mmrotate

MMRotate Author

2022 年 03 月 28 日

学习基础知识

1 学习基础知识 1

2 依赖 7

3 安装流程 9

4 验证 13

5 准备数据集 15

6 Test a model 17

7 训练一个模型 19

8 基准和模型库 21

9 教程 1：学习配置文件 23

10 Tutorial 2: Customize Datasets 33

11 教程 3: 自定义模型 39

12 教程 4: 自定义训练设置 47

13 Changelog 55

14 常见问题解答 57

15 English 61

16 简体中文 63

17 mmrotate 65

18 Indices and tables 69

Python 模块索引 71

索引 73

CHAPTER 1

学习基础知识

本章将向您介绍旋转目标检测的基本概念，以及旋转目标检测的框架 MMRotate，并提供了详细教程的链接。

1.1 什么是旋转目标检测

1.1.1 问题定义

受益于通用检测的蓬勃发展，目前绝大多数的旋转检测模型都是基于经典的通用检测器。随着检测任务的发展，水平框在一些细分领域上已经无法满足研究人员的需求。通过重新定义目标表示形式以及增加回归自由度数量的操作来实现旋转矩形框、四边形甚至任意形状检测，我们称之为旋转目标检测。如何更加高效地进行高精度的旋转目标检测已成为当下的研究热点。下面列举一些旋转目标检测已经被应用或者有巨大潜力的领域：人脸识别、场景文字、遥感影像、自动驾驶、医学图像、机器人抓取等。

1.1.2 什么是旋转框

旋转目标检测与通用目标检测最大的不同就是用旋转框标注来代替水平框标注，它们的定义如下：

- 水平框: 宽沿 x 轴方向，高沿 y 轴方向的矩形。通常可以用 2 个对角顶点的坐标表示 `(x_i, y_i)` (i = 1, 2)，也可以用中心点坐标以及宽和高表示 `(x_center, y_center, height, width)`。
- 旋转框: 由水平框绕中心点旋转一个角度 `angle` 得到，通过添加一个弧度参数得到其旋转框定义法 `(x_center, y_center, height, width, theta)`。其中，`theta = angle * pi / 180`，单位为 `rad`。当旋转的角度为 90° 的倍数时，旋转框退化为水平框。标注软件导出的旋转框标注通常为多边形 `(xr_i, yr_i)` (i = 1, 2, 3, 4)，在训练时需要转换为旋转框定义法。

注解： 在 MMRotate 中，角度参数的单位均为弧度。

1.1.3 旋转方向

旋转框可以由水平框绕其中心点顺时针旋转或逆时针旋转得到。旋转方向和坐标系的选择密切相关。图像空间采用右手坐标系 (y，x)，其中 y 是上-> 下，x 是左-> 右。此时存在 2 种相反的旋转方向：

- 顺时针（CW）

CW 的示意图

```
0-------------------> x (0 rad)
|  A-------------B
|  |             |
|  |     box     h
|  |   angle=0   |
|  D------w------C
v
y (pi/2 rad)
```

CW 的旋转矩阵

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

CW 的旋转变换

$$\begin{aligned} P_A = \begin{pmatrix} x_A \\ y_A \end{pmatrix} &= \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix} + \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} -0.5w \\ -0.5h \end{pmatrix} \\ &= \begin{pmatrix} x_{center} - 0.5w\cos\alpha + 0.5h\sin\alpha \\ y_{center} - 0.5w\sin\alpha - 0.5h\cos\alpha \end{pmatrix} \end{aligned}$$

- 逆时针（CCW）

CCW 的示意图

```
0-------------------> x (0 rad)
|  A-------------B
|  |             |
|  |     box     h
|  |   angle=0   |
|  D------w------C
v
y (-pi/2 rad)
```

CCW 的旋转矩阵

$$\begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

CCW 的旋转变换

$$\begin{aligned} P_A = \begin{pmatrix} x_A \\ y_A \end{pmatrix} &= \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix} + \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} -0.5w \\ -0.5h \end{pmatrix} \\ &= \begin{pmatrix} x_{center} - 0.5w\cos\alpha - 0.5h\sin\alpha \\ y_{center} + 0.5w\sin\alpha - 0.5h\cos\alpha \end{pmatrix} \end{aligned}$$

在 MMCV 中可以设置旋转方向的算子有：

- box_iou_rotated (默认为 `CW`)
- nms_rotated (默认为 `CW`)
- RoIAlignRotated (默认为 `CCW`)
- RiRoIAlignRotated (默认为 `CCW`)。

注解： 在 MMRotate 中，旋转框的旋转方向均为 `CW`。

1.1.4 旋转框定义法

由于 `theta` 定义范围的不同，在旋转目标检测中逐渐派生出如下 3 种旋转框定义法：

- $D_{oc'}$：OpenCV 定义法，`angle∈(0, 90°]`，`theta∈(0, pi / 2]`，height 与 x 正半轴之间的夹角为正的锐角。该定义法源于 OpenCV 中的 `cv2.minAreaRect` 函数，其返回值为 `(0, 90°]`。
- D_{le135}：长边 135° 定义法，`angle∈[-45°, 135°)`，`theta∈[-pi / 4, 3 * pi / 4)` 并且 `height > width`。
- D_{le90}：长边 90° 定义法，`angle∈[-90°, 90°)`，`theta∈[-pi / 2, pi / 2)` 并且 `height > width`。

三种定义法之间的转换关系在 MMRotate 内部并不涉及，因此不多做介绍。如果想了解更多的细节，可以参考这篇博客。

注解： MMRotate 同时支持上述三种旋转框定义法，可以通过配置文件灵活切换。

需要注意的是，在 4.5.1 之前的版本中，`cv2.minAreaRect` 的返回值为 `[-90°, 0°)`（参考资料）。为了便于区分，将老版本的 OpenCV 定义法记作 D_{oc}。

- $D_{oc'}$：OpenCV 定义法，`opencv>=4.5.1`，`angle∈(0, 90°]`，`theta∈(0, pi / 2]`。

- D_{oc}：老版的 OpenCV 定义法，`opencv<4.5.1, angle∈[-90°, 0°), theta∈[-pi / 2, 0)`。

两种 OpenCV 定义法的转换关系如下：

$$D_{oc'}\left(h_{oc'}, w_{oc'}, \theta_{oc'}\right) = \begin{cases} D_{oc}\left(w_{oc}, h_{oc}, \theta_{oc} + \pi/2\right), otherwise \\ D_{oc}\left(h_{oc}, w_{oc}, \theta_{oc} + \pi\right), \theta_{oc} = -\pi/2 \end{cases}$$

$$D_{oc}\left(h_{oc}, w_{oc}, \theta_{oc}\right) = \begin{cases} D_{oc'}\left(w_{oc'}, h_{oc'}, \theta_{oc'} - \pi/2\right), otherwise \\ D_{oc'}\left(h_{oc'}, w_{oc'}, \theta_{oc'} - \pi\right), \theta_{oc'} = \pi/2 \end{cases}$$

注解： 不管您使用的 OpenCV 版本是多少, MMRotate 都会将 OpenCV 定义法的 theta 转换为 (0, pi / 2]。

1.1.5 评估

评估 mAP 的代码中涉及 IoU 的计算，可以直接计算旋转框 IoU，也可以将旋转框转换为多边形，然后计算多边形 IoU (DOTA 在线评估使用的是计算多边形 IoU)。

1.2 什么是 MMRotate

MMRotate 是一个为旋转目标检测方法提供统一训练和评估框架的工具箱，以下是其整体框架：：

MMRotate 包括四个部分, `datasets`, `models`, `core` and `apis`.

