32x8d。`img_norm_cfg` 为 `dict(mean=[103.530, 116.280, 123.675], std=[57.375, 57.120, 58.395], to_rgb=False)`。
- Other styles: SSD 的 `img_norm_cfg` 为 `dict(mean=[123.675, 116.28, 103.53], std=[1, 1, 1], to_rgb=True)`，YOLOv3 的 `img_norm_cfg` 为 `dict(mean=[0, 0, 0], std=[255., 255., 255.], to_rgb=True)`。

MMdetection 常用到的主干网络细节如下表所示：

18.4 Baselines

18.4.1 RPN

请参考 RPN。

18.4.2 Faster R-CNN

请参考 Faster R-CNN。

18.4.3 Mask R-CNN

请参考 Mask R-CNN。

18.4.4 Fast R-CNN (使用提前计算的 proposals)

请参考 Fast R-CNN。

18.4.5 RetinaNet

请参考 RetinaNet。

18.4.6 Cascade R-CNN and Cascade Mask R-CNN

请参考 Cascade R-CNN。

18.4.7 Hybrid Task Cascade (HTC)

请参考 HTC。

18.4.8 SSD

请参考 SSD。

18.4.9 Group Normalization (GN)

请参考 Group Normalization。

18.4.10 Weight Standardization

请参考 Weight Standardization。

18.4.11 Deformable Convolution v2

请参考 Deformable Convolutional Networks。

18.4.12 CARAFE: Content-Aware ReAssembly of FEatures

请参考 CARAFE。

18.4.13 Instaboost

请参考 Instaboost。

18.4.14 Libra R-CNN

请参考 Libra R-CNN。

18.4.15 Guided Anchoring

请参考 Guided Anchoring。

18.4.16 FCOS

请参考 FCOS。

18.4.17 FoveaBox

请参考 FoveaBox。

18.4.18 RepPoints

请参考 RepPoints。

18.4.19 FreeAnchor

请参考 FreeAnchor。

18.4.20 Grid R-CNN (plus)

请参考 Grid R-CNN。

18.4.21 GHM

请参考 GHM。

18.4.22 GCNet

请参考 GCNet。

18.4.23 HRNet

请参考 HRNet。

18.4.24 Mask Scoring R-CNN

请参考 Mask Scoring R-CNN。

18.4.25 Train from Scratch

请参考 Rethinking ImageNet Pre-training。

18.4.26 NAS-FPN

请参考 NAS-FPN。

18.4.27 ATSS

请参考 ATSS。

18.4.28 FSAF

请参考 FSAF。

18.4.29 RegNetX

请参考 RegNet。

18.4.30 Res2Net

请参考 Res2Net。

18.4.31 GRoIE

请参考 GRoIE。

18.4.32 Dynamic R-CNN

请参考 Dynamic R-CNN。

18.4.33 PointRend

请参考 PointRend。

18.4.34 DetectoRS

请参考 DetectoRS。

18.4.35 Generalized Focal Loss

请参考 Generalized Focal Loss。

18.4.36 CornerNet

请参考 CornerNet。

18.4.37 YOLOv3

请参考 YOLOv3。

18.4.38 PAA

请参考 PAA。

18.4.39 SABL

请参考 SABL。

18.4.40 CentripetalNet

请参考 CentripetalNet。

18.4.41 ResNeSt

请参考 ResNeSt。

18.4.42 DETR

请参考 DETR。

18.4.43 Deformable DETR

请参考 Deformable DETR。

18.4.44 AutoAssign

请参考 AutoAssign。

18.4.45 YOLOF

请参考 YOLOF。

18.4.46 Seesaw Loss

请参考 Seesaw Loss。

18.4.47 CenterNet

请参考 CenterNet。

18.4.48 YOLOX

请参考 YOLOX。

18.4.49 PVT

请参考 PVT。

18.4.50 SOLO

请参考 SOLO。

18.4.51 QueryInst

请参考 QueryInst。

18.4.52 Other datasets

我们还在 PASCAL VOC，Cityscapes 和 WIDER FACE 上对一些方法进行了基准测试。

18.4.53 Pre-trained Models

我们还通过多尺度训练和更长的训练策略来训练用 ResNet-50 和 RegNetX-3.2G 作为主干网络的 Faster R-CNN 和 Mask R-CNN。这些模型可以作为下游任务的预训练模型。

18.5 速度基准

18.5.1 训练速度基准

我们提供 analyze_logs.py 来得到训练中每一次迭代的平均时间。示例请参考 Log Analysis。

我们与其他流行框架的 Mask R-CNN 训练速度进行比较（数据是从 detectron2 复制而来）。在 mmdetection 中，我们使用 mask-rcnn_r50-caffe_fpn_poly-1x_coco_v1.py 进行基准测试。它与 detectron2 的 mask_rcnn_R_50_FPN_noaug_1x.yaml 设置完全一样。同时，我们还提供了模型权重和训练 log 作为参考。为了跳过 GPU 预热时间，吞吐量按照 100-500 次迭代之间的平均吞吐量来计算。

