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 Train a model 19
8 Benchmark and Model Zoo 21
9 Tutorial 1: Learn about Configs 23
10 Tutorial 2: Customize Datasets 33
11 Tutorial 3: Customize Models 39
12 Tutorial 4: Customize Runtime Settings 47
13 Changelog 55
14 Frequently Asked Questions 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

- single GPU
- single node multiple GPU
- multiple node

You can use the following commands to infer a dataset.

```
# single-gpu
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments]

# multi-gpu
./tools/dist_test.sh ${CONFIG_FILE} ${CHECKPOINT_FILE} ${GPU_NUM} [optional arguments]

# multi-node in slurm environment
python tools/test.py ${CONFIG_FILE} ${CHECKPOINT_FILE} [optional arguments] --
↪launcher slurm
```

Examples:

Inference rotated RetinaNet on DOTA-1.0 dataset. (Please change the data_root firstly.)

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

You can also visualize the results.

```
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
```

Further, you can also generate compressed files for online submission.

```
python ./tools/test.py  \
  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
```

CHAPTER 7

Train a model

7.1 Train with a single GPU

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

If you want to specify the working directory in the command, you can add an argument `--work_dir ${YOUR_WORK_DIR}`.

7.2 Train with multiple GPUs

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

Optional arguments are:

- `--no-validate` (**not suggested**): By default, the codebase will perform evaluation during the training. To disable this behavior, use `--no-validate`.
- `--work-dir ${WORK_DIR}`: Override the working directory specified in the config file.
- `--resume-from ${CHECKPOINT_FILE}`: Resume from a previous checkpoint file.

Difference between `resume-from` and `load-from`: `resume-from` loads both the model weights and optimizer status, and the epoch is also inherited from the specified checkpoint. It is usually used for resuming the training process that is interrupted accidentally. `load-from` only loads the model weights and the training epoch starts from 0. It is usually used for finetuning.

7.3 Train with multiple machines

If you run MMRotate on a cluster managed with slurm, you can use the script `slurm_train.sh`. (This script also supports single machine training.)

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

If you have just multiple machines connected with ethernet, you can refer to PyTorch launch utility. Usually it is slow if you do not have high speed networking like InfiniBand.

7.4 Launch multiple jobs on a single machine

If you launch multiple jobs on a single machine, e.g., 2 jobs of 4-GPU training on a machine with 8 GPUs, you need to specify different ports (29500 by default) for each job to avoid communication conflict.

If you use `dist_train.sh` to launch training jobs, you can set the port in commands.

```
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
```

If you use launch training jobs with Slurm, you need to modify the config files (usually the 6th line from the bottom in config files) to set different communication ports.

In `config1.py`,

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

In `config2.py`,

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

Then you can launch two jobs with `config1.py` ang `config2.py`.

```
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

Benchmark and Model Zoo

- 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 Results on DOTA v1.0

- `MS` means multiple scale image split.
- `RR` means random rotation.

The above models are trained with 1 * 1080Ti and inferred with 1 * 2080Ti.

CHAPTER 9

Tutorial 1: Learn about Configs

We incorporate modular and inheritance design into our config system, which is convenient to conduct various experiments. If you wish to inspect the config file, you may run `python tools/misc/print_config.py /PATH/TO/CONFIG` to see the complete config. The mmrotate is built upon the mmdet, thus it is highly recommended learning the basic of mmdet.

9.1 Modify a config through script arguments

When submitting jobs using "tools/train.py" or "tools/test.py", you may specify `--cfg-options` to in-place modify the config.

- Update config keys of dict chains.

  The config options can be specified following the order of the dict keys in the original config. For example, `--cfg-options model.backbone.norm_eval=False` changes the all BN modules in model backbones to `train` mode.

- Update keys inside a list of configs.

  Some config dicts are composed as a list in your config. For example, the training pipeline `data.train.pipeline` is normally a list e.g. `[dict(type='LoadImageFromFile'), ...]`. If you want to change `'LoadImageFromFile'` to `'LoadImageFromWebcam'` in the pipeline, you may specify `--cfg-options data.train.pipeline.0.type=LoadImageFromWebcam`.

- Update values of list/tuples.

If the value to be updated is a list or a tuple. For example, the config file normally sets `workflow=[('train', 1)]`. If you want to change this key, you may specify `--cfg-options workflow="[(train,1),(val, 1)]"`. Note that the quotation mark " is necessary to support list/tuple data types, and that **NO** white space is allowed inside the quotation marks in the specified value.

9.2 Config file naming convention

We follow the below style to name config files. Contributors are advised to follow the same style.

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

`{xxx}` is required field and `[yyy]` is optional.

- `{model}`: model type like `rotated_faster_rcnn`, `rotated_retinanet`, etc.
- `[model setting]`: specific setting for some model, like `hbb` for `rotated_retinanet`, etc.
- `{backbone}`: backbone type like `r50` (ResNet-50), `swin_tiny` (SWIN-tiny).
- `{neck}`: neck type like `fpn`, `refpn`.
- `[norm_setting]`: `bn` (Batch Normalization) is used unless specified, other norm layer type could be `gn` (Group Normalization), `syncbn` (Synchronized Batch Normalization). `gn-head`/`gn-neck` indicates GN is applied in head/neck only, while `gn-all` means GN is applied in the entire model, e.g. backbone, neck, head.
- `[misc]`: miscellaneous setting/plugins of model, e.g. `dconv`, `gcb`, `attention`, `albu`, `mstrain`.
- `[gpu x batch_per_gpu]`: GPUs and samples per GPU, `1xb2` is used by default.
- `{dataset}`: dataset like `dota`.
- `{angle version}`: like `oc`, `le135` or `le90`.

9.3 An example of RotatedRetinaNet

To help the users have a basic idea of a complete config and the modules in a modern detection system, we make brief comments on the config of RotatedRetinaNet using ResNet50 and FPN as the following. For more detailed usage and the corresponding alternative for each modules, please refer to the API documentation.