- `datasets` 用于数据加载和数据增强。在这部分, 我们支持了各种旋转目标检测数据集和数据增强预处理。
- `models` 包括模型和损失函数。
- `core` 为模型训练和评估提供工具。
- `apis` 为模型训练、测试和推理提供高级 API。

MMRotate 的模块设计如下图所示：

其中由于旋转框定义法不同而需要注意的地方有如下几个：

- 读取标注
- 数据增强
- 指派样本
- 评估指标

1.3 如何使用教程

下面是 MMRotate 详细的分步指南:

1. 关于安装说明, 请参阅安装.
2. 开始 介绍了 MMRotate 的基本用法.
3. 如果想要更加深入了解 MMRotate，请参阅以下教程:

- 配置
- 自定义数据集
- 自定义模型
- 自定义运行时

CHAPTER 2

依赖

- Linux & Windows
- Python 3.7+
- PyTorch 1.6+
- CUDA 9.2+
- GCC 5+
- mmcv 1.4.5+
- mmdet 2.19.0+

MMRotate 和 MMCV, MMDet 版本兼容性如下所示，需要安装正确的版本以避免安装出现问题。

** 注意：** 如果已经安装了 mmcv，首先需要使用 `pip uninstall mmcv` 卸载已安装的 mmcv，如果同时安装了 mmcv 和 mmcv-full，将会报 `ModuleNotFoundError` 错误。

CHAPTER 3

安装流程

3.1 准备环境

1. 使用 conda 新建虚拟环境，并进入该虚拟环境；

```
conda create -n openmmlab python=3.7 -y
conda activate openmmlab
```

2. 基于 PyTorch 官网安装 PyTorch 和 torchvision，例如：

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

注意：需要确保 CUDA 的编译版本和运行版本匹配。可以在 PyTorch 官网查看预编译包所支持的 CUDA 版本。

例 1 例如在 `/usr/local/cuda` 下安装了 CUDA 10.1，并想安装 PyTorch 1.7，则需要安装支持 CUDA 10.1 的预构建 PyTorch：

```
conda install pytorch==1.7.0 torchvision==0.8.0 cudatoolkit=10.1 -c pytorch
```

3.2 安装 MMRotate

我们建议使用 MIM 来安装 MMRotate：

```
pip install openmim
mim install mmrotate
```

MIM 能够自动地安装 OpenMMLab 的项目以及对应的依赖包。

或者，可以手动安装 MMRotate：

1. 安装 mmcv-full，我们建议使用预构建包来安装：

   ```
   pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/{cu_version}/
   ↪{torch_version}/index.html
   ```

   需要把命令行中的 {cu_version} 和 {torch_version} 替换成对应的版本。例如：在 CUDA 11 和 PyTorch 1.7.0 的环境下，可以使用下面命令安装最新版本的 MMCV：

   ```
   pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu110/torch1.7.
   ↪0/index.html
   ```

   请参考 MMCV 获取不同版本的 MMCV 所兼容的的不同的 PyTorch 和 CUDA 版本。同时，也可以通过以下命令行从源码编译 MMCV：

   ```
   git clone https://github.com/open-mmlab/mmcv.git
   cd mmcv
   MMCV_WITH_OPS=1 pip install -e .  # 安装好 mmcv-full
   cd ..
   ```

   或者，可以直接使用命令行安装：

   ```
   pip install mmcv-full
   ```

2. 安装 MMDetection.

   你可以直接通过如下命令从 pip 安装使用 mmdetection：

   ```
   pip install mmdet
   ```

3. 安装 MMRotate.

   你可以直接通过如下命令从 pip 安装使用 mmrotate：

```
pip install mmrotate
```

或者从 git 仓库编译源码：

```
git clone https://github.com/open-mmlab/mmrotate.git
cd mmrotate
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

Note:

(1) 按照上述说明，MMDetection 安装在 `dev` 模式下，因此在本地对代码做的任何修改都会生效，无需重新安装；

(2) 如果希望使用 `opencv-python-headless` 而不是 `opencv-python`，可以在安装 MMCV 之前安装；

(3) 一些安装依赖是可以选择的。例如只需要安装最低运行要求的版本，则可以使用 `pip install -v -e .` 命令。如果希望使用可选择的像 `albumentations` 和 `imagecorruptions` 这种依赖项，可以使用 `pip install -r requirements/optional.txt` 进行手动安装，或者在使用 `pip` 时指定所需的附加功能（例如 `pip install -v -e .[optional]`），支持附加功能的有效键值包括 `all`、`tests`、`build` 以及 `optional` 。

3.3 另一种选择：Docker 镜像

我们提供了 Dockerfile to build an image. Ensure that you are using docker version >=19.03.

```
# 基于 PyTorch 1.6, CUDA 10.1 生成镜像
docker build -t mmrotate docker/
```

运行命令：

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

3.4 从零开始设置脚本

假设当前已经成功安装 CUDA 10.1，这里提供了一个完整的基于 conda 安装 MMDetection 的脚本：

```
conda create -n openmmlab python=3.7 -y
conda activate openmmlab

conda install pytorch==1.7.0 torchvision==0.8.0 cudatoolkit=10.1 -c pytorch
```

(下页继续)

(续上页)

```
# 安装最新版本的 mmcv
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu101/torch1.7.0/
↪index.html

# 安装 mmdetection
pip install mmdet

# 安装 mmrotate
git clone https://github.com/open-mmlab/mmrotate.git
cd mmrotate
pip install -r requirements/build.txt
pip install -v -e .  # or "python setup.py develop"
```

CHAPTER 4

验证

为了验证是否正确安装了 MMRotate 和所需的环境，我们可以运行示例的 Python 代码在示例图像进行推理：

具体的细节可以参考 demo。如果成功安装 MMRotate，则上面的代码可以完整地运行。

CHAPTER 5

准备数据集

具体的细节可以参考 准备数据 下载并组织数据集。

CHAPTER 6

Test a model

- 单个 GPU
- 单个节点多个 GPU
- 多个节点多个 GPU

您可以使用以下命令来推理数据集。

```
# 单个 GPU
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments]

# 多个 GPU
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM} [optional arguments]

# slurm 环境中多个节点
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments] --
↪launcher slurm
```

例子:

在 DOTA-1.0 数据集推理 RotatedRetinaNet，可以生成压缩文件用于在线提交。(首先请修改 data_root)

```
python ./tools/test.py  \
  configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le90.py \
  checkpoints/SOME_CHECKPOINT.pth --format-only \
  --eval-options submission_dir=work_dirs/Task1_results
```

或者

```
./tools/dist_test.sh  \
  configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le90.py \
  checkpoints/SOME_CHECKPOINT.pth 1 --format-only \
  --eval-options submission_dir=work_dirs/Task1_results
```

您可以修改 data_root 中测试集的路径为验证集或训练集路径用于离线的验证。

```
python ./tools/test.py \
  configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le90.py \
  checkpoints/SOME_CHECKPOINT.pth --eval mAP
```

或者

```
./tools/dist_test.sh  \
  configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le90.py \
  checkpoints/SOME_CHECKPOINT.pth 1 --eval mAP
```

您也可以可视化结果。

```
python ./tools/test.py \
  configs/rotated_retinanet/rotated_retinanet_obb_r50_fpn_1x_dota_le90.py \
  checkpoints/SOME_CHECKPOINT.pth \
  --show-dir work_dirs/vis
```

CHAPTER 7

训练一个模型

7.1 单 GPU 训练

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

如果您想在命令行中指定工作路径，您可以增加参数 `--work_dir ${YOUR_WORK_DIR}`。

7.2 多 GPU 训练

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

可选参数包括:

- `--no-validate` (**不建议**): 默认情况下代码将在训练期间进行评估。通过设置 `--no-validate` 关闭训练期间进行评估。
- `--work-dir ${WORK_DIR}`: 覆盖配置文件中指定的工作目录。
- `--resume-from ${CHECKPOINT_FILE}`: 从以前的检查点恢复训练。

`resume-from` 和 `load-from` 的不同点:

`resume-from` 读取模型的权重和优化器的状态，并且 epoch 也会继承于指定的检查点。通常用于恢复意外中断的训练过程。`load-from` 只读取模型的权重并且训练的 epoch 会从 0 开始。通常用于微调。

7.3 多机多 GPU 训练

如果您在 slurm 管理的集群上运行 MMRotate，您可以使用脚本 `slurm_train.sh` (此脚本还支持单机训练)。

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

如果您有多台机器联网，您可以参考 PyTorch launch utility。如果您没有像 InfiniBand 这样的高速网络，训练速度通常会很慢。

7.4 在一台机器上启动多个作业

如果您在一台机器上启动多个作业，如在一台机器上使用 8 张 GPU 训练 2 个作业，每个作业使用 4 张 GPU，您需要为每个作业指定不同的端口号 (默认为 29500) 进而避免通讯冲突。

如果您使用 `dist_train.sh` 启动训练，您可以在命令行中指定端口号。

```
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
```

如果您通过 Slurm 启动训练，您需要修改配置文件 (通常是配置文件底部的第 6 行) 进而设置不同的通讯端口。

在 `config1.py` 中,

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

在 `config2.py` 中,

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

之后您可以使用 `config1.py` 和 `config2.py` 开启两个作业。