18.5.2 推理时间基准

我们提供 benchmark.py 对推理时间进行基准测试。此脚本将推理 2000 张图片并计算忽略前 5 次推理的平均推理时间。可以通过设置 `LOG-INTERVAL` 来改变 log 输出间隔（默认为 50）。

```
python tools/benchmark.py ${CONFIG} ${CHECKPOINT} [--log-interval $[LOG-INTERVAL]] [--
↪fuse-conv-bn]
```

模型库中，所有模型在基准测量推理时间时都没设置 `fuse-conv-bn`, 此设置可以使推理时间更短。

18.6 与 Detectron2 对比

我们在速度和精度方面对 mmdetection 和 Detectron2 进行对比。对比所使用的 detectron2 的 commit id 为 185c27e(30/4/2020)。为了公平对比，我们所有的实验都在同一机器下进行。

18.6.1 硬件

- 8 NVIDIA Tesla V100 (32G) GPUs
- Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz

18.6.2 软件环境

- Python 3.7
- PyTorch 1.4
- CUDA 10.1
- CUDNN 7.6.03
- NCCL 2.4.08

18.6.3 精度

18.6.4 训练速度

训练速度使用 s/iter 来度量。结果越低越好。

18.6.5 推理速度

推理速度通过单张 GPU 下的 fps(img/s) 来度量，越高越好。为了与 Detectron2 保持一致，我们所写的推理时间除去了数据加载时间。对于 Mask RCNN，我们去除了后处理中 RLE 编码的时间。我们在括号中给出了官方给出的速度。由于硬件差异，官方给出的速度会比我们所测试得到的速度快一些。

18.6.6 训练内存

CHAPTER 19

基于 MMDetection 的项目

有许多开源项目都是基于 MMDetection 搭建的，我们在这里列举一部分作为样例，展示如何基于 MMDetection 搭建您自己的项目。由于这个页面列举的项目并不完全，我们欢迎社区提交 Pull Request 来更新这个文档。

19.1 MMDetection 的拓展项目

一些项目拓展了 MMDetection 的边界，如将 MMDetection 拓展支持 3D 检测或者将 MMDetection 用于部署。它们展示了 MMDetection 的许多可能性，所以我们在这里也列举一些。

- OTEDetection: OpenVINO training extensions for object detection.
- MMDetection3d: OpenMMLab’s next-generation platform for general 3D object detection.

19.2 研究项目

同样有许多研究论文是基于 MMDetection 进行的。许多论文都发表在了顶级的会议或期刊上，或者对社区产生了深远的影响。为了向社区提供一个可以参考的论文列表，帮助大家开发或者比较新的前沿算法，我们在这里也遵循会议的时间顺序列举了一些论文。MMDetection 中已经支持的算法不在此列。

- Involution: Inverting the Inherence of Convolution for Visual Recognition, CVPR21. [paper][github]
- Multiple Instance Active Learning for Object Detection, CVPR 2021. [paper][github]
- Adaptive Class Suppression Loss for Long-Tail Object Detection, CVPR 2021. [paper][github]
- Generalizable Pedestrian Detection: The Elephant In The Room, CVPR2021. [paper][github]

- Group Fisher Pruning for Practical Network Compression, ICML2021. [paper][github]
- Overcoming Classifier Imbalance for Long-tail Object Detection with Balanced Group Softmax, CVPR2020. [paper][github]
- Coherent Reconstruction of Multiple Humans from a Single Image, CVPR2020. [paper][github]
- Look-into-Object: Self-supervised Structure Modeling for Object Recognition, CVPR 2020. [paper][github]
- Video Panoptic Segmentation, CVPR2020. [paper][github]
- D2Det: Towards High Quality Object Detection and Instance Segmentation, CVPR2020. [paper][github]
- CentripetalNet: Pursuing High-quality Keypoint Pairs for Object Detection, CVPR2020. [paper][github]
- Learning a Unified Sample Weighting Network for Object Detection, CVPR 2020. [paper][github]
- Scale-equalizing Pyramid Convolution for Object Detection, CVPR2020. [paper] [github]
- Revisiting the Sibling Head in Object Detector, CVPR2020. [paper][github]
- PolarMask: Single Shot Instance Segmentation with Polar Representation, CVPR2020. [paper][github]
- Hit-Detector: Hierarchical Trinity Architecture Search for Object Detection, CVPR2020. [paper][github]
- ZeroQ: A Novel Zero Shot Quantization Framework, CVPR2020. [paper][github]
- CBNet: A Novel Composite Backbone Network Architecture for Object Detection, AAAI2020. [paper][github]
- RDSNet: A New Deep Architecture for Reciprocal Object Detection and Instance Segmentation, AAAI2020. [paper][github]
- Training-Time-Friendly Network for Real-Time Object Detection, AAAI2020. [paper][github]
- Cascade RPN: Delving into High-Quality Region Proposal Network with Adaptive Convolution, NeurIPS 2019. [paper][github]
- Reasoning R-CNN: Unifying Adaptive Global Reasoning into Large-scale Object Detection, CVPR2019. [paper][github]
- Learning RoI Transformer for Oriented Object Detection in Aerial Images, CVPR2019. [paper][github]
- SOLO: Segmenting Objects by Locations. [paper][github]
- SOLOv2: Dynamic, Faster and Stronger. [paper][github]
- Dense Peppoints: Representing Visual Objects with Dense Point Sets. [paper][github]
- IterDet: Iterative Scheme for Object Detection in Crowded Environments. [paper][github]
- Cross-Iteration Batch Normalization. [paper][github]
- A Ranking-based, Balanced Loss Function Unifying Classification and Localisation in Object Detection, NeurIPS2020 [paper][github]