```
angle_version = 'oc'  # The angle version
model = dict(
    type='RotatedRetinaNet',  # The name of detector
    backbone=dict(  # The config of backbone
        type='ResNet',  # The type of the backbone
        depth=50,  # The depth of backbone
```

(下页继续)

(续上页)

```
        num_stages=4,  # Number of stages of the backbone.
        out_indices=(0, 1, 2, 3),  # The index of output feature maps produced in␣
↪each stages
        frozen_stages=1,  # The weights in the first 1 stage are fronzen
        zero_init_residual=False,  # Whether to use zero init for last norm layer in␣
↪resblocks to let them behave as identity.
        norm_cfg=dict(  # The config of normalization layers.
            type='BN',  # Type of norm layer, usually it is BN or GN
            requires_grad=True),  # Whether to train the gamma and beta in BN
        norm_eval=True,  # Whether to freeze the statistics in BN
        style='pytorch',  # The style of backbone, 'pytorch' means that stride 2␣
↪layers are in 3x3 conv, 'caffe' means stride 2 layers are in 1x1 convs.
        init_cfg=dict(type='Pretrained', checkpoint='torchvision://resnet50')),  #␣
↪The ImageNet pretrained backbone to be loaded
    neck=dict(
        type='FPN',  # The neck of detector is FPN. We also support 'ReFPN'
        in_channels=[256, 512, 1024, 2048],  # The input channels, this is consistent␣
↪with the output channels of backbone
        out_channels=256,  # The output channels of each level of the pyramid feature␣
↪map
        start_level=1,  # Index of the start input backbone level used to build the␣
↪feature pyramid
        add_extra_convs='on_input',  # It specifies the source feature map of the␣
↪extra convs
        num_outs=5),  # The number of output scales
    bbox_head=dict(
        type='RotatedRetinaHead',# The type of bbox head is 'RRetinaHead'
        num_classes=15,  # Number of classes for classification
        in_channels=256,  # Input channels for bbox head
        stacked_convs=4,  # Number of stacking convs of the head
        feat_channels=256,  # Number of hidden channels
        assign_by_circumhbbox='oc',  # The angle version of obb2hbb
        anchor_generator=dict(  # The config of anchor generator
            type='RotatedAnchorGenerator',  # The type of anchor generator
            octave_base_scale=4,  # The base scale of octave.
            scales_per_octave=3,  #  Number of scales for each octave.
            ratios=[1.0, 0.5, 2.0],  # The ratio between height and width.
            strides=[8, 16, 32, 64, 128]),  # The strides of the anchor generator.␣
↪This is consistent with the FPN feature strides.
        bbox_coder=dict(  # Config of box coder to encode and decode the boxes during␣
↪training and testing
            type='DeltaXYWHAOBBoxCoder',  # Type of box coder.
            angle_range='oc',  # The angle version of box coder.
```

(下页继续)

(续上页)