```
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}
```

CHAPTER 8

基准和模型库

- Rotated RetinaNet-OBB/HBB (ICCV’2017)
- Rotated FasterRCNN-OBB (TPAMI’2017)
- Rotated RepPoints-OBB (ICCV’2019)
- RoI Transformer (CVPR’2019)
- Gliding Vertex (TPAMI’2020)
- R3Det (AAAI’2021)
- S2A-Net (TGRS’2021)
- ReDet (CVPR’2021)
- Beyond Bounding-Box (CVPR’2021)
- Oriented R-CNN (ICCV’2021)
- GWD (ICML’2021)
- KLD (NeurIPS’2021)
- SASM (AAAI’2022)
- KFIoU (arXiv)
- G-Rep (stay tuned)

8.1 DOTA v1.0 数据集上的结果

- MS 表示多尺度图像增强。
- RR 表示随机旋转增强。

上述模型都是使用 1 * 1080ti 训练得到的，并且在 1 * 2080ti 上进行推理测试。

CHAPTER 9

教程 1：学习配置文件

我们在配置文件中支持了继承和模块化，这便于进行各种实验。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。mmrotate 是建立在 mmdet 之上的，因此强烈建议学习 mmdet 的基本知识。

9.1 通过脚本参数修改配置

当运行 `tools/train.py` 或者 `tools/test.py` 时，可以通过 `--cfg-options` 来修改配置。

- 更新字典链的配置

  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=LoadImageFromWebcam`。

- 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 `workflow=[('train', 1)]`，如果需要改变这个键，可以通过 `--cfg-options workflow="[(train,1),(val,1)]"` 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。

9.2 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。

```
{model}_[model setting]_{backbone}_{neck}_[norm setting]_[misc]_[gpu x batch_per_gpu]_
↪{dataset}_{data setting}_{angle version}
```

`{xxx}` 是被要求的文件 `[yyy]` 是可选的。

- `{model}`: 模型种类，例如 `rotated_faster_rcnn`, `rotated_retinanet` 等。
- `[model setting]`: 特定的模型，例如 `hbb` for `rotated_retinanet` 等。
- `{backbone}`: 主干网络种类例如 `r50` (ResNet-50), `swin_tiny` (SWIN-tiny) 。
- `{neck}`: Neck 模型的种类包括 `fpn`, `refpn`。
- `[norm_setting]`: 默认使用 `bn` (Batch Normalization)，其他指定可以有 `gn` (Group Normalization), `syncbn` (Synchronized Batch Normalization) 等。`gn-head`/`gn-neck` 表示 GN 仅应用于网络的 Head 或 Neck，`gn-all` 表示 GN 用于整个模型，例如主干网络、Neck 和 Head。
- `[misc]`: 模型中各式各样的设置/插件，例如 `dconv`、`gcb`、`attention`、`albu`、`mstrain` 等。
- `[gpu x batch_per_gpu]`: GPU 数量和每个 GPU 的样本数，默认使用 `1xb2`。
- `{dataset}`: 数据集，例如 `dota`。
- `{angle version}`: 旋转定义方式，例如 `oc`, `le135` 或者 `le90`。

9.3 RotatedRetinaNet 配置文件示例

为了帮助用户对 MMRotate 检测系统中的完整配置和模块有一个基本的了解我们对使用 ResNet50 和 FPN 的 RotatedRetinaNet 的配置文件进行简要注释说明。更详细的用法和各个模块对应的替代方案，请参考 API 文档。

```
angle_version = 'oc'  # 旋转定义方式
model = dict(
    type='RotatedRetinaNet',  # 检测器 (detector) 名称
    backbone=dict(  # 主干网络的配置文件
        type='ResNet',  # # 主干网络的类别
        depth=50,  # 主干网络的深度
        num_stages=4,  # 主干网络阶段 (stages) 的数目
        out_indices=(0, 1, 2, 3),  # 每个阶段产生的特征图输出的索引
        frozen_stages=1,  # 第一个阶段的权重被冻结
        zero_init_residual=False,  # 是否对残差块 (resblocks) 中的最后一个归一化层使用零初始化
(zero init) 让它们表现为自身
        norm_cfg=dict(  # 归一化层 (norm layer) 的配置项
```

(下页继续)

(续上页)

```
            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 'ReFPN'
        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
        start_level=1,  # 用于构建特征金字塔的主干网络起始输入层索引值
        add_extra_convs='on_input',  # 决定是否在原始特征图之上添加卷积层
        num_outs=5),  # 决定输出多少个尺度的特征图 (scales)
    bbox_head=dict(
        type='RotatedRetinaHead',# bbox_head 的类型是 'RRetinaHead'
        num_classes=15,  # 分类的类别数量
        in_channels=256,  # bbox head 输入通道数
        stacked_convs=4,  # head 卷积层的层数
        feat_channels=256,  # head 卷积层的特征通道
        assign_by_circumhbbox='oc',  # obb2hbb 的旋转定义方式
        anchor_generator=dict(  # 锚点 (Anchor) 生成器的配置
            type='RotatedAnchorGenerator',  # 锚点生成器类别
            octave_base_scale=4,  # RetinaNet 用于生成锚点的超参数，特征图 anchor 的基本尺度。
值越大，所有 anchor 的尺度都会变大。
            scales_per_octave=3,  #  RetinaNet 用于生成锚点的超参数，每个特征图有 3 个尺度
            ratios=[1.0, 0.5, 2.0],  # 高度和宽度之间的比率
            strides=[8, 16, 32, 64, 128]),  # 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='DeltaXYWHAOBBoxCoder',  # 框编码器的类别
            angle_range='oc',  # 框编码器的旋转定义方式
            norm_factor=None,  # 框编码器的范数
            edge_swap=False,  # 设置是否启用框编码器的边缘交换
            proj_xy=False,  # 设置是否启用框编码器的投影
            target_means=(0.0, 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, 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='FocalLoss',  # 分类分支的损失函数类型
            use_sigmoid=True,  #  是否使用 sigmoid
            gamma=2.0,  # Focal Loss 用于解决难易不均衡的参数 gamma
            alpha=0.25,  # Focal Loss 用于解决样本数量不均衡的参数 alpha
            loss_weight=1.0),  # 分类分支的损失权重
```

(下页继续)

(续上页)

```
        loss_bbox=dict(  # 回归分支的损失函数配置
            type='L1Loss',  # 回归分支的损失类型
            loss_weight=1.0)),  # 回归分支的损失权重
    train_cfg=dict(  # 训练超参数的配置
        assigner=dict(  # 分配器 (assigner) 的配置
            type='MaxIoUAssigner',  # 分配器的类型
            pos_iou_thr=0.5,  # IoU >= 0.5(阈值) 被视为正样本
            neg_iou_thr=0.4,  # IoU < 0.4(阈值) 被视为负样本
            min_pos_iou=0,  # 将框作为正样本的最小 IoU 阈值
            ignore_iof_thr=-1,  # 忽略 bbox 的 IoF 阈值
            iou_calculator=dict(type='RBboxOverlaps2D')),  # IoU 的计算器类型
        allowed_border=-1,  # 填充有效锚点 (anchor) 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
    test_cfg=dict(  # 测试超参数的配置
        nms_pre=2000,  # NMS 前的 box 数
        min_bbox_size=0,  # box 允许的最小尺寸
        score_thr=0.05,  # bbox 的分数阈值
        nms=dict(iou_thr=0.1), # NMS 的阈值
        max_per_img=2000))  # 每张图像的最大检测次数
dataset_type = 'DOTADataset'  # 数据集类型，这将被用来定义数据集
data_root = '../datasets/split_1024_dota1_0/'  # 数据的根路径
img_norm_cfg = dict(  # 图像归一化配置，用来归一化输入的图像
    mean=[123.675, 116.28, 103.53],  # 预训练里用于预训练主干网络模型的平均值
    std=[58.395, 57.12, 57.375],  # 预训练里用于预训练主干网络模型的标准差
    to_rgb=True)  # 预训练里用于预训练主干网络的图像的通道顺序
train_pipeline = [  # 训练流程
    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),  # 是否加载标注框 (bounding box)， 目标检测需要设置为 True
    dict(type='RResize',  # 变化图像和其注释大小的数据增广的流程
         img_scale=(1024, 1024)),  # 图像的最大规模
    dict(type='RRandomFlip',  # 翻转图像和其注释大小的数据增广的流程
         flip_ratio=0.5,  # 翻转图像的概率
         version='oc'),  # 定义旋转的方式
    dict(
        type='Normalize',  # 归一化当前图像的数据增广的流程
        mean=[123.675, 116.28, 103.53],  # 这些键与 img_norm_cfg 一致，
        std=[58.395, 57.12, 57.375],  # 因为 img_norm_cfg 被用作参数
        to_rgb=True),
    dict(type='Pad',  # 填充当前图像到指定大小的数据增广的流程
         size_divisor=32),  # 填充图像可以被当前值整除
    dict(type='DefaultFormatBundle'),  # 流程里收集数据的默认格式包
```

(下页继续)

(续上页)

