mmrotate

MMRotate Author

Apr 01, 2022

LEARN THE BASICS

1 Learn the Basics 1

2 Prerequisites 5

3 Installation 7

4 Verification 11

5 Dataset Preparation 13

6 Test a model 15

7 Train a model 17

8 Benchmark and Model Zoo 19

9 Tutorial 1: Learn about Configs 21

10 Tutorial 2: Customize Datasets 29

11 Tutorial 3: Customize Models 35

12 Tutorial 4: Customize Runtime Settings 43

13 Changelog 51

14 Frequently Asked Questions 55

15 English 59

16 61

17 mmrotate.apis 63

18 mmrotate.core 65

19 mmrotate.datasets 83

20 mmrotate.models 87

21 mmrotate.utils 137

22 Indices and tables 139

Python Module Index **141**

Index **143**

CHAPTER

ONE

LEARN THE BASICS

This chapter introduces the basic conception of rotated object detection and the framework of MMRotate, and provides links to detailed tutorials about MMRotate.

1.1 What is rotated object detection

1.1.1 Problem definition

Benefiting from the vigorous development of general object detection, most current rotated object detection models are based on classic general object detector. With the development of detection tasks, horizontal boxes have been unable to meet the needs of researchers in some subdivisions. We call it rotating object detection by redefining the object representation and increasing the number of regression degrees of freedom to achieve rotated rectangle, quadrilateral, and even arbitrary shape detection. Performing high-precision rotated object detection more efficiently has become a current research hotspot. The following areas are where rotated object detection has been applied or has great potential: face recognition, scene text, remote sensing, self-driving, medical, robotic grasping, etc.

1.1.2 What is rotated box

The most notable difference between rotated object detection and generic detection is the replacement of horizontal box annotations with rotated box annotations. They are defined as follows:

- Horizontal box: A rectangle with the width along the x-axis and height along the y-axis. Usually, it can be represented by the coordinates of 2 diagonal vertices `(x_i, y_i)` (i = 1, 2), or it can be represented by the coordinates of the center point and the height and width `(x_center, y_center, height, width)`.
- Rotated box: It is obtained by rotating the horizontal box around the center point by an `angle`, and the definition method of its rotated box is obtained by adding a radian parameter `(x_center, y_center, height, width, theta)`, where `theta = angle * pi / 180`. The unit of `theta` is `rad`. When the rotation angle is a multiple of 90°, the rotated box degenerates into a horizontal box. The rotated box annotations exported by the annotation software are usually polygons, which need to be converted to the rotated box definition method before training.

Note: In MMRotate, angle parameters are in radians.

1.1.3 Rotation direction

A rotated box can be obtained by rotating a horizontal box clockwise or counterclockwise around its center point. The rotation direction is closely related to the choice of the coordinate system. The image space adopts the right-handed coordinate system `(y, x)`, where y is `up->down` and x is `left->right`. There are two opposite directions of rotation:

- ClockwiseCW

Schematic of `CW`

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

Rotation matrix of `CW`

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

Rotation transformation of `CW`

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

- CounterclockwiseCCW

Schematic of `CCW`

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

Rotation matrix of `CCW`

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

Rotation transformation of `CCW`

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

The operators that can set the rotation direction in MMCV are:

- box_iou_rotated (Defaults to `CW`)
- nms_rotated (Defaults to `CW`)
- RoIAlignRotated (Defaults to `CCW`)
- RiRoIAlignRotated (Defaults to `CCW`).

Note: In MMRotate, the rotation direction of the rotated boxes is `CW`.

1.1.4 Definition of rotated box

Due to the difference in the definition range of `theta`, the following three definitions of the rotated box gradually emerge in rotated object detection:

- $D_{oc'}$: OpenCV Definition, `angle(0, 90°]`, `theta(0, pi / 2]`, The angle between the height of the rectangle and the positive semi-axis of x is a positive acute angle. This definition comes from the `cv2.minAreaRect` function in OpenCV, which returns an angle in the range `(0, 90°]`.
- D_{le135}: Long Edge Definition (135°)`angle[-45°, 135°)`, `theta[-pi / 4, 3 * pi / 4)` and `height > width`.
- D_{le90}: Long Edge Definition (90°)`angle[-90°, 90°)`, `theta[-pi / 2, pi / 2)` and `height > width`.

The conversion relationship between the three definitions is not involved in MMRotate, so we will not introduce it much more. Refer to the below blog to dive deeper.

Note: MMRotate supports the above three definitions of rotated box simultaneously, which can be flexibly switched through the configuration file.

It should be noted that if the OpenCV version is less than 4.5.1, the angle range of `cv2.minAreaRect` is between `[-90°, 0°)`. Reference In order to facilitate the distinction, the old version of the OpenCV definition is denoted as D_{oc}.

- $D_{oc'}$: OpenCV definition, `opencv>=4.5.1`, `angle(0, 90°]`, `theta(0, pi / 2]`.
- D_{oc} : Old OpenCV definition, `opencv<4.5.1`, `angle[-90°, 0°)`, `theta[-pi / 2, 0)`.

The conversion relationship between the two OpenCV definitions is as follows:

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

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

Note: Regardless of the OpenCV version you are using, MMRotate will convert the theta of the OpenCV definition to (0, pi / 2].

1.1.5 Evaluation

The code for evaluating mAP involves the calculation of IoU. We can directly calculate the IoU of the rotated boxes or convert the rotated boxes to a polygons and then calculate the polygons IoU (DOTA online evaluation uses the calculation of polygons IoU).

1.2 What is MMRotate

MMRotate is a toolbox that provides a framework for unified implementation and evaluation of rotated object detection, and below is its whole framework:

MMRotate consists of 4 main parts, `datasets`, `models`, `core` and `apis`.

- `datasets` is for data loading and data augmentation. In this part, we support various datasets for rotated object detection algorithms, useful data augmentation transforms in `pipelines` for pre-processing image.
- `models` contains models and loss functions.
- `core` provides evaluation tools for model training and evaluation.
- `apis` provides high-level APIs for models training, testing, and inference.

The module design of MMRotate is as follows:

The following points need to be noted due to different definitions of rotated box:

- Loading annotations
- Data augmentation
- Assigning samples
- Evaluation

1.3 How to Use this Guide

Here is a detailed step-by-step guide to learn more about MMRotate:

1. For installation instructions, please see *install*.
2. *get_started* is for the basic usage of MMRotate.
3. Refer to the below tutorials to dive deeper:

- *Config*
- *Customize Dataset*
- *Customize Model*
- *Customize Runtime*

CHAPTER

TWO

PREREQUISITES

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

Compatible MMCV, MMClassification and MMDetection versions are shown as below. Please install the correct version of them to avoid installation issues.

Note: You need to run `pip uninstall mmcv` first if you have mmcv installed. If mmcv and mmcv-full are both installed, there will be `ModuleNotFoundError`.

CHAPTER

THREE

INSTALLATION

3.1 A from-scratch setup script

Assuming that you already have CUDA 10.1 installed, here is a full script for setting up MMRotate with conda. You can refer to the step-by-step installation instructions in the next section.

```
conda create -n open-mmlab python=3.7 pytorch==1.7.0 cudatoolkit=10.1 torchvision -c␣
↪pytorch -y
conda activate open-mmlab
pip install openmim
mim install mmcv-full
mim install mmdet
git clone https://github.com/open-mmlab/mmrotate.git
cd mmrotate
pip install -r requirements/build.txt
pip install -v -e .
```

3.2 Prepare environment

1. Create a conda virtual environment and activate it.

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

2. Install PyTorch and torchvision following the official instructions, e.g.,

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

   Note: Make sure that your compilation CUDA version and runtime CUDA version match. You can check the supported CUDA version for precompiled packages on the PyTorch website.

   `E.g` If you have CUDA 10.1 installed under `/usr/local/cuda` and would like to install PyTorch 1.7, you need to install the prebuilt PyTorch with CUDA 10.1.

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

3.3 Install MMRotate

It is recommended to install MMRotate with MIM, which automatically handle the dependencies of OpenMMLab projects, including mmcv and other python packages.

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

Or you can still install MMRotate manually:

1. Install mmcv-full.

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

   Please replace `{cu_version}` and `{torch_version}` in the url to your desired one. For example, to install the latest `mmcv-full` with `CUDA 11.0` and `PyTorch 1.7.0`, use the following command:

   ```
   pip install mmcv-full -f https://download.openmmlab.com/mmcv/dist/cu110/torch1.7.0/
   ↪index.html
   ```

   See here for different versions of MMCV compatible to different PyTorch and CUDA versions.

   Optionally you can compile mmcv from source if you need to develop both mmcv and mmrotate. Refer to the guide for details.

2. Install MMDetection.

   You can simply install mmdetection with the following command:

   ```
   pip install mmdet
   ```

3. Install MMRotate.

   You can simply install MMRotate with the following command:

   ```
   pip install mmrotate
   ```

   or clone the repository and then install it:

   ```
   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:

a. When specifying `-e` or `develop`, MMRotate is installed on dev mode , any local modifications made to the code will take effect without reinstallation.

b. If you would like to use `opencv-python-headless` instead of `opencv-python`, you can install it before installing MMCV.

c. Some dependencies are optional. Simply running `pip install -v -e .` will only install the minimum runtime requirements. To use optional dependencies like `albumentations` and `imagecorruptions` either install them manually with `pip install -r requirements/optional.txt` or specify desired extras when calling `pip` (e.g. `pip install -v -e .[optional]`). Valid keys for the extras field are: `all`, `tests`, `build`, and `optional`.

3.4 Another option: Docker Image

We provide a Dockerfile to build an image. Ensure that you are using docker version >=19.03.

```
# build an image with PyTorch 1.6, CUDA 10.1
docker build -t mmrotate docker/
```

Run it with

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

CHAPTER

FOUR

VERIFICATION

To verify whether MMRotate is installed correctly, we can run the demo code and inference a demo image.

Please refer to demo for more details. The demo code is supposed to run successfully upon you finish the installation.

CHAPTER

FIVE

DATASET PREPARATION

Please refer to data preparation for dataset preparation.

CHAPTER

SIX

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 RotatedRetinaNet on DOTA-1.0 dataset, which can generate compressed files for online submission. (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 --format-only \
  --eval-options submission_dir=work_dirs/Task1_results
```

or

```
./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
```

You can change the test set path in the data_root to the val set or trainval set for the offline evaluation.

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

or

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

CHAPTER

SEVEN

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 launch with multiple machines simply connected with ethernet, you can simply run following commands:

On the first machine:

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

On the second machine:

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

Usually it is slow if you do not have high speed networking like InfiniBand.

7.4 Manage jobs with Slurm

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.

7.5 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` and `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

EIGHT

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)
- 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)

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/2080Ti and inferred with 1 * 2080Ti.

CHAPTER

NINE

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 to learn the basics 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 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 types 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 the 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 module, 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
        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
```

(continues on next page)

(continued from previous page)

```
        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.
            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
```

(continues on next page)

(continued from previous page)

```
    std=[58.395, 57.12, 57.375],  # Standard variance used to pre-training the pre-
↪trained backbone models
    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],
                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'])
```

(continues on next page)

(continued from previous page)

```
        ])
]
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=[
            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
```

(continues on next page)

(continued from previous page)

```
        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'])
                ])
        ],
        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
```

(continues on next page)

(continued from previous page)

```
    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.
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, the user needs to pass the intermediate variables into corresponding fields again. For example, we would like to use an offline multi-scale strategy to train an RoI-Trans. `train_pipeline` are intermediate variables we would like to 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'),
```

(continues on next page)

(continued from previous page)

```
    dict(type='Collect', keys=['img', 'gt_bboxes', 'gt_labels'])
]
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/'))
```

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

TEN

TUTORIAL 2: CUSTOMIZE DATASETS

10.1 Support new data format

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

10.1.1 Reorganize new data formats to existing format

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

The annotation txt files in DOTA format:

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

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

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

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

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

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

1. Modify the config file for using the customized dataset

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

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

In `configs/my_custom_config.py`:

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

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

ELEVEN

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`.

```python
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,
```

(continues on next page)

(continued from previous page)

```
                 norm_cfg=dict(type='BN'),
                 **kwargs):
        kwargs.setdefault('with_avg_pool', True)
        super(DoubleConvFCBBoxHead, self).__init__(**kwargs)


    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,
                    img_metas,
                    proposals=None,
                    rescale=False):
```

(continues on next page)

(continued from previous page)

```
        """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:

```
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.

```python
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)
        return loss_bbox
```

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

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

Alternatively, you can add

```python
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

TWELVE

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`:

```
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:

```
from .my_optimizer import MyOptimizer
```

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

```
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:

```
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

```
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.

```
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):

```

(continues on next page)

(continued from previous page)

```
        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:

```
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
```

(continues on next page)

(continued from previous page)

```
    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

    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

THIRTEEN

CHANGELOG

13.1 v0.2.0 (30/3/2022)

13.1.1 New Features

- Support Circular Smooth Label (CSL, ECCV'20) (#153)
- Support multiple machines dist_train (#143)
- Add browse_dataset tool (#98)
- Add gather_models script (#162)

13.1.2 Bug Fixes

- Remove in-place operations in rbbox_overlaps (#155)
- Fix bug in docstring. (#137)
- Fix bug in HRSCDataset with `clasesswise=ture` (#175)

13.1.3 Improvements

- Add Chinese translation of `docs/zh_cn/tutorials/customize_dataset.md` (#65)
- Add different seeds to different ranks (#102)
- Update from-scratch install script in install.md (#166)
- Improve the arguments of all mmrotate scripts (#168)

13.1.4 Contributors

A total of 6 developers contributed to this release. Thanks @zytx121 @yangxue0827 @ZwwWayne @jbwang1997 @canoe-Z @matrixgame2018

13.2 v0.1.1 (14/3/2022)

13.2.1 New Features

- Support huge image inference (#34)
- Support HRSC Dataset (#96)
- Support mixed precision training (#72)
- Add colab tutorial for beginners (#66)
- Add inference speed statistics tool (#86)
- Add confusion matrix analysis tool (#93)

13.2.2 Bug Fixes

- Fix URL error of Swin pretrained model (#111)
- Fix bug for SASM during training (#105)
- Fix rbbox_overlaps abnormal when the box is too small (#61)
- Fix bug for visualization (#12, #81)
- Fix stuck when compute mAP (#14, #52)
- Fix 'RoIAlignRotated' object has no attribute 'out_size' bug (#51)
- Add missing init_cfg in dense head (#37)
- Fix install an additional mmcv (#17)
- Fix typos in docs (#3, #11, #36)

13.2.3 Improvements

- Move `eval_rbbox_map` from `mmrotate.datasets` to `mmrotate.core.evaluation` (#73)
- Add Windows CI (#31)
- Add copyright commit hook (#30)
- Add Chinese translation of `docs/zh_cn/get_started.md` (#16)
- Add Chinese translation of `docs/zh_cn/tutorials/customize_runtime.md` (#22)
- Add Chinese translation of `docs/zh_cn/tutorials/customize_config.md` (#23)
- Add Chinese translation of `docs/zh_cn/tutorials/customize_models.md` (#27)
- Add Chinese translation of `docs/zh_cn/model_zoo.md` (#28)
- Add Chinese translation of `docs/zh_cn/faq.md` (#33)

13.2.4 Contributors

A total of 13 developers contributed to this release. Thanks @zytx121 @yangxue0827 @jbwang1997 @liuyanyi @DangChuong-DC @RangeKing @liufeinuaa @np-csu @akmalulkhairin @SheffieldCao @BrotherHappy @Abyssaledge @q3394101

CHAPTER

FOURTEEN

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 MMRotate 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

- "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 E2CNN

- "ImportError: cannot import name 'container_bacs' from 'torch._six'"

