mmrotate

MMRotate Author

2023 年 07 月 02 日

学习基础知识

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, width, height)`。
- 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, width, height, 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]，width 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 width > height。
- D_{le90}：长边 90° 定义法，angle∈[-90°, 90°)，theta∈[-pi / 2, pi / 2) 并且 width > height。

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：

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

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

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

在本节中，我们将演示如何使用 PyTorch 准备环境。

MMRotate 能够在 Linux 和 Windows 上运行。它依赖于 Python 3.7+, CUDA 9.2+ 和 PyTorch 1.6+。

注解： 如果您对 PyTorch 有经验并且已经安装了它，只需跳过此部分并跳到下一节 。否则，您可以按照以下步骤进行准备。

第 0 步： 从 官网 下载并安装 Miniconda。

第 1 步： 创建一个 conda 环境并激活它.

```
conda create --name openmmlab python=3.8 -y
conda activate openmmlab
```

第 2 步： 根据 官方说明 安装 PyTorch, 例如：

```
conda install pytorch==1.8.0 torchvision==0.9.0 cudatoolkit=10.2 -c pytorch
```

CHAPTER 3

安装

我们建议用户按照我们的最佳实践安装 MMRotate。然而，整个过程是高度可定制的。有关详细信息，请参阅自定义安装 部分。

3.1 最佳实践

第 0 步：使用 MIM 安装 MMCV 和 MMDetection

```
pip install -U openmim
mim install mmcv-full
mim install mmdet\<3.0.0
```

第 1 步：安装 MMRotate.

案例 a：如果您直接开发并运行 mmrotate，请从源代码安装：

```
git clone https://github.com/open-mmlab/mmrotate.git
cd mmrotate
pip install -v -e .
# "-v" 表示详细或更多输出
# "-e" 表示以可编辑模式安装项目，
# 因此，对代码进行的任何本地修改都将在不重新安装的情况下生效。
```

案例 b：如果将 mmrotate 作为依赖项或第三方软件包，请使用 pip 安装它：

```
pip install mmrotate
```

3.2 验证

为了验证是否正确安装了 MMRotate，我们提供了一些示例代码来运行推理演示。

第 1 步：我们需要下载配置文件和检查点文件。

```
mim download mmrotate --config oriented_rcnn_r50_fpn_1x_dota_le90 --dest .
```

下载需要几秒钟或更长时间，具体取决于您的网络环境。当下载完成之后，您将会在当前文件夹下找到 `oriented_rcnn_r50_fpn_1x_dota_le90.py` 和 `oriented_rcnn_r50_fpn_1x_dota_le90-6d2b2ce0.pth` 这两个文件。

第 2 步：验证推理演示

选项（a）：如果从源代码安装 mmrotate，只需运行以下命令。

```
python demo/image_demo.py demo/demo.jpg oriented_rcnn_r50_fpn_1x_dota_le90.py␣
↪oriented_rcnn_r50_fpn_1x_dota_le90-6d2b2ce0.pth --out-file result.jpg
```

您将在当前目录下看到一张名为 `result.jpg` 的新图片，其中旋转边界框绘制在汽车、公共汽车等目标上。

选项（b）：如果使用 pip 安装 mmrotate，请打开 python 解释器并复制和粘贴以下代码。

```
from mmdet.apis import init_detector, inference_detector
import mmrotate

config_file = 'oriented_rcnn_r50_fpn_1x_dota_le90.py'
checkpoint_file = 'oriented_rcnn_r50_fpn_1x_dota_le90-6d2b2ce0.pth'
model = init_detector(config_file, checkpoint_file, device='cuda:0')
inference_detector(model, 'demo/demo.jpg')
```

您将看到打印的数组列表，表示检测到的旋转边界框。

3.3 自定义安装

3.3.1 CUDA 版本

安装 PyTorch 时，需要指定 CUDA 的版本。如果您不清楚选择哪一个，请遵循我们的建议：

- 对于基于安培架构的 NVIDIA GPU，如 GeForce 30 系列和 NVIDIA A100，必须使用 CUDA 11。
- 对于较旧的 NVIDIA GPU，CUDA 11 向后兼容，但 CUDA 10.2 更轻量并且具有更好的兼容性。