```
            norm_factor=None,  # The norm factor of box coder.
            edge_swap=False,  # The edge swap flag of box coder.
            proj_xy=False,  # The project flag of box coder.
            target_means=(0.0, 0.0, 0.0, 0.0, 0.0),  # The target means used to␣
↪encode and decode boxes
            target_stds=(1.0, 1.0, 1.0, 1.0, 1.0)),  # The standard variance used to␣
↪encode and decode boxes
        loss_cls=dict(  # Config of loss function for the classification branch
            type='FocalLoss',  # Type of loss for classification branch
            use_sigmoid=True,  #  Whether the prediction is used for sigmoid or␣
↪softmax
            gamma=2.0,  # The gamma for calculating the modulating factor
            alpha=0.25,  # A balanced form for Focal Loss
            loss_weight=1.0),  # Loss weight of the classification branch
        loss_bbox=dict(  # Config of loss function for the regression branch
            type='L1Loss',  # Type of loss
            loss_weight=1.0)),  # Loss weight of the regression branch
    train_cfg=dict(  # Config of training hyperparameters
        assigner=dict(  # Config of assigner
            type='MaxIoUAssigner',  # Type of assigner
            pos_iou_thr=0.5,  # IoU >= threshold 0.5 will be taken as positive samples
            neg_iou_thr=0.4,  # IoU < threshold 0.4 will be taken as negative samples
            min_pos_iou=0,  # The minimal IoU threshold to take boxes as positive␣
↪samples
            ignore_iof_thr=-1,  # IoF threshold for ignoring bboxes
            iou_calculator=dict(type='RBboxOverlaps2D')),  # Type of Calculator for␣
↪IoU
        allowed_border=-1,  # The border allowed after padding for valid anchors.
        pos_weight=-1,  # The weight of positive samples during training.
        debug=False),  # Whether to set the debug mode
    test_cfg=dict(  # Config of testing hyperparameters
        nms_pre=2000,  # The number of boxes before NMS
        min_bbox_size=0,  # The allowed minimal box size
        score_thr=0.05,  # Threshold to filter out boxes
        nms=dict(iou_thr=0.1), # NMS threshold
        max_per_img=2000))  # The number of boxes to be kept after NMS.
dataset_type = 'DOTADataset'  # Dataset type, this will be used to define the dataset
data_root = '../datasets/split_1024_dota1_0/'  # Root path of data
img_norm_cfg = dict(  # Image normalization config to normalize the input images
    mean=[123.675, 116.28, 103.53],  # Mean values used to pre-training the pre-
↪trained backbone models
    std=[58.395, 57.12, 57.375],  # Standard variance used to pre-training the pre-
↪trained backbone models
```

(下页继续)

(续上页)

```
    to_rgb=True)  # The channel orders of image used to pre-training the pre-trained␣
↪backbone models
train_pipeline = [  # Training pipeline
    dict(type='LoadImageFromFile'),  # First pipeline to load images from file path
    dict(type='LoadAnnotations',  # Second pipeline to load annotations for current␣
↪image
         with_bbox=True),  # Whether to use bounding box, True for detection
    dict(type='RResize',  # Augmentation pipeline that resize the images and their␣
↪annotations
         img_scale=(1024, 1024)),  # The largest scale of image
    dict(type='RRandomFlip',  # Augmentation pipeline that flip the images and their␣
↪annotations
         flip_ratio=0.5,  # The ratio or probability to flip
         version='oc'),  # The angle version
    dict(
        type='Normalize',  # Augmentation pipeline that normalize the input images
        mean=[123.675, 116.28, 103.53],  # These keys are the same of img_norm_cfg␣
↪since the
        std=[58.395, 57.12, 57.375],  # keys of img_norm_cfg are used here as␣
↪arguments
        to_rgb=True),
    dict(type='Pad',  # Padding config
         size_divisor=32),  # The number the padded images should be divisible
    dict(type='DefaultFormatBundle'),  # Default format bundle to gather data in the␣
↪pipeline
    dict(type='Collect',  # Pipeline that decides which keys in the data should be␣
↪passed to the detector
         keys=['img', 'gt_bboxes', 'gt_labels'])
]
test_pipeline = [
    dict(type='LoadImageFromFile'),  # First pipeline to load images from file path
    dict(
        type='MultiScaleFlipAug',  # An encapsulation that encapsulates the testing␣
↪augmentations
        img_scale=(1024, 1024),  # Decides the largest scale for testing, used for␣
↪the Resize pipeline
        flip=False,  # Whether to flip images during testing
        transforms=[
            dict(type='RResize'),  # Use resize augmentation
            dict(
                type='Normalize',  # Normalization config, the values are from img_
↪norm_cfg
                mean=[123.675, 116.28, 103.53],
```

(下页继续)

(续上页)