1. This is because `container_abcs` has been removed since PyTorch 1.9.
2. Replace

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

in `python3.7/site-packages/e2cnn/nn/modules/module_list.py` with

```
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. Or downgrade the version of Pytorch.

14.4 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 config 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 by setting `fp16 = dict(loss_scale='dynamic')` in the config file.
- "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.

14.5 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

FIFTEEN

ENGLISH

CHAPTER

SIXTEEN

CHAPTER

SEVENTEEN

MMROTATE.APIS

mmrotate.apis.**inference_detector_by_patches**(*model*, *img*, *sizes*, *steps*, *ratios*, *merge_iou_thr*, *bs=1*)

inference patches with the detector.

Split huge image(s) into patches and inference them with the detector. Finally, merge patch results on one huge image by nms.

Parameters

- **model** (*nn.Module*) – The loaded detector.
- **img** (*str | ndarray or*) – Either an image file or loaded image.
- **sizes** (*list*) – The sizes of patches.
- **steps** (*list*) – The steps between two patches.
- **ratios** (*list*) – Image resizing ratios for multi-scale detecting.
- **merge_iou_thr** (*float*) – IoU threshold for merging results.
- **bs** (*int*) – Batch size, must greater than or equal to 1.

Returns Detection results.

Return type list[np.ndarray]

mmrotate.apis.**train_detector**(*model*, *dataset*, *cfg*, *distributed=False*, *validate=False*, *timestamp=None*, *meta=None*)

Main function of train.

CHAPTER

EIGHTEEN

MMROTATE.CORE

18.1 anchor

class mmrotate.core.anchor.**PseudoAnchorGenerator**(*strides*)

Non-Standard pseudo anchor generator that is used to generate valid flags only!

property num_base_anchors

total number of base anchors in a feature grid

Type list[int]

single_level_grid_anchors(*featmap_sizes*, *device='cuda'*)

Calling its grid_anchors() method will raise NotImplementedError!

class mmrotate.core.anchor.**RotatedAnchorGenerator**(*strides*, *ratios*, *scales=None*, *base_sizes=None*, *scale_major=True*, *octave_base_scale=None*, *scales_per_octave=None*, *centers=None*, *center_offset=0.0*)

Fake rotate anchor generator for 2D anchor-based detectors.

Horizontal bounding box represented by (x,y,w,h,theta).

single_level_grid_priors(*featmap_size*, *level_idx*, *dtype=torch.float32*, *device='cuda'*)

Generate grid anchors of a single level.

Note: This function is usually called by method `self.grid_priors`.

Parameters

- **featmap_size** (*tuple[int]*) – Size of the feature maps.
- **level_idx** (*int*) – The index of corresponding feature map level.
- **(obj** (*dtype*) – *torch.dtype*): Date type of points.Defaults to
- **torch.float32.** –
- **device** (*str, optional*) – The device the tensor will be put on.
- **to 'cuda'.** (*Defaults*) –

Returns Anchors in the overall feature maps.

Return type torch.Tensor

mmrotate.core.anchor.**rotated_anchor_inside_flags**(*flat_anchors*, *valid_flags*, *img_shape*, *allowed_border=0*)

Check whether the rotated anchors are inside the border.

Parameters

- **flat_anchors** (*torch.Tensor*) – Flatten anchors, shape (n, 5).
- **valid_flags** (*torch.Tensor*) – An existing valid flags of anchors.
- **img_shape** (*tuple(int)*) – Shape of current image.
- **allowed_border** (*int, optional*) – The border to allow the valid anchor. Defaults to 0.

Returns Flags indicating whether the anchors are inside a valid range.

Return type torch.Tensor

18.2 bbox

class mmrotate.core.bbox.**ATSSKldAssigner**(*topk*, *use_reassign=False*)

Assign a corresponding gt bbox or background to each bbox.

Each proposals will be assigned with *0* or a positive integer indicating the ground truth index.

- 0: negative sample, no assigned gt
- positive integer: positive sample, index (1-based) of assigned gt

Parameters

- **topk** (*float*) – Number of bbox selected in each level.
- **use_reassign** (*bool, optional*) – If true, it is used to reassign samples.

AspectRatio(*gt_rbboxes*)

compute the aspect ratio of all gts.

Parameters **gt_rbboxes** (*torch.Tensor*) – Groundtruth polygons, shape (k, 8).

Returns The aspect ratio of gt_rbboxes, shape (k, 1).

Return type ratios (torch.Tensor)

assign(*bboxes*, *num_level_bboxes*, *gt_bboxes*, *gt_bboxes_ignore=None*, *gt_labels=None*)

Assign gt to bboxes.

The assignment is done in following steps

1. compute iou between all bbox (bbox of all pyramid levels) and gt
2. compute center distance between all bbox and gt
3. on each pyramid level, for each gt, select k bbox whose center are closest to the gt center, so we total select k*l bbox as candidates for each gt
4. get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold
5. compute the mean aspect ratio of all gts, and set exp((-mean aspect ratio / 4) * (mean + std) as the iou threshold
6. select these candidates whose iou are greater than or equal to the threshold as positive

7. limit the positive sample's center in gt

Parameters

- **bboxes** (*Tensor*) – Bounding boxes to be assigned, shape(n, 4).
- **num_level_bboxes** (*List*) – num of bboxes in each level
- **gt_bboxes** (*Tensor*) – Groundtruth boxes, shape (k, 4).
- **gt_bboxes_ignore** (*Tensor, optional*) – Ground truth bboxes that are labelled as *ignored*, e.g., crowd boxes in COCO.
- **gt_labels** (*Tensor, optional*) – Label of gt_bboxes, shape (k,).

Returns The assign result.

Return type AssignResult

get_horizontal_bboxes(*gt_rbboxes*)

get_horizontal_bboxes from polygons.

Parameters **gt_rbboxes** (*torch.Tensor*) – Groundtruth polygons, shape (k, 8).

Returns The horizontal bboxes, shape (k, 4).

Return type gt_rect_bboxes (torch.Tensor)

kld_mixture2single(*g1*, *g2*)

Compute Kullback-Leibler Divergence between two Gaussian distribution.

Parameters

- **g1** (*dict[str, torch.Tensor]*) – Gaussian distribution 1.
- **g2** (*torch.Tensor*) – Gaussian distribution 2.

Returns Kullback-Leibler Divergence.

Return type torch.Tensor

kld_overlaps(*gt_rbboxes*, *points*, *eps=1e-06*)

Compute overlaps between polygons and points by Kullback-Leibler Divergence loss.

Parameters

- **gt_rbboxes** (*torch.Tensor*) – Ground truth polygons, shape (k, 8).
- **points** (*torch.Tensor*) – Points to be assigned, shape(n, 18).
- **eps** (*float, optional*) – Defaults to 1e-6.

Returns Kullback-Leibler Divergence loss.

Return type Tensor

class mmrotate.core.bbox.**ConvexAssigner**(*scale=4*, *pos_num=3*)

Assign a corresponding gt bbox or background to each bbox. Each proposals will be assigned with *0* or a positive integer indicating the ground truth index.

- 0: negative sample, no assigned gt
- positive integer: positive sample, index (1-based) of assigned gt

Parameters

- **scale** (*float*) – IoU threshold for positive bboxes.

- **pos_num** (*float*) – find the nearest pos_num points to gt center in this
- **level.** –

assign(*points*, *gt_rbboxes*, *gt_rbboxes_ignore=None*, *gt_labels=None*, *overlaps=None*)
Assign gt to bboxes.

The assignment is done in following steps

1. compute iou between all bbox (bbox of all pyramid levels) and gt
2. compute center distance between all bbox and gt
3. on each pyramid level, for each gt, select k bbox whose center are closest to the gt center, so we total select k*l bbox as candidates for each gt
4. get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold
5. select these candidates whose iou are greater than or equal to the threshold as positive
6. limit the positive sample's center in gt

Parameters

- **points** (*torch.Tensor*) – Points to be assigned, shape(n, 18).
- **gt_rbboxes** (*torch.Tensor*) – Groundtruth polygons, shape (k, 8).
- **gt_rbboxes_ignore** (*Tensor, optional*) – Ground truth polygons that are labelled as *ignored*, e.g., crowd boxes in COCO.
- **gt_labels** (*Tensor, optional*) – Label of gt_bboxes, shape (k,).

Returns The assign result.

Return type AssignResult

get_horizontal_bboxes(*gt_rbboxes*)
get_horizontal_bboxes from polygons.

Parameters **gt_rbboxes** (*torch.Tensor*) – Groundtruth polygons, shape (k, 8).

Returns The horizontal bboxes, shape (k, 4).

Return type gt_rect_bboxes (torch.Tensor)

class mmrotate.core.bbox.**DeltaXYWHAHBBoxCoder**(*target_means=(0.0, 0.0, 0.0, 0.0, 0.0)*, *target_stds=(1.0, 1.0, 1.0, 1.0, 1.0)*, *angle_range='oc'*, *norm_factor=None*, *edge_swap=False*, *clip_border=True*, *add_ctr_clamp=False*, *ctr_clamp=32*)
Delta XYWHA HBBox coder.

this coder encodes bbox (x1, y1, x2, y2) into delta (dx, dy, dw, dh, da) and decodes delta (dx, dy, dw, dh, da) back to original bbox (cx, cy, w, h, a).

Parameters

- **target_means** (*Sequence[float]*) – Denormalizing means of target for delta coordinates
- **target_stds** (*Sequence[float]*) – Denormalizing standard deviation of target for delta coordinates
- **angle_range** (*str, optional*) – Angle representations. Defaults to 'oc'.
- **norm_factor** (*None|float, optional*) – Regularization factor of angle.

- **edge_swap** (*bool, optional*) – Whether swap the edge if w < h. Defaults to False.
- **clip_border** (*bool, optional*) – Whether clip the objects outside the border of the image. Defaults to True.
- **add_ctr_clamp** (*bool*) – Whether to add center clamp, when added, the predicted box is clamped is its center is too far away from the original anchor's center. Only used by YOLOF. Default False.
- **ctr_clamp** (*int*) – the maximum pixel shift to clamp. Only used by YOLOF. Default 32.

decode(*bboxes*, *pred_bboxes*, *max_shape=None*, *wh_ratio_clip=0.016*)

Apply transformation *pred_bboxes* to *boxes*.

Parameters

- **bboxes** (*torch.Tensor*) – Basic boxes. Shape (B, N, 4) or (N, 4)
- **pred_bboxes** (*torch.Tensor*) –

  Encoded offsets with respect to each roi. Has shape (B, N, num_classes * 5) or (B, N, 5) or

  (N, num_classes * 5) or (N, 5). Note N = num_anchors * W * H when rois is a grid of anchors.
- **(Sequence[int] or torch.Tensor or Sequence[** (*max_shape*) – Sequence[int]],optional): Maximum bounds for boxes, specifies (H, W, C) or (H, W). If bboxes shape is (B, N, 5), then the max_shape should be a Sequence[Sequence[int]] and the length of max_shape should also be B.
- **wh_ratio_clip** (*float, optional*) – The allowed ratio between width and height.

Returns Decoded boxes.

Return type torch.Tensor

encode(*bboxes*, *gt_bboxes*)

Get box regression transformation deltas that can be used to transform the `bboxes` into the `gt_bboxes`.

Parameters

- **bboxes** (*torch.Tensor*) – Source boxes, e.g., object proposals.
- **gt_bboxes** (*torch.Tensor*) – Target of the transformation, e.g., ground-truth boxes.

Returns Box transformation deltas

Return type torch.Tensor

class mmrotate.core.bbox.**DeltaXYWHAOBBoxCoder**(*target_means=(0.0, 0.0, 0.0, 0.0, 0.0)*, *target_stds=(1.0, 1.0, 1.0, 1.0, 1.0)*, *angle_range='oc'*, *norm_factor=None*, *edge_swap=False*, *proj_xy=False*, *add_ctr_clamp=False*, *ctr_clamp=32*)

Delta XYWHA OBBox coder. This coder is used for rotated objects detection (for example on task1 of DOTA dataset). this coder encodes bbox (xc, yc, w, h, a) into delta (dx, dy, dw, dh, da) and decodes delta (dx, dy, dw, dh, da) back to original bbox (xc, yc, w, h, a).

Parameters

- **target_means** (*Sequence[float]*) – Denormalizing means of target for delta coordinates
- **target_stds** (*Sequence[float]*) – Denormalizing standard deviation of target for delta coordinates
- **angle_range** (*str, optional*) – Angle representations. Defaults to 'oc'.

- **norm_factor** (*None|float, optional*) – Regularization factor of angle.
- **edge_swap** (*bool, optional*) – Whether swap the edge if w < h. Defaults to False.
- **proj_xy** (*bool, optional*) – Whether project x and y according to angle. Defaults to False.
- **add_ctr_clamp** (*bool*) – Whether to add center clamp, when added, the predicted box is clamped is its center is too far away from the original anchor's center. Only used by YOLOF. Default False.
- **ctr_clamp** (*int*) – the maximum pixel shift to clamp. Only used by YOLOF. Default 32.

decode(*bboxes*, *pred_bboxes*, *max_shape=None*, *wh_ratio_clip=0.016*)

Apply transformation *pred_bboxes* to *boxes*.

Parameters

- **bboxes** (*torch.Tensor*) – Basic boxes. Shape (B, N, 5) or (N, 5)
- **pred_bboxes** (*torch.Tensor*) – Encoded offsets with respect to each roi. Has shape (B, N, num_classes * 5) or (B, N, 5) or (N, num_classes * 5) or (N, 5). Note N = num_anchors * W * H when rois is a grid of anchors.
- **max_shape** (*Sequence[int] or torch.Tensor or Sequence[Sequence[int]],optional*) – Maximum bounds for boxes, specifies (H, W, C) or (H, W). If bboxes shape is (B, N, 5), then the max_shape should be a Sequence[Sequence[int]] and the length of max_shape should also be B.
- **wh_ratio_clip** (*float, optional*) – The allowed ratio between width and height.

Returns Decoded boxes.

Return type torch.Tensor

encode(*bboxes*, *gt_bboxes*)

Get box regression transformation deltas that can be used to transform the `bboxes` into the `gt_bboxes`.

Parameters

- **bboxes** (*torch.Tensor*) – Source boxes, e.g., object proposals.
- **gt_bboxes** (*torch.Tensor*) – Target of the transformation, e.g., ground-truth boxes.

Returns Box transformation deltas

Return type torch.Tensor

class mmrotate.core.bbox.**GVFixCoder**(*angle_range='oc'*, ***kwargs*)

Gliding vertex fix coder.

this coder encodes bbox (cx, cy, w, h, a) into delta (dt, dr, dd, dl) and decodes delta (dt, dr, dd, dl) back to original bbox (cx, cy, w, h, a).

Parameters **angle_range** (*str, optional*) – Angle representations. Defaults to 'oc'.

decode(*hbboxes*, *fix_deltas*)

Apply transformation *fix_deltas* to *boxes*.

Parameters

- **hbboxes** (*torch.Tensor*) – Basic boxes. Shape (B, N, 4) or (N, 4)
- **fix_deltas** (*torch.Tensor*) – Encoded offsets with respect to each roi. Has shape (B, N, num_classes * 4) or (B, N, 4) or (N, num_classes * 4) or (N, 4). Note N = num_anchors * W * H when rois is a grid of anchors.

Returns Decoded boxes.

Return type torch.Tensor

encode(*rbboxes*)
Get box regression transformation deltas.

Parameters **rbboxes** (*torch.Tensor*) – Source boxes, e.g., object proposals.

Returns Box transformation deltas

Return type torch.Tensor

class mmrotate.core.bbox.**GVRatioCoder**(*angle_range='oc'*, ***kwargs*)
Gliding vertex ratio coder.

this coder encodes bbox (cx, cy, w, h, a) into delta (ratios).

Parameters **angle_range** (*str, optional*) – Angle representations. Defaults to 'oc'.

decode(*bboxes*, *bboxes_pred*)
Apply transformation *fix_deltas* to *boxes*.

Parameters

- **bboxes** (*torch.Tensor*) –
- **bboxes_pred** (*torch.Tensor*) –

Returns NotImplementedError

encode(*rbboxes*)
Get box regression transformation deltas.

Parameters **rbboxes** (*torch.Tensor*) – Source boxes, e.g., object proposals.

Returns Box transformation deltas

Return type torch.Tensor

class mmrotate.core.bbox.**GaussianMixture**(*n_components*, *n_features=2*, *mu_init=None*, *var_init=None*, *eps=1e-06*, *requires_grad=False*)
Initializes the Gaussian mixture model and brings all tensors into their required shape.

Parameters

- **n_components** (*int*) – number of components.
- **n_features** (*int, optional*) – number of features.
- **mu_init** (*torch.Tensor, optional*) – (T, k, d)
- **var_init** (*torch.Tensor, optional*) – (T, k, d) or (T, k, d, d)
- **eps** (*float, optional*) – Defaults to 1e-6.
- **requires_grad** (*bool, optional*) – Defaults to False.

EM_step(*x*, *log_resp*)
From the log-probabilities, computes new parameters pi, mu, var (that maximize the log-likelihood). This is the maximization step of the EM-algorithm.

Parameters

- **x** (*torch.Tensor*) – (T, n, d) or (T, n, 1, d)
- **log_resp** (*torch.Tensor*) – (T, n, k, 1)

Returns pi (torch.Tensor): (T, k, 1) mu (torch.Tensor): (T, k, d) var (torch.Tensor): (T, k, d) or (T, k, d, d)

Return type tuple

`check_size`(*x*)
Make sure that the shape of x is (T, n, 1, d).

Parameters **x** (*torch.Tensor*) – input tensor.

Returns output tensor.

Return type torch.Tensor

`em_runner`(*x*)
Performs one iteration of the expectation-maximization algorithm by calling the respective subroutines.

Parameters **x** (*torch.Tensor*) – (n, 1, d)

`estimate_log_prob`(*x*)
Estimate the log-likelihood probability that samples belong to the k-th Gaussian.

Parameters **x** (*torch.Tensor*) – (T, n, d) or (T, n, 1, d)

Returns log-likelihood probability that samples belong to the k-th Gaussian with dimensions (T, n, k, 1).

Return type torch.Tensor

`fit`(*x*, *delta=0.001*, *n_iter=10*)
Fits Gaussian mixture model to the data.

Parameters

- **x** (*torch.Tensor*) – input tensor.
- **delta** (*float, optional*) – threshold.
- **n_iter** (*int, optional*) – number of iterations.

`get_score`(*x*, *sum_data=True*)
Computes the log-likelihood of the data under the model.

Parameters

- **x** (*torch.Tensor*) – (T, n, 1, d)
- **sum_data** (*bool,optional*) – Flag of whether to sum scores.

Returns score or per_sample_score.

Return type torch.Tensor

`log_resp_step`(*x*)
Computes log-responses that indicate the (logarithmic) posterior belief (sometimes called responsibilities) that a data point was generated by one of the k mixture components. Also returns the mean of the mean of the logarithms of the probabilities (as is done in sklearn). This is the so-called expectation step of the EM-algorithm.

Parameters **x** (*torch.Tensor*) – (T, n, d) or (T, n, 1, d)

Returns log_prob_norm (torch.Tensor): the mean of the mean of the logarithms of the probabilities. log_resp (torch.Tensor): log-responses that indicate the posterior belief.

Return type tuple

update_mu(*mu*)
Updates mean to the provided value.

Parameters **mu** (*torch.Tensor*) –

update_pi(*pi*)
Updates pi to the provided value.

Parameters **pi** (*torch.Tensor*) – (T, k, 1)

update_var(*var*)
Updates variance to the provided value.

Parameters **var** (*torch.Tensor*) – (T, k, d) or (T, k, d, d)

class mmrotate.core.bbox.**MaxConvexIoUAssigner**(*pos_iou_thr*, *neg_iou_thr*, *min_pos_iou=0.0*, *gt_max_assign_all=True*, *ignore_iof_thr=- 1*, *ignore_wrt_candidates=True*, *gpu_assign_thr=- 1*)

Assign a corresponding gt bbox or background to each bbox. Each proposals will be assigned with *-1*, or a semi-positive integer indicating the ground truth index.

- -1: negative sample, no assigned gt
- semi-positive integer: positive sample, index (0-based) of assigned gt

Parameters

- **pos_iou_thr** (*float*) – IoU threshold for positive bboxes.
- **neg_iou_thr** (*float or tuple*) – IoU threshold for negative bboxes.
- **min_pos_iou** (*float*) – Minimum iou for a bbox to be considered as a positive bbox. Positive samples can have smaller IoU than pos_iou_thr due to the 4th step (assign max IoU sample to each gt).
- **gt_max_assign_all** (*bool*) – Whether to assign all bboxes with the same highest overlap with some gt to that gt.
- **ignore_iof_thr** (*float*) – IoF threshold for ignoring bboxes (if *gt_bboxes_ignore* is specified). Negative values mean not ignoring any bboxes.
- **ignore_wrt_candidates** (*bool*) – Whether to compute the iof between *bboxes* and *gt_bboxes_ignore*, or the contrary.
- **gpu_assign_thr** (*int*) – The upper bound of the number of GT for GPU assign. When the number of gt is above this threshold, will assign on CPU device. Negative values mean not assign on CPU.

assign(*points*, *gt_rbboxes*, *overlaps*, *gt_rbboxes_ignore=None*, *gt_labels=None*)
Assign gt to bboxes.

The assignment is done in following steps

1. compute iou between all bbox (bbox of all pyramid levels) and gt
2. compute center distance between all bbox and gt
3. on each pyramid level, for each gt, select k bbox whose center are closest to the gt center, so we total select k*l bbox as candidates for each gt
4. get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold
5. select these candidates whose iou are greater than or equal to the threshold as positive

6. limit the positive sample's center in gt

Parameters

- **points** (*torch.Tensor*) – Points to be assigned, shape(n, 18).
- **gt_rbboxes** (*torch.Tensor*) – Groundtruth polygons, shape (k, 8).
- **overlaps** (*torch.Tensor*) – Overlaps between k gt_bboxes and n bboxes, shape(k, n).
- **gt_rbboxes_ignore** (*Tensor, optional*) – Ground truth polygons that are labelled as *ignored*, e.g., crowd boxes in COCO.
- **gt_labels** (*Tensor, optional*) – Label of gt_bboxes, shape (k,).

Returns The assign result.

Return type AssignResult

assign_wrt_overlaps(*overlaps*, *gt_labels=None*)

Assign w.r.t.

the overlaps of bboxes with gts.

Parameters

- **overlaps** (*torch.Tensor*) – Overlaps between k gt_bboxes and n bboxes, shape(k, n).
- **gt_labels** (*Tensor, optional*) – Labels of k gt_bboxes, shape (k,).

Returns The assign result.

Return type AssignResult

convex_overlaps(*gt_rbboxes*, *points*)

Compute overlaps between polygons and points.

Parameters

- **gt_rbboxes** (*torch.Tensor*) – Groundtruth polygons, shape (k, 8).
- **points** (*torch.Tensor*) – Points to be assigned, shape(n, 18).

Returns Overlaps between k gt_bboxes and n bboxes, shape(k, n).

Return type overlaps (torch.Tensor)

class mmrotate.core.bbox.**MidpointOffsetCoder**(*target_means=(0.0, 0.0, 0.0, 0.0, 0.0, 0.0)*, *target_stds=(1.0, 1.0, 1.0, 1.0, 1.0, 1.0)*, *angle_range='oc'*)

Mid point offset coder. This coder encodes bbox (x1, y1, x2, y2) into delta (dx, dy, dw, dh, da, db) and decodes delta (dx, dy, dw, dh, da, db) back to original bbox (x1, y1, x2, y2).

Parameters

- **target_means** (*Sequence[float]*) – Denormalizing means of target for delta coordinates
- **target_stds** (*Sequence[float]*) – Denormalizing standard deviation of target for delta coordinates
- **angle_range** (*str, optional*) – Angle representations. Defaults to 'oc'.

decode(*bboxes*, *pred_bboxes*, *max_shape=None*, *wh_ratio_clip=0.016*)

Apply transformation *pred_bboxes* to *bboxes*.

Parameters

- **bboxes** (*torch.Tensor*) – Basic boxes. Shape (B, N, 4) or (N, 4)
- **pred_bboxes** (*torch.Tensor*) – Encoded offsets with respect to each roi. Has shape (B, N, 5) or (N, 5). Note N = num_anchors * W * H when rois is a grid of anchors.
- **(Sequence[int] or torch.Tensor or Sequence[** (*max_shape*) – Sequence[int]],optional): Maximum bounds for boxes, specifies (H, W, C) or (H, W). If bboxes shape is (B, N, 6), then the max_shape should be a Sequence[Sequence[int]] and the length of max_shape should also be B.
- **wh_ratio_clip** (*float, optional*) – The allowed ratio between width and height.

Returns Decoded boxes.

Return type torch.Tensor

encode(*bboxes*, *gt_bboxes*)

Get box regression transformation deltas that can be used to transform the `bboxes` into the `gt_bboxes`.

Parameters

- **bboxes** (*torch.Tensor*) – Source boxes, e.g., object proposals.
- **gt_bboxes** (*torch.Tensor*) – Target of the transformation, e.g., ground-truth boxes.

Returns Box transformation deltas

Return type torch.Tensor

class mmrotate.core.bbox.**RBboxOverlaps2D**

2D Overlaps (e.g. IoUs, GIoUs) Calculator.

class mmrotate.core.bbox.**RRandomSampler**(*num*, *pos_fraction*, *neg_pos_ub=- 1*, *add_gt_as_proposals=True*, ***kwargs*)

Random sampler.

Parameters

- **num** (*int*) – Number of samples
- **pos_fraction** (*float*) – Fraction of positive samples
- **neg_pos_up** (*int, optional*) – Upper bound number of negative and positive samples. Defaults to -1.
- **add_gt_as_proposals** (*bool, optional*) – Whether to add ground truth boxes as proposals. Defaults to True.

random_choice(*gallery*, *num*)

Random select some elements from the gallery.

If *gallery* is a Tensor, the returned indices will be a Tensor; If *gallery* is a ndarray or list, the returned indices will be a ndarray.

Parameters

- **gallery** (*Tensor | ndarray | list*) – indices pool.
- **num** (*int*) – expected sample num.

Returns sampled indices.

Return type Tensor or ndarray

sample(*assign_result*, *bboxes*, *gt_bboxes*, *gt_labels=None*, ***kwargs*)

Sample positive and negative bboxes.

This is a simple implementation of bbox sampling given candidates, assigning results and ground truth bboxes.

Parameters

- **assign_result** (AssignResult) – Bbox assigning results.
- **bboxes** (*torch.Tensor*) – Boxes to be sampled from.
- **gt_bboxes** (*torch.Tensor*) – Ground truth bboxes.
- **gt_labels** (*Tensor, optional*) – Class labels of ground truth bboxes.

Returns Sampling result.

Return type SamplingResult

Example