```
    dict(type='Collect',  # 决定数据中哪些键应该传递给检测器的流程
         keys=['img', 'gt_bboxes', 'gt_labels'])
]
test_pipeline = [ # 测试流程
    dict(type='LoadImageFromFile'),  # 第 1 个流程，从文件路径里加载图像
    dict(
        type='MultiScaleFlipAug',  # 封装测试时数据增广 (test time augmentations)
        img_scale=(1024, 1024),  # 决定测试时可改变图像的最大规模。用于改变图像大小的流程
        flip=False,  # 测试时是否翻转图像
        transforms=[
            dict(type='RResize'),  # 使用改变图像大小的数据增广
            dict(
                type='Normalize',  # 归一化配置项，值来自 img_norm_cfg
                mean=[123.675, 116.28, 103.53],
                std=[58.395, 57.12, 57.375],
                to_rgb=True),
            dict(type='Pad',  # 将配置传递给可被 32 整除的图像
                 size_divisor=32),
            dict(type='DefaultFormatBundle'),  # 用于在管道中收集数据的默认格式包
            dict(type='Collect',  # 收集测试时必须的键的收集流程
                 keys=['img'])
        ])
]
data = dict(
    samples_per_gpu=2,  # 单个 GPU 的 Batch size
    workers_per_gpu=2,  # 单个 GPU 分配的数据加载线程数
    train=dict(  # 训练数据集配置
        type='DOTADataset',  # 数据集的类别
        ann_file=
        '../datasets/split_1024_dota1_0/trainval/annfiles/',  # 注释文件路径
        img_prefix=
        '../datasets/split_1024_dota1_0/trainval/images/',  # 图片路径前缀
        pipeline=[  # 流程，这是由之前创建的 train_pipeline 传递的
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations', with_bbox=True),
            dict(type='RResize', img_scale=(1024, 1024)),
            dict(type='RRandomFlip', flip_ratio=0.5, version='oc'),
            dict(
                type='Normalize',
                mean=[123.675, 116.28, 103.53],
                std=[58.395, 57.12, 57.375],
                to_rgb=True),
            dict(type='Pad', size_divisor=32),
```

(下页继续)

(续上页)

```
            dict(type='DefaultFormatBundle'),
            dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels'])
        ],
        version='oc'),
    val=dict(  # 验证数据集的配置
        type='DOTADataset',
        ann_file=
        '../datasets/split_1024_dota1_0/trainval/annfiles/',
        img_prefix=
        '../datasets/split_1024_dota1_0/trainval/images/',
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(
                type='MultiScaleFlipAug',
                img_scale=(1024, 1024),
                flip=False,
                transforms=[
                    dict(type='RResize'),
                    dict(
                        type='Normalize',
                        mean=[123.675, 116.28, 103.53],
                        std=[58.395, 57.12, 57.375],
                        to_rgb=True),
                    dict(type='Pad', size_divisor=32),
                    dict(type='DefaultFormatBundle'),
                    dict(type='Collect', keys=['img'])
                ])
        ],
        version='oc'),
    test=dict(  # 测试数据集配置，修改测试开发/测试 (test-dev/test) 提交的 ann_file
        type='DOTADataset',
        ann_file=
        '../datasets/split_1024_dota1_0/test/images/',
        img_prefix=
        '../datasets/split_1024_dota1_0/test/images/',
        pipeline=[  # 由之前创建的 test_pipeline 传递的流程
            dict(type='LoadImageFromFile'),
            dict(
                type='MultiScaleFlipAug',
                img_scale=(1024, 1024),
                flip=False,
                transforms=[
                    dict(type='RResize'),
```

(下页继续)

(续上页)

```
                    dict(
                        type='Normalize',
                        mean=[123.675, 116.28, 103.53],
                        std=[58.395, 57.12, 57.375],
                        to_rgb=True),
                    dict(type='Pad', size_divisor=32),
                    dict(type='DefaultFormatBundle'),
                    dict(type='Collect', keys=['img'])
                ])
        ],
        version='oc'))
evaluation = dict(  # evaluation hook 的配置
    interval=12,  # 验证的间隔
    metric='mAP')  # 验证期间使用的指标
optimizer = dict(  # 用于构建优化器的配置文件
    type='SGD',  # 优化器类型
    lr=0.0025,  # 优化器的学习率
    momentum=0.9,  # 动量 (Momentum)
    weight_decay=0.0001)  # SGD 的衰减权重 (weight decay)
optimizer_config = dict(  # optimizer hook 的配置文件
    grad_clip=dict(
        max_norm=35,
        norm_type=2))
lr_config = dict(  # 学习率调整配置，用于注册 LrUpdater hook
    policy='step',  # 调度流程 (scheduler) 的策略
    warmup='linear',  # 预热 (warmup) 策略，也支持 `exp` 和 `constant`
    warmup_iters=500,  # 预热的迭代次数
    warmup_ratio=0.3333333333333333,  # 用于预热的起始学习率的比率
    step=[8, 11])  # 衰减学习率的起止回合数
runner = dict(
    type='EpochBasedRunner',  # 将使用的 runner 的类别 (例如 IterBasedRunner 或␣
↪EpochBasedRunner)
    max_epochs=12) # runner 总回合 (epoch) 数， 对于 IterBasedRunner 使用 `max_iters`
checkpoint_config = dict(  # checkpoint hook 的配置文件
    interval=12)  # 保存的间隔是 12
log_config = dict(  # register logger hook 的配置文件
    interval=50,  # 打印日志的间隔
    hooks=[
        # dict(type='TensorboardLoggerHook')  # 同样支持 Tensorboard 日志
        dict(type='TextLoggerHook')
    ])  # 用于记录训练过程的记录器 (logger)
dist_params = dict(backend='nccl')  # 用于设置分布式训练的参数，端口也同样可被设置
log_level = 'INFO'  # 日志的级别
```

(下页继续)

(续上页)

```
load_from = None  # 从一个给定路径里加载模型作为预训练模型，它并不会消耗训练时间
resume_from = None  # 从给定路径里恢复检查点 (checkpoints)，训练模式将从检查点保存的轮次开始恢复训练。
workflow = [('train', 1)]  # runner 的工作流程，[('train', 1)] 表示只有一个工作流且工作流仅执行一次。根据 total_epochs 工作流训练 12 个回合 (epoch)。
work_dir = './work_dirs/rotated_retinanet_hbb_r50_fpn_1x_dota_oc'  # 用于保存当前实验的模型检查点 (checkpoints) 和日志的目录
```

9.4 常见问题 (FAQ)

9.4.1 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。值得注意的是，在修改子配置中的中间变量时，需要再次将中间变量传递到相应的字段中。例如，我们想使用离线多尺度策略 (multi scale strategy) 来训练 RoI-Trans。train_pipeline 是我们想要修改的中间变量。

```
_base_ = ['./roi_trans_r50_fpn_1x_dota_le90.py']

data_root = '../datasets/split_ms_dota1_0/'
angle_version = 'le90'
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='RResize', img_scale=(1024, 1024)),
    dict(
        type='RRandomFlip',
        flip_ratio=[0.25, 0.25, 0.25],
        direction=['horizontal', 'vertical', 'diagonal'],
        version=angle_version),
    dict(
        type='PolyRandomRotate',
        rotate_ratio=0.5,
        angles_range=180,
        auto_bound=False,
        version=angle_version),
    dict(type='Normalize', **img_norm_cfg),
    dict(type='Pad', size_divisor=32),
    dict(type='DefaultFormatBundle'),
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels'])
```

(下页继续)

(续上页)

```
]
data = dict(
    train=dict(
        pipeline=train_pipeline,
        ann_file=data_root + 'trainval/annfiles/',
        img_prefix=data_root + 'trainval/images/'),
    val=dict(
        ann_file=data_root + 'trainval/annfiles/',
        img_prefix=data_root + 'trainval/images/'),
    test=dict(
        ann_file=data_root + 'test/images/',
        img_prefix=data_root + 'test/images/'))
```

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_ = './roi_trans_r50_fpn_1x_dota_le90.py'
norm_cfg = dict(type='BN', requires_grad=True)
model = dict(
    backbone=dict(norm_cfg=norm_cfg),
    neck=dict(norm_cfg=norm_cfg),
    ...)
```

CHAPTER 10

Tutorial 2: Customize Datasets

10.1 Support new data format

To support a new data format, you can convert them to existing formats (DOTA format). You could choose to convert them offline (before training by a script) or online (implement a new dataset and do the conversion at training). In MMRotate, we recommend to convert the data into DOTA formats and do the conversion offline, thus you only need to modify the config's data annotation paths and classes after the conversion of your data.