```
                std=[58.395, 57.12, 57.375],
                to_rgb=True),
            dict(type='Pad',  # Padding config to pad images divisible by 32.
                 size_divisor=32),
            dict(type='DefaultFormatBundle'),  # Default format bundle to gather data␣
↪in the pipeline
            dict(type='Collect',  # Collect pipeline that collect necessary keys for␣
↪testing.
                 keys=['img'])
        ])
]
data = dict(
    samples_per_gpu=2,  # Batch size of a single GPU
    workers_per_gpu=2,  # Worker to pre-fetch data for each single GPU
    train=dict(  # Train dataset config
        type='DOTADataset',  # Type of dataset
        ann_file=
        '../datasets/split_1024_dota1_0/trainval/annfiles/',  # Path of annotation␣
↪file
        img_prefix=
        '../datasets/split_1024_dota1_0/trainval/images/',  # Prefix of image path
        pipeline=[  # pipeline, this is passed by the train_pipeline created before.
            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),
            dict(type='DefaultFormatBundle'),
            dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels'])
        ],
        version='oc'),
    val=dict(  # Validation dataset config
        type='DOTADataset',
        ann_file=
        '../datasets/split_1024_dota1_0/trainval/annfiles/',
        img_prefix=
        '../datasets/split_1024_dota1_0/trainval/images/',
        pipeline=[
```

(下页继续)

(续上页)

```
            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(  # Test dataset config, modify the ann_file for test-dev/test␣
↪submission
        type='DOTADataset',
        ann_file=
        '../datasets/split_1024_dota1_0/test/images/',
        img_prefix=
        '../datasets/split_1024_dota1_0/test/images/',
        pipeline=[  # Pipeline is passed by test_pipeline created before
            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'])
                ])
        ],
```

(下页继续)

(续上页)

```
        version='oc'))
evaluation = dict(  # The config to build the evaluation hook
    interval=12,  # Evaluation interval
    metric='mAP')  # Metrics used during evaluation
optimizer = dict(  # Config used to build optimizer
    type='SGD',  # Type of optimizers
    lr=0.0025,  # Learning rate of optimizers
    momentum=0.9,  # Momentum
    weight_decay=0.0001)  # Weight decay of SGD
optimizer_config = dict(  # Config used to build the optimizer hook
    grad_clip=dict(
        max_norm=35,
        norm_type=2))
lr_config = dict(  # Learning rate scheduler config used to register LrUpdater hook
    policy='step',  # The policy of scheduler
    warmup='linear',  # The warmup policy, also support `exp` and `constant`.
    warmup_iters=500,  # The number of iterations for warmup
    warmup_ratio=0.3333333333333333,  # The ratio of the starting learning rate used␣
↪for warmup
    step=[8, 11])  # Steps to decay the learning rate
runner = dict(
    type='EpochBasedRunner',  # Type of runner to use (i.e. IterBasedRunner or␣
↪EpochBasedRunner)
    max_epochs=12) # Runner that runs the workflow in total max_epochs. For␣
↪IterBasedRunner use `max_iters`
checkpoint_config = dict(  # Config to set the checkpoint hook
    interval=12)  # The save interval is 12
log_config = dict(  # config to register logger hook
    interval=50,  # Interval to print the log
    hooks=[
        # dict(type='TensorboardLoggerHook')  # The Tensorboard logger is also␣
↪supported
        dict(type='TextLoggerHook')
    ])  # The logger used to record the training process.
dist_params = dict(backend='nccl')  # Parameters to setup distributed training, the␣
↪port can also be set.
log_level = 'INFO'  # The level of logging.
load_from = None  # load models as a pre-trained model from a given path. This will␣
↪not resume training.
resume_from = None  # Resume checkpoints from a given path, the training will be␣
↪resumed from the epoch when the checkpoint's is saved.
workflow = [('train', 1)]  # Workflow for runner. [('train', 1)] means there is only␣
↪one workflow and the workflow named 'train' is executed once. The workflow trains␣
↪the model by 12 epochs according to the total_epochs.
```

(下页继续)

(续上页)

```
work_dir = './work_dirs/rotated_retinanet_hbb_r50_fpn_1x_dota_oc'  # Directory to␣
↪save the model checkpoints and logs for the current experiments.
```

9.4 FAQ

9.4.1 Use intermediate variables in configs

Some intermediate variables are used in the configs files, like `train_pipeline`/`test_pipeline` in datasets. It's worth noting that when modifying intermediate variables in the children configs, user need to pass the intermediate variables into corresponding fields again. For example, we would like to use offline multi scale strategy to train a RoI-Trans. `train_pipeline` are intermediate variable we would like modify.