```
>>> from mmdet.core.bbox import RandomSampler
>>> from mmdet.core.bbox import AssignResult
>>> from mmdet.core.bbox.demodata import ensure_rng, random_boxes
>>> rng = ensure_rng(None)
>>> assign_result = AssignResult.random(rng=rng)
>>> bboxes = random_boxes(assign_result.num_preds, rng=rng)
>>> gt_bboxes = random_boxes(assign_result.num_gts, rng=rng)
>>> gt_labels = None
>>> self = RandomSampler(num=32, pos_fraction=0.5, neg_pos_ub=-1,
>>>                      add_gt_as_proposals=False)
>>> self = self.sample(assign_result, bboxes, gt_bboxes, gt_labels)
```

class mmrotate.core.bbox.**SASAssigner**(*topk*)

Assign a corresponding gt bbox or background to each bbox. Each proposals will be assigned with *0* or a positive integer indicating the ground truth index.

- 0: negative sample, no assigned gt
- positive integer: positive sample, index (1-based) of assigned gt

Parameters

- **scale** (*float*) – IoU threshold for positive bboxes.
- **pos_num** (*float*) – find the nearest pos_num points to gt center in this
- **level.** –

assign(*bboxes*, *num_level_bboxes*, *gt_bboxes*, *gt_bboxes_ignore=None*, *gt_labels=None*)

Assign gt to bboxes.

The assignment is done in following steps

1. compute iou between all bbox (bbox of all pyramid levels) and gt
2. compute center distance between all bbox and gt
3. on each pyramid level, for each gt, select k bbox whose center are closest to the gt center, so we total select k*l bbox as candidates for each gt
4. get corresponding iou for the these candidates, and compute the mean and std, set mean + std as the iou threshold

5. select these candidates whose iou are greater than or equal to the threshold as positive
6. limit the positive sample's center in gt

Parameters

- **bboxes** (*torch.Tensor*) – Bounding boxes to be assigned, shape(n, 4).
- **num_level_bboxes** (*List*) – num of bboxes in each level
- **gt_bboxes** (*torch.Tensor*) – Groundtruth boxes, shape (k, 4).
- **gt_bboxes_ignore** (*Tensor, optional*) – Ground truth bboxes that are labelled as *ignored*, e.g., crowd boxes in COCO.
- **gt_labels** (*Tensor, optional*) – Label of gt_bboxes, shape (k,).

Returns The assign result.

Return type AssignResult

mmrotate.core.bbox.**bbox_mapping_back**(*bboxes*, *img_shape*, *scale_factor*, *flip*, *flip_direction='horizontal'*)
Map bboxes from testing scale to original image scale.

mmrotate.core.bbox.**build_assigner**(*cfg*, ***default_args*)
Builder of box assigner.

mmrotate.core.bbox.**build_bbox_coder**(*cfg*, ***default_args*)
Builder of box coder.

mmrotate.core.bbox.**build_sampler**(*cfg*, ***default_args*)
Builder of box sampler.

mmrotate.core.bbox.**gaussian2bbox**(*gmm*)
Convert Gaussian distribution to polygons by SVD.

Parameters **gmm** (*dict[str, torch.Tensor]*) – Dict of Gaussian distribution.

Returns Polygons.

Return type torch.Tensor

mmrotate.core.bbox.**gt2gaussian**(*target*)
Convert polygons to Gaussian distributions.

Parameters **target** (*torch.Tensor*) – Polygons with shape (N, 8).

Returns Gaussian distributions.

Return type dict[str, torch.Tensor]

mmrotate.core.bbox.**hbb2obb**(*hbboxes*, *version='oc'*)
Convert horizontal bounding boxes to oriented bounding boxes.

Parameters

- **hbbs** (*torch.Tensor*) – [x_lt,y_lt,x_rb,y_rb]
- **version** (*Str*) – angle representations.

Returns [x_ctr,y_ctr,w,h,angle]

Return type obbs (torch.Tensor)

mmrotate.core.bbox.**norm_angle**(*angle*, *angle_range*)
Limit the range of angles.

Parameters

- **angle** (*ndarray*) – shape(n,).
- **angle_range** (*Str*) – angle representations.

Returns shape(n,).

Return type angle (ndarray)

mmrotate.core.bbox.**obb2hbb**(*rbboxes*, *version='oc'*)
Convert oriented bounding boxes to horizontal bounding boxes.

Parameters

- **obbs** (*torch.Tensor*) – [x_ctr,y_ctr,w,h,angle]
- **version** (*Str*) – angle representations.

Returns [x_ctr,y_ctr,w,h,-pi/2]

Return type hbbs (torch.Tensor)

mmrotate.core.bbox.**obb2poly**(*rbboxes*, *version='oc'*)
Convert oriented bounding boxes to polygons.

Parameters

- **obbs** (*torch.Tensor*) – [x_ctr,y_ctr,w,h,angle]
- **version** (*Str*) – angle representations.

Returns [x0,y0,x1,y1,x2,y2,x3,y3]

Return type polys (torch.Tensor)

mmrotate.core.bbox.**obb2poly_np**(*rbboxes*, *version='oc'*)
Convert oriented bounding boxes to polygons.

Parameters

- **obbs** (*ndarray*) – [x_ctr,y_ctr,w,h,angle]
- **version** (*Str*) – angle representations.

Returns [x0,y0,x1,y1,x2,y2,x3,y3]

Return type polys (ndarray)

mmrotate.core.bbox.**obb2xyxy**(*rbboxes*, *version='oc'*)
Convert oriented bounding boxes to horizontal bounding boxes.

Parameters

- **obbs** (*torch.Tensor*) – [x_ctr,y_ctr,w,h,angle]
- **version** (*Str*) – angle representations.

Returns [x_lt,y_lt,x_rb,y_rb]

Return type hbbs (torch.Tensor)

mmrotate.core.bbox.**poly2obb**(*polys*, *version='oc'*)
Convert polygons to oriented bounding boxes.

Parameters

- **polys** (*torch.Tensor*) – [x0,y0,x1,y1,x2,y2,x3,y3]
- **version** (*Str*) – angle representations.

Returns [x_ctr,y_ctr,w,h,angle]

Return type obbs (torch.Tensor)

mmrotate.core.bbox.**poly2obb_np**(*polys*, *version='oc'*)
Convert polygons to oriented bounding boxes.

Parameters

- **polys** (*ndarray*) – [x0,y0,x1,y1,x2,y2,x3,y3]
- **version** (*Str*) – angle representations.

Returns [x_ctr,y_ctr,w,h,angle]

Return type obbs (ndarray)

mmrotate.core.bbox.**rbbox2result**(*bboxes*, *labels*, *num_classes*)
Convert detection results to a list of numpy arrays.

Parameters

- **bboxes** (*torch.Tensor*) – shape (n, 6)
- **labels** (*torch.Tensor*) – shape (n,)
- **num_classes** (*int*) – class number, including background class

Returns bbox results of each class

Return type list(ndarray)

mmrotate.core.bbox.**rbbox2roi**(*bbox_list*)
Convert a list of bboxes to roi format.

Parameters **bbox_list** (*list[Tensor]*) – a list of bboxes corresponding to a batch of images.

Returns shape (n, 6), [batch_ind, cx, cy, w, h, a]

Return type Tensor

mmrotate.core.bbox.**rbbox_overlaps**(*bboxes1*, *bboxes2*, *mode='iou'*, *is_aligned=False*)
Calculate overlap between two set of bboxes.

Parameters

- **bboxes1** (*torch.Tensor*) – shape (B, m, 5) in <cx, cy, w, h, a> format or empty.
- **bboxes2** (*torch.Tensor*) – shape (B, n, 5) in <cx, cy, w, h, a> format or empty.
- **mode** (*str*) – "iou" (intersection over union), "iof" (intersection over foreground) or "giou" (generalized intersection over union). Default "iou".
- **is_aligned** (*bool, optional*) – If True, then m and n must be equal. Default False.

Returns shape (m, n) if `is_aligned` is False else shape (m,)

Return type Tensor

18.3 patch

mmrotate.core.patch.**get_multiscale_patch**(*sizes*, *steps*, *ratios*)
Get multiscale patch sizes and steps.

Parameters

- **sizes** (*list*) – A list of patch sizes.
- **steps** (*list*) – A list of steps to slide patches.
- **ratios** (*list*) – Multiscale ratios. devidie to each size and step and generate patches in new scales.

Returns A list of multiscale patch sizes. new_steps (list): A list of steps corresponding to new_sizes.

Return type new_sizes (list)

mmrotate.core.patch.**merge_results**(*results*, *offsets*, *iou_thr=0.1*, *device='cpu'*)
Merge patch results via nms.

Parameters

- **results** (*list[np.ndarray]*) – A list of patches results.
- **offsets** (*np.ndarray*) – Positions of the left top points of patches.
- **iou_thr** (*float*) – The IoU threshold of NMS.
- **device** (*str*) – The device to call nms.

Retunrns: list[np.ndarray]: Detection results after merging.

mmrotate.core.patch.**slide_window**(*width*, *height*, *sizes*, *steps*, *img_rate_thr=0.6*)
Slide windows in images and get window position.

Parameters

- **width** (*int*) – The width of the image.
- **height** (*int*) – The height of the image.
- **sizes** (*list*) – List of window's sizes.
- **steps** (*list*) – List of window's steps.
- **img_rate_thr** (*float*) – Threshold of window area divided by image area.

Returns Information of valid windows.

Return type np.ndarray

18.4 evaluation

mmrotate.core.evaluation.**eval_rbbox_map**(*det_results*, *annotations*, *scale_ranges=None*, *iou_thr=0.5*, *use_07_metric=True*, *dataset=None*, *logger=None*, *nproc=4*)
Evaluate mAP of a rotated dataset.

Parameters

- **det_results** (*list[list]*) – [[cls1_det, cls2_det, ...], ...]. The outer list indicates images, and the inner list indicates per-class detected bboxes.

- **annotations** (*list[dict]*) – Ground truth annotations where each item of the list indicates an image. Keys of annotations are:
  - *bboxes*: numpy array of shape (n, 5)
  - *labels*: numpy array of shape (n,)
  - *bboxes_ignore* (optional): numpy array of shape (k, 5)
  - *labels_ignore* (optional): numpy array of shape (k,)
- **scale_ranges** (*list[tuple] | None*) – Range of scales to be evaluated, in the format [(min1, max1), (min2, max2), ...]. A range of (32, 64) means the area range between (32**2, 64**2). Default: None.
- **iou_thr** (*float*) – IoU threshold to be considered as matched. Default: 0.5.
- **use_07_metric** (*bool*) – Whether to use the voc07 metric.
- **dataset** (*list[str] | str | None*) – Dataset name or dataset classes, there are minor differences in metrics for different datasets, e.g. "voc07", "imagenet_det", etc. Default: None.
- **logger** (*logging.Logger | str | None*) – The way to print the mAP summary. See *mmcv.utils.print_log()* for details. Default: None.
- **nproc** (*int*) – Processes used for computing TP and FP. Default: 4.

Returns (mAP, [dict, dict, ...])

Return type tuple

18.5 post_processing

mmrotate.core.post_processing.**aug_multiclass_nms_rotated**(*merged_bboxes*, *merged_labels*, *score_thr*, *nms*, *max_num*, *classes*)

NMS for aug multi-class bboxes.

Parameters

- **multi_bboxes** (*torch.Tensor*) – shape (n, #class*5) or (n, 5)
- **multi_scores** (*torch.Tensor*) – shape (n, #class), where the last column contains scores of the background class, but this will be ignored.
- **score_thr** (*float*) – bbox threshold, bboxes with scores lower than it will not be considered.
- **nms** (*float*) – Config of NMS.
- **max_num** (*int, optional*) – if there are more than max_num bboxes after NMS, only top max_num will be kept. Default to -1.
- **classes** (*int*) – number of classes.

Returns

tensors of shape (k, 5), and (k). Dets are boxes with scores. Labels are 0-based.

Return type tuple (dets, labels)

mmrotate.core.post_processing.**multiclass_nms_rotated**(*multi_bboxes*, *multi_scores*, *score_thr*, *nms*, *max_num=- 1*, *score_factors=None*, *return_inds=False*)

NMS for multi-class bboxes.

Parameters

- **multi_bboxes** (*torch.Tensor*) – shape (n, #class*5) or (n, 5)
- **multi_scores** (*torch.Tensor*) – shape (n, #class), where the last column contains scores of the background class, but this will be ignored.
- **score_thr** (*float*) – bbox threshold, bboxes with scores lower than it will not be considered.
- **nms** (*float*) – Config of NMS.
- **max_num** (*int, optional*) – if there are more than max_num bboxes after NMS, only top max_num will be kept. Default to -1.
- **score_factors** (*Tensor, optional*) – The factors multiplied to scores before applying NMS. Default to None.
- **return_inds** (*bool, optional*) – Whether return the indices of kept bboxes. Default to False.

Returns tensors of shape (k, 5), (k), and (k). Dets are boxes with scores. Labels are 0-based.

Return type tuple (dets, labels, indices (optional))

CHAPTER

NINETEEN

MMROTATE.DATASETS

19.1 datasets

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

DOTA dataset for detection.

Parameters

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

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

Evaluate the dataset.

Parameters

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

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

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

Parameters

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

- **nproc** (*int, optional*) – number of process.

Returns

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

Return type tuple

load_annotations(*ann_folder*)

Parameters **ann_folder** – folder that contains DOTA v1 annotations txt files

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

Merging patch bboxes into full image.

Parameters

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

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

HRSC dataset for detection.

Parameters

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

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

Evaluate the dataset.

Parameters

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

load_annotations(*ann_file*)
Load annotation from XML style ann_file.

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

Returns Annotation info from XML file.

Return type list[dict]

class mmrotate.datasets.**SARDataset**(*ann_file*, *pipeline*, *version='oc'*, *difficulty=100*, ***kwargs*)
SAR ship dataset for detection (Support RSSDD and HRSID).

19.2 pipelines

class mmrotate.datasets.pipelines.**LoadPatchFromImage**(*to_float32=False*, *color_type='color'*, *channel_order='bgr'*, *file_client_args={'backend': 'disk'}*)
Load an patch from the huge image.

Similar with `LoadImageFromFile`, but only reserve a patch of `results['img']` according to `results['win']`.

class mmrotate.datasets.pipelines.**PolyRandomRotate**(*rotate_ratio=0.5*, *angles_range=180*, *auto_bound=False*, *rect_classes=None*, *version='le90'*)
Rotate img & bbox. Reference: https://github.com/hukaixuan19970627/OrientedRepPoints_DOTA

Parameters

- **rate** (*bool*) – (float, optional): The rotating probability. Default: 0.5.
- **angles_range** (*int, optional*) – The rotate angle defined by random (-angles_range, +angles_range).
- **auto_bound** (*bool, optional*) – whether to find the new width and height bounds.
- **rect_classes** (*None|list, optional*) – Specifies classes that needs to be rotated by a multiple of 90 degrees.
- **version** (*str, optional*) – Angle representations. Defaults to 'oc'.

apply_coords(*coords*)
coords should be a N * 2 array-like, containing N couples of (x, y) points

apply_image(*img*, *bound_h*, *bound_w*, *interp=1*)
img should be a numpy array, formatted as Height * Width * Nchannels

create_rotation_matrix(*center*, *angle*, *bound_h*, *bound_w*, *offset=0*)
Create rotation matrix.

filter_border(*bboxes*, *h*, *w*)
Filter the box whose center point is outside or whose side length is less than 5.

property is_rotate
Randomly decide whether to rotate.

class mmrotate.datasets.pipelines.**RRandomFlip**(*flip_ratio=None*, *direction='horizontal'*, *version='oc'*)

Parameters

- **flip_ratio** (*float | list[float], optional*) – The flipping probability. Default: None.
- **direction** (*str | list[str], optional*) – The flipping direction. Options are 'horizontal', 'vertical', 'diagonal'.
- **version** (*str, optional*) – Angle representations. Defaults to 'oc'.

bbox_flip(*bboxes*, *img_shape*, *direction*)
Flip bboxes horizontally or vertically.

Parameters

- **bboxes** (*ndarray*) – shape (..., 5*k)
- **img_shape** (*tuple*) – (height, width)

Returns Flipped bounding boxes.

Return type numpy.ndarray

class mmrotate.datasets.pipelines.**RResize**(*img_scale=None*, *multiscale_mode='range'*, *ratio_range=None*)
Resize images & rotated bbox Inherit Resize pipeline class to handle rotated bboxes.

Parameters

- **img_scale** (*tuple or list[tuple]*) – Images scales for resizing.
- **multiscale_mode** (*str*) – Either "range" or "value".
- **ratio_range** (*tuple[float]*) – (min_ratio, max_ratio).

CHAPTER

TWENTY

MMROTATE.MODELS

20.1 detectors

class mmrotate.models.detectors.**GlidingVertex**(*backbone*, *rpn_head*, *roi_head*, *train_cfg*, *test_cfg*, *neck=None*, *pretrained=None*, *init_cfg=None*)

Implementation of Gliding Vertex on the Horizontal Bounding Box for Multi-Oriented Object Detection

class mmrotate.models.detectors.**OrientedRCNN**(*backbone*, *rpn_head*, *roi_head*, *train_cfg*, *test_cfg*, *neck=None*, *pretrained=None*, *init_cfg=None*)

Implementation of Oriented R-CNN for Object Detection.

class mmrotate.models.detectors.**R3Det**(*num_refine_stages*, *backbone*, *neck=None*, *bbox_head=None*, *frm_cfgs=None*, *refine_heads=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

Rotated Refinement RetinaNet.

aug_test(*imgs*, *img_metas*, ***kwargs*)

Test function with test time augmentation.

extract_feat(*img*)

Directly extract features from the backbone+neck.

forward_dummy(*img*)

Used for computing network flops.

See *mmedetection/tools/get_flops.py*

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*)

Forward function.

simple_test(*img*, *img_meta*, *rescale=False*)

Test function without test time augmentation.

Parameters

- **imgs** (*list[torch.Tensor]*) – List of multiple images
- **img_metas** (*list[dict]*) – List of image information.
- **rescale** (*bool, optional*) – Whether to rescale the results. Defaults to False.

Returns BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

class mmrotate.models.detectors.**ReDet**(*backbone*, *rpn_head*, *roi_head*, *train_cfg*, *test_cfg*, *neck=None*, *pretrained=None*, *init_cfg=None*)

Implementation of ReDet: A Rotation-equivariant Detector for Aerial Object Detection.

class mmrotate.models.detectors.**RoITransformer**(*backbone*, *rpn_head*, *roi_head*, *train_cfg*, *test_cfg*, *neck=None*, *pretrained=None*, *init_cfg=None*)
Implementation of Learning RoI Transformer for Oriented Object Detection in Aerial Images.

class mmrotate.models.detectors.**RotatedBaseDetector**(*init_cfg=None*)
Base class for rotated detectors.

show_result(*img*, *result*, *score_thr=0.3*, *bbox_color=(226, 43, 138)*, *text_color='white'*, *thickness=2*, *font_scale=0.25*, *win_name=''*, *show=False*, *wait_time=0*, *out_file=None*, ***kwargs*)
Draw *result* over *img*.

Parameters

- **img** (*str or Tensor*) – The image to be displayed.
- **result** (*Tensor or tuple*) – The results to draw over *img* bbox_result or (bbox_result, segm_result).
- **score_thr** (*float, optional*) – Minimum score of bboxes to be shown. Default: 0.3.
- **bbox_color** (str or tuple or Color) – Color of bbox lines.
- **text_color** (str or tuple or Color) – Color of texts.
- **thickness** (*int*) – Thickness of lines.
- **font_scale** (*float*) – Font scales of texts.
- **win_name** (*str*) – The window name.
- **wait_time** (*int*) – Value of waitKey param. Default: 0.
- **show** (*bool*) – Whether to show the image. Default: False.
- **out_file** (*str or None*) – The filename to write the image. Default: None.

Returns Only if not *show* or *out_file*

Return type img (torch.Tensor)

class mmrotate.models.detectors.**RotatedFasterRCNN**(*backbone*, *rpn_head*, *roi_head*, *train_cfg*, *test_cfg*, *neck=None*, *pretrained=None*, *init_cfg=None*)
Implementation of Rotated Faster R-CNN.

class mmrotate.models.detectors.**RotatedRepPoints**(*backbone*, *neck*, *bbox_head*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*)
Implementation of Rotated RepPoints.

class mmrotate.models.detectors.**RotatedRetinaNet**(*backbone*, *neck*, *bbox_head*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
Implementation of Rotated RetinaNet.

class mmrotate.models.detectors.**RotatedSingleStageDetector**(*backbone*, *neck=None*, *bbox_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)
Base class for rotated single-stage detectors.

Single-stage detectors directly and densely predict bounding boxes on the output features of the backbone+neck.

aug_test(*imgs*, *img_metas*, *rescale=False*)
Test function with test time augmentation.

Parameters

- **imgs** (*list[Tensor]*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains all images in the batch.
- **img_metas** (*list[list[dict]]*) – the outer list indicates test-time augs (multiscale, flip, etc.) and the inner list indicates images in a batch. each dict has image information.
- **rescale** (*bool, optional*) – Whether to rescale the results. Defaults to False.

Returns

BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

extract_feat(*img*)
Directly extract features from the backbone+neck.

forward_dummy(*img*)
Used for computing network flops.

See *mmdetection/tools/analysis_tools/get_flops.py*

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*)

Parameters

- **img** (*Tensor*) – Input images of shape (N, C, H, W). Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – A List of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see `mmdet.datasets.pipelines.Collect`.
- **gt_bboxes** (*list[Tensor]*) – Each item are the truth boxes for each image in [tl_x, tl_y, br_x, br_y] format.
- **gt_labels** (*list[Tensor]*) – Class indices corresponding to each box
- **gt_bboxes_ignore** (*None | list[Tensor]*) – Specify which bounding boxes can be ignored when computing the loss.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

simple_test(*img*, *img_metas*, *rescale=False*)
Test function without test time augmentation.

Parameters

- **imgs** (*list[torch.Tensor]*) – List of multiple images
- **img_metas** (*list[dict]*) – List of image information.
- **rescale** (*bool, optional*) – Whether to rescale the results. Defaults to False.

Returns BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

class mmrotate.models.detectors.**RotatedTwoStageDetector**(*backbone*, *neck=None*, *rpn_head=None*, *roi_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*)

Base class for rotated two-stage detectors.

Two-stage detectors typically consisting of a region proposal network and a task-specific regression head.

async async_simple_test(*img*, *img_meta*, *proposals=None*, *rescale=False*)
Async test without augmentation.

aug_test(*imgs*, *img_metas*, *rescale=False*)
Test with augmentations.

If rescale is False, then returned bboxes and masks will fit the scale of imgs[0].

extract_feat(*img*)
Directly extract features from the backbone+neck.

forward_dummy(*img*)
Used for computing network flops.

See *mmdetection/tools/analysis_tools/get_flops.py*

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*, *gt_masks=None*, *proposals=None*, ***kwargs*)

Parameters

- **img** (*Tensor*) – of shape (N, C, H, W) encoding input images. Typically these should be mean centered and std scaled.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 5) in [cx, cy, w, h, a] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task.
- **proposals** – override rpn proposals with custom proposals. Use when *with_rpn* is False.

Returns a dictionary of loss components

Return type dict[str, Tensor]

simple_test(*img*, *img_metas*, *proposals=None*, *rescale=False*)
Test without augmentation.

property with_roi_head
whether the detector has a RoI head

Type bool

property with_rpn
whether the detector has RPN

Type bool

class mmrotate.models.detectors.**S2ANet**(*backbone*, *neck=None*, *fam_head=None*, *align_cfgs=None*, *odm_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*)

Implementation of Align Deep Features for Oriented Object Detection.

aug_test(*imgs*, *img_metas*, ***kwargs*)
Test function with test time augmentation.

extract_feat(*img*)
Directly extract features from the backbone+neck.

forward_dummy(*img*)
Used for computing network flops.

See *mmedetection/tools/get_flops.py*

forward_train(*img*, *img_metas*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*)
Forward function of S2ANet.

simple_test(*img*, *img_meta*, *rescale=False*)
Test function without test time augmentation.

Parameters

- **imgs** (*list[torch.Tensor]*) – List of multiple images
- **img_metas** (*list[dict]*) – List of image information.
- **rescale** (*bool, optional*) – Whether to rescale the results. Defaults to False.

Returns BBox results of each image and classes. The outer list corresponds to each image. The inner list corresponds to each class.

Return type list[list[np.ndarray]]

20.2 backbones

class mmrotate.models.backbones.**ReResNet**(*depth*, *in_channels=3*, *stem_channels=64*, *base_channels=64*, *expansion=None*, *num_stages=4*, *strides=(1, 2, 2, 2)*, *dilations=(1, 1, 1, 1)*, *out_indices=(3)*, *style='pytorch'*, *deep_stem=False*, *avg_down=False*, *frozen_stages=- 1*, *conv_cfg=None*, *norm_cfg={'requires_grad': True, 'type': 'BN'}*, *norm_eval=False*, *with_cp=False*, *zero_init_residual=True*, *pretrained=None*, *init_cfg=None*)

ReResNet backbone.

Please refer to the paper for details.

Parameters

- **depth** (*int*) – Network depth, from {18, 34, 50, 101, 152}.
- **in_channels** (*int*) – Number of input image channels. Default: 3.
- **stem_channels** (*int*) – Output channels of the stem layer. Default: 64.
- **base_channels** (*int*) – Middle channels of the first stage. Default: 64.
- **num_stages** (*int*) – Stages of the network. Default: 4.

- **strides** (*Sequence[int]*) – Strides of the first block of each stage. Default: `(1, 2, 2, 2)`.
- **dilations** (*Sequence[int]*) – Dilation of each stage. Default: `(1, 1, 1, 1)`.
- **out_indices** (*Sequence[int]*) – Output from which stages. If only one stage is specified, a single tensor (feature map) is returned, otherwise multiple stages are specified, a tuple of tensors will be returned. Default: `(3, )`.
- **style** (*str*) – *pytorch* or *caffe*. If set to "pytorch", the stride-two layer is the 3x3 conv layer, otherwise the stride-two layer is the first 1x1 conv layer.
- **deep_stem** (*bool*) – Replace 7x7 conv in input stem with 3 3x3 conv. Default: False.
- **avg_down** (*bool*) – Use AvgPool instead of stride conv when downsampling in the bottleneck. Default: False.
- **frozen_stages** (*int*) – Stages to be frozen (stop grad and set eval mode). -1 means not freezing any parameters. Default: -1.
- **conv_cfg** (*dict | None*) – The config dict for conv layers. Default: None.
- **norm_cfg** (*dict*) – The config dict for norm layers.
- **norm_eval** (*bool*) – Whether to set norm layers to eval mode, namely, freeze running stats (mean and var). Note: Effect on Batch Norm and its variants only. Default: False.
- **with_cp** (*bool*) – Use checkpoint or not. Using checkpoint will save some memory while slowing down the training speed. Default: False.
- **zero_init_residual** (*bool*) – Whether to use zero init for last norm layer in resblocks to let them behave as identity. Default: True.

forward(*x*)
Forward function of ReResNet.

make_res_layer(***kwargs*)
Build Reslayer.

property norm1
Get normalizion layer's name.

train(*mode=True*)
Train function of ReResNet.

20.3 necks

class mmrotate.models.necks.**ReFPN**(*in_channels*, *out_channels*, *num_outs*, *start_level=0*, *end_level=- 1*, *add_extra_convs=False*, *extra_convs_on_inputs=True*, *relu_before_extra_convs=False*, *no_norm_on_lateral=False*, *conv_cfg=None*, *norm_cfg=None*, *activation=None*, *init_cfg={'distribution': 'uniform', 'layer': 'Conv2d', 'type': 'Xavier'}*)

ReFPN.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale)
- **num_outs** (*int*) – Number of output scales.

- **start_level** (*int, optional*) – Index of the start input backbone level used to build the feature pyramid. Default: 0.
- **end_level** (*int, optional*) – Index of the end input backbone level (exclusive) to build the feature pyramid. Default: -1, which means the last level.
- **add_extra_convs** (*bool, optional*) – It decides whether to add conv layers on top of the original feature maps. Default to False.
- **extra_convs_on_inputs** (*bool, optional*) – It specifies the source feature map of the extra convs is the last feat map of neck inputs.
- **relu_before_extra_convs** (*bool*) – Whether to apply relu before the extra conv. Default: False.
- **no_norm_on_lateral** (*bool*) – Whether to apply norm on lateral. Default: False.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.
- **activation** (*str, optional*) – Activation layer in ConvModule. Default: None.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*inputs*)
Forward function of ReFPN.

20.4 dense_heads

class mmrotate.models.dense_heads.**CSLRRetinaHead**(*use_encoded_angle=True*, *shield_reg_angle=False*, *angle_coder={'angle_version': 'le90', 'omega': 1, 'radius': 6, 'type': 'CSLCoder', 'window': 'gaussian'}*, *loss_angle={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': True}*, *init_cfg={'layer': 'Conv2d', 'override': [{'type': 'Normal', 'name': 'retina_cls', 'std': 0.01, 'bias_prob': 0.01}, {'type': 'Normal', 'name': 'retina_angle_cls', 'std': 0.01, 'bias_prob': 0.01}], 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

Rotational Anchor-based refine head.

Parameters

- **use_encoded_angle** (*bool*) – Decide whether to use encoded angle or gt angle as target. Default: True.
- **shield_reg_angle** (*bool*) – Decide whether to shield the angle loss from reg branch. Default: False.
- **angle_coder** (*dict*) – Config of angle coder.
- **loss_angle** (*dict*) – Config of angle classification loss.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward_single(*x*)
Forward feature of a single scale level.

Parameters **x** (*torch.Tensor*) – Features of a single scale level.

Returns

- cls_score (torch.Tensor): Cls scores for a single scale level the channels number is num_anchors * num_classes.
- bbox_pred (torch.Tensor): Box energies / deltas for a single scale level, the channels number is num_anchors * 5.
- angle_cls (torch.Tensor): Angle for a single scale level the channels number is num_anchors * coding_len.

Return type tuple (torch.Tensor)

get_bboxes(*cls_scores*, *bbox_preds*, *angle_clses*, *img_metas*, *cfg=None*, *rescale=False*, *with_nms=True*)
Transform network output for a batch into bbox predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **angle_clses** (*list[Tensor]*) – Box angles for each scale level with shape (N, num_anchors * coding_len, H, W)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cfg** (*mmcv.Config | None*) – Test / postprocessing configuration, if None, test_cfg would be used
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Default: True.

Returns

Each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 5 columns are bounding box positions (cx, cy, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the predicted class label of the corresponding box.

Return type list[tuple[Tensor, Tensor]]

Example