请确保 GPU 驱动程序满足最低版本要求。请查询 表格 以获得更多信息。

注解： 如果遵循我们的最佳实践，安装 CUDA 运行时库就足够了，因为不会在本地编译 CUDA 代码。但是，如果您希望从源代码处编译 MMCV 或开发其他 CUDA 算子，则需要从 NVIDIA 的 网站 安装完整的 CUDA 工具包，其版本应与 PyTorch 的 CUDA 版本匹配。例如使用 `conda install` 命令指定 cudatoolkit 的版本。

3.3.2 不使用 MIM 安装 MMCV

MMCV 包含 C++ 和 CUDA 扩展，因此以复杂的方式依赖于 PyTorch。MIM 会自动解决此类依赖关系，并使安装更容易。然而，这不是必须的。

要使用 pip 而不是 MIM 安装 MMCV，请遵循 MMCV 安装指南 。这需要根据 PyTorch 版本及其 CUDA 版本手动指定 find-url。

例如, 以下命令安装了为 PyTorch 1.9.x 和 CUDA 10.2 构建的 mmcv-full。

```
pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu102/torch1.8/
↪index.html
```

3.3.3 在 Google Colab 安装

Google Colab 通常已经安装了 PyTorch，因此，我们只需要使用以下命令安装 MMCV 和 MMDetection。

第 1 步： 使用 MIM 安装 MMCV 和 MMDetection 。

```
!pip3 install -U openmim
!mim install mmcv-full
!mim install mmdet
```

第 2 步： 从源码安装 MMRotate。

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

第 3 步： 验证。

```
import mmrotate
print(mmrotate.__version__)
# Example output: 0.3.1
```

注解： 在 Jupyter 中，感叹号 ！用于调用外部可执行文件，`%cd` 是一个 魔术命令 用于更改 Python 的当前工作目录。

3.3.4 在 Docker 镜像中使用 MMRotate

我们提供了 Dockerfile 用于创建镜像。请确保您的 docker 版本 >=19.03。

```
# build an image with PyTorch 1.6, CUDA 10.1
# If you prefer other versions, just modified the Dockerfile
docker build -t mmrotate docker/
```

使用下列命令运行

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

3.4 故障排除

如果您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，您可以在 GIthub 上 提问 。

CHAPTER 4

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 5

测试一个模型

- 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 并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 6

训练一个模型

6.1 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}`。

6.2 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。

6.3 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。

6.4 使用 Slurm 来管理任务

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。

6.5 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 7

基准和模型库

- Rotated RetinaNet-OBB/HBB (ICCV’ 2017)
- Rotated FasterRCNN-OBB (TPAMI’ 2017)
- Rotated RepPoints-OBB (ICCV’ 2019)
- Rotated FCOS (ICCV’ 2019)
- RoI Transformer (CVPR’ 2019)
- Gliding Vertex (TPAMI’ 2020)
- Rotated ATSS-OBB (CVPR’ 2020)
- CSL (ECCV’ 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)

7.1 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/2080ti 训练得到的，并且在 1 * 2080ti 上进行推理测试。

CHAPTER 8

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。

8.1 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。

8.2 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`。

8.3 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
```

8.4 常见问题 (FAQ)

8.4.1 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 9

教程 2：自定义数据集

9.1 支持新的数据格式

要支持新的数据格式，您可以将它们转换为现有的格式（DOTA 格式）。您可以选择离线（在通过脚本训练之前）或在线（实施新数据集并在训练时进行转换）进行转换。在 MMRotate 中，我们建议将数据转换为 DOTA 格式并离线进行转换，如此您只需在数据转换后修改 config 的数据标注路径和类别即可。

9.1.1 将新数据格式重构为现有格式

最简单的方法是将数据集转换为现有数据集格式 (DOTA) 。

DOTA 格式的注解 txt 文件：

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

每行代表一个对象，并将其记录为一个 10 维数组 `A` 。

- `A[0:8]`: 多边形的格式 `(x1, y1, x2, y2, x3, y3, x4, y4)` 。
- `A[8]`: 类别
- `A[9]`: 困难

在数据预处理之后，用户可以通过两个步骤来训练具有现有格式（例如 DOTA 格式）的自定义新数据集：

1. 修改配置文件以使用自定义数据集。

2. 检查自定义数据集的标注。

下面给出两个例子展示上述两个步骤，它使用一个自定义的 5 类 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. 修改配置文件以使用自定义数据集

配置文件的修改主要涉及两个方面:

1. `data` 部分。具体来说，您需要在 `data.train`, `data.val` 和 `data.test` 中显式添加 classes 字段。
2. `model` 部分中的 `num_classes` 属性变量。特别是将所有 num_classes 的默认值（例如 COCO 中的 80）覆盖到您的类别编号中。

在 `configs/my_custom_config.py`：