10.1.1 Reorganize new data formats to existing format

The simplest way is to convert your dataset to existing dataset formats (DOTA).

The annotation txt files in DOTA format:

```
184 2875 193 2923 146 2932 137 2885 plane 0
66 2095 75 2142 21 2154 11 2107 plane 0
...
```

Each line represents an object and records it as a 10-dimensional array `A`.

- `A[0:8]`: Polygons with format `(x1, y1, x2, y2, x3, y3, x4, y4)`.
- `A[8]`: Category.
- `A[9]`: Difficulty.

After the data pre-processing, there are two steps for users to train the customized new dataset with existing format (e.g. DOTA format):

1. Modify the config file for using the customized dataset.
2. Check the annotations of the customized dataset.

Here we give an example to show the above two steps, which uses a customized dataset of 5 classes with COCO format to train an existing Cascade Mask R-CNN R50-FPN detector.

1. Modify the config file for using the customized dataset

There are two aspects involved in the modification of config file:

1. The `data` field. Specifically, you need to explicitly add the `classes` fields in `data.train`, `data.val` and `data.test`.
2. The `num_classes` field in the `model` part. Explicitly over-write all the `num_classes` from default value (e.g. 80 in COCO) to your classes number.

In `configs/my_custom_config.py`:

```
# the new config inherits the base configs to highlight the necessary modification
_base_ = './rotated_retinanet_hbb_r50_fpn_1x_dota_oc'

# 1. dataset settings
dataset_type = 'DOTADataset'
classes = ('a', 'b', 'c', 'd', 'e')
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
    train=dict(
        type=dataset_type,
        # explicitly add your class names to the field `classes`
        classes=classes,
        ann_file='path/to/your/train/annotation_data',
        img_prefix='path/to/your/train/image_data'),
    val=dict(
        type=dataset_type,
        # explicitly add your class names to the field `classes`
        classes=classes,
        ann_file='path/to/your/val/annotation_data',
        img_prefix='path/to/your/val/image_data'),
    test=dict(
        type=dataset_type,
        # explicitly add your class names to the field `classes`
```

(下页继续)

(续上页)

```
        classes=classes,
        ann_file='path/to/your/test/annotation_data',
        img_prefix='path/to/your/test/image_data'))

# 2. model settings
model = dict(
    bbox_head=dict(
        type='RotatedRetinaHead',
        # explicitly over-write all the `num_classes` field from default 15 to 5.
        num_classes=15))
```

2. Check the annotations of the customized dataset

Assuming your customized dataset is DOTA format, make sure you have the correct annotations in the customized dataset:

- The `classes` fields in your config file should have exactly the same elements and the same order with the `A[8]` in txt annotations. MMRotate automatically maps the uncontinuous `id` in `categories` to the continuous label indices, so the string order of `name` in `categories` field affects the order of label indices. Meanwhile, the string order of `classes` in config affects the label text during visualization of predicted bounding boxes.

10.2 Customize datasets by dataset wrappers

MMRotate also supports many dataset wrappers to mix the dataset or modify the dataset distribution for training. Currently it supports to three dataset wrappers as below:

- `RepeatDataset`: simply repeat the whole dataset.
- `ClassBalancedDataset`: repeat dataset in a class balanced manner.
- `ConcatDataset`: concat datasets.

10.2.1 Repeat dataset

We use `RepeatDataset` as wrapper to repeat the dataset. For example, suppose the original dataset is `Dataset_A`, to repeat it, the config looks like the following

```
dataset_A_train = dict(
        type='RepeatDataset',
        times=N,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
```

(下页继续)

(续上页)

```
            pipeline=train_pipeline
        )
    )
```

10.2.2 Class balanced dataset

We use `ClassBalancedDataset` as wrapper to repeat the dataset based on category frequency. The dataset to repeat needs to instantiate function `self.get_cat_ids(idx)` to support `ClassBalancedDataset`. For example, to repeat `Dataset_A` with `oversample_thr=1e-3`, the config looks like the following

```
dataset_A_train = dict(
        type='ClassBalancedDataset',
        oversample_thr=1e-3,
        dataset=dict(  # This is the original config of Dataset_A
            type='Dataset_A',
            ...
            pipeline=train_pipeline
        )
    )
```

10.2.3 Concatenate dataset

There are three ways to concatenate the dataset.

1. If the datasets you want to concatenate are in the same type with different annotation files, you can concatenate the dataset configs like the following.

   ```
   dataset_A_train = dict(
       type='Dataset_A',
       ann_file = ['anno_file_1', 'anno_file_2'],
       pipeline=train_pipeline
   )
   ```

   If the concatenated dataset is used for test or evaluation, this manner supports to evaluate each dataset separately. To test the concatenated datasets as a whole, you can set `separate_eval=False` as below.

   ```
   dataset_A_train = dict(
       type='Dataset_A',
       ann_file = ['anno_file_1', 'anno_file_2'],
       separate_eval=False,
       pipeline=train_pipeline
   )
   ```

2. In case the dataset you want to concatenate is different, you can concatenate the dataset configs like the following.

```
dataset_A_train = dict()
dataset_B_train = dict()

data = dict(
    imgs_per_gpu=2,
    workers_per_gpu=2,
    train = [
        dataset_A_train,
        dataset_B_train
    ],
    val = dataset_A_val,
    test = dataset_A_test
    )
```

   If the concatenated dataset is used for test or evaluation, this manner also supports to evaluate each dataset separately.

3. We also support to define `ConcatDataset` explicitly as the following.

```
dataset_A_val = dict()
dataset_B_val = dict()

data = dict(
    imgs_per_gpu=2,
    workers_per_gpu=2,
    train=dataset_A_train,
    val=dict(
        type='ConcatDataset',
        datasets=[dataset_A_val, dataset_B_val],
        separate_eval=False))
```

   This manner allows users to evaluate all the datasets as a single one by setting `separate_eval=False`.

Note:

1. The option `separate_eval=False` assumes the datasets use `self.data_infos` during evaluation. Therefore, COCO datasets do not support this behavior since COCO datasets do not fully rely on `self.data_infos` for evaluation. Combining different types of datasets and evaluating them as a whole is not tested thus is not suggested.

2. Evaluating `ClassBalancedDataset` and `RepeatDataset` is not supported thus evaluating concatenated datasets of these types is also not supported.

A more complex example that repeats `Dataset_A` and `Dataset_B` by N and M times, respectively, and then concatenates the repeated datasets is as the following.

```
dataset_A_train = dict(
    type='RepeatDataset',
    times=N,
    dataset=dict(
        type='Dataset_A',
        ...
        pipeline=train_pipeline
    )
)
dataset_A_val = dict(
    ...
    pipeline=test_pipeline
)
dataset_A_test = dict(
    ...
    pipeline=test_pipeline
)
dataset_B_train = dict(
    type='RepeatDataset',
    times=M,
    dataset=dict(
        type='Dataset_B',
        ...
        pipeline=train_pipeline
    )
)
data = dict(
    imgs_per_gpu=2,
    workers_per_gpu=2,
    train = [
        dataset_A_train,
        dataset_B_train
    ],
    val = dataset_A_val,
    test = dataset_A_test
)
```

CHAPTER 11

教程 3: 自定义模型

我们大致将模型组件分为了 5 种类型。

- 主干网络 (Backbone): 通常是一个全卷积网络 (FCN)，用来提取特征图，比如残差网络 (ResNet)。也可以是基于视觉 Transformer 的网络，比如 Swin Transformer 等。
- Neck: 主干网络和任务头 (Head) 之间的连接组件，比如 FPN, ReFPN。
- 任务头 (Head): 用于某种具体任务（比如边界框预测）的组件。
- 区域特征提取器 (Roi Extractor): 用于从特征图上提取区域特征的组件，比如 RoI Align Rotated。
- 损失 (loss): 任务头上用于计算损失函数的组件，比如 FocalLoss, GWDLoss, and KFIoULoss。