```
>>> import mmcv
>>> self = AnchorHead(
>>>     num_classes=9,
>>>     in_channels=1,
>>>     anchor_generator=dict(
>>>         type='AnchorGenerator',
>>>         scales=[8],
>>>         ratios=[0.5, 1.0, 2.0],
>>>         strides=[4,]))
>>> img_metas = [{'img_shape': (32, 32, 3), 'scale_factor': 1}]
>>> cfg = mmcv.Config(dict(
>>>     score_thr=0.00,
>>>     nms=dict(type='nms', iou_thr=1.0),
>>>     max_per_img=10))
```

(continues on next page)

(continued from previous page)

```
>>> feat = torch.rand(1, 1, 3, 3)
>>> cls_score, bbox_pred = self.forward_single(feat)
>>> # Note the input lists are over different levels, not images
>>> cls_scores, bbox_preds = [cls_score], [bbox_pred]
>>> result_list = self.get_bboxes(cls_scores, bbox_preds,
>>>                               img_metas, cfg)
>>> det_bboxes, det_labels = result_list[0]
>>> assert len(result_list) == 1
>>> assert det_bboxes.shape[1] == 5
>>> assert len(det_bboxes) == len(det_labels) == cfg.max_per_img
```

loss(*cls_scores*, *bbox_preds*, *angle_clses*, *gt_bboxes*, *gt_labels*, *img_metas*, *gt_bboxes_ignore=None*)

Compute losses of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **angle_clses** (*list[Tensor]*) – Box angles for each scale level with shape (N, num_anchors * coding_len, H, W)
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 5) in [cx, cy, w, h, a] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss. Default: None

Returns A dictionary of loss components.

Return type dict[str, Tensor]

loss_single(*cls_score*, *bbox_pred*, *angle_cls*, *anchors*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *angle_targets*, *angle_weights*, *num_total_samples*)

Compute loss of a single scale level.

Parameters

- **cls_score** (*torch.Tensor*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W).
- **bbox_pred** (*torch.Tensor*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W).
- **anchors** (*torch.Tensor*) – Box reference for each scale level with shape (N, num_total_anchors, 5).
- **labels** (*torch.Tensor*) – Labels of each anchors with shape (N, num_total_anchors).
- **label_weights** (*torch.Tensor*) – Label weights of each anchor with shape (N, num_total_anchors)
- **bbox_targets** (*torch.Tensor*) – BBox regression targets of each anchor weight shape (N, num_total_anchors, 5).

- **bbox_weights** (*torch.Tensor*) – BBox regression loss weights of each anchor with shape (N, num_total_anchors, 5).
- **angle_targets** (*torch.Tensor*) – Angle classification targets of each anchor weight shape (N, num_total_anchors, coding_len).
- **angle_weights** (*torch.Tensor*) – Angle classification loss weights of each anchor with shape (N, num_total_anchors, 1).
- **num_total_samples** (*int*) – If sampling, num total samples equal to the number of total anchors; Otherwise, it is the number of positive anchors.

Returns

- loss_cls (torch.Tensor): cls. loss for each scale level.
- loss_bbox (torch.Tensor): reg. loss for each scale level.
- loss_angle (torch.Tensor): angle cls. loss for each scale level.

Return type tuple (torch.Tensor)

class mmrotate.models.dense_heads.**KFIoUODMRefineHead**(*num_classes*, *in_channels*, *stacked_convs=2*, *conv_cfg=None*, *norm_cfg=None*, *anchor_generator={'strides': [8, 16, 32, 64, 128], 'type': 'PseudoAnchorGenerator'}*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'odm_cls', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

Rotated Anchor-based refine head for KFIoU. It's a part of the Oriented Detection Module (ODM), which produces orientation-sensitive features for classification and orientation-invariant features for localization. The difference from *ODMRefineHead* is that its loss_bbox requires bbox_pred, bbox_targets, pred_decode and targets_decode as inputs.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **feat_channels** (*int*) – Number of hidden channels. Used in child classes.
- **anchor_generator** (*dict*) – Config dict for anchor generator
- **bbox_coder** (*dict*) – Config of bounding box coder.
- **reg_decoded_bbox** (*bool*) – If true, the regression loss would be applied on decoded bounding boxes. Default: False
- **background_label** (*int | None*) – Label ID of background, set as 0 for RPN and num_classes for other heads. It will automatically set as num_classes if None is given.
- **loss_cls** (*dict*) – Config of classification loss.
- **loss_bbox** (*dict*) – Config of localization loss.
- **train_cfg** (*dict*) – Training config of anchor head.
- **test_cfg** (*dict*) – Testing config of anchor head.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward_single(*x*)

Forward feature of a single scale level.

Parameters **x** (*torch.Tensor*) – Features of a single scale level.

Returns

- cls_score (torch.Tensor): Cls scores for a single scale level the channels number is num_anchors * num_classes.
- bbox_pred (torch.Tensor): Box energies / deltas for a single scale level, the channels number is num_anchors * 4.

Return type tuple (torch.Tensor)

get_anchors(*featmap_sizes*, *img_metas*, *device='cuda'*)
Get anchors according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **img_metas** (*list[dict]*) – Image meta info.
- **bboxes_as_anchors** (*list[list[Tensor]]*) – before further regression just like anchors.
- **device** (*torch.device | str*) – Device for returned tensors

Returns

- anchor_list (list[Tensor]): Anchors of each image
- valid_flag_list (list[Tensor]): Valid flags of each image

Return type tuple

get_bboxes(*cls_scores*, *bbox_preds*, *img_metas*, *cfg=None*, *rescale=False*, *rois=None*)
Transform network output for a batch into labeled boxes.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **img_metas** (*list[dict]*) – size / scale info for each image
- **cfg** (*mmcv.Config*) – test / postprocessing configuration
- **rescale** (*bool*) – if True, return boxes in original image space
- **rois** (*list[list[Tensor]]*) – input rbboxes of each level of each image. rois output by former stages and are to be refined.

Returns

each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 5 columns are bounding box positions (xc, yc, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the class index of the corresponding box.

Return type list[tuple[Tensor, Tensor]]

loss(*cls_scores*, *bbox_preds*, *gt_bboxes*, *gt_labels*, *img_metas*, *rois=None*, *gt_bboxes_ignore=None*)
Loss function of KFIoUODMRefineHead.

class mmrotate.models.dense_heads.**KFIoURRetinaHead**(*num_classes*, *in_channels*, *stacked_convs=4*, *conv_cfg=None*, *norm_cfg=None*, *anchor_generator={'octave_base_scale': 4, 'ratios': [0.5, 1.0, 2.0], 'scales_per_octave': 3, 'strides': [8, 16, 32, 64, 128], 'type': 'AnchorGenerator'}*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'retina_cls', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

Rotated Anchor-based head for KFIoU. The difference from *RRetinaHead* is that its loss_bbox requires bbox_pred, bbox_targets, pred_decode and targets_decode as inputs.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **stacked_convs** (*int, optional*) – Number of stacked convolutions.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.
- **anchor_generator** (*dict*) – Config dict for anchor generator
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

loss_single(*cls_score*, *bbox_pred*, *anchors*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *num_total_samples*)

Compute loss of a single scale level.

Parameters

- **cls_score** (*torch.Tensor*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W).
- **bbox_pred** (*torch.Tensor*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W).
- **anchors** (*torch.Tensor*) – Box reference for each scale level with shape (N, num_total_anchors, 5).
- **labels** (*torch.Tensor*) – Labels of each anchors with shape (N, num_total_anchors).
- **label_weights** (*torch.Tensor*) – Label weights of each anchor with shape (N, num_total_anchors)
- **bbox_targets** (*torch.Tensor*) – BBox regression targets of each anchor weight shape (N, num_total_anchors, 5).
- **bbox_weights** (*torch.Tensor*) – BBox regression loss weights of each anchor with shape (N, num_total_anchors, 5).
- **num_total_samples** (*int*) – If sampling, num total samples equal to the number of total anchors; Otherwise, it is the number of positive anchors.

Returns

- loss_cls (torch.Tensor): cls. loss for each scale level.
- loss_bbox (torch.Tensor): reg. loss for each scale level.

Return type tuple (torch.Tensor)

class mmrotate.models.dense_heads.**KFIoURRetinaRefineHead**(*num_classes*, *in_channels*, *stacked_convs=4*, *conv_cfg=None*, *norm_cfg=None*, *anchor_generator={'strides': [8, 16, 32, 64, 128], 'type': 'PseudoAnchorGenerator'}*, *bbox_coder={'target_means': (0.0, 0.0, 0.0, 0.0, 0.0), 'target_stds': (1.0, 1.0, 1.0, 1.0, 1.0), 'type': 'DeltaXYWHABBoxCoder'}*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'retina_cls', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

Rotational Anchor-based refine head. The difference from *RRetinaRefineHead* is that its loss_bbox requires bbox_pred, bbox_targets, pred_decode and targets_decode as inputs.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **stacked_convs** (*int, optional*) – Number of stacked convolutions.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.
- **anchor_generator** (*dict*) – Config dict for anchor generator
- **bbox_coder** (*dict*) – Config of bounding box coder.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

get_anchors(*featmap_sizes*, *img_metas*, *device='cuda'*)

Get anchors according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **img_metas** (*list[dict]*) – Image meta info.
- **bboxes_as_anchors** (*list[list[Tensor]]*) – before further regression just like anchors.
- **device** (*torch.device | str*) – Device for returned tensors

Returns

- anchor_list (list[Tensor]): Anchors of each image
- valid_flag_list (list[Tensor]): Valid flags of each image

Return type tuple (list[Tensor])

get_bboxes(*cls_scores*, *bbox_preds*, *img_metas*, *cfg=None*, *rescale=False*, *rois=None*)

Transform network output for a batch into labeled boxes.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)

- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **img_metas** (*list[dict]*) – size / scale info for each image
- **cfg** (*mmcv.Config*) – test / postprocessing configuration
- **rois** (*list[list[Tensor]]*) – input rbboxes of each level of each image. rois output by former stages and are to be refined
- **rescale** (*bool*) – if True, return boxes in original image space

Returns

each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 5 columns are bounding box positions (xc, yc, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the class index of the corresponding box.

Return type list[tuple[Tensor, Tensor]]

loss(*cls_scores*, *bbox_preds*, *gt_bboxes*, *gt_labels*, *img_metas*, *rois=None*, *gt_bboxes_ignore=None*)
Loss function of KFIoURRetinaRefineHead.

refine_bboxes(*cls_scores*, *bbox_preds*, *rois*)
Refine predicted bounding boxes at each position of the feature maps. This method will be used in R3Det in refinement stages.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, 5, H, W)
- **rois** (*list[list[Tensor]]*) – input rbboxes of each level of each image. rois output by former stages and are to be refined

Returns best or refined rbboxes of each level of each image.

Return type list[list[Tensor]]

class mmrotate.models.dense_heads.**ODMRefineHead**(*num_classes*, *in_channels*, *stacked_convs=2*, *conv_cfg=None*, *norm_cfg=None*, *anchor_generator={'strides': [8, 16, 32, 64, 128], 'type': 'PseudoAnchorGenerator'}*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'odm_cls', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)
Rotated Anchor-based refine head. It's a part of the Oriented Detection Module (ODM), which produces orientation-sensitive features for classification and orientation-invariant features for localization.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **stacked_convs** (*int, optional*) – Number of stacked convolutions.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.

- **anchor_generator** (*dict*) – Config dict for anchor generator
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward_single(*x*)
Forward feature of a single scale level.

Parameters **x** (*torch.Tensor*) – Features of a single scale level.

Returns

- cls_score (torch.Tensor): Cls scores for a single scale level the channels number is num_anchors * num_classes.
- bbox_pred (torch.Tensor): Box energies / deltas for a single scale level, the channels number is num_anchors * 4.

Return type tuple (torch.Tensor)

get_anchors(*featmap_sizes*, *img_metas*, *device='cuda'*)
Get anchors according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **img_metas** (*list[dict]*) – Image meta info.
- **bboxes_as_anchors** (*list[list[Tensor]]*) – before further regression just like anchors.
- **device** (*torch.device | str*) – Device for returned tensors

Returns

- anchor_list (list[Tensor]): Anchors of each image
- valid_flag_list (list[Tensor]): Valid flags of each image

Return type tuple (list[Tensor])

get_bboxes(*cls_scores*, *bbox_preds*, *img_metas*, *cfg=None*, *rescale=False*, *rois=None*)
Transform network output for a batch into labeled boxes.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **img_metas** (*list[dict]*) – size / scale info for each image
- **cfg** (*mmcv.Config*) – test / postprocessing configuration
- **rois** (*list[list[Tensor]]*) – input rbboxes of each level of
- **image. rois output by former stages and are to be refined** (*each*) –
- **rescale** (*bool*) – if True, return boxes in original image space

Returns

each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 5 columns are bounding box positions (xc, yc, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the class index of the corresponding box.

Return type list[tuple[Tensor, Tensor]]

loss(*cls_scores*, *bbox_preds*, *gt_bboxes*, *gt_labels*, *img_metas*, *rois=None*, *gt_bboxes_ignore=None*)
Loss function of ODMRefineHead.

class mmrotate.models.dense_heads.**OrientedRPNHead**(*in_channels*, *init_cfg={'layer': 'Conv2d', 'std': 0.01, 'type': 'Normal'}*, *version='oc'*, ***kwargs*)
Oriented RPN head for Oriented R-CNN.

loss_single(*cls_score*, *bbox_pred*, *anchors*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *num_total_samples*)
Compute loss of a single scale level.

Parameters

- **cls_score** (*torch.Tensor*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W).
- **bbox_pred** (*torch.Tensor*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W).
- **anchors** (*torch.Tensor*) – Box reference for each scale level with shape (N, num_total_anchors, 4).
- **labels** (*torch.Tensor*) – Labels of each anchors with shape (N, num_total_anchors).
- **label_weights** (*torch.Tensor*) – Label weights of each anchor with shape (N, num_total_anchors)
- **bbox_targets** (*torch.Tensor*) – BBox regression targets of each anchor
- **shape** (*weight*) –
- **bbox_weights** (*torch.Tensor*) – BBox regression loss weights of each anchor with shape (N, num_total_anchors, 4).
- **num_total_samples** (*int*) – If sampling, num total samples equal to the number of total anchors; Otherwise, it is the number of positive anchors.

Returns

- loss_cls (torch.Tensor): cls. loss for each scale level.
- loss_bbox (torch.Tensor): reg. loss for each scale level.

Return type tuple (torch.Tensor)

class mmrotate.models.dense_heads.**RotatedAnchorHead**(*num_classes*, *in_channels*, *feat_channels=256*, *anchor_generator={'octave_base_scale': 4, 'ratios': [1.0, 0.5, 2.0], 'scales_per_octave': 3, 'strides': [8, 16, 32, 64, 128], 'type': 'RotatedAnchorGenerator'}*, *bbox_coder={'target_means': (0.0, 0.0, 0.0, 0.0, 0.0), 'target_stds': (1.0, 1.0, 1.0, 1.0, 1.0), 'type': 'DeltaXYWHAOBBoxCoder'}*, *reg_decoded_bbox=False*, *assign_by_circumhbbox='oc'*, *loss_cls={'alpha': 0.25, 'gamma': 2.0, 'loss_weight': 1.0, 'type': 'FocalLoss', 'use_sigmoid': True}*, *loss_bbox={'loss_weight': 1.0, 'type': 'L1Loss'}*, *train_cfg=None*, *test_cfg=None*, *init_cfg={'layer': 'Conv2d', 'std': 0.01, 'type': 'Normal'}*)

Rotated Anchor-based head (RotatedRPN, RotatedRetinaNet, etc.).

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **feat_channels** (*int*) – Number of hidden channels. Used in child classes.
- **anchor_generator** (*dict*) – Config dict for anchor generator
- **bbox_coder** (*dict*) – Config of bounding box coder.
- **reg_decoded_bbox** (*bool*) – If true, the regression loss would be applied on decoded bounding boxes. Default: False
- **assign_by_circumhbbox** (*str*) – If None, assigner will assign according to the IoU between anchor and GT (OBB), called RetinaNet-OBB. If angle definition method, assigner will assign according to the IoU between anchor and GT's circumbox (HBB), called RetinaNet-HBB.
- **loss_cls** (*dict*) – Config of classification loss.
- **loss_bbox** (*dict*) – Config of localization loss.
- **train_cfg** (*dict*) – Training config of anchor head.
- **test_cfg** (*dict*) – Testing config of anchor head.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

aug_test(*feats*, *img_metas*, *rescale=False*)

Test det bboxes with test time augmentation, can be applied in DenseHead except for `RPNHead` and its variants, e.g., `GARPNHead`, etc.

Parameters

- **feats** (*list[Tensor]*) – the outer list indicates test-time augmentations and inner Tensor should have a shape NxCxHxW, which contains features for all images in the batch.
- **img_metas** (*list[list[dict]]*) – the outer list indicates test-time augs (multiscale, flip, etc.) and the inner list indicates images in a batch. each dict has image information.
- **rescale** (*bool, optional*) – Whether to rescale the results. Defaults to False.

Returns

Each item in result_list is 2-tuple. The first item is `bboxes` with shape (n, 6), where 6 represent (x, y, w, h, a, score). The shape of the second tensor in the tuple is `labels` with shape (n,). The length of list should always be 1.

Return type list[tuple[Tensor, Tensor]]

`forward`(*feats*)

Forward features from the upstream network.

Parameters **feats** (*tuple[Tensor]*) – Features from the upstream network, each is a 4D-tensor.

Returns

A tuple of classification scores and bbox prediction.

- cls_scores (list[Tensor]): Classification scores for all scale levels, each is a 4D-tensor, the channels number is num_anchors * num_classes.
- bbox_preds (list[Tensor]): Box energies / deltas for all scale levels, each is a 4D-tensor, the channels number is num_anchors * 5.

Return type tuple

`forward_single`(*x*)

Forward feature of a single scale level.

Parameters **x** (*torch.Tensor*) – Features of a single scale level.

Returns

- cls_score (torch.Tensor): Cls scores for a single scale level the channels number is num_anchors * num_classes.
- bbox_pred (torch.Tensor): Box energies / deltas for a single scale level, the channels number is num_anchors * 5.

Return type tuple (torch.Tensor)

`get_anchors`(*featmap_sizes*, *img_metas*, *device='cuda'*)

Get anchors according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **img_metas** (*list[dict]*) – Image meta info.
- **device** (*torch.device | str*) – Device for returned tensors

Returns

- anchor_list (list[Tensor]): Anchors of each image.
- valid_flag_list (list[Tensor]): Valid flags of each image.

Return type tuple (list[Tensor])

`get_bboxes`(*cls_scores*, *bbox_preds*, *img_metas*, *cfg=None*, *rescale=False*, *with_nms=True*)

Transform network output for a batch into bbox predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)

- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cfg** (*mmcv.Config | None*) – Test / postprocessing configuration, if None, test_cfg would be used
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Default: True.

Returns

Each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 5 columns are bounding box positions (cx, cy, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the predicted class label of the corresponding box.

Return type list[tuple[Tensor, Tensor]]

Example