```
# 新配置继承了基础配置用于突出显示必要的修改
_base_ = './rotated_retinanet_hbb_r50_fpn_1x_dota_oc'

# 1. 数据集的设置
dataset_type = 'DOTADataset'
classes = ('a', 'b', 'c', 'd', 'e')
data = dict(
    samples_per_gpu=2,
    workers_per_gpu=2,
    train=dict(
        type=dataset_type,

        # 注意将你的类名添加到字段 `classes`
        classes=classes,
        ann_file='path/to/your/train/annotation_data',
        img_prefix='path/to/your/train/image_data'),
    val=dict(
        type=dataset_type,

        # 注意将你的类名添加到字段 `classes`
        classes=classes,
        ann_file='path/to/your/val/annotation_data',
        img_prefix='path/to/your/val/image_data'),
    test=dict(
        type=dataset_type,

        # 注意将你的类名添加到字段 `classes`
        classes=classes,
        ann_file='path/to/your/test/annotation_data',
```

(下页继续)

(续上页)

```
        img_prefix='path/to/your/test/image_data'))

# 2. 模型设置
model = dict(
    bbox_head=dict(
        type='RotatedRetinaHead',
        # 显式将所有 `num_classes` 字段从 15 重写为 5。。
        num_classes=15))
```

2. 查看自定义数据集的标注

假设您的自定义数据集是 DOTA 格式，请确保您在自定义数据集中具有正确的标注：

- 配置文件中的 `classes` 字段应该与 txt 标注的 `A[8]` 保持完全相同的元素和相同的顺序。MMRotate 会自动的将 `categories` 中不连续的 `id` 映射到连续的标签索引中，所以在 `categories` 中 `name` 的字符串顺序会影响标签索引的顺序。同时，配置文件中 `classes` 的字符串顺序也会影响预测边界框可视化过程中的标签文本信息。

9.2 通过封装器自定义数据集

MMRotate 还支持许多数据集封装器对数据集进行混合或修改数据集的分布以进行训练。目前它支持三个数据集封装器，如下所示：

- `RepeatDataset`: 简单地重复整个数据集。
- `ClassBalancedDataset`: 以类平衡的方式重复数据集。
- `ConcatDataset`: 拼接数据集。

9.2.1 重复数据集

我们使用 `RepeatDataset` 作为封装器来重复这个数据集。例如，假设原始数据集是 `Dataset_A`，我们就重复一遍这个数据集。配置信息如下所示：

```
dataset_A_train = dict(
        type='RepeatDataset',
        times=N,
        dataset=dict(  # 这是 Dataset_A 的原始配置信息
            type='Dataset_A',
            ...
            pipeline=train_pipeline
```

(下页继续)

(续上页)

```
        )
    )
```

9.2.2 类别平衡数据集

我们使用 ClassBalancedDataset 作为封装器，根据类别频率重复数据集。这个数据集的重复操作 ClassBalancedDataset 需要实例化函数 self.get_cat_ids(idx) 的支持。例如，Dataset_A 需要使用 oversample_thr=1e-3，配置信息如下所示：

```
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
        )
    )
```

9.2.3 拼接数据集

这里用三种方式对数据集进行拼接。

1. 如果要拼接的数据集属于同一类型且具有不同的标注文件，则可以通过如下所示的配置信息来拼接数据集：

```
dataset_A_train = dict(
    type='Dataset_A',
    ann_file = ['anno_file_1', 'anno_file_2'],
    pipeline=train_pipeline
)
```
```如果拼接后的数据集用于测试或评估，我们这种方式是可以支持对每个数据集分别进行评估。如果要测试的整个拼接数据集，如下所示您可以直接通过设置 separate_eval=False 来实现。

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

2. 如果您要拼接不同的数据集，您可以通过如下所示的方法对拼接数据集配置信息进行设置。

```
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
    )
```

```如果拼接后的数据集用于测试或评估，这种方式还支持对每个数据集分别进行评估。

3. 我们也支持如下所示的方法对 ConcatDataset 进行明确的定义。