11.1 开发新的组件

11.1.1 添加新的主干网络

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）

新建文件 mmrotate/models/backbones/mobilenet.py。

```
import torch.nn as nn

from mmrotate.models.builder import ROTATED_BACKBONES


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

    def __init__(self, arg1, arg2):
        pass

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

2. 导入模块

你可以将下面的代码添加到 mmrotate/models/backbones/__init__.py 中：

```
from .mobilenet import MobileNet
```

或者添加如下代码

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

到配置文件中以避免修改原始代码。

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),
    ...
```

11.1.2 添加新的 Neck

1. 定义一个 Neck（以 PAFPN 为例）

新建文件 mmrotate/models/necks/pafpn.py。

```
from mmrotate.models.builder import ROTATED_NECKS

@ROTATED_NECKS.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 mmrotate/models/necks/__init__.py 中

```
from .pafpn import PAFPN
```

或者添加

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

到配置文件中以避免修改原始代码。

3. 修改配置文件

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

11.1.3 添加新的 Head

这里，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 `mmrotate/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 mmrotate.models.builder import ROTATED_HEADS
from mmrotate.models.roi_heads.bbox_heads.bbox_head import BBoxHead

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

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

    def __init__(self,
                 num_convs=0,
                 num_fcs=0,
                 conv_out_channels=1024,
                 fc_out_channels=1024,
                 conv_cfg=None,
                 norm_cfg=dict(type='BN'),
                 **kwargs):
        kwargs.setdefault('with_avg_pool', True)
        super(DoubleConvFCBBoxHead, self).__init__(**kwargs)
```

(下页继续)

(续上页)

```
    def forward(self, x_cls, x_reg):
```

然后，如有必要，我们需要实现一个新的 RoI Head。我们打算从 StandardRoIHead 继承出新的 DoubleHeadRoIHead。我们发现 StandardRoIHead 已经实现了下述函数。

```
import torch

from mmdet.core import bbox2result, bbox2roi, build_assigner, build_sampler
from mmrotate.models.builder import ROTATED_HEADS, build_head, build_roi_extractor
from mmrotate.models.roi_heads.base_roi_head import BaseRoIHead
from mmrotate.models.roi_heads.test_mixins import BBoxTestMixin, MaskTestMixin


@ROTATED_HEADS.register_module()
class StandardRoIHead(BaseRoIHead, BBoxTestMixin, MaskTestMixin):
    """Simplest base roi head including one bbox head and one mask head.
    """

    def init_assigner_sampler(self):

    def init_bbox_head(self, bbox_roi_extractor, bbox_head):

    def forward_dummy(self, x, proposals):


    def forward_train(self,
                      x,
                      img_metas,
                      proposal_list,
                      gt_bboxes,
                      gt_labels,
                      gt_bboxes_ignore=None,
                      gt_masks=None):

    def _bbox_forward(self, x, rois):

    def _bbox_forward_train(self, x, sampling_results, gt_bboxes, gt_labels,
                            img_metas):

    def simple_test(self,
                    x,
                    proposal_list,
```

(下页继续)

(续上页)

```
                    img_metas,
                    proposals=None,
                    rescale=False):
        """Test without augmentation."""
```

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 mmrotate/models/roi_heads/double_roi_head.py 中, 我们实现如下的新的 RoI Head:

```
from mmrotate.models.builder import ROTATED_HEADS
from mmrotate.models.roi_heads.standard_roi_head import StandardRoIHead


@ROTATED_HEADS.register_module()
class DoubleHeadRoIHead(StandardRoIHead):
    """RoI head for Double Head RCNN

    https://arxiv.org/abs/1904.06493
    """

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

    def _bbox_forward(self, x, rois):
        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)
        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
```

最后，用户需要把这个新模块添加到 mmrotate/models/bbox_heads/__init__.py 以及 mmrotate/models/roi_heads/__init__.py 中。这样，注册机制就能找到并加载它们。

另外，用户也可以添加

```
custom_imports=dict(
    imports=['mmrotate.models.roi_heads.double_roi_head', 'mmrotate.models.bbox_heads.
↪double_bbox_head'])
```

到配置文件中来实现同样的目的。

11.1.4 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 mmrotate/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 mmrotate.models.builder import ROTATED_LOSSES
from mmdet.models.losses.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

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

(下页继续)

(续上页)

```
        return loss_bbox
```

然后，用户需要把下面的代码加到 mmrotate/models/losses/__init__.py 中。

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

或者，你可以添加：

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

到配置文件来实现相同的目的。

因为 MyLoss 是用于回归的，你需要在 Head 中修改 loss_bbox 字段：

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

CHAPTER 12

教程 4: 自定义训练设置

12.1 自定义优化设置

12.1.1 自定义 Pytorch 支持的优化器

我们已经支持了全部 Pytorch 自带的优化器，唯一需要修改的就是配置文件中 `optimizer` 部分。例如，如果您想使用 `ADAM` (注意如下操作可能会让模型表现下降)，可以使用如下修改：

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

为了修改模型训练的学习率，使用者仅需修改配置文件里 `optimizer` 的 `lr` 即可。使用者可以参考 PyTorch 的 API doc 直接设置参数。

12.1.2 自定义用户自己实现的优化器

1. 定义一个新的优化器

一个自定义的优化器可以这样定义：

假如您想增加一个叫做 `MyOptimizer` 的优化器，它的参数分别有 `a`, `b`, 和 `c`。您需要创建一个名为 `mmrotate/core/optimizer` 的新文件夹；然后参考如下代码段在 `mmrotate/core/optimizer/my_optimizer.py` 文件中实现新的优化器:

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


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

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

2. 增加优化器到注册表 (registry)

为了能够使得上述添加的模块被 mmrotate 发现，需要先将该模块添加到主命名空间（main namespace）。

- 修改 mmrotate/core/optimizer/__init__.py 文件来导入该模块。

  新的被定义的模块应该被导入到 mmrotate/core/optimizer/__init__.py 中，这样注册表才会发现新的模块并添加它：

```
from .my_optimizer import MyOptimizer
```

- 在配置文件中使用 custom_imports 来手动添加该模块

```
custom_imports = dict(imports=['mmrotate.core.optimizer.my_optimizer'], allow_failed_
↪imports=False)
```

mmrotate.core.optimizer.my_optimizer 模块将会在程序开始被导入，并且 MyOptimizer 类将会自动注册。需要注意只有包含 MyOptimizer 类的包 (package) 应当被导入。而 mmrotate.core.optimizer.my_optimizer.MyOptimizer **不能**被直接导入。

事实上，在这种导入方式下用户可以用完全不同的文件夹结构，只要这一模块的根目录已经被添加到 PYTHONPATH 里面。

3. 在配置文件中指定优化器

之后您可以在配置文件的 optimizer 部分里面使用 MyOptimizer。在配置文件里，优化器按照如下形式被定义在 optimizer 部分里：

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

要使用用户自定义的优化器，这部分应该改成：

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

12.1.3 自定义优化器的构造函数 (constructor)

有些模型的优化器可能有一些特别参数配置，例如批归一化层 (BatchNorm layers) 的权重衰减系数 (weight decay)。用户可以通过自定义优化器的构造函数去微调这些细粒度参数。

```
from mmcv.utils import build_from_cfg

from mmcv.runner.optimizer import OPTIMIZER_BUILDERS, OPTIMIZERS
from mmrotate.utils import get_root_logger
from .my_optimizer import MyOptimizer


@OPTIMIZER_BUILDERS.register_module()
class MyOptimizerConstructor(object):

    def __init__(self, optimizer_cfg, paramwise_cfg=None):

    def __call__(self, model):

        return my_optimizer
```

mmcv 默认的优化器构造函数实现可以参考 这里 ，这也可以作为新的优化器构造函数的模板。

12.1.4 其他配置

优化器未实现的技巧应该通过修改优化器构造函数（如设置基于参数的学习率）或者钩子（hooks）去实现。我们列出一些常见的设置，它们可以稳定或加速模型的训练。如果您有更多的设置，欢迎在 PR 和 issue 里面提出。

- **使用梯度裁剪 (gradient clip) 来稳定训练**: 一些模型需要梯度裁剪来稳定训练过程。使用方式如下：

  ```
  optimizer_config = dict(
      _delete_=True, grad_clip=dict(max_norm=35, norm_type=2))
  ```

  如果您的配置继承了已经设置了 optimizer_config 的基础配置（base config），你可能需要设置 _delete_=True 来覆盖不必要的配置参数。请参考 配置文档 了解更多细节。