```
>>> import mmcv
>>> self = AnchorHead(
>>>     num_classes=9,
>>>     in_channels=1,
>>>     anchor_generator=dict(
>>>         type='AnchorGenerator',
>>>         scales=[8],
>>>         ratios=[0.5, 1.0, 2.0],
>>>         strides=[4,]))
>>> img_metas = [{'img_shape': (32, 32, 3), 'scale_factor': 1}]
>>> cfg = mmcv.Config(dict(
>>>     score_thr=0.00,
>>>     nms=dict(type='nms', iou_thr=1.0),
>>>     max_per_img=10))
>>> feat = torch.rand(1, 1, 3, 3)
>>> cls_score, bbox_pred = self.forward_single(feat)
>>> # note the input lists are over different levels, not images
>>> cls_scores, bbox_preds = [cls_score], [bbox_pred]
>>> result_list = self.get_bboxes(cls_scores, bbox_preds,
>>>                               img_metas, cfg)
>>> det_bboxes, det_labels = result_list[0]
>>> assert len(result_list) == 1
>>> assert det_bboxes.shape[1] == 5
>>> assert len(det_bboxes) == len(det_labels) == cfg.max_per_img
```

get_targets(*anchor_list*, *valid_flag_list*, *gt_bboxes_list*, *img_metas*, *gt_bboxes_ignore_list=None*, *gt_labels_list=None*, *label_channels=1*, *unmap_outputs=True*, *return_sampling_results=False*)

Compute regression and classification targets for anchors in multiple images.

Parameters

- **anchor_list** (*list[list[Tensor]]*) – Multi level anchors of each image. The outer list indicates images, and the inner list corresponds to feature levels of the image. Each element of the inner list is a tensor of shape (num_anchors, 5).
- **valid_flag_list** (*list[list[Tensor]]*) – Multi level valid flags of each image. The outer list indicates images, and the inner list corresponds to feature levels of the image. Each element of the inner list is a tensor of shape (num_anchors,)
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **gt_bboxes_ignore_list** (*list[Tensor]*) – Ground truth bboxes to be ignored.
- **gt_labels_list** (*list[Tensor]*) – Ground truth labels of each box.
- **label_channels** (*int*) – Channel of label.
- **unmap_outputs** (*bool*) – Whether to map outputs back to the original set of anchors.

Returns

Usually returns a tuple containing learning targets.

- labels_list (list[Tensor]): Labels of each level.
- label_weights_list (list[Tensor]): Label weights of each level.
- bbox_targets_list (list[Tensor]): BBox targets of each level.
- bbox_weights_list (list[Tensor]): BBox weights of each level.
- num_total_pos (int): Number of positive samples in all images.
- num_total_neg (int): Number of negative samples in all images.

additional_returns: This function enables user-defined returns from *self._get_targets_single*. These returns are currently refined to properties at each feature map (i.e. having HxW dimension). The results will be concatenated after the end

Return type tuple

loss(*cls_scores*, *bbox_preds*, *gt_bboxes*, *gt_labels*, *img_metas*, *gt_bboxes_ignore=None*)

Compute losses of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 5) in [cx, cy, w, h, a] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss. Default: None

Returns A dictionary of loss components.

Return type dict[str, Tensor]

loss_single(*cls_score*, *bbox_pred*, *anchors*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *num_total_samples*)

Compute loss of a single scale level.

Parameters

- **cls_score** (*torch.Tensor*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W).
- **bbox_pred** (*torch.Tensor*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W).
- **anchors** (*torch.Tensor*) – Box reference for each scale level with shape (N, num_total_anchors, 5).
- **labels** (*torch.Tensor*) – Labels of each anchors with shape (N, num_total_anchors).
- **label_weights** (*torch.Tensor*) – Label weights of each anchor with shape (N, num_total_anchors)
- **bbox_targets** (*torch.Tensor*) – BBox regression targets of each anchor
- **shape** (*weight*) –
- **bbox_weights** (*torch.Tensor*) – BBox regression loss weights of each anchor with shape (N, num_total_anchors, 5).
- **num_total_samples** (*int*) – If sampling, num total samples equal to the number of total anchors; Otherwise, it is the number of positive anchors.

Returns

- loss_cls (torch.Tensor): cls. loss for each scale level.
- loss_bbox (torch.Tensor): reg. loss for each scale level.

Return type tuple (torch.Tensor)

merge_aug_bboxes(*aug_bboxes*, *aug_scores*, *img_metas*)

Merge augmented detection bboxes and scores.

Parameters

- **aug_bboxes** (*list[Tensor]*) – shape (n, 4*#class)
- **aug_scores** (*list[Tensor] or None*) – shape (n, #class)
- **img_shapes** (*list[Tensor]*) – shape (3,).

Returns `bboxes` with shape (n,4), where 4 represent (tl_x, tl_y, br_x, br_y) and `scores` with shape (n,).

Return type tuple[Tensor]

class mmrotate.models.dense_heads.**RotatedRPNHead**(*in_channels*, *init_cfg={'layer': 'Conv2d', 'std': 0.01, 'type': 'Normal'}*, *version='oc'*, ***kwargs*)

Rotated RPN head for rotated bboxes.

Parameters

- **in_channels** (*int*) – Number of channels in the input feature map.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward_single(*x*)
Forward feature map of a single scale level.

get_bboxes(*cls_scores*, *bbox_preds*, *img_metas*, *cfg=None*, *rescale=False*, *with_nms=True*)
Transform network output for a batch into bbox predictions.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **cfg** (*mmcv.Config | None*) – Test / postprocessing configuration, if None, test_cfg would be used
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Default: True.

Returns

Each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 5 columns are bounding box positions (cx, cy, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the predicted class label of the corresponding box.

Return type list[tuple[Tensor, Tensor]]

get_targets(*anchor_list*, *valid_flag_list*, *gt_bboxes_list*, *img_metas*, *gt_bboxes_ignore_list=None*, *gt_labels_list=None*, *label_channels=1*, *unmap_outputs=True*, *return_sampling_results=False*)
Compute regression and classification targets for anchors in multiple images.

Parameters

- **anchor_list** (*list[list[Tensor]]*) – Multi level anchors of each image. The outer list indicates images, and the inner list corresponds to feature levels of the image. Each element of the inner list is a tensor of shape (num_anchors, 4).
- **valid_flag_list** (*list[list[Tensor]]*) – Multi level valid flags of each image. The outer list indicates images, and the inner list corresponds to feature levels of the image. Each element of the inner list is a tensor of shape (num_anchors,)
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **gt_bboxes_ignore_list** (*list[Tensor]*) – Ground truth bboxes to be ignored.
- **gt_labels_list** (*list[Tensor]*) – Ground truth labels of each box.
- **label_channels** (*int*) – Channel of label.
- **unmap_outputs** (*bool*) – Whether to map outputs back to the original set of anchors.

Returns

Usually returns a tuple containing learning targets.

- labels_list (list[Tensor]): Labels of each level.

- label_weights_list (list[Tensor]): Label weights of each level.
- bbox_targets_list (list[Tensor]): BBox targets of each level.
- bbox_weights_list (list[Tensor]): BBox weights of each level.
- num_total_pos (int): Number of positive samples in all images.
- num_total_neg (int): Number of negative samples in all images.

additional_returns: This function enables user-defined returns from *self._get_targets_single*. These returns are currently refined to properties at each feature map (i.e. having HxW dimension). The results will be concatenated after the end

Return type tuple

loss(*cls_scores*, *bbox_preds*, *gt_bboxes*, *img_metas*, *gt_bboxes_ignore=None*)

Compute losses of the head.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 5) in [cx, cy, w, h, a] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **img_metas** (*list[dict]*) – Meta information of each image, e.g., image size, scaling factor, etc.
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss. Default: None

Returns A dictionary of loss components.

Return type dict[str, Tensor]

loss_single(*cls_score*, *bbox_pred*, *anchors*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *num_total_samples*)

Compute loss of a single scale level.

Parameters

- **cls_score** (*torch.Tensor*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W).
- **bbox_pred** (*torch.Tensor*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W).
- **anchors** (*torch.Tensor*) – Box reference for each scale level with shape (N, num_total_anchors, 4).
- **labels** (*torch.Tensor*) – Labels of each anchors with shape (N, num_total_anchors).
- **label_weights** (*torch.Tensor*) – Label weights of each anchor with shape (N, num_total_anchors)
- **bbox_targets** (*torch.Tensor*) – BBox regression targets of each anchor
- **shape** (*weight*) –

- **bbox_weights** (*torch.Tensor*) – BBox regression loss weights of each anchor with shape (N, num_total_anchors, 4).
- **num_total_samples** (*int*) – If sampling, num total samples equal to the number of total anchors; Otherwise, it is the number of positive anchors.

Returns A dictionary of loss components.

Return type dict[str, Tensor]

class mmrotate.models.dense_heads.**RotatedRepPointsHead**(*num_classes*, *in_channels*, *feat_channels*, *point_feat_channels=256*, *stacked_convs=3*, *num_points=9*, *gradient_mul=0.1*, *point_strides=[8, 16, 32, 64, 128]*, *point_base_scale=4*, *conv_bias='auto'*, *loss_cls={'alpha': 0.25, 'gamma': 2.0, 'loss_weight': 1.0, 'type': 'FocalLoss', 'use_sigmoid': True}*, *loss_bbox_init={'beta': 0.111111111111111, 'loss_weight': 0.5, 'type': 'SmoothL1Loss'}*, *loss_bbox_refine={'beta': 0.111111111111111, 'loss_weight': 1.0, 'type': 'SmoothL1Loss'}*, *conv_cfg=None*, *norm_cfg=None*, *train_cfg=None*, *test_cfg=None*, *center_init=True*, *transform_method='rotrect'*, *use_reassign=False*, *topk=6*, *anti_factor=0.75*, *version='oc'*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'reppoints_cls_out', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

Rotated RepPoints head.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int*) – Number of input channels.
- **feat_channels** (*int*) – Number of feature channels.
- **point_feat_channels** (*int, optional*) – Number of channels of points features.
- **stacked_convs** (*int, optional*) – Number of stacked convolutions.
- **num_points** (*int, optional*) – Number of points in points set.
- **gradient_mul** (*float, optional*) – The multiplier to gradients from points refinement and recognition.
- **point_strides** (*Iterable, optional*) – points strides.
- **point_base_scale** (*int, optional*) – Bbox scale for assigning labels.
- **conv_bias** (*str, optional*) – The bias of convolution.
- **loss_cls** (*dict, optional*) – Config of classification loss.
- **loss_bbox_init** (*dict, optional*) – Config of initial points loss.

- **loss_bbox_refine** (*dict, optional*) – Config of points loss in refinement.
- **conv_cfg** (*dict, optional*) – The config of convolution.
- **norm_cfg** (*dict, optional*) – The config of normlization.
- **train_cfg** (*dict, optional*) – The config of train.
- **test_cfg** (*dict, optional*) – The config of test.
- **center_init** (*bool, optional*) – Whether to use center point assignment.
- **transform_method** (*str, optional*) – The methods to transform RepPoints to bbox.
- **use_reassign** (*bool, optional*) – Whether to reassign samples.
- **topk** (*int, optional*) – Number of the highest topk points. Defaults to 9.
- **anti_factor** (*float, optional*) – Feature anti-aliasing coefficient.
- **version** (*str, optional*) – Angle representations. Defaults to 'oc'.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*feats*)
Forward function.

forward_single(*x*)
Forward feature map of a single FPN level.

get_bboxes(*cls_scores*, *pts_preds_init*, *pts_preds_refine*, *img_metas*, *cfg=None*, *rescale=False*, *with_nms=True*, ***kwargs*)
Transform network outputs of a batch into bbox results.

Parameters

- **cls_scores** (*list[Tensor]*) – Classification scores for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).
- **pts_preds_init** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 18D-tensor, has shape (batch_size, num_points * 2, H, W).
- **pts_preds_refine** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 18D-tensor, has shape (batch_size, num_points * 2, H, W).
- **img_metas** (*list[dict], Optional*) – Image meta info. Default None.
- **cfg** (*mmcv.Config, Optional*) – Test / postprocessing configuration, if None, test_cfg would be used. Default None.
- **rescale** (*bool*) – If True, return boxes in original image space. Default False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Default True.

Returns

Each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 4 columns are bounding box positions (cx, cy, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the predicted class label of the corresponding box.

Return type list[list[Tensor, Tensor]]

get_cfa_targets(*proposals_list*, *valid_flag_list*, *gt_bboxes_list*, *img_metas*, *gt_bboxes_ignore_list=None*, *gt_labels_list=None*, *stage='init'*, *label_channels=1*, *unmap_outputs=True*)
Compute corresponding GT box and classification targets for proposals.

Parameters

- **proposals_list** (*list[list]*) – Multi level points/bboxes of each image.
- **valid_flag_list** (*list[list]*) – Multi level valid flags of each image.
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **gt_bboxes_ignore_list** (*list[Tensor]*) – Ground truth bboxes to be ignored.
- **gt_bboxes_list** – Ground truth labels of each box.
- **stage** (*str*) – *init* or *refine*. Generate target for init stage or refine stage
- **label_channels** (*int*) – Channel of label.
- **unmap_outputs** (*bool*) – Whether to map outputs back to the original set of anchors.

Returns

- all_labels (list[Tensor]): Labels of each level.
- all_label_weights (list[Tensor]): Label weights of each level.
- all_bbox_gt (list[Tensor]): Ground truth bbox of each level.
- all_proposals (list[Tensor]): Proposals(points/bboxes) of each level.
- all_proposal_weights (list[Tensor]): Proposal weights of each level.
- pos_inds (list[Tensor]): Index of positive samples in all images.
- gt_inds (list[Tensor]): Index of ground truth bbox in all images.

Return type tuple

get_points(*featmap_sizes*, *img_metas*, *device*)

Get points according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **img_metas** (*list[dict]*) – Image meta info.

Returns points of each image, valid flags of each image

Return type tuple

get_pos_loss(*cls_score*, *pts_pred*, *label*, *bbox_gt*, *label_weight*, *convex_weight*, *pos_inds*)

Calculate loss of all potential positive samples obtained from first match process.

Parameters

- **cls_score** (*Tensor*) – Box scores of single image with shape (num_anchors, num_classes)
- **pts_pred** (*Tensor*) – Box energies / deltas of single image with shape (num_anchors, 4)
- **label** (*Tensor*) – classification target of each anchor with shape (num_anchors,)
- **bbox_gt** (*Tensor*) – Ground truth box.
- **label_weight** (*Tensor*) – Classification loss weight of each anchor with shape (num_anchors).
- **convex_weight** (*Tensor*) – Bbox weight of each anchor with shape (num_anchors, 4).

- **pos_inds** (*Tensor*) – Index of all positive samples got from first assign process.

Returns Losses of all positive samples in single image.

Return type Tensor

get_targets(*proposals_list*, *valid_flag_list*, *gt_bboxes_list*, *img_metas*, *gt_bboxes_ignore_list=None*, *gt_labels_list=None*, *stage='init'*, *label_channels=1*, *unmap_outputs=True*)

Compute corresponding GT box and classification targets for proposals.

Parameters

- **proposals_list** (*list[list]*) – Multi level points/bboxes of each image.
- **valid_flag_list** (*list[list]*) – Multi level valid flags of each image.
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **gt_bboxes_ignore_list** (*list[Tensor]*) – Ground truth bboxes to be ignored.
- **gt_bboxes_list** – Ground truth labels of each box.
- **stage** (*str*) – *init* or *refine*. Generate target for init stage or refine stage
- **label_channels** (*int*) – Channel of label.
- **unmap_outputs** (*bool*) – Whether to map outputs back to the original set of anchors.

Returns

- labels_list (list[Tensor]): Labels of each level.
- label_weights_list (list[Tensor]): Label weights of each level.
- bbox_gt_list (list[Tensor]): Ground truth bbox of each level.
- proposal_list (list[Tensor]): Proposals(points/bboxes) of each level.
- proposal_weights_list (list[Tensor]): Proposal weights of each level.
- num_total_pos (int): Number of positive samples in all images.
- num_total_neg (int): Number of negative samples in all images.

Return type tuple (list[Tensor])

loss(*cls_scores*, *pts_preds_init*, *pts_preds_refine*, *gt_bboxes*, *gt_labels*, *img_metas*, *gt_bboxes_ignore=None*)

Loss function of CFA head.

loss_single(*cls_score*, *pts_pred_init*, *pts_pred_refine*, *labels*, *label_weights*, *rbbox_gt_init*, *convex_weights_init*, *rbbox_gt_refine*, *convex_weights_refine*, *stride*, *num_total_samples_refine*)

Single loss function.

offset_to_pts(*center_list*, *pred_list*)

Change from point offset to point coordinate.

points2rotrect(*pts*, *y_first=True*)

Convert points to oriented bboxes.

reassign(*pos_losses*, *label*, *label_weight*, *pts_pred_init*, *convex_weight*, *gt_bbox*, *pos_inds*, *pos_gt_inds*, *num_proposals_each_level=None*, *num_level=None*)

CFA reassign process.

Parameters

- **pos_losses** (*Tensor*) – Losses of all positive samples in single image.
- **label** (*Tensor*) – classification target of each anchor with shape (num_anchors,)
- **label_weight** (*Tensor*) – Classification loss weight of each anchor with shape (num_anchors).
- **pts_pred_init** (*Tensor*) –
- **convex_weight** (*Tensor*) – Bbox weight of each anchor with shape (num_anchors, 4).
- **gt_bbox** (*Tensor*) – Ground truth box.
- **pos_inds** (*Tensor*) – Index of all positive samples got from first assign process.
- **pos_gt_inds** (*Tensor*) – Gt_index of all positive samples got from first assign process.
- **num_proposals_each_level** (*list, optional*) – Number of proposals of each level.
- **num_level** (*int, optional*) – Number of level.

Returns

Usually returns a tuple containing learning targets.

- label (Tensor): classification target of each anchor after paa assign, with shape (num_anchors,)
- label_weight (Tensor): Classification loss weight of each anchor after paa assign, with shape (num_anchors).
- convex_weight (Tensor): Bbox weight of each anchor with shape (num_anchors, 4).
- pos_normalize_term (list): pos normalize term for refine points losses.

Return type tuple

class mmrotate.models.dense_heads.**RotatedRetinaHead**(*num_classes*, *in_channels*, *stacked_convs=4*, *conv_cfg=None*, *norm_cfg=None*, *anchor_generator={'octave_base_scale': 4, 'ratios': [0.5, 1.0, 2.0], 'scales_per_octave': 3, 'strides': [8, 16, 32, 64, 128], 'type': 'AnchorGenerator'}*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'retina_cls', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

An anchor-based head used in RotatedRetinaNet.

The head contains two subnetworks. The first classifies anchor boxes and the second regresses deltas for the anchors.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **stacked_convs** (*int, optional*) – Number of stacked convolutions.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.
- **anchor_generator** (*dict*) – Config dict for anchor generator
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

filter_bboxes(*cls_scores*, *bbox_preds*)

Filter predicted bounding boxes at each position of the feature maps. Only one bounding boxes with highest score will be left at each position. This filter will be used in R3Det prior to the first feature refinement stage.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)

Returns best or refined rbboxes of each level of each image.

Return type list[list[Tensor]]

forward_single(*x*)

Forward feature of a single scale level.

Parameters **x** (*torch.Tensor*) – Features of a single scale level.

Returns

- cls_score (torch.Tensor): Cls scores for a single scale level the channels number is num_anchors * num_classes.
- bbox_pred (torch.Tensor): Box energies / deltas for a single scale level, the channels number is num_anchors * 4.

Return type tuple (torch.Tensor)

refine_bboxes(*cls_scores*, *bbox_preds*)

This function will be used in S2ANet, whose num_anchors=1.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, 5, H, W)

Returns refined rbboxes of each level of each image.

Return type list[list[Tensor]]

class mmrotate.models.dense_heads.**RotatedRetinaRefineHead**(*num_classes*, *in_channels*, *stacked_convs=4*, *conv_cfg=None*, *norm_cfg=None*, *anchor_generator={'strides': [8, 16, 32, 64, 128], 'type': 'PseudoAnchorGenerator'}*, *bbox_coder={'target_means': (0.0, 0.0, 0.0, 0.0), 'target_stds': (1.0, 1.0, 1.0, 1.0), 'type': 'DeltaXYWHABBoxCoder'}*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'retina_cls', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

Rotated Anchor-based refine head.

Parameters

- **num_classes** (*int*) – Number of categories excluding the background category.
- **in_channels** (*int*) – Number of channels in the input feature map.
- **stacked_convs** (*int, optional*) – Number of stacked convolutions.
- **conv_cfg** (*dict, optional*) – Config dict for convolution layer. Default: None.