```
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))
```

```这种方式允许用户通过设置 `separate_eval=False` 将所有数据集转为单个数据集进行评估。

**笔记:**

1. 假设数据集在评估期间使用 self.data_infos，就要把选项设置为 separate_eval=False。因为 COCO 数据集不完全依赖 self.data_infos 进行评估，所以 COCO 数据集并不支持这种设置操作。没有在组合不同类型的数据集并对其进行整体评估的场景进行测试，因此我们不建议使用这样的操作。
2. 不支持评估 ClassBalancedDataset 和 RepeatDataset，所以也不支持评估这些类型的串联组合后的数据集。

一个更复杂的例子，分别将 Dataset_A 和 Dataset_B 重复 N 次和 M 次，然后将重复的数据集连接起来，如下所示。

```
dataset_A_train = dict(
    type='RepeatDataset',
```

(下页继续)

(续上页)

```
    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 10

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。

10.1 开发新的组件

10.1.1 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),
    ...
```

10.1.2 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)
```

10.1.3 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。

10.1.4 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 11

教程 4: 自定义训练设置

11.1 自定义优化设置

11.1.1 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。

11.1.2 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)
```

11.1.3 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。

11.1.4 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,
)
```

11.2 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)
```

11.3 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。

11.4 自定义钩 (hooks)

11.4.1 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。

11.4.2 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')]
```

11.4.3 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')
```