```
_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'])
]
data = dict(
    train=dict(
        pipeline=train_pipeline,
```

(下页继续)

(续上页)

```
        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/'))
```

We first define the new `train_pipeline`/`test_pipeline` and pass them into `data`.

Similarly, if we would like to switch from `SyncBN` to `BN` or `MMSyncBN`, we need to substitute every `norm_cfg` in the config.

```
_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`:

```python
# 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

Tutorial 3: Customize Models

We basically categorize model components into 5 types.

- backbone: usually an FCN network to extract feature maps, e.g., ResNet, Swin.
- neck: the component between backbones and heads, e.g., FPN, ReFPN.
- head: the component for specific tasks, e.g., bbox prediction.
- roi extractor: the part for extracting RoI features from feature maps, e.g., RoI Align Rotated.
- loss: the component in head for calculating losses, e.g., FocalLoss, GWDLoss, and KFIoULoss.

11.1 Develop new components

11.1.1 Add a new backbone

Here we show how to develop new components with an example of MobileNet.

1. Define a new backbone (e.g. MobileNet)

Create a new file 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. Import the module

You can either add the following line to mmrotate/models/backbones/__init__.py

```
from .mobilenet import MobileNet
```

or alternatively add

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

to the config file to avoid modifying the original code.

3. Use the backbone in your config file

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

11.1.2 Add new necks

1. Define a neck (e.g. PAFPN)

Create a new file 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. Import the module

You can either add the following line to mmrotate/models/necks/__init__.py,

```
from .pafpn import PAFPN
```

or alternatively add

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

to the config file and avoid modifying the original code.

3. Modify the config file

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

11.1.3 Add new heads

Here we show how to develop a new head with the example of Double Head R-CNN as the following.

First, add a new bbox head in `mmrotate/models/roi_heads/bbox_heads/double_bbox_head.py`. Double Head R-CNN implements a new bbox head for object detection. To implement a bbox head, basically we need to implement three functions of the new module as the following.

```
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):
```

Second, implement a new RoI Head if it is necessary. We plan to inherit the new `DoubleHeadRoIHead` from `StandardRoIHead`. We can find that a `StandardRoIHead` already implements the following functions.

```
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' s modification is mainly in the bbox_forward logic, and it inherits other logics from the `StandardRoIHead`. In the `mmrotate/models/roi_heads/double_roi_head.py`, we implement the new RoI Head as the following:

```python
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
```

Last, the users need to add the module in `mmrotate/models/bbox_heads/__init__.py` and `mmrotate/`

models/roi_heads/__init__.py thus the corresponding registry could find and load them.

Alternatively, the users can add

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

to the config file and achieve the same goal.

11.1.4 Add new loss

Assume you want to add a new loss as MyLoss, for bounding box regression. To add a new loss function, the users need implement it in mmrotate/models/losses/my_loss.py. The decorator weighted_loss enable the loss to be weighted for each element.

```
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)
```

(下页继续)

(续上页)

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

Then the users need to add it in the `mmrotate/models/losses/__init__.py`.

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

Alternatively, you can add

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

to the config file and achieve the same goal.

To use it, modify the `loss_xxx` field. Since MyLoss is for regression, you need to modify the `loss_bbox` field in the head.

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

CHAPTER 12

Tutorial 4: Customize Runtime Settings

12.1 Customize optimization settings

12.1.1 Customize optimizer supported by Pytorch

We already support to use all the optimizers implemented by PyTorch, and the only modification is to change the `optimizer` field of config files. For example, if you want to use `ADAM` (note that the performance could drop a lot), the modification could be as the following.

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

To modify the learning rate of the model, the users only need to modify the `lr` in the config of optimizer. The users can directly set arguments following the API doc of PyTorch.

12.1.2 Customize self-implemented optimizer

1. Define a new optimizer

A customized optimizer could be defined as following.