- **norm_cfg** (*dict, optional*) – Config dict for normalization layer. Default: None.
- **anchor_generator** (*dict*) – Config dict for anchor generator
- **bbox_coder** (*dict*) – Config of bounding box coder.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

get_anchors(*featmap_sizes*, *img_metas*, *device='cuda'*)

Get anchors according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **img_metas** (*list[dict]*) – Image meta info.
- **bboxes_as_anchors** (*list[list[Tensor]]*) – before further regression just like anchors.
- **device** (*torch.device | str*) – Device for returned tensors

Returns

- anchor_list (list[Tensor]): Anchors of each image
- valid_flag_list (list[Tensor]): Valid flags of each image

Return type tuple (list[Tensor])

get_bboxes(*cls_scores*, *bbox_preds*, *img_metas*, *cfg=None*, *rescale=False*, *rois=None*)

Transform network output for a batch into labeled boxes.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_anchors * num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, num_anchors * 5, H, W)
- **img_metas** (*list[dict]*) – size / scale info for each image
- **cfg** (*mmcv.Config*) – test / postprocessing configuration
- **rois** (*list[list[Tensor]]*) – input rbboxes of each level of each image. rois output by former stages and are to be refined
- **rescale** (*bool*) – if True, return boxes in original image space

Returns

each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 5 columns are bounding box positions (xc, yc, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the class index of the corresponding box.

Return type list[tuple[Tensor, Tensor]]

loss(*cls_scores*, *bbox_preds*, *gt_bboxes*, *gt_labels*, *img_metas*, *rois=None*, *gt_bboxes_ignore=None*)
Loss function of RotatedRetinaRefineHead.

refine_bboxes(*cls_scores*, *bbox_preds*, *rois*)
Refine predicted bounding boxes at each position of the feature maps. This method will be used in R3Det in refinement stages.

Parameters

- **cls_scores** (*list[Tensor]*) – Box scores for each scale level Has shape (N, num_classes, H, W)
- **bbox_preds** (*list[Tensor]*) – Box energies / deltas for each scale level with shape (N, 5, H, W)
- **rois** (*list[list[Tensor]]*) – input rbboxes of each level of each image. rois output by former stages and are to be refined

Returns best or refined rbboxes of each level of each image.

Return type list[list[Tensor]]

class mmrotate.models.dense_heads.**SAMRepPointsHead**(*num_classes*, *in_channels*, *feat_channels*, *point_feat_channels=256*, *stacked_convs=3*, *num_points=9*, *gradient_mul=0.1*, *point_strides=[8, 16, 32, 64, 128]*, *point_base_scale=4*, *conv_bias='auto'*, *loss_cls={'alpha': 0.25, 'gamma': 2.0, 'loss_weight': 1.0, 'type': 'FocalLoss', 'use_sigmoid': True}*, *loss_bbox_init={'beta': 0.111111111111111, 'loss_weight': 0.5, 'type': 'SmoothL1Loss'}*, *loss_bbox_refine={'beta': 0.111111111111111, 'loss_weight': 1.0, 'type': 'SmoothL1Loss'}*, *conv_cfg=None*, *norm_cfg=None*, *train_cfg=None*, *test_cfg=None*, *center_init=True*, *transform_method='rotrect'*, *topk=6*, *anti_factor=0.75*, *version='oc'*, *init_cfg={'layer': 'Conv2d', 'override': {'bias_prob': 0.01, 'name': 'reppoints_cls_out', 'std': 0.01, 'type': 'Normal'}, 'std': 0.01, 'type': 'Normal'}*, ***kwargs*)

Rotated RepPoints head for SASM.

Parameters

- **num_classes** (*int*) – Number of classes.
- **in_channels** (*int*) – Number of input channels.
- **feat_channels** (*int*) – Number of feature channels.
- **point_feat_channels** (*int, optional*) – Number of channels of points features.
- **stacked_convs** (*int, optional*) – Number of stacked convolutions.
- **num_points** (*int, optional*) – Number of points in points set.
- **gradient_mul** (*float, optional*) – The multiplier to gradients from points refinement and recognition.
- **point_strides** (*Iterable, optional*) – points strides.
- **point_base_scale** (*int, optional*) – Bbox scale for assigning labels.

- **conv_bias** (*str, optional*) – The bias of convolution.
- **loss_cls** (*dict, optional*) – Config of classification loss.
- **loss_bbox_init** (*dict, optional*) – Config of initial points loss.
- **loss_bbox_refine** (*dict, optional*) – Config of points loss in refinement.
- **conv_cfg** (*dict, optional*) – The config of convolution.
- **norm_cfg** (*dict, optional*) – The config of normlization.
- **train_cfg** (*dict, optional*) – The config of train.
- **test_cfg** (*dict, optional*) – The config of test.
- **center_init** (*bool, optional*) – Whether to use center point assignment.
- **transform_method** (*str, optional*) – The methods to transform RepPoints to bbox.
- **topk** (*int, optional*) – Number of the highest topk points. Defaults to 9.
- **anti_factor** (*float, optional*) – Feature anti-aliasing coefficient.
- **version** (*str, optional*) – Angle representations. Defaults to 'oc'.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict.

forward(*feats*)
Forward function.

forward_single(*x*)
Forward feature map of a single FPN level.

get_bboxes(*cls_scores*, *pts_preds_init*, *pts_preds_refine*, *img_metas*, *cfg=None*, *rescale=False*, *with_nms=True*, ***kwargs*)
Transform network outputs of a batch into bbox results.

Parameters

- **cls_scores** (*list[Tensor]*) – Classification scores for all scale levels, each is a 4D-tensor, has shape (batch_size, num_priors * num_classes, H, W).
- **pts_preds_init** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 18D-tensor, has shape (batch_size, num_points * 2, H, W).
- **pts_preds_refine** (*list[Tensor]*) – Box energies / deltas for all scale levels, each is a 18D-tensor, has shape (batch_size, num_points * 2, H, W).
- **img_metas** (*list[dict], Optional*) – Image meta info. Default None.
- **cfg** (*mmcv.Config, Optional*) – Test / postprocessing configuration, if None, test_cfg would be used. Default None.
- **rescale** (*bool*) – If True, return boxes in original image space. Default False.
- **with_nms** (*bool*) – If True, do nms before return boxes. Default True.

Returns

Each item in result_list is 2-tuple. The first item is an (n, 6) tensor, where the first 4 columns are bounding box positions (cx, cy, w, h, a) and the 6-th column is a score between 0 and 1. The second item is a (n,) tensor where each item is the predicted class label of the corresponding box.

Return type list[list[Tensor, Tensor]]

get_points(*featmap_sizes*, *img_metas*, *device*)
Get points according to feature map sizes.

Parameters

- **featmap_sizes** (*list[tuple]*) – Multi-level feature map sizes.
- **img_metas** (*list[dict]*) – Image meta info.

Returns points of each image, valid flags of each image

Return type tuple

get_targets(*proposals_list*, *valid_flag_list*, *gt_bboxes_list*, *img_metas*, *gt_bboxes_ignore_list=None*, *gt_labels_list=None*, *stage='init'*, *label_channels=1*, *unmap_outputs=True*)
Compute corresponding GT box and classification targets for proposals.

Parameters

- **proposals_list** (*list[list]*) – Multi level points/bboxes of each image.
- **valid_flag_list** (*list[list]*) – Multi level valid flags of each image.
- **gt_bboxes_list** (*list[Tensor]*) – Ground truth bboxes of each image.
- **img_metas** (*list[dict]*) – Meta info of each image.
- **gt_bboxes_ignore_list** (*list[Tensor]*) – Ground truth bboxes to be ignored.
- **gt_bboxes_list** – Ground truth labels of each box.
- **stage** (*str*) – *init* or *refine*. Generate target for init stage or refine stage
- **label_channels** (*int*) – Channel of label.
- **unmap_outputs** (*bool*) – Whether to map outputs back to the original set of anchors.

Returns

- labels_list (list[Tensor]): Labels of each level.
- label_weights_list (list[Tensor]): Label weights of each level.
- bbox_gt_list (list[Tensor]): Ground truth bbox of each level.
- proposal_list (list[Tensor]): Proposals(points/bboxes) of each level.
- proposal_weights_list (list[Tensor]): Proposal weights of each level.
- num_total_pos (int): Number of positive samples in all images.
- num_total_neg (int): Number of negative samples in all images.

Return type tuple (list[Tensor])

loss(*cls_scores*, *pts_preds_init*, *pts_preds_refine*, *gt_bboxes*, *gt_labels*, *img_metas*, *gt_bboxes_ignore=None*)
Loss function of SAM RepPoints head.

loss_single(*cls_score*, *pts_pred_init*, *pts_pred_refine*, *labels*, *label_weights*, *rbbox_gt_init*, *convex_weights_init*, *sam_weights_init*, *rbbox_gt_refine*, *convex_weights_refine*, *sam_weights_refine*, *stride*, *num_total_samples_refine*)
Single loss function.

offset_to_pts(*center_list*, *pred_list*)
Change from point offset to point coordinate.

points2rotrect(*pts*, *y_first=True*)
Convert points to oriented bboxes.

20.5 roi_heads

class mmrotate.models.roi_heads.**GVRatioRoIHead**(*bbox_roi_extractor=None*, *bbox_head=None*, *shared_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*, *version='oc'*)

Gliding vertex roi head including one bbox head.

forward_dummy(*x*, *proposals*)

Dummy forward function.

Parameters

- **x** (*list[Tensors]*) – list of multi-level img features.
- **proposals** (*list[Tensors]*) – list of region proposals.

Returns list of region of interest.

Return type list[Tensors]

simple_test_bboxes(*x*, *img_metas*, *proposals*, *rcnn_test_cfg*, *rescale=False*)

Test only det bboxes without augmentation.

Parameters

- **x** (*tuple[Tensor]*) – Feature maps of all scale level.
- **img_metas** (*list[dict]*) – Image meta info.
- **proposals** (*List[Tensor]*) – Region proposals.
- **(obj** (*rcnn_test_cfg*) – *ConfigDict*): *test_cfg* of R-CNN.
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns The first list contains the boxes of the corresponding image in a batch, each tensor has the shape (num_boxes, 5) and last dimension 5 represent (cx, cy, w, h, a, score). Each Tensor in the second list is the labels with shape (num_boxes,). The length of both lists should be equal to batch_size.

Return type tuple[list[Tensor], list[Tensor]]

class mmrotate.models.roi_heads.**OrientedStandardRoIHead**(*bbox_roi_extractor=None*, *bbox_head=None*, *shared_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*, *version='oc'*)

Oriented RCNN roi head including one bbox head.

forward_train(*x*, *img_metas*, *proposal_list*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*, *gt_masks=None*)

Parameters

- **x** (*list[Tensor]*) – list of multi-level img features.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **proposals** (*list[Tensors]*) – list of region proposals.

- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 5) in [cx, cy, w, h, a] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task. Always set to None.

Returns a dictionary of loss components

Return type dict[str, Tensor]

simple_test_bboxes(*x*, *img_metas*, *proposals*, *rcnn_test_cfg*, *rescale=False*)
Test only det bboxes without augmentation.

Parameters

- **x** (*tuple[Tensor]*) – Feature maps of all scale level.
- **img_metas** (*list[dict]*) – Image meta info.
- **proposals** (*List[Tensor]*) – Region proposals.
- **(obj** (*rcnn_test_cfg*) – *ConfigDict*): *test_cfg* of R-CNN.
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns The first list contains the boxes of the corresponding image in a batch, each tensor has the shape (num_boxes, 5) and last dimension 5 represent (cx, cy, w, h, a, score). Each Tensor in the second list is the labels with shape (num_boxes,). The length of both lists should be equal to batch_size.

Return type tuple[list[Tensor], list[Tensor]]

class mmrotate.models.roi_heads.**RoITransRoIHead**(*num_stages*, *stage_loss_weights*, *bbox_roi_extractor=None*, *bbox_head=None*, *shared_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *version='oc'*, *init_cfg=None*)
RoI Trans cascade roi head including one bbox head.

Parameters

- **num_stages** (*int*) – number of cascade stages.
- **stage_loss_weights** (*list[float]*) – loss weights of cascade stages.
- **bbox_roi_extractor** (*dict, optional*) – Config of bbox_roi_extractor.
- **bbox_head** (*dict, optional*) – Config of bbox_head.
- **shared_head** (*dict, optional*) – Config of shared_head.
- **train_cfg** (*dict, optional*) – Config of train.
- **test_cfg** (*dict, optional*) – Config of test.
- **pretrained** (*str, optional*) – Path of pretrained weight.
- **version** (*str, optional*) – Angle representations. Defaults to 'oc'.
- **init_cfg** (*dict, optional*) – Config of initialization.

aug_test(*features*, *proposal_list*, *img_metas*, *rescale=False*)
Test with augmentations.

forward_dummy(*x*, *proposals*)
Dummy forward function.

Parameters

- **x** (*list[Tensors]*) – list of multi-level img features.
- **proposals** (*list[Tensors]*) – list of region proposals.

Returns list of region of interest.

Return type list[Tensors]

forward_train(*x*, *img_metas*, *proposal_list*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*, *gt_masks=None*)

Parameters

- **x** (*list[Tensor]*) – list of multi-level img features.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **proposals** (*list[Tensors]*) – list of region proposals.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 5) in [cx, cy, w, h, a] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task. Always set to None.

Returns a dictionary of loss components

Return type dict[str, Tensor]

init_assigner_sampler()
Initialize assigner and sampler for each stage.

init_bbox_head(*bbox_roi_extractor*, *bbox_head*)
Initialize box head and box roi extractor.

Parameters

- **bbox_roi_extractor** (*dict*) – Config of box roi extractor.
- **bbox_head** (*dict*) – Config of box in box head.

simple_test(*x*, *proposal_list*, *img_metas*, *rescale=False*)
Test without augmentation.

Parameters

- **x** (*list[Tensor]*) – list of multi-level img features.
- **proposal_list** (*list[Tensors]*) – list of region proposals.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.

- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns a dictionary of bbox_results.

Return type dict[str, Tensor]

class mmrotate.models.roi_heads.**RotatedBBoxHead**(*with_avg_pool=False*, *with_cls=True*, *with_reg=True*, *roi_feat_size=7*, *in_channels=256*, *num_classes=80*, *bbox_coder={'clip_border': True, 'target_means': [0.0, 0.0, 0.0, 0.0], 'target_stds': [0.1, 0.1, 0.2, 0.2], 'type': 'DeltaXYWHBBoxCoder'}*, *reg_class_agnostic=False*, *reg_decoded_bbox=False*, *reg_predictor_cfg={'type': 'Linear'}*, *cls_predictor_cfg={'type': 'Linear'}*, *loss_cls={'loss_weight': 1.0, 'type': 'CrossEntropyLoss', 'use_sigmoid': False}*, *loss_bbox={'beta': 1.0, 'loss_weight': 1.0, 'type': 'SmoothL1Loss'}*, *init_cfg=None*)

Simplest RoI head, with only two fc layers for classification and regression respectively.

Parameters

- **with_avg_pool** (*bool, optional*) – If True, use avg_pool.
- **with_cls** (*bool, optional*) – If True, use classification branch.
- **with_reg** (*bool, optional*) – If True, use regression branch.
- **roi_feat_size** (*int, optional*) – Size of RoI features.
- **in_channels** (*int, optional*) – Input channels.
- **num_classes** (*int, optional*) – Number of classes.
- **bbox_coder** (*dict, optional*) – Config of bbox coder.
- **reg_class_agnostic** (*bool, optional*) – If True, regression branch are class agnostic.
- **reg_decoded_bbox** (*bool, optional*) – If True, regression branch use decoded bbox to compute loss.
- **reg_predictor_cfg** (*dict, optional*) – Config of regression predictor.
- **cls_predictor_cfg** (*dict, optional*) – Config of classification predictor.
- **loss_cls** (*dict, optional*) – Config of classification loss.
- **loss_bbox** (*dict, optional*) – Config of regression loss.
- **init_cfg** (*dict, optional*) – Config of initialization.

property custom_accuracy
The custom accuracy.

property custom_activation
The custom activation.

property custom_cls_channels
The custom cls channels.

forward(*x*)
Forward function of Rotated BBoxHead.

get_bboxes(*rois*, *cls_score*, *bbox_pred*, *img_shape*, *scale_factor*, *rescale=False*, *cfg=None*)
Transform network output for a batch into bbox predictions.

Parameters

- **rois** (*torch.Tensor*) – Boxes to be transformed. Has shape (num_boxes, 5). last dimension 5 arrange as (batch_index, x1, y1, x2, y2).
- **cls_score** (*torch.Tensor*) – Box scores, has shape (num_boxes, num_classes + 1).
- **bbox_pred** (*Tensor, optional*) – Box energies / deltas. has shape (num_boxes, num_classes * 5).
- **img_shape** (*Sequence[int], optional*) – Maximum bounds for boxes, specifies (H, W, C) or (H, W).
- **scale_factor** (*ndarray*) – Scale factor of the image arrange as (w_scale, h_scale, w_scale, h_scale).
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.
- **(obj** (*cfg*) – *ConfigDict*): *test_cfg* of Bbox Head. Default: None

Returns First tensor is *det_bboxes*, has the shape (num_boxes, 6) and last dimension 6 represent (cx, cy, w, h, a, score). Second tensor is the labels with shape (num_boxes,).

Return type tuple[Tensor, Tensor]

get_targets(*sampling_results*, *gt_bboxes*, *gt_labels*, *rcnn_train_cfg*, *concat=True*)

Calculate the ground truth for all samples in a batch according to the sampling_results.

Almost the same as the implementation in bbox_head, we passed additional parameters pos_inds_list and neg_inds_list to *_get_target_single* function.

Parameters

- **(List[obj** (*sampling_results*) – SamplingResults]): Assign results of all images in a batch after sampling.
- **gt_bboxes** (*list[Tensor]*) – Gt_bboxes of all images in a batch, each tensor has shape (num_gt, 5), the last dimension 5 represents [cx, cy, w, h, a].
- **gt_labels** (*list[Tensor]*) – Gt_labels of all images in a batch, each tensor has shape (num_gt,).
- **(obj** (*rcnn_train_cfg*) – ConfigDict): *train_cfg* of RCNN.
- **concat** (*bool*) – Whether to concatenate the results of all the images in a single batch.

Returns

Ground truth for proposals in a single image. Containing the following list of Tensors:

- labels (list[Tensor],Tensor): Gt_labels for all proposals in a batch, each tensor in list has shape (num_proposals,) when *concat=False*, otherwise just a single tensor has shape (num_all_proposals,).
- label_weights (list[Tensor]): Labels_weights for all proposals in a batch, each tensor in list has shape (num_proposals,) when *concat=False*, otherwise just a single tensor has shape (num_all_proposals,).
- bbox_targets (list[Tensor],Tensor): Regression target for all proposals in a batch, each tensor in list has shape (num_proposals, 5) when *concat=False*, otherwise just a single tensor has shape (num_all_proposals, 5), the last dimension 4 represents [cx, cy, w, h, a].
- bbox_weights (list[tensor],Tensor): Regression weights for all proposals in a batch, each tensor in list has shape (num_proposals, 5) when *concat=False*, otherwise just a single tensor has shape (num_all_proposals, 5).

Return type Tuple[Tensor]

loss(*cls_score*, *bbox_pred*, *rois*, *labels*, *label_weights*, *bbox_targets*, *bbox_weights*, *reduction_override=None*)
Loss function.

Parameters

- **cls_score** (*torch.Tensor*) – Box scores, has shape (num_boxes, num_classes + 1).
- **bbox_pred** (*Tensor, optional*) – Box energies / deltas. has shape (num_boxes, num_classes * 5).
- **rois** (*torch.Tensor*) – Boxes to be transformed. Has shape (num_boxes, 5). last dimension 5 arrange as (batch_index, x1, y1, x2, y2).
- **labels** (*torch.Tensor*) – Shape (n*bs,).
- **label_weights** (*torch.Tensor*) – Labels_weights for all proposals, has shape (num_proposals,).
- **bbox_targets** (*torch.Tensor*) – Regression target for all proposals, has shape (num_proposals, 5), the last dimension 5 represents [cx, cy, w, h, a].
- **bbox_weights** (*list[tensor],Tensor*) – Regression weights for all proposals in a batch, each tensor in list has shape (num_proposals, 5) when *concat=False*, otherwise just a single tensor has shape (num_all_proposals, 5).
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

refine_bboxes(*rois*, *labels*, *bbox_preds*, *pos_is_gts*, *img_metas*)
Refine bboxes during training.

Parameters

- **rois** (*torch.Tensor*) – Shape (n*bs, 5), where n is image number per GPU, and bs is the sampled RoIs per image. The first column is the image id and the next 4 columns are x1, y1, x2, y2.
- **labels** (*torch.Tensor*) – Shape (n*bs,).
- **bbox_preds** (*torch.Tensor*) – Shape (n*bs, 5) or (n*bs, 5*#class).
- **pos_is_gts** (*list[Tensor]*) – Flags indicating if each positive bbox is a gt bbox.
- **img_metas** (*list[dict]*) – Meta info of each image.

Returns Refined bboxes of each image in a mini-batch.

Return type list[Tensor]

regress_by_class(*rois*, *label*, *bbox_pred*, *img_meta*)
Regress the bbox for the predicted class. Used in Cascade R-CNN.

Parameters

- **rois** (*torch.Tensor*) – shape (n, 4) or (n, 5)
- **label** (*torch.Tensor*) – shape (n,)
- **bbox_pred** (*torch.Tensor*) – shape (n, 5*(#class)) or (n, 5)
- **img_meta** (*dict*) – Image meta info.

Returns Regressed bboxes, the same shape as input rois.

Return type Tensor

class mmrotate.models.roi_heads.**RotatedConvFCBBoxHead**(*num_shared_convs=0*, *num_shared_fcs=0*, *num_cls_convs=0*, *num_cls_fcs=0*, *num_reg_convs=0*, *num_reg_fcs=0*, *conv_out_channels=256*, *fc_out_channels=1024*, *conv_cfg=None*, *norm_cfg=None*, *init_cfg=None*, **args*, ***kwargs*)

More general bbox head, with shared conv and fc layers and two optional separated branches.