CHAPTER 20

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，请使用 issue 模板创建问题，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。

20.1 PyTorch 2.0 支持

MMDetection 目前绝大部分算法已经支持了 PyTorch 2.0 及其 `torch.compile` 功能，用户只需要安装 MMDetection 3.0.0rc7 及其以上版本即可。如果你在使用中发现有不支持的算法，欢迎给我们反馈。我们也非常欢迎社区贡献者来 benchmark 对比 `torch.compile` 功能所带来的速度提升。

如果你想启动 `torch.compile` 功能，只需要在 `train.py` 或者 `test.py` 后面加上 `--cfg-options compile=True`。以 RTMDet 为例，你可以使用以下命令启动 `torch.compile` 功能：

```
# 单卡
python tools/train.py configs/rtmdet/rtmdet_s_8xb32-300e_coco.py  --cfg-options␣
↪compile=True

# 单机 8 卡
./tools/dist_train.sh configs/rtmdet/rtmdet_s_8xb32-300e_coco.py 8 --cfg-options␣
↪compile=True

# 单机 8 卡 + AMP 混合精度训练
./tools/dist_train.sh configs/rtmdet/rtmdet_s_8xb32-300e_coco.py 8 --cfg-options␣
↪compile=True --amp
```

(下页继续)

(续上页)

需要特别注意的是，PyTorch 2.0 对于动态 shape 支持不是非常完善，目标检测算法中大部分不仅输入 shape 是动态的，而且 loss 计算和后处理过程中也是动态的，这会导致在开启 `torch.compile` 功能后训练速度会变慢。基于此，如果你想启动 `torch.compile` 功能，则应该遵循如下原则：

1. 输入到网络的图片是固定 shape 的，而非多尺度的
2. 设置 `torch._dynamo.config.cache_size_limit` 参数。TorchDynamo 会将 Python 字节码转换并缓存，已编译的函数会被存入缓存中。当下一次检查发现需要重新编译时，该函数会被重新编译并缓存。但是如果重编译次数超过预设的最大值（64），则该函数将不再被缓存或重新编译。前面说过目标检测算法中的 loss 计算和后处理部分也是动态计算的，这些函数需要在每次迭代中重新编译。因此将 `torch._dynamo.config.cache_size_limit` 参数设置得更小一些可以有效减少编译时间

在 MMDetection 中可以通过环境变量 `DYNAMO_CACHE_SIZE_LIMIT` 设置 `torch._dynamo.config.cache_size_limit` 参数，以 RTMDet 为例，命令如下所示：

```
# 单卡
export DYNAMO_CACHE_SIZE_LIMIT = 4
python tools/train.py configs/rtmdet/rtmdet_s_8xb32-300e_coco.py  --cfg-options␣
↪compile=True

# 单机 8 卡
export DYNAMO_CACHE_SIZE_LIMIT = 4
./tools/dist_train.sh configs/rtmdet/rtmdet_s_8xb32-300e_coco.py 8 --cfg-options␣
↪compile=True
```

关于 PyTorch 2.0 的 dynamo 常见问题，可以参考 这里

20.2 安装

- MMCV 与 MMDetection 的兼容问题： “ConvWS is already registered in conv layer”； “AssertionError: MMCV==xxx is used but incompatible. Please install mmcv>=xxx, <=xxx.”

  MMDetection，MMEngine 和 MMCV 的版本兼容关系如下。请选择合适的版本避免安装错误。

注意：

1. 如果你希望安装 mmdet-v2.x, MMDetection 和 MMCV 版本兼容表可以在 这里 找到，请选择合适的版本避免安装错误。
2. 在 MMCV-v2.x 中，`mmcv-full` 改名为 `mmcv`，如果你想安装不包含 CUDA 算子的版本，可以选择安装 MMCV 精简版 `mmcv-lite`。

- “No module named ‘mmcv.ops’”；“No module named ‘mmcv._ext’”.

  原因是安装了 `mmcv-lite` 而不是 `mmcv`。

1. `pip uninstall mmcv-lite` 卸载安装的 `mmcv-lite`
2. 安装 `mmcv` 根据 安装说明。

- 在 Windows 环境下安装过程中遇到“Microsoft Visual C++ 14.0 or graeter is required”error .