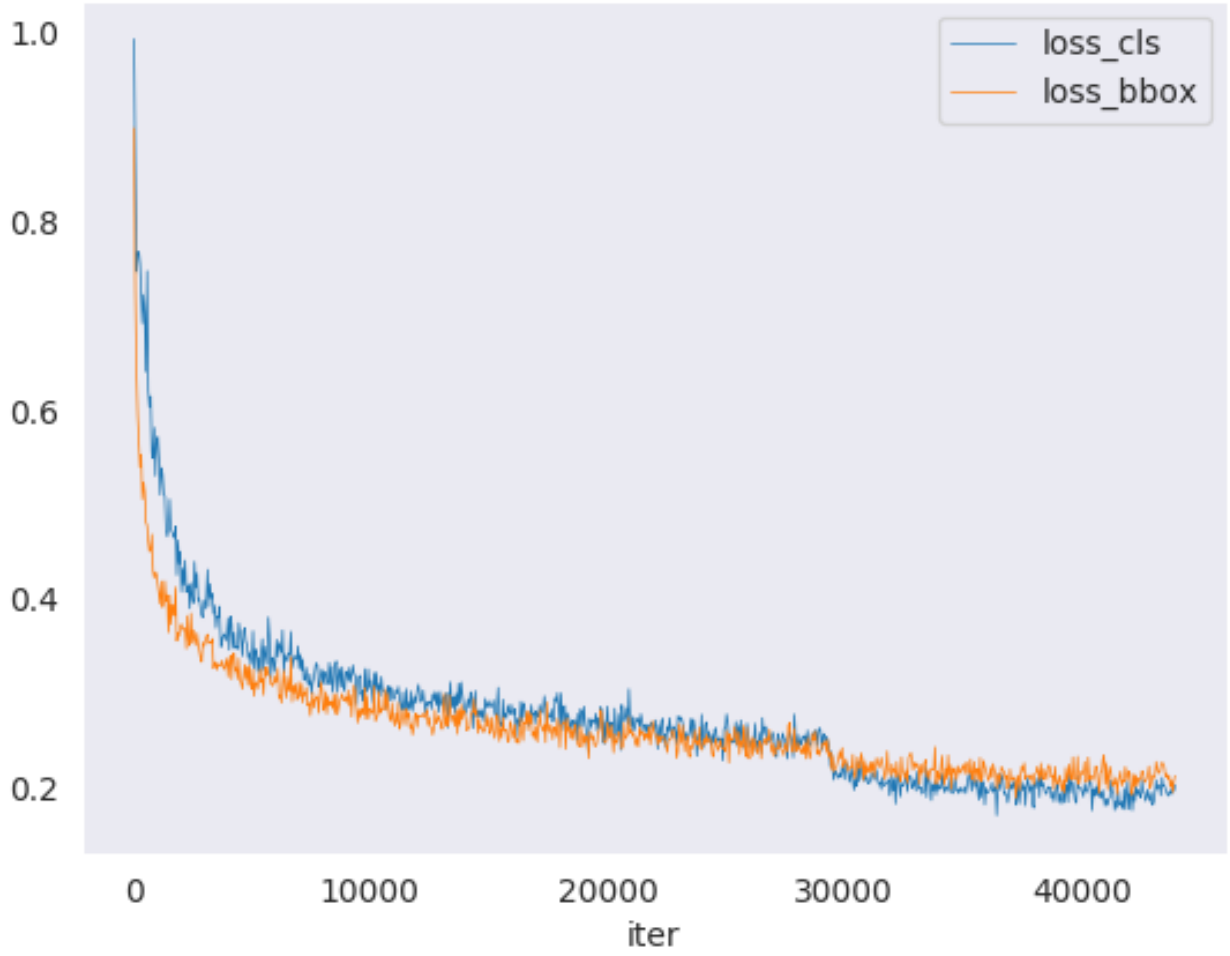
除了训练和测试脚本，我们在 tools/ 文件夹内还提供了一些有用的工具。

CHAPTER 12

日志分析

tools/analysis_tools/analyze_logs.py 通过给定的日志文件绘制 loss/mAP 曲线。需首先执行 pip install seaborn 安装依赖。

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



示例:

- 绘制某次执行的分类损失

```
python tools/analysis_tools/analyze_logs.py plot_curve log.json --keys loss_cls --
↪legend loss_cls
```

- 绘制某次执行的分类和回归损失，同时将图像保存到 pdf 文件

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

- 在同一张图像中比较两次执行的 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]
```

预计输出如下

```
-----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
```

CHAPTER 13

可视化

13.1 可视化数据集

tools/misc/browse_dataset.py 帮助用户浏览检测的数据集（包括图像和检测框的标注），或将图像保存到指定目录。

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

CHAPTER 14

模型部署

为了使用 TorchServe 部署一个 MMRotate 模型，需要进行以下几步：

14.1 1. 转换 MMRotate 模型至 TorchServe

```
python tools/deployment/mmrotate2torchserve.py ${CONFIG_FILE} ${CHECKPOINT_FILE} \
--output-folder ${MODEL_STORE} \
--model-name ${MODEL_NAME}
```

示例:

```
wget -P checkpoint  \
https://download.openmmlab.com/mmrotate/v0.1.0/rotated_faster_rcnn/rotated_faster_
↪rcnn_r50_fpn_1x_dota_le90/rotated_faster_rcnn_r50_fpn_1x_dota_le90-0393aa5c.pth

python tools/deployment/mmrotate2torchserve.py configs/rotated_faster_rcnn/rotated_
↪faster_rcnn_r50_fpn_1x_dota_le90.py checkpoint/rotated_faster_rcnn_r50_fpn_1x_dota_
↪le90-0393aa5c.pth \
--output-folder ${MODEL_STORE} \
--model-name rotated_faster_rcnn
```

Note: ${MODEL_STORE} 需要是一个文件夹的绝对路径。

14.2 2. 构建 mmrotate-serve docker 镜像

```
docker build -t mmrotate-serve:latest docker/serve/
```

14.3 3. 运行 mmrotate-serve 镜像

请参考官方文档 基于 docker 运行 TorchServe.

为了使镜像能够使用 GPU 资源，需要安装 nvidia-docker。之后可以传递 --gpus 参数以在 GPU 上运行。

示例：

```
docker run --rm \
--cpus 8 \
--gpus device=0 \
-p8080:8080 -p8081:8081 -p8082:8082 \
--mount type=bind,source=$MODEL_STORE,target=/home/model-server/model-store \
mmrotate-serve:latest
```

参考 该文档 了解关于推理 (8080)，管理 (8081) 和指标 (8082) 等 API 的信息。

14.4 4. 测试部署

```
curl -O https://raw.githubusercontent.com/open-mmlab/mmrotate/main/demo/demo.jpg
curl http://127.0.0.1:8080/predictions/${MODEL_NAME} -T demo.jpg
```

您应该获得类似于以下内容的响应：