```
                            /-> cls convs -> cls fcs -> cls
shared convs -> shared fcs
                            \-> reg convs -> reg fcs -> reg
```

Parameters

- **num_shared_convs** (*int, optional*) – number of shared_convs.
- **num_shared_fcs** (*int, optional*) – number of shared_fcs.
- **num_cls_convs** (*int, optional*) – number of cls_convs.
- **num_cls_fcs** (*int, optional*) – number of cls_fcs.
- **num_reg_convs** (*int, optional*) – number of reg_convs.
- **num_reg_fcs** (*int, optional*) – number of reg_fcs.
- **conv_out_channels** (*int, optional*) – output channels of convolution.
- **fc_out_channels** (*int, optional*) – output channels of fc.
- **conv_cfg** (*dict, optional*) – Config of convolution.
- **norm_cfg** (*dict, optional*) – Config of normalization.
- **init_cfg** (*dict, optional*) – Config of initialization.

forward(*x*)

Forward function.

class mmrotate.models.roi_heads.**RotatedShared2FCBBoxHead**(*fc_out_channels=1024*, **args*, ***kwargs*)

Shared2FC RBBox head.

class mmrotate.models.roi_heads.**RotatedSingleRoIExtractor**(*roi_layer*, *out_channels*, *featmap_strides*, *finest_scale=56*, *init_cfg=None*)

Extract RoI features from a single level feature map.

If there are multiple input feature levels, each RoI is mapped to a level according to its scale. The mapping rule is proposed in FPN.

Parameters

- **roi_layer** (*dict*) – Specify RoI layer type and arguments.
- **out_channels** (*int*) – Output channels of RoI layers.
- **featmap_strides** (*List[int]*) – Strides of input feature maps.
- **finest_scale** (*int*) – Scale threshold of mapping to level 0. Default: 56.
- **init_cfg** (*dict or list[dict], optional*) – Initialization config dict. Default: None

build_roi_layers(*layer_cfg*, *featmap_strides*)
Build RoI operator to extract feature from each level feature map.

Parameters

- **layer_cfg** (*dict*) – Dictionary to construct and config RoI layer operation. Options are modules under mmcv/ops such as RoIAlign.
- **featmap_strides** (*List[int]*) – The stride of input feature map w.r.t to the original image size, which would be used to scale RoI coordinate (original image coordinate system) to feature coordinate system.

Returns The RoI extractor modules for each level feature map.

Return type nn.ModuleList

forward(*feats*, *rois*, *roi_scale_factor=None*)
Forward function.

Parameters

- **feats** (*torch.Tensor*) – Input features.
- **rois** (*torch.Tensor*) – Input RoIs, shape (k, 5).
- **scale_factor** (*float*) – Scale factor that RoI will be multiplied by.

Returns Scaled RoI features.

Return type torch.Tensor

map_roi_levels(*rois*, *num_levels*)
Map rois to corresponding feature levels by scales.

- scale < finest_scale * 2: level 0
- finest_scale * 2 <= scale < finest_scale * 4: level 1
- finest_scale * 4 <= scale < finest_scale * 8: level 2
- scale >= finest_scale * 8: level 3

Parameters

- **rois** (*torch.Tensor*) – Input RoIs, shape (k, 5).
- **num_levels** (*int*) – Total level number.

Returns Level index (0-based) of each RoI, shape (k,)

Return type Tensor

roi_rescale(*rois*, *scale_factor*)
Scale RoI coordinates by scale factor.

Parameters

- **rois** (*torch.Tensor*) – RoI (Region of Interest), shape (n, 6)
- **scale_factor** (*float*) – Scale factor that RoI will be multiplied by.

Returns Scaled RoI.

Return type torch.Tensor

class mmrotate.models.roi_heads.**RotatedStandardRoIHead**(*bbox_roi_extractor=None*, *bbox_head=None*, *shared_head=None*, *train_cfg=None*, *test_cfg=None*, *pretrained=None*, *init_cfg=None*, *version='oc'*)

Simplest base rotated roi head including one bbox head.

Parameters

- **bbox_roi_extractor** (*dict, optional*) – Config of bbox_roi_extractor.
- **bbox_head** (*dict, optional*) – Config of bbox_head.
- **shared_head** (*dict, optional*) – Config of shared_head.
- **train_cfg** (*dict, optional*) – Config of train.
- **test_cfg** (*dict, optional*) – Config of test.
- **pretrained** (*str, optional*) – Path of pretrained weight.
- **init_cfg** (*dict, optional*) – Config of initialization.
- **version** (*str, optional*) – Angle representations. Defaults to 'oc'.

async **async_simple_test**(*x*, *proposal_list*, *img_metas*, *rescale=False*)

Async test without augmentation.

Parameters

- **x** (*list[Tensor]*) – list of multi-level img features.
- **proposal_list** (*list[Tensors]*) – list of region proposals.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns a dictionary of bbox_results.

Return type dict[str, Tensor]

aug_test(*x*, *proposal_list*, *img_metas*, *rescale=False*)

Test with augmentations.

forward_dummy(*x*, *proposals*)

Dummy forward function.

Parameters

- **x** (*list[Tensors]*) – list of multi-level img features.
- **proposals** (*list[Tensors]*) – list of region proposals.

Returns list of region of interest.

Return type list[Tensors]

forward_train(*x*, *img_metas*, *proposal_list*, *gt_bboxes*, *gt_labels*, *gt_bboxes_ignore=None*, *gt_masks=None*)

Parameters

- **x** (*list[Tensor]*) – list of multi-level img features.

- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'. For details on the values of these keys see *mmdet/datasets/pipelines/formatting.py:Collect*.
- **proposals** (*list[Tensors]*) – list of region proposals.
- **gt_bboxes** (*list[Tensor]*) – Ground truth bboxes for each image with shape (num_gts, 5) in [cx, cy, w, h, a] format.
- **gt_labels** (*list[Tensor]*) – class indices corresponding to each box
- **gt_bboxes_ignore** (*None | list[Tensor]*) – specify which bounding boxes can be ignored when computing the loss.
- **gt_masks** (*None | Tensor*) – true segmentation masks for each box used if the architecture supports a segmentation task. Always set to None.

Returns a dictionary of loss components.

Return type dict[str, Tensor]

init_assigner_sampler()
Initialize assigner and sampler.

init_bbox_head(*bbox_roi_extractor*, *bbox_head*)
Initialize bbox_head.

Parameters

- **bbox_roi_extractor** (*dict*) – Config of bbox_roi_extractor.
- **bbox_head** (*dict*) – Config of bbox_head.

simple_test(*x*, *proposal_list*, *img_metas*, *rescale=False*)
Test without augmentation.

Parameters

- **x** (*list[Tensor]*) – list of multi-level img features.
- **proposal_list** (*list[Tensors]*) – list of region proposals.
- **img_metas** (*list[dict]*) – list of image info dict where each dict has: 'img_shape', 'scale_factor', 'flip', and may also contain 'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns a dictionary of bbox_results.

Return type dict[str, Tensor]

simple_test_bboxes(*x*, *img_metas*, *proposals*, *rcnn_test_cfg*, *rescale=False*)
Test only det bboxes without augmentation.

Parameters

- **x** (*tuple[Tensor]*) – Feature maps of all scale level.
- **img_metas** (*list[dict]*) – Image meta info.
- **proposals** (*List[Tensor]*) – Region proposals.
- **(obj** (*rcnn_test_cfg*) – *ConfigDict*): *test_cfg* of R-CNN.
- **rescale** (*bool*) – If True, return boxes in original image space. Default: False.

Returns The first list contains the boxes of the corresponding image in a batch, each tensor has the shape (num_boxes, 5) and last dimension 5 represent (tl_x, tl_y, br_x, br_y, score). Each Tensor in the second list is the labels with shape (num_boxes,). The length of both lists should be equal to batch_size.

Return type tuple[list[Tensor], list[Tensor]]

20.6 losses

class mmrotate.models.losses.**BCConvexGIoULoss**(*reduction='mean'*, *loss_weight=1.0*)
BCConvex GIoU loss.

Computing the BCConvex GIoU loss between a set of predicted convexes and target convexes.

Parameters

- **reduction** (*str, optional*) – The reduction method of the loss. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

Returns Loss tensor.

Return type torch.Tensor

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*, ***kwargs*)
Forward function.

Parameters

- **pred** (*torch.Tensor*) – Predicted convexes.
- **target** (*torch.Tensor*) – Corresponding gt convexes.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

class mmrotate.models.losses.**ConvexGIoULoss**(*reduction='mean'*, *loss_weight=1.0*)
Convex GIoU loss.

Computing the Convex GIoU loss between a set of predicted convexes and target convexes.

Parameters

- **reduction** (*str, optional*) – The reduction method of the loss. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

Returns Loss tensor.

Return type torch.Tensor

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*, ***kwargs*)
Forward function.

Parameters

- **pred** (*torch.Tensor*) – Predicted convexes.
- **target** (*torch.Tensor*) – Corresponding gt convexes.

- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

class mmrotate.models.losses.**GDLoss**(*loss_type*, *representation='xy_wh_r'*, *fun='log1p'*, *tau=0.0*, *alpha=1.0*, *reduction='mean'*, *loss_weight=1.0*, ***kwargs*)

Gaussian based loss.

Parameters

- **loss_type** (*str*) – Type of loss.
- **representation** (*str, optional*) – Coordinate System.
- **fun** (*str, optional*) – The function applied to distance. Defaults to 'log1p'.
- **tau** (*float, optional*) – Defaults to 1.0.
- **alpha** (*float, optional*) – Defaults to 1.0.
- **reduction** (*str, optional*) – The reduction method of the loss. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

Returns loss (torch.Tensor)

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*, ***kwargs*)

Forward function.

Parameters

- **pred** (*torch.Tensor*) – Predicted convexes.
- **target** (*torch.Tensor*) – Corresponding gt convexes.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

class mmrotate.models.losses.**GDLoss_v1**(*loss_type*, *fun='sqrt'*, *tau=1.0*, *reduction='mean'*, *loss_weight=1.0*, ***kwargs*)

Gaussian based loss.

Parameters

- **loss_type** (*str*) – Type of loss.
- **fun** (*str, optional*) – The function applied to distance. Defaults to 'log1p'.
- **tau** (*float, optional*) – Defaults to 1.0.
- **reduction** (*str, optional*) – The reduction method of the loss. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

Returns loss (torch.Tensor)

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*, ***kwargs*)
Forward function.

Parameters

- **pred** (*torch.Tensor*) – Predicted convexes.
- **target** (*torch.Tensor*) – Corresponding gt convexes.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

class mmrotate.models.losses.**KFLoss**(*fun='none'*, *reduction='mean'*, *loss_weight=1.0*, ***kwargs*)
Kalman filter based loss.

Parameters

- **fun** (*str, optional*) – The function applied to distance. Defaults to 'log1p'.
- **reduction** (*str, optional*) – The reduction method of the loss. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

Returns loss (torch.Tensor)

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *pred_decode=None*, *targets_decode=None*, *reduction_override=None*, ***kwargs*)
Forward function.

Parameters

- **pred** (*torch.Tensor*) – Predicted convexes.
- **target** (*torch.Tensor*) – Corresponding gt convexes.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **pred_decode** (*torch.Tensor*) – Predicted decode bboxes.
- **targets_decode** (*torch.Tensor*) – Corresponding gt decode bboxes.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

Returns loss (torch.Tensor)

class mmrotate.models.losses.**KLDRepPointsLoss**(*eps=1e-06*, *reduction='mean'*, *loss_weight=1.0*)
Kullback-Leibler Divergence loss for RepPoints.

Parameters

- **eps** (*float*) – Defaults to 1e-6.
- **reduction** (*str, optional*) – The reduction method of the loss. Defaults to 'mean'.
- **loss_weight** (*float, optional*) – The weight of loss. Defaults to 1.0.

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*, ***kwargs*)
Forward function.

Parameters

- **pred** (*torch.Tensor*) – Predicted convexes.
- **target** (*torch.Tensor*) – Corresponding gt convexes.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Defaults to None.

Returns loss (torch.Tensor)

class mmrotate.models.losses.**SmoothFocalLoss**(*gamma=2.0*, *alpha=0.25*, *reduction='mean'*, *loss_weight=1.0*)
Smooth Focal Loss. Implementation of Circular Smooth Label (CSL).

Parameters

- **gamma** (*float, optional*) – The gamma for calculating the modulating factor. Defaults to 2.0.
- **alpha** (*float, optional*) – A balanced form for Focal Loss. Defaults to 0.25.
- **reduction** (*str, optional*) – The method used to reduce the loss into a scalar. Defaults to 'mean'. Options are "none", "mean" and "sum".
- **loss_weight** (*float, optional*) – Weight of loss. Defaults to 1.0.

Returns loss (torch.Tensor)

forward(*pred*, *target*, *weight=None*, *avg_factor=None*, *reduction_override=None*)
Forward function.

Parameters

- **pred** (*torch.Tensor*) – The prediction.
- **target** (*torch.Tensor*) – The learning label of the prediction.
- **weight** (*torch.Tensor, optional*) – The weight of loss for each prediction. Defaults to None.
- **avg_factor** (*int, optional*) – Average factor that is used to average the loss. Defaults to None.
- **reduction_override** (*str, optional*) – The reduction method used to override the original reduction method of the loss. Options are "none", "mean" and "sum".

Returns The calculated loss

Return type torch.Tensor

20.7 utils

class mmrotate.models.utils.**ORConv2d**(*in_channels*, *out_channels*, *kernel_size=3*, *arf_config=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*)

Oriented 2-D convolution.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale).
- **kernel_size** (*int, optional*) – The size of kernel.
- **arf_config** (*tuple, optional*) – a tuple consist of nOrientation and nRotation.
- **stride** (*int, optional*) – Stride of the convolution. Default: 1.
- **padding** (*int or tuple*) – Zero-padding added to both sides of the input. Default: 0.
- **dilation** (*int or tuple*) – Spacing between kernel elements. Default: 1.
- **groups** (*int*) – Number of blocked connections from input. channels to output channels. Default: 1.
- **bias** (*bool*) – If True, adds a learnable bias to the output. Default: False.

forward(*input*)
Forward function.

get_indices()
Get the indices of ORConv2d.

reset_parameters()
Reset the parameters of ORConv2d.

rotate_arf()
Build active rotating filter module.

class mmrotate.models.utils.**RotationInvariantPooling**(*nInputPlane*, *nOrientation=8*)

Rotating invariant pooling module.

Parameters

- **nInputPlane** (*int*) – The number of Input plane.
- **nOrientation** (*int, optional*) – The number of oriented channels.

forward(*x*)
Forward function.

mmrotate.models.utils.**build_enn_divide_feature**(*planes*)
build a enn regular feature map with the specified number of channels divided by N.

mmrotate.models.utils.**build_enn_feature**(*planes*)
build a enn regular feature map with the specified number of channels.

mmrotate.models.utils.**build_enn_norm_layer**(*num_features*, *postfix=''*)
build an enn normalizion layer.

mmrotate.models.utils.**build_enn_trivial_feature**(*planes*)
build a enn trivial feature map with the specified number of channels.

mmrotate.models.utils.**ennAvgPool**(*inplanes*, *kernel_size=1*, *stride=None*, *padding=0*, *ceil_mode=False*)
enn Average Pooling.

Parameters

- **inplanes** (*int*) – The number of input channel.
- **kernel_size** (*int, optional*) – The size of kernel.
- **stride** (*int, optional*) – Stride of the convolution. Default: 1.
- **padding** (*int or tuple*) – Zero-padding added to both sides of the input. Default: 0.
- **ceil_mode** (*bool, optional*) – if True, keep information in the corner of feature map.

mmrotate.models.utils.**ennConv**(*inplanes*, *outplanes*, *kernel_size=3*, *stride=1*, *padding=0*, *groups=1*, *bias=False*, *dilation=1*)

enn convolution.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale).
- **kernel_size** (*int, optional*) – The size of kernel.
- **stride** (*int, optional*) – Stride of the convolution. Default: 1.
- **padding** (*int or tuple*) – Zero-padding added to both sides of the input. Default: 0.
- **groups** (*int*) – Number of blocked connections from input. channels to output channels. Default: 1.
- **bias** (*bool*) – If True, adds a learnable bias to the output. Default: False.
- **dilation** (*int or tuple*) – Spacing between kernel elements. Default: 1.

mmrotate.models.utils.**ennInterpolate**(*inplanes*, *scale_factor*, *mode='nearest'*, *align_corners=False*)

enn Interpolate.

mmrotate.models.utils.**ennMaxPool**(*inplanes*, *kernel_size*, *stride=1*, *padding=0*)

enn Max Pooling.

mmrotate.models.utils.**ennReLU**(*inplanes*)

enn ReLU.

mmrotate.models.utils.**ennTrivialConv**(*inplanes*, *outplanes*, *kernel_size=3*, *stride=1*, *padding=0*, *groups=1*, *bias=False*, *dilation=1*)

enn convolution with trivial input featurn.

Parameters

- **in_channels** (*List[int]*) – Number of input channels per scale.
- **out_channels** (*int*) – Number of output channels (used at each scale).
- **kernel_size** (*int, optional*) – The size of kernel.
- **stride** (*int, optional*) – Stride of the convolution. Default: 1.
- **padding** (*int or tuple*) – Zero-padding added to both sides of the input. Default: 0.
- **groups** (*int*) – Number of blocked connections from input. channels to output channels. Default: 1.
- **bias** (*bool*) – If True, adds a learnable bias to the output. Default: False.
- **dilation** (*int or tuple*) – Spacing between kernel elements. Default: 1.

CHAPTER

TWENTYONE

MMROTATE.UTILS

mmrotate.utils.**collect_env**()
Collect environment information.

mmrotate.utils.**find_latest_checkpoint**(*path*, *suffix='pth'*)
Find the latest checkpoint from the working directory.

Parameters

- **path** (*str*) – The path to find checkpoints.
- **suffix** (*str*) – File extension. Defaults to pth.

Returns File path of the latest checkpoint.

Return type latest_path(str | None)

References

mmrotate.utils.**get_root_logger**(*log_file=None*, *log_level=20*)
Get root logger.

Parameters

- **log_file** (*str, optional*) – File path of log. Defaults to None.
- **log_level** (*int, optional*) – The level of logger. Defaults to logging.INFO.

Returns The obtained logger

Return type logging.Logger

CHAPTER

TWENTYTWO

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

mmrotate.apis, 63
mmrotate.core.anchor, 65
mmrotate.core.bbox, 66
mmrotate.core.evaluation, 80
mmrotate.core.patch, 80
mmrotate.core.post_processing, 81
mmrotate.datasets, 83
mmrotate.datasets.pipelines, 85
mmrotate.models.backbones, 91
mmrotate.models.dense_heads, 93
mmrotate.models.detectors, 87
mmrotate.models.losses, 130
mmrotate.models.necks, 92
mmrotate.models.roi_heads, 120
mmrotate.models.utils, 134
mmrotate.utils, 137

INDEX

A

apply_coords() (*mmrotate.datasets.pipelines.PolyRandomRotate method*), 85
apply_image() (*mmrotate.datasets.pipelines.PolyRandomRotate method*), 85
AspectRatio() (*mmrotate.core.bbox.ATSSKldAssigner method*), 66
assign() (*mmrotate.core.bbox.ATSSKldAssigner method*), 66
assign() (*mmrotate.core.bbox.ConvexAssigner method*), 68
assign() (*mmrotate.core.bbox.MaxConvexIoUAssigner method*), 73
assign() (*mmrotate.core.bbox.SASAssigner method*), 76
assign_wrt_overlaps() (*mmrotate.core.bbox.MaxConvexIoUAssigner method*), 74
async_simple_test() (*mmrotate.models.detectors.RotatedTwoStageDetector method*), 90