- **使用动量调度加速模型收敛**: 我们支持动量规划器（Momentum scheduler），以实现根据学习率调节模型优化过程中的动量设置，这可以使模型以更快速度收敛。动量规划器经常与学习率规划器（LR scheduler）一起使用，例如下面的配置经常被用于 3D 检测模型训练中以加速收敛。更多细节请参考 CyclicLrUpdater 和 CyclicMomentumUpdater。

  ```
  lr_config = dict(
      policy='cyclic',
  ```

(下页继续)

(续上页)

```
    target_ratio=(10, 1e-4),
    cyclic_times=1,
    step_ratio_up=0.4,
)
momentum_config = dict(
    policy='cyclic',
    target_ratio=(0.85 / 0.95, 1),
    cyclic_times=1,
    step_ratio_up=0.4,
)
```

12.2 自定义训练计划

默认地，我们使用 1x 计划（1x schedule）的步进学习率（step learning rate），这在 MMCV 中被称为 `StepLRHook`。我们支持很多其他的学习率规划器，参考 这里 ，例如 `CosineAnnealing` 和 `Poly`。下面是一些例子：

- `Poly`：

```
lr_config = dict(policy='poly', power=0.9, min_lr=1e-4, by_epoch=False)
```

- `ConsineAnnealing`：

```
lr_config = dict(
    policy='CosineAnnealing',
    warmup='linear',
    warmup_iters=1000,
    warmup_ratio=1.0 / 10,
    min_lr_ratio=1e-5)
```

12.3 自定义工作流 (workflow)

工作流是一个专门定义运行顺序和轮数（epochs）的列表。默认情况下它设置成：

```
workflow = [('train', 1)]
```

这是指训练 1 个 epoch。有时候用户可能想检查一些模型在验证集上的指标，如损失函数值（Loss）和准确性（Accuracy）。在这种情况下，我们可以将工作流设置为：

```
[('train', 1), ('val', 1)]
```

这样以来，1 个 epoch 训练，1 个 epoch 验证将交替运行。

注意:

1. 模型参数在验证的阶段不会被自动更新。
2. 配置文件里的键值 `total_epochs` 仅控制训练的 epochs 数目，而不会影响验证工作流。
3. 工作流 `[('train', 1), ('val', 1)]` 和 `[('train', 1)]` 将不会改变 `EvalHook` 的行为，因为 `EvalHook` 被 `after_train_epoch` 调用而且验证的工作流仅仅影响通过调用 `after_val_epoch` 的钩子 (hooks)。因此，`[('train', 1), ('val', 1)]` 和 `[('train', 1)]` 的区别仅在于 runner 将在每次训练阶段（training epoch）结束后计算在验证集上的损失。

12.4 自定义钩 (hooks)

12.4.1 自定义用户自己实现的钩子（hooks）

1. 实现一个新的钩子（hook）

在某些情况下，用户可能需要实现一个新的钩子。MMRotate 支持训练中的自定义钩子。因此，用户可以直接在 mmrotate 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。这里我们举一个例子：在 mmrotate 中创建一个新的钩子并在训练中使用它。

```
from mmcv.runner import HOOKS, Hook


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

    def __init__(self, a, b):
        pass

    def before_run(self, runner):
        pass

    def after_run(self, runner):
        pass

    def before_epoch(self, runner):
        pass

    def after_epoch(self, runner):
        pass

    def before_iter(self, runner):
        pass
```

(下页继续)

(续上页)

```
    def after_iter(self, runner):
        pass
```

用户需要根据钩子的功能指定钩子在训练各阶段中（before_run，after_run，before_epoch，after_epoch，before_iter，after_iter）做什么。

2. 注册新的钩子（hook）

接下来我们需要导入 MyHook。如果文件的路径是 mmrotate/core/utils/my_hook.py，有两种方式导入：

- 修改 mmrotate/core/utils/__init__.py 文件来导入

  新定义的模块需要在 mmrotate/core/utils/__init__.py 导入，注册表才会发现并添加该模块：

```
from .my_hook import MyHook
```

- 在配置文件中使用 custom_imports 来手动导入

```
custom_imports = dict(imports=['mmrotate.core.utils.my_hook'], allow_failed_
↪imports=False)
```

3. 修改配置

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

您也可以通过配置键值 priority 为 'NORMAL' 或 'HIGHEST' 来设置钩子的优先级：

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

默认地，钩子的优先级在注册时被设置为 NORMAL。

12.4.2 使用 MMCV 中实现的钩子 (hooks)

如果钩子已经在 MMCV 里实现了，您可以直接修改配置文件来使用钩子。

4. 示例: `NumClassCheckHook`

我们实现了一个自定义的钩子 NumClassCheckHook，用来检验 head 中的 `num_classes` 是否与 `dataset` 中的 `CLASSSES` 长度匹配。

我们在 default_runtime.py 中对其进行设置。

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

12.4.3 修改默认运行挂钩

有一些常见的钩子并不通过 `custom_hooks` 注册，这些钩子包括：

- log_config
- checkpoint_config
- evaluation
- lr_config
- optimizer_config
- momentum_config

这些钩子中，只有记录器钩子（logger hook）是 `VERY_LOW` 优先级，其他钩子的优先级为 `NORMAL`。前面提到的教程已经介绍了如何修改 `optimizer_config`，`momentum_config` 以及 `lr_config`。这里我们介绍一下如何处理 `log_config`，`checkpoint_config` 以及 `evaluation`。

Checkpoint config

MMCV runner 将使用 `checkpoint_config` 来初始化 `CheckpointHook`。

```
checkpoint_config = dict(interval=1)
```

用户可以设置 `max_keep_ckpts` 来仅保存一小部分检查点（checkpoint）或者通过设置 `save_optimizer` 来决定是否保存优化器的状态字典 (state dict of optimizer)。更多使用参数的细节请参考 这里。

Log config

log_config 包裹了许多日志钩 (logger hooks) 而且能去设置间隔 (intervals)。现在 MMCV 支持 WandbLoggerHook，MlflowLoggerHook 和 TensorboardLoggerHook。详细的使用请参照 文档。

```
log_config = dict(
    interval=50,
    hooks=[
        dict(type='TextLoggerHook'),
        dict(type='TensorboardLoggerHook')
    ])
```

Evaluation config

evaluation 的配置文件将被用来初始化 EvalHook。除了 interval 键，其他的像 metric 这样的参数将被传递给 dataset.evaluate()。

```
evaluation = dict(interval=1, metric='bbox')
```

CHAPTER 13

Changelog

CHAPTER 14

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。

14.1 MMCV 安装相关

- 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.”

  请按 安装说明 为你的 MMRotate 安装正确版本的 MMCV。

- “No module named ‘mmcv.ops’”；“No module named ‘mmcv._ext’”.

  原因是安装了 `mmcv` 而不是 `mmcv-full`。

  1. 使用 `pip uninstall mmcv` 卸载。
  2. 根据 安装说明 安装 `mmcv-full`。

14.2 PyTorch/CUDA 环境相关

- “RTX 30 series card fails when building MMCV or MMDet”

  1. 常见报错信息为 `nvcc fatal: Unsupported gpu architecture 'compute_86'` 意思是你的编译器应该为 sm_86 进行优化，例如，英伟达 30 系列的显卡，但这样的优化 CUDA toolkit 11.0 并不支持。此解决方案通过添加 `MMCV_WITH_OPS=1 MMCV_CUDA_ARGS='-gencode=arch=compute_80,code=sm_80' pip install -e .` 来修改编译标志，这告诉编译器 `nvcc` 为 **sm_80** 进行优化，例如 Nvidia A100，尽管 A100 不同于 30 系列的显卡，但他们使用相似的图灵架构。这种解决方案可能会丧失一些性能但的确有效。

  2. PyTorch 开发者已经在 pytorch/pytorch#47585 更新了 PyTorch 默认的编译标志，所以使用 PyTorch-nightly 可能也能解决这个问题，但是我们对此并没有验证这种方式是否有效。

- “invalid device function” or “no kernel image is available for execution”.

  1. 检查您的 cuda 运行时版本 (一般在 `/usr/local/`)、指令 `nvcc --version` 显示的版本以及 `conda list cudatoolkit` 指令显式的版本是否匹配。

  2. 通过运行 `python mmdet/utils/collect_env.py` 来检查是否为当前的 GPU 架构编译了正确的 PyTorch、torchvision 和 MMCV，你可能需要设置 `TORCH_CUDA_ARCH_LIST` 来重新安装 MMCV。可以参考 GPU 架构表，即通过运行 `TORCH_CUDA_ARCH_LIST=7.0 pip install mmcv-full` 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，例如，Tesla K80。

  3. 检查运行环境是否与 mmcv/mmdet 编译时相同，例如，您可能使用 CUDA 10.0 编译 MMCV，但在 CUDA 9.0 环境中运行它。

- “undefined symbol” or “cannot open xxx.so”.