Assume you want to add a optimizer named `MyOptimizer`, which has arguments `a`, `b`, and `c`. You need to create a new directory named `mmrotate/core/optimizer`. And then implement the new optimizer in a file, e.g., in `mmrotate/core/optimizer/my_optimizer.py`:

```python
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. Add the optimizer to registry

To find the above module defined above, this module should be imported into the main namespace at first. There are two options to achieve it.

- Modify `mmrotate/core/optimizer/__init__.py` to import it.

  The newly defined module should be imported in `mmrotate/core/optimizer/__init__.py` so that the registry will find the new module and add it:

```python
from .my_optimizer import MyOptimizer
```

- Use `custom_imports` in the config to manually import it

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

The module `mmrotate.core.optimizer.my_optimizer` will be imported at the beginning of the program and the class `MyOptimizer` is then automatically registered. Note that only the package containing the class `MyOptimizer` should be imported. `mmrotate.core.optimizer.my_optimizer.MyOptimizer` **cannot** be imported directly.

Actually users can use a totally different file directory structure using this importing method, as long as the module root can be located in `PYTHONPATH`.

3. Specify the optimizer in the config file

Then you can use `MyOptimizer` in `optimizer` field of config files. In the configs, the optimizers are defined by the field `optimizer` like the following:

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

To use your own optimizer, the field can be changed to

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

12.1.3 Customize optimizer constructor

Some models may have some parameter-specific settings for optimization, e.g. weight decay for BatchNorm layers. The users can do those fine-grained parameter tuning through customizing optimizer constructor.

```python
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
```

The default optimizer constructor is implemented here, which could also serve as a template for new optimizer constructor.

12.1.4 Additional settings

Tricks not implemented by the optimizer should be implemented through optimizer constructor (e.g., set parameter-wise learning rates) or hooks. We list some common settings that could stabilize the training or accelerate the training. Feel free to create PR, issue for more settings.

- **Use gradient clip to stabilize training**: Some models need gradient clip to clip the gradients to stabilize the training process. An example is as below:

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

  If your config inherits the base config which already sets the `optimizer_config`, you might need `_delete_=True` to override the unnecessary settings. See the config documentation for more details.

- **Use momentum schedule to accelerate model convergence**: We support momentum scheduler to modify model's momentum according to learning rate, which could make the model converge in a faster way. Momentum scheduler is usually used with LR scheduler, for example, the following config is used in 3D detection to accelerate convergence. For more details, please refer to the implementation of CyclicLrUpdater and CyclicMomentumUpdater.

```
lr_config = dict(
    policy='cyclic',
    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 Customize training schedules

By default we use step learning rate with 1x schedule, this calls `StepLRHook` in MMCV. We support many other learning rate schedule here, such as `CosineAnnealing` and `Poly` schedule. Here are some examples

- Poly schedule:

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

- ConsineAnnealing schedule:

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

12.3 Customize workflow

Workflow is a list of (phase, epochs) to specify the running order and epochs. By default it is set to be

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

which means running 1 epoch for training. Sometimes user may want to check some metrics (e.g. loss, accuracy) about the model on the validate set. In such case, we can set the workflow as

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

so that 1 epoch for training and 1 epoch for validation will be run iteratively.

Note:

1. The parameters of model will not be updated during val epoch.
2. Keyword `total_epochs` in the config only controls the number of training epochs and will not affect the validation workflow.
3. Workflows `[('train', 1), ('val', 1)]` and `[('train', 1)]` will not change the behavior of `EvalHook` because `EvalHook` is called by `after_train_epoch` and validation workflow only affect hooks that are called through `after_val_epoch`. Therefore, the only difference between `[('train', 1), ('val', 1)]` and `[('train', 1)]` is that the runner will calculate losses on validation set after each training epoch.

12.4 Customize hooks

12.4.1 Customize self-implemented hooks

1. Implement a new hook

There are some occasions when the users might need to implement a new hook. MMRotate supports customized hooks in training. Thus the users could implement a hook directly in mmrotate or their mmdet-based codebases and use the hook by only modifying the config in training. Here we give an example of creating a new hook in mmrotate and using it in training.

```
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):
```

(下页继续)

(续上页)

```
        pass

    def after_epoch(self, runner):
        pass

    def before_iter(self, runner):
        pass

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

Depending on the functionality of the hook, the users need to specify what the hook will do at each stage of the training in `before_run`, `after_run`, `before_epoch`, `after_epoch`, `before_iter`, and `after_iter`.

2. Register the new hook

Then we need to make `MyHook` imported. Assuming the file is in `mmrotate/core/utils/my_hook.py` there are two ways to do that:

- Modify `mmrotate/core/utils/__init__.py` to import it.

  The newly defined module should be imported in `mmrotate/core/utils/__init__.py` so that the registry will find the new module and add it:

```
from .my_hook import MyHook
```

- Use `custom_imports` in the config to manually import it

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

3. Modify the config

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

You can also set the priority of the hook by adding key `priority` to `'NORMAL'` or `'HIGHEST'` as below

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

By default the hook’s priority is set as `NORMAL` during registration.

12.4.2 Use hooks implemented in MMCV

If the hook is already implemented in MMCV, you can directly modify the config to use the hook as below

4. Example: NumClassCheckHook

We implement a customized hook named NumClassCheckHook to check whether the `num_classes` in head matches the length of `CLASSSES` in `dataset`.

We set it in default_runtime.py.

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

12.4.3 Modify default runtime hooks

There are some common hooks that are not registered through `custom_hooks`, they are

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

In those hooks, only the logger hook has the `VERY_LOW` priority, others' priority are `NORMAL`. The above-mentioned tutorials already covers how to modify `optimizer_config`, `momentum_config`, and `lr_config`. Here we reveals how what we can do with `log_config`, `checkpoint_config`, and `evaluation`.

Checkpoint config

The MMCV runner will use `checkpoint_config` to initialize `CheckpointHook`.

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

The users could set `max_keep_ckpts` to only save only small number of checkpoints or decide whether to store state dict of optimizer by `save_optimizer`. More details of the arguments are here

Log config

The `log_config` wraps multiple logger hooks and enables to set intervals. Now MMCV supports `WandbLoggerHook`, `MlflowLoggerHook`, and `TensorboardLoggerHook`. The detail usages can be found in the doc.

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

Evaluation config

The config of `evaluation` will be used to initialize the `EvalHook`. Except the key `interval`, other arguments such as `metric` will be passed to the `dataset.evaluate()`

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

CHAPTER 13

Changelog

CHAPTER 14

Frequently Asked Questions

We list some common troubles faced by many users and their corresponding solutions here. Feel free to enrich the list if you find any frequent issues and have ways to help others to solve them. If the contents here do not cover your issue, please create an issue using the provided templates and make sure you fill in all required information in the template.

14.1 MMCV Installation

- Compatibility issue between MMCV and MMDetection; "ConvWS is already registered in conv layer"; "AssertionError: MMCV==xxx is used but incompatible. Please install mmcv>=xxx, <=xxx."

  Please install the correct version of MMCV for the version of your MMDetection following the installation instruction.

- "No module named 'mmcv.ops'"; "No module named 'mmcv._ext'".

  1. Uninstall existing mmcv in the environment using `pip uninstall mmcv`.
  2. Install mmcv-full following the installation instruction.

14.2 PyTorch/CUDA Environment

- "RTX 30 series card fails when building MMCV or MMDet"
    1. Temporary work-around: do `MMCV_WITH_OPS=1 MMCV_CUDA_ARGS='-gencode=arch=compute_80,code=sm_80' pip install -e ..` The common issue is `nvcc fatal : Unsupported gpu architecture 'compute_86'`. This means that the compiler should optimize for sm_86, i.e., nvidia 30 series card, but such optimizations have not been supported by CUDA toolkit 11.0. This work-around modifies the compile flag by adding `MMCV_CUDA_ARGS='-gencode=arch=compute_80,code=sm_80'`, which tells `nvcc` to optimize for **sm_80**, i.e., Nvidia A100. Although A100 is different from the 30 series card, they use similar ampere architecture. This may hurt the performance but it works.
    2. PyTorch developers have updated that the default compiler flags should be fixed by pytorch/pytorch#47585. So using PyTorch-nightly may also be able to solve the problem, though we have not tested it yet.
- "invalid device function" or "no kernel image is available for execution".
    1. Check if your cuda runtime version (under `/usr/local/`), `nvcc --version` and `conda list cudatoolkit` version match.
    2. Run `python mmdet/utils/collect_env.py` to check whether PyTorch, torchvision, and MMCV are built for the correct GPU architecture. You may need to set `TORCH_CUDA_ARCH_LIST` to reinstall MMCV. The GPU arch table could be found here, i.e. run `TORCH_CUDA_ARCH_LIST=7.0 pip install mmcv-full` to build MMCV for Volta GPUs. The compatibility issue could happen when using old GPUS, e.g., Tesla K80 (3.7) on colab.
    3. Check whether the running environment is the same as that when mmcv/mmdet has compiled. For example, you may compile mmcv using CUDA 10.0 but run it on CUDA 9.0 environments.
- "undefined symbol" or "cannot open xxx.so".
    1. If those symbols are CUDA/C++ symbols (e.g., libcudart.so or GLIBCXX), check whether the CUDA/GCC runtimes are the same as those used for compiling mmcv, i.e. run `python mmdet/utils/collect_env.py` to see if `"MMCV Compiler"`/`"MMCV CUDA Compiler"` is the same as `"GCC"`/`"CUDA_HOME"`.
    2. If those symbols are PyTorch symbols (e.g., symbols containing caffe, aten, and TH), check whether the PyTorch version is the same as that used for compiling mmcv.
    3. Run `python mmdet/utils/collect_env.py` to check whether PyTorch, torchvision, and MMCV are built by and running on the same environment.
- setuptools.sandbox.UnpickleableException: DistutilsSetupError("each element of 'ext_modules' option must be an Extension instance or 2-tuple")
    1. If you are using miniconda rather than anaconda, check whether Cython is installed as indicated in #3379. You need to manually install Cython first and then run command `pip install -r requirements.txt`.

2. You may also need to check the compatibility between the `setuptools`, `Cython`, and `PyTorch` in your environment.

- "Segmentation fault".

1. Check you GCC version and use GCC 5.4. This usually caused by the incompatibility between PyTorch and the environment (e.g., GCC < 4.9 for PyTorch). We also recommend the users to avoid using GCC 5.5 because many feedbacks report that GCC 5.5 will cause "segmentation fault" and simply changing it to GCC 5.4 could solve the problem.

2. Check whether PyTorch is correctly installed and could use CUDA op, e.g. type the following command in your terminal.

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

And see whether they could correctly output results.

3. If Pytorch is correctly installed, check whether MMCV is correctly installed.