async_simple_test() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 128
ATSSKldAssigner (*class in mmrotate.core.bbox*), 66
aug_multiclass_nms_rotated() (*in module mmrotate.core.post_processing*), 81
aug_test() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 103
aug_test() (*mmrotate.models.detectors.R3Det method*), 87
aug_test() (*mmrotate.models.detectors.RotatedSingleStageDetector method*), 88
aug_test() (*mmrotate.models.detectors.RotatedTwoStageDetector method*), 90
aug_test() (*mmrotate.models.detectors.S2ANet method*), 91
aug_test() (*mmrotate.models.roi_heads.RoITransRoIHead method*), 121
aug_test() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 128

B

bbox_flip() (*mmrotate.datasets.pipelines.RRandomFlip method*), 86
bbox_mapping_back() (*in module mmrotate.core.bbox*), 77
BCConvexGIoULoss (*class in mmrotate.models.losses*), 130
build_assigner() (*in module mmrotate.core.bbox*), 77
build_bbox_coder() (*in module mmrotate.core.bbox*), 77
build_enn_divide_feature() (*in module mmrotate.models.utils*), 134
build_enn_feature() (*in module mmrotate.models.utils*), 134
build_enn_norm_layer() (*in module mmrotate.models.utils*), 134
build_enn_trivial_feature() (*in module mmrotate.models.utils*), 134
build_roi_layers() (*mmrotate.models.roi_heads.RotatedSingleRoIExtractor method*), 126
build_sampler() (*in module mmrotate.core.bbox*), 77

C

check_size() (*mmrotate.core.bbox.GaussianMixture method*), 72
collect_env() (*in module mmrotate.utils*), 137
convex_overlaps() (*mmrotate.core.bbox.MaxConvexIoUAssigner method*), 74
ConvexAssigner (*class in mmrotate.core.bbox*), 67
ConvexGIoULoss (*class in mmrotate.models.losses*), 130
create_rotation_matrix() (*mmrotate.datasets.pipelines.PolyRandomRotate method*), 85
CSLRRetinaHead (*class in mmrotate.models.dense_heads*), 93
custom_accuracy (*mmrotate.models.roi_heads.RotatedBBoxHead property*), 123
custom_activation (*mmrotate.models.roi_heads.RotatedBBoxHead*

property), 123
custom_cls_channels (*mmrotate.models.roi_heads.RotatedBBoxHead property*), 123

D

decode() (*mmrotate.core.bbox.DeltaXYWHAHBBoxCoder method*), 69
decode() (*mmrotate.core.bbox.DeltaXYWHAOBBoxCoder method*), 70
decode() (*mmrotate.core.bbox.GVFixCoder method*), 70
decode() (*mmrotate.core.bbox.GVRatioCoder method*), 71
decode() (*mmrotate.core.bbox.MidpointOffsetCoder method*), 74
DeltaXYWHAHBBoxCoder (*class in mmrotate.core.bbox*), 68
DeltaXYWHAOBBoxCoder (*class in mmrotate.core.bbox*), 69
DOTADataset (*class in mmrotate.datasets*), 83

E

em_runner() (*mmrotate.core.bbox.GaussianMixture method*), 72
EM_step() (*mmrotate.core.bbox.GaussianMixture method*), 71
encode() (*mmrotate.core.bbox.DeltaXYWHAHBBoxCoder method*), 69
encode() (*mmrotate.core.bbox.DeltaXYWHAOBBoxCoder method*), 70
encode() (*mmrotate.core.bbox.GVFixCoder method*), 71
encode() (*mmrotate.core.bbox.GVRatioCoder method*), 71
encode() (*mmrotate.core.bbox.MidpointOffsetCoder method*), 75
ennAvgPool() (*in module mmrotate.models.utils*), 134
ennConv() (*in module mmrotate.models.utils*), 135
ennInterpolate() (*in module mmrotate.models.utils*), 135
ennMaxPool() (*in module mmrotate.models.utils*), 135
ennReLU() (*in module mmrotate.models.utils*), 135
ennTrivialConv() (*in module mmrotate.models.utils*), 135
estimate_log_prob() (*mmrotate.core.bbox.GaussianMixture method*), 72
eval_rbbox_map() (*in module mmrotate.core.evaluation*), 80
evaluate() (*mmrotate.datasets.DOTADataset method*), 83
evaluate() (*mmrotate.datasets.HRSCDataset method*), 84
extract_feat() (*mmrotate.models.detectors.R3Det method*), 87
extract_feat() (*mmrotate.models.detectors.RotatedSingleStageDetector method*), 89
extract_feat() (*mmrotate.models.detectors.RotatedTwoStageDetector method*), 90
extract_feat() (*mmrotate.models.detectors.S2ANet method*), 91

F

filter_bboxes() (*mmrotate.models.dense_heads.RotatedRetinaHead method*), 114
filter_border() (*mmrotate.datasets.pipelines.PolyRandomRotate method*), 85
find_latest_checkpoint() (*in module mmrotate.utils*), 137
fit() (*mmrotate.core.bbox.GaussianMixture method*), 72
format_results() (*mmrotate.datasets.DOTADataset method*), 83
forward() (*mmrotate.models.backbones.ReResNet method*), 92
forward() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 104
forward() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 111
forward() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 118
forward() (*mmrotate.models.losses.BCConvexGIoULoss method*), 130
forward() (*mmrotate.models.losses.ConvexGIoULoss method*), 130
forward() (*mmrotate.models.losses.GDLoss method*), 131
forward() (*mmrotate.models.losses.GDLoss_v1 method*), 131
forward() (*mmrotate.models.losses.KFLoss method*), 132
forward() (*mmrotate.models.losses.KLDRepPointsLoss method*), 132
forward() (*mmrotate.models.losses.SmoothFocalLoss method*), 133
forward() (*mmrotate.models.necks.ReFPN method*), 93
forward() (*mmrotate.models.roi_heads.RotatedBBoxHead method*), 123
forward() (*mmrotate.models.roi_heads.RotatedConvFCBBoxHead method*), 126
forward() (*mmrotate.models.roi_heads.RotatedSingleRoIExtractor method*), 127
forward() (*mmrotate.models.utils.ORConv2d method*), 134

forward() (*mmrotate.models.utils.RotationInvariantPooling method*), 134
forward_dummy() (*mmrotate.models.detectors.R3Det method*), 87
forward_dummy() (*mmrotate.models.detectors.RotatedSingleStageDetector method*), 89
forward_dummy() (*mmrotate.models.detectors.RotatedTwoStageDetector method*), 90
forward_dummy() (*mmrotate.models.detectors.S2ANet method*), 91
forward_dummy() (*mmrotate.models.roi_heads.GVRatioRoIHead method*), 120
forward_dummy() (*mmrotate.models.roi_heads.RoITransRoIHead method*), 121
forward_dummy() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 128
forward_single() (*mmrotate.models.dense_heads.CSLRRetinaHead method*), 93
forward_single() (*mmrotate.models.dense_heads.KFIoUODMRefineHead method*), 96
forward_single() (*mmrotate.models.dense_heads.ODMRefineHead method*), 101
forward_single() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 104
forward_single() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 111
forward_single() (*mmrotate.models.dense_heads.RotatedRetinaHead method*), 115
forward_single() (*mmrotate.models.dense_heads.RotatedRPNHead method*), 107
forward_single() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 118
forward_train() (*mmrotate.models.detectors.R3Det method*), 87
forward_train() (*mmrotate.models.detectors.RotatedSingleStageDetector method*), 89
forward_train() (*mmrotate.models.detectors.RotatedTwoStageDetector method*), 90
forward_train() (*mmrotate.models.detectors.S2ANet method*), 91
forward_train() (*mmrotate.models.roi_heads.OrientedStandardRoIHead method*), 120
forward_train() (*mmrotate.models.roi_heads.RoITransRoIHead method*), 122
forward_train() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 128

G

gaussian2bbox() (*in module mmrotate.core.bbox*), 77
GaussianMixture (*class in mmrotate.core.bbox*), 71
GDLoss (*class in mmrotate.models.losses*), 131
GDLoss_v1 (*class in mmrotate.models.losses*), 131
get_anchors() (*mmrotate.models.dense_heads.KFIoUODMRefineHead method*), 97
get_anchors() (*mmrotate.models.dense_heads.KFIoURRetinaRefineHead method*), 99
get_anchors() (*mmrotate.models.dense_heads.ODMRefineHead method*), 101
get_anchors() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 104
get_anchors() (*mmrotate.models.dense_heads.RotatedRetinaRefineHead method*), 116
get_bboxes() (*mmrotate.models.dense_heads.CSLRRetinaHead method*), 94
get_bboxes() (*mmrotate.models.dense_heads.KFIoUODMRefineHead method*), 97
get_bboxes() (*mmrotate.models.dense_heads.KFIoURRetinaRefineHead method*), 99
get_bboxes() (*mmrotate.models.dense_heads.ODMRefineHead method*), 101
get_bboxes() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 104
get_bboxes() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 111
get_bboxes() (*mmrotate.models.dense_heads.RotatedRetinaRefineHead method*), 116
get_bboxes() (*mmrotate.models.dense_heads.RotatedRPNHead*

method), 108
get_bboxes() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 118
get_bboxes() (*mmrotate.models.roi_heads.RotatedBBoxHead method*), 123
get_cfa_targets() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 111
get_horizontal_bboxes() (*mmrotate.core.bbox.ATSSKldAssigner method*), 67
get_horizontal_bboxes() (*mmrotate.core.bbox.ConvexAssigner method*), 68
get_indices() (*mmrotate.models.utils.ORConv2d method*), 134
get_multiscale_patch() (*in module mmrotate.core.patch*), 80
get_points() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 112
get_points() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 118
get_pos_loss() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 112
get_root_logger() (*in module mmrotate.utils*), 137
get_score() (*mmrotate.core.bbox.GaussianMixture method*), 72
get_targets() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 105
get_targets() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 113
get_targets() (*mmrotate.models.dense_heads.RotatedRPNHead method*), 108
get_targets() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 119
get_targets() (*mmrotate.models.roi_heads.RotatedBBoxHead method*), 124
GlidingVertex (*class in mmrotate.models.detectors*), 87
gt2gaussian() (*in module mmrotate.core.bbox*), 77
GVFixCoder (*class in mmrotate.core.bbox*), 70
GVRatioCoder (*class in mmrotate.core.bbox*), 71
GVRatioRoIHead (*class in mmrotate.models.roi_heads*), 120

H

hbb2obb() (*in module mmrotate.core.bbox*), 77
HRSCDataset (*class in mmrotate.datasets*), 84

I

inference_detector_by_patches() (*in module mmrotate.apis*), 63
init_assigner_sampler() (*mmrotate.models.roi_heads.RoITransRoIHead method*), 122
init_assigner_sampler() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 129
init_bbox_head() (*mmrotate.models.roi_heads.RoITransRoIHead method*), 122
init_bbox_head() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 129
is_rotate (*mmrotate.datasets.pipelines.PolyRandomRotate property*), 85

K

KFIoUODMRefineHead (*class in mmrotate.models.dense_heads*), 96
KFIoURRetinaHead (*class in mmrotate.models.dense_heads*), 97
KFIoURRetinaRefineHead (*class in mmrotate.models.dense_heads*), 98
KFLoss (*class in mmrotate.models.losses*), 132
kld_mixture2single() (*mmrotate.core.bbox.ATSSKldAssigner method*), 67
kld_overlaps() (*mmrotate.core.bbox.ATSSKldAssigner method*), 67
KLDRepPointsLoss (*class in mmrotate.models.losses*), 132

L

load_annotations() (*mmrotate.datasets.DOTADataset method*), 84
load_annotations() (*mmrotate.datasets.HRSCDataset method*), 84
LoadPatchFromImage (*class in mmrotate.datasets.pipelines*), 85
log_resp_step() (*mmrotate.core.bbox.GaussianMixture method*), 72
loss() (*mmrotate.models.dense_heads.CSLRRetinaHead method*), 95
loss() (*mmrotate.models.dense_heads.KFIoUODMRefineHead method*), 97

loss() (*mmrotate.models.dense_heads.KFIoURRetinaRefineHead method*), 100
loss() (*mmrotate.models.dense_heads.ODMRefineHead method*), 102
loss() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 106
loss() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 113
loss() (*mmrotate.models.dense_heads.RotatedRetinaRefineHead method*), 116
loss() (*mmrotate.models.dense_heads.RotatedRPNHead method*), 109
loss() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 119
loss() (*mmrotate.models.roi_heads.RotatedBBoxHead method*), 125
loss_single() (*mmrotate.models.dense_heads.CSLRRetinaHead method*), 95
loss_single() (*mmrotate.models.dense_heads.KFIoURRetinaHead method*), 98
loss_single() (*mmrotate.models.dense_heads.OrientedRPNHead method*), 102
loss_single() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 107
loss_single() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 113
loss_single() (*mmrotate.models.dense_heads.RotatedRPNHead method*), 109
loss_single() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 119

M

make_res_layer() (*mmrotate.models.backbones.ReResNet method*), 92
map_roi_levels() (*mmrotate.models.roi_heads.RotatedSingleRoIExtractor method*), 127
MaxConvexIoUAssigner (*class in mmrotate.core.bbox*), 73
merge_aug_bboxes() (*mmrotate.models.dense_heads.RotatedAnchorHead method*), 107
merge_det() (*mmrotate.datasets.DOTADataset method*), 84
merge_results() (*in module mmrotate.core.patch*), 80
MidpointOffsetCoder (*class in mmrotate.core.bbox*), 74
mmrotate.apis
    module, 63
mmrotate.core.anchor
    module, 65
mmrotate.core.bbox
    module, 66
mmrotate.core.evaluation
    module, 80
mmrotate.core.patch
    module, 80
mmrotate.core.post_processing
    module, 81
mmrotate.datasets
    module, 83
mmrotate.datasets.pipelines
    module, 85
mmrotate.models.backbones
    module, 91
mmrotate.models.dense_heads
    module, 93
mmrotate.models.detectors
    module, 87
mmrotate.models.losses
    module, 130
mmrotate.models.necks
    module, 92
mmrotate.models.roi_heads
    module, 120
mmrotate.models.utils
    module, 134
mmrotate.utils
    module, 137
module
    mmrotate.apis, 63
    mmrotate.core.anchor, 65
    mmrotate.core.bbox, 66
    mmrotate.core.evaluation, 80
    mmrotate.core.patch, 80
    mmrotate.core.post_processing, 81
    mmrotate.datasets, 83
    mmrotate.datasets.pipelines, 85
    mmrotate.models.backbones, 91
    mmrotate.models.dense_heads, 93
    mmrotate.models.detectors, 87
    mmrotate.models.losses, 130
    mmrotate.models.necks, 92
    mmrotate.models.roi_heads, 120
    mmrotate.models.utils, 134
    mmrotate.utils, 137
multiclass_nms_rotated() (*in module mmrotate.core.post_processing*), 81

N

norm1 (*mmrotate.models.backbones.ReResNet property*), 92
norm_angle() (*in module mmrotate.core.bbox*), 77
num_base_anchors (*mmrotate.core.anchor.PseudoAnchorGenerator property*), 65

O

obb2hbb() (*in module mmrotate.core.bbox*), 78
obb2poly() (*in module mmrotate.core.bbox*), 78
obb2poly_np() (*in module mmrotate.core.bbox*), 78
obb2xyxy() (*in module mmrotate.core.bbox*), 78
ODMRefineHead (*class in mmrotate.models.dense_heads*), 100
offset_to_pts() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 113
offset_to_pts() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 119
ORConv2d (*class in mmrotate.models.utils*), 134
OrientedRCNN (*class in mmrotate.models.detectors*), 87
OrientedRPNHead (*class in mmrotate.models.dense_heads*), 102
OrientedStandardRoIHead (*class in mmrotate.models.roi_heads*), 120

P

points2rotrect() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 113
points2rotrect() (*mmrotate.models.dense_heads.SAMRepPointsHead method*), 119
poly2obb() (*in module mmrotate.core.bbox*), 78
poly2obb_np() (*in module mmrotate.core.bbox*), 79
PolyRandomRotate (*class in mmrotate.datasets.pipelines*), 85
PseudoAnchorGenerator (*class in mmrotate.core.anchor*), 65

R

R3Det (*class in mmrotate.models.detectors*), 87
random_choice() (*mmrotate.core.bbox.RRandomSampler method*), 75
rbbox2result() (*in module mmrotate.core.bbox*), 79
rbbox2roi() (*in module mmrotate.core.bbox*), 79
rbbox_overlaps() (*in module mmrotate.core.bbox*), 79
RBboxOverlaps2D (*class in mmrotate.core.bbox*), 75
reassign() (*mmrotate.models.dense_heads.RotatedRepPointsHead method*), 113
ReDet (*class in mmrotate.models.detectors*), 87
refine_bboxes() (*mmrotate.models.dense_heads.KFIoURRetinaRefineHead method*), 100
refine_bboxes() (*mmrotate.models.dense_heads.RotatedRetinaHead method*), 115
refine_bboxes() (*mmrotate.models.dense_heads.RotatedRetinaRefineHead method*), 117
refine_bboxes() (*mmrotate.models.roi_heads.RotatedBBoxHead method*), 125
ReFPN (*class in mmrotate.models.necks*), 92
regress_by_class() (*mmrotate.models.roi_heads.RotatedBBoxHead method*), 125
ReResNet (*class in mmrotate.models.backbones*), 91
reset_parameters() (*mmrotate.models.utils.ORConv2d method*), 134
roi_rescale() (*mmrotate.models.roi_heads.RotatedSingleRoIExtractor method*), 127
RoITransformer (*class in mmrotate.models.detectors*), 88
RoITransRoIHead (*class in mmrotate.models.roi_heads*), 121
rotate_arf() (*mmrotate.models.utils.ORConv2d method*), 134
rotated_anchor_inside_flags() (*in module mmrotate.core.anchor*), 65
RotatedAnchorGenerator (*class in mmrotate.core.anchor*), 65
RotatedAnchorHead (*class in mmrotate.models.dense_heads*), 102
RotatedBaseDetector (*class in mmrotate.models.detectors*), 88
RotatedBBoxHead (*class in mmrotate.models.roi_heads*), 123
RotatedConvFCBBoxHead (*class in mmrotate.models.roi_heads*), 126
RotatedFasterRCNN (*class in mmrotate.models.detectors*), 88
RotatedRepPoints (*class in mmrotate.models.detectors*), 88
RotatedRepPointsHead (*class in mmrotate.models.dense_heads*), 110
RotatedRetinaHead (*class in mmrotate.models.dense_heads*), 114
RotatedRetinaNet (*class in mmrotate.models.detectors*), 88
RotatedRetinaRefineHead (*class in mmrotate.models.dense_heads*), 115
RotatedRPNHead (*class in mmrotate.models.dense_heads*), 102

tate.models.dense_heads), 107
RotatedShared2FCBBoxHead (*class in mmrotate.models.roi_heads*), 126
RotatedSingleRoIExtractor (*class in mmrotate.models.roi_heads*), 126
RotatedSingleStageDetector (*class in mmrotate.models.detectors*), 88
RotatedStandardRoIHead (*class in mmrotate.models.roi_heads*), 127
RotatedTwoStageDetector (*class in mmrotate.models.detectors*), 89
RotationInvariantPooling (*class in mmrotate.models.utils*), 134
RRandomFlip (*class in mmrotate.datasets.pipelines*), 85
RRandomSampler (*class in mmrotate.core.bbox*), 75
RResize (*class in mmrotate.datasets.pipelines*), 86

S

S2ANet (*class in mmrotate.models.detectors*), 91
sample() (*mmrotate.core.bbox.RRandomSampler method*), 75
SAMRepPointsHead (*class in mmrotate.models.dense_heads*), 117
SARDataset (*class in mmrotate.datasets*), 85
SASAssigner (*class in mmrotate.core.bbox*), 76
show_result() (*mmrotate.models.detectors.RotatedBaseDetector method*), 88
simple_test() (*mmrotate.models.detectors.R3Det method*), 87
simple_test() (*mmrotate.models.detectors.RotatedSingleStageDetector method*), 89
simple_test() (*mmrotate.models.detectors.RotatedTwoStageDetector method*), 90
simple_test() (*mmrotate.models.detectors.S2ANet method*), 91
simple_test() (*mmrotate.models.roi_heads.RoITransRoIHead method*), 122
simple_test() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 129
simple_test_bboxes() (*mmrotate.models.roi_heads.GVRatioRoIHead method*), 120
simple_test_bboxes() (*mmrotate.models.roi_heads.OrientedStandardRoIHead method*), 121
simple_test_bboxes() (*mmrotate.models.roi_heads.RotatedStandardRoIHead method*), 129
single_level_grid_anchors() (*mmrotate.core.anchor.PseudoAnchorGenerator method*), 65
single_level_grid_priors() (*mmrotate.core.anchor.RotatedAnchorGenerator method*), 65
slide_window() (*in module mmrotate.core.patch*), 80
SmoothFocalLoss (*class in mmrotate.models.losses*), 133

T

train() (*mmrotate.models.backbones.ReResNet method*), 92
train_detector() (*in module mmrotate.apis*), 63

U

update_mu() (*mmrotate.core.bbox.GaussianMixture method*), 72
update_pi() (*mmrotate.core.bbox.GaussianMixture method*), 73
update_var() (*mmrotate.core.bbox.GaussianMixture method*), 73

W

with_roi_head (*mmrotate.models.detectors.RotatedTwoStageDetector property*), 90
with_rpn (*mmrotate.models.detectors.RotatedTwoStageDetector property*), 90