  这个错误发生在 pycotools 的‘pycocotools._mask’扩展构建过程，其原因是缺少了对应 C++ 环境依赖。你需要到微软官方下载对应工具，选择“使用 C++ 的桌面开发”选项安装最小依赖，随后重新安装 pycocotools。

- 使用 albumentations

如果你希望使用 `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>=0.3.2` 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 。

- 在某些算法中出现 ModuleNotFoundError 错误

一些算法或者数据需要额外的依赖，例如 Instaboost、Panoptic Segmentation、LVIS dataset 等。请注意错误信息并安装相应的包，例如:

```
# 安装 instaboost 依赖
pip install instaboostfast
# 安装 panoptic segmentation 依赖
pip install git+https://github.com/cocodataset/panopticapi.git
# 安装 LVIS dataset 依赖
pip install git+https://github.com/lvis-dataset/lvis-api.git
```

20.3 代码

- 修改一些代码后是否需要重新安装 mmdet

如果你遵循最佳实践，即使用 `pip install -v -e .` 安装的 mmdet，则对本地代码所作的任何修改都会生效，无需重新安装

- 如何使用多个 MMDetection 版本进行开发

你可以拥有多个文件夹，例如 mmdet-3.0，mmdet-3.1。

要使环境中安装默认的 MMDetection 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
```

20.4 PyTorch/CUDA 环境相关

- “RTX 30 series card fails when building MMCV or MMDet”

  1. 临时解决方案为使用命令 `MMCV_WITH_OPS=1 MMCV_CUDA_ARGS='-gencode=arch=compute_80,code=sm_80' pip install -e .` 进行编译。常见报错信息为 `nvcc fatal : Unsupported gpu architecture 'compute_86'` 意思是你的编译器不支持 sm_86 架构 (包括英伟达 30 系列的显卡) 的优化，至 CUDA toolkit 11.0 依旧未支持. 这个命令是通过增加宏 `MMCV_CUDA_ARGS='-gencode=arch=compute_80,code=sm_80` 让 nvcc 编译器为英伟达 30 系列显卡进行 `sm_80` 的优化，虽然这有可能会无法发挥出显卡所有性能。
  2. 有开发者已经在 pytorch/pytorch#47585 更新了 PyTorch 默认的编译 flag，但是我们对此并没有进行测试。

- “invalid device function” 或者 “no kernel image is available for execution”.

  1. 检查您正常安装了 CUDA runtime (一般在`/usr/local/`)，或者使用 `nvcc --version` 检查本地版本，有时安装 PyTorch 会顺带安装一个 CUDA runtime，并且实际优先使用 conda 环境中的版本，你可以使用 `conda list cudatoolkit` 查看其版本。
  2. 编译 extension 的 CUDA Toolkit 版本与运行时的 CUDA Toolkit 版本是否相符，
     - 如果您从源码自己编译的，使用 `python mmdet/utils/collect_env.py` 检查编译编译 extension 的 CUDA Toolkit 版本，然后使用 `conda list cudatoolkit` 检查当前 conda 环境是否有 CUDA Toolkit，若有检查版本是否匹配，如不匹配，更换 conda 环境的 CUDA Toolkit，或者使用匹配的 CUDA Toolkit 中的 nvcc 编译即可，如环境中无 CUDA Toolkit，可以使用 `nvcc -V`。

       等命令查看当前使用的 CUDA runtime。
     - 如果您是通过 pip 下载的预编译好的版本，请确保与当前 CUDA runtime 一致。
  3. 运行 `python mmdet/utils/collect_env.py` 检查是否为正确的 GPU 架构编译的 PyTorch，torchvision，与 MMCV。你或许需要设置 `TORCH_CUDA_ARCH_LIST` 来重新安装 MMCV，可以参考 GPU 架构表，例如，运行 `TORCH_CUDA_ARCH_LIST=7.0 pip install mmcv` 为 Volta GPU 编译 MMCV。这种架构不匹配的问题一般会出现在使用一些旧型号的 GPU 时候出现，例如，Tesla K80。

- “undefined symbol” 或者 “cannot open xxx.so”.

  1. 如果这些 symbol 属于 CUDA/C++ (如 libcudart.so 或者 GLIBCXX)，使用 `python mmdet/utils/collect_env.py` 检查 CUDA/GCC runtime 与编译 MMCV 的 CUDA 版本是否相同。
  2. 如果这些 symbols 属于 PyTorch，(例如, symbols containing caffe, aten, and TH), 检查当前 Pytorch 版本是否与编译 MMCV 的版本一致。
  3. 运行 `python mmdet/utils/collect_env.py` 检查 PyTorch，torchvision，MMCV 等的编译环境与运行环境一致。

- setuptools.sandbox.UnpickleableException: DistutilsSetupError(“each element of ‘ext_modules’ option must be an Extension instance or 2-tuple”)

  1. 如果你在使用 miniconda 而不是 anaconda，检查是否正确的安装了 Cython 如 #3379.
  2. 检查环境中的 `setuptools`, `Cython`, and `PyTorch` 相互之间版本是否匹配。

- “Segmentation fault”.

  1. 检查 GCC 的版本，通常是因为 PyTorch 版本与 GCC 版本不匹配（例如 GCC < 4.9），我们推荐用户使用 GCC 5.4，我们也不推荐使用 GCC 5.5，因为有反馈 GCC 5.5 会导致“segmentation fault”并且切换到 GCC 5.4 就可以解决问题。
  2. 检查是否正确安装了 CUDA 版本的 PyTorch 。

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

     是否返回 True。
  3. 如果 `torch` 的安装是正确的，检查是否正确编译了 MMCV。

     ```
     python -c 'import mmcv; import mmcv.ops'
     ```

  4. 如果 MMCV 与 PyTorch 都被正确安装了，则使用 `ipdb`, `pdb` 设置断点，直接查找哪一部分的代码导致了 `segmentation fault`。

20.5 Training 相关

- “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（例如，instaboost）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，您可以把 `warmup_iters` 从 500 更改为 1000 或 2000。
  4. 添加 gradient clipping: 一些模型需要梯度裁剪来稳定训练过程。默认的 `grad_clip` 是 `None`, 你可以在 config 设置 `optimizer_config=dict(_delete_=True, grad_clip=dict(max_norm=35, norm_type=2))` 如果你的 config 没有继承任何包含 `optimizer_config=dict(grad_clip=None)`, 你可以直接设置 `optimizer_config=dict(grad_clip=dict(max_norm=35, norm_type=2))`.

- “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。您可以在 assigner 的配置中设置 `gpu_assign_thr=N`，这样当超过 N 个 GT boxes 时，assigner 会通过 CPU 计算 IOU。
2. 在 backbone 中设置 `with_cp=True`。这使用 PyTorch 中的 `sublinear strategy` 来降低 backbone 占用的 GPU 显存。
3. 使用 `config/fp16` 中的示例尝试混合精度训练。`loss_scale` 可能需要针对不同模型进行调整。
4. 你也可以尝试使用 `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
```