```
[
  {
    "class_name": "small-vehicle",
    "bbox": [
      584.9473266601562,
      327.2749938964844,
      38.45665740966797,
      16.898427963256836,
      -0.7229751944541931
    ],
    "score": 0.9766026139259338
  },
  {
```

(下页继续)

(续上页)


```
    "class_name": "small-vehicle",
    "bbox": [
      152.0239715576172,
      305.92572021484375,
      43.144744873046875,
      18.85024642944336,
      0.014928221702575684
    ],
    "score": 0.972826361656189
  },
  {
    "class_name": "large-vehicle",
    "bbox": [
      160.58056640625,
      437.3690185546875,
      55.6795654296875,
      19.31710433959961,
      0.007036328315734863
    ],
    "score": 0.888836681842804
  },
  {
    "class_name": "large-vehicle",
    "bbox": [
      666.2868041992188,
      1011.3961181640625,
      60.396209716796875,
      21.821645736694336,
      0.8549195528030396
    ],
    "score": 0.8240180015563965
  }
]
```


另外，你也可以使用 `test_torchserver.py` 来比较 TorchServe 和 PyTorch 的结果，并进行可视化。

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

示例：

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

(下页继续)

(续上页)

```
configs/rotated_faster_rcnn/rotated_faster_rcnn_r50_fpn_1x_dota_le90.py \
rotated_faster_rcnn_r50_fpn_1x_dota_le90-0393aa5c.pth \
rotated_fater_rcnn
```

CHAPTER 15

模型复杂度

tools/analysis_tools/get_flops.py 是改编自 flops-counter.pytorch 的脚本，用于计算给定模型的 FLOPs 和参数量。

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

预计输出如下

```
==============================
Input shape: (3, 1024, 1024)
Flops: 215.92 GFLOPs
Params: 36.42 M
==============================
```

注意: 此工具仍处于实验阶段，我们并不能保证计算结果是绝对正确的。你可以将结果用于简单的比较，但在技术报告或论文中采用之前请仔细检查

1. FLOPs 与输入大小相关，但参数量与其无关。默认输入大小是 (1, 3, 1024, 1024)。
2. 一些算子例如 DCN 或自定义算子并未包含在 FLOPs 计算中，所以 S2A-Net 和基于 RepPoints 的模型的 FLOPs 计算是错误的。详细信息请查看 `mmcv.cnn.get_model_complexity_info()`。
3. 两阶段检测器的 FLOPs 取决于候选的数量。

15.1 准备发布模型

tools/model_converters/publish_model.py 帮助用户准备他们将发布的模型。

在将模型上传到 AWS 之前，你可能需要

1. 将模型权重转换至 CPU
2. 删除优化器的状态
3. 计算权重文件的哈希值并附加到文件名后

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

例如,

```
python tools/model_converters/publish_model.py work_dirs/rotated_faster_rcnn/latest.
↪pth rotated_faster_rcnn_r50_fpn_1x_dota_le90_20190801.pth
```

最终输出的文件名是 rotated_faster_rcnn_r50_fpn_1x_dota_le90_20190801-{hash id}.pth。

CHAPTER 16

基准测试

16.1 FPS 基准

`tools/analysis_tools/benchmark.py` 帮助用户计算 FPS。FPS 值包括模型前向传播和后处理。为了得到更准确的数值，目前只支持单 GPU 分布式启动。

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

示例: 假设你已经下载了 `Rotated Faster R-CNN` 模型权重到 `checkpoints/` 文件夹