  1. 如果这些 symbols 属于 CUDA/C++ (例如，libcudart.so 或者 GLIBCXX)，检查 CUDA/GCC 运行时环境是否与编译 MMCV 的一致。例如使用 `python mmdet/utils/collect_env.py` 检查 `"MMCV Compiler"/"MMCV CUDA Compiler"` 是否和 `"GCC"/"CUDA_HOME"` 一致。

  2. 如果这些 symbols 属于 PyTorch，(例如，symbols containing caffe、aten 和 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。您需要先手动安装 Cpython 然后运命令 `pip install -r requirements.txt`。

  2. 检查环境中的 `setuptools`、`Cython` 和 `PyTorch` 相互之间版本是否匹配。

- “Segmentation fault”.

1. 检查 GCC 的版本并使用 GCC 5.4，通常是因为 PyTorch 版本与 GCC 版本不匹配（例如，对于 Pytorch 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. 检查是是否 PyTorch 被正确的安装并可以使用 CUDA 算子，例如在终端中键入如下的指令。

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

   并判断是否是否返回 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'
```

   如果 MMCV 被正确的安装了，那么上面的两条指令不会有问题。

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`。

14.3 E2CNN

- “ImportError: cannot import name ‘container_bacs’ from ‘torch._six’”

  1. 这是因为 `container_abcs` 在 PyTorch 1.9 之后被移除.

  2. 将文件 `python3.7/site-packages/e2cnn/nn/modules/module_list.py` 中的

```
from torch.six import container_abcs
```

     替换成

```
TORCH_MAJOR = int(torch.__version__.split('.')[0])
TORCH_MINOR = int(torch.__version__.split('.')[1])
if TORCH_MAJOR ==1 and TORCH_MINOR < 8:
    from torch.six import container_abcs
else:
    import collections.abs as container_abcs
```

  3. 或者降低 Pytorch 的版本。

14.4 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. 通过在配置文件中设置 `fp16 = dict(loss_scale='dynamic')` 来尝试混合精度训练。

- “RuntimeError: Expected to have finished reduction in the prior iteration before starting a new one”

  1. 错误表明，您的模块有没用于产生损失的参数，这种现象可能是由于在 DDP 模式下运行代码中的不同分支造成的。
  2. 您可以在配置中设置 `find_unused_parameters = True` 来解决上述问题，或者手动查找那些未使用的参数。

14.5 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物体即 GT，AP 与 AR 将被设置为 -1。

CHAPTER 15

English

CHAPTER 16

简体中文

CHAPTER 17

mmrotate

17.1 mmrotate.models

17.1.1 backbones

17.1.2 dense_heads

17.1.3 detectors

17.1.4 losses

17.1.5 roi_heads

17.2 mmrotate.core

17.3 mmrotate.datasets

class mmrotate.datasets.**DOTADataset**(*ann_file*, *pipeline*, *version='oc'*, *difficulty=100*, ***kwargs*)

DOTA dataset for detection.

参数

- **ann_file** (*str*) –Annotation file path.

- **pipeline** (*list[dict]*) –Processing pipeline.
- **version** (*str, optional*) –Angle representations. Defaults to 'oc'.
- **difficulty** (*bool, optional*) –The difficulty threshold of GT.

evaluate (*results*, *metric='mAP'*, *logger=None*, *proposal_nums=(100, 300, 1000)*, *iou_thr=0.5*, *scale_ranges=None*, *nproc=4*)

Evaluate the dataset.

参数

- **results** (*list*) –Testing results of the dataset.
- **metric** (*str | list[str]*) –Metrics to be evaluated.
- **logger** (*logging.Logger | None | str*) –Logger used for printing related information during evaluation. Default: None.
- **proposal_nums** (*Sequence[int]*) –Proposal number used for evaluating recalls, such as recall@100, recall@1000. Default: (100, 300, 1000).
- **iou_thr** (*float | list[float]*) –IoU threshold. It must be a float when evaluating mAP, and can be a list when evaluating recall. Default: 0.5.
- **scale_ranges** (*list[tuple] | None*) –Scale ranges for evaluating mAP. Default: None.
- **nproc** (*int*) –Processes used for computing TP and FP. Default: 4.

format_results (*results*, *submission_dir=None*, *nproc=4*, ***kwargs*)

Format the results to submission text (standard format for DOTA evaluation).

参数

- **results** (*list*) –Testing results of the dataset.
- **submission_dir** (*str, optional*) –The folder that contains submission
- **files.** –If not specified, a temp folder will be created. Default: None.
- **nproc** (*int, optional*) –number of process.

返回

(result_files, tmp_dir), result_files is a dict containing the json filepaths, tmp_dir is the temporal directory created for saving json files when submission_dir is not specified.

返回类型 tuple

load_annotations (*ann_folder*)

Params: ann_folder: folder that contains DOTA v1 annotations txt files

merge_det(*results*, *nproc=4*)

Merging patch bboxes into full image.

Params: results (list): Testing results of the dataset. nproc (int): number of process. Default: 4.

class mmrotate.datasets.**HRSCDataset**(*ann_file*, *pipeline*, *img_subdir='JPEGImages'*, *ann_subdir='Annotations'*, *classwise=False*, *version='oc'*, ***kwargs*)

HRSC dataset for detection.

参数

- **ann_file** (*str*) –Annotation file path.
- **pipeline** (*list[dict]*) –Processing pipeline.
- **img_subdir** (*str*) –Subdir where images are stored. Default: JPEGImages.
- **ann_subdir** (*str*) –Subdir where annotations are. Default: Annotations.
- **classwise** (*bool*) –Whether to use all classes or only ship.
- **version** (*str, optional*) –Angle representations. Defaults to 'oc'.

evaluate(*results*, *metric='mAP'*, *logger=None*, *proposal_nums=(100, 300, 1000)*, *iou_thr=0.5*, *scale_ranges=None*, *use_07_metric=True*, *nproc=4*)

Evaluate the dataset.

参数

- **results** (*list*) –Testing results of the dataset.
- **metric** (*str | list[str]*) –Metrics to be evaluated.
- **logger** (*logging.Logger | None | str*) –Logger used for printing related information during evaluation. Default: None.
- **proposal_nums** (*Sequence[int]*) –Proposal number used for evaluating recalls, such as recall@100, recall@1000. Default: (100, 300, 1000).
- **iou_thr** (*float | list[float]*) –IoU threshold. It must be a float when evaluating mAP, and can be a list when evaluating recall. Default: 0.5.
- **scale_ranges** (*list[tuple] | None*) –Scale ranges for evaluating mAP. Default: None.
- **use_07_metric** (*bool*) –Whether to use the voc07 metric.
- **nproc** (*int*) –Processes used for computing TP and FP. Default: 4.

load_annotations(*ann_file*)

Load annotation from XML style ann_file.

参数 **ann_file** (*str*) –Path of Imageset file.

返回 Annotation info from XML file.

返回类型 list[dict]

class mmrotate.datasets.**SARDataset**(*ann_file*, *pipeline*, *version='oc'*, *difficulty=100*, ***kwargs*)

SAR ship dataset for detection (Support RSSDD and HRSID).

CHAPTER 18

Indices and tables

- genindex
- search

Python 模块索引

m

mmrotate.core, 65
mmrotate.datasets, 65

索引

D

DOTADataset (*mmrotate.datasets* 中的类), 65

E

evaluate() (*mmrotate.datasets.DOTADataset* 方法), 66

evaluate() (*mmrotate.datasets.HRSCDataset* 方法), 67

F

format_results() (*mmrotate.datasets.DOTADataset* 方法), 66

H

HRSCDataset (*mmrotate.datasets* 中的类), 67

L

load_annotations() (*mmrotate.datasets.DOTADataset* 方法), 66

load_annotations() (*mmrotate.datasets.HRSCDataset* 方法), 67

M

merge_det() (*mmrotate.datasets.DOTADataset* 方法), 66

mmrotate.core
    模块, 65

mmrotate.datasets
    模块, 65

S

SARDataset (*mmrotate.datasets* 中的类), 68

模块
    mmrotate.core, 65
    mmrotate.datasets, 65