```
python -c 'import mmcv; import mmcv.ops'
```

If MMCV is correctly installed, then there will be no issue of the above two commands.

4. If MMCV and Pytorch is correctly installed, you man use `ipdb`, `pdb` to set breakpoints or directly add 'print' in mmdetection code and see which part leads the segmentation fault.

14.3 Training

- "Loss goes Nan"

1. Check if the dataset annotations are valid: zero-size bounding boxes will cause the regression loss to be Nan due to the commonly used transformation for box regression. Some small size (width or height are smaller than 1) boxes will also cause this problem after data augmentation (e.g., instaboost). So check the data and try to filter out those zero-size boxes and skip some risky augmentations on the small-size boxes when you face the problem.

2. Reduce the learning rate: the learning rate might be too large due to some reasons, e.g., change of batch size. You can rescale them to the value that could stably train the model.

3. Extend the warmup iterations: some models are sensitive to the learning rate at the start of the training. You can extend the warmup iterations, e.g., change the `warmup_iters` from 500 to 1000 or 2000.

4. Add gradient clipping: some models requires gradient clipping to stabilize the training process. The default of `grad_clip` is `None`, you can add gradient clippint to avoid gradients that are too large, i.e., set `optimizer_config=dict(_delete_=True, grad_clip=dict(max_norm=35, norm_type=2))` in your config file. If your config does not inherits from any basic con-

fig that contains `optimizer_config=dict(grad_clip=None)`, you can simply add `optimizer_config=dict(grad_clip=dict(max_norm=35, norm_type=2))`.

- ' GPU out of memory"
  1. There are some scenarios when there are large amounts of ground truth boxes, which may cause OOM during target assignment. You can set `gpu_assign_thr=N` in the config of assigner thus the assigner will calculate box overlaps through CPU when there are more than N GT boxes.
  2. Set `with_cp=True` in the backbone. This uses the sublinear strategy in PyTorch to reduce GPU memory cost in the backbone.
  3. Try mixed precision training using following the examples in `config/fp16`. The `loss_scale` might need further tuning for different models.
- "RuntimeError: Expected to have finished reduction in the prior iteration before starting a new one"
  1. This error indicates that your module has parameters that were not used in producing loss. This phenomenon may be caused by running different branches in your code in DDP mode.
  2. You can set `find_unused_parameters = True` in the config to solve the above problems or find those unused parameters manually.

14.4 Evaluation

- COCO Dataset, AP or AR = -1
  1. According to the definition of COCO dataset, the small and medium areas in an image are less than 1024 (32*32), 9216 (96*96), respectively.
  2. If the corresponding area has no object, the result of AP and AR will set to -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*)

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.

format_results (*results, submission_dir=None, **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.

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*)

Merging patch bboxes into full image.

Params: results (list): Testing results of the dataset.

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

F

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

L

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

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*), 66

�

模块
    mmrotate.core, 65
    mmrotate.datasets, 65