```
python -m torch.distributed.launch --nproc_per_node=1 --master_port=29500 tools/
↪analysis_tools/benchmark.py \
       configs/rotated_faster_rcnn/rotated_faster_rcnn_r50_fpn_1x_dota_le90.py \
       checkpoints/rotated_faster_rcnn_r50_fpn_1x_dota_le90-0393aa5c.pth \
       --launcher pytorch
```

CHAPTER 17

杂项

17.1 打印完整配置文件

tools/misc/print_config.py 输出整个配置文件并整合其所有导入。

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

CHAPTER 18

混淆矩阵

混淆矩阵是预测结果的概要

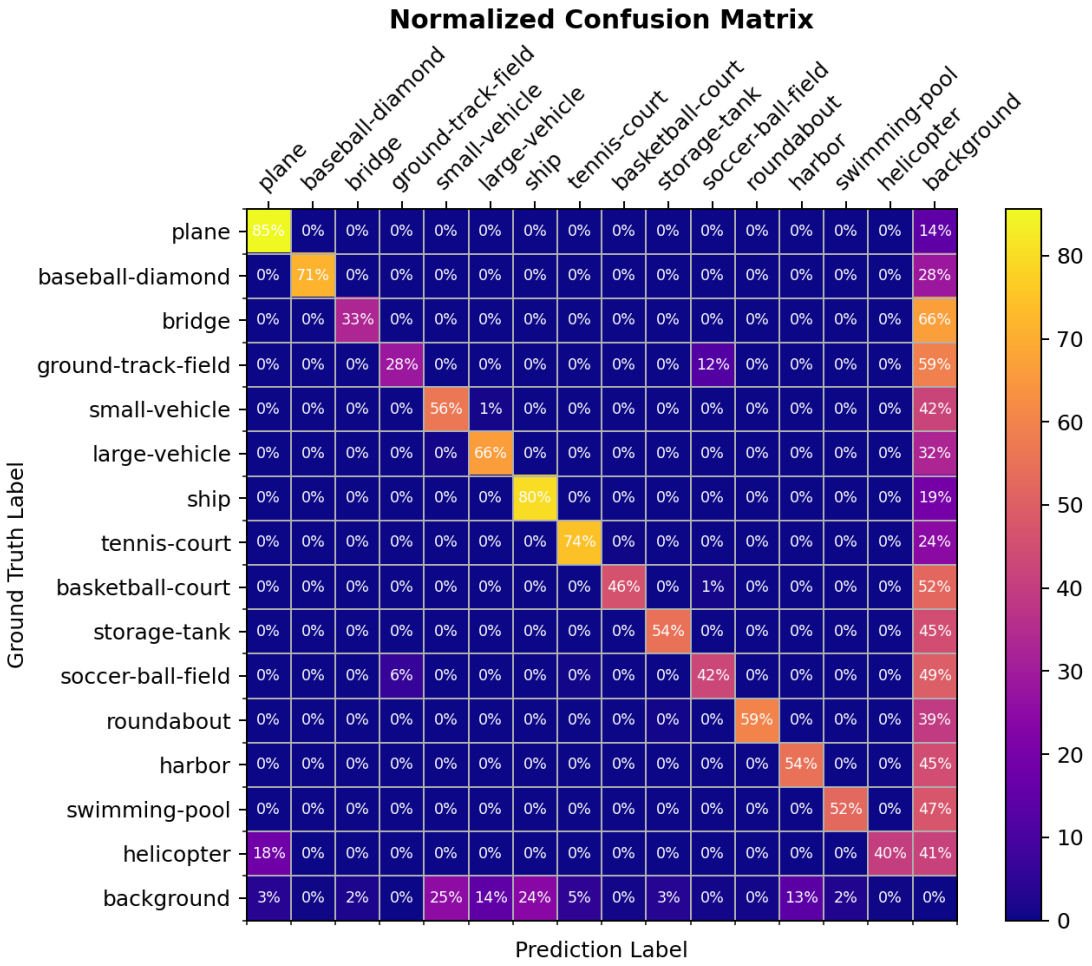
`tools/analysis_tools/confusion_matrix.py` 可以分析预测结果并绘制混淆矩阵。

首先执行 `tools/test.py` 将检测结果保存为 `.pkl` 文件。

之后执行

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

你会得到一个类似于下图的混淆矩阵:


Normalized Confusion Matrix
plane
baseball-diamond
bridge
ground-track-field
small-vehicle
large-vehicle
ship
tennis-court
basketball-court
storage-tank
soccer-ball-field
roundabout
harbor
swimming-pool
helicopter
background
Ground Truth Label
Prediction Label
80
70
60
50
40
30
20
10
0

CHAPTER 19

Changelog

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 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.”

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。

- “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`。

20.2 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`。

20.3 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。

20.4 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。

20.5 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 21

English

CHAPTER 22

简体中文

CHAPTER 23

mmrotate.apis

CHAPTER 24

mmrotate.core

CHAPTER 25

mmrotate.datasets

25.1 datasets

25.2 pipelines

CHAPTER 26

mmrotate.models

CHAPTER 27

mmrotate.utils

CHAPTER 28

Indices and tables

- genindex
- search