- “RuntimeError: Expected to have finished reduction in the prior iteration before starting a new one”
  1. 这个错误出现在存在参数没有在 forward 中使用，容易在 DDP 中运行不同分支时发生。
  2. 你可以在 config 设置 `find_unused_parameters = True` 进行训练 (会降低训练速度)。
  3. 你也可以通过在 config 中的 `optimizer_config` 里设置 `detect_anomalous_params=True` 查找哪些参数没有用到，但是需要 MMCV 的版本 >= 1.4.1。
- 训练中保存最好模型

  可以通过配置 `default_hooks = dict(checkpoint=dict(type='CheckpointHook', interval=1, save_best='auto')` 开启。在 `auto` 参数情况下会根据返回的验证结果中的第一个 key 作为选择最优模型的依据，你也可以直接设置评估结果中的 key 来手动设置，例如 `save_best='coco/bbox_mAP'`。
- 在 Resume 训练中使用 `ExpMomentumEMAHook`

  如果在训练中使用了 `ExpMomentumEMAHook`，那么 resume 时候不能仅仅通过命令行参数 `--resume-from` 或 `--cfg-options resume_from` 实现恢复模型参数功能例如 `python tools/train.py configs/yolox/yolox_s_8x8_300e_coco.py --resume-from ./work_dir/yolox_s_8x8_300e_coco/epoch_x.pth`。以 `yolox_s` 算法为例，由于 `ExpMomentumEMAHook` 需要重新加载权重，你可以通过如下做法实现：

```
# 直接打开 configs/yolox/yolox_s_8x8_300e_coco.py 修改所有 resume_from 字段
resume_from=./work_dir/yolox_s_8x8_300e_coco/epoch_x.pth
custom_hooks=[...
    dict(
        type='ExpMomentumEMAHook',
        resume_from=./work_dir/yolox_s_8x8_300e_coco/epoch_x.pth,
        momentum=0.0001,
        priority=49)
    ]
```

20.6 Evaluation 相关

- 使用 COCO Dataset 的测评接口时, 测评结果中 AP 或者 AR = -1

  1. 根据 COCO 数据集的定义，一张图像中的中等物体与小物体面积的阈值分别为 9216（96*96）与 1024（32*32）。
  2. 如果在某个区间没有检测框 AP 与 AR 认定为 -1.

20.7 Model 相关

- **ResNet style 参数说明**

  ResNet style 可选参数允许 `pytorch` 和 `caffe`，其差别在于 Bottleneck 模块。Bottleneck 是 `1x1-3x3-1x1` 堆叠结构，在 `caffe` 模式模式下 stride=2 参数放置在第一个 `1x1` 卷积处，而 `pyorch` 模式下 stride=2 放在第二个 `3x3` 卷积处。一个简单示例如下：

```
if self.style == 'pytorch':
      self.conv1_stride = 1
      self.conv2_stride = stride
else:
      self.conv1_stride = stride
      self.conv2_stride = 1
```

- **ResNeXt 参数说明**

  ResNeXt 来自论文 `Aggregated Residual Transformations for Deep Neural Networks`. 其引入分组卷积，并且通过变量基数来控制组的数量达到精度和复杂度的平衡，其有两个超参 `baseWidth` 和 `cardinality` 来控制内部 Bottleneck 模块的基本宽度和分组数参数。以 MMDetection 中配置名为 `mask_rcnn_x101_64x4d_fpn_mstrain-poly_3x_coco.py` 为例，其中 `mask_rcnn` 代表算法采用 Mask R-CNN，`x101` 代表骨架网络采用 ResNeXt-101，`64x4d` 代表 Bottleneck 一共分成 64 组，每组的基本宽度是 4。

- **骨架网络 eval 模式说明**

  因为检测模型通常比较大且输入图片分辨率很高，这会导致检测模型的 batch 很小，通常是 2，这会使得 BatchNorm 在训练过程计算的统计量方差非常大，不如主干网络预训练时得到的统计量稳定，因此在训练是一般都会使用 `norm_eval=True` 模式，直接使用预训练主干网络中的 BatchNorm 统计量，少数使用大 batch 的算法是 `norm_eval=False` 模式，例如 NASFPN。对于没有 ImageNet 预训练的骨架网络，如果 batch 比较小，可以考虑使用 `SyncBN`。

CHAPTER 21

MMDetection v2.x 兼容性说明

21.1 MMDetection 2.25.0

为了加入 Mask2Former 实例分割模型，对 Mask2Former 的配置文件进行了重命名 PR #7571：

```
'mask2former_xxx_coco.py' 代表全景分割的配置文件
```

```
'mask2former_xxx_coco.py' 代表实例分割的配置文件
'mask2former_xxx_coco-panoptic.py' 代表全景分割的配置文件
```

21.2 MMDetection 2.21.0

为了支持 CPU 训练，MMCV 中进行批处理的 scatter 的代码逻辑已经被修改。我们推荐使用 MMCV v1.4.4 或更高版本，更多信息请参考 MMCV PR #1621.

21.3 MMDetection 2.18.1

21.3.1 MMCV compatibility

为了修复 BaseTransformerLayer 中的权重引用问题, MultiheadAttention 中 batch first 的逻辑有所改变。我们推荐使用 MMCV v1.3.17 或更高版本。更多信息请参考 MMCV PR #1418 。

21.4 MMDetection 2.18.0

21.4.1 DIIHead 兼容性

为了支持 QueryInst，在 DIIHead 的返回元组中加入了 attn_feats。

21.5 MMDetection v2.14.0

21.5.1 MMCV 版本

为了修复 EvalHook 优先级过低的问题，MMCV v1.3.8 中所有 hook 的优先级都重新进行了调整，因此 MMDetection v2.14.0 需要依赖最新的 MMCV v1.3.8 版本。相关信息请参考PR #1120 ，相关问题请参考#5343 。

21.5.2 SSD 兼容性

在 v2.14.0 中，为了使 SSD 能够被更灵活地使用，PR #5291 重构了 SSD 的 backbone、neck 和 head。用户可以使用 tools/model_converters/upgrade_ssd_version.py 转换旧版本训练的模型。

```
python tools/model_converters/upgrade_ssd_version.py ${OLD_MODEL_PATH} ${NEW_MODEL_
↪PATH}
```

- OLD_MODEL_PATH：旧版 SSD 模型的路径。
- NEW_MODEL_PATH：保存转换后模型权重的路径。

21.6 MMDetection v2.12.0

在 v2.12.0 到 v2.18.0（或以上）版本的这段时间，为了提升通用性和便捷性，MMDetection 正在进行大规模重构。在升级到 v2.12.0 后 MMDetection 不可避免地带来了一些 BC Breaking，包括 MMCV 的版本依赖、模型初始化方式、模型 registry 和 mask AP 的评估。

21.6.1 MMCV 版本

MMDetection v2.12.0 依赖 MMCV v1.3.3 中新增加的功能，包括：使用 `BaseModule` 统一参数初始化，模型 registry，以及Deformable DETR 中的 `MultiScaleDeformableAttn` CUDA 算子。注意，尽管 MMCV v1.3.2 已经包含了 MMDet 所需的功能，但是存在一些已知的问题。我们建议用户跳过 MMCV v1.3.2 使用 v1.3.3 版本。

21.6.2 统一模型初始化

为了统一 OpenMMLab 项目中的参数初始化方式，MMCV 新增加了 `BaseModule` 类，使用 `init_cfg` 参数对模块进行统一且灵活的初始化配置管理。现在用户需要在训练脚本中显式调用 `model.init_weights()` 来初始化模型（例如 这行代码 ，在这之前则是在 detector 中进行处理的。**下游项目必须相应地更新模型初始化方式才能使用 MMDetection v2.12.0**。请参阅 PR #4750 了解详情。

21.6.3 统一模型 registry

为了能够使用在其他 OpenMMLab 项目中实现的 backbone，MMDetection v2.12.0 继承了在 MMCV (#760) 中创建的模型 registry。这样，只要 OpenMMLab 项目实现了某个 backbone，并且该项目也使用 MMCV 中的 registry，那么用户只需修改配置即可在 MMDetection 中使用该 backbone，不再需要将代码复制到 MMDetection 中。更多详细信息，请参阅 PR #5059 。

21.6.4 Mask AP 评估

在 PR #4898 和 v2.12.0 之前，对小、中、大目标的 mask AP 的评估是基于其边界框区域而不是真正的 mask 区域。这导致 `APs` 和 `APm` 变得更高但 `APl` 变得更低，但是不会影响整体的 mask AP。PR #4898 删除了 mask AP 计算中的 `bbox` ，改为使用 mask 区域。新的计算方式不会影响整体的 mask AP 评估，与 Detectron2一致。

21.7 与 MMDetection v1.x 的兼容性

MMDetection v2.0 经过了大规模重构并解决了许多遗留问题。MMDetection v2.0 不兼容 v1.x 版本，在这两个版本中使用相同的模型权重运行推理会产生不同的结果。因此，MMDetection v2.0 重新对所有模型进行了 benchmark，并在 model zoo 中提供了新模型的权重和训练记录。

新旧版本的主要的区别有四方面：坐标系、代码库约定、训练超参和模块设计。

21.7.1 坐标系

新坐标系与 Detectron2 一致，将最左上角的像素的中心视为坐标原点 (0, 0) 而不是最左上角像素的左上角。因此 COCO 边界框和分割标注中的坐标被解析为范围 `[0, width]` 和 `[0, height]` 中的坐标。这个修改影响了所有与 bbox 及像素选择相关的计算，变得更加自然且更加准确。

- 在新坐标系中，左上角和右下角为 (x1, y1) (x2, y2) 的框的宽度及高度计算公式为 `width = x2 - x1` 和 `height = y2 - y1`。在 MMDetection v1.x 和之前的版本中，高度和宽度都多了 `+ 1` 的操作。本次修改包括三部分：
  1. box 回归中的检测框变换以及编码/解码。
  2. IoU 计算。这会影响 ground truth 和检测框之间的匹配以及 NMS 。但对兼容性的影响可以忽略不计。
  3. Box 的角点坐标为浮点型，不再取整。这能使得检测结果更为准确，也使得检测框和 RoI 的最小尺寸不再为 1，但影响很小。
- Anchor 的中心与特征图的网格点对齐，类型变为 float。在 MMDetection v1.x 和之前的版本中，anchors 是 `int` 类型且没有居中对齐。这会影响 RPN 中的 Anchor 生成和所有基于 Anchor 的方法。
- ROIAlign 更好地与图像坐标系对齐。新的实现来自 Detectron2 。当 RoI 用于提取 RoI 特征时，与 MMDetection v1.x 相比默认情况下相差半个像素。能够通过设置 `aligned=False` 而不是 `aligned=True` 来维持旧版本的设置。
- Mask 的裁剪和粘贴更准确。
  1. 我们使用新的 RoIAlign 来提取 mask 目标。在 MMDetection v1.x 中，bounding box 在提取 mask 目标之前被取整，裁剪过程是 numpy 实现的。而在新版本中，裁剪的边界框不经过取整直接输入 RoIAlign。此实现大大加快了训练速度（每次迭代约加速 0.1 秒，1x schedule 训练 Mask R50 时加速约 2 小时）并且理论上会更准确。
  2. 在 MMDetection v2.0 中，修改后的 `paste_mask()` 函数应该比之前版本更准确。此更改参考了 Detectron2 中的修改，可以将 COCO 上的 mask AP 提高约 0.5%。

21.7.2 代码库约定

- MMDetection v2.0 更改了类别标签的顺序，减少了回归和 mask 分支里的无用参数并使得顺序更加自然（没有 +1 和 -1）。这会影响模型的所有分类层，使其输出的类别标签顺序发生改变。回归分支和 mask head 的最后一层不再为 K 个类别保留 K+1 个通道，类别顺序与分类分支一致。
  - 在 MMDetection v2.0 中，标签 “K” 表示背景，标签 [0, K-1] 对应于 K = num_categories 个对象类别。
  - 在 MMDetection v1.x 及之前的版本中，标签 “0” 表示背景，标签 [1, K] 对应 K 个类别。
  - **注意**：softmax RPN 的类顺序在 version<=2.4.0 中仍然和 1.x 中的一样，而 sigmoid RPN 不受影响。从 MMDetection v2.5.0 开始，所有 head 中的类顺序是统一的。
- 不使用 R-CNN 中的低质量匹配。在 MMDetection v1.x 和之前的版本中，`max_iou_assigner` 会在 RPN 和 R-CNN 训练时给每个 ground truth 匹配低质量框。我们发现这会导致最佳的 GT 框不会被分配给某些边界框，因此，在 MMDetection v2.0 的 R-CNN 训练中默认不允许低质量匹配。这有时可能会稍微改善 box AP（约为 0.1%）。
- 单独的宽高比例系数。在 MMDetection v1.x 和以前的版本中，`keep_ratio=True` 时比例系数是单个浮点数，这并不准确，因为宽度和高度的比例系数会有一定的差异。MMDetection v2.0 对宽度和高度使用单独的比例系数，对 AP 的提升约为 0.1%。
- 修改了 config 文件名称的规范。由于 model zoo 中模型不断增多，MMDetection v2.0 采用新的命名规则：

```
[model]_(model setting)_[backbone]_[neck]_(norm setting)_(misc)_(gpu x batch)_
↪[schedule]_[dataset].py
```

  其中 (`misc`) 包括 DCN 和 GCBlock 等。更多详细信息在 配置文件说明文档中说明

- MMDetection v2.0 使用新的 ResNet Caffe backbone 来减少加载预训练模型时的警告。新 backbone 中的大部分权重与以前的相同，但没有 `conv.bias`，且它们使用不同的 `img_norm_cfg`。因此，新的 backbone 不会报 `unexpected keys` 的警告。

21.7.3 训练超参

训练超参的调整不会影响模型的兼容性，但会略微提高性能。主要有：

- 通过设置 `nms_post=1000` 和 `max_num=1000`，将 nms 之后的 proposal 数量从 2000 更改为 1000。使 mask AP 和 bbox AP 提高了约 0.2%。
- Mask R-CNN、Faster R-CNN 和 RetinaNet 的默认回归损失从 smooth L1 损失更改为 L1 损失，使得 box AP 整体上都有所提升（约 0.6%）。但是，将 L1-loss 用在 Cascade R-CNN 和 HTC 等其他方法上并不能提高性能，因此我们保留这些方法的原始设置。
- 为简单起见，RoIAlign 层的 `sampling_ratio` 设置为 0。略微提升了 AP（约 0.2% 绝对值）。
- 为了提升训练速度，默认设置在训练过程中不再使用梯度裁剪。大多数模型的性能不会受到影响。对于某些模型（例如 RepPoints），我们依旧使用梯度裁剪来稳定训练过程从而获得更好的性能。

- 因为不再默认使用梯度裁剪，默认 warmup 比率从 1/3 更改为 0.001，以使模型训练预热更加平缓。不过我们重新进行基准测试时发现这种影响可以忽略不计。

21.7.4 将模型从 v1.x 升级至 v2.0

用户可以使用脚本 `tools/model_converters/upgrade_model_version.py` 来将 MMDetection 1.x 训练的模型转换为 MMDetection v2.0。转换后的模型可以在 MMDetection v2.0 中运行，但性能略有下降（小于 1% AP）。详细信息可以在 `configs/legacy` 中找到。

21.8 pycocotools 兼容性

`mmpycocotools` 是 OpenMMLab 维护的 `pycocotools` 的复刻版，适用于 MMDetection 和 Detectron2。在 PR #4939 之前，由于 `pycocotools` 和 `mmpycocotool` 具有相同的包名，如果用户已经安装了 `pyccocotools`（在相同环境下先安装了 Detectron2 ），那么 MMDetection 的安装过程会跳过安装 `mmpycocotool`。导致 MMDetection 缺少 `mmpycocotools` 而报错。但如果在 Detectron2 之前安装 MMDetection，则可以在相同的环境下工作。PR #4939 弃用 mmpycocotools，使用官方 pycocotools。在 PR #4939 之后，用户能够在相同环境下安装 MMDetection 和 Detectron2，不再需要关注安装顺序。

CHAPTER 22

中文解读文案汇总（待更新）

22.1 1 官方解读文案（v2.x）

22.1.1 1.1 框架解读

- **轻松掌握 MMDetection 整体构建流程 (一)**
- **轻松掌握 MMDetection 整体构建流程 (二)**
- **轻松掌握 MMDetection 中 Head 流程**

22.1.2 1.2 算法解读

- **轻松掌握 MMDetection 中常用算法 (一)：RetinaNet 及配置详解**
- **轻松掌握 MMDetection 中常用算法 (二)：Faster R-CNN|Mask R-CNN**
- 轻松掌握 MMDetection 中常用算法 (三)：FCOS
- 轻松掌握 MMDetection 中常用算法 (四)：ATSS
- 轻松掌握 MMDetection 中常用算法 (五)：Cascade R-CNN
- 轻松掌握 MMDetection 中常用算法 (六)：YOLOF
- 轻松掌握 MMDetection 中常用算法 (七)：CenterNet
- 轻松掌握 MMDetection 中常用算法 (八)：YOLACT

- 轻松掌握 MMDetection 中常用算法 (九)：AutoAssign
- YOLOX 在 MMDetection 中复现全流程解析
- 喂喂喂！你可以减重了！小模型 - MMDetection 新增 SSDLite 、MobileNetV2YOLOV3 两大经典算法

22.1.3 1.3 工具解读

- OpenMMLab 中混合精度训练 AMP 的正确打开方式
- 小白都能看懂！手把手教你使用混淆矩阵分析目标检测
- MMDetection 图像缩放 Resize 详细说明 OpenMMLab
- 拿什么拯救我的 4G 显卡
- MMDet 居然能用 MMCls 的 Backbone？论配置文件的打开方式

22.1.4 1.4 知乎问答

- 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?
- 单阶段、双阶段、anchor-based、anchor-free 这四者之间有什么联系吗?
- 目标检测的深度学习方法，有推荐的书籍或资料吗?
- 大佬们，刚入学研究生，想入门目标检测，有什么学习路线可以入门的?
- 目标检测领域还有什么可以做的?
- 如何看待 Transformer 在 CV 上的应用前景，未来有可能替代 CNN 吗?
- MMDetection 如何学习源码?
- 如何具体上手实现目标检测呢?

22.1.5 1.5 其他

- **不得不知的 MMDetection 学习路线 (个人经验版)**
- OpenMMLab 社区专访之 YOLOX 复现篇

22.2 2 社区解读文案（v2.x）

- 手把手带你实现经典检测网络 Mask R-CNN 的推理

CHAPTER 23

English

CHAPTER 24

简体中文

CHAPTER 25

Indices and tables

- genindex
- search

Python 模块索引

m

索引

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

Y

模块